Proj3 bison file

```
//Vivian current 12/06
%{
%}

%skeleton "lalr1.cc"
%require "3.0.4"
%defines
%define api.token.constructor
%define api.value.type variant
%define parse.error verbose
%locations


%code requires
{
  /* you may need these header files
   * add more header file if you need more
   */
#include <list>
#include <string>
#include <functional>
using namespace std;
  /* define the sturctures using as types for non-terminals */
struct dec_type{
  string code;
  list<string> id;
};
struct var_type{
string name;
string exp1;
string exp2;
bool scalar;
bool array1;
bool array2;
};

struct expression_type{
string name;
list<string> id;
bool array1;
bool array2;
string index;
};

struct term_type{
string name;
string exp1;
```

```
string exp2;
bool scalar;
bool array1;
bool array2;
list<string> id;
};

  /* end the structures for non-terminal types */
}


%code
{
#include "parser.tab.hh"
struct tests
{
string name;
yy::location loc;
};
  /* you may need these header files
   * add more header file if you need more
   */
#include <sstream>
#include <map>
#include <regex>
#include <set>
yy::parser::symbol_type yylex();
void yyerror(const char *msg);
int temp_count = 0;
string temp(){
return "__temp__" + to_string(temp_count++);
}
int label_count = 0;
string label(){
return "__label__" + to_string(label_count++);
}

  /* define your symbol table, global variables,
   * list of keywords or any function you may need here */

  /* end of your code */
}


%token END 0 "end of file";

  /* specify tokens, type of non-terminals and terminals here */

%token FUNCTION
%token <int> NUMBER
%token <string> 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
%start prog_start;

%type <string> program function ident statement number statements
%type <dec_type> declarations declaration
%type <list<string>> declaration_loop
%type <var_type> var
%type <expression_type> expression expressions multiplicative_expression relation_and_exp
 relation_exp bool_exp comp
%type <list<var_type>> vars
%type <term_type> term


/* end of token specifications */

%%

prog_start: program {cout << $1 << endl;}

program : {$$ = "";}
      |program function {$$ = $1 + "\n" + $2;}
      ;
//function
function : FUNCTION ident SEMICOLON BEGIN_PARAMS declarations END_PARAMS BEGIN_LOCALS decl
arations END_LOCALS BEGIN_BODY statements END_BODY
{
  $$ = "func " + $2 + "\n";
  $$ += $5.code;
  int i = 0;
  for (list<string>::iterator it = $5.id.begin(); it != $5.id.end(); ++it){
    $$ += *it + " $" + to_string(i) + "\n";
```

```
    ++i;
  }
  $$ += $8.code;
  $$ += $11;
  $$ += "endfunc";
}
      ;
//ident
ident : IDENT {$$ = $1;}
      ;
//declaration
declaration : declaration_loop COLON INTEGER
{

  for (list<string>::iterator it = $1.begin(); it != $1.end(); ++it){
    $$.code += ". " + *it + "\n";
    $$.id.push_back(*it);
  }
}
      | declaration_loop COLON ARRAY L_SQUARE_BRACKET NUMBER R_SQUARE_BRACKET OF INTEGER
{
  for (list<string>::iterator it = $1.begin(); it != $1.end(); ++it){
    $$.code += ".[] " + *it + ", " + to_string($5) + "\n";
    $$.id.push_back(*it);
  }
}     |  declaration_loop COLON ARRAY L_SQUARE_BRACKET NUMBER R_SQUARE_BRACKET L_SQUARE_BR
ACKET NUMBER R_SQUARE_BRACKET OF INTEGER
{
  for (list<string>::iterator it = $1.begin(); it != $1.end(); ++it){
    $$.code += ".[] " + *it + ", " + to_string($5*$8) + "\n";
    $$.id.push_back(*it);
  }
}
      ;
declaration_loop : ident {$$.push_back($1);}
| declaration_loop COMMA ident
{
  $$ = $1;
  $$.push_back($3);
