GoLite Final Report

Pavel Kondratyev	Kaylee Kutschera	Kabilan Sriranjan
`pashakondratyev`	`kkutschera`	`kabsri`
260653115	260608470	260671847

Contents

1 Go and GoLite

Go is a popular language initially released in 2009 that had very quick adoption as it is simpler than most popular languages and is thus easy to learn. Go encompasses all the classical compiler phases which makes it a great candidate to build a compiler for. Additionally, the specification provided for Go is detailed and easy to follow: `https://golang.org/ref/spec`. Since we only had two months to build our compiler, we built it for a subset of Go called GoLite. GoLite covers many of the features of Go, including its imperative programming style, slices, and optional semicolons.

This final report will cover our choice of tools and languages, the design of the core components of our compiler, our experience, and finally each of our contributions to the overall project.

2 Language and Tool Choices

We chose to implement our compiler in C using flex and bison. The main motivation behind this was that we were already familiar with using these tools from the assignments as well as the lecture content. This let us focus more on creating a successful compiler rather than learning new tools.

For our target language we decided to work with Java. We all had good familiarity with Java, and we felt that would help us create a better project than if we chose a target only one of us was familiar with, or worse yet, none of us. Java, as an object oriented language, is very nice to work with when multiple objects might have the same type of operation performed on them. For example we can perform comparisons on Strings, Integers, Structs, etc. by calling .equals, instead of having to handle each case individually.

While Java's object oriented approach lends itself to some great reusability, it is unable to natively support certain go constructs, such as redeclaring variables or initialization in if/switch statements. Both Go and Java are pass-by-value, but when an object in Java is being passed the value is a reference to the object but in Go the value is the actual object. Hence we will have to remember to create copies objects as necessary.

3 Compiler Design

3.1 Scanner

For most of the features of GoLite the implementation of the scanner was very similar to the MiniLang compiler from previous assignments, with the following exceptions:

3.1.1 Semicolons

Semicolons were inserted based on the rules given in the specifications-namely keeping tracking of the final token per line. If there was an EOF character we also had to add a semicolon.

3.1.2 Multiline Comments

The multi-line comments were handled by using start statements as described in the flex documentation for C-style commenting. Semicolons were also inserted in this state. While we were in this state, we also had to check if we reached an EOF character and insert a semicolon accordingly.

3.1.3 Strings, Raw Strings, and Runes

For valid plain strings we scan the entire scanned token and use a function to check if there is an escaped character as per the specifications. If there is, the full escaped character is inserted as a character instead of the '\' followed by whatever character is. Raw strings are left exactly as scanned. Runes are handled the same way as each character of the plain string, with the difference being the valid escaped characters.

3.2 Parser

The grammar was designed almost entirely off of the official GoLang specifications.

3.2.1 EBNF to CFG

There were mild difficulties for constructs that were specified using shorthand in the official documentation that couldn't be directly translated in bison. For

example, the official specification defines a Block in the following manner:

$$\begin{aligned} Block &\rightarrow \text{“\{”}\ StatementList\ \text{“\}”} \\ StatementList &\rightarrow \{\ Statement\ \text{“;”}\ \} \end{aligned}$$

Since this notation doesn't apply to bison we implemented the rule like so:

$$\begin{aligned} Block &\rightarrow \text{“\{”}\ StatementList\ \text{“\}”} \\ StatementList &\rightarrow Statement\ \text{“;”} \\ &\quad | Statement\ \text{“;”}\ StatementList \end{aligned}$$

Another situation that required a different type of workaround were for clauses. Officially they were specified as:

$$\begin{aligned} ForClause &\rightarrow [\ InitStmt\ ]\ \text{“;”}\ [\ Condition\ ]\ \text{“;”}\ [\ PostStmt\ ] \\ InitStmt &\rightarrow SimpleStmt \\ PostStmt &\rightarrow SimpleStmt \end{aligned}$$

On top of this notation not being directly transferable to bison, the constructs InitStmt and PostStmt add unnecessary bulk to the grammar. We instead implemented the for clause as:

$$\begin{aligned} ForClause &\rightarrow SimpleStatement\ \text{“;”}\ ExpressionOrEmpty\ \text{“;”} SimpleStatement \\ ExpressionOrEmpty &\rightarrow \%empty \\ &\quad | Expression \end{aligned}$$

Taking note that a SimpleStatement can already be empty, this was a concise way to represent the official spec in bison.

3.2.2 Short Variable Declarations

One place where we had difficulty directly translating the requirements was when we wanted to implement short variable declarations. The official spec pegs it as:

$$ShortVarDecl \rightarrow IdentifierList\ \text{“:=”}\ ExpressionList$$

This caused shift reduce errors as the left hand side could have been either an expression list or a identifier list. So the rule was instead implemented as

follows:

$$ShortVarDecl \rightarrow ExpressionList \text{ ``}:=\text{'' } ExpressionList$$

And during the weeding phase we will make sure that each expression on the left hand side is an identifier.

