CSC 306 EXECUTION PHASE

GODFREY MUGANDA

1. Project 4 Overview

This project will complete the compiler/interpreter that we have been working on. Here is an outline of what you have to do:

1. Add **exit** and **newline** statements. This is equivalent to adding productions

STMT ::= **exit** | **newline**

to the grammar. You will need to make appropriate changes to the lexical analyzer, parser, and internal representation.

The **exit** statement is equivalent to calling

```
exit(0);
```

in C++, and the **newline** statement is equivalent to the statement

```
cout << endl;
```

in C++.

You will need to modify the compiler phase so that a representation of the exit statement is loaded as the last statement in the instruction table.

2. Add a global variable

```
int non_fatal_error_count = 0;
```

to keep track of the number of non-fatal errors. Best way to proceed is to add a

```
void error(const string &message)
```

method to the parser or compiler class and use this method to report all non-fatal errors to the listing class. This method will increment the variable for the number of fatal errors everytime it is called.

3. Modify the compiler to prohibit the occurrence of **continue** and **break** statements outside of loops (**loop** or **while** statements). Occurrences of break and continue outside a loop will be flagged as a non-fatal error.

4. Implement the `get_value()` methods for all subclasses of `cmm_super_expr`. For the `cmm_string_expr`, the `get_value()` method returns the index of the string in the string table.

5. Implement all the `execute()` methods of the subclasses of `cmm_stmt`. Each of these methods will be responsible for executing a single statement taken from the instruction table.

2. Other Functions and Data Structures

You will need the following data structures (and one function) in the `CmmVirtualMachine.cpp` file:

```
// Global variables
vector<string> string_table;
vector<cmm_stmt *> instruction_table;
cmm_variable_table  variable_table;
int cmm_pc = 0;

// positions of top of loop in the instruction table
stack<int> loop_top_stack;
// stack of lists of break statements that appear in a loop.
stack<vector<break_stmt *>> break_stmts_stack;

int non_fatal_error_count = 0;


void cmm_execute()
{
    if (non_fatal_error_count > 0)
    {
         throw "CMM program has errors.\n";
    }
    while (true)
    {
       cmm_stmt * p_current_stmt = instruction_table[cmm_pc];
       //cerr << "executing statement at " << cmm_pc << endl;
       cmm_pc ++;
       p_current_stmt->execute();
    }
}
```

These should all make sense to you because they were discussed in class. The variable

```
int non_fatal_error_count
```

keeps track of the number of non-fatal errors: the code will not be executed unless this variable is equal to 0 at the end of the compilation process.

The `execute()` method seems to have an infinite loop; but in fact, the program will terminate because execution of the `exit_stmt` will cause the interpreter to exit.

3. Declarations for the header file

In addition to the above, you need the following declarations in the `CmmVirtualMachine.h` file:

```
/**
 * Declarations of the string table, variable table,
 * and instruction table, and program counter variable
 */
```

```
extern vector<string> string_table;
extern cmm_variable_table  variable_table;
extern vector<cmm_stmt *> instruction_table;

 // positions of top of loop in the instruction table
extern stack<int> loop_top_stack;

class break_stmt;
extern stack<vector<break_stmt *>> break_stmts_stack;
extern int non_fatal_error_count;
extern int cmm_pc;

 // execute the internal representation
extern void cmm_execute();

// declarations of functions to print the above tables
void string_table_print(ostream &);
void variable_table_print(ostream &);
void instruction_table_print(ostream &);
```

4. Test Files

A sample of test files will be posted at the course web site to get you started with testing. As always, however, exhaustive testing is your own responsibility.

5. Due Dates

This is due the Friday night at the end of week 10. Any projects turned in after that will be assessed a 10% penalty. No course project will be accepted for grading if turned in after the sunday night prior to Finals week.

In additions, all projects other than the last one (this project) must be turned in by Friday night at the end of week 10 or they will not be graded.