Programming Assignment 1: Report

Due on May 30, 2024 at 3:10pm

Professor CHUNG YUNG CS Group

LEE CHIH PIN

Problem 1

Description

In the Assignment 1, We create a programming language call T and need to finish the following tasks:

1. Create a scanner for the language T.
2. Create a parser for the language T.
3. Given a test data, obtain the output of the program.

The T Lexicons

The language T has the following lexicons:

Keywords: WRITE, READ, IF, ELSE, RETURN, BEGIN, END, MAIN, INT, REAL

Single-character Separators: ; , ( )

Single-character Operators: + - * / > <

Multi-character Operators: :=, ==, !=, >=, <=

Identifiers: An integer number is a sequence of digits, where a digit has the following definition:

Digit: $\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Real Numbers: A real number is a sequence of digits, followed by a dot, and followed by digits.

Comments: A comment is a string between `/*` and `*/`. Comments can be longer than one line.

QString: A QString is any sequence of characters except double quote itself, enclosed in double quotes.

Problem 1 continued on next page. . .

The T Grammar

High-level program structures:

- Program → MethodDecl MethodDecl*
- Type → INT | REAL
- MethodDecl → Type [MAIN] Id '(' FormalParams ')' Block
- FormalParams → FormalParam [',' FormalParam]*
- FormalParam → Type Id

Statements:

- Block → BEGIN Statement* END
- Statement → Block

  | LocalVarDecl
  | AssignStmt
  | ReturnStmt
  | IfStmt
  | WriteStmt
  | ReadStmt

- LocalVarDecl → Type Id ';' | Type AssignStmt
- AssignStmt → Id `:=` Expression ';'
- ReturnStmt → RETURN Expression ';'
- IfStmt → IF '(' BoolExpression ')' Statement

  | IF '(' BoolExpression ')' Statement ELSE Statement

- WriteStmt → WRITE '(' Expression ',' QString ')' ';'
- ReadStmt → READ '(' Id ',' QString ')' ';'

Expressions:

- Expression → MultiplicativeExpr { `'+'` | `'-')` MultiplicativeExpr }
- MultiplicativeExpr → PrimaryExpr { `*`|`/` PrimaryExpr }
- PrimaryExpr → Id | '(' Expression ')' | Id '(' ActualParams ')'

  | Id
  | '(' Expression ')'
  | Id '(' ActualParams ')'

- BoolExpr → Expression `==` Expression

  | Expression `!=` Expression

Problem 1 continued on next page. . .

| Expression > Expression

| Expression >= Expression

| Expression < Expression

| Expression <= Expression

- ActualParams → [Expression (‘,’ Expression)*]

Problem 1 continued on next page...

Implementation

Scanner:

The scanner is implemented using the flex tool. The scanner reads the input file and tokenizes the input file. It will output the token and the lexeme of the token.

Parser:

The parser is implemented using the bison tool. The parser will parse the tokenized input file and check if the input file is syntactically correct. It will output the parse tree of the input file.

Test Data:

The test data is a text file that contains the program written in the language T. The test data will be used to test the scanner, the parser, the semantic analyzer, and the code generator.

Problem 1 continued on next page...

The program listing

In the section, we will show the program listing of the scanner, the parser, use gcc to compile the scanner and the parser, finally give the test data and the output of the program.

Scanner:

This is all the code in the $t_lex.l$ file.

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

    %x C_COMMENT

    ID    [A-Za-z][A-Za-z0-9]*
    DIG   [0-9][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;}
```

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




{ID}          { sscanf(yytext, "%s", name); return lID; }
{DIG}         { sscanf(yytext, "%d", &ival); return lINUM; }
{RNUM}        { sscanf(yytext, "%lf", &rval); return lRNUM; }

%%

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;
```

Problem 1 continued on next page. . .

```
            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!!!");
            }
        }
```

Explaination:

The scanner will read the input file and tokenize the input file. It will output the token and the lexeme of the token.

for me, I add three lines in the $t_lex.l$ file.

```
{ID}         { sscanf(yytext, "%s", name); return lID; }
{DIG}        { sscanf(yytext, "%d", &ival); return lINUM; }
{RNUM}       { sscanf(yytext, "%lf", &rval); return lRNUM; }
```

These three lines will read the lexeme of the token and store it in the variable *name*, *ival*, and *rval*.

