Programming language and Compiler

Programming Assignment 1:

department: CSIE

grade: sphomore

group: 胡師睿、劉家均

student ID: 411221301、411221426

professor: Chung Yung

Content

I. Problem Description

The objective of this assignment is to implement the front end of a compiler for the T programming language using **Flex** (Lex) and **Bison** (Yacc). This includes building a **lexical analyzer** and a **syntax recognize**r, capable of parsing T programs and showing the grammar rules applied during parsing.

The T language specification includes:

- **Lexical elements:**
  keywords (e.g., **WRITE**, **READ**, **IF**), operators, separators, identifiers, integers, real numbers, comments, and quoted strings.
- **Grammar rules:**
  specified in EBNF for expressions, statements, blocks, method declarations, etc.

The compiler must recognize valid T programs and provide traceable output of applied grammar rules.

II. Highlights of Implementation

The implementation is modularized into distinct files:

- **t_lex.l**: defines the lexical rules using Flex for tokenizing input.
- **t_parse.y**: specifies the grammar and parsing rules using Bison.
- **t2c.c**: contains the main function used to launch the parser.
- **t2c.h**: header file shared between components.

Lexical Analyzer (t_lex.l)

- Implements all token definitions including keywords (**WRITE**, **READ**, etc.), identifiers, integers, real numbers, and string literals.
- Special handling for multi-line comments and escaped strings.
- Tokens such as **:=**, **>=**, **<=**, **==**, etc., are correctly recognized using longest-match rules.
- Uses global variables (**ival**, **rval**, **name**, **qstr**) to carry semantic values to the parser.

Syntax Recognizer (t_parse.y)

- Follows the T language grammar closely, implementing rules for **Program, MethodDecl, Block, Statement, Expression,** and **BoolExpr.**
- Each rule is instrumented with printf() to display the production used, which helps in visualizing the parsing process.
- Includes operator precedence and associativity for correct parsing of expressions.
- Handles nested blocks, multiple parameters, and function calls as expressions.

Main Driver (t2c.c)

- Initializes parsing with **yyparse()** and optionally prints the result of the compilation process.
- Designed for future extension into intermediate code generation or semantic checking.

Build System (Makefile)

- Automates the steps of converting **.l** and **.y** files to **.c**, then compiling and linking them into an executable named **parse**.
- Allows developers to rebuild the project with a single **make** command.

Additional Tools

- **t_lexMain.c** provided for debugging or testing just the lexical scanner.
- Modular structure and strict file separation make the code maintainable and scalable.

III. Program Listing

t_lex.l

Defines tokens including:

- Identifiers
- Keywords
- Operators
- Numbers (integers and reals)
- Strings
- Comments

Lexical Rules

```
{ID}   { strcpy(name, yytext); return lID; }
{DIG}  { ival = atoi(yytext); return lINUM; }
{RNUM} { rval = atof(yytext); return lRNUM; }
```

Explanation of Rules:

- **Identifiers ({ID})**
    - Matches variable names.
    - **yytext** contains the matched identifier string.
    - Copies the text into the global variable **name**.
    - Returns the token **lID**.
- **Integer Numbers ({DIG})**
    - Matches digit sequences (e.g., 123, 4567).
    - Converts the text using **atoi(yytext)** and stores it in **ival.**
    - Returns the token **lINUM**.
- **Real Numbers ({RNUM})**
    - Matches real numbers with a decimal (e.g., 3.14, 0.001).
    - Converts the text using **atof(yytext)** and stores it in **rval.**
    - Returns the token **lRNUM.**

This setup allows the parser to use the exact value of identifiers and literals during grammar reductions or future semantic analysis.

t_parse.y

Implements grammar rules for the T language, including:

- Program
- Expression
- Statement
- BoolExpression

Also includes:

