NINC INTERPRETER
CSC 306 FINAL PROJECT

DR. GODFREY C. MUGANDA

This project will put the finishing touches on the NINC interpreter we have been writing.

1. Keep Track of Non Fatal Errors

If you have not already done so, add the following variable to your `Parser` class in the `parser.h` file:

```
int typeErrorsCount = 0;  // Number of  static semantics errors
```

Next, modify the error-reporting function

```
// Error reporting functions
// Reports error, but does not exit.
void Parser::errorMessage(string error)
{
    typeErrorsCount ++;
    listing << "^^^^" << error << endl;
}
```

so it increments the count for the number of non-fatal errors each time such an error is reported.

The idea is that we will proceed to the execution phase only if there are no errors in the program.

2. Add an Execute Member Function to the `RunTime` Class

Add a static member function `execute()` to the `RunTime` class:

```
class RunTime
{
public:
    static  STRTABLE strTable;
    static  VARTABLE varTable;
    static  STMTTABLE stmtTable;
    static int pc; // program counter

    static void execute()
    {
        while (true)
        {
           STMT *IR = stmtTable.get(pc);
           RunTime::pc ++;
           IR->execute();
```

```
            }
        }
    };
```

This function was discussed in class: it stays in a loop, fetching the statement at the program counter (pc) and executing it.

3. Add code to `program()` to execute the code in the statement table

At the end of the `program()` parsing function to check the `typeErrorsCount` variable for errors. If this variable is zero, call the `execute()` function on the statement table; otherwise; print error messages to both standard output and the listing file.

4. Implement The Various Statements

Follow the following steps to complete the interpreter:

(1) Implement the evaluation of expressions.

All expressions other than `BinaryOpEXPR` have already been implemented. Implement the `eval()` method of `BinaryOpEXPR`.

(2) Implement the output statement.

To do this, you will need to use `dynamic_cast` to determine the exact class of each EXPR so you can print the appropriate value.

(3) Implement the exit statement.

This is implemented by calling the C++ library function `exit(0);`.

(4) Test with the `helloWorld.txt` test file.

At this point we have enough of the interpreter written to run the first test file. Test with `helloWorld.txt` test file provided by the instructor.

(5) Implement the input statement. Test with `input.txt`.

(6) Implement the assignment statement. Test with `assignment.txt`.

(7) Test the implementation of expressions using `expr.txt`.

(8) Implement the goto statement and the Conditional goto statement. Test with `ifstatement.txt` and `whilestatement.txt`.

(9) Test the break and continue statements in the context of a while loop using `primes.txt`.

(10) Implement the switch statement and test with `switchstatement.txt`.

5. Grading

Grading will be based on the number of test files that your interpreter executes correctly. It is better to have an interpreter that does not implement all the statements, but executes correctly on those that are implemented, than to have an interpreter that implements all statements but executes incorrectly on all or most of the statements.

sectionDue Date.

Saturday of Week 10 at midnight. Due to the end of the term, this is an absolute deadline, and nothing handed in past this time will be graded.

When you submit, make sure you go into your sent folder and unzip the submission to verify that what you submitted is good. You will not have a chance to resubmit after the deadline if your submission turns out to be empty, corrupted, or incorrect.