GOJAVA COMPILER REPORT

May 1, 2020

Name: Matthew Lesko-Krleza
Student ID: 260692352
McGill University
School of Computer Science
matthew.lesko-krleza@mail.mcgill.ca

Name: Sophia Hu
Student ID: 260705681
McGill University
School of Computer Science
yu.t.hu@mail.mcgill.ca

Contents

1 Introduction, Language and Tool Choices

Go is a statically typed, compiled programming language designed at Google. Fun fact: it originally started as a 20-percent side project, meaning that engineers working on Go were still expected to do their normal work on top of designing Go, wild!!! Since its release in 2009, it has become quite a popular language, in part due to its simple syntax, concurrency support, and open-source community. We definitely appreciated the detailed Go specification, which helped us build our compiler: `https://golang.org/ref/spec#Switch_statements`.

We built a compiler for a subset of Go, called **GoLite**. Golite includes many Go features, including multiple variable declarations in one line, short declarations, untagged structs, slices, and optional semi-colons.

Our compiler is written in C, and we used the flex *(fast lexical analyzer)* and bison toolchains. We chose this toolchain because of our positive experience with it, and we wanted to create a fast compiler. Our target language is Java. We chose Java because we are comfortable in using it, and we wanted to leverage its generic class feature as well as several of its APIs for arrays, and string formatting.

2 Scanner

To implement the scanner, we followed the GoLite language specification handed out by course overlord. There weren't many noticeable design decisions to make, and the scanner's implementation using flex was similarly implemented as the one for MiniLang The major interesting notes include the following:

Optional Semi-Colons

Semi-colons were inserted after a given token when necessary if it encountered either a new line or EOF. The rules for inserting a semi-colons are included in the GoLite specification. In addition, there were instructions on how to correctly implement semi-colons in the COMP 520 repository.

Multi-line comments

The multi-line comments are handled by consulting the flex documentation for C-style comments.

Strings

We allowed any character inside the quotes "" except for the unescaped string literal.

Rune literals

If a rune literal was a valid escaped character, like '\n', we set '\n' as the value in `yylval.runeval` identifier.

3 Parser

For implementing our compiler's grammar, we consulted the GoLang program specification and attempted to replicate its grammar specifications as closely as possible. Initially, our rule for expressions was significantly different from GoLang's specifications. More precisely, we initially didn't use their design involving the `primaryExp` for expression rules. However, after receiving the solution tests for milestone 1, we made significant changes to our parser as to incorporate GoLang's expression rules more closely: to implement the `primaryExp` rule. Now, we have split up unary and primary expressions into two separate rules. We did so because we were facing failing tests and issues due to function call expressions. This took a significant amount of time to fix, but resolved numerous bugs. Additionally, for lists of expressions, or identifiers, we've written non-terminals such as `expressions` and `identifiers` that contain rules that recursively call on themselves or on a single expression/identifier. When certain non-terminals, such as assignment statements and function calls, allow for multiple expressions or identifiers, we use the `expressions` or `identifiers` non-terminals respectively. We used an identical logic and similar rules for type and variable declarations that can involve multiple types and variables respectively, as well as top-level declarations. Statements are handled similarly to how they are handled in the GoLang specification where simple statements are a subset of statements. If statements are handled using a recursive strategy as discussed in class so that we can support if-else statements

4 AST

When designing the AST, we largely followed the structure of the grammar. We also wanted to create as simple and minimal an AST as possible.

4.1 Handling Lists

An interesting component of GoLite is that it allows declaring multiple identifiers or expressions in the same line. To handle such constructs, the AST uses a linked list structure of other AST nodes. An example code snippet follows:

```
struct IDENT {
    char *ident;
    IDENT *next;
};
```

We used a linked-list structure because it would allow us to easily use it in recursive function calling, which is something we do a lot in the following components of the compiler pipeline.

4.2 AST Structure

The program AST node `PROG` contains exactly one package identifier and a linked list of top level declarations, `TOPLEVELDECL`.

Top level declarations are one of:

- Variable declarations `VARSPEC`;
  - Since a variable declaration can be distributed over multiple lines, the AST will parse this as a list. `VARSPEC` has a `next` pointer that handles a list of variable declarations.
  - A single variable declaration can have multiple identifiers and expressions, so both the `IDENT` and `EXP` AST nodes have `next` pointers.
- Type declarations `TYPESPEC`;
  - This is similar to `VARSPEC`, except each identifier will have a `TYPE`. The `TYPE` node can be a slice, array, struct, or an inferred type that we will later determine in the typechecker.
  - It also contains a `TypeSpecKind` enum to differentiate between a type declaration, and a function parameter list. Instead of having a new AST node to represent a function parameter list, we wanted to simplify the AST and use the `TYPESPEC` AST node.
- Function declarations `FUNC`;

- Each function has a function name, optional return `TYPE`, a `TYPESPEC` of input parameters, and a root statement `STMT` (which is a linked list structure).

We separated the AST into the following main nodes:

```
PROG, TOPLEVELDECL, TYPE, STMT, EXP, FUNC
```

and helper structures:

```
IDENT, VARSPEC, TYPESPEC, STRUCTSPEC, EXPRCASECLAUSE
```

We desired to create this design of AST nodes because its representation is identical to the parser rules we created. We wanted to keep its representation as similar as possible to the parser rules, since the rules are almost one-to-one representations of the actual GoLang language.