3.2.3 If/Else-If

The rule in the spec for if statements was as follows:

$$IfStmt \rightarrow \text{``}if\text{''} \ [SimpleStmt\text{``};\text{''}] \ Expression \ Block \ [\text{``}else\text{''}(IfStmt|Block)]$$

However this led to shift reduce errors if implemented directly so we instead broke it up into two rules:

$$\begin{aligned} IfStmt \rightarrow & \text{``}if\text{''} \ Expression \ Block \ ElseStatement \\ & |\text{``}if\text{''} \ SimpleStatement\text{``};\text{''} \ Expression \ Block \ ElseStatement \end{aligned}$$

$$\begin{aligned} ElseStatement \rightarrow & \%empty \\ & |\text{``}else\text{''} \ IfStmt \\ & |\text{``}else\text{''} \ Block \end{aligned}$$

For rules that weren't mentioned, all the tricks used to contruct the grammar in bison were introducing new rules in place of the [] and {} operators and removing extraneous constructs.

3.3 Abstract Syntax Tree

The abstract syntax tree largely followed the structure of the grammar. We have 5 primary nodes, which are EXP for expressions, DECL for declarations, STMT for statements, TYPE for types, and PROGRAM for storing the program. There are many secondary nodes which aggregate the above types in order to help handle other constructs.

PACKAGE holds the name of the package, but for the GoLite compiler this is not used for anything, other than making the files compatible with a full Go compiler.

Constructs like statement lists (STMT_LIST), where there can be one or many components were handled by using linked lists. Namely, this technique

was used for block statements - which are a list of statements - and switch statements - which are a list of case clauses. Declarations also are implemented as linked lists, having a pointer to the next declaration.

The switch statement structure was slightly more complex to implement because on top of including all the relevant information per node such as clause kind and case expression, one of the attributes was itself a statement list. This led to the switch expression being stored as a somewhat 2D linked list in the AST. This caused some difficulties in later stages like weeding, but it seemed like the most elegant solution to capture the nature of switch expression. In GoLite it is possible to specify multiple variables with a single assignment operator.

Since in GoLite we can only return single values, within the AST we had to check that number of items on each side of the "=" token in a specification are equal. At some points there are multiple declarations or assignments, these are broken down into individual declarations or assignments all contained in a linked list within the AST. Thus, it is simple to determine what the declaration or assignment is, yet still maintain a record that it was a multiple declaration or assignment. However this was not necessary as the multiple declaration format is just syntactic sugar.

There were also nodes to help with each type of declaration to hold the structure in a more maintainable fashion than using large unions of structs, namely VAR_SPECS, SHORT_SPECS, TYPE_SPECS, and FUNC_DECL.

3.4 Weeder

There were several aspects that weren't handled in the grammar and were instead dealt with in the weeding phase. Bison was giving shift/reduce errors parsing the left hand side of short declarations as identifier lists and so to combat this we transformed the identifier list into a more general expression list. Then during weeding phase each expression in the list was checked.

The grammar accepts programs that have continue statements inside a switch statement or breaks and continues outside of loops. Since in GoLite these cases are not allowed we used a weeder to reject such programs. The way we implemented this is by weeding from the very beginning of the program and use recursive calls to traverse through the program tree. As soon as a for statement is parsed, all the recursive calls within the for block are passed with arguments signalling that break and continue statements are allowed. If a break or continue statement is parsed with these arguments absent, the weeder will reject the program.

Switch statements can only have at most one default case so we weeded switch statements that had many of them. The default case can go anywhere in

the list of clauses. Since all the clauses are stored in a linked list, and the kind of clause - whether it is a default or a case - is stored per clause all that needed to be done was count the number of nodes of the default kind present in the clause list.

All non void functions must return a value. Initially we made the weeder traverse all possible paths of the control flow and make sure all return statements have an expression. For the switch statements this meant we had to check that each case had a return statement, which was made easier by GoLite not handling fallthrough. In the reverse case, for void functions we weeded to check for the existence of a return statement with an expression. In the end we moved this functionality to the typechecker instead. Since return statements must have their expression typechecked anyway it was simpler to just check that the expression was null for void functions.

3.5 Symbol Table

The symbol table was implemented using a similar skeleton to what was covered in class. A symbol is a struct which stores the identifier, the declaration kind (function, variable, short, or type), as well as the relevant information depending on the kind of declaration. We also store the constants true anF false. dor a variable declaration we needed to store the type the variable resolves to. For a function declaration we stored the return type and parameter list. For a type declaration we stored the type and the type it resolves to. The symbol table itself is a hashmap using the following hash function, given in class.

```
while (*str) hash = (hash << 1) + *str++;
```

Each declaration and statement is read in order to open and close scopes. New scopes are only opened on blocks as well as switch cases. Special note should be brought to if/for statements as they may contain a declaration so a scope is opened around both the if and all the subsequent if-else/else statements.

