Milestone 2

Team Members

- Michael Sukkarieh
- Nicholas Paun
- Justin Tremblay

Design Decisions

Parser/Lexer Fixes

The first thing that had to be done for this milestone was to fix the parser/lexer to ensure that it passed all the tests that were given after the first milestone. The biggest problem had to do with builtin function calls and due to the complexity of the rules surrounding them, two short weeding functions were added to the parser to ensure that:

1. Calls to builtins have the right number of arguments. (see `weed_builtin_call` in `parser.mly`)
2. Builtins are **not** used as expression statements. (see `weed_exprstmt_funcall` in `parser.mly`)

Other small issues had to do with short variable declarations and various small things but those were fairly easy to fix.

Symbol Table Generation

In order to generate the symbol table, we implemented a symbol table module (see `symtbl.ml`) that contains the symbol table and symbol types and various other utility functions to manipulate them. The file also contains the code for recursively traversing the AST, throwing errors, and pretty-printing the symbols.

Symbol Table

The symbol table is implemented in a very conventional way; it consists of a hash table of symbols, mapped to their names. It also holds references to its parent and children aswell as it's depth, the global scope having a depth of 0. The depth is primarly used for pretty-printing the symbol table, as it controls how many tabs are printed.

Symbols

The symbol implementation is also very straight forward; it consts of a name, the kind of symbol (variable, typedef, function or constant), and its type. The type of the symbol is implemented as a list of types in order to support function signatures. Symbol entries for variables and types just have a single entry in this type list, while symbol entries for functions have a type list in the form of: [return type; arg1 type; arg2 type, etc]. This allows for easy pretty-printing of

the function signature and make the job of the typechecker. Lastly, it should be noted that all of the fields in the symbol struct are mutable, which allows for the symbol's to be modified in later stages of the compiler (ex. typechecker when it performs type inference)

Global Symbol Entries

A set of symbol entries are added into the root symbol table at the start of symbol generation. These symbol entries include all of GoLite's primitive types (int, float64, bool, rune, string), true/false constants, as well as the built-in functions (append, len, and cap). The actual program is contained within a symbol table which is the child of this global symbol table.

Scoping Rules

For the most part, most of the scoping rules just involve adding/getting symbols from the same scope. However, there are some notable statements that require careful handling of symbol table scoping. Some of these rules are noted below.

Functions

Functions are toplevel declerations, and are composed of an identifier, signature list and a block. Firstly, we add an entry into the program's symbol table (the one under the "global" symbol table exaplined in the Global Symbol Entries section above) for the function's identifier. Next, we scope a new symbol table and add in all of the parameters, followed by adding all of the statements within the block. This ensures that the function identifier is visible from the program's symbol table but the parameters and contents of the block are not. ###### Invalid main/init check According the GoLite spec, function(s) with the name "main" or "init" cannot have parameters or a return type. This is enforced in our compiler by doing a check to ensure that no signatures exist and that the function's type is `VOID before any symbol entries are added. If this check passes, we through an error stating that the special function must be void and have no parameters. ###### Multiple init/blank funcs GoLite supports multiple definitions of blank functions as well as functions named "init". Since we use a hashtable to map the function's name to it's symbol, we prefix the name of the function with "$" and postfix it with a number. The postfixed number is the current count of unbounded functions (init/blank funcs are not bounded). This allows us to have an entry in the symbol table's hashtable for each of the unbounded function definitions. Lastly, we set the type list of these functions to` [AUTO] to indicate that it is not bounded (this is used in the pretty-printer and typechecker).

If and Switch Statements

Special care was taken to ensure that the if and switch statements scoped properly. The main difficulty here lies in the fact that each case can declare new variables, which can be used in following cases. To implement this, we first scope a new symbol table for the entire if/switch statement, let's call this *outer_scope*. Next, we iterate through each of the cases (if and switch statements use the same case ast node), and do the following: 1) add a symbol entry for the case's statement into *outer_scope* 2) scope a new symbol table for the case's block and go through the block 3) unscope the block's symbol table and continue to the next case.

```
if x := <expr>; <cond> {
    // block scope, x is accessible
} else if y := <expr>; <cond> {
    // block scope, x and y are accessible
}

// equivalent to:

{
// outer scope
x := <expr>
if <cond> {
    // block scope, x is accessible
}
y := <expr>
else if <cond> {
    // block scope, x and y are accessible
}
}
```

For Statements

For statements are a much simpler case of the if/switch statement scoping rules. Here we still make sure to start with scoping a new symbol table, here we call it *for_scope*. We do the usual symbol adds/checks for the stmts and expr contained within the for statement using *for_scope*. Next, we scope a new symbol table for the block and we go through it. We finish by unscoping the block symbol table and then *for_scope*. ##### Short Variable Redecleration GoLite allows for redeclerations of variables within short variable declerations statements as long as there is a new variable being declared within the statement. This is implemented within our compiler by first scanning through the list of identifiers in the short var declarations and checking to see if there are any new variables (i.e. if there exists a non blank identifier which is not in the current scope). If so, then we simple ignore the identifiers in the short var declaration which already exist. ### Type Checker

The implementation of the type checker is very straight forward; it takes the AST

and the previously generated symbol table as inputs and performs a traversal of the AST, recursively type-checking the nodes following the rules in the language specification. When it encounters a variable, type name, or function call, it performs a lookup in the symbol table in order to resolves its type (or signature for functions).

