Justin Domingue – 260588454

Daniel Pham – 260526252

Seguei Nevarko – 260583807

Milestone 1

COMP 520

Work presented to Prof. Laurie Hendren

McGil University

Thursday, February 26, 2016

LANGUAGE AND TOOLS

We decided to use flex/bison because that's what we used in the assignment and while SableCC seems like a promising avenue, we were already comfortable with flex/bison.

SCANNER

As discussed in the Language and Tools, we used flex to generate a lexical analysis of the input Golite code. Most rules are basic regex similar to what we have seen in class and done in the assignment. However, some intricacies of Go(lite) needed to be parsed at a finer level. To do so, we used a regex to capture the first few characters ("/*" for comments, " ' " and " ` " for comments, interpreted strings and raw strings). We then analyzed the input character by character, checking for allowed escapes and saving to *yytext* as needed. This implementation was suggested to us by the ANSI C grammar.

PARSER

The parser was completely handwritten from scratch in Bison following the Go(lite) specification document. It was rather similar to what we did in the assignments. It should be noted that while we used to allow optional semi-colon, this feature was turned off to comply with the Golite specification. Also, parenthesized types are allowed, to be inline with the Go specs (although the reference compiler does not allow them). https://golang.org/ref/spec#Types

To make the grammar simpler, we allowed over-generation to mitigate reduce/shift-reduce conflicts.

1. In function calls, the id is an expression to allow nested parentheses.
2. Function declaration with returns do not need to end with a terminating statement
3. In function declarations, the number of arguments in a return statement doesn't have to match the number of return arguments defined in the function signature
4. Continue and break statement can be anywhere in the code
5. Expression statements can be anything.
6. Post conditions for for loops can contain variable declarations statements
7. In short variable declarations, a list of identifiers cannot be distinguished from a list of expression, leading to shift reduce conflicts. Therefore, we use an exp_list on the left side instead of id_list.
8. Left-hand side of assignment, increment and decrement statements doesn't have to be an lvalue
9. A switch statement can have multiple default cases
10. In a variable declaration and assignments, the number of identifiers doesn't have to match the number of expressions
11. Division by 0
12. '_' can be used as value

Pretty Printer

The pretty printer follows has a standard implementation. Runes are escaped. It should be noted that we support indentation.

Tree

Serguei tried to build a tool to automatically convert our grammar to a tree. It didn't work so well because the generated tree was following the grammar too closely. We ended up with a concrete syntax tree which was much more difficult to use than an abstract syntax tree. There didn't seem any straightforward way to convert the grammar to an AST directly (letting the computer choose what tree nodes to merge).

Had we been able to write the tool to convert our CFG to an AST automatically (without specifying any node merging rules) we would have able to rewrite the grammar with the greatest of ease. Indeed, for any change to the CFG, we need to fix the grammar.

A particularity of our abstract syntax tree is that we have a node that is that our EXP (expression) node has a kind for every operation (binary, unary, or other) but there is a single structure for binary expressions and a single structure for unary expressions. It made the AST shorter and more readable.

Weeding

Every weeding prospects identified in the Parser section were dealt with in this milestone. However, the following over-generations have a few particularities:

- 1. (Function calls) will be weeded out by the type checker
- 11. (Division by 0) used to be weeded out – but after talking with Vincent, we simply allow the possibility of having runtime errors
- 12. (Blank ids) can only be used on the left of assignments and short variable declarations. They also aren't allowed as function ids since Golite does not support function literals. The same reasoning is applied to struct declarations.

Note that return statements in function declarations (2. and 3.) are disallowed although the reference compiler allows them. They follows the Golang spec:
https://golang.org/ref/spec#Terminating_statements

Team organization

We started working on the first milestone before our repository was set up. Here is a quick overview of the work done by each member (do note that peer programming was extensively used throughout this milestone):

Dan:

- Lexer
- Fixed reduce/shift-reduce conflicts in the parser
- Handwriting parts of the tree
- Fixed the weeder
- Fixed the bugs reported by colleagues
- Tested
- Set up the initial environment and base package

Justin

- First iteration of the parser
- Handwriting most of the tree
- First iteration of the weeder
- Various bug fix and improvements
- Tested

Serguei

- Investigated automating tree generation
- Wrote the entirety of the pretty printer and tree printer
- Reported many bugs related to implementation and portability
- Ran final set of tests