3.5.1 Type Mapping

Due to the way the parser was constructed, every time a type was read in a new type node was created in the abstract syntax tree. This makes sense as during the parser stage we do not have the symbol table. But when we read

```
type int string
var test string = 4
```

we are not able to lookup if string is a reference type, a base type, or a composite type (such as arrays, slices, or structs). We originally tried to mark everything as a reference type and then resolve it but then we couldn't distinguish real reference types from other types without having a convoluted type checker. The

streamlined solution was to mark all types in the parser using an enum so that they could be linked in the abstract syntax tree using the symbol table. This way types are pointing to the same type nodes from the symbol table and can be compared directly by reference, resulting in simpler code and easier debugging. This also saves space as each type has only one node in the AST that can be referenced many times. The implementation for this was to recursively descend through the type (such as in the case of compound types) and resolve each of these fields to the proper type and return the properly mapped one in the end. Note that even though we are using the word resolve we are not resolving the type to the base type or resolving a type to its reference.

When we put types into the symbol table, we also keep track of the type it resolves to in a pointer. This will make it easier later to check if a type resolves to a base type without having to do a traversal of the symbol table each time.

3.5.2 Scopes

In the symbol table scopes are added as per the rules given to us. Namely in block statements such as those found in for loops, if/else statements, switch cases, as well as standalone block statements. Additionally, scopes must be opened up for initialization statements that are found in such statements which are separate, so that if a variable is declared it can also be shadowed in the body of the statements. The scopes are attached to the node of the body in the abstract syntax tree, a new field was added in the relevant nodes to hold scope.

3.5.3 Short Variable Declarations

For the syntax of short variable declarations to be correct, the left hand side must contain at least one new variable in the scope. In order to implement this, each declaration in the left hand side is checked to be in the current scope in the symbol table. If the current scope (not including any of the parent scopes) contains the symbol an error is returned. Otherwise all the new symbols are added to the scope.
Default Entries Before we begin traversing and building the symbol table, we need to establish the base types, so we add all the 5 base types (int, string, float64, rune, bool) as well as the constants for true and false. Everything else follows from those 5 types and two constants. After these are all added a new scope is added, so that they can be shadowed, and we can proceed with the rest of the symbol table.

3.5.4 Unmapped Identifiers

In Go and GoLite, the identifier "_" does not get mapped into the symbol table. The init function also does not get mapped and there can be multiple init

functions in the top level declarations. For type, variable, and short declarations this just means that if the identifier is "_", that it should not be put into the symbol table. However for functions, the inner scopes must still all be put into a symbol table and type checked later on so a scope is created for the body of the function. We still must keep track of blank parameters, as we must be able to pass expressi ns that will be evaluated during runtime, such as function calls.o

3.6 Typechecker

The typechecking rules were all directly implemented from the specification. There were a few challenges along the way and the solutions are outlined below.

3.6.1 Inferred Types

During the symbol table phase, certain declarations do not have types which are known immediately such as the type of test below:

```
var test = 4
```

We would need to check this declaration during the typechecker phase and fill in the blank in the symbol table, so that we know for later usages of test.

3.6.2 Type Comparisons

For assignment statements and comparison statements it was important to know if the two types are the same (without resolutions). Because of the way that we mapped the types, comparing two non-composite types was easy as they point to the same space in memory. When we are dealing with arrays and slices we check the inner types as well. For untagged structs we check that all the fields appear in the same order, have the same names, and have the same types (in other words, are identical). These are all requirements based on the Go specification.

3.6.3 Assignments

One step we had to take when typechecking assignment statements in addition to checking that the types on each side matched was to also make sure the the left hand side of every assignment was an Lvalue. This applied to variable declarations as well as op-assignments. Since these can be nested within arbitrary amounts of parentheses, index expressions, and struct fields we had to implement a recursive method to verify whether an identifier was an Lvalue or not.

3.6.4 Type Resolution

To resolve types, as mentioned in the symbol table section, for each type definition we keep track of the type it resolves to. This makes checking if two types resolve to the same type very easy, as we can just retrieve the symbols and directly check if the types are equal using the method above. In all situations we only needed to check if two types resolved to the same underlying type so there was no need to implement traversals through the symbol table to compare types.

3.6.5 Return Statements

When typechecking return statements there are a few criteria to consider.

1. If the function has a void return type, there must be no return statements which try to return a value
2. If the function has a non void return type it must never terminate without returning a value, i.e all branches of the function return or there is an infinite loop
3. If a function has a non void return type all the return statements must return the correct type

We were first unsure of whether we should write a separate traversal to check these conditions(as 1 and 2 seem like something we could have handled in the weeding stage) or attempt to add it onto our function which typechecks a statement. The solution we came up with was to return whether a statement is guaranteed to return in the function for typechecking statements. In the case of infinite for loops we check if there is no break statement. For statement lists, such as blocks, we only need to check that one of the statements is guaranteed to return. Technically this should be the last statement. This way we can recursively build up whether all possible paths return. Here we have some special cases which we address:

- If Statements - For if statements, the if statement and all the else/if statements must be guaranteed to return. The optional final else statement must also be present and guaranteed to return.
- Switch Cases - For switch cases, in order to consider the block as "returning" each case must return, since we do not have fallthrough statements and break statements are implicit. The optional default clause must also be present and guaranteed to return.
- For Loops - For infinite for loops which do not break, we can consider as a valid typechecking statement regardless of what the return type of the function is. This makes sense as if a function never terminates then it can never return an incorrect type. Hence we say a for loop returns either

if it has no condition and contains no break statements or if all possible branches inside the for loop return. In all other cases it is not guaranteed to return and thus does not typecheck.