}
;
//declarations
declarations : declarations declaration SEMICOLON
{
  $$.code = $1.code + "\n" + $2.code;
  int i = 0;
  $$.id = $1.id;
  for (list<string>::iterator it = $2.id.begin(); it != $2.id.end(); ++it){
  $$.id.push_back(*it);
  }
}
      |  {$$.id = list<string>(); $$.code = "";}
      ;
//statements
```

```
statements : statements statement SEMICOLON
{
$$ = $1;
$$ += $2;
}
      | {$$ = "";}
      ;
statement : var ASSIGN expression {
string temp_str;
if ($1.scalar)
{
for (list<string>::iterator it = $3.id.begin(); it != $3.id.end(); ++it){
$$ += *it + "\n";
}
$$ += "= " + $1.name + "," + $3.name + "\n";
}
//if ($1.array1){$$ += "[]= " + $1.name + ", " + $3.name;}
//if($1.array2){$$ += "[]= " + $1.name + ", " + $3.name;}
}
      | IF bool_exp THEN statements ENDIF
{
string ifTRUE = label();
string endIF = label();
string temp_str = temp();
//$$ += ifTRUE ;
//$$ += endIF ;

for (list<string>::iterator it = $2.id.begin(); it != $2.id.end(); ++it){
$$ += *it + "\n";
}
$$ += "?:= " + ifTRUE + "," + temp_str + "\n";
$$ += ":= " + endIF + "\n";
$$ += ": " + ifTRUE + "\n";
}
      | IF bool_exp THEN statements ELSE statements ENDIF
{
string ifTRUE = label();
string ifFALSE = label();
string endIF = label();
string temp_str = temp();
//$$ = $2.name;
$$ += ifTRUE ;
$$ += endIF ;
for (list<string>::iterator it = $2.id.begin(); it != $2.id.end(); ++it){
$$ += *it + "\n";
}
$$ += "?:= " + ifTRUE + "," + temp_str + "\n";
$$ += ":= " + ifFALSE + "\n";
$$ += ": " + ifTRUE + "\n";

//$$ += ":= " + endIF+ "\n";
//$$ += ": " + ifFALSE + "\n";

}
```

```
      | WHILE bool_exp BEGINLOOP statements ENDLOOP
{
string start = label();
string cond = label();
string end = label();


}
      | 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
{

}
      ;

//bool_exp
bool_exp : relation_and_exp
{
$$.name = $1.name;
$$.id = $1.id;
}
      | bool_exp OR relation_and_exp
{
string dest = temp();
string temp_str = temp();

//$$.id = $1.id;
for (list<string>::iterator it = $1.id.begin(); it != $1.id.end(); ++it){
$$.id.push_back(*it);
}
for (list<string>::iterator it = $3.id.begin(); it != $3.id.end(); ++it){
$$.id.push_back(*it);
}

//$$.name += temp_str;

$$.id.push_back(". " + dest + "\n");
$$.id.push_back("|| " + dest + ", " + $1.name + ", " + $3.name + "\n");
}
      ;

//relation_and_exp
relation_and_exp : relation_exp
{
$$.name = $1.name;
$$.id = $1.id;
}
```

```
      | relation_and_exp AND relation_exp
{
string dest = temp();
string temp_str = temp();
$$.id = $1.id;
for (list<string>::iterator it = $1.id.begin(); it != $1.id.end(); ++it){
$$.id.push_back(*it);
}
for (list<string>::iterator it = $3.id.begin(); it != $3.id.end(); ++it){
$$.id.push_back(*it);
}
$$.name += temp_str;
$$.id.push_back(". " + dest + "\n");
$$.id.push_back("&& " + dest + ", " + $1.name + ", " + $3.name + "\n");
}
      ;

//relation_exp
relation_exp : expression comp expression
{
string exp = $1.name ;
string exp2 = $3.name ;
//$$.id = $1.id;
for (list<string>::iterator it = $1.id.begin(); it != $1.id.end(); ++it){
$$.id.push_back(*it);
}
for (list<string>::iterator it = $3.id.begin(); it != $3.id.end(); ++it){
$$.id.push_back(*it);
}

if ($1.array1){
exp = temp();
$$.id.push_back(". " + exp );
$$.id.push_back( "=[] " + exp + ", " + $1.name + ", " + $1.index + "\n");
$$.name = exp ;
$$.array1 = $1.array1;
$$.array2 = $1.array2;
$$.index = $1.index;
}
if($3.array1) {
exp2 = temp();
$$.id.push_back(". " + exp2 );
$$.id.push_back( "[]= " + exp + ", " + $3.name + ", " + $3.index + "\n");
