Consider the following language. It is a tiny, C-like language:

```
<program> -> class Program { <field_decl>* <method_decl>* }
<field_decl> -> <type> (<id> | <id> [ int_literal ] ) ( , <id> | <id> [ int-literal ] )* ;
<field_decl> -> <type> <id> = <constant> ;
<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>
<constant> -> <int_literal> | <bool_literal>
<id> -> <alpha> <alpha_num>*
<alpha> -> [a-zA-Z_]
<alpha_num> -> <alpha> | <digit>
```

<digit> -> [**0**-**9**]
<hex_digit> -> <digit> | [**a**-**fA**-**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>*****”**

The project for this semester is to build a compiler for this language. It will be divided into the following 4 parts:

1. Lexical Analysis, 2 weeks, weight 10%
2. Syntax Analysis, 2 weeks, weight 10%
3. Intermediate Code Generation, 4 weeks, weight 40%
4. Machine Code Generation, 4 weeks, weight 40%

The task is to output the token strings for programs written in the language above. You will be using the *antlr* compiler-builder tool in this class. Write a lexer grammar file for the language given above and test it with programs written in the language given above.

1. TAs will provide a few test programs before the deadline. Please make sure your programs works on those test cases. I encourage you to write test cases of your own as well to get a deeper understanding.
2. The test cases used for grading will not be disclosed before the deadline.
3. We will be checking for plagiarism and following the plagiarism policy on the website. So, feel free to discuss solutions with others, but write the code yourself.