# Project Report - Milestone 1

# Group 7

Neil Pereira
Denali Relles
Yann Sartori

## Organization and division of work

We organized ourselves in the following manner : Denali wrote the scanner and the declarations and types portion (2.1 - 2.7) of the parser and AST; Neil wrote the statement (2.8) portion of the parser, AST, and pretty printer; Yann wrote the expression (2.9) portion of the parser, AST, and pretty printer along with the pretty printer portion of the work Denali did for the parser and AST (2.1 - 2.7). The reason we divided up this way is the crux of the work is in the parser and AST-- the pretty printer and scanner was just standard coding. Additionally, the statement portion of the assignment had the most subtleties and was deemed to be the most difficult. We then of course changed each others' portions as we saw fit for correctness or for ease of cohesion.

## Toolchain and language choice

We decided to use Flex, GNU Bison, and C as our languages for the primary reason being we already had great familiarity with them. However, they also had the added bonus of speed and strong interoperability, the first two being quite simple yet powerful, and C having the benefit of being versatile. The main downside of using C was the lack of objects and algebraic data types, which led to numerous verbose switch statements.

The only resources we consulted were the example programs in the Github, mainly the optional semicolons, the slides, and our work for the MiniLang compiler.

## Decisions made regarding the scanner

For the scanner, most decisions were straightforward. Reserved words and operators are each given their own token. Integer literals are all put under a single token, with hex and octal being converted to decimal integers. There are separate tokens for raw string literals and interpreted string literals. A decision of note concerns the handling of runes: we initially considered storing them as single characters, since they should fit in a single byte in all cases, but we decided it was easier to store them as strings (char*) because we would have to reprint them during the pretty printing and codegen phases, which would be easier if they were stored as strings. For the optional semicolons, we followed the pattern on the course github page: a global variable records the last read token, and then a newline character determines whether or not to insert a semicolon.

## Decisions made regarding the parser

There were many issues we faced during the construction of the parser. The most trivial of them was enforcing precedence within expressions, which was done by using Bison's precedence directives in accordance with the GoLang specifications, along with using expression unrolling (with primary expressions mainly) to ensure they have higher precedence. Another issue we encountered was the fact that blank identifiers couldn't be used in certain expressions and statements expecting a value (like the right hand side of assignments, function calls, switch statement expressions, etc), but other cases were permitted (like in the left hand side of assignments). Rather than duplicating rules wherein some were allowed to have blank identifiers, others not, we weeded them out in the parser prior to the construction of our AST node. Finally, there was an issue with distinguishing between function call and type casts. However, at this stage of the pass, since both can be any type of identifier, it is impossible to distinguish the two, hence we treat both as the same and will fix them at the symbolcheck phase, where we check the genre of the identifier. Additionally, for our expression lists, we used left recursion for LALR efficiency reasons. These lists are reversed when the AST nodes are created.

In the grammar, variable and type declarations are defined by their respective keywords followed by either declarations within parentheses or a list of identifiers and then possibly type information or value information. However, short declarations, due to parsing conflicts, are parsed as a list of expressions followed by the assignment token and then another list of expressions. We use weeding to ensure that those expressions are in fact identifiers, as well as to verify that the lists of expressions have the same length.

## Decisions made regarding the structure of the AST

Our AST we had 7 main types of nodes: expression and statement nodes; variable, type, and function declaration nodes; and a type holder and top declaration node. We decided to have these nodes as we believed that it would strike a good balance between nodes being too specific and having a lot of structures to account for in our AST traversal, and too vague and losing a lot of information. In addition to the attributes we have now, we fully expect to augment our nodes with, for instance type information for future phases of the compiler-- we just held off until we actually started implementing such phases-- as well as line numbers in *all* nodes to account for errors in future passes.

The root node has a special type, and it stores the package name, which is unique. It is always the root of the AST.

The program is comprised of top level declarations, forming the “backbone” of the AST. Each of these nodes points to either a function declaration node, a type declaration node, or a variable declaration node. Function nodes contain the expected information: name, argument information, return type, and a pointer to the block of code within. Variable and type declarations are always expanded as much as possible. “Distributed” declarations with parentheses are broken apart into separate nodes, and declarations with more than one identifier separated by commas are also separated out. We decided that this would not affect performance, because while this requires that the type information is duplicated, expressions cannot be part of type information (at least in Golite arrays must be created with integer literals) so this decision will have no side effects.

## Structure of expression nodes

Our expression nodes contain the kind of expression it is (which is just an enum), then a union which contains the values of interest for the more vague genre of expression it can be-- binary (which has a left expression and right expression), unary (just an expression), append (which has the base slice expression and the element to add), funcCall (which has the parameters as a list of expressions and the name of the function as an expression), and an expression for the body of built-in functions-- cap, len, etc. We also have the value for the literals and the identifier (which is just a string). As remarked earlier, our runes are stored as strings. In addition, casts and function calls are represented by the same node; proper usage is checked at a later phase.

## Structure of statement nodes

For the statement portion of the AST, the structure of the tree mostly followed the structure of the EBNF grammar as indicated on the Golang specification. The statement nodes consist of a line number, a field that tags each node with its kind, a union that has various structs to reflect the differences in various statement nodes and a pointer to the following statement as statements are represented as linked lists. A decision common to expression lists as well as to statement lists, was to parse them using left recursion for the reason of efficiency. This required us to include functions that would reverse a linked list once its construction was complete.

It was decided to weed out incorrect uses of blank identifiers during the parse phase as opposed to reflecting it in the grammar as it was easier to express and allowed us to give better error messages. If statements were designed to minimize the complexity of the tree; each if statement contained a pointer to an else statement (which was set to NULL to denote its absence). The else if statements were not represented as their own node, instead they were represented as an if statement inside an else branch. Each of the three loop variants were represented by their own special statement node type. The increment, decrement and compound operators ( += , *= for example) were all desugared and represented as regular assignment

statements in the AST. Switch case statements required the creation of a special struct type to represent switch case clauses.

The following issues are caught during a weeding phase after the construction of the AST:

Continue statements may only occur inside a loop, break statement may occur inside a loop or a switch statement and return statements must occur inside functions. The weeder function kept track of the scope of statements by using special arguments to determine whether a given statement was in a function, switch statement or a loop. Multiple default cases in switch case statements were also weeded at this stage.

## Issues encountered building the pretty-printer

The pretty-printer did not present many challenges. We liberally parenthesized expressions to ensure that the precedence rules of various operations were respected. Otherwise, for expressions, we carried out in-order traversal so as to reflect the infix notation of expressions. With regards to declarations with distribution (comma separated declaration and parenthesized declaration), we printed each on their own line. This is due to reasons mentioned above in the AST construction. We initially had made no distinction between short declarations and full declarations, however we later changed our minds as it was required to parse optional declarations in control statements as otherwise, the invariant property [pretty(parse(pretty(program))) == pretty(parse)] would not have been satisfied. Similarly, the statements portion of the AST did not pose a significant challenge and the pretty printer just followed the structure of the statement nodes.