}

string dest = temp();
$$.id.push_back(" " + dest + "\n");
$$.id.push_back($2.name + dest + ", " + $1.name + "," + $3.name + "\n");
}
      | TRUE
{
string dest = temp();
$$.name = "1";
}
```

```
      | FALSE
{
$$.name = "0";
}
      | L_PAREN bool_exp R_PAREN
{
$$.name = $2.name;
$$.id = $2.id;
}
      | NOT expression comp expression
{
$$.id = $2.id;
for (list<string>::iterator it = $3.id.begin(); it != $3.id.end(); ++it){
$$.id.push_back(*it);
}
for (list<string>::iterator it = $4.id.begin(); it != $4.id.end(); ++it){
$$.id.push_back(*it);
}
string temp_str;
$$.id.push_back(". " + temp_str + "\n");
$$.id.push_back("! " + $3.name + temp_str + $2.name + "," + $4.name + "\n");
}

      | NOT TRUE
{
$$.name = "0";
}
      | NOT FALSE
{
$$.name = "1";
}
      | NOT L_PAREN bool_exp R_PAREN
{
$$.name = $3.name;
$$.id.push_back("! " + $3.name);
}
      ;

//comp
comp : EQ
{
string temp_str = temp();
$$.id.push_back("== ");
$$.name += temp_str;
}
      | NEQ
{
string temp_str = temp();
$$.id.push_back("!= ");
$$.name += temp_str;
}
      | LT
{
string temp_str = temp();
```

```
$$.id.push_back("< ");
$$.name += temp_str;
}
      | GT
{
string temp_str = temp();
$$.id.push_back("> ");
$$.name += temp_str;
}
      | LTE
{
string temp_str = temp();
$$.id.push_back("<= ");
$$.name += temp_str;
}
      | GTE
{
string temp_str = temp();
$$.id.push_back(">= ");
$$.name += temp_str;
}
      ;

//expressions
expressions : expression
{
$$.name = $1.name;
$$.id = $1.id;
$$.name += "parem ";
//$$.id = $1.id;
for (list<string>::iterator it = $1.id.begin(); it != $1.id.end(); ++it){
$$.id.push_back(*it);
}

$$.name += "\n ";

}
      | expressions COMMA expression
{
$$.name = $1.name;
$$.name += "parem ";
//$$.id = $1.id;
for (list<string>::iterator it = $1.id.begin(); it != $1.id.end(); ++it){
$$.id.push_back(*it);
}

$$.name += "\n ";
$$.name += $3.name;
for (list<string>::iterator it = $3.id.begin(); it != $3.id.end(); ++it){
$$.id.push_back(*it);
}
}
      ;
//expression
```

```
expression : multiplicative_expression
{
$$.name = $1.name;
$$.id = $1.id;
}
      | expression SUB multiplicative_expression
{

$$.id = $1.id;
for (list<string>::iterator it = $3.id.begin(); it != $3.id.end(); ++it){
$$.id.push_back(*it);
}

string temp_str = temp();
$$.id.push_back(". " + temp_str + "\n");
$$.name += temp_str;
$$.id.push_back("- " + $$.name + "," + $1.name + "," + $3.name + "\n");
}
      | expression ADD multiplicative_expression
{

$$.id = $1.id;
for (list<string>::iterator it = $3.id.begin(); it != $3.id.end(); ++it){
$$.id.push_back(*it);
}

string temp_str = temp();
$$.id.push_back(". " + temp_str + "\n");
$$.name += temp_str;
$$.id.push_back("+ " + $$.name + "," + $1.name + "," + $3.name + "\n");
}
      ;

//multiplicative_expression
multiplicative_expression : term {$$.name = $1.name; $$.id = $1.id;}
      | multiplicative_expression MULT term
{
$$.id = $1.id;
string temp_str = temp();
$$.id.push_back(". " + temp_str +"\n");
$$.name += temp_str;
$$.id.push_back("* " + temp_str + "," + $1.name + "," + $3.name + "\n");

}
      | multiplicative_expression DIV term
{
$$.id = $1.id;
string temp_str = temp();
$$.id.push_back(". " + temp_str +"\n");
$$.name += temp_str;
$$.id.push_back("/ " + temp_str + "," + $1.name + "," + $3.name + "\n");
}
      | multiplicative_expression MOD term
{
```

```
$$.id = $1.id;
string temp_str = temp();
$$.id.push_back(". " + temp_str +"\n");
$$.name += temp_str;
$$.id.push_back("% " + temp_str + "," + $1.name + "," + $3.name + "\n");
