Syntax Analyzer Assignment

Assignment objective:

Implement a recursive descent parser for the Teeny Tiny Pascal Language given on page 4 and 5.

Design:

1. Download the files **Assignment2.cpp**, **SyntaxAnalyzer.cpp**, **SyntaxAnalyzer.h**, **Token.cpp**, **Token.h**, **TokenCodes.h**, **test1.pas**, and **test2.pas** provided along with this assignment in Blackboard and include them in your project. Add your **LexicalAnayzer.cpp** and **LexicalAnayzer.h** files to your project.

2. The file **Assignment2.cpp** is the driver code for the assignment. The main function receives, via the command line argument, the name of the Teeny Tiny Pascal source code program file that your parser processes. The main function 1) creates the lexical analyzer object, passing into it the filename of the Teeny Tiny Pascal source code program file, 2) creates the parser object, passing into it the lexical analyzer object, and 3) calling the method **Program** of the parser object to start the parsing process. You **cannot** modify any of the code in this file.

3. The files **Token.cpp** and **Token.h** contain the Token class and you **cannot** modify any of the code in these two files. The file **TokenCodes.h** contains the enumerated type which includes the token symbols for all lexemes in the Teeny Tiny Pascal Language and you **cannot** modify any of the code in this file.

4. Modify your lexical analyzer so when it reads in the next line from the Teeny Tiny Pascal source code program file the lexical analyzer prints that line via cout.

5. Next, add this method to your LexicalAnalyzer.cpp file:
   ```
   int LexicalAnalyzer::getCurrentTokenPosition()
   {
     return currentTokenPosition;
   }
   ```

   Add this line to your LexicalAnalyzer.h file in the private section:
   ```
   int currentTokenPosition;
   ```

   Add this line to your LexicalAnalyzer.h file in the public section:
   ```
   int getCurrentTokenPosition();
   ```

   This added code is part of the code needed by the lexical analyzer (while it recognizes the current token) to store in the class's data member currentTokenPosition the index of the current token's first character. You must add code to your lexical analyzer to produce the value for currentTokenPosition. See the next number for an example of producing the value for currentTokenPosition.

6. An example of producing the value for currentTokenPosition:
   Assume the lexical analyzer works on the current line: `x, y, z : integer;`
   Assume the next token the lexical analyzer returns via getNextToken() is the reserved word `integer`. At index 0 is `x`, the character at the beginning of the line. The index of the first character, `i`, of the next token `integer` is 10. While the lexical analyzer works on recognizing the next token `integer`, the lexical analyzer stores a 10 in the data member currentTokenPosition. Therefore, in this example, after the call to getNextToken() returns the token `integer`, a call to the method getCurrentTokenPosition() returns 10.

7. SyntaxAnalyzer is the syntax analyzer's class. Complete the code to implement the parser for Teeny Tiny Pascal by adding your code to the files **SyntaxAnalyzer.cpp** and **SyntaxAnalyzer.h**. You **cannot** change any of the code that is already in those files, but you can add code anywhere within those two files. You must declare private, all the data members you add to the syntax analyzer class by adding them under **private:** in the file **SyntaxAnalyzer.h**. You must declare private, all the methods you add to the syntax analyzer class by adding their declaration under **private:** in the file **SyntaxAnalyzer.h** and defining them in the file **SyntaxAnalyzer.cpp**. You create one method for each nonterminal symbol in the grammar. You create an error method to print out a syntax error message and then terminate the parser. You can create any additional methods for your parser, as needed. You need to rewrite some of the rules of the grammar given on pages 4 and 5 to make it possible to write the code for your recursive descent parser.

8. The Teeny Tiny Pascal source code program is either syntactically correct or contains one or more syntax errors.
   a. If the Teeny Tiny Pascal source code program is syntactically correct, then the parser reaches the last token EOI. In this case, have the parser print "The program is syntactically correct." on the line after the last line of the source code program.
   b. If the Teeny Tiny Pascal source code program contains syntax errors, then the parser (in this assignment) only finds the first syntax error, prints the appropriate error message for that syntax error, and then terminates. Print the error message on the two lines below the current source code program line containing the syntax error. Also, use the **^** character to point to the location in the line where the syntax error possibly occurred. See below for an example. The parser terminates right away by using the function call exit(0).

