Milestone 3

Team Members

- Michael Sukkarieh
- Justin Tremblay
- Nicholas Paun

Target Language Decision

We have chosen C as the target language for our compiler. The main reason for this decision is that everyone on the team is familiar with C - reducing the overhead caused by learning a new language. Our main hesitation with targeting C initially was due to the need to manage memory (probably by developing a custom garbage collector), but the professor has said that managing memory is not necessary for this project.

Go Semantics

In this section we will describe the semantics of three core areas of Golang: short variable declarations, loops, and assign statements.

Short Variable Declarations

The formal grammar for short variable declarations, according to the golang spec, is[1]:

```
ShortVarDecl = IdentifierList ":=" ExpressionList .
```

Intuitively, this statement is similar to a regular variable declaration in the sense that we are allowed to define, potentially multiple, variables. However, there are some subtle semantics involved with this statement that we need to keep in mind. The main difference is the ability to redeclare variables.

Variable Redeclaration

One major difference between the short variable declaration statement and the regular variable declaration statement in Go is that we are allowed to redeclare variables which are in our current scope - if certain conditions are met. These conditions are:

1. The short variable declaration contains multiple identifiers on the left hand side
2. At least one identifier on the left hand side is not defined in the current scope (i.e. it is a new variable declaration). Note that blank identifiers do not count as a new variable declaration

Loops

The formal grammar for for statements, according to the golang spec, is [2]:

```
ForStmt = "for" [ Condition | ForClause | RangeClause ] Block .
```

A for statement in Go works just like it does in most other languages, it allows us to iterate over the same block of code zero or more times. What makes Go unique however, is that there are three ways that this iteration can be controlled: by a single condition, or a “for” clause.

Single Condition

```
Condition = Expression .
```

The single condition clause expects an expression which evaluates to a **bool** type. The for loop continues iterating until this expression is false. This expression can also be empty, which allows us to define an infinite loop.

For Clause

```
ForClause = [ InitStmt ] ";" [ Condition ] ";" [ PostStmt ] .
InitStmt = SimpleStmt .
PostStmt = SimpleStmt .
```

The for clause grammar is similar to the for loop syntax in C-like languages. The for loop starts by executing the initial statement, continues looping while the condition statement is true, and executes the post statement at the end of every iteration. One notable difference between Golite’s for loop semantics is that the init and post statement are *SimpleStmt*’s - which includes short and regular variable declarations (unlike C,C++, which only allow expressions).

Assignment

```
Assignment = ExpressionList assign_op ExpressionList .

assign_op = [ add_op | mul_op ] "=" .
```

An assignment statement allows us to set the value of several different variables at once. However, there are some semantic subtleties that should be mentioned, including: assign operators, blank identifiers, addressability, and assignability.

Assign Operators

As described in the grammar above, assignment statements are allowed to have a variety of different operators (*assign_op*). The semantic meaning for these is very similar to that in other languages. An *assign_op* can optionally have an operator *op* (ex. +, -, <<, etc) prepend the '=' symbol. As a result, an assign statement such as `x += 5` is semantically equivalent to `x = x + 5`.

Blank Identifiers

Assignment statements are allowed to have blank identifiers on the left hand side, as long as *assign_op* is the ordinary '=' operator. Semantically, blank identifiers in assignment statements allow us to ignore values which are present on the right hand side. This is particularly useful when dealing with functions which can return multiple values (which Golite doesn't support), where only certain values returned by the function are of interest.

Addressability

All expressions on the left hand side must be lvalue's (i.e. they are addressable). This includes[3]:

- Variables (non-constant)
- Slice indexing
- Array indexing of an addressable array
- Field selection of an addressable struct

The result of a function call not assignable, except in one case: when we index a slice that is returned. This is because slices are passed by reference whereas anything else, such as arrays and structs, is passed by copy, and therefore not assignable.

Assignability

In order for the assignment statement to typecheck, the type of the lhs expression must be equal to the type of the rhs expression without resolving. This means even if both types resolve to the same underlying type, they are still not assignable.

