CSC 306 PROJECT 3
INTERNAL REPRESENTATION

PROFESSOR GODFREY C. MUGANDA

The purpose of this project is to extend the parser project so that in addition to the task of syntax analysis, it translates the source file into an internal representation that will be executed by the fourth and final project.

As discussed in lecture, the internal representation consists of a

(1) A class hierarchy rooted in a base class `cmm_super_expr`, with two subclasses, `cmm_string_expr` for representing strings, and `cmm_expr` for representing arithmetic expressions. The `cmm_expr` class has further subclasses as discussed in class: you will find the details in the source files provided by the instructor.
(2) A class `cmm_stmt` that serves as the base class for statements. Again, you will find the details for the various subclasses in the provided source files.
(3) Three major tables: a string table, a variable table, and an instruction table.

1. Overview: The `main` Function

A file with a main function is provided, which should be used as is. It takes command-line arguments for the source and listing files, creates file stream objects to open the files, and passes the address of the two file stream objects to a `Compiler` object. The `Compiler` class is just the old `Parser` class, modified to generate the code for the internal representation. The main function then invokes a `compile()` method to compile the source code and generate the results into the string table, variable table, and instruction table, which are then printed to the end of the listing file by the main function.

2. Modification to the Lexical Analyzer

Make sure your lexical analyzer recognizes all tokens, including the additional keyword `do`, `then` and `else`, as well as the extraction and insertion operator tokens. In addition, modify the lexical analyzer so that it stores pointers to `istream` for the source, and `ostream` for the listing, and takes pointers to the same as constructor parameters.

3. The Virtual Machine classes and Functions

You can think of the internal representation as defining an architecture for a CMM virtual machine, sort of like the Java Virtual Machine defines and internal representation for Java. This code is provided to you in the files `cmmVirtualMachine.h` and the accompanying .cpp file. Look through it and make sure you understand how it realizes the design we looked at in lecture. You will need this understanding in your modification of the parser.

4. Modification of the Parser

Rename the `Parser` class to `Compiler`, and add a public `compile()` method that calls `program()`, which may now become private.

Secondly, modify the declarations for all functions that parse expressions so that they take as parameter, a reference to pointer to an expression of the appropriate type:

```
void output_expr(cmm_super_expr *&);
void expr(cmm_expr *&);
void comp_expr(cmm_expr *&);
void simple_expr(cmm_expr *&);
void factor(cmm_expr *&);
```

Proceed as follows:

(1) Modify `vardeclist` so that it builds the variable table. An identifier that is declared more than once should result in an non-fatal error message being written to the listing file for each repeat declaration.
(2) Create an internal representation for each break / continue statement encountered, and add the internal representation to the instruction table. You may use a target of -1 for now.
(3) Modify the factor parsing functions to create internal representations for the expressions they recognize.
(4) Create an internal representation for each assignment statement encountered and add to the instruction table. First occurrence of an undeclared identifier on the left hand side should be flagged as a non-fatal error.
(5) At this point, you can test assignment statement with different types of factors. For example a program

```
int x;
begin
  x = 23;
end
```

can be used to test both assignment and a number expression factor. The program

```
int  x, y;
begin
  x = 23;
  x = -23;
  x = y;
  x = (y);
  x = !y;
end
```

can be used to test all types of factors. Use this type of reasoning to construct tests for other types of expressions.
(6) Modify, in order, the `simpleexpr`, `compexpr`, `expr`, and `outputexpr` parsing functions and test.
(7) Create an internal representation for output statements and test.
(8) Create an internal representation for input statements and test.
(9) Create an internal representation for while statements and test.
(10) Create an internal representation for if statements and test.

(11) Create an internal representation for loop statements and test. Loop statements are potentially infinite loops, so you put a `loop` marker statement at the top, and at the bottom, you put a `goto` statement that transfers control back to the marker statement at the top. Put an `end loop` marker statement right after the `goto`.

5. Some Things to Think About

In the next project, we will modify the break statements and continue statement so that

(1) They can only appear in loops or while statements: their appearance outside of a loop or while statement is a non-fatal error.
(2) The continue statement transfers control to the top of the smallest enclosing loop or while statement, while the break statement transfers control to the end of the smallest enclosing loop or while statement.

See if you can figure out how this can be done.

The source code for the main file, as well as the virtual machine classes and functions, will be posted on the course website. A few sample test files will be posted as well. As always, remember that exhaustive testing is your own responsibility.

6. Due Date

This is due Sunday at the end of week 8.