Milestone 3 - Group 06

Semantics & codegen status report

Our group has decided to choose C as our target language.

There are many reasons for this choice of target language. Some straightforward ones are the familiarity of C of our whole team, the fact that we are also using it as the implementation language (thus “getting us into the swing of things”), the “simplicity” of C, the syntactical resemblance of Go and C (most important reason) and the related fact that the Go developers were motivated by “their shared dislike of C++”.

The main reason of Go and C being syntactically similar should be obvious enough in and of itself, sharing many statement and expression structures, albeit differing a lot in other ones, but not as much in GoLite. This facilitates translating from GoLite to C as many statements or expressions can be close to a 1:1 translation, or at the very least closer than probably many other target languages we could have chosen.

GoLite is also a statically typed and compiled language just like C, and sits at a similar “level” than C (though Go would probably be considered more high-level, or at least more powerful, depending how you see it). They are also both imperative languages, with Go being more of a mix of paradigms. However, the simpler GoLite language is mainly a subset of the imperative side of Go. Picking C, a very low-level language (by today’s standards), we also avoid the risk of being unable to directly translate some language concepts (e.g. what if GoLite had pointers and we chose a language without pointer support).

As mentioned above, they are both statically typed, and this is a very important detail as it defines a large part of the AST and the language design and deciding the typing of a language is a core decision to make that affects it in its entirety. Picking a target language without static typing may or may not be harder, but, in our opinion, would certainly be much more “awkward”. There would also be a risk, as if we were to translate the program of a statically typed language to a dynamically typed one directly without any care, the behaviour could be very different even if not obvious at first. Many things that could be allowed in a dynamically typed language would not even compile in a statically typed one. Implementing type checking in a dynamically typed language equivalent to a statically typed one, at least in most ones we know, would require special tricks which may or may not be an exact adaptation (e.g. Python type annotations, TypeScript to JavaScript). In any case, we consider it more straightforward to pick a target language with similar typing for a more direct translation and not having to worry about statically typing the compiled code.

As such, GoLite and C being low-level, statically typed and syntactically similar make for very compelling reasons to choose C as the target language, along with our familiarity with the C language itself.

The three areas for which we will describe semantics for will be Identifiers & Keywords, Loops, and Functions.

Identifiers & Keywords:

No matter what target language we pick, there will be naming conflicts due to how flexible Go is with identifier names. For example, `var extern int` should be valid, but if we translate to C naively, then our target program will encounter a compiler error, since extern is a reserved keyword in C but not in Go. What we will do is append a fixed suffix to all identifiers. This way, there will be no possibility of a naming conflict.

Codegenning the blank identifier is also not as trivial as we originally thought. Any statement that declares or assigns a variable with the blank identifier as the name can be tossed aside, since there is no way/it is unallowed to reference the declared variable anyways. However, if the expression on the right-hand side is an expression statement (a function call), then it would be wrong to ignore the whole line. Therefore, when we codegen, we will check the RHS of an assignment if the LHS is a blank identifier, and if the type of expression is a function call, then we will, on a separate line, execute the function call. The same will apply for assignments, declarations of multiple variables on the same line.

Finally, blank identifiers inside the parameters list for a function declaration should be valid, meaning that

```
func a(_ int, _ string, _ float64){ ... }
```

should be valid. In order to keep this valid in C, we'll use temporary variable names when codegenning with blank identifiers. This will be translated to something similar to:

```
void a(int _a1, char* _a2, float _a3){ ... }
```

Since this problem will only occur in function declarations, it'll be simple inside codegen to make sure these temporary variable names are unique.

Loops:

Aside from 3-part loops, for-loops are pretty simple to translate to C. However, because of what's allowed by Go inside the init and post statements, a 3-part loop is more difficult to codegen. In particular, if the post statement contains an assignment of multiple variables, then we cannot do a one-to-one mapping to C, since C only allows a single statement in a for-loop post statement.

To deal with this, we will decompose the post statement (or just move it, if it's a equivalent to a single statement in C) and place it at the end of the block for the loop. The loop itself becomes a while-loop on the condition of the original for-loop. To make sure that it's always reached (in case there is a continue statement earlier in the block, we'll add a goto statement upon seeing a

continue statement). As for the initial statement, we'll perform all initializations before the loop begins, and we'll enter a new scope so that no declarations in the init statement remain after the loop ends. Here is an example of the transformation:

```
Go:
for(stmt; condition; post){
      if(condition2) { continue; }
}
```

```
C:
{
    stmt //separated if needed into multiple lines
    while(condition){
          if(condition2){
                goto continue_label;
          }
          ...
          continue_label:
                post //decomposed into multiple statements if needed
    }

}
```

Functions:

Pass-by and return-by semantics are a bit strange for Go. Arrays and structs are pass-by reference and return-by value, but slices are return-by and pass-by references. A naive solution is to make object copies before passing them to a function if it is a struct or an array, and to pass the object itself if it's a slice (no extra work in C), but this won't necessarily work. If a struct contains a slice field, then any modifications to the slice field during the function call will also modify the struct's slice field.

We believe the correct approach is to carefully watch what kind of assignments are made inside a function. If a function makes assignments to its parameters, then check what kind of structure the parameter has, and if it is an array or struct, then to assign into a temporary variable, and if it is a slice, then assign into it directly.

Finally, we have to make sure that structs (or an array of structs, slice of structs) can actually be passed into a function. In C, the following is illegal, but it is legal in Go:

```
void a(x struct{ b int; }){ ... }
```

Because of this, all structs in the Go file must be typedef declared earlier in the file so that they can be legally passed in the C output file. Therefore, the following mapping will happen:

Go:

```
func a(x struct{ b int; } ) { ... }
```

C:

```
typedef struct _gostruct_1 _gostruct_1;

struct _gostruct_1{
      int b;
}

void a(_gostruct_1 x){ ... }
```

Status Report

Our current codegen.c file is mostly a copy of our pretty printer file because of their similar formats (so ignore most of the file if you decide to look at it!), but we will be changing things as we progress throughout the next milestone. However, we have implemented some of the simpler language features:

For loops are implemented: making sure that variable scoping is correct and that post statements are always executed regardless of a continue statement.

Variable renaming has been added (the codegen statements have not been done, but the renamings have been assigned). We added a field inside the SYMBOL struct called “codeName” which will be the renamed name of any declared variable. Each SymbolTable has a local counter corresponding to the postfix value, and this value will be assigned to the codeName field of any Symbol field inside the SymbolTable.