Target Language Strategies

Short Variable Declarations

The difficulty of implementing Go's short variable declaration in C comes from the redeclaration semantics of the statement and blank identifiers, both of which are not supported natively in C. As a result, we will logically divide each declaration in the short variable declaration statement into three categories: newly defined variables, redeclarations, and blank identifiers. As a result, if we are able to generate each of these different types of declarations, any short variable

declaration can simple be decomposed into the individual declarations and they can be generated one after the other (based on the declaration's class).

Newly Defined Variables

Handling newly defined variables doesn't require any special tricks. We can simply convert this type of declaration into an ordinary variable declaration in C. Since we have already ran the typechecker before the code generation, each expression will be annotated with type information, allowing us to easily generate the statically typed variable declaration statement in C.

Redeclarations

Semantically, redeclarations are really just assignments. We are already guaranteed that the redeclaration is valid (i.e. the variable is already defined in the scope and the type being assigned to it matches the declared type of the variable) by the typechecker. As a result, we can simply convert a redeclaration into a basic assignment in C.

Blank Identifiers

At first glance, one may think that we can simply ignore declarations involving blank identifiers all together. However, this is not true, since the expression call on the right hand side may be a function call with a side effect (ex. printing). As result, we will implement this behaviour in C by simply translating the declaration to be just the expression on the left hand side (ignoring it's value).

Loops

Most of the semantics for for loops in Golite map directly to for loops in C. However, there is a significant different: initialization and post statements may have several target statements. An example of a for loop in Golite which is not easily translated to C is provided below:

```
for x,y := 0,20; x < y; x,y = y+1, x-1 {
      if (x == 5) {
            continue
      }
}
```

One thing to note in the above example is that we are assigning values to multiple variables in the post statement, something which is not supported by C.

As a result, we will choose to generate Golite for loops in C by using a while loop and taking advantage of C's goto functionality. The condition of the while loop will be the same as the condition in the source language, no extra generation tricks needed to be used there. As for the initialization statements, since we may be declaring new variables which can only be used within the loop, we will create a new scope just for the while loop and place the initialization

statement above the while loop but have it be contained within its scope. Finally, we will implement the post statement by placing it at the end of the while loop with it's own unique label. The label is added in order to support proper continue functionality. Continue statements in the source language are converted into goto's to the post statement at the end of the while loop. No special care is needed to support break statements within the loop.

Assignment

The easiest (but incorrect) way of translating assignments in Golite to C would be to simply decompose the assignment list into multiple individual assignments. However, this does reflect Golite's assignment semantics, which call for simultaneous assignment into all lvalues. As a result, we must first save the current values of the variables into temporary variables, and we can decompose the assignment list into multiple individual assignments - making sure that we use the temporary variable in place of the variable it is corresponding to.

Blank Identifiers

In the case of blank identifiers, we cannot simply ignore them in the translation since the expression being assigned to them may have a side effect. As a result, we can simply translate an assignment into a blank identifier as an expression statement (i.e. we simply generate the expression by itself and ignore the value).

Code Generation Patterns

Variable Declaration

```
// Golite
var LHS1, LHS2, _ = E1, E2, E3

// C
// note: here typeofEX is the type as annotated in the ast node for the
// rhs expressions
typeofE1 LHS1 = E1;
tyepofE2 LHS2 = E2;
E3;
```

Short Variable Declaration

```
// Golite
// NEW denotes a variable which is not in the current scope
// OLD denotes a variable which is defined in the current scope
NEW1, _, NEW2, OLD1 := E1, E2, E3, E4
```

```
// C
typeofE1 NEW1 = E1;
E2;
typeofE3 NEW2 = E3;
OLD1 = E4;
```

Type Handling/Declaration

User-Defined Type Declaration

Golite allows developers to define their own user defined types which share the same underlying type as another type but are semantically different. However, once we have ran the typechecker and ensured that the source program type checks correctly, user defined types are no longer necessary for us. As a result, we simply ignore all basic type declarations and use the resolved type when dealing with all user defined types or aliases in our generated code.