4.3 Statements

Statements in the AST are represented with a recursive `STMT` structure with a `StatementKind` to differentiate the statement type. Some interesting notes:

4.3.1 Simple Statements

The concept of "simple statements" don't exist in the AST. Simple statements are just statements for simplification.

4.3.2 Reducing Kinds of Statements

In some cases, we were able to simplify the `STMT` node by combining two similar kinds of statements into one statement with extra information. For example, instead of having two different statement kinds for an increment and decrement statement, we made them the same statement kind `k_stmtKindIncDec`. We use an `amount` flag that differentiates an increment `amount = 1` or decrement `amount = -1`. Another example is with `print` and `println`. We used an `newLine` flag to differentiate the two.

4.3.3 Switch Statements

We created a recursive data type `EXPRCASECLAUSE` to handle case clauses in switch statements that includes an `EXP *expList, CaseClauseKind kind`, and `STMT *stmtList`.

The expression list is the condition that is mapped to the case clause, and the statement list is the list of statements that follow. `CaseClauseKind` include `k_caseClause` and `k_defaultClause` to simplify work in weeding, when the weed needs to make sure each switch statement has a default clause.

4.4 Expressions

Expressions in Golite are handled very similarly to Minilang. For example, every binary expression has a type, but contains the same `lhs` and `rhs` values. This is intuitive to use for the programmer, and allows us to adopt a similar design which we already had experience with when creating Minilang.

5 Weeder

The rationale behind our weeder is that we'd like a weed phase than can remove as many illegal statements and expressions as possible to save us work in downstream tasks. The weeder's most important feature is its ability in dealing with blank identifiers, since we'd like to avoid dealing with blank identifiers in expressions that evaluate to a value in the type check phase.

Checks done in the weeder include:

1. Verifying whether a post-statement in a for loop doesn't contain a short declaration. We do this by checking the post-statement `STMT` AST node's `StatementKind`.

2. Verifying that a break statement is within a for-loop or a switch statement block. We do this by keeping a count of the depth of previous for-loop or switch statements, and checking that the current break statement is within a depth of greater than 0.

3. Verifying that a continue statement is within a for-loop block. This is similar to Rule 2, but we only verify that the current continue statement is within a positive depth of previous for-loop blocks.

4. Verifying that expression statements are only function calls. We do this by checking the `EXP` AST node's `ExpressionKind`.

5. Verifying that an assignment statement contains the same number of identifiers on the left-hand-side of the assignment operator as expressions on the right-hand-side of the assignment operator. We do this by counting the number of identifiers and expressions in an assignment statement, and verifying that their counts match.

6. Verifying that there is exactly one identifier and one expression in an assignment operation. This is implemented similarly to rule 5, counting the number of identifiers and expressions and checking they both count to 1.

7. Verifying that a function with a return type ends in a terminating statement. This is implemented using a `bool isTerminatingStmt(STMT *s)` helper function. Terminating statements include return statements, and special kinds of if, for, and switch statements. For more details, refer to the golang spec: `https://golang.org/ref/spec#Terminating_statements`.

Whenever one of these checks don't hold true, the weeder prints a descriptive error to the console and quits compilation.

The main design choice that we'd like to discuss is how we weed expressions differently depending on whether the expression should evaluate to a value or not. We implemented rules for the former in a function called `weedEXP_eval` and rules for the latter in a function called `weedEXP_nonEval`. We separated the two functions since the linguistic rules for expressions that evaluate to values differ greatly from expressions that don't evaluate to a value.

When weeding expressions that should evaluate to a value, we make sure that the blank identifier '`_`' cannot be used as a value for any binary, unary, field access, array access, function call, or builtin function call expression.

As for expressions that don't evaluate to a value, such as expressions on the left-hand side of an assignment statement, we implemented rules that only allow the expression to either be an identifier, an array access, or a slice access. Additionally, in operation assignments that don't include the ':=' and '=' operators, we implemented a rule that doesn't allow the left-hand-side operand to be a blank identifier `_`, otherwise blank identifiers are allowed. This helps us avoid needing to perform additional special behavior for blank identifiers in the symbol table and type check phases of the compiler.

6 Pretty Printer

Pretty printing is done by traversing the AST to output a valid GoLite program. Special care was required when indenting and ending the lines of statements. If, switch, and for statement nodes have children that are statements that should not be indented or be the end of a line. The pretty printer handles this by not indenting or ending the line when the parent node is one of the previously mentioned statements.

7 Symbol Table

The symbol table's purpose is to store declared symbols and type information into the AST. Additionally, given some use of a symbol, the compiler checks if the symbol exists in the current or in any parent scope. If the symbol doesn't exist, then the compiler throws a relevant error message. It also throws an error message if the symbol has already been declared in the same scope.

The symbol pass is done before the typechecker because the typechecker is dependent on information in the AST that is provided by the symbol pass. The first design choice we'd like to discuss is the addition of the `SYMBOL struct` data structure. A code snippet of the data structure is available below:

```c
typedef struct SYMBOL {
  char *identifier;
  SymbolKind kind;
  int shadowNum;
  union {
    TYPE_SPEC* typeSpec;
    VAR_SPEC* varSpec;
    FUNC_SPEC* funcSpec;
  } val;
  struct SYMBOL *next;
} SYMBOL;
```

The symbol is used to store information for type, variable, and function declarations. It additionally stores the identifier of the symbol, the kind of symbol (type, variable, or function), and the shadow depth. The pointer to the next symbol allows us to implement a linked-list of symbols which can be traversed in a symbol table.