}
      ;

//term (good)
term : var
{
$$.name = $1.name;
$$.scalar = $1.scalar;
$$.array1 = $1.array1;
$$.array2 = $1.array2;
$$.exp1 = $1.exp1;
$$.exp2 = $1.exp2;
$$.id = list<string>();
}
      | number
{
$$.name = $1;
$$.scalar = false;
$$.array1 = false;
$$.array2 = false;
$$.exp1 = "";
$$.exp2 = "";
$$.id = list<string>();

}
      | L_PAREN expression R_PAREN
{
$$.name = $2.name;
$$.scalar = false;
$$.array1 = false;
$$.array2 = false;
$$.exp1 = "";
$$.exp2 = "";
$$.id = $2.id;
}
      | ident L_PAREN expressions R_PAREN
{
string temp_str = temp();
$$.id.push_back(". " + temp_str);
$$.id.push_back("= " + temp_str + ", " + $3.name);
$$.id.push_back("param " + temp_str);
string temp_str1 = temp();
$$.id.push_back(". " + temp_str1);
$$.id.push_back("call " + $1 + ", " + temp_str1);
$$.id.push_back("= " + $1 + ", " + temp_str1);
$$.name = $$.name + temp_str1;
}
      | SUB var %prec UMINUS
{
```

```
if ($2.scalar){
$$.name = "-" + $2.name;
$$.scalar = $2.scalar;
$$.array1 = $2.array1;
$$.array2 = $2.array2;
$$.exp1 = $2.exp1;
$$.exp2 = $2.exp2;
$$.id = list<string>();
}

else if ($2.array1){
string temp_str = temp();
$$.id.push_back(". " + temp_str);
$$.id.push_back("=[] " + temp_str + ", " + $2.name + ", " + $2.exp1);
$$.name = temp_str;
$$.array1 = $2.array1;
$$.array2 = $2.array2;
$$.exp1 = $2.exp1;
$$.exp2 = $2.exp2;
}

else if ($2.array2){
//need to implement this code
}
}
      | SUB number %prec UMINUS
{
$$.name = "-" + $2;
$$.scalar = true;
$$.array1 = false;
$$.array2 = false;
$$.exp1 = "";
$$.exp2 = "";
$$.id = list<string>();
}
      | SUB L_PAREN expression R_PAREN %prec UMINUS
{
$$.name = "-" + $3.name;
$$.id = $3.id;
}
      ;

//vars (Changed to left recursion)
vars : var
{
$$.push_back($1);
}
      | vars COMMA var
{
$$ = $1;
$$.push_back($3);
}
      ;
//var
```

```
var : ident {
$$.name = $1;
$$.scalar = true;
$$.array1 = $$.array2 = false;
$$.exp1 = "";
$$.exp2 = "";
}
      | ident L_SQUARE_BRACKET expression R_SQUARE_BRACKET
{
$$.name = $1;
$$.scalar = false;
$$.array1 = true;
$$.array2 = false;
$$.exp1 = $3.name;
$$.exp2 = "";
}
      | ident L_SQUARE_BRACKET expression R_SQUARE_BRACKET L_SQUARE_BRACKET expression R_S
QUARE_BRACKET
{
$$.name = $1;
$$.scalar = false;
$$.array1 = false;
$$.array2 = true;
$$.exp1 = $3.name;
$$.exp2 = $6.name;
}
      ;
//number
number : NUMBER {$$ = to_string($1);}
      ;

%%

int main(int argc, char *argv[])
{
  yy::parser p;
  return p.parse();
}

void yy::parser::error(const yy::location& l, const std::string& m)
{
  std::cerr << l << ": " << m << std::endl;
}
```

```
//sample code
%{
%}

%skeleton "lalr1.cc"
%require "3.0.4"
%defines
```

```
%define api.token.constructor
%define api.value.type variant
%define parse.error verbose
%locations


%code requires
{
  /* you may need these header files
   * add more header file if you need more
   */
#include <list>
#include <string>
#include <functional>
using namespace std;
  /* define the sturctures using as types for non-terminals */
struct dec_type{
  string code;
  list<string> id;
};

struct expr_type{
  string code;
  list<string> id;
};

struct var_type{
string name;
string exp1;
string exp2;
bool scalar;
bool array1;
bool array2;
};

struct stat_type{
  string code;
  list<string> id;
  struct var_type;
};

  /* end the structures for non-terminal types */
}


%code
{
#include "parser.tab.hh"
struct tests
{
string name;
yy::location loc;
};
  /* you may need these header files
```

```
   * add more header file if you need more
