Syntax Analyzer for Toy language
Instructions

You are to implement a syntax analyzer for the programming language Toy, as defined in project #1. You should first design a CFG G for Toy based on the following Backus Normal Form (BNF) description, and then write a program to (1) create a parsing table for G, and (2) perform a one-symbol lookahead parsing on various input Toy programs and print appropriate parsing actions.

The grammar of Toy is given in a variant of extended BNF. The meta-notation used:

x	means that x is a terminal i.e. a token. All terminal names are lowercase.
X	(in italic) means *X* is a nonterminal. All nonterminal names are capitalized.
<x>	means zero or one occurrence of x, i.e., x is optional
x*	means zero or more occurrences of x
x+	means one or more occurrences of x
x+,	a comma-separated list of one or more x's (commas appear only between x's)
|	separates production alternatives

For readability, we represent operators by the lexeme that denotes them, such as + or != as opposed to the token (_plus, _notequal) returned by the scanner.

1. *Program* ::= *Decl*+
2. *Decl* ::= *VariableDecl* | *FunctionDecl* | *ClassDecl* | *InterfaceDecl*
3. *VariableDecl* ::= *Variable* ;
4. *Variable* ::= *Type* id
5. *Type* ::= int | double | boolean | string | *Type* [] | id
6. *FunctionDecl* ::= *Type* id (*Formals*) *StmtBlock* | void id (*Formals*) *StmtBlock*
7. *Formals* ::= *Variable*+, | ε
8. *ClassDecl* ::= class id <extends id> < implements id+, > { *Field** }
9. *Field* ::= *VariableDecl* | *FunctionDecl*
10. *InterfaceDecl* ::= interface id { *Prototype** }
11. *Prototype* ::= *Type* id (*Formals*) ; | void id (*Formals*) ;
12. *StmtBlock* ::= { *VariableDecl** *Stmt** }
13. *Stmt* ::= <*Expr*> ; | *IfStmt* | *WhileStmt* | *ForStmt* | *BreakStmt* | *ReturnStmt* | *PrintStmt* | *StmtBlock*
14. *IfStmt* ::= if (*Expr*) *Stmt* <else *Stmt*>
15. *WhileStmt* ::= while (*Expr*) *Stmt*
16. *ForStmt* ::= for (<*Expr*> ; *Expr* ; <*Expr*>) *Stmt*
17. *BreakStmt* ::= break ;
18. *ReturnStmt* ::= return <*Expr*> ;
19. *PrintStmt* ::= println (*Expr*+,) ;
20. *Expr* ::= *Lvalue* = *Expr* | *Constant* | *Lvalue* | *Call* | (*Expr*) |
    Expr + *Expr* | *Expr* – *Expr* | *Expr* * *Expr* | *Expr* / *Expr* | *Expr* % *Expr* | - *Expr* |
    Expr < *Expr* | *Expr* <= *Expr* | *Expr* > *Expr* | *Expr* >= *Expr* |
    Expr == *Expr* | *Expr* != *Expr* | *Expr* && *Expr* | *Expr* || *Expr* | ! *Expr*
    readln () | new (id) | newarray (intconstant , *Type*)
21. *Lvalue* ::= id | *Lvalue* [*Expr*] | *Lvalue* . id
22. *Call* ::= id (*Actuals*) | id . id (*Actuals*)
23. *Actuals* ::= *Expr*+, | ε
24. *Constant* ::= intconstant | doubleconstant | stringconstant | booleanconstant | null

Operator precedence from highest to lowest:

[.	(array indexing and field selection)
! -	(logical not, unary minus)
* / %	(multiply, divide, mod)
+ -	(addition, subtraction)
< <= > >=	(relational)
== !=	(equality)
&&	(logical and)
||	(logical or)
=	(assignment)

- All binary arithmetic and logical operators are left-associative.
- The assignment and relational operators are not associate, which means we cannot chain a sequence of operators that are on the same precedence level. For example, a < b >= c should not parse, but a < b == c is allowed.
- Parentheses may override precedence and associativity.

For every input Toy program, you should print out a sequence of tokens generated by your first project (the lexical analyzer), and print out the action (either "shift" or "reduce") decided by your parser for each token. If the action is "reduce", also print out the production number. For instance, given the following CFG:

1. S → a X c
2. X → b X
3. X → b
4. X → Y d
5. Y → Y d
6. Y → d

The sample output for string ***abbbc*** should be

> a [shift]
> b [shift]
> b [shift]
> b [shift]
> c [reduce 3][reduce 2][reduce 2][shift]
> [reduce 1]
> [accept]

and the output for ***abcd*** should be

> a [shift]
> b [shift]
> c [reduce 3][shift]
> d [reduce 1]
> [reject]

	Input	Expected outcome	Program result
1	`void f (double x, double y) { }`		
2	`int i = 1;`		
3	`// in function` `result = 5.times(4);`		
4	`// in function` `front = in.nextLine();`		
5	`// in function` `front = in.nextLine;`		
6	`int[][][] super;`		
7	`a[3][4.5][b] = result = x + y + z;`		
8	`// in function` `for ( ; ; ) x = 1;`		

9	`// in function`
	`if (h>w) g=h;`
	`else h=g;`
	`double a;`		

10	`// in class`
	`boolean b;`
	`userDefinedClassType f(){}`
	`double d;`
	`string g(){}`		

11	the sample Toy program given in project #1		
12	design your own test case		
13	design your own test case		
14	design your own test case		