COMP 520 Milestone 3:

Code Generation for GoLite

Members:
Ralph Bou Samra 260503614
Jonathan Lucuix-Andre 260632816
Yan Qing Zhang 260520930

Language Choice:

We chose to have our compiler generate Java code.

We chose a high-level language because all of our team members are more familiar with higher-level languages as opposed to byte code. Thus, using Java diminishes the learning curve needed to complete the milestone.

Furthermore, we chose Java as opposed to C because it allows us to accomplish string concatenation more easily. In fact, integers and strings can be concatenated using the "+" operator, without having to worry about explicit type casting, or about creating helper functions to aid in concatenation. Printing expressions is also easier, since it doesn't require using different printf codes such as %d, %f, etc.

In addition, slices can be easily converted to ArrayLists, and the *append()* function naturally translates to the *ArrayList.add()* function. This alleviates the use of *malloc* every time a new slice is declared, thereby simplifying our generated code.

What is more, nested structs can be easily defined as inner classes. Whenever a struct variable is declared, it will be instantiated using the *new* keyword on its corresponding generated class. Any nested structs will be recursively instantiated, and the resulting member fields will be accessed using the dot operator. The similarity between Java and GoLite's dot operator will make it easy to generate code for selectors on structs.

In terms of disadvantages, the generated Java code will likely be slower than its equivalent C code. However, given Java 8's optimizations, these inefficiencies shouldn't be overly prevalent.

We will also have to explicitly differentiate between array indexing and slice indexing. This is due to our representation of a slice as an ArrayList. As a consequence, slices will be indexed using the *ArrayList.get()* function, which will slightly complicate our code generation.

In addition, the code generator will need to wrap the code inside a class, since Java functions cannot exist outside classes. This is more complicated than in C, where functions can exist outside an encompassing class or struct.

Furthermore, the member fields and methods will be declared static in our generated Java code. This will ensure that all fields and functions will be accessible from the static main method. For this reason, the generated fields and function signatures will be somewhat lengthier than they would be in C.

Example Programs:

Var_decl.go:

Construct: *Variable Declarations*
Use Cases:
This program attempts to cover every aspect of variable declarations. This includes:

1. Checking single variable declarations of all base types (string, rune, int, float64, bool)
2. Checking single variable declarations with inferred types
3. Checking multiple variable declarations of all base types
4. Checking multiple variable declarations with a multiple inferred types (i.e. var y1, y2 = ‘a’, 6)
5. Checking distributed variable declarations covering all of 1-4
6. Checking variable declarations of arrays and slices
7. Checking short declarations

Type_decl.go:

Construct: *Type Declarations*
Use Cases:
This program attempts to cover every aspect of type declarations. This includes:

1. Aliasing of all base types
2. Aliasing of aliases
3. Struct declarations with inner structs
4. Array struct declarations
5. Distributed type declarations with structs and aliasing

Struct.go:

Construct: *Structs*
Use Cases:
This program attempts to cover every aspect of structs. This includes:

1. Struct declarations with inner structs
2. Variable declarations of struct
3. Struct assignments (including inner struct assignments)

Corner case to be investigated: array struct assignments

Binary_exp.go:

Construct: *Binary expressions*
Use Cases:
This program attempts to cover every aspect of binary expressions. This includes:

1. Simple binary operations: “||”, “&&”
2. Relational operations with binary operands: “==”, “!=”, “<”, “<=”, “>”, “>=”

3. Addition operations with binary operands: “+”, “-”, “|”, “^”
4. Multiplication operations with binary operands: “*”, “/”, “%”, “<<”, “>>”, “&”, “&^”
5. Same level operator precedence: Multiple level operators within an expression for all precedence levels(for e.g: a*b+2)
6. Different level operator precedence:
    a. Precedence level 5 with level 4
    b. Precedence level 3 with level 1
    c. Precedence level 3 with level 2
    d. Precedence level 2 with level 1

Precedence defined by GoLite:

Precedence	Operator
5	* / % << >> & &^
4	+ - | ^
3	== != < <= > >=
2	&&
1	||

Unary_exp.go:
Construct: *Unary expressions*
Use Cases:
This program attempts to cover every aspect of unary expressions. This includes:

1. Unary operators with single operands: +, -, !, ^
2. Multiple unary operators(!, ^) on single operands

Current Implementation:

For milestone 3, our code generator generates a *Main* class along with a top-level package declaration.

Additionally, it generates code for if-statements, for-statements, and switch-statements. The init statements were handled by creating a new anonymous code block to wrap these statements. The blocks create a new scope for init statements, allowing the generated code to imitate GoLite’s scoping rules.

In addition, the switch-statements were converted to if-statements. This allows the code generator to handle switch-statements with empty expressions. In fact, comparisons like “x < 0” are not feasible in Java’s switch-statements. Consequently, a different construct like an if-statement was required to handle this case.

Also, the default statement is converted into an else-statement, and is always printed last. If a break statement exists inside a switch-statement, any subsequent statements are discarded.

This is implemented to work around Java's lack of support for the "break" keyword inside if-statements.

Furthermore, simple binary expressions involving numeric types were implemented for this milestone. This includes code generation for the binary operators +, -, /, *, & and ^.

Additionally, our current implementation generates code for return, break and continue statements. The translation from GoLite to Java was straight-forward, as the syntax for these statements is almost identical in both languages.

Finally, our code generator supports print and println statements, which are converted to *System.out.print/println()* calls. An empty string is printed before every expression, which ensures that Java's type coercion system correctly casts each expression into a string.