CMSC 330 Project 2

The second project involves completing and extending the C++ program that evaluates statements of an expression language contained in the module 3 case study in the week 5 module reading.

The skeleton code for this project is attached. It differs slightly from the what is provided in the case study. The code in the case study allows only one expression to be input from the keyboard whereas the code in the attached skeleton accepts input from a file named `input.txt`. That file contains one statement per line.

The statements of that expression language consist of an arithmetic expression followed by a list of assignments. Assignments are separated from the expression and each other by commas. A semicolon terminates the expression. The arithmetic expressions are fully parenthesized infix expressions containing integer literals and variables. The syntax of a single input file line is described grammar below:

```
statement → expression ',' assignments ';'

expression → '('  operand operator operand ')'

operator → '+' | '-'

operand → literal | variable | expression

assignments → assignments ',' assignment | assignment

assignment → variable '=' literal
```

In the above grammar, terminal symbols are upper case names or character literals shown in blue and nonterminal symbols are lower case names shown in red. EBNF metacharacters are shown in black. Tokens can be separated by any number of spaces.

Tokens can be separated by any number of spaces. Variable names begin with an alphabetic character, followed by any number of alphanumeric characters. Variable names are case sensitive. The regular expressions defining the variables and literal tokens are the following:

```
variable    [a-zA-Z][a-zA-Z0-9]*

literal     [0-9]+
```

The program reads in the arithmetic expression and encodes the expression as a binary tree. After the expression has been read in, the variable assignments are read in and the variables and their values of the variables are placed into the symbol table. Finally the expression is evaluated recursively.

Your first task is to modify the program so that it will parse additional types of expressions defined by the expanded grammar shown below with the additions to the grammar highlighted in yellow:

```
statement → expression ',' assignments ';'
```

```
expression → '('  expressions ')'

expressions → unary_expression | binary_expression | ternary_expression |
  quaternary_expression | operand

unary_expression → expression '~'

binary_expression  → expression binary_operator expression

binary_operator → '+' | '-' | '*' | '/' | '%' | '^' | '<' | '>' | '&'

ternary_expression → expression '?' expression expression

quaternary_expression → expression '#' expression expression expression

operand → literal | variable | expression

assignments → assignments ',' assignment | assignment

assignment → variable '=' literal
```

The semantics of the additional binary arithmetic operators are as follows:

- `*` Multiplication
- `/` Division
- `%` Remainder
- `^` Exponentiation

Although two of the three additional binary operators are customarily relational operators in most languages, that is not true in this language. The semantics of all three of those operators are as follows:

- `<` Minimum (Evaluates to the minimum of the left and right operand)
- `>` Maximum (Evaluates to the maximum of the left and right operand)
- `&` Average (Evaluates to the average of the left and right operand)

The single unary operator `~` is the negation operator. Unlike the unary minus in most languages, it is a postfix operator rather than a prefix one.

The single ternary operator `?` is the conditional expression operator. Unlike the conditional expression operator in C++ and Java, no colon separates the second and third operands. This expression is evaluated as follows. If the expression to the left of the operator `?` is not 0, the value of the expression is the value of the first expression after the operator `?`. If it is 0, the value of the expression is the value of the second expression after the operator `?`.

The single quaternary operator `#` is a variation of the typical conditional expression operator. Like the ternary conditional expression operator, the remaining three operands are delimited only by whitespace. This expression is evaluated as follows. If the expression to the left of the operator `#` is less than 0, the value of the expression is the value of the first expression after the operator `#`. If it is equal to 0, the value of the expression is the value of the second expression after the operator `#`. If it is greater than 0, the value of the expression is the value of the third expression after the operator `#`.

The second task is to modifier the variable token so that underscores are permitted in all but the first character and modify the literal token so that it accepts unsigned floating point literals. Assignments also should be modified to allow assignment to values that are should also floating point rather than just integers.

The final task is to make the following modifications:

- The symbol table should be initialized before each statement is evaluated, so that variables that are reused do not contain the value from a previous statement
- Statements containing uninitialized variables should be reported as an error
- A variable initialized more than once in a statement should be reported as an error (Creating an exception class to accommodate this error and the previous one is the recommended approach)

You may assume that all input is syntactically correct. No checks for syntax errors is required.

Each new class must be in a separate `.h` and `.cpp` pair of files. If all the functions in a class are one line functions, they can be implemented inline in `.h` file and the `.cpp` file can be omitted.

The deliverables for this project include the following:

1. A `.zip` file containing all the source code correctly implementing all required functionality.
    a. All the `.h` and `.cpp` files provided in the skeleton should included regardless of whether they required any changes
    b. All new files must include a header with the student name, date, project and a description of what the file contains
    c. All modified files must include a header with the same information
2. A Word or PDF file that contains the following:
    a. A discussion of how you approached the project
    b. A test plan that contains test cases that include all additional images and test any new functionality. For each test case, the output produced should be included.
    c. A discussion of lessons learned from the project and any improvements that could be made