      Error message example:
      ```
      var x, y z : integer;
               ^
      Error: colon expected.
      ```

9. The error method prints out the two lines of the syntax error. Hence, the error method calls the getCurrentTokenPosition() method of the lexical analyzer for use in printing the first line of the syntax error, the one containing the **^** character, to print the appropriate number of blanks before the **^** character. The last statement the error method executes is exit(0).

   Hint: you can number each unique syntax error message passing the syntax error number to the error method rather than passing the text of the syntax error message. Hence, the text of each syntax error message is a part of the error method code.

10. The files **test1.pas** and **test2.pas** are examples of Teeny Tiny Pascal source code programs. The file **test1.pas** is syntactically correct and the file **test2.pas** contains the syntax error as seen in number 9 above. You must test your parser with more test cases created by learning the Teeny Tiny Pascal syntax using the grammar given on pages 4 and 5.

11. I execute your syntax analyzer in Linux compiled using g++ using this command:
    ```
    g++ -std=c++11 *.cpp
    ```
    The command I use to execute your program, assuming example.pas contains a Teeny Tiny Pascal source code program and the name of the executable code of your program is a.out, is:
    ```
    a.out example.pas
    ```

12. **Tip:** Make your program as modular as possible, not placing all your code in a single method. You can create as many methods as you need in addition to the methods already in the syntax analyzer class. Methods being reasonably small follow the guidance that "A function does one thing and does it well." You will lose a lot of points for code readability if you don't make your program as modular as possible. But, do not go overboard on creating methods. Your common sense guides your creation of methods.

13. Do **NOT** type any comments in your program. If you do a good job of programming by following the advice in number 12 above, then it will be easy for me to determine the task of your code.

Teeny Tiny Pascal Language (TTPAS) BNF Grammar

```
<PROGRAM> → program IDENT ; <DECPART> <COMPSTMT> . EOI

<DECPART> → ε
<DECPART> → var <DECLARATIONS>

<DECLARATIONS> → <DECLARATION>
<DECLARATIONS> → <DECLARATION> <DECLARATIONS>

<DECLARATION> → <IDENTLIST> : Boolean ;
<DECLARATION> → <IDENTLIST> : integer ;
<DECLARATION> → <IDENTLIST> : real ;

<IDENTLIST> → IDENT
<IDENTLIST> → IDENT , <IDENTLIST>

<COMPSTMT> → begin <SEQOFSTMT> end

<SEQOFSTMT> → <STATEMENT>
<SEQOFSTMT> → <STATEMENT> ; <SEQOFSTMT>

<BLOCK> → <COMPSTMT>
<BLOCK> → <STATEMENT>

<STATEMENT> → for IDENT := <EXPRESSION> to <EXPRESSION> do <BLOCK>
<STATEMENT> → for IDENT := <EXPRESSION> downto <EXPRESSION> do <BLOCK>
<STATEMENT> → IDENT := <EXPRESSION>
<STATEMENT> → if <EXPRESSION> then <BLOCK>
<STATEMENT> → if <EXPRESSION> then <BLOCK> else <BLOCK>
<STATEMENT> → repeat <SEQOFSTMT> until <EXPRESSION>
<STATEMENT> → while <EXPRESSION> do <BLOCK>
<STATEMENT> → read ( <IDENTLIST> )
<STATEMENT> → readln ( <IDENTLIST> )
<STATEMENT> → write ( <IDENTLIST> )
<STATEMENT> → writeln ( <IDENTLIST> )

<EXPRESSION> → <SIMPEXPR>
<EXPRESSION> → <SIMPEXPR> = <SIMPEXPR>
<EXPRESSION> → <SIMPEXPR> <> <SIMPEXPR>
<EXPRESSION> → <SIMPEXPR> < <SIMPEXPR>
<EXPRESSION> → <SIMPEXPR> <= <SIMPEXPR>
<EXPRESSION> → <SIMPEXPR> > <SIMPEXPR>
<EXPRESSION> → <SIMPEXPR> >= <SIMPEXPR>
```

```
<SIMPEXPR> → <SIMPEXPR> + <TERM>
<SIMPEXPR> → <SIMPEXPR> - <TERM>
<SIMPEXPR> → <SIMPEXPR> or <TERM>
<SIMPEXPR> → <TERM>

<TERM> → <TERM> * <FACTOR>
<TERM> → <TERM> / <FACTOR>
<TERM> → <TERM> div <FACTOR>
<TERM> → <TERM> mod <FACTOR>
<TERM> → <TERM> and <FACTOR>
<TERM> → <FACTOR>

<FACTOR> → not <PRIMARY>
<FACTOR> → <PRIMARY>

<PRIMARY> → ( <EXPRESSION> )
<PRIMARY> → IDENT
<PRIMARY> → NUMLIT
<PRIMARY> → true
<PRIMARY> → false
```