- Precedence rules
- Associativity rules

Parse Rules

```
stmt
  : block
      { printf("Statement -> Block\n"); }
  | localvardecl
      { printf("Statement -> LocalVarDecl\n"); }
  | assignstmt
      { printf("Statement -> AssignStmt\n"); }
  | returnstmt
      { printf("Statement -> ReturnStmt\n"); }
  | ifstmt
      { printf("Statement -> IfStmt\n"); }
  | writestmt
      { printf("Statement -> WriteStmt\n"); }
  | readstmt
      { printf("Statement -> ReadStmt\n"); }
  ;

block
  : lBEGIN stmts lEND
      { printf("Block -> BEGIN Statement+ END\n"); }
  ;

stmts
  : stmt stmts
      { printf("Statements -> Statement Statements\n"); }
  | stmt
      { printf("Statements -> Statement\n"); }
  ;

localvardecl
  : type lID lSEMI
      { printf("LocalVarDecl -> Type ID SEMI\n"); }
```

```
  | type assignstmt
      { printf("LocalVarDecl -> Type AssignStmt\n"); }
  ;

assignstmt
  : lID lASSIGN expr lSEMI
      { printf("AssignStmt -> ID := Expression SEMI\n"); }
  ;

returnstmt
  : lRETURN expr lSEMI
      { printf("ReturnStmt -> RETURN Expression SEMI\n"); }
  ;

ifstmt
  : lIF lLP boolexpr lRP stmt
      { printf("IfStmt -> IF ( BoolExpression ) Statement\n"); }
  | lIF lLP boolexpr lRP stmt lELSE stmt
      { printf("IfStmt -> IF ( BoolExpression ) Statement ELSE Statement\n"); }
  ;

writestmt
  : lWRITE lLP expr lCOMMA lQSTR lRP lSEMI
      { printf("WriteStmt -> WRITE ( Expression , QString ) SEMI\n"); }
  ;

readstmt
  : lREAD lLP lID lCOMMA lQSTR lRP lSEMI
      { printf("ReadStmt -> READ ( ID , QString ) SEMI\n"); }
  ;

expr : mexpr exprtail { printf("Expr -> MExpr ExprTail\n"); }
     ;

exprtail : lADD mexpr exprtail {
             printf("ExprTail -> + MExpr ExprTail\n");
           }
         | lMINUS mexpr exprtail {
             printf("ExprTail -> - MExpr ExprTail\n");
           }
         | { printf("ExprTail -> ε\n"); }
         ;

mexpr : factor mexprtail { printf("MExpr -> Factor MExprTail\n"); }
      ;

mexprtail : lTIMES factor mexprtail {
              printf("MExprTail -> * Factor MExprTail\n");
            }
          | lDIVIDE factor mexprtail {
              printf("MExprTail -> / Factor MExprTail\n");
```

```
            }
          | { printf("MExprTail -> ε\n"); }
          ;

factor
  : lINUM
      { printf("PrimaryExpr -> INUM\n"); }
  | lRNUM
      { printf("PrimaryExpr -> RNUM\n"); }
  | lID
      { printf("PrimaryExpr -> ID\n"); }
  | lLP expr lRP
      { printf("PrimaryExpr -> ( Expression )\n"); }
  | lID lLP actualparams lRP
      { printf("PrimaryExpr -> ID ( ActualParams )\n"); }
  ;

boolexpr
  : expr lEQU expr
      { printf("BoolExpr -> Expression == Expression\n"); }
  | expr lNEQ expr
      { printf("BoolExpr -> Expression != Expression\n"); }
  | expr lGT expr
      { printf("BoolExpr -> Expression > Expression\n"); }
  | expr lGE expr
      { printf("BoolExpr -> Expression >= Expression\n"); }
  | expr lLT expr
      { printf("BoolExpr -> Expression < Expression\n"); }
  | expr lLE expr
      { printf("BoolExpr -> Expression <= Expression\n"); }
  ;

actualparams
  : expr lCOMMA actualparams
      { printf("ActualParams -> Expression , ActualParams\n"); }
  | expr
      { printf("ActualParams -> Expression\n"); }
  |
      { printf("ActualParams -> \n"); }
  ;
```

Explanation of Major Grammar Rules:

- **stmt**:
  A single statement can be a block, declaration, assignment, return, if-statement, write, or read. Each sub-rule prints its structure for debugging.
- **block**:
  Represents a sequence of statements enclosed by BEGIN ... END.
- **stmts**:
  Allows recursive definition of multiple statements.
- **localvardecl**:
  Covers both simple declarations (e.g., INT x;) and initialized declarations (e.g., INT x := 3;).
- **assignstmt**:
  Assigns an expression to an identifier (e.g., x := 5;).
- **returnstmt**:
  A return statement like RETURN x + y;.
- **ifstmt**:
  Handles both IF (...) stmt and IF (...) stmt ELSE stmt forms.
- **writestmt & readstmt**:
  Model I/O syntax using expressions and quoted strings.
- **expr, exprtail, mexpr, mexprtail**:
  Represent standard expression parsing using recursive descent for left-to-right associativity.
- **factor**:
  Covers literals, variables, grouped expressions, and function calls.
- **boolexpr**:
  Logical comparisons like x == y, a >= b, etc.
- **actualparams**:
  Parses the parameters passed into function calls, including optional empty argument lists.