Array

In Golite, array's have copy semantics when used in assign statements. To implement this in C, we will typedef a struct for each array type (size and type). Since structs are deep copied in C, wrapping our array's with a struct allows us to automatically take care of array copy semantics. The code block below shows examples of this in use.

```
// Golite
var x [10]int
var y [2]float64
var z [5]int

//C

// note: this function will only work with int/float/rune arrays
// char* must be initialized to '\0'
void __golite__init_array(void* arr, size_t len, size_t el_size) {
    for(int i = 0; i < len; i++) {
      *arr = 0;
      arr += el_size;
    }
}

typedef struct {
      int data[10];
} arr_int_10;

typedef struct {
```

```
      float data[2];
} arr_float_2

typedef struct {
      int data[5];
} arr_int_5;

arr_int_10 x; __golite__init_array(&x, 10, sizeof(int));
arr_float_2 y; __golite__init_array(&y, 2, sizeof(float));
arr_int_5 z; __golite__init_array(&z, 5, sizeof(int));
```

Slices

To implement slices, we will create a struct which maintains length/capacity information as well as a pointer to the underlying data array. This works very nicely with Golite's copy semantics, since the length/capacity values and the pointer to the underlying data is copied. As a result, copying a slice (in this case, the struct representation of the slice) will result in both copies of the slice pointing to the same underlying data (just as is done in Golite). The struct which will be generated is displayed below:

```
typedef struct {

      // length: amount of elements currently in data array
      // capacity: max size of data array
      // element_size: byte size of underlying element in data array
      size_t length, capacity, el_size;
      void* data;
} __golite__slice;

#define __golite__INIT_INT_SLICE { length = 0; capacity = 0; el_size =
sizeof(int); data = NULL; }

// Golite
var x []int

// C
__golite__slice* x = malloc(sizeof(slice)); *x = __golite__INIT_INT_SLICE
```

Struct

Golite allows developers to define their own struct types. We can easily translate this to C since C also allows for user defined struct types. Note that struct fields with blank identifiers are completely ignored when generated the target code.

```
// Golite
type person struct (
      name string
      age int
      _ float64
)

// C
typedef struct {
      string name;
      int age;
} person;
```

Assignment

```
// Golite
LHS1, _, LHS2 = E1, E2, E3

// C
LHS1 = E1;
E2;
LHS2 = E3;
```

If Statements

```
// Golite
if S; E {
      B
} else if S2, E2 {
      B2
} else {
      B3
}

// C
{
      S;
      if(E) {
          B
      }
      else {
          S2;
```

```
          if(E2) {
              B2
          }
          else {
              B3
          }
     }
}
```

Switch Statements

```
// Golite
switch S; E {
      case E_list: S_list
      case E2_list: S2_list
      default: S3_list
}



// C
{
    S;
    typeofE tmp_var = E;
    if(E == E01 || E == E02 .. || E == E0n) {
        S_list
    }
    else if (E == E11 || E == E12 .. || E ==  E1n) {
        S2_list
    }
    else {
        S3_list
    }
}
```

For Loop

```
// Golite
for IS; C; PS {
    B
}
```

```
// C
{
    IS;
    while(C) {
        B;

    post_statement:
        PS;
    }
}
```

Printing

```
// Golite
// Where E_list = E1, E2, E3.. En
print(E_list)
println(E_list)

// C
// here we show the printing where for a single expression but it is
// trivial to map this into several different expressions by extending the
// string/args given to printf

// rt(E) == int || rune
printf("%d", E);
printf("%d\n", E);

// rt(E) == string
printf("%s", E);
printf("%s\n", E);

// rt(E) == bool
printf("%s", E ? "true" : "false");
printf("%s\n", E ? "true" : "false");

// rt(E) == float64
printf("%.6e", E);
printf("%.6e\n", E);
```

Functions

```
// Golite
// where P_list = P1, P2, P3 ... Pn
func ID(P_list) RTYPE {
      B
}

// C
// note: if RTYPE is empty it is replaced with void
// note: parameters with blank identifiers in P_list are given a random
// unique identifier
RTYPE ID(typeofP1 P1, typeofP2, P2, .. typeofPn Pn) {
      B
}
```

Code Generation Status

Since we did not complete our type checker for milestone 2, a large amount of our efforts were spent completing the type checker. As a result, we did not have a lot of time to work on codegen. The codegen features that we have so far:

- top level variable declarations
- functions (not including parameters)
- literal types
- expr and incdec expressions

References

[1]: https://golang.org/ref/spec#Short_variable_declarations
[2]: https://golang.org/ref/spec#For_statements
[3]: https://www.cs.mcgill.ca/~cs520/2019/project/Milestone2_Specifications.pdf