7.1 Weaving

The symbol table is weaved with the AST. When we traverse the symbol table, as we look up identifiers and their prior uses, we are able to directly store type information into the AST node. This helps simplify typechecking. We decided to implement a weaving type of symbol table feature, by adding the symbol to the `EXP` AST node's identifier value. We did so because we needed to store the kind of symbol (whether it's a function, variable, or type), as well as the shadow depth for the identifier. Otherwise, another possibility would be to store the shadow depth directly as a field in the identifier field of the `EXP` AST node, and create a new enum to identify the kind of symbol. This latter approach is a little more convoluted, since now various pieces of the symbol's information are spread across different fields as opposed to just one symbol field.

7.2 Base Type Symbols

When the symbol table is created, we first add GoLite base types to the global scope of the input program. Symbols need to be created for these base types because they are not reserved keywords in GoLite. This allows the user to be able to use the base types for declarations, but also allows for the user to be able to re-declare them as something else, which is a GoLite feature requirement. The global scope is higher than the program scope, which allows programmers to re-declare base types.

We create a symbol of kind `k_symbolKindType` for each base type: `int`, `float64`, `rune`, `string`, and `bool`. We also create a symbol of kind `k_symbolKindConstant` for the `true` and `false` constants. We use the special `k_symbolKindConstant` symbol kind for the `true` and `false` constants, so that they can be differentiated from type symbols, function symbols, and variable symbols.

7.3 Top-Level Declarations

The entry point for the symbol phase is the root node of the AST. The rest of the AST is traversed by visiting the first top-level declaration.

The symbols in the top-level declaration are stored either through a `symFUNC`, `symTYPESPEC` or `symVARSPEC` function respectively for function, type, and variable declarations. We pass the current symbol table to each function call so that symbols can be queried for, created in the scope, and new scopes can be extended from the given scope. This is an example of one area where the linked-list structure comes in handy, allowing the programmer to easily use a recursive function call structure to parse symbols.

7.4 Function Declarations

If the function declaration has a return type, we query for the symbol that represents the return type and assign the symbol's type information to the return type's type information. Afterwards, we create a symbol for the function name itself and add it to the given symbol table which represents the current scope.

Then, a child symbol table is created which acts as an inner scope. The function's input parameters are added as symbols with their respective declared types into the aforementioned inner scope of the function. The function statements and input parameters are in the same inner scope.

7.5 Statements

If a statement is a type or variable declaration, then the `symTYPESPEC()` and `symVARSPEC()` function calls are used respectively once again.

Any kind of expressions and block statements that make up the current statement, like in cases for expression statements, for-loop statements, if-statements, switch statements, and return statements, are further parsed for their symbols using the `symEXP()` and `symSTMT()` function calls appropriately.

Additionally, for any statement that contains block statements, a symbol table is created and extended on the one given to the function for any kind of initializing statement (part of for-loop, and if statements), then another symbol table is created and extended from the previously created one for any differentiating block statement.

So in the case of an if-statement, an inner scope is declared for the initializing statement, then two other scopes are declared that extend from this inner scope, and are used for the true-condition block statement and for the false-condition block statement respectively. Most notably, assignment statements are split between colon-assign and other assignment statements. *Aside:* The `else-if` structure does not exist in our AST. The `else-if` simply becomes an `if` nested inside an `else`.

7.6 Expressions

When an expression is simply an identifier, its symbol is queried from the given symbol table, and is added as a field into the identifier's respective EXP AST node, which will be used later on typecheck for type checking the identifier's type. Furthermore, any sort of binary, unary, field access, array access, or built-in expressions have their respective expressions passed as arguments to the `symEXP` function call so that symbols can be checked for existence.

7.7 Variable Declaration

In a variable declaration, the symbol for the identifier and the variable's type are queried to see if the exist. If the symbol for the variable already exists in the current scope, then an error is thrown, whereas if a type is given, then a symbol for the type identifier is queried to see if it exists. We symbol the RHS before entering the new declarations.

Additionally, identifiers are associated with their type's symbol's type information, so that they can be typechecked in downstream tasks.

7.8 Short Declaration

Before adding any symbols into the symbol table, we ensure that, **1)** at least one identifier on the LHS is undeclared, and **2)** all identifiers on the LHS are unique. If either condition is not fulfilled, we throw an error.

Then, for each RHS expression, we call `symEXP`. Also, for each identifier on the LHS, we check if it has been previously declared. We add this `colonAssignDeclared` flag onto the AST, which becomes useful later in code generation. For undeclared identifiers, we create a new symbol of type `k_symbolKindVar`. Then, we call `symEXP` on the LHS regardless of if the symbol was declared or undeclared.

7.9 Type Declaration

Type declarations are done similarly to variable declarations, where the newly defined type's identifier is queried as a symbol in the current scope to see if there already exists a type declaration of the same symbol in the same scope. The type declaration's type is also queried to see if it exists. If the uniqueness and existence checks pass, then the newly defined type is assigned a symbol and added to the symbol table.

