Milestone 1

Team Members

- Michael Sukkarieh
- Nicholas Paun
- Justin Tremblay

Design Decisions

Tools

- Language used: Ocaml
- Scanner generator: Ocamllex
- Parser generator: Menhir
- Other: Opam (package manager for Ocaml), ocamlbuild (build system)

We selected Ocaml as the implementation language for two important reasons. Firstly, the ML family is particularly suited to the implementation of programming languages as its support for variant types and pattern matching allow abstract syntax trees to be manipulated with ease. The strong type systems also help to reduce errors in the design of the parser and inconsistencies between components imlemented by different team members. Another advantage of Ocaml is that it provides excellent lexer and parser generating libraries which are not dissimilar to lex and yacc, making these tools easy to learn. At this stage, there have been few frustrations with Ocaml, except that its error messages are appallingly poor and occasionally misleading especially when combined with a parser generator.

Scanner

The scanner was generated using Ocamllex. Each significant word in the GoLite language was represented with its own regular expression (regex) which it would match and return its corresponding token. The majority of these regex expressions are trivial, but a brief description of some notably interesting regex expressions/scanner features are provided below. These include: **line numbers, semicolon insertion, block comments, and int literal types**.

Line Numbers

Line numbers are implemented using a global int ref which is incremented whenever a newline character is encountered. This ref is then used to print useful error methods whenever a syntax error is encountered.

Semicolon insertion

As mentioned in the GoLite spec, only the first semicolon insertion rule mentioned in the original Go spec will be implemented. Our compiler implemented this

feature using a boolean flag (`insert_semi`, implemented as a `int` ref) which records if a semicolon should be inserted. This flag is set to `1` after we encounter a token which invokes semicolon insertion (ex. an identifier), and is set to `0` in all other cases. Once an EOL character is given, we check if this flag is `1` and return a `SEMI` token if it is. Special care had to be taken to ensure this worked with block comments, this is explained in the block comment section below.

Block Comments

Ocamllex allows us to implement multiple parse methods, and we take advantage of that to implement block comments in our scanner. Effectively, this is similar to having different "states" within our scanner, allowing the scanner to behave differently on the given input based on its "state". A separate *block_comment* parse method is called when the scanner encounters "/*". In this new parse method, we ignore all input except for EOL and "*/". If an EOL is encountered, we increment the current line number and we recursively call the *block_comment* method - making sure to pass in true which signals that an EOL has been encountered. This is important since a semicolon must inserted at the end of the block comment if the comment spanned more than one line. Lastly, if "*/" is encountered the block comment is complete; we return a SEMI token if an EOL was encountered and we return to the original parse method.

Int Literal Types

GoLite supports integers represented in decimal, octal, and hexadecimal units. We chose to represent an integer with a single token (INTLIT), and convert all encountered integers into decimal units when the token is generated. This is done automatically by Ocaml's *int_of_string* method.

Parser

The parser was implemented using Menhir, a drop-in replacement for ocamlyacc which provides better explanations of rule conflicts as well as a number of convenience features, such as defining higher-order grammar rules for common cases like comma-separated lists. The overall design of the parser follows the AST and the Go specification fairly closely. For statements like declarations and assignments, our parser is responsible for immediately pairing the lvalues to the expressions. This would not work for Go itself due to the possibility of multiple return values. However, we have generally steered clear of complex semantic actions.

Errors are handled using Menhir's 'old' API, which upon a syntax error shifts a synthetic token called 'error' which rules can incorporate in order to target messages to the appropriate circumstances. We also considered using Menhir's 'messages files' in which samples of invalid programs are paired with a generated message. Once the parsing stage has been completed without syntactic issues, error messages can also use the pretty printer to illustrate the fault.

We chose to enforce certain syntactic rules with a set of weeding passes, each of which is responsible for enforcing a few closely related rules. `Terminal` ensures that `break` and `continue` are used only within loops, and that functions which must return a value in fact do so (the latter aspect is disabled in milestone 1). `Lvalue` currently does nothing useful, but will eventually check that expressions used on the left-hand side are appropriate. `Switch` ensures that `switch` statements have either 0 or 1 default cases (the default case is not always last in Go, making detecting these issues in the parser unpleasant.)

AST

The design of the AST is largely conventional, and much of the terminology used was chosen in accordance with the Go specification. Broadly, a program is structured as a list of top-level declarations, which may introduce types, variables and functions. The body of a function is a block, modelled as a list of statements. Some statements may contain expressions, declarations, and blocks. In order to attach line number and debugging information to various nodes in the AST, we created a polymorphic type named `annotated`, which is currently used to mark statements and expression operands. This type also holds a field named `_derived` which will allow expressions to be annotated with types by the symbol table generator and the type checker. A few more unusual aspects of our AST design are worth noting. For this project, we decided to make the AST as flat as possible, in order to reduce cognitive load due to levels of nesting. This leads to using polymorphic types in cases where some node types maybe subsets of another type, rather than chaining variant tags. The correct use of the blank identifier is enforced at the Ocaml type system level by distinguishing between a group of polymorphic variants: `identifier'` (blank or named identifier), `lvalue` (only assignable expressions) and `lvalue'` (assignable expressions and the blank identifier).

Pretty-Printer

Implementing a basic pretty printer is trivial in Ocaml. We simply implemented a group of mutually recursive functions which traverse the AST. The current indent level is passed as a parameter and increased as required.

Team Organization

- Michael
  - Scanner
  - Token printing
  - Pretty-Printer
  - Tests:
    - "Interesting" valid:
      - rev_string.go
      - is_palindrome.go

        * invalid:
            - simple_stmt_invinv.go
            - simple_stmt_opassign.go
            - simple_stmt_short_no_id.go
            - invalid_break.go
            - invalid_continue.go
            - invalid_continue2.go
            - var_block_comment.go
            - blank_selector.go
            - int_selector.go
            - simple_stmt.go
- Nicholas
    - Parser
    - AST
    - Core program
    - Build scripts
    - Weeding passes (Terminal)
    - Interpretation of the Go specification
- Justin
    - Parser
    - AST
    - Pretty-Printer
    - Weeding passes (Switch)