Proj2 bison file

```
%{
 #include <stdio.h>
 #include <stdlib.h>
 #include "y.tab.h"
 extern const char* yytext;
 int yylex();
 void yyerror(const char *msg);
 extern int currLine;
 extern int currPos;
 FILE * yyin;
%}

%union{
char* cval;
int ival;
}
//type
/*
%type <ival> NUMBER
%type <cval> IDENT
*/

%error-verbose
%start program
%token FUNCTION
%token <ival> NUMBER
%token <cval> IDENT
%token BEGIN_PARAMS END_PARAMS
%token BEGIN_LOCALS END_LOCALS
%token BEGIN_BODY END_BODY
%token INTEGER
%token ARRAY
%token OF
%token IF THEN ENDIF ELSE
%token WHILE DO
%token FOR
%token BEGINLOOP ENDLOOP
%token CONTINUE
%token READ WRITE
%token TRUE FALSE
%token RETURN
%token COMMA SEMICOLON COLON
%token MOD
//operators
%right ASSIGN
%left OR
%left AND
```

```
%right NOT
%left EQ NEQ LT GT LTE GTE
%left SUB ADD
%left MULT DIV
%right UMINUS
%left L_PAREN R_PAREN
%left L_SQUARE_BRACKET R_SQUARE_BRACKET


%%
//program (correct)
program : /*epsilon */ {printf("program -> Epsilon\n");}
      |program function {printf("program -> program function\n");}
      ;
//function
function : FUNCTION ident SEMICOLON BEGIN_PARAMS declarations END_PARAMS BEGIN_LOCALS decl
arations END_LOCALS BEGIN_BODY statements END_BODY{printf("function -> FUNCTION ident SEMI
COLON BEGIN_PARAMS declarations END_PARAMS BEGIN_LOCALS declarations END_LOCALS BEGIN_BODY
statements END_BODY\n");}
      ;
//ident
ident : IDENT {printf("ident -> IDENT %s\n", $1);}
      ;
//declaration (changed the ident comma loop) (added declaration loop)
declaration : declaration_loop COLON INTEGER  {printf("declaration -> declaration_loop COL
ON INTEGER\n");}
      | declaration_loop COLON ARRAY L_SQUARE_BRACKET NUMBER R_SQUARE_BRACKET OF INTEGER
 {printf("declaration -> declaration_loop COLON ARRAY L_PAREN NUMBER R_PAREN OF INTEGER
\n"); }
      |  declaration_loop COLON ARRAY L_SQUARE_BRACKET NUMBER R_SQUARE_BRACKET L_SQUARE_BR
ACKET NUMBER R_SQUARE_BRACKET OF INTEGER {printf(" declaration -> declaration_loop COLON A
RRAY L_PAREN NUMBER R_PAREN L_PAREN NUMBER R_PAREN OF INTEGER\n"); }
      ;
declaration_loop : ident {printf("delcaration_loop -> ident\n");}
| declaration_loop COMMA ident {printf("declaration_loop -> declaration_loop COMMA ident
\n");}
;
//declarations
declarations : declarations declaration SEMICOLON{printf("declarations -> declarations dec
laration SEMICOLON \n");}
      | /* */ {printf("declarations -> epsilon\n");}
      ;
//statements
statements : statements statement SEMICOLON {printf("statements -> statements statement SE
MICOLON\n");}
      | {printf("statements -> epsilon\n");}
      ;
statement : var ASSIGN expression {printf("statement -> var ASSIGN expression\n");}
      | IF bool_exp THEN statements ENDIF {printf("statement -> IF bool_exp THEN statement
s ENDIF\n");}
      | IF bool_exp THEN statements ELSE statements ENDIF {printf("statement -> IF bool_ex
p THEN statements ELSE statements ENDIF\n");}
      | WHILE bool_exp BEGINLOOP statements ENDLOOP {printf("statement -> WHILE bool_exp B
EGINLOOP statements ENDLOOP\n");}
```

```
      | DO BEGINLOOP statements ENDLOOP WHILE bool_exp {printf("statement -> DO BEGINLOOP
 statements ENDLOOP WHILE bool_exp\n");}
      | FOR var ASSIGN NUMBER SEMICOLON bool_exp SEMICOLON var ASSIGN expression BEGINLOOP
statements ENDLOOP {printf("statement -> FOR var ASSIGN NUMBER SEMICOLON bool_exp SEMICOLO
N var ASSIGN expression BEGINLOOP statements ENDLOOP\n");}
      | READ vars {printf("statement -> READ vars\n");}
      | WRITE vars {printf("statement -> WRITE vars\n");}
      | CONTINUE {printf("statement -> CONTINUE\n");}
      | RETURN expression {printf("statement -> RETURN expression\n");}
      ;

//bool_exp (good)
bool_exp : relation_and_exp {printf("bool_exp -> relation_and_exp\n");}
