Compiler Project

The compiler project will be the major assignment for the quarter. This project will be divided in to four phases, where the due dates for each phase will be announced in class.

1. Scanner and Tokens

   At the end of this phase, a scanner will be produced that can read a C– file (.cm). The scanner will print each token along with the line number of the token. If the token is an identifier, the lexeme or string value of the identifier is printed; or, if the token is a number, the value of the number is printed.

   The scanner, the parser, and the code generator will need to use a standard set of constants. To make this as standardized as possible, an interface will be provided that implements the following as public integer constants.

   EOF, ERROR, ELSE, IF, INT, RETURN, VOID, WHILE, PLUS, MINUS, MULT, DIV, LS, LEQ, GT, GEQ, EQ, NEQ, ASSIGN, SEMI, COMMA, LPAREN, RPAREN, LBRACKET, RBRACKET, LBRACE, RBRACE, READ, WRITE, NUMBER, ID, PROGRAM, DECLARATION, VARIABLE, ARRAY, FUNCTION, EXPRESSION, CALL, COMPOUND, TYPE_SPECIFIER, PARAMETER_LIST, PARAMETER, STATEMENT_LIST, STATEMENT, and ARGUMENTS.

   The scanner and the tokens will be objects of separate classes. The scanner should have a constructor that takes a parameter that represents the file to be scanned. For example, it may be a `File` object or an object of a separate class that represents the file. You will also need a separate driver program that will contain the main method. This testing program can be used throughout the project, and it does not need to implement a GUI. For the first phase, the testing program gets a file name from the user, opens the file in a manner compatible with the scanner, instantiates a scanner object, and repeatedly calls the method to get a token until the EOF token is obtained. The testing program prints each token as it is obtained from the scanner.

2. Parser and Syntax Tree

   At the end of this phase, a recursive descent parser will be written that accepts a syntactically correct C– (.cm) program and reports an error for a syntactically incorrect program. The parser does not have to do any error correction, it can report the first error it finds and then quit. If you examine the grammar for a C– program in Appendix A, you will find a BNF grammar for C–. This has to be converted to an EBNF grammar with the left-recursion removed before it can be used with a recursive descent parser. In addition, I am adding two statements to the grammar: one that will read a value and the other will write an expression.

   As the program is parsed, it builds an abstract syntax tree. If the program is successfully parsed, a copy of the syntax tree is printed out for this phase. I have a document that gives you the specifications for the syntax tree along with a sample program and the tree that is

produced for it. I am afraid the document is somewhat long, 25 pages, so it will come out as a separate hand out.

3. Checker and Symbols Table

The checker takes the root of the syntax tree as a parameter and makes one or more passes over the tree. As the tree is traversed, the symbols table is built. This means that variables must be declared before they can be used and cannot be declared twice in the same block. In addition, as the symbols table is built, variables are given a unique rename value. Since there is only one data type, integer, we do not have to check that a floating point value is being assigned to a Boolean variable, for example. However, we do need to check that a void function is not being used in an expression, that a void function does not have a return statement within it, and that a non-void function contains a return statement. It is not necessary to check if the return statement in a non-void function is reachable, nor is it necessary to check that formal and actual parameters match in type and quantity. However, when an array value is used and if the array is defined in the same block, the value is checked to see if it is in bounds.

Since the symbols table grows and shrinks as blocks are opened and closed, for this project, create a second table. When an item is added to the symbols table, add it to this duplicate table as well; however, this duplicate table does not shrink when the symbols table does. At the end of this phase, print out this table. In this way, all items that were added to the symbols table can be checked.

What items should be in the symbols table? The following is my definition.

```
public String ID;                  //The lexeme
public int entryType;              //variable, array, etc.
public int dataType;               //INT or VOID
public int blockLevel;             //The nesting level: 0 is lowest level
public TreeNode parameterList;     //Just copied from the syntax tree
public int returnType;             //For functions: INT or VOID
public int arrayMax                //The size of an array
public String rename;              //Each variable is given a unique name
```

4. Code Generator

The target language for this compiler is VM4 assembly language. In this manner, the VM4 assembler can translate the object file into a VM4 executable file that can be executed by the VM4 simulator. However, the code generator can be the most difficult phase of a compiler to write; so, I will give you a class file called `CodeGen.class`. The constructor for this class takes two objects, the first is the root of the syntax tree (a `TreeNode` object) and the second is a `File` object into which the output will go. The `File` object should be opened before being passed into the constructor and it should refer to a file with an `.asm` extension. The method to generate the code is `public` and is called `genCode()`. For this phase of the project, you just

have to integrate the code generator into your compiler. If everything else has been done correctly, this should be simple.

Of course, if you want, you may write your own code generator!. I will give you documentation on VM4 assembly language.

Grading

The project will be graded using the following criteria.

Phase 1: 20 points for the scanner and 10 points for a workable definition of a token.

Phase 2: 20 points for the parser and 20 points for the syntax tree.

Phase 3: 20 points for the checker and 10 points for the printout of the symbols table.

Phase 4: 20 points for integrating the code generator (or writing your own) in such a manner that a correct C– program can be compiled, assembled, and executed using the VM4 simulator.