3.7 Code Generator

3.7.1 Semantics & Mapping

3.7.1.1 Identifiers

3.7.1.1 Blank Identifiers

In Go, blank identifiers are not mapped to the symbol table and they are not type-checked, so there can be unlimited blank identifiers, being assigned any type.

In Java, the idea of a blank identifier does not exist. The strategy for handling this is Whenever a blank identifier is encountered, a properly typed dummy variable prefixed with "blank" is created. This value is never used again.

3.7.1.1 Keywords

All identifiers and struct fields are prefixed with __golite__ to denote that they're GoLite elements so they don't interfere with Java keywords. All helper methods and helper classes are added to the beginning of the class before we being generating the main public class.

3.7.1.2 Scoping Rules

In GoLite, scopes are opened for functions declarations, if statements, for loops, switch statements, and standalone blocks. For all the situations except for function declarations, there is an optional init statement that has its own scope between the original scope and the block of code inside. The parameters of a function belong to the same scope as its body. Variables can be redeclared when inside a new scope to have any available type.

In Java, new scopes are added for all the same situations as in GoLite, plus a few other instances like while loops and subclasses. However there are no init statements for if or switch statements, and the init statement in a Java for loop is not as flexible as it is in GoLite. Variables in a higher scope cannot redeclared in a lower one.

Mapping the scoping rules from GoLite to Java requires special care for init statements and the redeclaration of variables. Whenever there is an if, for, or

switch statement, before opening the scope of the statement we open a scope for the init statement. The translation of the init statement may be multiple lines long, and once it is done then the actual statement is mapped. The innermost scope is captured automatically due to the similarities between the scoping rules of each language. To handle the redeclaration of variables in inner scopes, we just add a suffix to each identifier with its scope level. This ensures a declaration in an inner scope can never have the same name as one from the outer scope.

3.7.1.3 Variable Declarations

In GoLite there are a few different ways to do long variable declarations but they all basically fall under three categories. Either both the type and initial value are specified and the types have to match, only the type of the variable is specified and it is initialized at a default value for that type, or the initial value is specified and the type is inferred based on that value. The default values for each base type are:

$$
\begin{aligned}
int &\leftarrow 0 \\
rune &\leftarrow 0 \\
float64 &\leftarrow 0.0 \\
bool &\leftarrow false \\
string &\leftarrow \text{“”}
\end{aligned}
$$

The default value of a struct is a struct where all the fields are their default values, the default value of an array is an array where each element is the default value of the array type, and the default value of a slice is the empty slice.

In Java a variable must be declared along with its type, and optionally initialized to some value. If there is no initialization a default value is assigned, just as in GoLite. For the GoLite base types the default values are the same as the corresponding types we chose to use in Java. However, reference types in Java default to *null* which is a problem. The constructor for the Array class also initializes each element to the default value, which could be *null*.

Structs and slices both come from custom built Java classes, so to ensure they were always initialized properly we made a constructor for each one and made sure that whenever a variable belonging to a custom class is declared we invoke the constructor. To solve the issue of the array constructor not initializing each element properly, we generate a for loop to call the constructor on each element whenever an array is declared. The function to do this had to be recursive because it is possible to have an array of arrays in which case we would need nested for loops. This solution works in many cases, but not when an array variable is declared at the top level. The top level in GoLite corresponds to the

inside of the Java class, where the class attributes and methods are declared. It is not possible to write for loops in this part of a Java program, so we generate a method that runs before any init function that takes all the top level arrays and executes initializing for loops for each of them.

3.7.1.4 Expressions

We implemented all the expressions in Java. Most expressions mapped directly because Java is a high level language. We created a utility class for typecasting which has methods to convert all base types to any other base type in which casting is allowed. For comparing expressions we used the compareTo() function because we opted to use reference type representations of literals, and this let us reuse code. For equality we used equals() for all objects except for arrays, where we used deepEquals() which calls equals() on each pair of elements.

3.7.1.5 Types

3.7.1.5 Base Types

In GoLite, the base types supported are a subset of the types supported by Go. In particular we support int, float64, bool, rune, and string (both interpreted and raw) whereas for Go there are integer and floating point numbers of different sizes such as uint8 and float32, as well as support for complex numbers. In GoLite runes behave identically to ints in all but one case- when an int is casted to a string each digit becomes the its own character but when a rune is casted to a string the character with the proper integer ASCII value is used.