These rules allow the parser to handle real T programs such as:

- **t2c.c**: Main function invoking yyparse().
- **t2c.h**: Provides shared declarations.
- **Makefile**: Automates the compilation process.
- **t_lexMain.c**: An alternative or debugging entry point.
- **README.md**: Provides basic instructions.

IV. Test Run Results

Multiple .t test files were prepared and executed:

- **test.t**: Full sample program with function calls and I/O.
- **test1.t to test7.t**: Various edge cases including:
    - Empty parameters
    - Different arithmetic combinations
    - Nested blocks
    - If-else branches
    - Return expressions

test.t

In this **test.t** file, we intentionally apply left-factoring to the grammar rules for statement, specifically by decomposing **Expression** into **expr** and **exprtail**, and **MultiplicativeExpr** into **mexpr** and **mexprtail.** This transformation ensures correct parsing of various expressions. As a result, the file executes successfully and processes exactly 110 lines of code.

```
Type -> INT
Type -> INT
Formal -> Type ID
Type -> INT
Formal -> Type ID
OtherFormals ->
OtherFormals -> COMMA Formal OtherFormals
Formals -> Formal OtherFormals
Type -> INT
LocalVarDecl -> Type Id ;
Statement -> LocalVarDecl
PrimaryExpr -> ID
PrimaryExpr -> ID
MultiTail ->
MultiTail -> times PrimaryExpression MultTail
MultiplicativeExpr -> PrimaryExpression MultiTail
PrimaryExpr -> ID
PrimaryExpr -> ID
MultiTail ->
MultiTail -> times PrimaryExpression MultTail
MultiplicativeExpr -> PrimaryExpression MultiTail
ExpressionTail ->
ExpressionTail -> minus MultiplicativeExpr ExpressionTail
Expression -> MultiplicativeExpr ExpressionTail
AssignStmt -> Id := Expression ;
Statement -> AssignStmt
PrimaryExpr -> ID
```

```
MultiTail ->
MultiplicativeExpr -> PrimaryExpression MultiTail
ExpressionTail ->
Expression -> MultiplicativeExpr ExpressionTail
ReturnStmt -> RETURN Expression ;
Statement -> ReturnStmt
Statements -> Statement
Statements -> Statement Statements
Statements -> Statement Statements
Block -> BEGIN Statements END
MethodDecl -> Type ID LP Formals RP Block
Type -> INT
Formals ->
Type -> INT
LocalVarDecl -> Type Id ;
Statement -> LocalVarDecl
ReadStmt -> READ ( Id , QString ) ;
Statement -> Readstmt
Type -> INT
LocalVarDecl -> Type Id ;
Statement -> LocalVarDecl
ReadStmt -> READ ( Id , QString ) ;
Statement -> Readstmt
Type -> INT
LocalVarDecl -> Type Id ;
Statement -> LocalVarDecl
PrimaryExpr -> ID
MultiTail ->
```

```
MultiplicativeExpr -> PrimaryExpression MultiTail
ExpressionTail ->
Expression -> MultiplicativeExpr ExpressionTail
PrimaryExpr -> ID
MultiTail ->
MultiplicativeExpr -> PrimaryExpression MultiTail
ExpressionTail ->
Expression -> MultiplicativeExpr ExpressionTail
OtherParams ->
OtherParams -> COMMA Expression OtherParams
ActualParams -> Expression OtherParams
PrimaryExpr -> ID ( ActualParams )
MultiTail ->
MultiplicativeExpr -> PrimaryExpression MultiTail
PrimaryExpr -> ID
MultiTail ->
MultiplicativeExpr -> PrimaryExpression MultiTail
ExpressionTail ->
Expression -> MultiplicativeExpr ExpressionTail
PrimaryExpr -> ID
MultiTail ->
MultiplicativeExpr -> PrimaryExpression MultiTail
ExpressionTail ->
Expression -> MultiplicativeExpr ExpressionTail
OtherParams ->
OtherParams -> COMMA Expression OtherParams
ActualParams -> Expression OtherParams
PrimaryExpr -> ID ( ActualParams )
MultiTail ->
MultiplicativeExpr -> PrimaryExpression MultiTail
```

```
ExpressionTail ->
ExpressionTail -> plus MultiplicativeExpr ExpressionTail
Expression -> MultiplicativeExpr ExpressionTail
AssignStmt -> Id := Expression ;
Statement -> AssignStmt
PrimaryExpr -> ID
MultiTail ->
MultiplicativeExpr -> PrimaryExpression MultiTail
ExpressionTail ->
Expression -> MultiplicativeExpr ExpressionTail
WriteStmt -> WRITE ( Expression , QString ) ;
Statement -> WriteStmt
Statements -> Statement
Statements -> Statement Statements
Statements -> Statement Statements
Statements -> Statement Statements
Statements -> Statement Statements
Statements -> Statement Statements
Statements -> Statement Statements
Block -> BEGIN Statements END
MethodDecl -> Type MAIN ID LP Formals RP Block
MethodDecls -> MethodDecl
MethodDecls -> MethodDecl MethodDecls
Program -> MethodDecls
Parsed OK!
```

