Design Document for Milestone 1

Lore, J., Lougheed D., Wang A.

March 2, 2019

Contents

This document is for explaining the design decisions we had to make whilst implementing the components for milestone 1.

1 Introduction

1.1 Implementation Language

For our `GoLite` compiler, we decided to use `Haskell` as the implementation language, as many of the required tasks can be done more naturally (i.e. tree traversal, recursing over self defined structures, enforcing types, etc.).

1.2 Tools

For scanning, we use the program `Alex`, which is similar to `flex` but produces `Haskell` code instead. We use `Happy` for parsing, which resembles `bison/yacc`. `Alex` and `Happy` can be interlinked nicely, passing on useful information (encompassed through a state monad) such as the line number, column number, character offset in file, and error messages.

There are several other good Haskell scanning/parsing options, such as `Megaparsec`. We decided on `Alex/Happy` because of their syntactic similarity to `flex/bison`, which we learned in class.

We use `stack` to manage our Haskell project and corresponding tests. We implemented additional tests using the `hspec` package.

1.3 Miscellaneous

1.3.1 Continuous Integration

We use Travis CI to build our project on every push to GitHub, as well as run our tests to make sure that new changes did not break existing functionality.

1.3.2 File Organization

- `src/`
  - `generated/` (Code generated by `golite.y`, using `Happy`, and `golite.x`, using `Alex`)
  - Modules for each feature
- `app/`
  - `Main.hs` (main, imports the needed modules and uses `ParseCLI` to parse the command line arguments and call the function we want)
- `test/`
  - `Spec.hs` (auto discovers `hspec` tests to run them all with `stack test`)
  - Spec modules, each exposing a test suite

2 Scanner

The first step of this milestone was to scan user input into tokens. Handing of most GoLite tokens was straightforward and similar to class assignments. In other cases, we had to deal with some of Go/GoLite's quirkiness when compared to something simple like `MiniLang`.

2.1 Semicolon Insertion

Semicolon insertion was the first scanning hurdle. To solve this, we use start codes, a built in feature of `Alex`, and have special rules for certain states.

If the last token we scan is something that can take an optional semicolon, we enter the *nl* state, where newlines are transformed into semicolons after specific tokens. Scanning anything else (that isn't whitespace or otherwise ignored) returns the scanner to the default state (0), where newlines are just ignored.

This solution seems elegant, as we don't have to traverse the whole stream post-scanning or store context, other than the start code. It also makes sense to let the scanner handle these situations automatically, rather than trying to deal with them in the parsing phase.

2.2 Block comment support

Block comments cannot be easily scanned using only regular expressions. One potential solution was to use start codes again (changing state upon opening a comment). This was ruled out since it would potentially add a lot more start codes (i.e. states), as we'd have combinatorial possibilities with the newline insertion state (since block comments spanning newlines should also insert semicolons when they're optional). Using a state approach would also make it harder to catch unclosed comment block errors.

Our solution iterates through the scanner's input once we receive an open comment block and ignore everything until the comment is closed. If the comment never closes, we can easily emit an error.

To account for semicolon insertion with block comments, we set a semicolon flag to true if we encounter any new line inside the block comment, and we insert a semicolon if we're in the aforementioned newline insertion state.

2.2.1 Adding newlines at the end of the file if they aren't present already

`Go` requires a semicolon or newline at the end of function declarations. Sometimes, there may not be a newline at the end of a file, so no semicolon insertion occurs, resulting in a parsing error.

In order to correct for this type of file, we enforce a final newline by appending one if necessary. To do this, the code string is preprocessed prior to scanning. This is the easiest solution, as otherwise we would have to try inserting a semicolon if we are in the *nl* state when we encounter an `EOF`, while also making sure to return an `EOF`. This would be difficult without additional context information.

2.3 Nicer error messages

We decided to use `ErrorBundle` from `Megaparsec` in order to output nicer error messages, with program context for easier debugging from an end-user programming perspective:

```
Error: parsing error, unexpected ) at 5:22:
  |
5 | func abstract(a, b, c) {
  |                      ^
```

With `Alex`' default behaviour, we did not have access to the entire source file string, as it is not kept between steps. In order to generate the contextual message, we modified the `monad` wrapper provided with `Alex` (see `TokensBase.hs`) and changed the `Alex` monad to wrap over a `Either (String, Int) a` instead of `Either String a`, i.e. in addition to storing an error message on the left side of the monad we also carry an `Int` which represents the offset of the error. When we want to print the error message, we can then append the part in the source file where the error occurred.

3 Parser

3.1 Grammar

Many of our difficulties in the grammar were associated with identifier and expression lists, used in declarations/signatures and assignment/function calls respectively. The grammar was refactored to fix this by allowing identifier lists to become expression lists if needed, in a way which avoided introducing other conflicts.