Java supports all these types except strings as primitives but we had a much easier time using the wrapper classes (Integer, Double, and Boolean) for them instead. By using the wrapper classes we took advantage of the usage of .equals(Object o), such as when we handled array equality, which we address below. Strings in Java have all the comparison and copying functionality we need already so we just used the built-in implementation available to us. Initially we had also used the Character class to represent runes but then switched to using Integers. In GoLite a rune can have a negative value since it's essentially an int, but in Java it's impossible to have a Character with a negative numeric value.

3.7.1.5 Reference Types

In GoLite, users can declare any type as a new type and overwrite the existing types. While this can be handled in Java using Classes and Subclassing, by the code generation phase of the compiler we already had type-checked so we can

ignore reference types and just have everything resolve either to a base type, a struct, an array, or a slice.

3.7.1.5 Array Types

In Go, an array is a collection of homogeneous data of a certain, fixed size. The implementation in Go has 2 features which are important to consider: bounds checking and equality. Go gives a runtime error if you try to access memory outside of bounds. For equality, Go checks element-wise equality.

In Java arrays are a primitive construct and throw index out of bounds errors. For array equality, the array class already has a deepEquals method

```
public static boolean deepEquals(Object[] a1, Object[] a2)
```

which iterates through the elements and checks element-wise equality of the entire array. Th re is a caveat that thie calls .equals() on each elemenst

All the indices of the array are given the default value for the given type. Since we are using the Object wrappers in Java as well as specially defined Objects, we must also zero out all the values manually when they are declared, as otherwise they wi l be null,land we must perform deep copies during assignments.

3.7.1.5 Slice Types

In Go, a slice is a collection of homogeneous data which can be easily expanded. Unlike arrays, we do not need to declare a size, nor are we limited by a size. Slices have capacity and length, and as items get added the slice will increase its capacity to accommodate more. The default size is 0, and as we add elements if the new length exceeds the capacity we first increase the capacity to 2 elements, and then double the size as necessary. Length, capacity, and a reference to the underlying data of the slice are stored in the header information of the slice.

In Java, this behaviour is very similar to an ArrayList, however there are some quirks where the behavior is different, which required us to create a new class. In Java, the capacity of the ArrayList neither starts at 0 nor does it increase in multiples of 2. Based on the OpenJDK implementation, the capacity increases based on this formula

```
int newCapacity = oldCapacity + (oldCapacity >> 1);
```

which is $\frac{3}{2}$ of the old capacity. In order to accurately capture the implementation of the slice, we constructed a class Slice < T > which can be a slice of any type. Since arrays in Java cannot take generic types, our underlying data structure is an ArrayList, but we enforce that it grows according to the rules of

slices as we described above.

3.7.1.5 Struct Types

In GoLite, structs are a form of composite type that contain zero or more field declarations. These declarations can be of any previously declared types, including more struct types which leads to nesting. Fields within the same "scope" of a struct must have unique identifiers - unless they use the blank identifier. Struct types that reference themselves aren't allowed except in the case where the inner field is a slice of the original struct. Structs can also be tagged or untagged.

In Java, there are no such things as structs so we instead created a class for each new struct that we needed. Classes can have other objects included in their field declarations so we can also support the nesting structure. While in a GoLite program all the structs can be defined within different scopes, we define all the corresponding Java classes at the top level during code generation. It is always possible to create self referencing classes in Java so we didn't need to handle the situation with slices as a special case. When creating the classes we didn't worry about whether a struct was tagged or untagged, we just created a class for each struct that appeared and created a way to retrieve the class during compilation.

In GoLite, structs are comparable if each of their fields are comparable. In Java, unless overriden, the .equals() method compares references, so if all the fields are comparable we would also need to generate a new non-static .equals() method which compares all of the corresponding fields. Also, since in Java references can act as values, we needed to generate a clone method which properly clones each field, including other structs and slices. Another part of being comparable in GoLite is two structs can be defined in two places as long as all the fields match. In Java, while you can compare any field of two different objects, you would need a method for every pair of objects if you want to access their parameters.

Before generating any code, we traversed the whole program and gathered all usages of the word "struct", unrolled them recursively, and then stored them in a separate table if they did not already exist. This way if two structs are identical they will appear as objects from the same class and we can use the .equals() method. Also since we threw away all references, we did not have to worry about cases like

```
type s1 struct {a int;}
type int struct {c string;}
type s2 struct {a int;}
```

where the field in s1 is of the base int type, but the field in s2 is of the struct int

type. At the time of declaring s1, int is a base type but at the time of declaring s2, int is a struct.

3.7.1.6 Assign Statements

In Go, assignment statements are represented as a comma separated list of lvalue expressions being assigned a list of expressions, with both lists having the same length. In Go the expressions on the right hand side are evaluated left to right, with all the function calls being evaluated first. In GoLite, the expressions on the right hand side are evaluated left to right, regardless of the type of expression.

When dealing with multiple assignments, all the values are assigned simultaneously so the following code can be seen as an in-place swap:

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

When the left hand side of an assignment statement is an empty identifier, the expression on the right is evaluated, but the value is not stored. This is important when the right hand side contains a function call.

