Group 09:
Nathan Sandum 260716168
Jiawen Wang 260683456
Grégoire Moreau 260874685

COMP 520 WINTER 2019: MILESTONE 3

Lecturer Alexander Krolik

After some discussion, we have decided that our compiler will be generating C code. We wanted something relatively high-level, with the memory allocation and other advantages of some of the lower-level languages. None of us were comfortable enough in assembly to attempt generating LLVM. In addition, similar to our decision to write the compiler in C, we decided that it was the only high-level language that all of us had strong experience in.

Some advantages and disadvantages of C as a target language can be summarized as follows:

Advantages	**Disadvantages**
High-level with lower level capabilities	Finnicky special cases
Portability	Naming conflicts (e.g. restrict, …)
Relative speed	Untagged struct parameters disallowed
Pass-by-value	
Return-by-value	
Presence of structs	
Similar increment/decrement functionality as Go	
Statically-typed for fewer type errors	

Group 09:
Nathan Sandum 260716168
Jiawen Wang 260683456
Grégoire Moreau 260874685

4. Types in GoLite and C

GoLite varies between strongly typed, in the standard declarations, and weakly typed, as in short declarations, but generally has the characteristics of strongly typed. This fits nicely for C, being strongly typed as well; the rare weakly typed examples that could be described in GoLite will not be complicated, as they should be dealt with in the type checking phase of compilation, where their types will become indicated..

Bounds checking, likely among the simplest to check, will be done by a simple function that will encapsulate any array access, check the bounds, and if it succeeds, the function itself will return a void pointer to the data, which will be casted and dereferenced at the function call. Of course if it fails, it will write to stderr and exit. This function would be very similar between arrays and slices.

Checking equality will be slightly more complicated. My original thought was to take any struct and check that each has the same size (which should be a given since they've passed type checking) and from there, to bitwise compare equality. Of course this has the concern that an array would not necessarily be stored contiguously within the struct the way the rest of the data is, but instead as a pointer, so instead a recursive approach would be needed. A function will iterate through the fields and use a switch/case to narrow exactly which checks to do on the type, and so check the equality of every field. Arrays are similar but simpler, and iterate over every element, and then use a similar switch case to narrow the checks, and so check equality on every element. Other types will be represented without aliasing. Since they've all passed typechecking, we know that no "nums" will be compared with any ints, so both can be represented as ints. All base comparisons would be done the usual way. Though strings will be using strcmp. String a compared to string b is analogous to strcmp(a,b) compared to zero, including ordering comparisons. Slices are not comparable.

Tagging will be very simple. Before any declaration requiring it, simple iterate through the parameters, see that if a declaration needs a typedef, write that to the C file before the declaration is written.

Implementation is the final discussion required. As mentioned above, all basic types will be represented as the type they fully resolve to, i.e. nums as ints. We know the code passes type checking, so no problems will arise. Arrays and Slices will be represented using the same structs. They will contain pointers, which will be filled with malloc, capacities and lengths. To match the behavior of both, whenever an array is on the right hand side of an assignment, copy_array will be used instead, which will return a struct defined with the same capacity/length, but the pointers will be different, and the new one will be assigned a new pointer with newly malloc-ed data will will have data copied from the original pointer's data. copy_slice in the same context, however, will behave similarly, except that the same all of the length, capacity AND pointer will be identical. The pointer will only be changed when the capacity of a slice needs to be modified, and so a new pointer is created. This should almost certainly contain all the nuance of both arrays and slices in GoLite.

Group 09:
Nathan Sandum 260716168
Jiawen Wang 260683456
Grégoire Moreau 260874685

Structs, simply containers of base types, arrays, and slices (and of course other structs), will be implemented exactly as C structs, with the fields implemented as explained above. Though, since arrays and slices contain the nuances they do, it won't be possible to copy structs the way C does by default. Instead, a helper function will be needed to copy structs, iterating over the fields, and, where encountering arrays or slices, use the respective copy function, using default copying otherwise.

5. Assign statements: multiple vs. single, copying

We will add a prefix to all variable identifiers from the GoLite program into our generated C code to avoid conflicts with reserved keywords in C.

GoLite assignments contain a list of expressions on the left-hand side and a list of expressions on the right-hand side. The weeding phase will make sure that both lists are the same size, while the typechecking phase will make sure that all expressions on the LHS are valid lvalues. During the assignment, the values from the RHS are copied entirely to the LHS expressions, even for composite types. In a GoLite assignment, the order of the values in both lists should not matter : a, b = b , a should be equivalent to b, a = a, b. This means that the value assigned to a variable in an assignment should always have the same value as if it was computed before the assignment statement. The blank identifier can also be used as the target for an assignment, but they do not introduce bindings. GoLite also allows op-assignments using the following operators : +=, &=, -=, |=, *=, ^=, /=, <<=, >>=, %= and &^=.

