Computer Science 426--Fall 2013
Project 4--A Recursive Descent Parser

1. Administrative Details: This project is due at 5pm on **Wednesday** October 30th. Make a copy of your stage3<lastname> folder named stage4p<lastname> (for stage4 preproject) and work on the project there. When you are done with the preproject, copy the folder to stage4<name> and complete stage4 there. Email me to tell me when the project is uploaded and ready for grading.

2. A PreProject: This small homework step is to be turned in (by uploading to Moodle sending me an email) as a homework by the date announced in class. Your task for the preproject is to implement a recursive descent parser for the small subset of YASL given by the following grammar:

```
<statement>   ->  while <expression> do <statement> |
                  identifier = <expression> ;
```

For example you should be able to parse input files that consist of a single assignment statement as well as input files of the form:

```
while ((x+2) <> (y+6)) do
   while ((z>6) and (domore)) do
        x= (y+z div 2);
```

where there can be an arbitrarily large number of nested while loops.

The recursive descent parser should use the expression parser (project 3) to recognize expressions. If the expression parser detects an error it should print an appropriate message and exit the program. If the expression parser parses a valid expression then control should return to the recursive descent parser.

If the recursive descent parser detects an error, it should a message of the following form.

Error: Found <xyz> when expecting one of <abc, def, geh, hij>.

And then should **print the current line.**

In the above message, xyz is the lexeme that is being parsed at the time of the error and <abc, def, geh, hij> are the tokens (don't print numeric values for these, print some meaningful text or symbol) that would have allowed the parse to continue. Once the message has been printed your program should exit.

You may find that your compiler seems to print "one line too late" when it prints the offending line. This is actually a classic problem and is present in most commercial compilers. To see the problem, consider the input:

```
x = y + z
cat = 12;
```

The error is the missing semi-colon after the z. The error will be detected upon reading the "cat" and by that point printCurrentLine will print the line "cat = 12;" rather than "x = y+z". There is not much you can do about this other than print the message

"Error blah blah blah at **or near** line:"
<Print current line here>

For syntactically correct YASL programs, your parser need not produce output except for a statement saying that YALS-XY compiled Z lines successfully (you have a file manager function from project 1 that can return this number of lines and XY are your initials).

Remember that we are not worrying about type checking at this point. I suggest writing the parser using the following guidelines:

1. The main() function should be very very simple

```
int main ()
{       parseClass parser:

        parser.parseProgram();

        PRINT INITIALS AND NUMBER OF LINES COMPILED HERE
        cin.get();
}
```

2. Complete this project by **making additions** to the parser.h and parser.cpp files that you started in project 3 **rather than** making new files.

3. In project three your parseExpression function made use of a variable (perhaps named token) that stored the next token to process. In project four this variable will have to be accessible in the recursive descent parser as well as in parseExpression function. To deal with this, declare this variable as a private instance field in the parser class. You may have already done this in project 3 – or you may have declared the token variable to be a local variable in the parseExpression function. In the latter case you will need to move its definition so that it is a private instance field.

Add a public method to the parser class named parseProgram(). This routine should use getToken() to load the first token into the token variable. The routine can then call the recursive routine named statement() that tries to expand the non-terminal <statement> in the above grammar.

4. You will probably have to make a minor modification to your expression parser to account for the fact that the first token of the expression will have already been read into the private data members described in (3) above when the expression parser is called. In other words, your expression parser should no longer read the first token before entering the loop.

5. A second minor modification deals with the fact that expressions in the above grammar can be followed either by a SEMICOLON or by the keyword "do". Therefore, your expression parser should return if your bottom of stack marker (presumably a ;) is the top most terminal on the stack and if the current token is either a SEMICOLON or the keyword "do".

6. Currently if you send the input "do" to your precedence table lookup, "do" will map into the other column and your precedence function will return an error. You will need to modify your precedence function for the preproject so that "do" is treated just like a SEMICOLON in terms of what the prec function returns. For the full project you will need other tokens that can follow an expression to be treated just like SEMICOLON.

7. Currently if you feed your expression parser the input:

```
;    //just a semi-colon, the expression is empty
```

no error will be reported. This is fine for project 3, but will be a problem later on. For example if you feed your parser:

```
while    do ...
```

you want it to report an error due to the empty expression between the parens.

In order to handle this you have two choices: (a) check to be sure the token is in the first set of an Expression before calling the expression parser (otherwise print error and exit), or (b) add code to the very top of the expression parser that checks to see if the token is in the first set of an expression. If not, print an error message and exit the program. The first approach is in some ways more consistent, but the latter approach will save you some redundant code. The choice is yours, but I would lean toward (b).

8. Finally, observe that your expression parser is already leaving the next token in the token variable when it finishes. This is great because it will integrate well with the recursive descent parser. You should not have to change any code in order to deal with this, just remember that the next token (the first one after the expression) has already been read in when the expression parser finishes.

3. The Actual Project:

Copy **stage4p<name>** to **stage4<name>** and then implement a recursive descent parser for YASL programs. Use the context free grammar we develop in class to drive the parser (note that this grammar is different from the one you used in the stage4 preproject. All of the comments in the preproject described above apply to the full parser. The only difference between the preproject and the full parser is that you will now be using the grammar for the entire YASL language.

Your parser will only detect those errors which can be explicitly determined by the grammar. For example, the parser will not detect variables that are used without being declared, type mismatches, and function calls where the number of parameters in the call does not match the number (or type) in the function header. Have faith... these issues will be dealt with, in subsequent projects.

I suggest that you approach this project by working backwards: first develop the code needed to parse all statements. Then add the code for the functions, program body, etc.