test1.t

```
Type -> INT
Formals ->
Type -> REAL
LocalVarDecl -> Type Id ;
Statement -> LocalVarDecl
ReadStmt -> READ ( Id , QString ) ;
Statement -> Readstmt
PrimaryExpr -> ID
MultiTail ->
MultiplicativeExpr -> PrimaryExpression MultiTail
ExpressionTail ->
Expression -> MultiplicativeExpr ExpressionTail
WriteStmt -> WRITE ( Expression , QString ) ;
Statement -> WriteStmt
Statements -> Statement
Statements -> Statement Statements
Statements -> Statement Statements
Block -> BEGIN Statements END
MethodDecl -> Type MAIN ID LP Formals RP Block
MethodDecls -> MethodDecl
Program -> MethodDecls
Parsed OK!
```

test2.t

```
Type -> INT
Formals ->
Type -> REAL
LocalVarDecl -> Type Id ;
Statement -> LocalVarDecl
ReadStmt -> READ ( Id , QString ) ;
Statement -> Readstmt
Type -> REAL
PrimaryExpr -> RNUM
MultiTail ->
MultiplicativeExpr -> PrimaryExpression MultiTail
ExpressionTail ->
Expression -> MultiplicativeExpr ExpressionTail
AssignStmt -> Id := Expression ;
LocalVarDecl -> Type AssignStmt
Statement -> LocalVarDecl
PrimaryExpr -> ID
MultiTail ->
MultiplicativeExpr -> PrimaryExpression MultiTail
ExpressionTail ->
Expression -> MultiplicativeExpr ExpressionTail
WriteStmt -> WRITE ( Expression , QString ) ;
Statement -> WriteStmt
Statements -> Statement
Statements -> Statement Statements
Statements -> Statement Statements
Statements -> Statement Statements
Block -> BEGIN Statements END
MethodDecl -> Type MAIN ID LP Formals RP Block
MethodDecls -> MethodDecl
Program -> MethodDecls
Parsed OK!
```

test3.t

```
Type -> INT
Formals ->
Type -> REAL
LocalVarDecl -> Type Id ;
Statement -> LocalVarDecl
ReadStmt -> READ ( Id , QString ) ;
Statement -> Readstmt
Type -> REAL
PrimaryExpr -> INUM
PrimaryExpr -> RNUM
PrimaryExpr -> ID
MultiTail ->
MultiTail -> times PrimaryExpression MultTail
MultiTail -> times PrimaryExpression MultTail
MultiplicativeExpr -> PrimaryExpression MultiTail
ExpressionTail ->
Expression -> MultiplicativeExpr ExpressionTail
AssignStmt -> Id := Expression ;
LocalVarDecl -> Type AssignStmt
Statement -> LocalVarDecl
PrimaryExpr -> ID
MultiTail ->
MultiplicativeExpr -> PrimaryExpression MultiTail
ExpressionTail ->
Expression -> MultiplicativeExpr ExpressionTail
WriteStmt -> WRITE ( Expression , QString ) ;
```

```
Statement -> WriteStmt
PrimaryExpr -> RNUM
PrimaryExpr -> ID
PrimaryExpr -> ID
MultiTail ->
MultiTail -> times PrimaryExpression MultTail
MultiTail -> times PrimaryExpression MultTail
MultiplicativeExpr -> PrimaryExpression MultiTail
ExpressionTail ->
Expression -> MultiplicativeExpr ExpressionTail
WriteStmt -> WRITE ( Expression , QString ) ;
Statement -> WriteStmt
Statements -> Statement
Statements -> Statement Statements
Statements -> Statement Statements
Statements -> Statement Statements
Statements -> Statement Statements
Block -> BEGIN Statements END
MethodDecl -> Type MAIN ID LP Formals RP Block
MethodDecls -> MethodDecl
Program -> MethodDecls
Parsed OK!

