COMP 520 : Compiler Design

# Milestone 4 - Group 06
Final Report

Introduction:

Throughout this semester we worked on implementing a compiler for GoLite, a subset of Google's GoLang. In this report, we talk about our language and tool choices, contributions, design choices, and our journey as we worked on the various parts of the project; scanner, parser, AST, weeder, symbol table, and codegen.

Language & Tool Choices:

We chose C as our implementation language as everyone in our team was familiar with it and Flex/Bison are for C. The class also mainly uses C examples which helps.

There were some concerns about the limitations of C (being what it is) and we wanted to use a functional programming language initially (as they are easier to deal with when creating compilers) such as OCaml (which is based on ML "MetaLanguage" which was made for this purpose) but since we had all done our assignments in C and because of the "better" tooling we chose C.

It went well at first, but unfortunately the concerns ended up justified as we encountered problems with the parser. Lacking most features of functional and object-oriented languages, it was difficult coding the abstract syntax tree generator (no polymorphism, no classes, etc.) being limited with enumerations and simple structs. Since GoLite's program structure, like most larger languages, is more complex than ones like MiniLang (i.e. everything is a statement), it was much more difficult designing the grammar, the AST and the code. Deciding "what was what" and what was a "subset" of something else, how to pass information about this in C and be able to determine inside the code what to do (switching on the kind, etc.) was very difficult with the limitations of our implementation language. Flag parameters were needed everywhere, or at least it is what we should have done more.

In consequence, this greatly affected how we designed the grammar and the parser and gave us some headaches. One example is the toplevel declarations, and how to separate them between each other. Toplevel declarations are a special kind of declaration (only certain type, variable and function declarations) but are still “normal” declarations, except for function declarations which are only allowed at the toplevel.

The usual trick is to use kinds and pass them everywhere to be able to switch on them and call the correct node generator. Even then, with a more complex grammar it was not easy. For example, it would have been great with class types and inheritance along with polymorphism. Pattern matching would have also been of a great quality of life feature.

Making a model or draft on paper of our designs would have been helpful at first too. In fact, a lot of times we made mistakes because certain rules were allowed in GoLang but not GoLite. For example, we spent some time trying to support embedded struct fields (which GoLite does not handle) only to realize it wasn’t needed.

C structs are not nearly as flexible and convenient as classes in typical OOP languages. It would have been better to use C++, as we could have still used Flex and Bison, and since it is a language based on C, it would have been perfect to choose for all of us. In fact, C++ and C code can be used together quite conveniently, so we plan on maybe doing this in the future.
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.

## Scanner

Our scanner was done in almost the same way as in the first assignment for the course. However, to support GoLite's semicolon insertion and block comments, a little bit of extra work had to be done. Semicolon insertion was handled with inspiration from the examples in the public repo: the lexer kept track of the last token that is returned, and upon seeing a newline character, it would check the last token and if it is one of the tokens specified in Go's language specifications under semicolon insertion, then we would return a semicolon token. To handle block comments, we defined a new state to enter upon seeing a `/*` sequence, and anything seen before a `*/` character sequence would be "thrown away". If we reach EOF before `*/` then we would throw an error. Otherwise, we would exit the state upon seeing the end-of-block-comment sequence, and scan the rest of the input as usual.
In order to support the three types of integer representations (decimal, hexadecimal, octal) we had the three appropriate regular expressions. For true/false bool values, we imported `stdbool.h` and used that. Raw strings could be made distinct from interpreted strings since their opening and closing quote characters are different (`` ` `` vs `"`). One final trick we had to consider was any program that ended on a `}` but no newline at the end, then there will be no semicolon insertion (but the grammar must be able to accept it anyways). In order to deal with this, we added an action upon seeing `<<EOF>>` that checks the last token, and if it isn't a semicolon, to additionally return a semicolon before finally terminating with `yyterminate()`.

## Parser

A lot of our grammar was taken directly from Go's official language specification page, in addition to their precedence levels, where we made sure to add precedence directives inside the grammar when it came to tokens with multiple uses such as `*, &, -, +` . However, we could not take everything from their specification, and there were things that had to be changed.

GoLite enforces that there is exactly one package declaration in a program, which was easy to handle with the rule `program -> packageClause topLevelDecls` , where `program` was our start symbol. `topLevelDecls` was analogous to `stmt` from our assignments; `program` was just a list of top-level declarations.