Go is also entirely pass by value, so the left hand side is given a new copy of whatever data is passed to it.

In Java, multiple assignments are not possible, except for the case where multiple values are given a single expression such as below:

```
int a, b, c;
a = b = c = 3;
```

Java is also pass by value, with the subtle difference being that when an Object is being passed the value is a reference. This is important because in order to capture all the semantics of Go, a deep copy will need to be written for all mutable constructs (structs, arrays, and slices).

In order to support multiple assignments, all the right hand side expressions are stores in temp variables, and then the temp variables are assigned to the left hand side lvalues. This preserves the execution order described above.

When the left hand side is a blank identifier a temporary variable, as mentioned above, is created to store the value and is never used. This way the right hand side is still evaluated properly.

3.7.1.6 Copying

Copying base types translates naturally as they are immutable. For the composite types there had to be a bit more work done. Structs have their own copy method, so when assigning a struct value to a struct we just create a new struct and copy each field. Arrays are assigned by doing an element-wise deep copy. Slices require quite a bit of care to get right.

When assigning one slice to another variable, each variable holds its own header but the reference is copied so they share the underlying array. Because of this, there is subtle behavior which can lead to accidental overwriting of data, such as in the case

```
var a []int
a = append(a, 0)
var b = a
var a = append(a,1)
var b = append(b,2)
```

Since they share underlying array, the last line would overwrite a[1].

3.7.1.7 Short Declarations

In Go, multiple variables can be declared at the same time, in a similar fashion to how variables can be assigned expressions. As long as one variable in the declaration has not been declared in the same scope, variables can be reused such as in the example:

```
var a int
a,b := 3,4
```

In Java as we are not able to redeclare variables, however during the symbol table stage we mark which variables have already been declared in the scope. First we evaluate all the right hand side expressions and store them in temp values. Then for each lvalue on the left hand side, we check if the value has been declared in the scope, and if it has, we use the original, if it has not been, we create a new identifier to use.

3.7.1.8 If Statements

In Go, if statements are associated with an init statement, a conditional expression, a block of code that is evaluated if the expression evaluates to $true$, an optional else statement which is followed by another if statement (to create an else-if statement) or its own block of code executed if the conditional expression of the matching if evaluated to $false$.

In both Go and Java the else-if statement is equivalent to a regular if statement nested within an else block. When generating code in Java, we only used the latter method and did not use the else-if statements provided within Java. In both Go and Java, the else statement is optional within the if statement. In Go, else statements are matched to the previously unmatched if statement whose block ends on the same line in which the else statement starts, in other words the else statement is matched to the nearest, previously unmatched if statement that is in the same scope. In Java, else statements are matched to the nearest, previous unmatched if statement that is in the same scope. As clearly shown, matching else statements to their corresponding if statements will be a direct translation from Go to Java. In Go, curly braces are required to surround both if and else blocks. In Java, those braces are sometimes optional. For clarity and continuity, we always used curly braces to surround the if and else blocks in our generated Java code.

As noted in the specification provided in milestone 2, in GoLite the conditional expression must resolve to a boolean type. In Java, if statements require that the conditional expression must also resolve to a boolean type, thus this did not present an issue.

The largest difference between if statements in Go and Java is the init statement. Init statements can shadow variables in the same scope as the if statement, thus the init statement needs it own scope below the current working scope but above the if expression and block scope. To ensure proper scoping of an if statement with an init statement, we created a new code block in Java containing the init statement and the equivalent Java if statement for the Go if statement without the init statement. If in GoLite the init statement is part of an else-if statement, it does not cause a problem since we treated else-ifs as an if nested in an else. The block scope of such an else-if falls within the else block before the if statement.

3.7.1.9 Switch Statements

In GoLite a switch statement consists of an optional init statement, an optional switch expression, zero or more case clauses, and an optional default clause place anywhere among the other case clauses. Each case clause consists of a list of one or more expressions and a list of statements with optional breaks. The default clause has a list of statements with optional breaks but no expression list. If there is an init statement it gets executed before the rest of the switch statement, and belongs in its own scope. If there is a switch expression then all the expressions in each case clause must have a matching type. Each clause is checked in lexical order and if any of the expressions in the list are equal to the switch expression then the corresponding list of statements is executed. If there is no switch expression then all the case expressions must be bools and the first clause with an expression that evaluates to true is executed.

In both situations the default clause will always be executed if none of the cases match. Each statement list is implicitly interpreted to end with a break statement which exits the entire switch clause.

In Java, switch statements must have an expression that evaluates to an integer-convertible type, enum, or string. Each case clause must have exactly one constant or literal expression that has the same type as the original switch expression. The first clause with a matching value is executed, and all the subsequent statements are executed until a break is encountered. Once the end of the clause is reached, the execution flow falls through to the next clause. Optionally, the last case clause can instead be a default clause with no expression.

