Milestone 1: Teamwork Report
Design + Implementation Decisions

Alexander Iannantuono, Asa Kohn, and William Chien

February 21, 2020

1 Design and Implementation Decisions

1.1 Rationale of Implementation Tools Chosen

The first milestone as well as subsequent ones have and will be written in C using flex and bison. Given that we all used this in prior coursework, it made sense to go down this path of implementation tools. In the sections that follow, we will detail the implementation decisions undertaken for each component of our GoLite compiler required for this milestone.

1.2 Scanning

Most of the scanning implementation was the same as that in the assignments earlier in the semester. Given that GoLite has more keywords and built-in features than MiniLang, there was of course more to account and scan for.

Multiline comments were a point of discussion and something work on. We ended up deciding to match the first `/*` with the first occurence of the closing token `*/` for the comment block, as GoLite does not support nested comments. This was implemented using an elegant regular expression:

```
\/\*[^\*]*\*+([^\*\/][^\*]*\*+)*\/
```

As far as our testing goes, it appears to be functional.

For semicolon insertion rules, we decided to keep track of the last token prior to seeing any of the tokens found in the insertion rules. The insertion rules can be found in the Go language documentation [3].

No problems as far as we can tell is to be reported for scanning, as it followed from work done in the assignments, but more lines of code were written.

1.3 Parsing

In order to parse functions parameters, we decided to segregate those with no arguments and those with non-zero arguments as different rules in the grammar, where it was required (either as a statement list or an expression list). This allowed for better legibility and we hoped that it would cause less issues when it comes to grammar rules. Writing it with the hope of being clever (although having a mistake without realizing it) would result in unwanted things to be allowed by the parser.

For functions calls, we used the `%nonassoc` pragma on the `(` and `[` tokens. We gave this the highest precedence as this needs to be considered first. It appears as if that this could also be completed using the `%right` pragma, although it didn't seem as elegant and appeared to be more work to implement correctly.

We decided to defer the most of the verification of the blank specifier in the type checking phase of the compiler. We implemented weeding out the blank package identifier directly in the parser.

We will note that the rest of the parser implementation was straightforward given the Go language specifications [4].

1.4 Abstract Syntax Tree (AST)

The AST that we designed is quite simple, despite the many structures used. We also used interleaving as our method of implementation in regards to the AST. It follows the same structure as the assignment although there were a couple of differences as to what Alex and Will had done in their respective solo assignments. In this part we decided to create many C structures in order to keep the different kind of node types separated. The `else if` linking follows more of what Asa had done in his assignment. This actually helped immensely when it came to coding up the pretty printer as the structure of the AST was sound (after we sorted out all of the kinks that we could find).

1.5 Pretty Printing

Thankfully, a lot of the boilerplate code for pretty printing from the notes and the second assignment was used for this milestone. Given the very rich structure of the AST design (modulo the "back and forth" of design choice changes), writing the pretty printer was rather straight forward.

1.6 Implementation difficulties encountered

When working through this milestone, we realized that declarations can have arbitrarily long variables and values. This means that there does not exist a fixed k such that the parser is LR(k). Thus, we decided to make it a GLR parser and with some work, the issues seemed to work themselves out. This can be done with the `%glr` pragma in the bison file.

Designing the AST posed some issues as well. There were a couple of periods of "back and forth" where something that we thought would fix an issue did not and vice versa. Examples of this includes how to organize expression lists. There was also a grand realization of all the things that could be expression lists and we had to rectify that before moving forward. We decided on one way to do it which was as a linked list, which we then agreed to fix after this milestone as time until submission was rapidly approaching.

Given the back and forth nature of our progress, this set us back in progressing in other parts of the compiler components required for this milestone, as we'd have to change the code written based on the new structure of some of the AST nodes.

Finally, time constraints induced some challenges and late nights.

2 Division of labour and team organization

2.1 Dividing the work

We'd like to note that everyone in the group contributed equal amounts of respective work and time. We also all did testing and debugging together.

2.1.1 Asa

Asa was mainly in charge with writing most of the lexer and parser, although decisions and discussions were done as a group. He also spearheaded the AST part of the compiler, although we all decided together how to implement certain intracacies that presented themselves. Given that Asa did most of the structure of core components of the compiler, other group members had to rely on his ideas on how he thought components further down in the pipeline should be implemented. He also spent some time making sure that there were limited memory leaks in the compiler so far.

2.1.2 Will

Will had written half of programs (both valid and invalid ones) to be submitted for this milestone. He did some testing to make sure that the code was correct and debated on what were 'interesting' valid programs with Alex. He was responsible for organizing that part of the project to confirm it met the specifications of the first milestone. He also took care of weeding, which was then amended by Asa before submission.

2.1.3 Alex

Alex was mostly in charge with writing programs and verifying the work of others. This included going over the bison source code file in an attempt to discover *elegant* ways of writing the required grammar for GoLite. He also took charge in organizing the work and being in charge of keeping the repository in check. By keeping the repository 'in check' we mean that what has to be accomplished is (or attempts to be) well established within the repository. This also includes making sure that everything is done as properly as possible. Finally, taking charge of making pull requests and merging is a part of this task.

Alex also did most of the pretty printing, which was of course verified and discussed with the other members of the group.

2.2 Organization

In order to be productive, we decided to set up certain communication channels. This was done using Matrix which is a free alternative to Slack. Further, some markdown files were created to do basic TODO lists, but given the time frame, it wasn't as successful – each team member had their own list of things to focus on instead.

We also met quite frequently as the deadline for submission approached, this is true as of writing this report as we all work together in Burnside basement.

References

[1] Babylonian square root method. `https://en.wikipedia.org/wiki/Methods_of_computing_square_roots#Babylonian_method`. Accessed: 2020-02-17.

[2] Go language rune literal specification. `https://golang.org/ref/spec#Rune_literals`. Accessed: 2020-02-17.

[3] Go language semicolon rules. `https://golang.org/ref/spec#Semicolons`. Accessed: 2020-02-20.

[4] Tour of the go language. `https://tour.golang.org/list`. Accessed: 2020-02-22.