To handle recursive types, we added a `*parent` pointer to our AST's `TYPE` struct. For example, in the code block `type num int`, the type `num` would have a parent pointer to type `int`.

A challenging implementation was for declaring new struct types. For determining whether an identifier in a struct declaration has already been declared, we initialized a separate symbol table only for identifiers in a struct. As the struct's field are parsed and its symbols are added to this separate table, any repeated fields are discovered and an appropriate error message is displayed to the user. Additionally, we determine if each field's type's symbol exists from the current scope of code. So the function's signature looks like this: `void symSTRUCTSPEC(STRUCTSPEC *ss, SymbolTable *scope, SymbolTable *structScope)`. The struct spec is a linked list of a struct's fields and their associated types. The `*scope` is the symbol table used to determine if a field's type's symbol exists, and the `*structScope` is used a separate symbol table to determine uniqueness for each field's identifier in the struct declaration. We designed it in this way, because it allowed us to use less code (can simply use the same code for putting and getting symbols as the other functions) and can retrieve symbols from the separate struct symbol table in O(1) time.

7.10 Printing the Symbol Table

Any time a new symbol is declared it also gets printed if the `print_sym_table` flag is set, and its scope is defined by the symbol table in which it was declared. Their types aren't typechecked yet, so some symbols have a type `<infer>` that needs to be further typechecked, most notably for short-hand declared variables.

8 TYPECHECKER

At the typecheck stage, each AST node is type checked as per the requirements in the Milestone 2 Specifications handout. Note that we do not need to type check declarations, since those are well-typed as per the symbol phase.

8.1 Type Equality

Trivially, two types `TYPE *t1` and `TYPE *t2` are equal if they both resolve to the same base types.

Two slice types are identical if they have identical element types. Two array types are identical if they have identical element types and the same array length. Two struct types are identical if they have the same sequence of fields, and the corresponding fields have the same names and identical types.

Recursively-defined types are equal if they refer to the same type specification. In our compiler, we check that the two types have the same identifier name; however, this is not sufficient. We should also check that they were defined in the same scope using the corresponding symbol's `shadowNum`.

8.2 Statements

1. Empty, Break, Continue statements: These are trivially well-typed. There is no symbol to type check. *Aside:* Empty statements are represented explicitly in the AST with the statement kind `k_stmtKindEmpty`.

2. Expression statements: We simply type check the expression;

3. Return statements: We type check that a return statement returns a symbol of the same type as that of the function declaration's return type;

4. Increment/Decrement statements: Here we need to type check that the expression resolves to a numeric type;
5. Assign and op-assign statements: We have to make sure the left-hand side is addressable, and that the left-hand side and right-hand side expressions resolve to the same types. We wrote some helper functions to verify these conditions. These functions simply either return true or false depending if a type's resolved base type is addressable (as specified in the language specifications) and whether two types are equal;
6. Short declarations: For each LHS-RHS expression pair in a short declaration statement, we first make sure that the right-hand side expression is well-typed. Then, we ensure that the LHS expression is well-typed and the identifier refers to a symbol of kind `k_symbolKindVar`. We also check for type equality. The LHS expression type must be type equal the RHS expression type.

   Aside: The symbol table ensures that at least one variable on the left-hand side is undeclared in the current scope.
7. Print statements: We simply type check that the expression for a print statement resolves to a base type (int, float64, string, rune);
8. Variable declarations: Variable declarations are type checked if they're assigned an expression and specified a type. The right-hand side expression must be type checked by using the `symEXP()` function, and if a type is specified, then the type of the right-hand side and the type specified must match.
9. If-statements: The statements and expressions in an if-statement are all type checked. Additionally, we verify whether the condition expression resolves to a boolean, and throw an error message otherwise;
10. For-loop statements: Similarly, we type check each statement and expression, and verify whether the condition expression resolves to a boolean, and throw an error message otherwise;
11. Switch Statements: We type check the switch statement's statements and expressions. Additionally, we check if the condition expression contains a symbol and assign the switch-statement's expression type to that of the symbol's type, otherwise we set it to a boolean type.

8.3 Expression

Similarly to creating the symbols for an expression, depending on the expression AST node kind, we either associate a type to a symbol, or verify if some expression is of an appropriate type. The language specifications details what an expression's type must resolve to if it's a certain expression kind. For brevity reasons, we urge the reader to read the language specifications handout specifications on expression types, since we followed this document for determining how to type check each kind of expression.

A trickier implementations we'd like to discuss is that for field access expressions. First we need to verify that the expression's object type checks, but the challenging part is determining the field's type.

To do so, we iterate over all of the object's type's struct specification data structures. We first validate if the queried field matches with a field declared in a struct spec. Then we resolve the entire expression to the type from this field.

This was tricky because accessing this data requires a long list of data structure referencing as noted here: `e->val.fieldAccess.object->type->val.structType`. Fun fact: According to the Law of Demeter, this is super error-prone :) !

8.4 Expression Case Clauses

Type checking expression case clauses is involves type checking the `init` type and the `expr` type. We also type check that all the switch expression types are well-typed and have the same type as `expr`.

8.5 Functions

Function declarations type check if the statements of its body type check. If specified, the type of a function is its return type. Note that return statements need to be the same type of the enclosing function.

