GoLite Deliverable 1: Report

The rationale of the implementation tools and language;

For the GoLite project, we chose to use the C programming language, and the flex (fast lexical analyzer) and bison toolchains. We chose these tools because having prior experience with them allows us to implement compiler features quickly. Given the aggressive timeline of our projects' deadlines, being capable of delivering our features in a timely manner is one of our core team doctrines. The second dogma on our list of tenets is efficiency. As a metaphor to the tenet, consider Drake's thoughts on sluggish behavior: "better late than never, but never late is better." This highlights our thoughts that the GoLite compiler should have low compile time. C is a natural choice since it has less overhead and better performance time than higher-level programming languages. Implementing the compiler in a programming language such as Java or Python would result in slow compilation performance. Our final two doctrines are simplicity and correctness. Implementing programs in C forces the developer to write simple functions and data structures.

Team organization;

For the sake of efficiency, we adopted a random-round-robin dictatorship style approach to team dynamics. At random intervals of time, one of the team members is selected at random to take on a leadership role in the team. This leader has the final say in each and every decision.We adopted an agile methodology to development. Each team member takes on a compiler feature that they would like to work on and plan out how their work can be done in parallel. We used GitHub's Pull Request, and Issue tracking features to help review code and organize tasks. We also revised the AST and parser as we discovered issues while weeding and pretty printing. For any bugs or fixes, we discussed them in-person or on Facebook, created a GitHub issue, and assigned the person most familiar with the code to write a fix.

Braedon:

- Designed and implemented parser
- Designed and implemented pretty print

Matthew:

- Designed and implemented parser
- Designed and implementing all weeding cases

Sophia:

- Designed and implemented scanner
- Designed and implemented AST

Resources consulted;

Flex documentation for C-style commenting: https://www.cs.virginia.edu/~cr4bd/flex-manual/How-can-I-match-C_002dstyle-comments_003f.html#How-can-I-match-C_002dstyle-comments_003f

Design and Implementation Decisions

Scanner

To implement the scanner, we followed the language specification. There weren't many design decisions to make, and the scanner's implementation was similar to the scanner written for minilang, with the interesting notes:

Optional Semi-Colons

Semi-colons were inserted if necessary, given the lsat token, if the scanner encountered either a new line or EOF. The rules for inserting a semi-colons were in the Golite specification. In addition, there were instructions on how to correctly implement semi-colons in the COMP 520 repository.

Multi-line comments

The multi-line comments are handled by consulting the flex documentation for C-style comments.

Strings
We allowed any character inside the "" except for the unescaped string literal.
Rune literals
If a rune literal was a valid escaped character, like \n, we set \n as the value in yylval.runeval.
Helper File to Avoid Copy-Pasting
Fun fact: we made a helper file in Java so we didn't have to copy-paste too many tokens.
For example, to generate the below code where x is some token name:

```
x { if (g_tokens) printf("tX\n"); RETURN(tX); }
```

We made a helper function that takes in x and prints the above code to the console so we can copy-paste it into our lexer! Saved quite a lot of time since there were 40+ reserved words in Golite.

Parser

For implementing our compiler's grammar, we consulted the GoLang program specification and attempted to replicate its grammar specifications as closely as possible. The largest difference in our grammar is with the expression rule:

```
expression: binaryExpr
      | unaryExpr
      | builtinExpr
      | functionCallExpr
      | expression tLBRACKET expression tRBRACKET
      | expression tPERIOD tIDENTIFIER
      | tINTVAL
      | tFLOATVAL
      | tRUNEVAL
      | tSTRVAL
      | tIDENTIFIER
      ;
```

Whereas the authors of GoLang defined a primaryExp non-terminal and an expression non-terminal separately, we merged the two into one called expression which can be viewed above. Therefore, our grammar definitions for expressions and statements that involve expressions differ slightly from what's written in the official GoLang language specification. Whereas the authors of Go defined an operand non-terminal which make up binary and unary expressions, we simply make up binary and unary expressions from expression non-terminals.
Additionally, for lists of expressions, or identifiers, we've written non-terminals such as expressions and identifiers that contain rules that recursively call on themselves or on a single expression/identifier. A code snippet below shows the two non-terminals and their rules:

```
expressions: expression
      | expressions tCOMMA expression
      ;
identifiers: identifier
      | identifiers tCOMMA identifier
      ;
```