   */
#include <sstream>
#include <map>
#include <regex>
#include <set>
yy::parser::symbol_type yylex();
void yyerror(const char *msg);

  /* define your symbol table, global variables,
   * list of keywords or any function you may need here */

  /* end of your code */
}


%token END 0 "end of file";

  /* specify tokens, type of non-terminals and terminals here */

%token FUNCTION
%token <int> NUMBER
%token <string> 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
%start prog_start;

%type <string> program function ident number
%type <dec_type> declarations declaration
```

```
%type <stat_type> statements statement
%type <expr_type> expressions expression multiplicative_expression term
%type <list<string>> declaration_loop
%type <var_type> var
  /* end of token specifications */

%%

prog_start: program {cout << $1 << endl;}

program : {$$ = "";}
      |program function {$$ = $1 + "\n" + $2;}
      ;
//function
function : FUNCTION ident SEMICOLON BEGIN_PARAMS declarations END_PARAMS BEGIN_LOCALS decl
arations END_LOCALS BEGIN_BODY statements END_BODY
{
  $$ = "func " + $2 + "\n";
  $$ += $5.code;
  int i = 0;
  for (list<string>::iterator it = $5.id.begin(); it != $5.id.end(); ++it){
    $$ += *it + " $" + to_string(i) + "\n";
    ++i;
  }
  $$ += $8.code;
//  $$ += $11.code;
  $$ += "endfunc";
}
      ;
//ident
ident : IDENT {$$ = $1;}
      ;
//declaration
declaration : declaration_loop COLON INTEGER
{

  for (list<string>::iterator it = $1.begin(); it != $1.end(); ++it){
    $$.code += ". " + *it + "\n";
    $$.id.push_back(*it);
  }


}
      | declaration_loop COLON ARRAY L_SQUARE_BRACKET NUMBER R_SQUARE_BRACKET OF INTEGER
{
  for (list<string>::iterator it = $1.begin(); it != $1.end(); ++it){
    $$.code += ".[] " + *it + "\n";
    $$.id.push_back(*it);
  }

}
      |  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 {$$.push_back($1);}
| declaration_loop COMMA ident
{
  $$ = $1;
  $$.push_back($3);
}
;
//declarations
declarations : declarations declaration SEMICOLON
{
  $$.code = $1.code + "\n" + $2.code;
  int i = 0;
  $$.id = $1.id;
  for (list<string>::iterator it = $2.id.begin(); it != $2.id.end(); ++it){
  $$.id.push_back(*it);
  }
}
      |  {$$.id = list<string>(); $$.code = "";}
      ;
//statements
statements : statements statement SEMICOLON {
  $$.code = $1.code + "\n" + $2.code;
  int i = 0;
  $$.id = $1.id;
  for (list<string>::iterator it = $2.id.begin(); it != $2.id.end(); ++it){
  $$.id.push_back(*it);
  }
}
      | {$$.id = list<string>(); $$.code = "";}
      ;
statement : var ASSIGN expression {
if ($1.scalar)
{
$$.code = ". " + $1.name;
}
//if ($1.array1){$$.code = "[]= " + $1.name + ", " + $3.name;}
//if($1.array2){$$.code = "[]= " + $1.name + ", " + $3.name;}
 }
      | 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
expressions : expression {$$.code = $1.code;}
| expressions COMMA expression {$$.code = $1.code+ ", " + $3.code;}
      ;
//expression
expression : multiplicative_expression {
//$$.code = $1.code;
}
      | expression SUB multiplicative_expression {printf("expression -> multiplicative_exp
ression SUB multiplicative_expression\n");}
      | expression ADD multiplicative_expression {
//$$.code = "+ " + $$.name + ", " + $1.name + ", " + $3.name;
}
      ;

//multiplicative_expression (sounds good)
```

```
multiplicative_expression : term {
//$$.code = $1.code;
}
      | 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 {
$$.name = $1;
$$.scalar = true;
$$.array1 = $$.array2 = false;