      | bool_exp OR relation_and_exp {printf("bool_exp -> bool_exp OR relation_and_exp
\n");}
      ;

//relation_and_exp (good)
relation_and_exp : relation_exp {printf("relation_and_exp -> relation_exp\n");}
      | relation_and_exp AND relation_exp {printf("relation_and_exp -> relation_and_exp AN
D relation_exp\n");}
      ;

//relation_exp (good)
relation_exp : expression comp expression {printf("relation_and_exp -> expression comp exp
ression\n");}
      | TRUE {printf("relation_and_exp -> TRUE\n");}
      | FALSE {printf("relation_and_exp -> FALSE\n");}
      | L_PAREN bool_exp R_PAREN {printf("relation_and_exp -> L_PAREN bool_exp R_PAREN
\n");}
      | NOT expression comp expression {printf("relation_and_exp -> NOT expression comp ex
pression\n");}
      | NOT TRUE {printf("relation_and_exp -> NOT TRUE\n");}
      | NOT FALSE {printf("relation_and_exp -> NOT FALSE\n");}
      | NOT L_PAREN bool_exp R_PAREN {printf("relation_and_exp -> NOT L_PAREN bool_exp R_P
AREN\n");}
      ;

//comp (good)
comp : EQ {printf("comp -> EQ\n");}
      | NEQ{printf("comp -> NEQ\n");}
      | LT {printf("comp -> LT\n");}
      | GT {printf("comp -> GT\n");}
      | LTE{printf("comp -> LTE\n");}
      | GTE {printf("comp -> GTE\n");}
      ;

//expressions (good) (removed epsilon and changed rule)
expressions : expression {printf("expressions -> expression\n");}
| expressions COMMA expression {printf("expressions -> expressions comma expression\n");}
      ;
//expression  (good)
expression : multiplicative_expression {printf("expression -> multiplicative_expression
\n");}
```

```
      | expression SUB multiplicative_expression {printf("expression -> multiplicative_exp
ression SUB multiplicative_expression\n");}
      | expression ADD multiplicative_expression {printf("expression -> multiplicative_exp
ression ADD multiplicative_expression\n");}
      ;

//multiplicative_expression (sounds good)
multiplicative_expression : term {printf("multiplicative_expression -> term\n");}
      | multiplicative_expression MULT term {printf("multiplicative_expression -> term MUL
T term\n");}
      | multiplicative_expression DIV term{printf("multiplicative_expression -> term DIV t
erm\n");}
      | multiplicative_expression MOD term {printf("multiplicative_expression -> term MOD
 term\n");}
      ;

//term (good)
term : var {printf("term -> var\n");}
      | number {printf("term -> number\n");}
      | L_PAREN expression R_PAREN {printf("term -> L_PAREN expression R_PAREN\n");}
      | ident L_PAREN expressions R_PAREN {printf("term -> ident L_PAREN expressions R_PAR
EN\n");}
      | SUB var %prec UMINUS{printf("term -> SUB var\n");}
      | SUB number %prec UMINUS {printf("term -> SUB number\n");}
      | SUB L_PAREN expression R_PAREN %prec UMINUS {printf("term -> SUB L_PAREN expressio
n R_PAREN\n");}
      ;

//vars (Changed to left recursion)
vars : var {printf("vars -> var\n");}
      | vars COMMA var{printf("vars -> vars COMMA var\n");}
      ;
//var
var : ident {printf("var -> ident\n");}
      | ident L_SQUARE_BRACKET expression R_SQUARE_BRACKET {printf("var -> ident L_SQUARE_
BRACKET expression R_SQUARE_BRACKET\n");}
      | ident L_SQUARE_BRACKET expression R_SQUARE_BRACKET L_SQUARE_BRACKET expression R_S
QUARE_BRACKET {printf("var -> ident L_SQUARE_BRACKET expression R_SQUARE_BRACKET L_SQUARE_
BRACKET expression R_SQUARE_BRACKET\n");}
      ;
//number
number : NUMBER {printf("number -> NUMBER %d\n",$1);}
      ;

%%

int main(int argc, char **argv) {
   if (argc > 1) {
      yyin = fopen(argv[1], "r");
      if (yyin == NULL){
         printf("syntax: %s filename\n", argv[0]);
      }//end if
   }//end if
   yyparse(); // Calls yylex() for tokens.
