GoLite Milestone III

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

1 Code Generation Language

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, makes 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, we are unable to support certain go constructs, such as redeclaring variables or initialization in if/switch statements. Java is also return by reference, whereas Go is return by value, meaning we will have to remember to create new objects as necessary.

2 Semantics & Mapping

2.1 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 will only use the latter method and 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 will always use 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 will not present as 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 will create 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, it will not cause a problem since we are treating else-ifs as an if nested in an else. The block scope of such an else-if will fall within the else block before the if statement.

2.2 Types

2.2.1 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.

Java supports all these types except strings as primitives but we will have a much easier time if we use the wrapper classes (Integer, Character, Double, and Boolean) for them instead. By using the wrapper classes we can take advantage of the usage of .equals(Object o), such as when we handle array equality, which we address below. Strings in Java have all the comparison and copying functionality we need already so we just use the built-in implementation available to us.

2.2.2 Reference Types

In GoLite, users can typedef 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 have already 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.

2.2.3 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.

Mapping these requirements to Java is not difficult as plain arrays already 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.

2.2.4 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.

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].
In Java this behaviour is very similar to an ArrayList, however there are some quirks where the behavior is different, which will require 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 construct 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.

2.2.5 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 create a class for each new struct that we need. 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, when we generate the Java code all their corresponding classes will be defined at the top level. It is always possible to create self referencing classes in Java so we do not need to handle the situation with slices as a special case. When creating the classes we don't worry about whether a struct is tagged or untagged, we just create a class for each struct that appears and create a way to retrieve the class during compilation.

In Go, structs are comparable given that all their fields are comparable. In Java, unless overriden, the .equals() method compares references, so if all the fields are comparable, we will also need to generate a static .equals() method which compares all of the corresponding fields. Also since Java is pass by reference, we will need generate a clone method which properly clones each field, including other structs and slices. Another part of being comparable in Go is two structs can be defined in two place as long as all the parameters 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 will traverse the whole program and gather all usages of the word "struct" and unroll them recursively and then store them in a seperate table if they do not already exist. This way if two structs are identical they will be the same object and we can use the .equals() method. Also since we are throwing away all references, we will 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, because at the time of declaring s1, int is a base type but at the type of declaring s2, int is a struct.

2.3 Functions

2.3.1 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 return types, just as in GoLite, as well as arbitrary input parameters. The only minor difference is that Java does not support the second way of specifying types in GoLite. 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.

2.3.2 Passing and Returning

In GoLite function 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 creates 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 pass-by-reference and pass-by-value. Hence, to generate Java code, before any function call we must create deep copies of all the reference types in the signature and before any return statement we must create a deep copy of the returned expression. This affects all the classes we generate from struct definitions as we have to implement a deep copy method for each one.

2.3.3 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 is 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 will be able to call all the init functions in the order they are defined from inside our java main method. We will define the Go main function as a static java function with the name _golite_main which we will call from our Java main method after all the init functions.

3 Code Generation Implementation

Aside from the constructs described below, we also generate a proper public java class with a main method. All identifiers are prefixed to denote that they're GoLite functions so they don't interfere with built in Java functions. All helper methods and helper classes are added to the begining of the class before we being generating the main public class.

3.1 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. 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.

3.2 Declarations

In our codegen we've only implemented function signatures as we have not done statements yet. 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.

3.3 Types

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.

3.3.1 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 has 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 we will use for each struct will replace 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() method.

3.3.2 Slices

To implement slices which work for any type and function identically to Go we created a generic Slice class as we discussed in the section above as well as all the methods necessary to perfectly simulate a slice. The generic aspect of this makes it very easy to translate Go Slice operations into Java.

4 Contributions

Kaylee worked on most of the tests and the implementation of generating expressions. Pavel worked on some tests, generating function signatures, generating structs, and the Java implementation of slice types. Kabilan worked on some tests and generating structs.