$$.exp1 = "";
$$.exp2 = "";
}
      | 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
{
$$.name = $1;
$$.scalar = false;
$$.array1 = false;
$$.array2 = true;
$$.exp1 = "";
$$.exp2 = "";
}
      ;
//number
number : NUMBER {$$ = $1;}
      ;

%%
```

```
int main(int argc, char *argv[])
{
  yy::parser p;
  return p.parse();
}

void yy::parser::error(const yy::location& l, const std::string& m)
{
  std::cerr << l << ": " << m << std::endl;
}
```

```
%{
%}

%skeleton "lalr1.cc"
%require "3.0.4"
%defines
%define api.token.constructor
%define api.value.type variant
%define parse.error verbose
%locations


%code requires
{
  /* you may need these header files
   * add more header file if you need more
   */
#include <list>
#include <string>
#include <functional>
using namespace std;
  /* define the sturctures using as types for non-terminals */
struct dec_type{
  string code;
  list<string> id;
};
struct var_type{
string name;
string exp1;
string exp2;
bool scalar;
bool array1;
bool array2;
};

struct expression_type{
string name;
list<string> id;
};
```

```
struct term_type{
string name;
string exp1;
string exp2;
bool scalar;
bool array1;
bool array2;
list<string> id;
};

  /* end the structures for non-terminal types */
}


%code
{
#include "parser.tab.hh"
struct tests
{
string name;
yy::location loc;
};
  /* you may need these header files
   * add more header file if you need more
   */
#include <sstream>
#include <map>
#include <regex>
#include <set>
yy::parser::symbol_type yylex();
void yyerror(const char *msg);
int temp_count = 0;
string temp(){
return "temp" + to_string(temp_count++);
}
  /* define your symbol table, global variables,
   * list of keywords or any function you may need here */

  /* end of your code */
}


%token END 0 "end of file";

  /* specify tokens, type of non-terminals and terminals here */

%token FUNCTION
%token <int> NUMBER
%token <string> 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
%start prog_start;

%type <string> program function ident statement number statements
%type <dec_type> declarations declaration
%type <list<string>> declaration_loop
%type <var_type> var
%type <expression_type> expression expressions multiplicative_expression
%type <list<var_type>> vars
%type <term_type> term


/* end of token specifications */

%%

prog_start: program {cout << $1 << endl;}

program : {$$ = "";}
      |program function {$$ = $1 + "\n" + $2;}
      ;
//function
function : FUNCTION ident SEMICOLON BEGIN_PARAMS declarations END_PARAMS BEGIN_LOCALS decl
arations END_LOCALS BEGIN_BODY statements END_BODY
{
  $$ = "func " + $2 + "\n";
  $$ += $5.code;
  int i = 0;
  for (list<string>::iterator it = $5.id.begin(); it != $5.id.end(); ++it){
    $$ += *it + " $" + to_string(i) + "\n";
    ++i;
  }
  $$ += $8.code;
```

```
  $$ += $11;
  $$ += "endfunc";
}
      ;
//ident
ident : IDENT {$$ = $1;}
      ;
//declaration
declaration : declaration_loop COLON INTEGER
{

  for (list<string>::iterator it = $1.begin(); it != $1.end(); ++it){
    $$.code += ". " + *it + "\n";
    $$.id.push_back(*it);
  }


}
      | declaration_loop COLON ARRAY L_SQUARE_BRACKET NUMBER R_SQUARE_BRACKET OF INTEGER
{
  for (list<string>::iterator it = $1.begin(); it != $1.end(); ++it){
    $$.code += ".[] " + *it + "\n";
    $$.id.push_back(*it);
  }

}
      |  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 {$$.push_back($1);}
| declaration_loop COMMA ident
{
  $$ = $1;
  $$.push_back($3);
}
;
//declarations
declarations : declarations declaration SEMICOLON