```

```
   return 0;
}

void yyerror(const char *msg) {
   printf("** Line %d, position %d: %s\n", currLine, currPos, msg);
}
```

```
%{
 #include <stdio.h>
 #include <stdlib.h>
 #include "y.tab.h"
 extern const char* yytext;
 int yylex();
 void yyerror(const char *msg);
 extern int currLine;
 extern int currPos;
 FILE * yyin;
%}

%union{
char* cval;
int ival;
}
//type
/*
%type <ival> NUMBER
%type <cval> IDENT
*/

%error-verbose
%start program
%token FUNCTION
%token <ival> NUMBER
%token <cval> IDENT
%token BEGIN_PARAMS END_PARAMS
%token BEGIN_LOCALS END_LOCALS
%token BEGIN_BODY END_BODY
%token INTEGER
%token ARRAY
%token OF
%token IF THEN ENDIF ELSE
%token WHILE DO
%token FOR
%token BEGINLOOP ENDLOOP
%token CONTINUE
%token READ WRITE
%token TRUE FALSE
%token RETURN
%token COMMA SEMICOLON COLON
```

```
%token MOD
//operators
%right ASSIGN
%left OR
%left AND
%right NOT
%left EQ NEQ LT GT LTE GTE
%left SUB ADD
%left MULT DIV
%right UMINUS
%left L_PAREN R_PAREN
%left L_SQUARE_BRACKET R_SQUARE_BRACKET


%%
//program (correct)
program : /*epsilon */ {printf("program -> epsilon\n");}
                        | program function {printf("program -> program function\n");}
                        ;
//function
function : FUNCTION ident SEMICOLON BEGIN_PARAMS declarations END_PARAMS BEGIN_LOCALS decl
arations END_LOCALS BEGIN_BODY statements END_BODY{ printf("function -> FUNCTION ident SEM
ICOLON BEGIN_PARAMS declarations END_PARAMS BEGIN_LOCALS declarations END_LOCALS BEGIN_BOD
Y statements END_BODY\n"); }
                        | FUNCTION error {printf("****invalid function\n");}
                        ;
//ident
ident : IDENT {printf("ident -> IDENT %s\n", $1);}
                        ;
//declaration (changed the ident comma loop) (added declaration loop)
declaration : declaration_loop COLON INTEGER  {printf("declaration -> declaration_loop COL
ON INTEGER\n");}
                        | declaration_loop COLON ARRAY L_SQUARE_BRACKET NUMBER R_SQUARE_BR
ACKET OF INTEGER {printf("declaration -> declaration_loop COLON ARRAY L_SQUARE_BRACKET NUM
BER R_SQUARE_BRACKET OF INTEGER\n"); }
                        |  declaration_loop COLON ARRAY L_SQUARE_BRACKET NUMBER R_SQUARE_B
RACKET L_SQUARE_BRACKET NUMBER R_SQUARE_BRACKET OF INTEGER {printf(" declaration -> declar
ation_loop COLON ARRAY L_SQUARE_BRACKET NUMBER R_SQUARE_BRACKET L_SQUARE_BRACKET NUMBER R_
SQUARE_BRACKET OF INTEGER\n"); }
                        ;
declaration_loop : ident {printf("delcaration_loop -> ident\n");}
                        | declaration_loop COMMA ident {printf("declaration_loop -> declar
ation_loop COMMA ident\n");}
                        ;
//declarations
declarations : declarations declaration SEMICOLON{printf("declarations -> declarations dec
laration SEMICOLON \n");}
                        | /* */ {printf("declarations -> epsilon\n");}
                        | declarations declaration error{printf("*****invalid decs\n");}
                        ;
//statements
statements : statements statement SEMICOLON {printf("statements -> statements statement SE
MICOLON\n");}
                        | {printf("statements -> epsilon\n");}
```

```
                        //| statements error{printf("statements error\n");}
                        ;
statement : var ASSIGN expression {printf("statement -> var ASSIGN expression\n");}
                        | IF bool_exp THEN statements ENDIF {printf("statement -> IF bool_
exp THEN statements ENDIF\n");}
                        | IF bool_exp THEN statements ELSE statements ENDIF {printf("state
ment -> IF bool_exp THEN statements ELSE statements ENDIF\n");}
                        | WHILE bool_exp BEGINLOOP statements ENDLOOP {printf("statement -
> WHILE bool_exp BEGINLOOP statements ENDLOOP\n");}
