CS 441G – Fall 2018
Project 2.1: The Parser – Phase I
100 points – Due Fri Nov 16

General Description:
You are to implement the Parser of the compiler as a *Push Down Automata* (PDA) for an LL(1) Context Free Grammar (CFG). Use the algorithm given in the class notes on the course website. Note that there is *only* one central loop in the PDA (other loops will be needed to search the Grammar for selecting a production; see below).

Use your Scanner and String Table from previous projects, or use the ones provided on the course website (there are some known minor bugs in the scanner – see `scanner.h` for details)

For debugging and grading purposes, your parser should report the current token from the source and print the current stack at the beginning of each iteration of the loop. At the end, it should report an error when found, or success.

Files Provided: you are not required to use any of these, but they may be helpful

- **token** (.h/.cpp) – defines constants for Terminals, Non-Terminals and Errors related to Token ID numbers, as well as arrays of string representations for these for use in debugging. The .cpp contains free functions for using the arrays for converting integer values to their string representation.
- **tokStack (**.h/.cpp) a simple stack of tokens class
- **scanner** (.h/.cpp) implementation of the scanner. Known bugs are documented in the .h
- **stringTable** (.h/.cpp) implementation of the string table.
- **grammar.h** a 2D array expressing the Pascal Grammar with select sets, along with other related constants. The constants for Terminals and Non-Terminals in the Grammar are defined in the `token.h` given. See details in the comments of the header file.
- **grammar** (.xlsx and .pdf's) we will use this as our official documentation of the grammar to be implemented. Note this has been expanded somewhat over the grammar used for the homework (it has been rewritten as an LL(1) grammar).
- **main.cpp** see further details below. Minor modifications may be made to implement in C rather than C++.
- ***.pas files** – these are test files for you to try; more will be added shortly.

Specifications:

- **main() and parse()**
  Since the parser is the "main" of the compiler (in the design we are using), the main() should only retrieve the command line argument, assumed to be the name of the input file (error check for missing argument), and pass it to the *parse()* method or function. *parse()* should not return any value, and main() should perform no further tasks after invoking *parse().*

- **Error Messages:** Remember the scanner may issue error messages, and the *getNextToken()* method should return an "error token" when this happens. Of course, this stops the action of the parser. If you find you need other messages, email the instructor and they may be added to the list.

  `Parse Error: no production selected for Top Token`
  Given the current Top of Stack token and input token from the source, no production could be found in the Grammar matching the select set for any token.

  `Parse Error: no productions for non-terminal`
  No production has a left hand side matching the given Top Token (never happens unless there is an error in the Grammar itself, but "just in case").

  `Parse Error: input Token found, but Top Token expected`
  The current Top Token and Input Token do not match (are not equal)

  `Parse Error: scan error reported`
  The scanner's getNextToken() reported/printed an error message; the parser just recognizes this.

  `Parse Error: unexpected end of source`
  The scanner reports end of source, but the PDA's stack is not empty

  `Parse Error: unexpected tokens in source past end of program`
  The PDA's stack is empty, but the scanner does not report end of source.

  Note some of these conditions are checked *after* the PDA's main loop has exited.

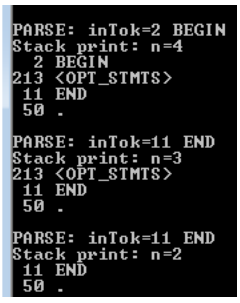
  On the next line, after the error message, print the current tokens: top of stack and input from source, using the string representation (ie. use *getTokenString()*)

  `Top Token = topToken  Input Token = inputToken`

  `Parser: success!`
  When the parse ends successfully.

- **Debugging/Grading Messages:**

  At the top of (and inside of) the primary loop of the PDA algorithm:
  - print a blank line
  - print the current input token from the scanner (integer and string representations)
  - print the contents of the stack:
    - print `Parse: n=numElementsOnStack`
    - print one token per line, with the top first (integer and string representations)
    - format: cout << setw(3) << intTokenId << " " << stringTokenId
  - The example to the right shows 3 iterations of the loop



- **Program Design:**
  You are free to design the program as you like, but within some basic "good practice" and style guidelines: ("method" here can also mean "function", especially for those using C)
  - Use good style: meaningful names, proper indentation, sufficient commenting and blank lines.
  - Use only approved C/C++ libraries (email the instructor to add a library to the approved list)
  - Methods should generally be no more than 30 lines of code, including blank lines, between the { and }. If it is too long, break it into sub methods.
  - A comment box for each method, with the method name centered. Constructors and related small methods may be grouped under one box.
  - For OOP: all data members should be private; all sub-methods should also be private (ex: in class parser, probably only the *parse()* method should be public.)
  - As always, variations from these rules can be granted, but ask (email) first to avoid point deductions.

  Here are some suggestions for `class parser`:
  - The string table, scanner and token stack objects should be data members of the class
  - *chooseProd(topToken, inputToken)*: returns: index into the grammar array of the production chosen, or return an error code (see 2 of the error messages above). Search down the list to find the first production where the LHS equals the topToken. Search the Select Set Members of a production to see if any matches the given inputToken; *or* if the first member of the Select Set is **E** (meaning this is the default token).
  - *pushRHS(prodNdx)*
    push the tokens on the RHS of the production indexed by the given index: remember, push them in reverse order:
    N → aZb push(b), push(Z), push(a)
  - *checkFinish(?)*: check all the conditions/situations for why the primary loop ended and report an error message or report success.

  There could be others.

Submit in Canvas: a .zip file
containing:
- all .cpp and .h files needed to compile your program (even those copied from the website)
- a **make** file that compiles to **p.out**
- optional: README.TXT if you like

not containing:
- anything else! especially .o, .out, or .exe files.
- also, no Mac project files

if the make file is missing, or does not work, the following will be used on MultiLab/Linux to compile:

```
g++ *.cpp –o p.out
```

If make and the compile command given results in compiler errors, 40% will be deducted from the grade. **Be sure it compiles on MultiLab before zipping/submitting!**