{
  $$.code = $1.code + "\n" + $2.code;
  int i = 0;
  $$.id = $1.id;
  for (list<string>::iterator it = $2.id.begin(); it != $2.id.end(); ++it){
  $$.id.push_back(*it);
  }
}
      |  {$$.id = list<string>(); $$.code = "";}
      ;
//statements
statements : statements statement SEMICOLON
{
$$ = $1;
$$ += $2;
```

```
}
      | {$$ = "";}
      ;
statement : var ASSIGN expression {
string temp_str;
if ($1.scalar)
{
for (list<string>::iterator it = $3.id.begin(); it != $3.id.end(); ++it){
$$ += *it + "\n";
}
$$ += "= " + $1.name + "," + $3.name + "\n";
}

if ($1.array1){}
if($1.array2){}
 }
      | 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
{
$$.name = $1.name;
$$.id = $1.id;
}
| expressions COMMA expression {
$$.name = $1.name;
$$.id = $1.id;
$$.name += $3.name;
for (list<string>::iterator it = $3.id.begin(); it != $3.id.end(); ++it){
$$.id.push_back(*it);
}
}
      ;
//expression  (good)
expression : multiplicative_expression
{
$$.name = $1.name;
$$.id = $1.id;
}
      | expression SUB multiplicative_expression
{

$$.id = $1.id;
for (list<string>::iterator it = $3.id.begin(); it != $3.id.end(); ++it){
$$.id.push_back(*it);
}


string temp_str = temp();
$$.id.push_back(". " + temp_str + "\n");
$$.name += temp_str;

$$.id.push_back("- " + $$.name + "," + $1.name + "," + $3.name + "\n");
}
      | expression ADD multiplicative_expression {printf("expression -> multiplicative_exp
ression ADD multiplicative_expression\n");}
      ;
```

```
//multiplicative_expression
multiplicative_expression : term {$$.name = $1.name; $$.id = $1.id;}
      | multiplicative_expression MULT term
{
$$.id = $1.id;
string temp_str = temp();
$$.id.push_back(". " + temp_str +"\n");
$$.name += temp_str;
$$.id.push_back("* " + temp_str + "," + $1.name + "," + $3.name + "\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
{
$$.name = $1.name;
$$.scalar = $1.scalar;
$$.array1 = $1.array1;
$$.array2 = $1.array2;
$$.exp1 = $1.exp1;
$$.exp2 = $1.exp2;
$$.id = list<string>();
}
      | number
{
$$.name = $1;
$$.scalar = false;
$$.array1 = false;
$$.array2 = false;
$$.exp1 = "";
$$.exp2 = "";
$$.id = list<string>();

}
      | L_PAREN expression R_PAREN
{
$$.name = $2.name;
$$.scalar = false;
$$.array1 = false;
$$.array2 = false;
$$.exp1 = "";
$$.exp2 = "";
$$.id = $2.id;
}
      | 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
{
$$.push_back($1);
}
      | vars COMMA var
{
$$ = $1;
$$.push_back($3);
}
      ;
//var
var : ident {
$$.name = $1;
$$.scalar = true;
$$.array1 = $$.array2 = false;
$$.exp1 = "";
$$.exp2 = "";
}
      | ident L_SQUARE_BRACKET expression R_SQUARE_BRACKET
{
$$.name = $1;
$$.scalar = false;
$$.array1 = true;
$$.array2 = false;
$$.exp1 = $3.name;
$$.exp2 = "";
}
      | ident L_SQUARE_BRACKET expression R_SQUARE_BRACKET L_SQUARE_BRACKET expression R_S
QUARE_BRACKET
{
$$.name = $1;
$$.scalar = false;
$$.array1 = false;
$$.array2 = true;
$$.exp1 = $3.name;
$$.exp2 = $6.name;
}
      ;
//number
number : NUMBER {$$ = to_string($1);}
      ;

%%

int main(int argc, char *argv[])
{
  yy::parser p;
  return p.parse();
}
```

```
void yy::parser::error(const yy::location& l, const std::string& m)
{
  std::cerr << l << ": " << m << std::endl;
}
```

```
//sample code
%{
%}

%skeleton "lalr1.cc"
%require "3.0.4"
%defines
%define api.token.constructor
%define api.value.type variant
%define parse.error verbose
%locations


%code requires
{
  /* you may need these header files
   * add more header file if you need more
   */
#include <list>