                        | DO BEGINLOOP statements ENDLOOP WHILE bool_exp {printf("statemen
t -> DO BEGINLOOP statements ENDLOOP WHILE bool_exp\n");}
                        | FOR var ASSIGN NUMBER SEMICOLON bool_exp SEMICOLON var ASSIGN ex
pression BEGINLOOP statements ENDLOOP {printf("statement -> FOR var ASSIGN NUMBER SEMICOLO
N bool_exp SEMICOLON var ASSIGN expression BEGINLOOP statements ENDLOOP\n");}
                        | READ vars {printf("statement -> READ vars\n");}
                        | WRITE vars {printf("statement -> WRITE vars\n");}
                        | CONTINUE {printf("statement -> CONTINUE\n");}
                        | RETURN expression {printf("statement -> RETURN expression\n");}
                        | DO error{printf("****DO statement exp error\n");}
                        | IF error{printf("****IF statement exp error\n");}
                        | WHILE error{printf("****IF statement exp error\n");}
                        | FOR error{printf("****IF statement exp error\n");}
                        | var error{printf("**** var statement exp error\n");}
                        //| READ error{printf("**** READ statement exp error\n");}
                        ;

//bool_exp (good)
bool_exp : relation_and_exp {printf("bool_exp -> relation_and_exp\n");}
                        | bool_exp OR relation_and_exp {printf("bool_exp -> bool_exp OR re
lation_and_exp\n");}
                        //rr| error{printf("****bool exp error\n");}
                        ;

//relation_and_exp (good)
relation_and_exp : relation_exp {printf("relation_and_exp -> relation_exp\n");}
                        | relation_and_exp AND relation_exp {printf("relation_and_exp -> r
elation_and_exp AND relation_exp\n");}
                        //| error{printf("****relation exp error\n");}
                        ;

//relation_exp (good)
relation_exp : expression comp expression {printf("relation_and_exp -> expression comp exp
ression\n");}
                        | TRUE {printf("relation_and_exp -> TRUE\n");}
                        | FALSE {printf("relation_and_exp -> FALSE\n");}
                        | L_PAREN bool_exp R_PAREN {printf("relation_and_exp -> L_PAREN bo
ol_exp R_PAREN\n");}
| NOT expression comp expression {printf("relation_and_exp -> NOT expression comp expressi
on\n");}
                        | NOT TRUE {printf("relation_and_exp -> NOT TRUE\n");}
                        | NOT FALSE {printf("relation_and_exp -> NOT FALSE\n");}
                        | NOT L_PAREN bool_exp R_PAREN {printf("relation_and_exp -> NOT L_
PAREN bool_exp R_PAREN\n");}
                        | error{printf("****relation exp error\n");}
```

```
                        ;

//comp (good)
comp : EQ {printf("comp -> EQ\n");}
                        | NEQ{printf("comp -> NEQ\n");}
                        | LT {printf("comp -> LT\n");}
                        | GT {printf("comp -> GT\n");}
                        | LTE{printf("comp -> LTE\n");}
                        | GTE {printf("comp -> GTE\n");}
                        | error{printf("****comp exp error\n");}
                        ;

//expressions (good) (removed epsilon and changed rule)
expressions : expression {printf("expressions -> expression\n");}
| expressions COMMA expression {printf("expressions -> expressions comma expression\n");}
                        //| error{printf("**exp error\n");}
                        ;
//expression  (good)
expression : multiplicative_expression {printf("expression -> multiplicative_expression
\n");}
                        | expression SUB multiplicative_expression {printf("expression ->
 multiplicative_expression SUB multiplicative_expression\n");}
                        | expression ADD multiplicative_expression {printf("expression ->