While switch statements are present in both languages and have many similarities, we decided to do the mapping using if and else-if statements. First a scope is opened to put the init statements. The switch statement is put inside a

```
while(true){}
```

statement to allow the use of break statements and then the switch expression is evaluated. Each case clause is an if or else-if statement. Note that at the end of the `while(true)` loop a break statement is placed to avoid infinite recursion. Although, this break statement is omitted if all cases return otherwise we would receive an unreachable statement error in Java. The conditions for each if statement is the disjunction of equality tests of each expression in the expression list. If the switch expression is omitted, it is set to `true` as that is its equivalent. If there is a default clause, it is placed at the end as an else statement no matter where it existed in the source code. The last statement in each if-elseif-else branch is a break statement to exit the while loop.

3.7.1.10 For Loops

In GoLite, there are 3 types of for loops: infinite loops, while loops, and 3-part loops.

3.7.1.10 Infinite Loops

In GoLite, the infinite loop is simply the for keyword followed by a block statement. During execution, the block statement is repeated until the loop is broken out of using a break statement. Since this is a fairly simple construct, it was easily mapped to Java. In Java, we used

```
while (true)
```

followed by the mapped block statement. Thus, resulting in the same infinite loop construct.

3.7.1.10 While Loops

In GoLite, the while loop is the for keyword, a boolean expression which is the condition to enter the loop, and finally a block statement. This directly maps to Java's while loop as a while loop's condition must also be a boolean.

3.7.1.10 3-Part Loop

In GoLite, the 3-part loop is constructed of the for keyword, an optional initialization statement, an optional boolean conditional expression, and an optional post statement. Both the init and post statement may be several statements within our target language which is not allowed within Java.

Special care needs to be taken to properly map 3-part loops to Java. To handle the init statement, a code block was created to surround the loop similarly to what was done for if statements. The 3-part loop is mapped to a while loop using the boolean conditional expression. If the conditional expression is missing, it is filled with `true` so it matches the execution of the 3-part loop in GoLite. Finally, the post statement was placed at the end of the while loop in order for it to be executed before checking the loops conditional expression.

Unfortunately, the described mapping above does not properly handle continue statements within the block statement. It was mentioned in class how this could be solved with goto statements or labels, but unfortunately Java does not support goto statements and Java labels have very limited functionality. Instead, we propagated the post statement within the code block and generated the code for the post statement before any continue statement, thus ensuring that the post statement is executed before the next iteration of the loop is done.

3.7.1.11 Printing

In GoLite, both `print` and `println` functions can take any number of arguments, including zero arguments. The `print` function, prints each argument one after another, whereas `println` leaves a space between each argument and finishes with a new line character. Additionally, rune type expressions are printed out as their integer representation and not their character representation.

To map this into Java, every argument is printed using `System.out.print()`, with the exception of expressions of type `float64`. To get the proper scientific notation format for float expressions, `System.out.printf(\"%%+7.6e\", floatExp)` was used. To get the proper formatting for `println`, after each argument was processed a space is printed out, unless it is the final argument in which a new

line is printed.

3.7.1.12 Function Calls

3.7.1.12 Function Declarations

In GoLite, functions have one or zero return types and an arbitrary number of input parameters. Each input parameter must be named and have its type specified in one of two ways. The first way is each parameter has the type specified right after it's defined, and in the second way many parameters are defined and the type is specified for all of them at once.

In Java, it is possible to have one or zero (void) return types, just as in GoLite, as well as arbitrary input parameters. The only minor difference is that Java does not support GoLite's second way of specifying types. However in our implementation, both methods of specifying parameter types yield the same AST- namely each parameter is a struct that has the type and name. Hence when generating code it is possible to explicitly specify the type of each parameter in both methods of GoLite function declaration.

For function signatures, the return types and parameter types are all translated into the corresponding Java type, except for structs where we substitute the name of the corresponding Class that we've created. Functions were written to convert all the Go types into valid Java code with particular attention placed on the two types below, as all others have a natural mapping.

3.7.1.12 Passing and Returning

In GoLite functions are both pass-by-value and return-by-value. Pass-by-value means that when variables are passed into a function, the function actually works with a copy of the variable and not the original one. So any modifications made to the structure do not persist once exiting the function. Return-by-value means that functions create a copy of the variable it returns before actually returning. This affects functions that return global variables, as the variable being returned isn't the true global but a copy of it.

Java is also pass-by-value, with the subtlety that when an Object is being passed, the value is the reference to the object. Hence, when generating Java code, before any function call we needed to create deep copies of all the reference types in the signature and before any return statement we needed to create a deep copy of the returned expression. This affected all the classes we generate from struct definitions as we have implemented a deep copy method for each one.

3.7.1.12 Special Functions

In GoLite there exist two special functions which have different behavior than any other functions. Init functions are a type of function which do not take any parameters and do not have a return type. There can be multiple init functions per file so they are called in the order that they are defined and they are the first functions called when the program is run. The main function is a function which also does not take any parameters and does not have a return type. The main function is called after all init functions execute.

