COMP 520 - Milestone 1 Report

Shafin Ahmed - 260782651
Nathaniel Branchaud - 260800960
Brendon Keirle - 260685377

February 23, 2020

1. Development Language and Tools:

The development language we chose for our project is OCaml. We chose OCaml for three main reasons. The first is that it is an language that all three of us were familiar with from COMP 302, it being very important to us to choose a language we were all comfortable with. The second is that OCaml has very strong pattern matching functionality, which we knew would greatly facilitate the design and traversal of our AST. The third was its reputation for producing programs whose execution times were roughly in line with programs written in C. It seemed like the perfect language to allow us to produce a fast compiler without giving up access to high-level data structures like strings and lists.

2. Team Organization:

In examining the GoLite specifications, we noticed that the language divided roughly evenly into three categories: declarations, statements, and expressions. With this in mind, we decided that if each of us did the scanner, parser, and AST/pretty-printer implementation for one of these sections, we would each have roughly the same amount of work. It was then decided that Nathaniel would cover the declarations, Brendon would cover the expressions, and Shafin would cover the statements.

3. Scanner Design:

Much of the design of our scanner was more or less straightforward with the exception of a few particular features. One such feature was the implementation of optional semicolons to terminate lines. Go allows user to optionally terminate lines in a program with a semicolon by having the scanner insert a semicolon into the token stream depending on whether or not a line ends with a particular token. To implement this feature, we defined a variable to keep track of the last token encountered by the scanner. Each time the token is encountered, this token is stores in the variable, then when the scanner encounters a newline character, it checks whether the last token was one of the ones requiring a semicolon to be inserted, and if it was it accordingly inserts a semicolon into the stream. Another difficulty that came up in the scanner was how to recognize all special characters in interpreted strings. The primary source of this problem comes from our decision to use Ocaml. When Ocaml sees what resembles a escape sequence in a string, it replaces backslashes with two backslashes if and only if the escape sequence is not known to Ocaml. An example of such coercion can be seen with the input string: ”Hi! \z \t ” which is coerced to: ”Hi! \\z \t”. Here, Ocaml recognizes ’\t’ and does not change the escape sequence, but ’\z’ is not an escape sequence in Ocaml and so a second backslash is automatically inserted in this case. The problem that arises from this is that we are meant to throw errors when an illegal escape sequence is found, but this coercion makes it impossible to know whether the user entered the valid escape sequence: ”\\i” or the invalid sequence: ”\i”. In the end this was solved when we realized that moving the matching of string contents to the lexer from a function allowed us to match the raw input with no interference from Ocaml.

4. Parser Design:

Our parser includes many recursively designed grammar elements. One of the most complex is the one covering variable declarations and assignments. Go allows users to declare or assign multiple variables in a sigle line by specifying them as a comma sepatated list. To implement this, we designed a recursive grammar element with a base case of a single identifier and corresponding expression, then allows identifiers separated by commas to be recursively added to the left-had-side of the element and similarly for expressions on the right-hand-side. This allows us to enforce the requirement that the number of identifiers on the left-hand-side equals the number of expressions on the right-hand-side without resorting to an additional check in the weeder.

5. AST Design:

Our main objective in designing our AST was to keep the number of node types low. In the end, we managed to keep this number down to three, statements, declarations, and expressions. This made sense since it (roughly) corresponds with the different kinds of grammar elements defined in our grammar, and because it (again roughly) corresponds with the way we divided the work among team members. Lists are used exptensively, with our AST being represeted by a list of statement nodes, and block statements (e.g. If or For nodes) having statement lists corresponding to the statements contained in their scopes. Another place lists are used is in declarations. When the user declares multiple variables using the single line grammar element described above, it is converted into a list of individual variable declaration nodes. The downside of this is that the variables are assigned expressions in reverse order because of the recursive definition of the grammar element. This is addressed by traversing through a reversed copy of the list and re-assigning variables in the original list based on the new reversed order of the expressions. The use of lists in our AST allows much of the AST to be traversed tail-recursively, which is very convenient in OCaml.

6. Pretty-Printer Design:

The readablility of the pretty-printer's output naturally drove our decision making when designing it. To this end, we defined an elaborate system to keep track of how much particular lines should be indented. This is achieved through a halper function which prepends tab characters to strings associated with AST nodes based of their indentation level. This level is determined by how many blocks deep the node is. Every time a new block node is reached, the nodes in this block's scope are indented at a level which is one higher than the block node's. The result of this is that scope is very clear in the output of the pretty-printer.

7. Weeder Design:

Our weeder simply goes through the AST recursively and when it enters a for loop or switch statement node, a reference variable 'loop' is incremented, and decremented when it exits the node. This allows break and continue statement nodes to be checked against this variable, and an error is thrown if the loop variable is 0, i.e, if we are not in a loop. A similar thing is done in dealing with default case in switch statements by keeping track of the number of default cases encountered so far and throwing errors as necessary. Lastly, we also weed out any invalid uses of blank identifiers by matching the blank identifier string in incorrect nodes and throwing errors if there is a match.