 multiplicative_expression ADD multiplicative_expression\n");}
                        //rr| error{printf("****bool exp error\n");}
                        ;

//multiplicative_expression (sounds good)
multiplicative_expression : term {printf("multiplicative_expression -> term\n");}
                        | multiplicative_expression MULT term {printf("multiplicative_expr
ession -> term MULT term\n");}
                        | multiplicative_expression DIV term{printf("multiplicative_expres
sion -> term DIV term\n");}
                        | multiplicative_expression MOD term {printf("multiplicative_expre
ssion -> term MOD term\n");}
//| error{printf("mult exp error\n");}
                        ;

//term (good)
term : var {printf("term -> var\n");}
                        | number {printf("term -> number\n");}
                        | L_PAREN expression R_PAREN {printf("term -> L_PAREN expression R
_PAREN\n");}
                        | ident L_PAREN expressions R_PAREN {printf("term -> ident L_PAREN
expressions R_PAREN\n");}
                        | SUB var %prec UMINUS{printf("term -> SUB var\n");}
                        | SUB number %prec UMINUS {printf("term -> SUB number\n");}
                        | SUB L_PAREN expression R_PAREN %prec UMINUS {printf("term -> SUB
L_PAREN expression R_PAREN\n");}
                        //| error{printf("term error\n");}
                        ;

//vars (Changed to left recursion)
vars : var {printf("vars -> var\n");}
```

```
                        | vars COMMA var{printf("vars -> vars COMMA var\n");}
                        | error{printf("****vars error\n");}
                        ;
//var
var : ident {printf("var -> ident\n");}
                        | ident L_SQUARE_BRACKET expression R_SQUARE_BRACKET {printf("var
 -> ident L_SQUARE_BRACKET expression R_SQUARE_BRACKET\n");}
                        | ident L_SQUARE_BRACKET expression R_SQUARE_BRACKET L_SQUARE_BRAC
KET expression R_SQUARE_BRACKET {printf("var -> ident L_SQUARE_BRACKET expression R_SQUARE
_BRACKET L_SQUARE_BRACKET expression R_SQUARE_BRACKET\n");}
                        |
                        ;
//number
number : NUMBER {printf("number -> NUMBER %d\n",$1);}
                        //| error{printf("vars error\n");}
                        ;

%%

int main(int argc, char **argv) {
   if (argc > 1) {
      yyin = fopen(argv[1], "r");
      if (yyin == NULL){
         printf("syntax: %s filename\n", argv[0]);
      }//end if
   }//end if
   yyparse(); // Calls yylex() for tokens.
   return 0;
}

void yyerror(const char *msg) {
  printf("**** Syntax error at line %d: %s\n", currLine, msg);
}
```

```
//Bao's version
%{
 #include <stdio.h>
 #include <stdlib.h>
 #include "y.tab.h"
 extern const char* yytext;
 int yylex();
 void yyerror(const char *msg);
 extern int currLine;
 extern int currPos;
 FILE * yyin;
%}

%union{
char* cval;
int ival;
}
```

```
//type
/*
%type <ival> NUMBER
%type <cval> IDENT
*/

%error-verbose
%start program
%token FUNCTION
%token <ival> NUMBER
%token <cval> IDENT
%token BEGIN_PARAMS END_PARAMS
%token BEGIN_LOCALS END_LOCALS
%token BEGIN_BODY END_BODY
%token INTEGER
%token ARRAY
%token OF
%token IF THEN ENDIF ELSE
%token WHILE DO
%token FOR
%token BEGINLOOP ENDLOOP
%token CONTINUE
%token READ WRITE
%token TRUE FALSE
%token RETURN
%token COMMA SEMICOLON COLON
%token MOD
//operators
%right ASSIGN
%left OR
%left AND
%right NOT
%left EQ NEQ LT GT LTE GTE
%left SUB ADD
%left MULT DIV
%right UMINUS
%left L_PAREN R_PAREN
%left L_SQUARE_BRACKET R_SQUARE_BRACKET

%%
//program (correct)
program : /*epsilon */ {printf("program -> Epsilon\n");}
      |program function {printf("program -> program function\n");}
      ;
//function
function : FUNCTION ident SEMICOLON BEGIN_PARAMS declarations END_PARAMS BEGIN_LOCALS decl