Statements or declarations that resided in blocks (inside loops, functions, switch blocks, etc) were named differently since they are not allowed on the top level of the program. We called these blocks `statementList` and `blockStmt`. The reason we had both was because of GoLite's specific rule that states that function bodies must not be empty. So for function bodies only we used `blockStmt` and for other block bodies we used `statementList`. This way, the grammar allows empty bodies in if/else, for, and switch statement bodies.

We decided not to handle correct placement of break/continue statements in the parser. **`break`** statements should only be allowed in **`case`** blocks, and **`continue`** statements should only be allowed in a **`for`** loop block body. It is cumbersome to deal with this in the parser since the introduction of new nonterminals specifically for this purpose makes the parser a lot more verbose than it needs to be. Instead, we plan to handle these in next Milestone when we do symbol table creation and type-checking. Similarly with pre-statements and post-statements in a **`for`** loop, we do not check in the parser that that post-statement cannot be a short statement.

Finally, dealing with the blank identifier was a hassle. It's valid as a function name, but not in a function call, nor can it be used in the arguments list to a function call. Initially we had two separate tokens: one for regular identifiers and one specifically for the blank identifier. We tried to make the distinction and enforce rules in the grammar, but realized that introducing more rules to keep it strict caused us a lot more shift-reduce and reduce-reduce errors, so we decided that we will handle improper use of blank identifiers in a later phase. As a result, we changed the lexer to return blank identifiers as an identifier of value `_.` We've decided now to handle these in the second milestone.

<u>Weeding</u>

A lot of the weeding was done in the parser. To avoid shift-reduce conflicts we had no choice but to accept a larger language and weed out what was unwanted. Here is what was checked in the parser:

- Continue statements allowed only inside for-stmt-blocks

- Break statements allowed only inside for-stmt-blocks and switch-case blocks
- We accepted a larger language which allowed expressionlists on the LHS of a shortdecl, meaning the grammar initially accepts something like `obj.b, arr[12] := 3, "x"`, so upon a shortdecl, we weed the LHS and ensure that it is actually only identifiers that are being declared, with our well-named "`isThisExprlistAnIdentifierList(EXPRLIST* exprlist)`" function. If it passed correctly, we casted the exprlist into an identifier list with another function.
- For assignments, shortdecls, and regular variable declarations, we have functions that check if there are an equal amount of arguments on the LHS and RHS.
- Allowing expressions as the name of a function call, eg `(((a)))()` but disallowing something like `x[3]()`.

## AST

The root of our AST is the package declaration, since it is the only required part of the program. The rest of the tree is a similar (but long) extension of our assignment tree. In particular, we had to consider how to handle type definitions. Since users could define their own types, our TYPE node had four fields: identifier (char*), basetype (TYPE*), size (int), and a struct (FIELDDECL*, for the structure detail of the type that it is built on, such as array of arrays). The identifier would be the name of the type, and the basetype would be the type that this node is based on. In the case of "primitives" (int, float64,...) then baseType would be NULL. For example, an [3][5]int array would actually be a [3] array of basetype [5]int array, and the structure of the latter would be stored inside the struc field.

## Pretty Printer

For distributed variable and distributed type declarations, we decided that those are purely syntactic sugar. While we will accept those in the grammar, we won't make a distinction between that and a regular variable/type declaration when we pretty print. As such,

```
var{
	a int
	b int
}
```

will be printed as:

```
var a int
var b int
```

We believe this decision is fine because in the end the semantic action is the thing that matters, and semantically, the two are equivalent.

Scoping

Like in the assignment, we implemented a cactus stack of hashtables to implement the symbol table. At first, we thought it would be hard to differentiate between variables and types, but since GoLite forbids same-name declaration (`type a int`, then `var a a`), this became a bit easier to handle. We handle type-checking as a separate step from symbol table creation.

We have 6 kinds for our SYMBOL object, which tells us if the symbol is for a function, array, slice, struct, variable, or type. Where applicable, the information is stored inside the SYMBOL's val field. For example, upon a function declaration, the signature, function name, and return type is stored inside the symbol associated with it, and upon a function call, we can easily access the stored fields. This makes it easier to type-check in the later stages. This also ensures that, for example `var a b = 3` that `b` is actually associated with a Symbol of kind Type, and not of kind Variable.