Aside: For functions that return a value, checking that the statement body ends in a terminating statement is done in the weeding phase.

9 Codegen

9.1 Java Class Names and File Names

Since Java does not support class names with `_` or `0-9`, file names should contain only alphabetical characters. We also append filenames with `GoLite` to avoid conflicts with reserved keywords. Additionally, the name of the outer most class in a Java file needs to be identical to the file name, it's a syntactical rule. Therefore, we change the file name and the class name so that it can be compiled.

In C, we used a `removeNonAlpha` function that used the C builtin `isalpha()` to remove all non-alpha characters. We created and used a function `lastForwardSlashIndex()` that returned the index of the last forward slash to make sure to only modify the file name and not the directory.

In our Bash `execute.sh` script, we used regex:

```
FILENAME=${FILENAME//[0-9_-]/}
FILENAME="${FILENAME}GoLite"
```

Since the GoLang input file name no longer matches with the modified Java file name, and we need to make the same changes as what our compiler did to the output file name.

9.2 Types

9.2.1 Base and Inherited Types

We use a `getStringFromType(TYPE *t, bool isPrimitive)` helper function in codegen that takes in a type and outputs the appropriate string. Note that inherited types can use their base types. For example, given the type `type num int`, our function would return `int`.

For most cases, we use primitive types in Java. This is because the default uninitialized value in primitive types in Java tend to match up with Golite. For example, `int[] arr = new int[3]` will have all the elements initialized to `0`, whereas if we used the non-primitive type `Integer`, the values would be initialized to `null`.

A rune type is interpreted as an `int`. This is because rune type expressions are printed out as their integer representation, not their character representation.

9.2.2 Slices

In Go, a slice is a dynamically-sized, flexible array. It does not need to be declared with a size, nor is it limited by a size. Slices have capacity and length, and as items get added to the slice, it will automatically expand its capacity.

The default size of a slice is 0. As we add elements, if the new length exceeds the capacity, we will increase the capacity by a linearly-increasing exponent power of base 2: first by 2, then 4, then 8, and so on, continually doubling the amount that we expand the slice.

To implement this type in Java, we created a generic supporting `class Slice<T>`, which can hold a slice of any type. The underlying structure of the `Slice` class in an ArrayList, but we also used additional `cap` and `len` fields to mirror the functionality of Go slices. When appending to the `Slice`, we check to see if we are at capacity. If so, we double the capacity and copy over all the elements to a new Slice. The following is a code snippet from our Java `Slice` class:

```java
public Slice<T> append(T node) {
    // If at capacity, double capacity and copy over all elements
    if (len == cap) {
        ArrayList<T> newData = new ArrayList<>();
        int doubleCap = cap == 0 ? 2 : cap*2;
        for (int i=0; i<len; i++) {
            newData.add(i, data.get(i));
        }
        newData.add(len, node);
        return new Slice<T>(doubleCap, len+1, newData);
    }
    data.add(len, node);
    return new Slice<T>(cap, len+1, data);
}
```

When we assign a `Slice` element using `put(int index, T node)` or retrieve the value of a `Slice` element using `get(int index)`, we first check to see if the index is out-of-bounds. The index is out of bounds if it is less than zero, greater than or equal to the underlying length, or greater than or equal to the underlying capacity. If this is the case, we throw an `ArrayOutOfBoundsException`.

9.2.3 Structs

Untagged and Tagged Structs: To handle untagged structs requires a global traverse of the program prior to any code generation. Our function `generateGlobalStructs` traverses the Golite program, and for each unique struct (tagged and untagged) encountered in the program, outputs a Java object accessible in the global scope.

To ensure each struct outputted as a Java object is unique, we keep a list of structs that have already been created in the program. Before generating a new struct, we first check to see if it is already in the list using `isEqualType`, a function written in `typecheck.c` to check for type equality.

To ensure we can reference the correct Java object later in codegen, we mark each struct created with a unique identifier, `codegenTag`. We annotated our `TYPE` node in our AST to include this `codegenTag`:

```c
struct TYPE {
    union {
        /* ... */
        struct { STRUCTSPEC *structSpec; int codegenTag; } structType;
    }
}
```

We name our Java objects as `__golite__struct_tagNumber`. For example: The first struct encountered in our program would be outputted as `class __golite__struct_0`. The second struct, if type unique from the first, would be outputted as `class __golite__struct_1`. Say the third struct is type equal to the first struct with `codegenTag` 0. In this case, we would not output a Java object, but we would annotate the AST with the `codegenTag` 0.

Now, during codegen, when we encounter a struct, we simply output the variable and call the constructor of the appropriate Java object using the `codegenTag`. Note that when we encounter a struct during a type declaration, we omit any code generation since we do not need to re-declare the type (it is already declared in the global struct).

Pass-by value: When structs are assigned or passed into a function, they are passed by value. To implement this functionality, each Java class that represented a struct has a `copy()` helper function. Suppose that the Java object is called `__golite__struct_0`. The `copy()` function will create a new `__golite__struct_0` object, copy each field (as long as the field name is not the blank identifier), and return the newly created object.

Now, during codegen, when assigning a struct or passing a struct into a function, we need only call the `.copy()` function to implement pass-by value.