arations END_LOCALS BEGIN_BODY statements END_BODY{printf("function -> FUNCTION ident SEMI
COLON BEGIN_PARAMS declarations END_PARAMS BEGIN_LOCALS declarations END_LOCALS BEGIN_BODY
statements END_BODY\n");}
      ;
//ident
ident : IDENT {printf("ident -> IDENT %s\n", $1);}
      ;
//declaration (changed the ident comma loop) (added declaration loop)
```

```
declaration : declaration_loop COLON INTEGER  {printf("declaration -> declaration_loop COL
ON INTEGER\n");}
      | declaration_loop COLON ARRAY L_SQUARE_BRACKET NUMBER R_SQUARE_BRACKET OF INTEGER
 {printf("declaration -> declaration_loop COLON ARRAY L_PAREN NUMBER R_PAREN OF INTEGER
\n"); }
      |  declaration_loop COLON ARRAY L_SQUARE_BRACKET NUMBER R_SQUARE_BRACKET L_SQUARE_BR
ACKET NUMBER R_SQUARE_BRACKET OF INTEGER {printf(" declaration -> declaration_loop COLON A
RRAY L_PAREN NUMBER R_PAREN L_PAREN NUMBER R_PAREN OF INTEGER\n"); }
      | declaration_loop error INTEGER {printf("Syntax error at line %d: invalid declarati
on\n",currLine);}
      | declaration_loop error ARRAY L_SQUARE_BRACKET NUMBER R_SQUARE_BRACKET OF INTEGER
 {printf("Syntax error at line %d: expected token ':'\n",currLine);}
      | declaration_loop error ARRAY L_SQUARE_BRACKET NUMBER R_SQUARE_BRACKET L_SQUARE_BRA
CKET NUMBER R_SQUARE_BRACKET OF INTEGER {printf("Syntax error at line %d: expected token
 ':'\n",currLine);}
      ;
declaration_loop : ident {printf("delcaration_loop -> ident\n");}
| declaration_loop COMMA ident {printf("declaration_loop -> declaration_loop COMMA ident
\n");}
| declaration_loop error ident {printf("Syntax error at line %d: expected token ','\n",cur
rLine);}
;
//declarations
declarations : declarations declaration SEMICOLON{printf("declarations -> declarations dec
laration SEMICOLON \n");}
      | /* */ {printf("declarations -> epsilon\n");}
      | declarations declaration error {printf("Syntax error at line %d: expected token
 ';'\n",currLine);}
      ;
//statements
statements : statements statement SEMICOLON {printf("statements -> statements statement SE
MICOLON\n");}
      | {printf("statements -> epsilon\n");}
    //  | statements statement error {printf("Syntax error at line %d: expected token
 ';'\n",currLine);}
      ;
statement : var ASSIGN expression {printf("statement -> var ASSIGN expression\n");}
      | IF bool_exp THEN statements ENDIF {printf("statement -> IF bool_exp THEN statement
s ENDIF\n");}
      | IF bool_exp THEN statements ELSE statements ENDIF {printf("statement -> IF bool_ex
p THEN statements ELSE statements ENDIF\n");}
      | WHILE bool_exp BEGINLOOP statements ENDLOOP {printf("statement -> WHILE bool_exp B
EGINLOOP statements ENDLOOP\n");}
      | DO BEGINLOOP statements ENDLOOP WHILE bool_exp {printf("statement -> DO BEGINLOOP
 statements ENDLOOP WHILE bool_exp\n");}
      | FOR var ASSIGN NUMBER SEMICOLON bool_exp SEMICOLON var ASSIGN expression BEGINLOOP
statements ENDLOOP {printf("statement -> FOR var ASSIGN NUMBER SEMICOLON bool_exp SEMICOLO
N var ASSIGN expression BEGINLOOP statements ENDLOOP\n");}
      | READ vars {printf("statement -> READ vars\n");}
      | WRITE vars {printf("statement -> WRITE vars\n");}
      | CONTINUE {printf("statement -> CONTINUE\n");}
      | RETURN expression {printf("statement -> RETURN expression\n");}
      | var error expression {printf("Syntax error at line %d: expected token ':='\n",curr
Line);}
```

```
      ;

//bool_exp (good)
bool_exp : relation_and_exp {printf("bool_exp -> relation_and_exp\n");}
      | bool_exp OR relation_and_exp {printf("bool_exp -> bool_exp OR relation_and_exp
\n");}
      ;

//relation_and_exp (good)
relation_and_exp : relation_exp {printf("relation_and_exp -> relation_exp\n");}
      | relation_and_exp AND relation_exp {printf("relation_and_exp -> relation_and_exp AN
D relation_exp\n");}