Upon any declaration where appropriate (type, variable, function), a special call `getSymbolScope` (identical to getSymbol, but only searches in the current scope and not any parent scopes) to see if the variable has been declared already. If not, then they are added to the symbol table and stored inside a new corresponding SYMBOL* field in the node of the AST as well.

Upon any access or use of an identifier, it's checked first if it exists inside the current or any parent scope with a call to getSymbol. Since type checking is done separately, we do not compare or infer any types at this point.

We change scopes when we encounter a blockStmt. We create a new symbol table and point to the scope we're leaving from as it's parent. This happens when we see function declarations, for loops, if statements, or switch statements.

User-defined types and nested structures were tricky to implement.

To cover nested structures, we had a helper function called getType which took in a TYPE* struct and outputted in a pretty format its structure. For example,

```
struct{
            a int
            b int
}
```

would be stored as "struct{ a int, b int }", and

```
var a [3][4]float64
```

would be stored in the "kind" (aka type) field as "[3][4]float64". In the end, this makes type-checking a bit simpler, since we only need to perform a strcmp to make sure types match. In fact, this also allows us to easily check if a comparison between the following two structs is valid or not (this is valid):

```
var q struct{
      a int
      b bool
}
var p struct{
      a int
      b bool
}
```

Since there is no assignment compatibility (aka int + float operations being valid), then the way that the types are stored should be fine.

Functions have a function kind symbol associated with the symbol, but what about type-casting? Upon a type declaration, we should also add a symbol to the table with the same name, but as with a function-kind. In this way, once we declare `type a int`, we know that `a(3.0)` is a typecast from float64 to int.

## Type Checking

Typechecking was particularly difficult, not just because of the volume of things to take into account but also because it made us question some design decisions we made in the parser and tree parts. We had to work with what we had and realized some decisions could have been made differently (especially for complex data structures such as arrays, slices, and structs). While trying to implement array and struct comparison for in binary expression operations that requires two comparable argument types, we realized that the implementation of our expression node cannot support that. In fact, we stored all their different properties but provided no way to access them for typechecking. Because of that, we are currently unable to resolve the type of arrays, slices, and structs. A second issue of the same kind that we encountered is that we did not provide access to the return type of functions as well as the type of each struct field, i.e. we cannot typecheck function calls and struct field selection correctly. One discrepancy that we found in the specifications and the reference compiler is that the function name is an expression and we implemented it as an identifier. We know that now it is because the function name is allowed to be parenthesized. To allow this, we accept expressions but verify that they are essentially identifiers, with a weeding check (make sure that `(((a))))() = a()` ). Each of the above mentioned implementation issues are described in details in the comments of type_checker.c. We also found out that some parts of our tree was not well linked together or did not make as much sense as we initially thought (at least for typechecking) but we still work through it. There were also surprises about things that were allowed or specified further in the specifications that we did not completely take into account for in our design. Particularly how expansively expressions were allowed in the GoLite code structure.

## Invalid programs

The descriptions of the type-checking or symbol-table-creation problems of each invalid program we wrote is detailed briefly at the top of each file as a comment. We essentially tested that functions returned the proper types, that there are no duplicate var declarations, that short declarations override previous declarations iff there is at least one variable on the LHS that is undeclared, etc. We tested that user defined types behaved properly, and we tested scope rules for if/else statements, like in the scope3.go file.

## Codegen

### 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
    }

}
```

We also use labels and __*golite*_ affixes as explained above for the labels and pass it through the codegen functions to remember the current scope's label.

Functions:

Pass-by and return-by semantics are a bit strange for Go. Arrays and structs are pass-by value 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 believed 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. However, upon realizing how difficult this is, we will just pass everything by reference.

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){ ... }
```

For the special functions *main*, *init*, we used the same technique but also made separate functions prepended with __*golite*__ that would call the C *main* properly. Our codegen function for generating functions checks for creation of *main* and *init* and makes sure to do this instead of normal function generation.

