COMP 520 Milestone 1

Grégoire Moreau, Nathan Sandem, Jiawen Wang

March 2, 2019

1 Tools

Milestone 1 was completed with the `C` programming language. It was a language that every team member had experience with, and what all three of us used for Assignment 1 and 2, so familiarity was a huge factor in the decision. In addition, using C had its own advantages including: its balance of both high-level and low-level capabilities, its portability (our OS tally is: one MacOS, one Windows, one Windows/Linux dual-boot), its memory allocation functions, and its relative speed in most domains compared to languages like Java.

In addition, we used the flex/bison scanning and parsing tools presented in class. Again, all of us had experience with both of these for the first two assignments, so using them again made for a painless transition into GoLite.

2 Setup

Project setup The GitHub repository's first commit was made by Grégoire. The initial starter code, directories, starting Makefile, and bash scripts were taken directly from the same repository as the first two assignments. We also familiarized ourselves with repository tagging and some of us with GitHub in general. Some of the skeleton code and structure, especially in `golite.l` and `golite.y`,was taken directly from our old assignments.

3 Scanning phase

The scanning done in `golite.l` , and token declarations in `golite.y` was written by Jiawen. The scanning patterns came almost directly from the golang specifications. It closely resembles the general pattern of the first assignment, with some added helper definitions (like `ASCII`, etc.), and the addition of a method to add semicolons where needed. Here the github repository on optional semicolons was used as reference.

4 Parsing phase

The parsing done in `golite.y` was written by Nathan. The decisions behind its design were largely to follow the grammar rules specified by the golang.org website as closely as possible. As the parser began to take shape, it became necessary for members of the group to modify it to fit the needs of other sections of the project as a whole, such as in the case of the semicolon rules used by the scanner. The actions implemented by the parser after each rule were also implemented by various members as was convenient, and ensured the synchronization and agreement of the parser and the AST.

5 AST

The abstract syntax tree done in tree.c and tree.h was written by Grégoire although some modifications were made to it by the other group members to integrate it better with the other parts of the compiler and make the other phases easier. Some design decisions were made during its implementation.

First, we used a `TOPSTMT` structure that allows us to separate top level declarations from regular statements that could happen inside function definitions. This distinction assures us that none of those regular statements can happen outside a function, and that no function can be defined inside another function, as that would be impossible in the tree structure.

Unlike in `Minilang` where there was a finite number of possible types, GoLite allows the user to define structures and arrays of different capacities. It also allows the user to rename types. All those new features mean that we can't just store types as enums. Therefore, we use a structure called `TYPE` which itself contains an enum that describes the type's "`kind`". If the type we want to describe is a base type (`int, float64, bool`,...), then the enum will fully describe the type. However, in all other cases, a union in the `TYPE` structure will store additional information about the given type. For example, if the given type is a simple identifier, the `TYPE`'s kind will be stored as "`unknownType`" and the identifier will be stored in the union. This identifier could well be "`int`", "`float64`" or the name of another base type as they are not reserved keywords. The identifier's validity and meaning at that point of the program will be checked in a later phase of the compiler using the symbol table. For structure definitions, the union will store the list of fields declared for that structure. For slices, the union will simply store the type of the slice's entries, while for arrays it will also store their declared capacities.

Finally, `tree.h` also contains the definition of structure `LIST`. This structure is implemented as a simple linked list and allows us to easily store lists of identifiers and expressions for declarations and assignments, and lists of parameters for function definitions and calls.

6 Weeding

The weeding phase was implemented in `weed.c` and `weed.h` by Nathan. The weeder traverses the tree and marks boolean values `true` in subtrees where `break, continue` are valid and rejects `break, continue` in subtrees outside of where marked valid. To weed out multiple defaults, a boolean value is marked as true at the beginning of a `switch/case` and marked false again after a single default is found, whereafter all defaults are rejected. All expression statements are inspected to be sure they are functions. It was also decided to disallow the empty identifier for the package name at this stage, though to only disallow other incorrect uses of the empty identifier at later stages. And lastly, the weeder checks the number of expressions on either side of an assignment operation and counts the number of them and rejects the AST where they do not match in count.

7 Pretty printing

Pretty printing was implemented in `pretty.c` and `pretty.h` by Jiawen, with indentation done by Grégoire. It followed the typical AST traversal, starting from the 'root', of the tree and recursively calling the children nodes. The core methods were `prettyPROG`, `prettyTOPSTMT`, and so on for each corresponding struct from the AST.

For the sake of keeping the appropriate associativity of expressions, every binary and unary expression was parenthesized. Things like the Go feature of distributing the keywords `'var'` and `'type'` over a list of declarations and types, respectively, was implemented in the AST and pretty printer in a way that printed each field as if they were declared individually. Rune literals, especially those of escape characters, were handled with extra care to avoid having the escape sequences affect the `C printf` statements themselves.

The indentation was done by passing an integer parameter—signifying the current depth of scope of the program—between every parent node and their child. The printer also kept track of the boolean value `line_end` when printing statements, to keep track of which statements require an indentation or a new line.

8 Test programs

Each group member wrote 2 valid and 10 invalid GoLite files. Those programs tested a variety of GoLite constructs to ensure that invalid files were refused by the compiler. The valid files also allowed us to check the output of the pretty printing phase and the validity of the pretty invariant.