String arrays: In Golite, the default type of Strings is `""`; however, in Java, the default type of Strings is `null`. Our normal behaviour to fix this, given a variable initialized as a String array, is to immediately follow the variable declaration with the `Arrays.fill` command:

```
String[] a = new String[5];
Arrays.fill(a, "");
```

However, field declarations in Java cannot be followed with statements. Statements can only be in functions.

The solution is to generate a constructor to take care of this edge case. The constructor will go through each field in the struct, and if it is an array of type String, we will output the `Arrays.fill` statement.

9.3 Statements

9.3.1 Variable Declarations

For Strings, the uninitialized value in Java is `null` instead of the `""` as in Golite. Therefore, if we print an uninitialized String, the behaviour will not be as expected. Given a variable declaration of type `String`, if there is no associated expression, we assign the String a value of `stringName = ""`. Given a variable declaration of a `String` array, we immediately output a `Arrays.fill(arrayName, "")` statement following the variable declaration.

There is an exception to using primitive types. When we initialize slices, we must use the non-primitive type: `Slice<Integer> slice = new Slice<>();` Java only supports reference types for generics, using a primitive type throws an error.

9.3.2 Type Declarations

Non-composite type declarations are omitted, since we simply use a declared type's base type for generation. For structs, we generate a new class, where the class contains the structs fields and are accessed using dot notation.

9.3.3 Function Declarations

Since the output program is ran within the input file's `public static void main()` method, all generated functions have to be made static as well. If the identifier for a function is the blank identifier, the function is simply not generated, since valid programs never call blank identifier functions.

Function Parameters

Before executing any statements in the body of the function, we go through all the function parameters to implement pass-by value. Our behaviour when encountering a struct or array is the same as described in the Assignment subsection's Pass-by Value.

Here is an example. For the sake of readability, the prefix `__golite__` is omitted from variable names. The parameter `arr` is an array, while the parameter `c` is a struct.

```
public static void foo(int[] arr, struct__1 c) {
        c = c.copy();
        int[] tmpArray__0 = new int[arr.length];
        for (int i=0; i<tmpArray__0.length; i++)
            tmpArray__0[i] = arr[i];
        arr = tmpArray__0;
        /* Rest of function */
}
```

Additionally, we support blank identifiers in function parameters by generating a blank identifier parameter placeholder that's never actually used within the function's body.

9.3.4 Printing

To handle printing, we used a `traverseExpForPrint(EXP *e)` function that called `System.out.print(e)`. For float expressions, we used `System.out.printf("\%\%+.6e", e)` to output the correct scientific notation.

In Golite, both the `print` and `println` functions can take as input any number of arguments. Therefore, `traverseExpForPrint()` was a recursive function called all expressions in the recursive linked list structure `EXP`.

In the case of `println`, after each expression, as long as it wasn't the last expression, we printed out a space to separate the arguments. If it was the last expression, it is followed by a new line instead of a space.

9.3.5 Assignment

Pass-by value

Since Go is pass-by value, we need to implement extra functionality when assigning structs and arrays:

When assigning a struct, we simply call the `.copy()` function of the right-hand side expression. Each Java object representing a struct comes with a `copy()` helper function. This is described further in the Structs subsection, in *Pass-by value*.

When assigning an array, we create a temporary array `tmp`. We copy all the elements of the original array into the `tmp` array using a for loop to traverse the length of the original array. Finally, we assign the `tmp` array into the original array.

Multiple Assignments

In Golite, assigning multiple values in one line is supported. For example, given the statement `a, b = b, a` all the values are assigned simultaneously and the statement is essentially an in-place swap. In Java, this is not possible.

To support multiple assignments, all the right-hand side expressions are stored in temporary variables. The temporary variables are then assigned to the left-hand side expressions. When the left-hand side is a blank identifier, the temporary variable used to store the assignment is never used. However, we still need to evaluate the corresponding right-hand side expression in case of possible side effects - for example, a function call that prints to stdout.

Short declarations

In short declarations, multiple variables can be declared and re-declared at the same time. As long as there is one undeclared variable in the scope, a short declaration is valid.

In Java, we are not allowed to redeclare variables. To know which variables on the left-hand side are declared and which are not declared, we annotate the AST with a `colonAssignDeclared` flag. In the symbol table phase, we mark the identifiers accordingly.

During codegen, we traverse each variable on the left-hand side of a short declaration. If it is undeclared, we generate a variable declaration. After traversing all the left-hand side declarations, all the variables have now been declared! We can then treat the short declaration as a normal assign statement.

9.3.6 Break and Continue

The design choice we'd like to discuss with `break` and `continue` statements is that we enclose them with a conditional statement that always evaluates to true. For example, we'll enclose a `break` statement like so:

```
if (true) {break Loop_0;}
```

We do so because we append any for-loop post statements to the body of a for-loop, so it's possible that we append statements after a break or continue statement. Java's compiler throws errors for any unreachable statements, so this conditionally-enclosed-statement trick bypasses the compilers checks for unreachable statements. The reason why this works is not definitively known, however we believe that the inclusion of the conditional tricks the compiler into behaving like the break or continue statement isn't definitively achievable, hence making it believe that any statements coming after the break or continue statement are reachable.

