Milestone 1 Report

1. Framework

We are developing our compiler with a Haskell based toolchain.

- Alex
- Happy
- argparser

2. Team Work

Yi Qiao - Developed the Scanner - Contributed to the development of the Parser and the Weeder - Wrote tests for the Scanner and the Parser

Thomas - Wrote the majority of the Parser - Wrote the Pretty Printer - Wrote most of the [Language](../src/Language.hs)

Charlie - Developed the test suite and automated testing (programs/) - Wrote the Weeder - Wrote the compiler wrapper and the CLI code

3. Scanner

The scanner was written using a standard Alex setup. Some macros were defined for certain character sets to improve ease of use of these sets later on in the scanner. Tokens are recognized via regexes and returned in their respective productions.

One of the main design decisions taken in the scanner was to explicitly give everything its own token, in as detailed a way as possible. For example, one can define a token, `BinaryOp`, for all binary operations and store the associated characters with `BinaryOp`. This does eliminate some initial code repetition, but it introduces plenty of case matching in further stages. While the decision to assign each binary operation its own token has an initial cost, it reduces the cost of all subsequent uses of binary operations.

Another design decision made in the scanner was to recognize and split up the three different integer representations, assigning each its own token, and parsing all as decimal for use in the rest of the compiler. The decision to split up the integers in the scanner follows the same philosophy as the previously mentioned design decision; we would rather take an initial one time cost and save on all subsequent uses of integers, instead of saving on a one time cost and having more complexity on every use.

The decision to use a monadic scanner is twofold. Coupled with the monadic parser, it allowed us to generate reasonable error messages. Furthermore, the state monad was used to insert optional semicolons into the token stream wherever the conditions to do so were satisfied. This was acheived by storing the previous token and, on newlines, checking to see if the previous token matched one of those outlined in rule 1.

4. Parser

To make the parser more readable, the tokens were given appropriate aliases, usually just using their names in Go. Precedences were created to allow us to put all the expressions into one data type, without having to seperate them into factors and terms.

A Program data type was used to describe the entire program. It's seperated into the package declaration followed by a list of statements (declarations included as statements). The statement data type was split into three types:

1. All - This can either be any statement, or it can be a function declaration. This is to make it impossible for functions to be declared inside blocks. Only the program data type has a field for an All list.
2. Stmt - This data type contains most of the statement options. It includes if statements, loops, switch statements, variable and type declarations, return statements, print statements, breaks and continues.
3. SimpleStmt - This data type contains statements that can be used before blocks in for loops, if statements, and switch statements. This includes expression statements, increment and decrement statements, assignment, short variable declarations, and op-assign statements.

If statements were also given their own data types, to allow them to recursively add if statements like so:

```
if x < 0 {
} else if x < 1 {
} else if x < 2 {
} else if x < 3 {
} else {}
```

Identifiers were given a data type where they could either be a regular identifier (represented by a string), an indexed identifier for arrays and slices, or an identifier with field access for structs.

Types were also given a data type where they could either be a regular type (represented by a string), an array type, a slice type, or a struct type.

A few other data types were created as well for parameters in functions, literal values, clauses in switch statements, variables for variable declarations, and typenames for type declarations.

5. Weeder

The weeder is responsible for verifying the syntactic validity of a small set of constructs that are difficult to verify directly in the parser's CFG. We defer the following verifications to the weeding phase.

- Verifying that there is at most 1 `default` clause per `switch` statement.
- Verifying that there are an equal number of identifiers and strings on each side of a multiple assignment operation.

```
var a, b = 1, 2, 3;
a, b := 1;
```

- Verifying that no identifier is repeated in a multiple short variable declaration.

```
a, a := 1, 2;
```

- Verifying that `break` and `continue` statements only occur inside of loop constructs.

```
func main() {
  break;
  continue;
}
```

- Verifying that the post statement in a for loop is valid.

```
for i := 0; i < 10; i := 20 {}
```

- Verifying that blank identifiers are used correctly

```
a[0] = a[1 + _]
```

6. Pretty Printer

In the pretty printer we used a typeclass in order to automatically generate a function that can pretty print a list of instances of the typeclass given a function that can pretty print a single element of that instance. Then we recursively defined the pretty print function for each data type, concatenating all the results into a single string that could be written to a file.

In order to keep track of the indentation, an integer was passed along to every data type that recorded the number of indents those statements should have. The indentation was then assigned to the appropriate statements.