1. Write a YACC program for the grammar in Appendix A. Use the Lex program developed for assignment #1 (or supplied in the solutions) to pass tokens to the Yacc program. Please submit your yacc and your lex program file.
    a. List all the conflicts in the Yacc program and give a discussion on how these conflicts will be resolved.
    b. Output the preorder traversal of the abstract syntax tree of the program. Each line should output one node. You should leave four spaces for each level of the AST. For example, if a node is at the top level, there should not be any spaces before it. If a node is on the second level, four spaces should precede the print info about the node. Do NOT use <alpha>, <alpha_num>, <digit>, <hex_digit> in the AST.

    Example:
    class Program {
    int i = 0;
    }

    program
      field_decl
        type
          int
        id
          i
        literal
          int_literal
            decimal_literal
              0
    (25)

CSE 601 (CSE 301 students can do this for extra credit):

2. Extend the regular expression parser in Assignment 1 to accept or reject strings. It should take as input one regular expression and one string and output “accepted” or “rejected”.

    (25)

Appendix A

<program> -> **class Program {** <field_decl>* <method_decl>* **}**
<field_decl> -> <type> (<id> | <id> **[** <int_literal> **]**) (**,** <id> | <id> **[** <int_literal> **]**)* **;**

<field_decl> -> <type> <id> **=** <literal> **;**
<method_decl> -> (<type> | **void**) <id> **(** ((<type> <id>) (**,** <type> <id>)*)? **)** <block>
<block> -> **{** <var_decl>* <statement>* **}**
<var_decl> -> <type> <id> (**,** <id>)* **;**
<type> -> **int** | **boolean**
<statement> -> <location> <assign_op> <expr> **;**
<statement> -> <method_call> **;**
<statement> -> **if (** <expr> **)** <block> (**else** <block>)?
<statement> -> **for** <id> **=** <expr> **,** <expr> <block>
<statement> -> **return** (<expr>)? **;**
<statement> -> **break ;**
<statement> -> **continue ;**
<statement> -> <block>
<assign_op> -> **=**
<assign_op> -> **+=**
<assign_op> -> **-=**
<method_call> -> <method_name> **(** (<expr> (, <expr>)*)? **)**
<method_call> -> **callout (** <string_literal> (**,** <callout_arg>)* **)**
<method_name> -> <id>
<location> -> <id>
<location> -> <id> **[** <expr> **]**
<expr> -> <location>
<expr> -> <method_call>
<expr> -> <literal>
<expr> -> <expr> <bin_op> <expr>
<expr> -> **-** <expr>
<expr> -> **!** <expr>
<expr> -> **(** <expr> **)**
<callout_arg> -> <expr> | <string_literal>
<bin_op> -> <arith_op> | <rel_op> | <eq_op> | <cond_op>
<arith_op> -> **+** | **-** | ***** | **/** | **%**
<rel_op> -> **<** | **>** | **<=** | **>=**
<eq_op> -> **==** | **!=**
<cond_op> -> **&&** | **||**
<literal> -> <int_literal> | <char_literal> | <bool_literal>
<id> -> <alpha> <alpha_num>*
<alpha> -> [**a**-**z**A-**Z**_]
<alpha_num> -> <alpha> | <digit>
<digit> -> [**0**-**9**]
<hex_digit> -> <digit> | [**a**-**f**A-**F**]
<int_literal> -> <decimal_literal> | <hex_literal>
<decimal_literal> -> <digit> <digit>*
<hex_literal> -> **0x** <hex_digit> <hex_digit>*
<bool_literal> -> **true** | **false**

<char_literal> -> **‘**<char>**’**
<string_literal> -> **“**<char>*”