CS 316 Fall 2009
All projects in this course must be completed **individually and independently**. All programs must be written in Java or C++.

PROJECT 3: Top-Down Parser
Due: 11/20/09, Friday, Midnight

This project is a continuation of Project 2. You will implement a top-down parser for the following grammar for a small functional language:

<fun defs> → {<fun def>}$^{+}$
<fun def> → <header> <body>
<header> → <fun name> "(" <parameter list> ")"
<fun name> → <id>
<parameter list> → ε | <id> {"," <id>}
<body> → "{" <Exp> "}"
<Exp> → "if" "(" <expr> ")" <Exp> "else" <Exp> | <expr>
<expr> → <boolTerm> { "||" <boolTerm> }
<boolTerm> → <boolPrimary> { "&&" <boolPrimary> }
<boolPrimary> → <E> [<rel op> <E>]
<rel op> → "<" | "<=" | ">" | ">=" | "==" | "!="
<E> → <term> { (+|−) <term> }
<term> → <primary> { (*|/) <primary> }
<primary> → <id> | <int> | <float> | <floatE> | <fun E> "(" <expr> ")" | − <primary>
| ! <primary>
<fun E> → <fun name> "(" <E list> ")"
<E list> → ε | <expr> {"," <expr>}

A lexical analyzer for this grammar's tokens has been implemented in Project 2. The category <expr> extends arithmetic expressions with Boolean and relational operators by incorporating the following precedence (listed from highest to lowest):

```
!, unary -    highest precedence
*, /
+, binary -
<, <=, >, >=, ==, !=
&&
||      lowest precedence
```

The grammar for <expr> allows mistyped expressions like "!316" and "a+b || a−c"; if the source language is statically typed, such type errors would be caught by the compiler's type checking phase. (Type checking is beyond the scope of this project.)

Your program will read any text file that contains (what is intended to be) a string in the category <fun defs>. It will then display the parse tree in linearly indented form: each syntactic category name labeling a node in the parse tree is displayed on a separate line, prefixed with the integer *i* representing the node's depth and indented by *i* blanks. Note that the displayed parse tree records the expansion and contraction of the runtime stack caused by the parsing functions' calls, and the times the runtime stack reaches the peak size can be spotted.

An appropriate error message should be issued when the first syntax error is found; in this project there is no need to recover from it and attempt to find subsequent syntax errors. (Real-world compilers do some type of syntax-error recovery and attempt to find more syntax errors.)

The I/O file names should be entered interactively from the terminal or as external arguments for the main() function.

You may use a sample lexical analyzer for Project 2. You may also modify a sample parser for arithmetic expressions; if you do so, modify the comments suitably as well.

Here's a sample set of test input files:
in1 | out1
in2 | out2
in3 | out3
in4 | out4
in5 | out5
in6 | out6
You should make your own additional input files to test the program. Your outputs don't have to be identical to the samples, but they should display the parse tree structure clearly.

In the next and last project, the parser will be expanded to emit instructions for a stack-based virtual machine.

Submission
Your source program, including the lexical analyzer, must be emailed to yukawa@cs.qc.cuny.edu with the subject header:

CS 316, Project 3, your full name

The due date is 11/20/09, Friday, Midnight. Please include the program as attachment **without compression into .zip, .rar, etc**.