Types:

For types, we thought of mainly two ways to map from Go to C. Originally we planned on typedef'ing everything, similarly to all Go type declarations (named types, structs) since we already do structs this way as shown previously in Functions. But this would make scoping very complicated, as in C typdefs are in the global scope (toplevel) so instead it was suggested to simply treat variables and values as their primitive type in C. So our tree would store both the "true" type and the "primitive" type. The typechecker would guard against type errors so the codegen could freely use the primitive types without worrying that in the original GoLite code it would be type errors.

Certain Go(Lite) types also cannot be directly mapped to C. For raw strings, we just used strings in C (char array) but escape any special character (which is the purpose of raw strings).

Variables:

Dealing with variables is trickier than expected as the rules are not as close as they look between GoLite and C, especially since short declarations do not exist in C. This also depended on how we built our AST and how we dealt with multiple declarations. We opted for separate declarations as much as possible (some cases were not possible otherwise in C) but had to be careful about how the left-hand side and right-hand side were treated in multiple declarations. Short declarations are also special, they can redeclare variables but only if at least one of the left-hand side variables was never declared. It is also possible to do things such as one-line swapping which cannot be naively translated in C (excluding the one-line XOR-equals swapping trick which is just as "complex" as the temporary variable trick when dealing with its edge cases):

Go:

```
a,b := 0,1
a,b = b,a
```

C:

```
int a = 0;
int b = 1;

// Wrong!
a = b;
b = a;
```

Scoping:

Fortunately C supports brace scoping which helps enormously in mapping GoLite scopes to C somewhat a little more elegantly.

Expressions:

To avoid ambiguities, we surround expressions with parentheses.
For the *len* GoLite function, the types that are accepted by this special function are all implemented as arrays in C (e.g. strings are char arrays) so returning the length of the array (or used length of the array rather for things like slices) is good. For *cap* (slices) just return the full length. We also wanted to use the *__golite_* affix trick and make them special functions as slices would be a special struct in the generated C code with an array of capacity length and a field for the used length.

C:

```
typedef __golite_slice __golite_slice;
struct __golite_slice {
    union {
        bool content[DEFAULT_SLICE_SIZE];
        ...
    } val;
    int length = 0;
};
```

The *append* function for slices would also be a special function with the affix in the generated C code.

Miscellaneous:

In GoLite everything is "native" to the language so in C we had to add standard library includes to cover everything needed to map from GoLite: *stdio.h*, *stdlib.h*, *string.h*, *stdbool.h* .

## Conclusion

Despite the reasons for choosing our target language (and the reason Go was even created according to its creators), we think that C++, for the *implementation* language would have been a better choice. The lack of library was manageable but it got more and more annoying at each step/milestone. We spent a lot of time debugging problems because of limitations of C and workarounds we used that other more "modern" languages would have easily handled. One big example are strings, passing around strings and manipulating strings is a huge headache in C and we must be very, very careful. A lot of "quality-of-life" functions that are often used are not standard and don't work on some compilers and computers. We wanted to use *strcat*, *strdup* and *strcpy* a lot but just using them gave us problems, which is strange as in the lexer *strdup* is used without a problem. Just manipulating pointers and arrays in general is a well-known difficulty in C, along with cryptic segmentation faults.

Since C++ is "close" to C and it is possible to code C in C++ (somewhat), it would possibly have been a better idea. It was not agreed on originally as not all members had experience in C++ and since our target language is C, we thought it would be good. Also, as was mentioned before, it could also have been beneficial for writing our AST/parser since C++ is object-oriented, instead of having to deal with only structs and passing many, many arguments everywhere. Though ultimately maybe C works very well once we are experienced in writing compilers, we think C++ could have made things a little bit easier.

## Contributions

We worked on the lexer and parser together, and all three of us debugged the lexer.
After milestone1 was submitted, Ting debugged the parser and added weeding.

Su worked on the pretty printer. After milestone1 was submitted, the three of us all worked on debugging it together.

Ting worked on the symbol table. After milestone2 was submitted, Simon and Ting worked to debug it together.

Su and Simon worked on the codegen (while Ting worked on the Peephole Assignment). Ting worked on the three codegen parts mentioned in Milestone3 (identifiers, functions, loops), and Su and Simon did all the rest.

Tree design was worked on by all three of us.

The invalid/valid programs for milestone1 were all written by Simon. The programs for milestone2 were written by Simon and Ting. The three programs for milestone3 were written by Ting. Milestone4 programs were written by all three team members.