```

test4.t

```
Type -> INT
Formals ->
Type -> REAL
LocalVarDecl -> Type Id ;
Statement -> LocalVarDecl
ReadStmt -> READ ( Id , QString ) ;
Statement -> Readstmt
Type -> REAL
PrimaryExpr -> INUM
PrimaryExpr -> RNUM
PrimaryExpr -> ID
MultiTail ->
MultiTail -> times PrimaryExpression MultTail
MultiTail -> times PrimaryExpression MultTail
MultiplicativeExpr -> PrimaryExpression MultiTail
ExpressionTail ->
Expression -> MultiplicativeExpr ExpressionTail
AssignStmt -> Id := Expression ;
LocalVarDecl -> Type AssignStmt
Statement -> LocalVarDecl
PrimaryExpr -> ID
MultiTail ->
MultiplicativeExpr -> PrimaryExpression MultiTail
ExpressionTail ->
Expression -> MultiplicativeExpr ExpressionTail
```

```
WriteStmt -> WRITE ( Expression , QString ) ;
Statement -> WriteStmt
PrimaryExpr -> RNUM
PrimaryExpr -> ID
PrimaryExpr -> ID
MultiTail ->
MultiTail -> times PrimaryExpression MultTail
MultiTail -> times PrimaryExpression MultTail
MultiplicativeExpr -> PrimaryExpression MultiTail
ExpressionTail ->
Expression -> MultiplicativeExpr ExpressionTail
WriteStmt -> WRITE ( Expression , QString ) ;
Statement -> WriteStmt
Statements -> Statement
Statements -> Statement Statements
Statements -> Statement Statements
Statements -> Statement Statements
Statements -> Statement Statements
Block -> BEGIN Statements END
MethodDecl -> Type MAIN ID LP Formals RP Block
MethodDecls -> MethodDecl
Program -> MethodDecls
Parsed OK!
```

test5.t

```
Type -> INT
Type -> INT
Formal -> Type ID
Type -> INT
Formal -> Type ID
OtherFormals ->
OtherFormals -> COMMA Formal OtherFormals
Formals -> Formal OtherFormals
Type -> INT
LocalVarDecl -> Type Id ;
Statement -> LocalVarDecl
PrimaryExpr -> ID
PrimaryExpr -> ID
MultiTail ->
MultiTail -> times PrimaryExpression MultTail
MultiplicativeExpr -> PrimaryExpression MultiTail
PrimaryExpr -> ID
PrimaryExpr -> ID
MultiTail ->
MultiTail -> times PrimaryExpression MultTail
MultiplicativeExpr -> PrimaryExpression MultiTail
ExpressionTail ->
ExpressionTail -> minus MultiplicativeExpr ExpressionTail
Expression -> MultiplicativeExpr ExpressionTail
AssignStmt -> Id := Expression ;
```

```
Statement -> AssignStmt
PrimaryExpr -> ID
MultiTail ->
MultiplicativeExpr -> PrimaryExpression MultiTail
ExpressionTail ->
Expression -> MultiplicativeExpr ExpressionTail
ReturnStmt -> RETURN Expression ;
Statement -> ReturnStmt
Statements -> Statement
Statements -> Statement Statements
Statements -> Statement Statements
Block -> BEGIN Statements END
MethodDecl -> Type ID LP Formals RP Block
Type -> INT
Formals ->
Type -> INT
LocalVarDecl -> Type Id ;
Statement -> LocalVarDecl
ReadStmt -> READ ( Id , QString ) ;
Statement -> Readstmt
Type -> INT
LocalVarDecl -> Type Id ;
Statement -> LocalVarDecl
ReadStmt -> READ ( Id , QString ) ;
Statement -> Readstmt
Type -> INT
LocalVarDecl -> Type Id ;
```

```
Statement -> LocalVarDecl
PrimaryExpr -> ID
MultiTail ->
MultiplicativeExpr -> PrimaryExpression MultiTail
ExpressionTail ->
Expression -> MultiplicativeExpr ExpressionTail
PrimaryExpr -> ID
MultiTail ->
MultiplicativeExpr -> PrimaryExpression MultiTail
ExpressionTail ->
Expression -> MultiplicativeExpr ExpressionTail
OtherParams ->
OtherParams -> COMMA Expression OtherParams
ActualParams -> Expression OtherParams
PrimaryExpr -> ID ( ActualParams )
MultiTail ->
MultiplicativeExpr -> PrimaryExpression MultiTail
PrimaryExpr -> ID
MultiTail ->
MultiplicativeExpr -> PrimaryExpression MultiTail
ExpressionTail ->
Expression -> MultiplicativeExpr ExpressionTail
PrimaryExpr -> ID
MultiTail ->
MultiplicativeExpr -> PrimaryExpression MultiTail
ExpressionTail ->
Expression -> MultiplicativeExpr ExpressionTail
OtherParams ->
```

```
OtherParams -> COMMA Expression OtherParams
ActualParams -> Expression OtherParams
PrimaryExpr -> ID ( ActualParams )
MultiTail ->
MultiplicativeExpr -> PrimaryExpression MultiTail
ExpressionTail ->
ExpressionTail -> plus MultiplicativeExpr ExpressionTail
Expression -> MultiplicativeExpr ExpressionTail
AssignStmt -> Id := Expression ;
Statement -> AssignStmt
PrimaryExpr -> ID
MultiTail ->
MultiplicativeExpr -> PrimaryExpression MultiTail
ExpressionTail ->
Expression -> MultiplicativeExpr ExpressionTail
WriteStmt -> WRITE ( Expression , QString ) ;
Statement -> WriteStmt
Statements -> Statement
Statements -> Statement Statements
Statements -> Statement Statements
Statements -> Statement Statements
Statements -> Statement Statements
Statements -> Statement Statements
Statements -> Statement Statements
Block -> BEGIN Statements END
MethodDecl -> Type MAIN ID LP Formals RP Block
MethodDecls -> MethodDecl
MethodDecls -> MethodDecl MethodDecls
Program -> MethodDecls
Parsed OK!
```