When certain non-terminals, such as assignment statements and function calls, allow for multiple expressions or identifiers, we use the expressions or identifiers non-terminals respectively.
We used an identical logic and similar rules for type and variable declarations that can involve multiple types and variables respectively, as well as top-level declarations.

```
topLevelDecls: /* empty */
      | topLevelDecls topLevelDecl
      ;
topLevelDecl: variableDecl
```

```
    | typeDecl
    | functionDecl
    ;
typeDecl: tTYPE typeSpec
    | tTYPE tLPAR typeSpecs tRPAR tSEMICOLON
    ;
typeSpecs: /* empty */
    | typeSpecs typeSpec
    ;
     <etc.>
```
Statements are handled similarly to how they are handled in the Golang specification where simple statements are a subset of statements.

If statements are handled using a recursive strategy as discussed in class so that we don't have to keep track of how many else-if branches there are

```
ifStmt: tIF simpleStmt tSEMICOLON expression block
        | tIF simpleStmt tSEMICOLON expression block tELSE block
        | tIF simpleStmt tSEMICOLON expression block tELSE ifStmt
        | tIF expression block
        | tIF expression block tELSE block
        | tIF expression block tELSE ifStmt
        ;
```

AST

When desigining the AST, we largely followed the structure of the grammar. We also wanted to create as simple and minimal an AST as possible.

Handling Lists

An interesting component of GoLite is that it allows declaring multiple identifiers or expressions in the same line. To handle such constructs, the AST uses a linked list structure of other AST nodes. An example code snippet follows:

```
struct IDENT {
    char *ident;
    IDENT *next;
};
```
The corresponding token in the parser would create the linked lists:
```
identifiers:
    identifier { $$ = $1; }
    | identifiers tCOMMA identifier { $$ = $3; $$->next=$1; }
    ;
```
AST Structure

The program AST node PROG contains:

- One package identfier
- A linked list of top level declarations, TOPLEVELDECL

Top level declarations are one of:

- Variable declarations VARSPEC;
  - Since a variable declaration can be distributed over multiple lines, the AST will parse this as a list. VARSPEC has a next pointer that handles a list of variable declarations.
  - A single variable declaration can have multiple identifiers and expressions, so both the IDENT and EXP AST nodes have next pointers.

- Type declarations TYPESPEC;
    - This is similar to VARSPEC, except each identifier will have a TYPE. The TYPE node can be a slice, array, struct, or an inferred type that we will later determine in the typechecker.
    - It also contains a TypeSpecKind to differentiate between a type declaration, and a function parameter list. Instead of having a new AST node to represent a function parameter list, we wanted to simplify the AST and use the TYPESPEC AST node.
- Function declarations FUNC;
    - Each function has a function name, optional return TYPE, a TYPESPEC of input parameters, and a root statement STMT (which is a linked list structure).

We separated the AST into main nodes:

`PROG, TOPLEVELDECL, TYPE, STMT, EXP, FUNC`

and helper structures:

`IDENT, VARSPEC, TYPESPEC, STRUCTSPEC, EXPRCASECLAUSE,`

Statements

Because of the numerous different statement types in Golite, we created separate stmt.h and stmt.c files to modularize the code.

1. *"Simple Statements"*

The concept of "simple statements" don't exist in the AST. Simple statements are just statements for simplification.

2. *Reducing Kinds of Statements*

In some cases, we were able to simplify the STMT node by making two similar kinds of statements into one kind of statement with an extra piece of information.

For example, instead of having two different statement kinds for an increment and decrement statement, we made them the same statement kind k_stmtKindIncDec. In the value of the statement, there is an amount attribute that differentiates an increment or decerement. An increment will have an amount 1, whereas a decrement will have an amount -1.

