資訊工程學系

程式語言與編譯器 程式設計作業報告

學號:411121273

姓名:鄭宇翔

目錄:

The problem description

使用 FLEX 與 Bison 來設計一個詞法與語法分析，並建立一個能夠解析特定語言的解析器。

需要完成以下步驟：

1. 撰寫 lex 文件（t_lex.l）：定義詞法規則。
2. 撰寫 yacc 文件（t_parse.y）：定義語法規則。
3. 寫一個主函數（t2c.c）
4. 寫一個標頭檔（t2c.h）
5. 使用 Bison 來編譯 t_parse.y 生成 t_parse.c 和標頭檔 t_parse.h。
   指令: bison -d -o t_parse.c t_parse.y
6. 使用 FLEX 編譯 t_lex.l 生成 t_lex.c。
   指令: flex -o t_lex.c t_lex.l
7. 使用 gcc 對生成的.c 文件進行編譯。
   指令: gcc -c t_lex.c
   gcc -c t_parse.c
   gcc-c t2c.c
8. 鏈接所有的.o 文件生成可執行文件 parse。
   指令: gcc-o parse t2c.o t_lex.o t_parse.o
9. 通過七個測試檔的測試。
   需滿足前六個回報正確，以及第七個回報錯誤。

以下是已提供之文件:

.DS_Store	test1.t
Makefile	test2.t
t_lexMain.c	test3.t
t2t.c	test4.t
t2t.h	test5.t
test.	test7.t

任務目標:

我們需要完成 t_lex.l 以及 t_parse.y 中缺少的程式碼，將其缺少的詞法以及文法補上並且通過 test-test7 的所有測試。

The way I write the program

t_flex.l

```
{ID}{sscanf(yytext,"%s", name);yylval.sr = strdup( name );return lID;}
{DIG}    {sscanf(yytext,"%d", &ival);yylval.iv = ival;return lINUM;}
{RNUM}   {sscanf(yytext,"%f", &rval);yylval.rv = rval;return lRNUM;}
```

這三行程式碼定義了當 flex 遇到識別符（ID）、整數（DIG），以及實數（RNUM）時該怎麼做處理。

ID 的處理：[A-Za-z][A-Za-z0-9]

sscanf(yytext,"%s", name);

從 flex 的輸入緩衝區 yytext 中讀取並保存到 name 變數中。

yylval.sr = strdup(name);

將 name 中的字串複製一份（使用 strdup 函數），

並將複製的字串地址丟給 yylval.sr。

yylval 是 flex 和 bison 之間通信的橋樑，用來傳遞詞素的值。

return lID;

表示當前匹配到的文本是一個識別符，通知 bison。

DIG 的處理：[0-9][0-9]

sscanf(yytext,"%d", &ival);

從 yytext 中讀取匹配到的文本，並將其解析為一個整數，保存到 ival 變數中。

yylval.iv = ival;

將讀取到的整數賦值給 yylval.iv。

return lINUM;

表示當前匹配到的文本是一個整數，並通知 bison 這一結果。

RNUM 的處理：{DIG}"."{DIG}

sscanf(yytext,"%f", &rval);

從 yytext 中讀取，並將其解析為一個浮點數，保存到 rval 變數中。

yylval.rv = rval;

將讀取到的浮點數賦值給 yylval.rv。

return lRNUM;

當匹配到一個實數時，通知 bison。

增加了這三條規則讓我的編譯器能夠識別並處理識別符、整數和實數，並將它們傳遞給 bison 進一步處理。

t_bison.y

```
Block   :   lBEGIN stmts lEND
        { printf("block ok!\n"); }
        ;

stmts   :   stmt stmts
        |   stmt
        ;

stmt    :   Block
        |   vardcl
        |   astm
        |   rstm
        |   istm
        |   wstm
        |   dstm
        ;

vardcl: type lID lSEMI
           { printf("LocalVarDecl -> Type ID SEMI\n"); }
           | type lID lASSIGN expr lSEMI
           { printf("LocalVarDecl -> Type ID ASSIGN Expr SEMI\n"); }
           ;

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

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

istm: lIF lLP bexpr lRP stmt
       { printf("IfStmt -> IF LP BoolExpr RP Stmt\n"); }
       | lIF lLP bexpr lRP stmt lELSE stmt
       { printf("IfStmt -> IF LP BoolExpr RP Stmt ELSE Stmt\n"); }
       ;
```

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


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


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


mexprs: lADD mexpr mexprs
      { printf("MExprs -> ADD MExpr MExprs\n"); }
      | lMINUS mexpr mexprs
      { printf("MExprs -> MINUS MExpr MExprs\n"); }
      |
      { printf("MExprs -> \n"); }
      ;

mexpr: pexpr pexprs
     { printf("MExpr -> PExpr PExprs\n"); }
     ;

pexprs: lTIMES pexpr pexprs
      { printf("PExprs -> TIMES PExpr PExprs\n"); }
      | lDIVIDE pexpr pexprs
      { printf("PExprs -> DIVIDE PExpr PExprs\n"); }
      |
      { printf("PExprs -> \n"); }
      ;
```