test7.t

The syntax error encountered during the execution of **test7.t** is due to a missing right parenthesis in one of the lines. As a result, the parser detects the invalid syntax, terminates the parsing process, and reports a “syntax error” at the end of the output line.

```
Type -> INT
Type -> INT
Formal -> Type ID
Type -> INT
Formal -> Type ID
OtherFormals ->
OtherFormals -> COMMA Formal OtherFormals
Formals -> Formal OtherFormals
Type -> INT
LocalVarDecl -> Type Id ;
Statement -> LocalVarDecl
PrimaryExpr -> ID
PrimaryExpr -> ID
MultiTail ->
MultiTail -> times PrimaryExpression MultTail
MultiplicativeExpr -> PrimaryExpression MultiTail
PrimaryExpr -> ID
PrimaryExpr -> ID
MultiTail ->
MultiTail -> times PrimaryExpression MultTail
MultiplicativeExpr -> PrimaryExpression MultiTail
ExpressionTail ->
ExpressionTail -> minus MultiplicativeExpr ExpressionTail
Expression -> MultiplicativeExpr ExpressionTail
AssignStmt -> Id := Expression ;
Statement -> AssignStmt
```

```
PrimaryExpr -> ID
MultiTail ->
MultiplicativeExpr -> PrimaryExpression MultiTail
ExpressionTail ->
Expression -> MultiplicativeExpr ExpressionTail
ReturnStmt -> RETURN Expression ;
Statement -> ReturnStmt
Statements -> Statement
Statements -> Statement Statements
Statements -> Statement Statements
Block -> BEGIN Statements END
MethodDecl -> Type ID LP Formals RP Block
Type -> INT
Formals ->
Type -> INT
LocalVarDecl -> Type Id ;
Statement -> LocalVarDecl
ReadStmt -> READ ( Id , QString ) ;
Statement -> Readstmt
Type -> INT
LocalVarDecl -> Type Id ;
Statement -> LocalVarDecl
syntax error
```

```

/* This is a comment line in the sample program. */

INT f2(INT x, INT y)
BEGIN
   INT z;
   z := x*x - y*y;
   RETURN z;
END
INT MAIN f1()
BEGIN
   INT x;
   READ(x, "A41.input");
   INT y;
   READ(y, "A42.input)";
   INT z;
   z := f2(x,y) + f2(y,x);
   WRITE(z, "A4.output");
END
```

In summary, each test case successfully triggered the appropriate grammar rule reductions and reported lexical tokens as expected. The program correctly identifies invalid syntax when introduced.

V. Discussion

This assignment demonstrates a practical understanding of how compilers tokenize and parse high-level languages.

Key Insights

- The significance of precise token definitions for correct parsing.
- How operator precedence and associativity are handled in Bison.
- The value of separating lexical and syntax concerns for modularity.

Challenges

- Managing nested rules.
- Ensuring that the parser properly handled all expression forms, particularly function calls as part of expressions.

Future Improvements

- Adding a semantic analysis phase.
- Generating intermediate code (e.g., 3-address code).
- Enhanced error reporting and recovery.

