CSC173: Project 2
Grammars and Parsing

1. Implement a recursive-descent parser for the grammar of arithmetic expressions seen in class (FOCS Figure 11.2 or any reasonable/interesting variant).

   - Your parser should read successive expressions from standard input, construct a parse tree for the expression if it is well-formed, and then print that parse tree to standard output (see description below).
   - If the input is not well-formed, your parser should print an appropriate message and resume processing at the next line of input (*i.e.*, skip to the next newline).
   - You may assume that expressions are on a single line of input if you want, or deal with multi-line input.
   - Your program should exit on end-of-file on standard input.

   A parse tree is a dynamic data structure. You have seen them in Java and they're the same in C.

2. Implement a table-driven parser for the grammar of arithmetic expressions (as above). Read expressions from standard input, try to parse the input, print the parse tree or an error message. Most of the infrastructure of the program will be the same part 1. Design it right and much of your code can be shared between the programs.

3. Turn your parser into a calculator by *evaluating* the parse tree produced by your parser. That is, write code to traverse a parse tree computing the value of subexpressions as appropriate and printing the final result. You may use either version of your parser—they should produce equivalent parse trees. The result is a command-line calculator like the UNIX program `bc`.

Extra Credit Ideas (10% each; max 20% extra credit)

1. Add variables, assignment statements, and the use of variables in expressions to your system.

2. Add builtin functions (`sin`, *etc.*).

3. Add user-defined functions.

FYI: *The UNIX Programming Environment* by Kernighan and Pike, Chapter 8, describes building such a program using a compiler-generator (`yacc`, now superceded by `bison`). Of course you're writing the parser by hand, but some of the ideas might be useful for these extra credit parts of the project.

Printing Parse Trees

Your programs for parts 1 and 2 of this project must print the resulting parse tree to standard output. There are many ways to do this, but for this project you will produce output in an *indented, pretty-printed format*. What this means is:

- Each node is printed on a separate line.
- The children of a node are indented relative to their parent.
- All children of a node are at the same level of indentation.

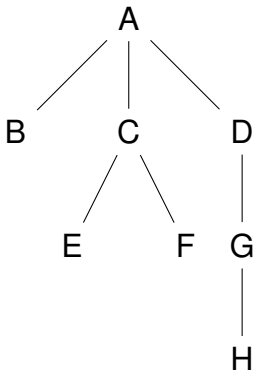
If you are familiar with XML or HTML, this is what "standard" (pretty-printed) XML/HTML looks like, except for the text content of HTML documents, which is implicitly within its enclosing elements.

Additionally for our parse trees:

- A node and its children are preceded by an open parenthesis ("`(`") and followed by a close parenthesis ("`)`").

XML/HTML documents use open and close "tags" rather than parentheses. The use of parentheses makes this output a form of "s-expressions" as used in Lisp.

Example parse tree and corresponding output:




```
(A
  (B)
  (C
    (E)
    (F))
  (D
    (G
      (H))))
```

As for the code, printing a tree involves doing a tree traversal. So it's a recursive function, right? First you print the node, then you print its children in order. The open paren (or tag) comes before a node; a close paren (or tag) comes after the last child.

You also need to keep track of the current indentation level. So this will be a parameter to your printing function. In C, this usually means two functions: a toplevel pretty-print function with no indentation parameter, and an internal function with that parameter, called from the toplevel function with indentation 0 to get the ball rolling.

Project Submission

Your project submission MUST include the following:

1. A README.txt file or PDF document describing:
   (a) How to build your project
   (b) How to run your project's program(s) to demonstrate that it/they meet the requirements
   (c) Any collaborators (see below)
   (d) Acknowledgements for anything you did not code yourself (you should avoid this, other than the code we've given you, which you don't have to acknowledge)
2. All source code for your project, including a `Makefile` or shell script that will build the program per your instructions. TAs may accept Eclipse projects, but you should try not to count on it.

The TAs must be able to cut-and-paste from your documentation in order to build and run your code. The easier you make this for them, the better your grade will be.

Please watch BlackBoard for updates as we may need to make adjustments to these requirements.

Late Policy

Don't be late. But if you are, 5% penalty for the first hour or part thereof, 10% penalty per hour or part thereof after the first.

Course Collaboration Policy

Collaboration on projects is permitted, subject to the following requirements:

- Groups of no more than 3 students, all currently taking CSC173.
- You must be able to explain anything you or your group submit, IN PERSON AT ANY TIME, at the instructor's or TA's discretion.
- One member of the group should submit on the group's behalf and the grade will be shared with other members of the group. Other group members should submit a short comment naming the other collaborators.
- All members of a collaborative group will get the same grade on the project.