Milestone #1 – GoLite Scanner/Parser/Pretty-Printer
COMP 520

Keith Strickling
Team 15
February, 2017

Introduction

The project is managed, built and tested using the C language. This is a straightforward scanner, parser, and pretty printer implementation. A description of each phase in implementation will follow in next section.

Design Decisions

We elected to write this compiler in C and to use Flex/Bison to write the scanner and parser as we already gained some experience in using these tools to write the compiler for the MiniLang language. Additionally, we are all comfortable in C.

Scanner

There is nothing particularly interesting about the design of the scanner. After all, it is nothing more than a collection of regular expressions. However, as GoLite requires semicolons as terminators for essentially everything, but we allow the programmer to omit them in most cases, we keep track of two boolean flags, `shouldInsertBeforeNewLine` and `shouldInsertBeforeRightBrace` throughout the scanning process and insert semicolons in appropriate places based on the value of these flags. We support both semicolon insertion rules of the Go language (https://golang.org/ref/spec#Semicolons), though only the first is required. The first flag is used to support the first insertion rule, while the second flag is used to support the second rule.

Parser

Again, there's nothing too special about the parser implementation. However, some language requirements cannot be expressed using a context-free grammar (for example, ensuring that the expression lists on either side of a multiple-assignment statement are of equal length cannot be done using a context free language – this can be proven using the pumping lemma). Additionally, other aspects are very difficult to specify in the parser without creating shift/reduce and reduce/reduce conflicts. To account for these two things, we defer the following language requirements to a weeding phase:

- For variable declarations of the following forms,
    - `idList '=' expList ';'`
    - `idList type '=' expList ';'`

the idList and expList must be of equal length. We perform this check in the weeding phase.

- Similarly, a single assignment statement can assign many expressions to many variables. Moreover, each "variable" on the left hand side of the equals sign must be an lvalue. i.e. in a perfect world, the parser production to match an assignment statement would be `lvalueList '=' expList`, where both lists are of equal length. However, we use the production `expList '=' expList` instead, and use the weeding phase to ensure both lists of of the same length and that the left expList consists of only lvalues.
- The second kind of assignment statement is a "binary" assignment statement where the assignment operator is a binary operator, e.g. `*=`. In such statements, the left hand side must be an lvalue. However, we instead match on any expression and then ensure that the expression is an lvalue in the weeding phase.
- The expressions operated on by increment and decrement operators (++ and --) must be lvalues. Again, we match on any expression instead and check that the expression is an lvalue in the weeder.
- A short declaration statement consists of a list of identifiers set equal to a list of expressions, i.e. `idList ':=' expList`, where both lists are of the same length. In our parser, we instead match on `expList ':=' expList` and use the weeder to check that the two lists are of equal length and that the left expList consists of only identifiers.
- Expression statements (i.e. expressions that appear by themselves as statements) can only be function calls or receive operations. Our parser accepts any expression as a standalone "expression statement" and later weeds out the expressions that are not function calls or receive operations.
- Break and continue statements are only allowed within loops. This is enforced in the weeding phase.
- Switch statements can only have one default case. This is enforced in the weeding phase.

AST

The AST is composed of the following nodes:

- PROGRAM – the root of the tree
- PACKAGE – describes the package statement that every GoLite program begins with
- TOPLEVELDECLARATION – a wrapper on VARDECLARATION, TYPEDECLARATION, and FUNCTIONDECLARATION
- VARDECLARATION – encapsulates a variable declaration
- TYPEDECLARATION – encapsulates a type declaration
- FUNCTIONDECLARATION – encapsulates a function signature and its statements
- PARAMETER – parameter of a function
- ID – an identifier
- FIELD – a field of a struct
- TYPE – captures a type
- CAST – a wrapper around the TYPE and EXP nodes of a cast operation
- STATEMENT – encapsulates one statement of the program
- SWITCHCASE – encapsulates one case of a switch-case block
- EXP – encapsulates one expression

The parser never produces any CAST nodes as it is unable to distinguish between a cast and a function call. Thus, it represents all casts as function calls. We will later convert these to CAST nodes (likely in the type checking phase).

Pretty Printer

The pretty printer simply traverses the AST produced by the parser and converts each node into GoLite code! There is nothing special going on here.

Test Programs

There are two folders for test programs – valid and invalid. The description of each valid test program and of the error in each invalid test program is mentioned in the first line of each file as a comment. The GoLite specification has been followed for each valid program. Also each valid program has been tested at using Go play for validation. Additionally, the invalidity of the invalid programs has been confirmed through the same site.

Miscellaneous

The following has also been deferred to a later stage:

- The first expression in an append statement needs to evaluate to a slice, and the second needs to evaluate to the same type as that of the elements in the slice. This will probably be enforced in the type checking phase.
- A short variable declaration may re-declare variables provided they were originally declared earlier in the same block or as parameters to a function if the block is a function body. Moreover, if re-declaring a variable in a short declaration statement, it must be re-declared with the same type as it was originally declared. Lastly, at least one of the variables in the short declaration statement must be new. These things will likely be checked and enforced in the symbol/type checking phases.

Contributions

Keith Strickling: Scanner, parser, AST, pretty printer
Shruti Bhanderi (from previous group): wrote all test programs

Keith would like to give a big shout-out to Alex, Hanfeng, and Prabhjot (especially Prabhjot for the many hours she spent helping him eliminate parser conflicts) for their generous support throughout this initial Milestone.

Resources used

- Semicolon insertion: http://stackoverflow.com/questions/10826744/semicolon-insertion-ala-google-go-with-flex

- The Go specification
- The GoLite specification
- Go play