CS39003 Compiler Lab
Assignment 8

LALR Parser using Flex-Bison

Submission Deadline: 1st November 2013 **Marks: 40**

1. Consider the following context-free grammar:

Terminals:

Expression	**Symbol in Grammar**
`and`	`AND`
`:=`	`ASSIGN`
`:`	`COLON`
`,`	`COMMA`
`def`	`DEF`
`else`	`ELSE`
`/`	`DIV`
`end`	`END`
`=`	`EQ`
`float`	`FLOAT`
`>=`	`GE`
`global`	`GLOBAL`
`>`	`GT`
`if`	`IF`
`int`	`INT`
`(`	`LEFT_PAREN`
`[`	`LEFT_SQ_BKT`
`<=`	`LE`
`<`	`LT`
`-`	`MINUS`
`%`	`MOD`
`*`	`MULT`
`<>`	`NE`
`not`	`NOT`
`null`	`NUL`
`or`	`OR`
`+`	`PLUS`
`print`	`PRINT`
`product`	`PRODUCT`
`read`	`READ`
`return`	`RETURN`
`)`	`RIGHT_PAREN`
`]`	`RIGHT_SQ_BKT`
`;`	`SEMICOLON`
`while`	`WHILE`

Note: `ID, INT_CONST, FLOAT_CONST` are identifier, integer constant and floating point constant, as mentioned in Assignment 4. `FORMAT` are `%d %f %s`

Non-terminals:

```
prog declList decl typeList varList var sizeListO sizeList type
typeDef
stmtListO stmt assignmentStmt dotId readStmt
printStmt ifStmt elsePart whileStmt returnStmt
expO id indexListO indexList bExp relOp exp actParamListO actParamList
```

Start symbol: `prog`

Production Rules

```
prog       -->  GLOBAL declList stmtListO END
declList    -->  decl declList
            --> epsilon
decl        -->  DEF typeList END
typeList    --> typeList SEMICOLON varList COLON type
            --> typeList SEMICOLON typeDef
            -->  varList COLON type
            -->  typeDef
varList     -->  var COMMA varList
            -->  var
var         --> ID sizeListO
sizeListO   -->  sizeList
            -->  epsilon
sizeList    -->  sizeList LEFT_SQ_BKT INT_CONST RIGHT_SQ_BKT
            -->  LEFT_SQ_BKT INT_CONST RIGHT_SQ_BKT
type        --> INT
            --> FLOAT
            -->  NUL
            -->  ID
typeDef     --> ID ASSIGN PRODUCT typeList END
stmtListO   -->  stmtList
            -->  epsilon
stmtList    --> stmtList SEMICOLON stmt
            -->  stmt
stmt        --> assignmentStmt
            -->  readStmt
            -->  printStmt
            -->  ifStmt
            --> whileStmt
            -->  returnStmt
assignmentStmt --> dotId ASSIGN exp
dotId       --> id
            -->  id DOT dotId
readStmt    --> READ FORMAT exp
printStmt   --> PRINT FORMAT exp
ifStmt      -->  IF bExp COLON stmtList elsePart END
elsePart    -->  ELSE stmtList
            --> epsilon
whileStmt   -->  WHILE bExp COLON stmtList END
returnStmt -->  RETURN expO
```

```
expO        -->  exp
            -->  epsilon
id          -->  ID indxListO
indxListO  --> indxList
            --> epsilon
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
            -->  exp DOT exp
            -->  LEFT_PAREN exp RIGHT_PAREN
            -->  id
            -->  LEFT_PAREN ID COLON actParamListO RIGHT_PAREN
            -->  INT_CONST
            -->  FLOAT_CONST
actParamListO --> actParamList
               -->  epsilon
actParamList --> actParamList COMMA exp
              -->  exp
```

2. Comment in the language is `// ....`, up to the end of the line.

3. Operator precedence: {+–} < {* / %} < { . }

4. Write flex-bison specification for parsing the complete language. There should not be any reported conflict.

5. You have three source files: <group-no>`.8.l`, <group-no>`.8.y` and `Makefile`. The name of the executable file should be `lalrParser`.

6. Run the parser on the sample input and generate and submit the output traces using **verbose** mode in Bison. Comment all relevant lines in the output trace.

7. Prepare a `tar`-archive with the name <group-no>`.8.tar` containing the `Makefile`, <group-no>`.8.y`, <group-no>`.8.l`, <group-no>`.8.output`.

Sample Input:

```
global
def        // Definitions
      a:int;
      b:int;
      sum:float;
      point := product
            xpos:float;
            ypos:float
      end
end
a:=1;
sum:=1.0;
point.xpos:=2.331;
point.ypos:=sum+a;
read %d b;
if b = 0 :
      print %f point.xpos;
else
      print %f point.ypos;
end
while a < b:
      a:= a * 2;
      sum := sum + 1;
end
return sum;
end
```