Milestone 1

Tools and languages

We chose SableCC as a compiler-compiler. This is because most team members had used it for assignment 1, and because it was explained in class. We used the latest stable version: 3.7.

Using SableCC limits us into using a JVM language. Again, based on previous experience, we chose Java. We used version 8 since it is now widely supported and offers a lot of useful features over previous versions.

To simplify building across Linux, mac OS and Windows, we opted to use Gradle as a build tool. We used a plugin to integrate SableCC as part of the build process. The only caveat is that we had to make our own plugin, and so it must be bundled with the project. Otherwise, Gradle downloads dependencies, compiles, tests and runs the project with a few simple commands. Gradle doesn't need to be installed on the machine either, thanks to the Gradle Wrapper.

Other than SableCC, the project depends Apache Commons CLI for the command line application, and on JUnit for testing.

Implementation

Both the lexer and parser where implemented using SableCC. We also use the AST definitions to transform the CST. There isn't much to say about this, it's a rather straight forward translation of the Go syntax specification.

There is only one case were we had to do something different. In short variable declarations, the left side should be a list of identifiers, but due to a shift-reduce conflict, we use a list of expressions instead. This is weeded out after.

The weeder also: takes care of proper usage of `break`, `continue` and `return`; checks for redundant names in declarations; checks that lists are balanced on either side of multiple declarations and assignments; checks for multiple default cases; check that statement expressions are calls only; and checks that blank identifiers are used in valid places.

The pretty printer is written around the `SourcePrinter` class, which outputs to a `Writer`. This can either be a string or a file. It inserts indentation automatically after new lines, according to the current level. The `PrettyPrinter` class is a switch that recursively traverses the AST to print the nodes, from bottom to top.

To print nicer error messages about nodes, a `NodePosition` switch can be applied to a node to extract the start and ending position. This is done by visiting the `Token`s, which are the leaves of the AST, and using the minimum and maximum position values.

For automatic semicolons, an extension of the Lexer called `GoliteLexer` is used. It implements the insertion rule as per the Go specification. More information can be found inside.

A `Golite` class is used to perform compilation tasks. Right now only `parse` and `prettyPrint` are implemented.

The `App` class implements the command line interface, and runs the proper task on the input file, then outputs to the given output file (or a default if none is given).

The other main package classes are used for exception management.

The test package contains a runner called `SyntaxTest`, which runs all valid and invalid programs through the parser. It also checks the pretty printer invariant on the valid ones. It automates testing. Tests are run when building, or by using `./gradlew test`.

Team work summary

Aleksi Sapon set up the project structure. He implemented the command line application and test runner. He contributed parts of the pretty printer and weeder. He wrote the entire expression grammar, and part of the declaration one. As for test cases, he wrote two valid programs, ten invalid ones, and a few more for the extra tests. Finally he also took care of overall code quality and consistency.

Rohit Verma setup the initial grammar file. He implemented the Helpers, Tokens and Declarations section in the CST and did the CST to AST conversion for the same part. He also did the statements for the pretty printer. For general testing of the parser, he wrote a bunch of unit test cases which are in valid_extra directory. Found many bugs towards the end and helped with bug fixes at many places including the weeder. For submission, he wrote two valid and ten invalid programs.

Ayesha Krishnamurthy contributed to the grammar, primarily by adding statements to the grammar and adding AST transformations and nodes for statements. She also worked on the Weeder with Aleksi and helped out with bug fixes in the Weeder and PrettyPrinter. For general testing and the submission, she wrote two valid and 10 invalid programs.