

Write an *syntax-directed translator* (SDT) in ANTLR to generate the *abstract syntax tree* (AST) for a program written in the language described below. Each AST node must be labeled with exactly one word from the following set: Program, FieldDecl, InitFieldDecl, MethodDecl, MethodArg, Block, VarDecl, Assign, Call, If, IfElse, Switch, While, Ret, Break, Cont, UserMeth, ExtMeth, Loc, ArrayLoc, LocExpr, CallExpr, ConstExpr, BinExpr, NegExpr, NotExpr, ExprArg, StringArg, **or, terminal value**.

The AST must be written in pre-order format, and must match the reference output exactly:

```
<str rep of a tree> =  
    {<str rep of parent node>  
    <str reps of child subtrees from left to right>}  
<str rep of a node> = <label of the node>
```

We will test your submission in the following way:

```
unzip <student ID>.zip  
cd <student ID>  
                                     //Your submission folder MUST be named  
                                     <student ID>.  
./build.sh                          //Compile the parser i.e. your folder MUST  
                                     //have a build.sh script that builds the  
                                     //parser.  
grun Cmp379Compiler program <path to test case>  
> <path to output file>  
                                     //Your parser MUST be called  
                                     //Cmp379Compiler. The number and content of  
                                     //grading test cases will not be revealed  
                                     //before the deadline.
```

Grammar:

```
<program>  
-> class Program { <field_decl>* <method_decl>* }  
<field_decl>  
-> <type> (<id> | <id> [ int_literal ] )  
    ( , <id> | <id>[int_literal ] )* ;  
    | <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> ;
    | <method_call> ;
    | if ( <expr> ) <block> ( else <block> )?
    | switch <expr> {(case <literal> : <statement>*)+}
    | while ( <expr> ) <statement>
    | return ( <expr> )? ;
    | break ;
    | continue ;
    | <block>
<assign_op>
    -> =
    | +=
    | -=
<method_call>
    -> <method_name> ( ( <expr> ( , <expr> )*)? )
    | callout ( <string_literal> ( , <callout_arg> )* )
<method_name>
    -> <id>
<location>
    -> <id>
    | <id> [ <expr> ]
<expr>
    -> <location>
    | <method_call>
    | <literal>
    | <expr> <bin_op> <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-zA-Z_]
<alpha_num>
    -> <alpha>
    | <digit>
<digit>
    -> [0-9]
<hex_digit>
    -> <digit>
    | [a-fA-F]
<int_literal>
    -> <decimal_literal>
    | <hex_literal>
<decimal_literal>
    -> <digit>+
<hex_literal>
    -> 0x <hex_digit>+
<bool_literal>
    -> true
    | false
<char_literal>
    -> '<char>'
<string_literal>
    -> "<char>*"

```