      ;

//relation_exp (good)
relation_exp : expression comp expression {printf("relation_and_exp -> expression comp exp
ression\n");}
      | TRUE {printf("relation_and_exp -> TRUE\n");}
      | FALSE {printf("relation_and_exp -> FALSE\n");}
      | L_PAREN bool_exp R_PAREN {printf("relation_and_exp -> L_PAREN bool_exp R_PAREN
\n");}
      | NOT expression comp expression {printf("relation_and_exp -> NOT expression comp ex
pression\n");}
      | NOT TRUE {printf("relation_and_exp -> NOT TRUE\n");}
      | NOT FALSE {printf("relation_and_exp -> NOT FALSE\n");}
      | NOT L_PAREN bool_exp R_PAREN {printf("relation_and_exp -> NOT L_PAREN bool_exp R_P
AREN\n");}
      ;

//comp (good)
comp : EQ {printf("comp -> EQ\n");}
      | NEQ{printf("comp -> NEQ\n");}
      | LT {printf("comp -> LT\n");}
      | GT {printf("comp -> GT\n");}
      | LTE{printf("comp -> LTE\n");}
      | GTE {printf("comp -> GTE\n");}
      ;

//expressions (good) (removed epsilon and changed rule)
expressions : expression {printf("expressions -> expression\n");}
| expressions COMMA expression {printf("expressions -> expressions comma expression\n");}
| expressions error expression {printf("Syntax error at line %d: expected token ','\n",cur
rLine);}
      ;
//expression  (good)
expression : multiplicative_expression {printf("expression -> multiplicative_expression
\n");}
      | expression SUB multiplicative_expression {printf("expression -> multiplicative_exp
ression SUB multiplicative_expression\n");}
      | expression ADD multiplicative_expression {printf("expression -> multiplicative_exp
ression ADD multiplicative_expression\n");}
      ;

//multiplicative_expression (sounds good)
```

```
multiplicative_expression : term {printf("multiplicative_expression -> term\n");}
      | multiplicative_expression MULT term {printf("multiplicative_expression -> term MUL
T term\n");}
      | multiplicative_expression DIV term{printf("multiplicative_expression -> term DIV t
erm\n");}
      | multiplicative_expression MOD term {printf("multiplicative_expression -> term MOD
 term\n");}
      ;

//term (good)
term : var {printf("term -> var\n");}
      | number {printf("term -> number\n");}
      | L_PAREN expression R_PAREN {printf("term -> L_PAREN expression R_PAREN\n");}
      | ident L_PAREN expressions R_PAREN {printf("term -> ident L_PAREN expressions R_PAR
EN\n");}
      | SUB var %prec UMINUS{printf("term -> SUB var\n");}
      | SUB number %prec UMINUS {printf("term -> SUB number\n");}
      | SUB L_PAREN expression R_PAREN %prec UMINUS {printf("term -> SUB L_PAREN expressio
n R_PAREN\n");}
      ;

//vars (Changed to left recursion)
vars : var {printf("vars -> var\n");}
      | vars COMMA var{printf("vars -> vars COMMA var\n");}
      | vars error var {printf("Syntax error at line %d: expected token ','\n",currLine);}
      ;
//var
var : ident {printf("var -> ident\n");}
      | ident L_SQUARE_BRACKET expression R_SQUARE_BRACKET {printf("var -> ident L_SQUARE_
BRACKET expression R_SQUARE_BRACKET\n");}
      | ident L_SQUARE_BRACKET expression R_SQUARE_BRACKET L_SQUARE_BRACKET expression R_S
QUARE_BRACKET {printf("var -> ident L_SQUARE_BRACKET expression R_SQUARE_BRACKET L_SQUARE_
BRACKET expression R_SQUARE_BRACKET\n");}
    //  | ident error {printf("Syntax error at line %d: expected token ':='\n",currLine);}
      ;
//number
number : NUMBER {printf("number -> NUMBER %d\n",$1);}
      ;

%%

int main(int argc, char **argv) {
   if (argc > 1) {
      yyin = fopen(argv[1], "r");
      if (yyin == NULL){
         printf("syntax: %s filename\n", argv[0]);
      }//end if
   }//end if
   yyparse(); // Calls yylex() for tokens.
   return 0;
}

void yyerror(const char *msg) {
   printf("** Line %d, position %d: %s\n", currLine, currPos, msg);
```

```
}
```