Question 1.

Given the description of a program in mC as follows:

A program in mC consists of many declarations, which are variable and function declarations.

A variable declaration starts with a type, which is *int* or *float*, then a comma-separated list of identifiers and ends with a semicolon.

A function declaration also start with a type and then an identifier, which is the function name, and then parameter declaration and ends with a body. The parameter declaration starts with a left round bracket '(' and a null-able semicolon-separated list of parameters and ends with a right round bracket ')'. Each parameter always starts with a type and then a comma-separated list of identifier. A body starts with a left curly bracket '{', follows by a null-able list of variable declarations or statements and ends with a right curly bracket '}'.

There are 3 kinds of statements: assignment, call and return. All statements must end with a semicolon. An assignment statement starts with an identifier, then an equal '=', then an expression. A call starts with an identifier and then follows by a null-able comma-separated list of expressions enclosed by round brackets. A return statement starts with a symbol 'return' and then an expression.

An expression is a construct which is made up of operators and operands. They calculate on their operands and return new value. There are four kinds of infix operators: '+', '-', '*' and '/' where '+' have lower precedence than '-' while '*' and '/' have the highest precedence among these operators. The '+' operator is right associative, '-' is non-associative while '*' and '/' is left-associative. To change the precedence, a sub-expression is enclosed in round brackets. The operands can be an integer literal, float literal, an identifier, a call or a sub-expression.

For example,

```
int a,b,c;
float foo(int a;float c,d) {
        int e;
        e = a + 4;
        c = a * d / 2.0;
        return c + 1;
}
float goo(float a,b) {
        return foo(1,a,b);
}
```

The following tokens can be used for the grammar:

ID (for identifiers), **INTLIT** (for integer literals), **FLOATLIT** (for float literals), **INT**, **FLOAT**, **RETURN**, **LB** (for '{'), **RB** (for '}') , **SM** (for ';'), **CM** (for ','), **EQ** (for '='), **LP** (for '('), **RP** (for ')'), **ADD** (for '+'), **SUB** (for '-'), **MUL** (for '*'), **DIV** (for '/').

a. Write the grammar of a program in mC in BNF format.

b. Write a recognizer in ANTLR to detect if a mC program is written correctly or not

prog -> manyDecl
manyDecl -> decl manyDecl | decl
decl -> varDecl | funcDecl
varDecl -> type idsList SM
type -> INT | FLOAT
idsList -> ID CM idsList | ID
funcDecl -> type ID paramDecl body
paramDecl -> LP paramsList RP
paramsList -> nonNullParamsList | eps
nonNullParamsList -> param SM nonNullParamsList | param
param -> type idsList
body -> LB varDeclsOrStmtsList RP
varDeclsOrStmtsList -> nonNullVarDeclsOrStmtsList | eps
nonNullVarDeclsOrStmtsList -> varDeclOrStmt nonNullVarDeclsOrStmtsList | varDeclOrStmt
varDeclOrStmt -> varDecl | stmt

stmt -> assStmt | callStmt | retStmt
assStmt -> ID EQ exp SM
callStmt -> ID LP expsList RP SM
retStmt -> RETURN exp SM
expsList -> nonNullExpsList | eps
nonNullExpsList -> exp CM nonNullExpsList | exp

exp -> exp1 ADD exp | exp1
exp1 -> exp2 SUB exp2 | exp2
exp2 -> exp2 MUL exp3 | exp2 DIV exp3 | exp3
exp3 -> operands
operands -> INTLIT | FLOATLIT | id | call | subExp
subExp -> LP exp RP