VI. Appendix(complete code)

t_lex.l:

```
%{
#include "t2c.h"
#include "t_parse.h"
%}

%x C_COMMENT

ID  [A-Za-z][A-Za-z0-9]*
DIG [1-9][0-9]*
RNUM {DIG}"."{DIG}
NQUO [^"]

%%

WRITE       {return lWRITE;}
READ        {return lREAD;}
IF          {return lIF;}
ELSE        {return lELSE;}
RETURN            {return lRETURN;}
BEGIN       {return lBEGIN;}
END         {return lEND;}
MAIN        {return lMAIN;}
INT         {return lINT;}
REAL        {return lREAL;}
";"         {return lSEMI;}
","         {return lCOMMA;}
"("         {return lLP;}
")"         {return lRP;}
"+"         {return lADD;}
"-"         {return lMINUS;}
"*"         {return lTIMES;}
"/"         {return lDIVIDE;}
">"         {return lGT;}
"<"         {return lLT;}
":="        {return lASSIGN;}
"=="        {return lEQU;}
"!="        {return lNEQ;}
">="        {return lGE;}
"<="        {return lLE;}

{ID}                    { return lID; }
{DIG}                   { sscanf(yytext,"%d", &ival); return lINUM; }
```

```
{RNUM}                  { sscanf(yytext,"%f", &rval); return lRNUM; }

\"{NQUO}*\" { sscanf(yytext,"%s", qstr); return lQSTR;}
"/*"        { BEGIN(C_COMMENT); }
<C_COMMENT>"*/"   { BEGIN(INITIAL); }
<C_COMMENT>\n     { }
<C_COMMENT>.      { }
[ \t\n]           {}
.           {}

%%

int yywrap() {return 1;}

void print_lex( int t ) {
    switch( t ) {
    case lWRITE: printf("WRITE\n");
        break;
    case lREAD: printf("READ\n");
        break;
    case lIF: printf("IF\n");
        break;
    case lELSE: printf("ELSE\n");
        break;
    case lRETURN: printf("RETURN\n");
        break;
    case lBEGIN: printf("BEGIN\n");
        break;
    case lEND: printf("END\n");
        break;
    case lMAIN: printf("MAIN\n");
        break;
    case lSTRING: printf("STRING\n");
        break;
    case lINT: printf("INT\n");
        break;
    case lREAL: printf("REAL\n");
        break;
    case lSEMI: printf("SEMI\n");
        break;
    case lCOMMA: printf("COMMA\n");
        break;
    case lLP: printf("LP\n");
        break;
    case lRP: printf("RP\n");
        break;
```

```
    case lADD: printf("ADD\n");
        break;
    case lMINUS: printf("MINUS\n");
        break;
    case lTIMES: printf("TIMES\n");
        break;
    case lDIVIDE: printf("DIVIDE\n");
        break;
    case lASSIGN: printf("ASSIGN\n");
        break;
    case lEQU: printf("EQU\n");
        break;
    case lNEQ: printf("NEQ\n");
        break;
    case lID: printf("ID(%s)\n", name);
        break;
    case lINUM: printf("INUM(%d)\n", ival);
        break;
    case lRNUM: printf("RNUM(%f)\n", rval);
        break;
    case lQSTR: printf("QSTR(%s)\n", qstr);
        break;
    default: printf("******** lexical error!!!");
    }
}
```