While performing the type-check phase, the type-checker also performs type inference when needed and annotates the AST node with their type information, as long as they are well-typed.

Invalid Programs

- 1-2-1-defined-types.go : In order for two types to be identical, they must have the same underlying type, unless they are user-defined, then they must be the exact same type.
- 2-1-1-bad-expr-type.go : When declaring a variable, the RHS expression's type must be identical to its declared type if it is present.
- 2-1-2-decl-defined-type.go : When declaring a variable of a user-defined type, the expression cannot be of its underlying type, it must be cast to the user-defined type.
- 2-2-alias-not-resolved.go : Even though two types may be aliases of each other, they are not assignment-compatible.
- 2-2-type-shadow.go : Once int is shadowed by a different type, it cannot be used with int literals.
- 2a-reassign-constant.go : While you can shadow a constant, you can't assign to it.
- 2-3b-lisp-1.go : Golite is a Lisp-1 (one name space for functions and variables).
- 2-3-5-bad-func-call.go : types are only identical if they point to the same type spec (1.2.2)
- 3-4-1-return-noexpr.go : A function declaration is ill-typed if it has a specified return-type but returns no expression.
- 3-4-2-return-expr1.go : The type of a returned expression must match the function's specified return-type.
- 3-5-3-short-var-decl-partial-redeclare-to-different-type.go : A variable cannot be redeclared to a different type using a short variable declaration.
- 3-12-2-if-init-bad-type.go : In order for an if-statement to be well-typed, its init-statement must also be well-typed.
- 3-12-3-if-expr-bad-type.go : In order for an if-statement to be well-typed, the expression in its condition must resolve to type `bool`.
- 3-13-1-switch-bad-init.go : In order for a switch-statement to be well-typed, its init-statement must also be well-typed.
- 3-12-2-switch-bad-expr.go : In order for a switch-statement to be well-typed, its expression must also be well-typed.
- 3-13-4-switch-bad-case.go : In order for a switch-statement to be well-typed, all of its cases' expressions must also be well-typed and resolve to

the same type as the switch-statement's expression.
- 4-1-intlit-not-promoted.go : you cannot assign an int to a variable of type float64.
- 4-1-lit-no-aliases.go : A literal is not assignment-compatible to an alias of its type.
- 4-5-funcall1.go : In order for a function call to be well typed, all the arguments' types must match the function's formal parameters' types.
- 4-5-funcall2.go : In order for a function call `(<expr>)(a1, ..., an)` to be well-typed, the expression's type must resolve to a function type, in this case, it must be a function name.
- 4-6-1-indexing-badindex.go : When indexing, the index must be well-typed and resolve to int.
- 4-6-2-indexing-badexpr.go : An expression that is being indexed must resolve to type array or slice.
- 4-8-1-append-expr1.go : The first argument of append myst resolve to a slice type.
- 4-8-1-append-expr2.go : The second argument of append must resolve to the same type as the slice's elements' type.
- 4-8-1-append-return.go : The variable to which the result of append is assigned must be the exact same type as the slice that is used as argument.
- 4-8-2-cap-expr.go : The expression in cap must resolve to either an array or slice type.
- 4-8-2-cap-return.go : The result of cap is of type int, cannot assign to non-int assignable values.
- 4-8-3-len-expr.go : The expression in len must resolve to either an array or slice type.
- 4-8-3-len-return.go : The result of len is of type int, cannot assign to non-int assignable values.
- 4-9-float64-into-string.go : float64 cannot be cast into string
- 4-9-float64-into-string2.go : float64 cannot be cast into string (even through indirect types)

Team Organization

- Michael
  - Symbol table implementation
  - Symbol table building
  - Symbol table printing
  - tests
- Nicholas
  - Symbol table implementation
  - Type checking
- Justin
  - Symbol table implementation
  - Symbol table building
  - Parser/Lexer fixes from previous milestone

  - Weeding passes for builtin functions calls
  - tests

References

- ossamaAhmed/GoLite-compiler
  - Ideas for parser/lexer fixes