The first issue we encountered was with list ordering. LR parsers work more intuitively with rules that put the newly-created terminal after the recursively-expanding non-terminal. However, since Haskell uses recursive lists defined in the opposite way, it is significantly more efficient to prepend items. This prepending results in a reversed ordering, which must be handled after the list is 'complete'.

Adding an extra non terminal to manage reversals for each list would needlessly increase our grammar and generated code size, so we decided against it as a solution. The solution we use is to differentiate lists containing at least one non-identifier expression (i.e. using either all non-identifiers, identifiers plus one non-identifier, or otherwise mixed lists, as grammar base cases) from lists of entirely identifiers. Then, the expression list non-terminal is allowed to yield either a mixed list or a pure identifier list depending on what is needed.

Another caveat of how lists are handled in the grammar, again a compromise to prevent ambiguity, is that the actual grammar constructs that represent lists correspond to a list of size two or more, which doesn't exactly match the Go spec (where a list may be 0/1 or more, depending on the case). The actual single-item non-terminals are allowed to represent a list of size one when needed, meaning this disparity is resolved in the actual AST construct, which is closer to a direct representation of the Go / GoLite specifications.

3.2 AST

The AST is largely a one to one mapping of the Golang specs, with parts we don't support removed and additional parts for Golite added. In some cases, there are minor deviations from the CFG.

3.2.1 Accurate Type Representation

We modeled our AST as close as possible to the actual Go and GoLite specs, to try and ensure that impossible states are inherently prevented by the Haskell type checker, reducing run-time errors. Although we don't have type-checking implemented at this milestone, we can use this technique to enforce definitions such as 'exactly one', 'one or more', and 'zero or one'. This modeling is not always perfect. For example, a list of identifiers is 'one or more' (in Haskell, `NonEmpty`). Many locations make it optional. While a direct translation would be `Maybe (NonEmpty a)`, we choose to make it a possibly empty list `[a]` as it makes more sense.

3.2.2 Simplified Data Type Categories

Some splits, such as `add_op` and `mul_op` are distinguished purely to demonstrate precedence; they are in fact only used once in the specs, so we decide to merge them directly in our `ArithmOp` model. Several other instances exist.

Given we are creating an AST, rather than a CST, we can further compact parts of the grammar. For instance, an `if` clause in the spec leads to an `IfStmt` construction, whose `else` body is either a block (with surrounding braces) or another `if` statement (no surrounding braces). In our case, we don't need to model the braces, so we can treat the `else` body exclusively as `Stmt` rather than the more verbose `Either Block IfStmt`. The grammar enforces that this `Stmt` is not any other type.

3.2.3 Structure Simplification

For `var` and `type` declaration, we make no distinction between single declaration (exactly one) and block declaration (0 or more). Unlike types, which produce different formats, we decide to enforce all declarations of one var to be single declaration. In other words, `var ( a = 2 )` would become `var a = 2`. Note that we cannot further simplify group declarations `var ( a, b = 2, 3)`, as there is no guarantee at this stage that the number of identifiers matches the number of values.

3.3 Weeding

Most of the weeding operations needed are simple and don't rely on external context. As a result, we were able to define recursive traversal methods to verify relevant statements, and create verifiers that validate at a single level. Haskell helped immensely here, as we were able to use pattern matching to produce performant and independent functions.

For verification where statement context was important (`break` and `continue`) we made a simple modification to the recursive traversal to avoid exploring scopes under `for` or `switch` statements where applicable.

Each verifier returns an optional error, and we are able to map the results and return the first error, if any.

4 Pretty Printer

When creating our pretty printer, we chose a top down approach. Every node has the ability to output a list of strings, which makes it easier to format indentation. Each node is also only concerned with its respective subtree, and does not require context from its parent. We focused on aesthetics, focusing on proper spacing and alignments. In the case of expressions, we tried to add brackets sparingly, though further optimizations can be done down the road (a nested binary op does not always need brackets, if the order of precedence matches). To produce the full program, we simply join the list of strings in the full program, intercalated with new lines.

5 Team

5.1 Team Organization

We started the project by dividing the main components (scanner, parser, AST/weeding) among the three group members (Julian, David, and Allan respectively). We used GitHub's organization features extensively in order to keep track of design goals, report bugs, and keep code quality as high as possible.

5.2 Contributions

- **Julian Lore:** Wrote the majority of the scanner and handled weird cases, wrote a large amount of valid/invalid programs, implemented many other tests (`hspec` or small tests in our program) and looked over the parser, contributing a few things to it as well.
- **David Lougheed:** Wrote the bulk of the parser grammar and contributed to the weeder. Also wrote 3 of the valid programs and 8 of the invalid ones and had minor contributions to miscellaneous other components. Contributed to the testing of the parser and pretty printer.
- **Allan Wang:** Created the AST and helper classes for pretty printing and error handling. Wrote the base package for testing as well as some of the embedded test cases within `hspec`. Added integrations (Travis + Slack), and gave code reviews to the other components.