t_parse.y:

```
%{
      #include <stdio.h>
      #include <stdlib.h>
      #include "t2c.h"
      #include "t_parse.h"
%}

%token lWRITE lREAD lIF lASSIGN
%token lRETURN lBEGIN lEND
%left  lEQU lNEQ lGT lLT lGE lLE
%left  lADD lMINUS
%left  lTIMES lDIVIDE
%token lLP lRP
%token lINT lREAL lSTRING
```

```
%token lELSE
%token lMAIN
%token lSEMI lCOMMA
%token lID lINUM lRNUM lQSTR

%expect 1

%%
prog  :     mthdcls
            { printf("Program -> MethodDecls\n");
              printf("Parsed OK!\n"); }
      |
            { printf("****** Parsing failed!\n"); }
      ;

mthdcls     :     mthdcl mthdcls
            { printf("MethodDecls -> MethodDecl MethodDecls\n"); }
      |     mthdcl
            { printf("MethodDecls -> MethodDecl\n"); }
      ;

type  :     lINT
            { printf("Type -> INT\n"); }
      |     lREAL
            { printf("Type -> REAL\n"); }
      ;

mthdcl      :     type lMAIN lID lLP formals lRP block
            { printf("MethodDecl -> Type MAIN ID LP Formals RP
Block\n"); }
      |     type lID lLP formals lRP block
            { printf("MethodDecl -> Type ID LP Formals RP Block\n"); }
      ;

formals     :     formal oformal
            { printf("Formals -> Formal OtherFormals\n"); }
      |
            { printf("Formals -> \n"); }
      ;

formal      :     type lID
            { printf("Formal -> Type ID\n"); }
      ;

oformal     :     lCOMMA formal oformal
            { printf("OtherFormals -> COMMA Formal OtherFormals\n"); }
```

```
      |
            { printf("OtherFormals -> \n"); }
      ;


block   :   lBEGIN stmts lEND
            { printf("Block -> BEGIN Statements END\n"); }
      ;

stmts :     stmt stmts
            { printf("Statements -> Statement Statements\n"); }
      |     stmt
            { printf("Statements -> Statement\n"); }
      ;

stmt  :     block
            { printf("Statement -> Block\n"); }
      |     localdcl
            { printf("Statement -> LocalVarDecl\n"); }
      |     assignstmt
            { printf("Statement -> AssignStmt\n"); }
      |     returnstmt
            { printf("Statement -> ReturnStmt\n"); }
      |     ifstmt
            { printf("Statement -> IfStmt\n"); }
      |     writestmt
            { printf("Statement -> WriteStmt\n"); }
      |     readstmt
            { printf("Statement -> Readstmt\n"); }
      ;

localdcl:   type lID lSEMI
            { printf("LocalVarDecl -> Type Id ;\n"); }
      |     type assignstmt
            { printf("LocalVarDecl -> Type AssignStmt\n"); }
      ;

assignstmt: lID lASSIGN expr lSEMI
            { printf("AssignStmt -> Id := Expression ;\n"); }
      ;

returnstmt: lRETURN expr lSEMI
            { printf("ReturnStmt -> RETURN Expression ;\n"); }
      ;

ifstmt      :     lIF lLP boolexpr lRP stmt
```

```
            { printf("IfStmt -> IF ( BoolExpression ) Statement\n"); }
      |     lIF lLP boolexpr lRP stmt lELSE stmt
            { printf("IfStmt -> IF ( BoolExpression ) Statement ELSE
Statement\n"); }
      ;

writestmt:  lWRITE lLP expr lCOMMA lQSTR lRP lSEMI
            { printf("WriteStmt -> WRITE ( Expression , QString ) ;\n");
}
      ;

readstmt:   lREAD lLP lID lCOMMA lQSTR lRP lSEMI
            { printf("ReadStmt -> READ ( Id , QString ) ;\n"); }
      ;


expr  :     mulexpr exprtail
            { printf("Expression -> MultiplicativeExpr
ExpressionTail\n"); }
      ;

exprtail:
            { printf("ExpressionTail -> \n"); }
      |     lADD mulexpr exprtail
            { printf("ExpressionTail -> plus MultiplicativeExpr
ExpressionTail\n"); }
      |     lMINUS mulexpr exprtail
            { printf("ExpressionTail -> minus MultiplicativeExpr
ExpressionTail\n"); }
      ;

mulexpr :   primaryexpr multail
            { printf("MultiplicativeExpr -> PrimaryExpression
MultiTail\n"); }
      ;

multail     :
            { printf("MultiTail -> \n"); }
      |     lTIMES primaryexpr multail
            { printf("MultiTail -> times PrimaryExpression MultTail\n");
}
      |     lDIVIDE primaryexpr multail
            { printf("MultiTail -> divides PrimaryExpression
MultTail\n"); }
      ;
```

```
primaryexpr:      lINUM
            { printf("PrimaryExpr -> INUM\n"); }
      |     lRNUM
            { printf("PrimaryExpr -> RNUM\n"); }
      |     lID
            { printf("PrimaryExpr -> ID\n"); }
      |     lLP expr lRP
            { printf("PrimaryExpr -> ( Expression )\n"); }
      |     lID lLP actuals lRP
            { printf("PrimaryExpr -> ID ( ActualParams )\n"); }
      ;

boolexpr:   expr lEQU expr
            { printf("BoolExpr -> Expression == Expression\n"); }
      |     expr lNEQ expr
            { printf("BoolExpr -> Expression != Expression\n"); }
      |     expr lGT expr
            { printf("BoolExpr -> Expression > Expression\n"); }
      |     expr lGE expr
            { printf("BoolExpr -> Expression >= Expression\n"); }
      |     expr lLT expr
            { printf("BoolExpr -> Expression < Expression\n"); }
      |     expr lLE expr
            { printf("BoolExpr -> Expression <= Expression\n"); }
      ;

actuals :   expr oactuals
            { printf("ActualParams -> Expression OtherParams\n"); }
      |
            { printf("ActualParams -> \n"); }
      ;

oactuals:   lCOMMA expr oactuals
            { printf("OtherParams -> COMMA Expression OtherParams\n"); }
      |
            { printf("OtherParams -> \n"); }
      ;

%%

int yyerror(char *s)
{
      printf("%s\n",s);
      return 1;
}
```