Parsing:

This is all the code in the $t_parse.y$ file.

```
%{
    #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"); }
    ;

// Statements and Expressions

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

stmt          :        block
        { printf("Statement -> Block\n"); }
    |     lvardecl
        { 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  Statements END\n"); }
    ;

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

Problem 1 continued on next page...

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

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

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

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

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

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

expr        :       multexpr
            { printf("Expression -> MultiplicativeExpr\n"); }
        |       expr lADD multexpr
            { printf("Expression -> Expression ADD MultiplicativeExpr\n"); }
        |       expr lMINUS multexpr
            { printf("Expression -> Expression MINUS MultiplicativeExpr\n"); }
        ;

multexpr    :       primaryexpr
            { printf("MultiplicativeExpr -> PrimaryExpr\n"); }
        |       multexpr lTIMES primaryexpr
            { printf("MultiplicativeExpr -> MultiplicativeExpr TIMES PrimaryExpr\n"); }
        |       multexpr lDIVIDE primaryexpr
            { printf("MultiplicativeExpr -> MultiplicativeExpr DIVIDE PrimaryExpr\n"); }
        ;

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

Problem 1 continued on next page. . .

```
        |       lID lLP actualparams lRP
            { printf("PrimaryExpr -> ID LP ActualParams RP\n"); }
        ;

    boolexpr    :       expr lEQU expr
            { printf("BoolExpr -> Expression EQU Expression\n"); }
        |       expr lNEQ expr
            { printf("BoolExpr -> Expression NEQ Expression\n"); }
        |       expr lGT expr
            { printf("BoolExpr -> Expression GT Expression\n"); }
        |       expr lGE expr
            { printf("BoolExpr -> Expression GE Expression\n"); }
        |       expr lLT expr
            { printf("BoolExpr -> Expression LT Expression\n"); }
        |       expr lLE expr
            { printf("BoolExpr -> Expression LE Expression\n"); }
        ;

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

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

Explaination:

The parser will parse the tokenized input file and check if the input file is syntactically correct. It will output the parse tree of the input file.

for me, I add these codes in the $t_parse.y$ file.

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

stmt    :       block
    { printf("Statement -> Block\n"); }
|       lvardecl
    { 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 Statements END\n"); }
;

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

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

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

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

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

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

expr    :       multexpr
    { printf("Expression -> MultiplicativeExpr\n"); }
|       expr lADD multexpr
    { printf("Expression -> Expression ADD MultiplicativeExpr\n"); }
|       expr lMINUS multexpr
    { printf("Expression -> Expression MINUS MultiplicativeExpr\n"); }
```

```
;

multexpr        :       primaryexpr
    { printf("MultiplicativeExpr -> PrimaryExpr\n"); }
|       multexpr lTIMES primaryexpr
    { printf("MultiplicativeExpr -> MultiplicativeExpr TIMES PrimaryExpr\n"); }
|       multexpr lDIVIDE primaryexpr
    { printf("MultiplicativeExpr -> MultiplicativeExpr DIVIDE PrimaryExpr\n"); }
;

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

boolexpr        :       expr lEQU expr
    { printf("BoolExpr -> Expression EQU Expression\n"); }
|       expr lNEQ expr
    { printf("BoolExpr -> Expression NEQ Expression\n"); }
|       expr lGT expr
    { printf("BoolExpr -> Expression GT Expression\n"); }
|       expr lGE expr
    { printf("BoolExpr -> Expression GE Expression\n"); }
|       expr lLT expr
    { printf("BoolExpr -> Expression LT Expression\n"); }
|       expr lLE expr
    { printf("BoolExpr -> Expression LE Expression\n"); }
;

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

Problem 1 continued on next page...