In Java, it is not possible for multiple methods to have the same name except for when a method is being overloaded, which means they have a different set of parameters. Our solution was to have a counter as we declare init functions and assign each function a name of the form __golite__init_i, where i is the counter. This way we were be able to call all the init functions in the order they are defined from inside our Java main method. We defined the Go main function as a static Java function with the name __golite__main which we call from our Java main method after all the init functions.

3.7.2 Design

3.7.2.1 Identifier Table

We implemented an identifier table to keep track of almost all identifiers used throughout the GoLite program. The identifier table was implemented like the symbol table, by using a tree of hashmaps. Also similar to the symbol table, the identifier table was properly scoped to match the program. For each identifier, their GoLite identifier name and their scope was stored within the table.

Identifiers are added to the identifier table as they are declared by first checking if the identifier was used in the current scope. If it was not in the current scope, we traverse all the parent identifier tables, until we encounter the identifier. We then create a new entry for the identifier in the current scope by taking a count from the encountered identifier and incrementing it by one. Whenever we generate an identifier in the java code we suffix it with this scope count, and hence do not have to deal with the troubles of redeclaring variables.

3.7.2.2 Struct Types

We have implemented structs fully by using a hash table to keep track of the string representation as described above. To create all the classes to represent the necessary struct types we do one initial pass of the source code and any time there is a struct type node we either create a new class for it or we check

whether it already had a corresponding class. There is also a special cases we handle of a struct having a struct as a field that we haven't seen before, in which case we add both as objects and the appropriate reference.

All the classes we have created are stored in a hash table where the keys are the string representations of the structs and the values are the new class names in the output code. The string representation used for each struct replaced all type aliases with the original type, as we will not use reference types. After creating the hash table of classes, we traverse through it and write the Java class code to the beginning output file including the .equals(), .copy(), and constructor methods.

For the .equals() method we have to be careful to overload the Object .equals() method properly so that we do not have runtime errors. This means that we start off the method by including this code:

```
@override
public boolean equals(Object o){
  if(this == o) return true;
  if(!(o instanceof Struct_n)) return false;
  Struct_n other = (Struct_n)o;
  return *deep comparison*
}
```

We implement two constructors. One constructor with zero arguments, which initializes and zeros out all the fields, and another constructor which takes a struct object and performs a deep copy. All these methods are first written into a buffer, and then written into the output-file to be able to reuse code.

3.7.3 Arrays

While we mentioned Java is a pass-by-value language, if an array is passed as a parameter and modified, the array is modified. In Go however that is not the case, so we must have a way of deep copying an array. We created a method which takes two identifiers and the type of the array, and unrolls the array type in order to copy the entire array over, index by index, calling .copy() if the item is a struct or a slice. Special care should be taken as indexing in Go might not seem as intuitive depending on how you read the types

```
//Java
int[5][3] test = new int[5][3]
test[4][2] = 1
//Go
var a [3][5]int
a[2][4] = 1
```

In our implementation we used the Object wrappers for primitive types, and their default value is null, so whenever we declared an array it was completely

null instead of the corresponding 0 values. This meant a deep zero-out had to be implemented as well, to go through each value and put an appropriate zero value. For structs and slices we call our default constructor.

Both of the above functions were written in two ways, one to print directly to the output file, and one to print to a buffer. This was necessary to be able to generate the code and then put it in later. In the case of top level array declarations, since we cannot have a for loop, we add a method called "__golite__init_arrays()" which is called at the very beginning of execution to zero out all the arrays.

3.7.4 Testing

Our initial testing suite used for code generation was created based on the contents of the GoLite code generation tutorial. We then added tests throughout development to ensure that the solutions we implemented to solve any difficulties were robust. In addition to those tests, the tests from milestones I II were modified and used. This additional suite of tests caught many of the edge cases our original suite missed.

4 Final Thoughts

Our choice of tools and languages played a large role in the success of the project. Using tools and languages we were familiar with enabled us to focus on the complexities of compiling GoLite rather than struggling with the details of the tools and languages. The compiler was a challenging assignment, but doable within the time allotted based on our choices.

Additionally, our AST construction was a key component in making the later stages of the compiler manageable. Although, there were some complexity using the AST during the weeder phase, the overall structure of the AST was clear and logical. Basing the AST off the grammar and using linked lists to store recursively defined components, was in the end a good choice to represent GoLite.

Through this process we learned the importance of writing good tests. If we were to redo this project, we would write more tests at the beginning of the process and base the tests directly from the specification document provided to ensure we miss as few edge cases as possible. Having more tests cases greatly aided finding the functionalities we had yet to implement and to ensure the functionalities we had implemented were working correctly.

5 Contributions

Although we each touched most parts of the compiler, the following is a breakdown of our key contributions. Pavel worked on the scanner, pretty printer, symbol table, typechecker, and the declarations, slices, identifier table, and the generation the structs of the code generation. Additionally, Pavel fixed most of the errors found in the previous milestones. Kaylee created the majority of the test programs created throughout the project. Additionally, she worked on the AST and the expressions and statements in the code generation. Kabilan worked on the parser, weeder, typechecker, and worked on generating structs for the code generation.