COMP520 - Group 10
Milestone #1

David Herrera
Id: 260422001

Maurice Zhang
Id: 260583688

Adam Bognat
Id: 260550169

February 27, 2017

1 Implementation Tools and Language

We have opted to use Flex+Bison and the C programming language to implement our GoLite compiler due to familiarity and flexibility. Separating the scanner from the parser allows for greater modularity, and the precedence directives available in Bison simplify writing the grammar. C is then natural to use with these tools. As we used these tools to implement the Minilang compiler, we all have facility with the tools and can thus focus on the challenges of the problem rather than fighting with the tools.

2 Group Structure

Our team has no hierarchy. We endeavor to divide the work equally, and there is often overlap between the tasks taken on. We validated each other's design decisions and code implementations and had great collaboration among all team members. The tasks were broken down roughly as follows:

- Adam contributed to the scanner, wrote most of the test suite (approximately 45 valid programs, 100 invalid programs), selected 30 invalid programs for submission, and completed the weeding phase. The test suite was refined with the help of the other members.
- Maurice, took charge of the parser and the implementation of the abstract syntax tree. Collaborated throughly with tests and decisions in the project. Implementation of valid programs.
- David, took over from Adam's scanner section and finalized the details of the scanner. Design decisions in terms of the AST and aid with it's implementation. Set up for project, Implementation of pretty printer.

3 Compiler Phases

3.1 Scanner

For the scanner the decisions were pretty straight forward. The following is a list of the main highlights for this part.

- To handle block comments the Regex from the O. Miller blog was used. [1]
- In terms of the escaped characters for the interpreted strings and runes, we did not translate them or interpret them yet i.e. '\n'. The runes, as a consequence, are treated internally as a char * in c as opposed to a char. This made things easier for the pretty printer. Having said that, eventually a decision has to be made of when to translate them.

3.2 Parser

For the parser, most rules were fairly straightforward to implement and were done so simply by following the Go specifications. The biggest issue that arose was a conflict between assignments and short assignments. On the left hand side of an assignment, expressions are allowed (only those that are addressable). This includes identifiers. For a short assignment, the left hand side must only comprise of identifiers. This created a conflict because if the first element on the left hand side of an assignment was an identifier followed by a comma, the parser would not know whether to reduce it to a list of identifiers, or to a list of expressions since they both could start with identifiers. For example,

```
varName, a, b := 2, 1, 4 // short assignment
varName, a[3], b.field = 2, 1, 4 // assignment
```

Therefore, to resolve this conflict, we allowed the left hand side of a short assignment to be expressions in the grammar. However, when constructing the abstract syntax tree, we would check and see if those expressions were only identifiers and throw an error if that wasn't the case.

Also, for structures that should be grouped together as lists, like a list of statements, linked lists were used. This was implemented with a field in each node type pointing to the next node. These links were created in the parser actions in reversed order to make for simpler code. Thus, when traversing through the tree, we had to make sure that lists were traversed in reversed order.

3.3 Abstract Syntax Tree

The construction of the AST was also a simple procedure without any pecularities other than verifying short assignments as previously mentionned. For some nodes that had many fields, like the `if` statement, we defined a separate structure. Also, although a block is only a list of statements, we gave it its own structure type to make for clearer code.

3.4 Weeding Phase

Several parsing details were deferred to a weeding phase:

1. Checking that the number of identifiers on the left-hand side of an assignment statement equals the number of expressions on the right-hand side. In hindsight, this could have been more easily implemented when constructing the assignment node during AST construction.
2. Ensuring a `switch` statement has no more than one `default` case.
3. Ensuring that a `continue` statement only appears within the body of a loop.
4. Ensuring that a `break` statement only appears within the body of a loop or `switch` statement.

Furthermore, semantic analysis will be used to ensure the correct use of `return` statements, but this is work left for the next milestone.

3.5 Pretty Printer

The only noteworthy part of the pretty printer is that the type and var declarations given by:

```
type (
    Point struct{ x, y float64 }
    Polar Point
)
&
var (
    i int
    u = "bar"
)
```

There were translated into the equivalent form of multiple type and var declarations respectively. The last modification is the function declaration, in this part, all the parameters have their type specified which is still valid and equivalent to the original version.

4 Testing

In this section, we made a variety of valid and invalid tests, a good portion of this test have been added to src/test in the assignment folder. We validated with 97 invalid tests and 50 valid ones, by testing different features of the language.

References

[1] O. Miller. *Finding Comments in Source Code Using Regular Expressions*, 2010.