These codes to defined the grammar of the language T:

Statement Parsing

The parser defines rules for statements, which can be either a single statement or a sequence of statements:

- **stmts**: Matches one or more consecutive statements. This recursive definition allows for the flexible expansion of statement sequences.
  - `stmt stmts` – Matches at least two consecutive statements.
  - `stmt` – Matches a single statement.
- **stmt**: Defines various possible types of statements, such as blocks, variable declarations, assignment statements, return statements, conditional statements, input, and output statements.

Specific Statement Types

- **block**: Matches a sequence of statements surrounded by `BEGIN` and `END`.
- **lvardecl**: Matches local variable declarations either as a simple type and identifier or as an initialized assignment statement.
- **assignstmt**: Matches an assignment statement that includes an identifier, an assignment operator, and an expression, followed by a semicolon.
- **returnstmt**: Matches a return statement used to return the value of an expression from a function.
- **ifstmt**: Matches a conditional statement with an optional `ELSE` branch for alternate execution.
- **writestmt** and **readstmt**: Match output and input statements, respectively, for basic I/O operations.

Expression Parsing

Expressions are constructed using the following rules:

- **expr**: Defines basic arithmetic expressions supporting addition and subtraction.
- **multexpr**: Handles multiplication and division, essential for constructing arithmetic operations.
- **primaryexpr**: Defines atomic components of expressions, including integers, reals, identifiers, bracketed expressions, and function calls.

Boolean Expressions

- **boolexpr**: Used for matching and evaluating boolean expressions with operators like equals, not equals, greater than, etc.

Function Call Parameters

- **actualparams**: Matches a list of actual parameters in a function call, supporting one or more expressions separated by commas.

Problem 1 continued on next page. . .

Test run results

In the testing, the professional give us 7 .txt files about T programming language, and we need to use the scanner, the parser to get the output of the program.

Result of test.t-test5.t:

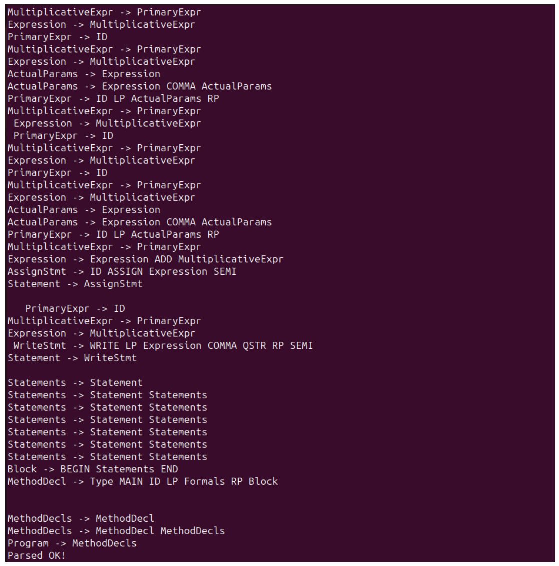

Figure 1: Test

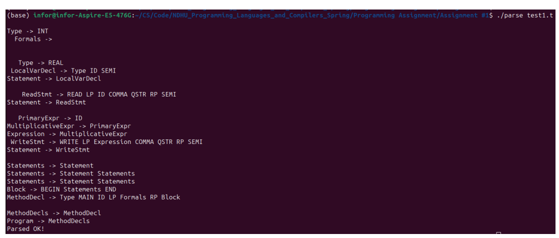

Figure 2: Test1

According to the test data, the scanner and the parser can parse the input file test.t-test5.t test correctly.