Another example is with print and println. We used an newLine flag to differentiate the two. A println statement is just a print statement with the newLine flag set to 1.

3. *Reducing Amount of Repetitive Code*

We observed that for when creating each statement node, we were calling malloc(sizeof(STMT)), setting the line number, and setting the statement kind. To reduce the amount of repetitive code written, we made a helper function getGenericStmt() that takes in a StatementKind.

Therefore, all other functions will call getGenericStmt with their corresponding StatementKind type, then set their unique attributes.

Switch Statements

We created a helper data type EXPRCASECLAUSE to handle case clauses in switch statements:

```
struct EXPRCASECLAUSE {
    int lineno;
    CaseClauseKind kind;
    EXP *expList;
    STMT *stmtList;
    EXPRCASECLAUSE *next;
};
```

The expression list is the condition that is mapped to the case clause, and the statement list is the list of statements that follow. CaseClauseKind include k_caseClause and k_defaultClause to simplify work in weeding, when the weed needs to make sure each switch statement has a default clause.

Expressions

Expressions in Golite are handled very similarly to Minilang. For example, every binary expression has a type, but contains the same lhs and rhs values.
There is also a helper function getGenericExpr() in our AST to reduce the amount of repetitive code written.

Weed

Here, we'd like to summarize our weeding features and our rationale for making it (apart from its implementation being a requirement for the project). The rationale behind our weeder is that we'd like a weed phase than can remove as many illegal statements and expressions as possible to save us work in downstream tasks. The weeder's most important feature is its ability in dealing with blank identifiers, since we'd like to avoid dealing with blank identifiers in expressions that evaluate to a value in the type check phase.
The additional features we've implemented in the weeder include:

1. Verifying whether a post-statement in a for loop doesn't contain a short declaration;
    a. By checking the post-statement STMT AST node's StatementKind.
2. Verifying that a break statement is within a for-loop or a switch statement block;
    a. By keeping a count of the depth of previous for-loop or switch statements, and checking that the current break statement is within a depth of greater than 0.
3. Verifying that a continue statement is within a for-loop block;
    a. Similarly to rule feature #2, but we only verify that the current continue statement is within a positive depth of previous for-loop blocks.
4. Verifying that expression statements are only function calls;
    a. By checking the expression EXP AST node's ExpressionKind.
5. Verifying that an assignment statement contains the same number of identifiers on the left-hand-side of the assignment operator as expressions on the right-hand-side of the assignment operator;
    a. By simply counting the number of identifiers and expressions in an assignment statement, and verifying that their counts match.
6. Verifying that there is exactly one identifier and one expression in an assignment operation;
    a. Similarly to rule #5, counting the number of identifiers and expressions and checking they both count to 1.

The reason we developed these features is because they enforce rules for expressions and statements that are simply illegal in the GoLite language. Whenever one of these checks don't hold true, the weeder prints a descriptive error to the console and quits compilation. The main design choice that we'd like to discuss is how we weed expressions differently depending on whether the expression should evaluate to a value or not. We implemented rules for the former in a function called weedEXP_eval and rules for the latter in a function called weedEXP_nonEval. We separated the two functions since the linguistic rules for expressions that evaluate to values differ greatly from expressions that don't evaluate to a value.
When weeding expressions that should evaluate to a value, we make sure that the blank identifier _ cannot be used as a value for any binary, unary, field access, array access, function call, or builtin function call expression.
As for expressions that don't evaluate to a value, such as expressions on the left-hand side of an assignment statement, we implemented rules that only allow the expression to either be an identifier, an array access, or a slice access. Additionally, in operation assignments that don't include the := and = operators, we implemented a rule that doesn't allow the left-hand-side operand to be a blank identifier _ , otherwise blank identifiers are allowed.

Pretty Printer

Pretty printing is done by traversing the AST to output a valid GoLite program. Special care was required when indenting and ending the lines of statements. If, switch, and for statement nodes have children that are statements that should not be indented or be the end of a line. The pretty printer handles this by not indenting or ending the line when the parent node is one of the previously mentioned statements.