COMP 520 Design Document - Milestone 1

Zoe Guan, Samuel Laferriere, Emil Rose

February 16, 2015

For the GoLite project, we decided to use SableCC as our implementation tool. SableCC was a difficult tool to use at first, but proved to be very efficient in the long run. In particular, the convenience of having all of the node files automatically generated with corresponding getter and setter functions is a major advantage over bison/yacc. SableCC provides useful analysis classes, such as DepthFirstAdapter for AST traversal, which can be extended to apply customized actions to nodes; this facilitated the implementation of the weeder and pretty printer. An advantage of using a tool that produces code written in Java is that limitations of SableCC could be solved using object-oriented features like inheritance, e.g. for our implementation of the GoLiteLexer discussed below. Other tools we used included Makefile and Bash scripts to allow rapid compilation and testing of the changes we made.

The scanner part of the project was mostly straightforward, except for the rules for inserting semicolons. To deal with the fact that the lexer is automatically generated, we decided to create a class GoLiteLexer that extends Lexer. When a newline/EOF token is preceded by a token that could end a statement, GoLiteLexer replaces the newline/EOF token with a semicolon token to meet the GoLite requirements; otherwise, it functions the same as its superclass. Our main function uses this extended scanner, rather than the generated scanner, to read through the file. We also extensively used helpers to make the list of tokens easier to both read and write.

For the parser, we first made a rough grammar based on the GoLite specification, and debugged it. We then went more carefully through the overall Go specification and made sure every valid phrase in GoLite could be expressed by our grammar. We tried to express as many rules as possible in the grammar to minimize the number of things to weed for. For example, we enforced the rule that the left-hand side of an assignment must be a lvalue by defining lvalue in the grammar. We also enforced most of the rules for the placement of blank identifiers by making a distinction between blank and non-blank identifiers in the grammar. At times, we ran into phrases that we couldn't, or had trouble, accounting for with an unambiguous grammar. We decided to relegate some of

these problems to the weeding phase to make our grammar simpler while still guaranteeing correctness of the parse. For example, to ensure that the left side of a short variable declaration is a list of identifiers, we defined the left side to be a list of lvalues and used the weeder to verify that the lvalues were identifiers. We implemented the weeder by extending the DepthFirstAdapter class. The weeder also handles the following cases: the left side of an assignment has the same number of elements as the right, "break" and "continue" only appear within a loop and there is at most one default per switch statement.

For the Abstract Syntax Tree, the transformations shorten the Concrete Syntax Tree considerably by reducing most constituents of the program to statements and expressions. The only part that proved troublesome was the transformation from production rules that used the * EBNF notation instead of having specified list-unwrapping helper rules. When building the AST, it would have been useful to unwrap some of these lists (for example if the * notation is used on a list to make a list of list, and we want to store each individual list in a node by itself), but SableCC does not seem to offer this possibility. Being reluctant to modify our grammar once we finally got it working, we overcame this problem by instead introducing helper rules in the AST itself, such as the else_if rule.

For our pretty printer, we again extended DepthFirstAdapter. Our pretty printer emulates idiomatic Go in omitting semicolons except when absolutely necessary (for example, in a three-part for loop clause). Thus, each new declaration or statement starts on a new line. Type and variable declarations where the "type" or "var" keyword is distributed over a list of specifications are expanded into multiple short declarations that each start with the keyword. For indentation, we used a tabCount variable that is incremented at the start of a new block and decremented after the end of a block. The printer checks the tabCount at the start of each new line to determine the size of the indent.