In C, assignments can only assign one expression to one valid lvalue. It also doesn't have anything corresponding to the blank identifier. It has all the op-assignment operators described above except for "&^=" which corresponds to bit-clear. Finally, in C, variables that were declared with an array type are pointers, they are simply a reference to the memory.

There are thus a few tricky cases for which we have to be careful:

1. An assignment containing multiple expressions on both sides in GoLite (a, b = c, d) can't simply be translated to a list of C assignments that decomposes the original assignment (a=c; b = d;) because expression d might contain identifier a, whose values might have changed in the first assignment a=c;
2. In the generated code, nothing should be assigned to the blank identifier (since it doesn't exist in C) but the RHS expression that is assigned in the GoLite program should still be evaluated as it might contain a function call.
3. There might be an op-assignment using the bit-clear operator in the GoLite program, which can't be simply copied in C.
4. We can't copy composite types by using simply their identifiers in C, as arrays would simply be passed by reference, while structures would be copied but could contain underlying arrays for which we would also only copy by reference.

To solve the first problem, our implementation for the code generation of assignments will use temporary variables. For an assignment a, b = c, d in GoLite, we will simply create 2 temporary variables to store c and d, then assign the first temporary variable to a and the second to d. This means that each expressions will be evaluated the same way it would have been before the first

assignment of the list. We don't need to use temporary variables for op-assignments as they may only have one expression on each size in GoLite.
The second problem is partly solved with temporary variables as well. We will have created a temporary variable for each RHS expression, so each of them will be evaluated and the function calls that they may contain will be executed. When we see that the expression to which we have to copy the value of the temp variable is the blank identifier, we can simply skip that expression.
For the third problem, we can simply rewrite" a &^=b" as "a &= ~(b)".
Finally, for the fourth problem described above, we will simply use the copy operations (copy_array, copy_slice, copy_struct) everytime a composite type is used as a RHS expression.

11. Functions in GoLite and C

In GoLite, functions are pass-by-value and return-by-value. They are considered top statements and though Go allows for nested functions, GoLite does not. Each function must have either no input parameters or a list of parameters, and either zero or one return type. In Go grammar, a parameter list consists of a list of "parameter declarations", with each declaration being a list of identifiers followed by a valid type. Out of list of input parameters, the names of each must be unique (excluding the blank identifier), and each identifier represents one parameter.

There are two special functions in Go. The init function(s) and main functions do not have any input nor any return types. There can be zero or any number of init functions, and they are invoked based on the order in which the scanner sees them. There must be exactly one main function, which is invoked after all the init functions have been executed.

In C, there is a mandatory main function that acts as the entry point of all programs. From there, other functions (user-defined or library functions) may be invoked, and nested function declarations are disallowed, similar to Go. Like GoLite, it is able to accept zero or more parameters as input to a function. In addition, forward function declarations are allowed.

In terms of implementing the code generation, we will use a similar tactic as the reference compiler. We will gather each init function, keeping track of their order. We then rename both the init and main functions. In the main program of the generated C code, we simulate the invoking of these special functions by calling the different init functions in order (__golite__init<curr_init_num>), followed by the renamed main program.

User-defined functions will be generated in a similar way as they were in pretty printing. That is, we'll first generate the return type. We'll use the function cgTYPE in codegen.c, which takes in a TYPE * t, and outputs the C function's return type to the generated C file. This may be a base type or a tagged type's typedef name. If there is no return type, we output void instead. The function names

will follow, with an added prefix to avoid any naming conflicts. Parameters will be handled the same way, listing each parameter name, with a prefix, preceeded by a type determined by cgTYPE. We call cgSTMT on the function body, and from there basically pretty print the return statement, with the identifier names and their prefixes.

Our Code Generation so far

We have set up the basic code in main.c, the Makefile, codegen.c and codegen.h to begin implementing code generation into C code. In main.c, the new mode generates a new .c file from a given .go file, and places it in the same folder as the Go source file (i.e. programs/3-semantics+codegen/*). It then parse, weeds, and typechecks the program, and passes on the root and the name of the output .c file to codegen.c.

From the standard AST traversal, now typechecked, we implemented the declaration and invoking of the special functions described above. We have also added the necessary prefixes for function names and identifiers so far.

We have also begun the code generation for simple expressions like generating prefixed identifiers, literals, unary expressions, and binary expressions (with parentheses). We have also started cgTYPE, which takes a TYPE * t type in GoLite and translates it to a corresponding type in C.

Different statement kinds have also been handled, such as the empty, block, expression, if/else, and return statements.

We also handled the op-assign statements, including the handling of the bit-clear operator.

In addition, in codegen.h, a function called cg_general_code_init has been added which writes the helper code we'll need to the output C file, allowing general functionality such as copying slices or arrays in the manner that they require.