#include <string>
#include <functional>
using namespace std;
  /* define the sturctures using as types for non-terminals */
struct dec_type{
  string code;
  list<string> id;
};
struct var_type{
string name;
string exp1;
string exp2;
bool scalar;
bool array1;
bool array2;
};
  /* end the structures for non-terminal types */
}


%code
{
#include "parser.tab.hh"
struct tests
{
string name;
```

```
yy::location loc;
};
  /* you may need these header files
   * add more header file if you need more
   */
#include <sstream>
#include <map>
#include <regex>
#include <set>
yy::parser::symbol_type yylex();
void yyerror(const char *msg);

  /* define your symbol table, global variables,
   * list of keywords or any function you may need here */

  /* end of your code */
}


%token END 0 "end of file";

  /* specify tokens, type of non-terminals and terminals here */

%token FUNCTION
%token <int> NUMBER
%token <string> 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
%start prog_start;
```

```
%type <string> program function ident statement number
%type <dec_type> declarations declaration
%type <list<string>> declaration_loop
%type <var_type> var
  /* end of token specifications */

%%

prog_start: program {cout << $1 << endl;}

program : {$$ = "";}
      |program function {$$ = $1 + "\n" + $2;}
      ;
//function
function : FUNCTION ident SEMICOLON BEGIN_PARAMS declarations END_PARAMS BEGIN_LOCALS decl
arations END_LOCALS BEGIN_BODY statements END_BODY
{
  $$ = "func " + $2 + "\n";
  $$ += $5.code;
  int i = 0;
  for (list<string>::iterator it = $5.id.begin(); it != $5.id.end(); ++it){
    $$ += *it + " $" + to_string(i) + "\n";
    ++i;
  }
  $$ += $8.code;
//  $$ += $11.code;
  $$ += "endfunc";
}
      ;
//ident
ident : IDENT {$$ = $1;}
      ;
//declaration
declaration : declaration_loop COLON INTEGER
{

  for (list<string>::iterator it = $1.begin(); it != $1.end(); ++it){
    $$.code += ". " + *it + "\n";
    $$.id.push_back(*it);
  }


}
      | declaration_loop COLON ARRAY L_SQUARE_BRACKET NUMBER R_SQUARE_BRACKET OF INTEGER
{
  for (list<string>::iterator it = $1.begin(); it != $1.end(); ++it){
    $$.code += ".[] " + *it + "\n";
    $$.id.push_back(*it);
  }

}
      |  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 {$$.push_back($1);}
| declaration_loop COMMA ident
{
  $$ = $1;
  $$.push_back($3);
}
;
//declarations
declarations : declarations declaration SEMICOLON
{
  $$.code = $1.code + "\n" + $2.code;
  int i = 0;
  $$.id = $1.id;
  for (list<string>::iterator it = $2.id.begin(); it != $2.id.end(); ++it){
  $$.id.push_back(*it);
  }
}
      |  {$$.id = list<string>(); $$.code = "";}
      ;
//statements
statements : statements statement SEMICOLON {printf("statements -> statements statement SE
MICOLON\n");}
      | {printf("statements -> epsilon\n");}
      ;
statement : var ASSIGN expression {
if ($1.scalar)
{
//$$ = "= " + $1.name
}
if ($1.array1){}
if($1.array2){}
 }
      | 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 {
$$.name = $1;
$$.scalar = true;
$$.array1 = $$.array2 = false;
$$.exp1 = "";
$$.exp2 = "";
}
      | 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
{
$$.name = $1;
$$.scalar = false;
$$.array1 = false;
$$.array2 = true;
$$.exp1 = "";
$$.exp2 = "";
}
      ;
//number
number : NUMBER {$$ = $1;}
      ;

%%

int main(int argc, char *argv[])
{
  yy::parser p;
  return p.parse();
}

void yy::parser::error(const yy::location& l, const std::string& m)
{
```

```
  std::cerr << l << ": " << m << std::endl;
}
```