Lab 1: Stacks

In this lab you'll experiment with some linked lists to refresh your memory.
You'll write a C and Python implementation of a stack.
And you'll write a parenthesis checker in both languages, using the stack
data structure.

Minimal test code will be provided. Write your own test code, but
ensure that you strip any test code, except where indicated.

The lab will be due IN YOUR LAB SESSION, at the end of session.
You will submit this using the submitcsc190s command.

I.e., to submit file.c
submitcsc190s 0 file.c

This lab is practice for your tests and final exam. Start this lab early, so
that you complete it prior to your lab session.

Due: in lab, during the lab sessions during the week of Jan 14.

A. Introduction: getData.c

Read and confirm you understand getData.c
Compile it, run it and test it for various cases, as per the instructions
in the comments.

We have implemented a simple linked list that deals with integers; the
linked list was necessary because, as you can see from the code, the input
data that is read in is of a length (i.e., number of data items) that is
unknown, a priori. Hence, a while loop is necessary to get the data from
the user, and a dynamic sequential data structure --- a list --- is needed
to store that data.

We have provided three helper functions:
* llnode_add_to_tail
* llnode_print_from_head
* llnode_print_from_tail

These functions enable you to simulate the behavior of
* first-in first-out (FIFO, aka "queue")
* last-in first-out (LIFO, aka "stack")
data strutures.

Once again, Stacks and Queues are merely special cases of general lists.
Whereas a general list can be read from and written to in arbitrary
ways, a stack and queue are special cases of lists where the write
and read policies are restricted.

Specifically:
* Stacks: write to the tail, read from the tail
* Queues: write to the tail, read from the head

(Convince yourself by drawing a diagram
that the alternative spec also works:
* Stacks: write to the head, read from the head
* Queues: write to the head, read from the tail
)

Assignment:
A.1. cp getData.c getData_char.c
     and then modify getData_char.c so that it will

```
      handle characters instead of integers
 A.2. cp getData_char.c getData_charHead.c
      and then modify getData_charHead.c so that it has a new function
      llnode_add_to_head
 A.3. cp getData_charHead.c getData_charStack.c
      and then modify it so that it has two new functions:
      int push(llnode **x, char value);
      and
      int pop(llnode **x, char *return_value);

      Use the appropriate llnode_add_... function for the push function, which
      will add the value to the linked list so that the list behaves like a stack.
      The return value is 0 for success, and -1 for failure.

      Now, write the pop function that will assign to *return_value the
      appropriate value from the linked list so that the list behaves like a stack.
      (Recall: return_value is a mechanism via which we can "return" a value via
       the argument list, rather than via the return statement. Why do we require
       a pointer?)
      The function should ensure that the linked list node corresponding to the
      pop'd value is also deleted from the linked list.
      The return value is 0 for success, and -1 for failure.

      Modify the main function so that instead of calling add and print, it calls
      push and pop. Test your code.

 Submit all of the .c files you created for this section:
    getData_char.c, getData_charHead.c, and getData_charStack.c

 ------------------------------------------------------------------------------
 B. Parenthesis Checking

 The algorithm for Parenthesis Checking is as follows:

 * Read in an expression from the user (e.g., 1+2*(3+4*[5-6*{100+200}]))
 * In your while loop:
   * Every time you get a left bracket input (i.e., '(', '[', or '{')
     push that bracket onto the stack

   * Every time you get a right bracket input:
     - pop stack to a variable (popped_value, say)
     - make sure that the popped value is the correct closing bracket
       that corresponds to the input, and continue
     -> otherwise: break out of the loop

 * If the Stack is empty by the time you get to the end of the input
   stream (i.e., once you exit the while loop), then the brackets
   are balanced.

 Before the program ends: print out a message with the EXACT FORMAT:
 PASS
 or
 FAIL,5
 where 5 is a number indicating that the bracket matcher failed at the
 fifth char input

 Make sure your program does NOT print out anything other than specified.
 Submit this program as "bc.c"

 Test your program with a variety of inputs.
 For example: 1+2*(3+4*[5-6*{100+200}])
 should yield "PASS"

     Note: this expression can be read in by the
           getData_char.c and getData_charHead.c programs
```

```
          Test your getData_char code by:
          gcc getData_char.c
          echo "1+2*(3+4*[5-6*{100+200}]))" | ./a.out
          or:
          ./a.out (press enter)
          and then enter the string 1+2*(3+4*[5-6*{100+200}]))
          press enter
          and then Ctrl-D

          Note: in UNIX, the echo "1+2*(3+4*[5-6*{100+200}]))"
                simply gets UNIX to print out that string.
                Above, we pipe that printed string into the ./a.out
                program.



--------------------------------------------------------------------------------
C. Python
Write a class for a stack, as stackLib.py, which will implement a stack
data structure (LIFO).

It will have an internal list variable, to handle the data that is
input to it.
It should have two methods (in addition to __init__(self)):
* push
* pop
You may choose where to write to/read from the underlying list data structure
as long as you are implementing a stack data structure.

C.1. Submit stackLib.py, which should online contain the complete class
     definition and no test code.
C.2. In bc.py create a function "bc" that will
     take in a string, and return a length-2 list, L.

     The 0 element (i.e., L[0]) of the return value will be True if the
     result of bracket checking indicates that all brackets are matched, and
     False if otherwise.
     The 1 element of the return value will indicate the index of the input
     string where bracket checking failed.

     Submit bc.py, which will only contain the function definition, and
     whatever helpers, import statements, etc, you require; no test code
     should be included (if you python bc.py --- nothing should happen).
     Note: the function bc must use the stack data structure you created;
     hence, there will be an import stackLib.py in bc.py.


Notes:

1. When you input to the while loop in the above code, you will need to
signify the End of Input. To do that: Ctrl-D.

So if you call ./a.out on getData.c and you want to enter the numbers yourself
instead of via echo (as above), you will have to press Ctrl-D after you enter your
last data item.

2. For gdb to work you will have to delete your .mycshrc file from last semster.
Or comment out the bash line in .mycshrc
```