Lab Assignment 7

Lexical analysis and parsing using Lex/Flex and Yacc/Bison

1. Consider the following *context-free grammar*:
   Terminals:
   `and (AND) := (ASSIGN) : (COLON) , (COMMA) def (DEF) / (DIV)`
   `. (DOT) else (ELSE) end (END) = (EQ) exit (EXITLOOP) float (FLOAT)`
   `(FLOAT\_CONST) (FORMAT) from (FROM) fun (FUN) >= (GE) global (GLOBAL)`
   `> (GT) (ID) if (IF) int (INT) (INT\_CONST) ( (LEFT\_PAREN)`
   `[ (LEFT\_SQ\_BKT) <= (LE) < (LT) - (MINUS) mod (MOD) * (MULT) <> (NE)`
   `not (NOT) null (NUL) or (OR) + (PLUS) print (PRINT) product (PRODUCT)`
   `read (READ) return (RETURN) -> (RETURNS) ) (RIGHT\_PAREN)`
   `] (RIGHT\_SQ\_BKT) ; (SEMICOLON) skip (SKIP) step (STEP) (STRING) to (TO)`
   `while (WHILE)`

   Note: `ID`, `INT_CONST`, `FLOAT_CONST` are identifier, integer constant and floating point constant. `STRING` is a string constant. `FORMAT` are `%d %f %s` ⋯.

   Non-terminals:
   `prog declList decl typeList varList var sizeList0 sizeList type`
   `typeDef funDef funID fparamList0 pList idP funBody stmtList stmtList0`
   `stmt assignmentStmt dotId readStmt printStmt ifStmt elsePart whileStmt`
   `loopStmt stepPart callStmt returnStmt exp0 exitLoop skip id indxList0`
   `indxList bExp relOP exp actParamList0 actParamList`

   Start symbol: `prog`

Production Rules

prog	→	GLOBAL declList stmtListO END
declList	→	decl declList
	→	ε
decl	→	DEF typeList END
	→	FUN funDef END
typeList	→	typeList SEMICOLON varList COLON type
	→	varList COLON type
	→	typeDef
varList	→	var COMMA varList
	→	var
var	→	ID sizeListO
sizeListO	→	sizeList
	→	ε
sizeList	→	sizeList LEFT_SQ_BKT INT_CONST RIGHT_SQ_BKT
	→	LEFT_SQ_BKT INT_CONST RIGHT_SQ_BKT
type	→	INT
	→	FLOAT
	→	STRING
	→	NUL
	→	typeDef
	→	ID
typeDef	→	ID ASSIGN PRODUCT typeList END
funDef	→	funID fparamListO RETURNS type funBody
funID	→	ID
fparamListO	→	fparamList
	→	ε
fparamList	→	fparamList SEMICOLON pList COLON type
	→	pList COLON type
pList	→	pList COMMA idP
	→	idP

```
          idP  →  ID sizeListO
      funBody  →  declList stmtListO

    stmtListO  →  stmtList
               →  ε

     stmtList  →  stmtList SEMICOLON stmt
               →  stmt

         stmt  →  assignmentStmt
               →  readStmt
               →  printStmt
               →  ifStmt
               →  whileStmt
               →  loopStmt
               →  callStmt
               →  returnStmt
               →  exitLoop
               →  skip

assignmentStmt →  dotId ASSIGN exp

        dotId  →  id
               →  id DOT dotId

     readStmt  →  READ FORMAT exp

    printStmt  →  PRINT STRING
               →  PRINT FORMAT exp

       ifStmt  →  IF bExp COLON stmtList elsePart END

     elsePart  →  ELSE stmtList
               →  ε

    whileStmt  →  WHILE bExp COLON stmtList END

     loopStmt  →  FROM id ASSIGN exp TO exp stepPart COLON stmtListO END

     stepPart  →  STEP exp
               →  ε

     callStmt  →  LEFT_PAREN ID COLON actParamList RIGHT_PAREN

   returnStmt  →  RETURN expO

         expO  →  exp
               →  ε

     exitLoop  →  EXITLOOP
```

```
        skip  →  SKIP

          id  →  ID indxListO

   indxListO  →  indxList
              →  ε

    indxList  →  indxList LEFT_SQ_BKT exp RIGHT_SQ_BKT
              →  LEFT_SQ_BKT exp RIGHT_SQ_BKT

        bExp  →  bExp OR bExp
              →  bExp AND bExp
              →  NOT bExp
              →  LEFT_PAREN bExp RIGHT_PAREN
              →  exp relOP exp

       relOP  →  EQ
              →  LE
              →  LT
              →  GE
              →  GT
              →  NE

         exp  →  exp PLUS exp
              →  exp MINUS exp
              →  exp MULT exp
              →  exp DIV exp
              →  exp MOD exp
              →  MINUS exp
              →  PLUS exp
              →  exp DOT exp
              →  LEFT_PAREN exp RIGHT_PAREN
              →  id
              →  LEFT_PAREN ID COLON actParamListO RIGHT_PAREN
              →  INT_CONST
              →  FLOAT_CONST

actParamListO →  actParamList
              →  ε

 actParamList →  actParamList COMMA exp
              →  exp
```

2. Comment in the language is `//  ....`, up to the end of the line.
3. Opeartor precedence: $\{+-\} < \{*/ \text{ mod}\} < \{-_u +_u\} < \cdot$
4. Write *flex-bison* specification for parsing the complete language. There should not be any reported conflict. The precedence of the operators are usual.