Computer Science 426 --- Fall 2013
Project 3 --- An Operator Precedence Expression Parser

1. Details and What to Turn In: This project is due at 5pm on Wednesday, October 9th. When you are done with the project, upload it to Moodle and email me to let me know you are done. Strong suggestion: make a copy of your *stage2LastName* folder and call it *stage3LastName*. Then complete this project in the *stage3LastName* folder.

2. The Project: Using class discussion and handouts you are to develop a precedence table for YASL expressions. Assume (for now) that every expression is terminated with a ";" (in other words, use a ";" the same way that the "$" was used in the class examples -- this includes pushing a ";" on the stack when you start the parsing algorithm.) Be sure to include a column labeled "other" for which every entry in the table is an error condition.

A grammar for YASL expressions is:

```
E -> E + E | E * E | E - E | E div E | E mod E | (E) | E or E
E -> E and E | E == E | E < E | E <= E | E > E | E >= E | E <> E
E -> id | integer-constant | true | false
```

There are four precedence levels in YASL listed below from highest to lowest:

1.(parens) ( )
2.(mulops) * div mod and
3.(addops) + - or
4.(relops) == <> <= >= < >

After you have designed the table, use it to implement a shift-reduce parser for YASL expressions. Whenever your parser needs a new input symbol it should call the getToken() method written in project 2.

Implement a new pair of compiler directives {$e+}, {$e-} that allow expression debugging to be turned on and off (the default is off). Add a new member function to the scanner class that returns the current status of expression debugging (true or false). The parser can then call this function whenever it does a reduction. If the function returns true, the parser should print the reduction in a format such as "E -> E + E". Thus the parser, when in debugging mode, will print a right-most derivation backwards for each expression it encounters.

Your expression parser should be placed in the files parser.h and parser.cpp (parser.cc on Linux) which will #include "scanner.h". The parser should be implemented using a class named **parserClass**. This class will have a private data field which is of type scannerClass. Thus the parserClass will aggregate the scannerClass much as the scannerClass aggregates the fileManagerClass.
The driver program can be as simple as:

```
void main()
{ parseClass parser;

  while (1)   //an infinite loop
  {         cout << "About to parse an expression: \n";
            parser.parse_expr();
            cout << "Parsed one expression\n";
  }
}
```

If a syntax error is encountered, print the following error messages:

Syntax Error: Invalid Expression.

Then print the current line and exit the program.

Since the routine printCurrentLine() is a member of the fileManagerClass and this class is not accessible from your parserClass you will have to add a new member function to the scannerClass called scannerClass::printCurrentLine(). All this function will do is call the fileManagerClass::printCurrentLine() function.

Note that parseExpr() will terminate with a syntax error when it encounters the EOF at the end of your test input file. This is fine for now and, indeed, is how the driver program will terminate in this stage of the project.

You should be able to parse a file like the following which happens to contain 5 expressions:

```
{$e+}
A + B;
(A>=B)and
    (Cat34<Dog1);
{Here is a comment}
45*(x45-34);
A mod (B div (2 + 4));Sum == 10;
```

Note that you should look only at syntactic issues as you parse the expressions. In particular, you should not do type checking. For example the expression A + B is valid even though it might be the case that "A" is of type boolean. Similarly the grammar allows an expression like: true + 10 and your parser should (for now) consider this valid even though it obviously is not.

Notes on the Stack:

In order to complete this project you will need to implement a stack class. Do this in a header file named **pstack.h (don't call it stack.h as this already has meaning in C++)**. The stack should be dynamic (i.e. use pointers). You should think carefully about which member functions you want the stack to have (you do not have to limit yourself to the standard "push" and "pop" functions), by looking at the pseudo-code for the operator precedence parsing algorithm to see how the stack is accessed. For example, member functions for "isThereATerminalOnTop()" and "returnTopMostTerminal()" would be **very useful**.

One final note with respect to the stack, each cell of the stack should store the token type as well as the token subtype. There is no need to store the lexeme or any other information.

Notes on your Table:

If you are observant, you will note that all of the "mulops" are equivalent to each other in terms of the table entries, similarly all of the addops are equivalent to each other, similarly for all of the relops. Therefore, if you wish, you can collapse your table so as to only have one row and column for all the mulops, another row and column for all the addops, etc. In addition, if you are clever, you might want to renumber your const definitions so that those tokens that are in the table are numbered starting from 0 with no gaps. If your earlier projects are well-written, renumbering the constants should be very easy to do, and should have no impact on your existing code. If you do this renumbering, you can use the token type directly to index the table.