9.3.7 If-Statements

Generating if statements is straightforward, we simply generate the `if` keyword, with its respective parentheses and conditional expression. Generating else statements is also straightforward, we simply generate the the `else` statement and its body. However, for else-if-statements, we generate an `else` statement for the prior if-statement, and generate the else-if-statement conditional as just another if-statement within the else body scope before the prior if-statement's else-statement body.

9.3.8 For-Loop Statements

To be able to create every kind of for-loop, we generated `while` loops enclosed in their own scope with their respective initialization and post statements. Some of the major design choices we would like to discuss are highlighted in the code snippet below:

```
{
        //Init statements
        int __golite__x1 = 0;
        int __golite__x2 = 0;
5
6          boolean do_post_loop_0 = false;
7          boolean continue_loop_0 = true;
8          boolean true_literal_0 = true;
9          Loop_0:
10         while(continue_loop_0)
11         {
12                 if (do_post_loop_0)
13                 {
14                         __golite__x1++;
15                 }
16                 while (__golite__x1 < 3)
17                 {
18                     if (__golite__x1 == 2)
19                     {
20                             if (true) {do_post_loop_0 = true; continue
       Loop_0;}
21                         }
22                         if (__golite__x1 == 3)
23                         {
24                             if (true) break Loop_0;
25                         }
26                         __golite__x1++;
27                 }
28                 continue_loop_0 = false;
29                 break Loop_0;
30         }
31 }
```

We need to create a new scope for the initialization statements since they need to be contained only within the for-loop scope, hence the newly defined scope's curly brackets at lines 1 and 31. This allows us to generate multiple initialization statements as seen at lines 3 and 4. Since, Java doesn't support `goto` statements, we made use of labelled break statements and used enclosing while-loops as the one generated at line 10: `while(continue_loop_0_)`. Java labels have to be followed by a loop statement, such as a while loop, hence the use of enclosed while-loops. We append post-statements to the body of a while loop such as line 26. However, it's possible that a `continue` statement is used such as at line 20, making the appended post statement unreachable. Therefore, we included a conditional as seen at lines 12 to 15, that executes the post-statements. The post-statements at that location only ever occurs when a continue statement is executed, to avoid executing them before the first iteration, hence the use of the `do_post_loop_0` variable at lines 6, 12 set to true with the continue statement, and 20. Additionally, now we can break from specific for-loops using labelled break statements such as at line 24, this allows us to better control the behavior of the output program such that it behaves identically to the source program. Next, we only want outermost enclosing while-loop to ever iterate once on its own without a continue statement, hence the use of the `continue_loop_0` variable at lines 8 where it's set to true, 10, and 28 where it's set to false. Finally, we use the `true_literal_0` variable when generating infinite while-loop statements. The reason why we use a variable instead of the literal value true, is that otherwise the Java compiler recognizes that the statements after the infinite while loop are unreachable (such as lines 28, and 29 if the inner while-loop were infinite) and throws a respective compiler error. Again, we need to bypass the compiler's unreachable statement error by using an ambiguous-value variable instead of a definitive-value literal. We are not definitively sure as to why this works, but we hypothesize that since the compiler cannot know the actual value of the variable at compile time, it tricks it into behaving like any statement after an infinite-loop is reachable, allowing us to compile the output program.

9.3.9 Switch Statements

Some of the major design choices we would like to discuss are highlighted in the code snippet below:

```
 1  {
 2          // Switch statement
 3          String __golite__x = "";
 4          String __golite__tmp_0 = "Hello";
 5          __golite__x = __golite__tmp_0;
 6
 7          boolean continue_loop_0 = true;
 8          Loop_0:
 9          while (continue_loop_0) {
10                  boolean switch_tag_0 = true;
11                  if (false) {}
12                  else if (switch_tag_0 == (__golite__x.compareTo("hello")
        == 0))
13                  {
14                          if (true) {break Loop_0;}
15                  }
16                  else
17                  {
18                          // Default case
19                  }
20                  continue_loop_0 = false;
21          }
22  }
```

Similarly to for-loops, we create a new scope for the switch-statement (lines 1 and 22), such that initialization statements are contained only within the switch-statement's scope (lines 3, 4, 5). We introduce the same outermost while-loop structure as for-loops (lines 7-9, 20-21), so that we can use labelled break statements (line 14). We create the variable for the switch tag (line 10), so that it can be used for switch case expressions (line 12). The non-default switch case statements are generated as else-if-statements and default switch cases are generated as else-statements. One thing to note is that at line 11, we generate an if-statement that never occurs, because every non-default switch case is generated as an else-if-statement. For some odd reason, when traversing the linked-list of switch cases, the conditional expression for determining the tail node (next == NULL) never passes.

9.4 Expressions

9.4.1 Identifiers and Literals

Generating identifiers and literals is straight forward, we simply generate the value of the respective identifier or literal. Identifiers are prepended with the `__golite__` prefix so that input identifiers don't conflict with reserved keywords or other identifiers that we generate to help the output program behave like the input one. Additionally, rune literals are output as int values.

9.4.2 Binary Operations

Binary expressions have certain edge cases that we'd like to discuss. For generating array binary operations, we made use of Java's `Arrays` API for its equality operations. Most Java binary operators map one-to-one to GoLang binary operators, the only one we needed special handling was for the Bit Clear operator. There isn't any reserved Bit Clear operator in Java, however the operator is equivalent to performing bitwise AND NOT, hence we generate those two operators instead `& ~`.