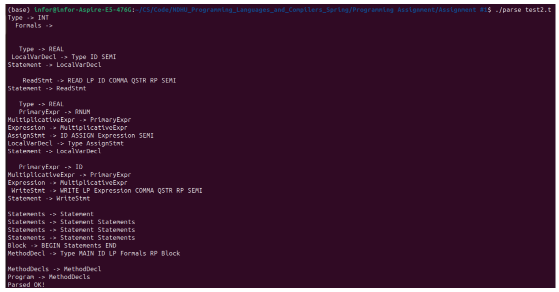

Figure 3: Test2

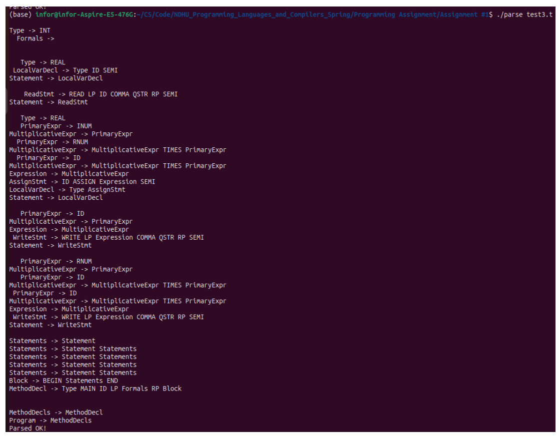

Figure 4: Test3

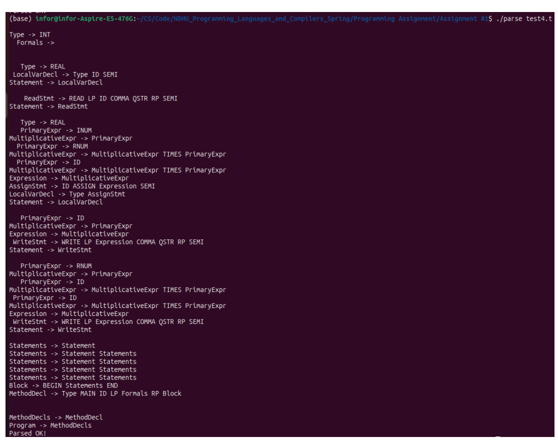

Figure 5: Test4

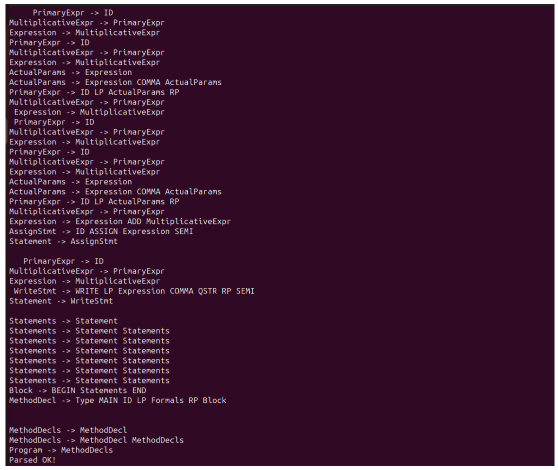

Figure 6: Test5

According to the test data test7.t, we can find that the code syntax is wrong, so that the parser will output the error message.

Result of test7.t:

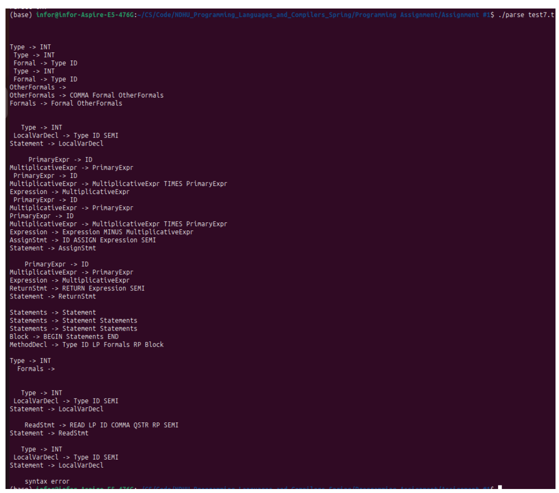

Figure 7: Test7

Discussion

In the lab, we learn how to create a scanner, a parser for a programming language. We also learn how to use the flex and bison tools to create a scanner and a parser. We also learn how to use the gcc compiler to compile the scanner and the parser.

In my opinion, the most challenging part of the lab is to create the grammar for the programming language. It is challenging because we need to define the grammar for the programming language, which is not an easy task.