Milestone 1

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.

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.

Project

Description of the implementation language and tools and the impact on your project

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.

Description of group organization and individual contributions

It was initially hard to split work because due to the nature of the project, the scanner had to be done, then we had to agree on our tree design before moving to AST/pretty_printing. Because of this, we weren't able to work remotely on any section by ourselves, so we decided to meet up everyday to work on it together.

Pair programming and code review

The lexer and parser (and as such along with the grammar and AST) being so closely coupled makes it hard to work separately (as mentioned previously) so we did a lot of pair programming and we mutually reviewed our code a lot too. We would also often mix up our way of doing things which further confused us and introduced problems or inconsistencies in our code and design, resulting in further difficulties. Despite fixing them, the structure still reflects our design decisions that could have been more consistent. It is much better after "cleaning up" a bit. It was difficult to synchronize our work (concurrent programming is hard, for every meaning of "concurrent programming") as we would often talk about doing a certain thing, then go our separate ways and assuming someone else did it, which ended up on everyone forgetting to do the actual thing. We had to really work together to "get in sync".

Division of work

We tried to find the best way of dividing the work. From previous experience, most "real" big programming projects cannot be divided so cleanly without losing track of each other's work (as explained) and then encountering difficulties when merging the work. It can also cause problems similar to information silos while each member should at least have some knowledge of every part of the compiler. So we would constantly update each other and work closely as well as each contributing to every part. Of course, at some point the best way is to divide the work and at some point it is necessary that each member is more "specialized" at their part, but communication is important. Division was originally mainly done in the obvious ways: lexer,

parser, tree, pretty printer, example programs. The parser, tree and pretty printer are tightly coupled so they could not be divided well. We chose instead to split into lexer, example programs and then all focus on small parts of the parser together.

Ting is the main contributor to the lexer and parser.
The initial lexer and parser grammar were done by him and participated in the tree.

Su is the main contributor to the parser and pretty printer.
She participated in the parser grammar and tree as well as creating the pretty printer.

Simon is the main contributor to all the example programs.
He participated in the parser grammar and tree design, as well as the pretty printer. Valid and invalid program examples were written by him.

Overall, the AST part being heavy and the “meat” of the milestone was worked on heavily by all members. As mentioned previously, we all did a peer review of each other’s code and debugged together.