9.4.3 Array Access

If the array access is performed on a slice, since slices are custom-made classes, we use the slice class's appropriate getter function. Whereas for arrays, since they are generated as regular Java arrays, we simply perform a square-bracket array access.

9.4.4 Builtins

For appending to slices, we output the Slice class's `append()` method. There are multiple edge cases for the `len` builtin. For a slice type, we output the `len` field access, for an array, we output the `length` field access, and for strings we output Java's String API `length()` method. As for the `cap` builtin, we simply either output the Slice class's `cap` field access or `length` field access for arrays. This highlights one of the reasons why we chose to use Java, using Java's builtin length methods as well as being able to implement methods for the Slice class helped us save on time.

9.4.5 Function Calls

The arguments in the function call have to be generated in the same order as their declared input parameters. This was tricky because we needed to know whether to append a comma or closed parentheses for every generated argument while calling the generation function recursively on the linked-list of arguments. So we kept track of the number of currently generated arguments and the number of parameters the function expects. When the number of generated arguments is one less than the number of parameters, a closed parentheses is appended, otherwise a comma is.

9.4.6 Field Access

Since structs are simply newly defined classes in Java, we simply use dot notation just like in GoLang to get a certain field from a struct type variable.

9.4.7 Type Casting

We used a helper class in Java to implement casting. This is because there were some complexities in casting an int to a String or a char to a String, and we could not simply prepend the `(type)`. Code snippet:

```java
public static String castToString(int i) {
    Character c = (char) i;
    return c.toString();
}
```

For simplicity, we took advantage of Java's method overloading, so that we could call the same cast helper function regardless of input type.

9.5 Init and Main Functions

In Golite, the `init` and `main` functions have special behaviour. When we came across the main function, we prepended the function name with `__golite__` to avoid a keyword naming conflict.

Init functions have no parameters or return types and are executed before the main function in the top-down order in which they are defined. In Java, it is impossible for multiple functions to have the same name and same parameters. Our solution was to have a counter as we traversed init functions and assign each function a name in the form of `__golite__init_0`, `__golite__init_1`, etc.

Inside our Java main method, we called each init function in order, then the Golite main function.

10 Conclusion

Given the challenges of working on Milestone 2, Milestone 4, and the final report with having a group member dropping the course, as well as the disruption of COVID-19, we are very happy with the way our compiler turned out!

Online Collaboration: Our familiarity with GitHub and online tools for collaboration greatly contributed to the success of our project. Excluding Milestone 1, the entire project was done remotely. Although we were apprehensive about our ability to work productively online, it was an easy transition once we figured out how to split up tasks and dependencies to work in parallel. We used GitHub issues, branching, and Dropbox Paper to keep track on each other's progress. We made use of pull requests as a way to review each other's code and features. During these reviews, we would suggest better ways to write certain functions, suggest the addition of more documentation and tests, and clarify any uncertainty or comments about certain code snippets. During the typechecking phase, when we would run into bugs, we would screen share and walk through code together. We are proud to say that we didn't ever have a single merge conflict. :)

AST: Our AST really enabled making the later stages of the compiler more manageable. The overall structure was clear and logical, and it was easy to make annotations to the AST later during typecheck and codegen where necessary.

Testing: We used GitHub's *Actions* feature for CI/CD to execute `./test.sh` and `./test-solutions.sh` and report any errors for each pull request. This extra step ensured that even if we forgot to test a small change, it would be caught by our CI. It ensured that we were all aware of any failing tests without needing to explicitly communicate it to one another or if the newly developed features were passing all our tests and safe to merge. We found this reassuring and was useful during code reviews to discuss possible solutions for any failing tests.

Fixing Previous Milestones: Prior to beginning each milestone, we spent quite a lot of time fixing every test from the previous milestone. This really paid off for us in terms of being able to catch bugs early and not have technical debt built up.

What we would change? Originally, we chose to implement our project in C/flex/bison because of familiarity, given our prior assignments in Minilang. However, in retrospect, that decision was made mainly based not having to learn new tools for the scanner and parser. The scanner and parser were arguably the simplest phases of the compiler, especially given the Go specification. Later in the project, it became quite time consuming to debug segmentation faults in C. Given the opportunity, we might instead choose the Java/SableCC toolsuite because of the more useful error messages in Java.

11 CONTRIBUTIONS

- **Sophia:** *Milestone 1*: Scanner, AST. *Milestone 2*: Half of symbol, majority of typecheck. *Milestone 4*: Half of codegen, including: builtins, typecasting, untagged structs, slices, arrays, short declarations, multi-line variable declarations (swapping). *Final Project*: Half of report.
- **Matthew:** *Milestone 1*: Half of parser, weeding. *Milestone 2*: Half of symbol, some typecheck, report. *Milestone 4*: Half of codegen, including: expressions, for statements, switch statements, if statements, break statements, continue statements, function return statements. *Final Project*: Half of report.
- **Braedon:** *Milestone 1*: Half of parser, pretty print.

12 REFERENCES

Flex documentation for C-style commenting: https://www.cs.virginia.edu/~cr4bd/flex-manual/How-can-I-match-C_
002dstyle-comments_003f.html#How-can-I-match-C_002dstyle-comments_003f