Grading Criteria:

The assignment is worth a total of 20 points, broken down as follows:

1. If your code does not implement the task described in this assignment, then the grade for the assignment is zero.
2. If your program does not compile successfully then the grade for the assignment is zero.
3. If your program produces runtime errors which prevents the grader from determining if your code works properly then the grade for the assignment is zero.

If the program compiles successfully and executes without significant runtime errors, then the grade computes as follows:
Followed proper submission instructions, 4 points:
1. Was the file submitted a Zip file?
2. The Zip file has the correct filename.
3. The contents of the Zip file are in the correct format.

Code implementation and Program execution, 12 points:
- The code performs all the tasks as described in the assignment description.
- The code is free from logical errors.
- Program input, the program properly processes the input.
- Program output, the program produces the proper results for the assignment.

Code readability, 4 points:
- Good variable, method, and class names.
- Variables, classes, and methods that have a single small purpose.
- Consistent indentation and formatting style.
- Reduction of the nesting level in code.

Late submission penalty: assignments submitted after the due date are subjected to a 2-point deduction for each day late.

Late submission policy: you **CAN** submit your assignment early, before the due date. You are given plenty of time to complete the assignment well before the due date. Therefore, I do **NOT** accept any reason for not counting late points if you decide to wait until the due date (and the last possible moment) to submit your assignment and something happens to cause you to submit your assignment late. I only use the date submitted, ignoring the time as well as Blackboard's late submission label.

Submission Instructions:

Go to the folder containing your **LexicalAnalyzer.cpp**, **LexicalAnalyzer.h**, **SyntaxAnalyzer.cpp** and **SyntaxAnalyzer.h** files, select them, and place only them in a Zip file. The file can **NOT** be a **7z** or **rar** file! Follow the directions below for creating a Zip file depending on the operating system running on the computer containing your assignment's syntax analyzer files.

Creating a Zip file in Microsoft Windows (any version):

1. Right-click the selected files to display a pop-up menu.
2. Click on **Send to**.
3. Click on **Compressed (zipped) Folder**.
4. Rename your Zip file as described below.
5. Follow the directions below to submit your assignment.

Creating a Zip file in Mac OS X:

1. Click **File** on the menu bar.
2. Click on **Compress ? Items** where ? is 4, the number of files you selected.
3. Mac OS X creates the file **Archive.zip**.
4. Rename **Archive** as described below.
5. Follow the directions below to submit your assignment.

Save the Zip file with the filename having the following format:
    your last name,
    followed by an underscore _,
    followed by your first name,
    followed by an underscore _,
    followed by the word **Assignment2**.
For example, if your name is John Doe then the filename would be: **Doe_John_Assignment2**

Once you submit your assignment you will not be able to resubmit it!
Make absolutely sure the assignment you want to submit is the assignment you want graded.
There will be **NO** exceptions to this rule!

You will submit your Zip file via your CUNY Blackboard account.
The only accepted submission method!

Follow these instructions:
  Log onto your CUNY Blackboard account.
  Click on the CSCI 316 course link in the list of courses you are taking this semester.
  Click on the **Assignments** tab in the red area on the left side of the webpage.
  You will see the **Syntax Analyzer Assignment**.
  Click on the assignment.
  Upload your Zip file and then click the submit button to submit your assignment.

Due Date: Submit this assignment on or before 11:59 p.m. Monday, November 28, 2022.