```
pexpr: lINUM
      { printf("PExpr -> INUM\n"); }
      | lRNUM
      { printf("PExpr -> RNUM\n"); }
      | lID
      { printf("PExpr -> ID\n"); }
      | lLP expr lRP
      { printf("PExpr -> LP Expr RP\n"); }
      | lID lLP aparams lRP
      { printf("PExpr -> ID LP ActualParams RP\n"); }
        ;


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

aparams: expr oparams
           { printf("ActualParams -> Expr MoreExprs\n"); }
           |
           { printf("ActualParams -> \n"); }
           ;

oparams: lCOMMA expr oparams
         { printf("MoreExprs -> COMMA Expr MoreExprs\n"); }
         |
         { printf("MoreExprs -> \n"); }
         ;
```

Block（表是一個函數區塊的開始與結束）：

表示一個程式碼區塊是從 BEGIN 開始，到 END 結束。裡面包含許多 stmts。當匹配成功時，會輸出"block ok!"表示一個程式函數正確解析。

stmts（多個聲明）：

表示一系列的聲明，可以是一個聲明跟著更多的聲明，或一個單獨的聲明。

stmt（單個聲明）：

定義了一個聲明可以是很多種類型，例如:Block,vardcl,astm,rstm 等等。

vardcl（變數宣告）：

可以是簡單的宣告一個變數，也可以是宣告的同時賦值。

astm（賦值聲明）：

將一個表達式（expr）的結果賦給一個識別符（ID）。

rstm（返回聲明）：

將一個表達式的結果作為函式的返回值。

istm（if 聲明）：

處理 if 聲明，可以包含一個 else 分支。

wstm（寫入聲明）：

表示一條寫入操作的聲明。

dstm（讀取聲明）

表示一條讀取操作的聲明。

expr（表達式）：

一個表達式可以由多個部分組合而成。

mexprs（數學表達式序列）和 pexprs（優先表達式序列）：

處理加法、減法、乘法、除法等運算的表達式。

pexpr（優先表達式）：

這是基本的表達式單元，可以是數字、變數名、或者括號中的表達式等。

bexpr（布林表達式）：

這是用於 if 聲明中的條件表達式，可以比較兩個表達式的值。

aparams 和 oparams：

這些規則用來處理函數調用時傳遞的參數。

以上為我新增的語法規則，定義了一個程式語言的語法結構，告訴電腦如何根據不同的規則去理解程式碼中的各種元素，每條文法都像是一條定律，告訴電腦如果看到這種模式，就應該怎麼理解它，怎麼處理它，並且返回相應的輸出。

The program listing.

t_flex.l

```
%{
#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;}

{ID}        {sscanf(yytext,"%s", name);
         yylval.sr = strdup( name );
         return lID;}
{DIG}       {sscanf(yytext,"%d", &ival);
         yylval.iv = ival;
         return lINUM;}
{RNUM}      {sscanf(yytext,"%f", &rval);
         yylval.rv = 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_bison.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

%union { int iv;
         float rv;
         char* sr;
       }

%type <sr> lID
%type <iv> lINUM
%type <rv> lRNUM
%type <sr> 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 ok!\n"); }
        ;

stmts   :   stmt stmts
        |   stmt
        ;

stmt    :   Block
        |   vardcl
        |   astm
        |   rstm
        |   istm
        |   wstm
        |   dstm
        ;

vardcl: type lID lSEMI
           { printf("LocalVarDecl -> Type ID SEMI\n"); }
           | type lID lASSIGN expr lSEMI
           { printf("LocalVarDecl -> Type ID ASSIGN Expr SEMI\n"); }
           ;

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

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

istm: lIF lLP bexpr lRP stmt
       { printf("IfStmt -> IF LP BoolExpr RP Stmt\n"); }
```

```
        | lIF lLP bexpr lRP stmt lELSE stmt
        { printf("IfStmt -> IF LP BoolExpr RP Stmt ELSE Stmt\n"); }
        ;

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

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

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

mexprs: lADD mexpr mexprs
       { printf("MExprs -> ADD MExpr MExprs\n"); }
       | lMINUS mexpr mexprs
       { printf("MExprs -> MINUS MExpr MExprs\n"); }
       |
       { printf("MExprs -> \n"); }
       ;

mexpr: pexpr pexprs
     { printf("MExpr -> PExpr PExprs\n"); }
     ;

pexprs: lTIMES pexpr pexprs
       { printf("PExprs -> TIMES PExpr PExprs\n"); }
       | lDIVIDE pexpr pexprs
       { printf("PExprs -> DIVIDE PExpr PExprs\n"); }
       |
       { printf("PExprs -> \n"); }
       ;
pexpr: lINUM
       { printf("PExpr -> INUM\n"); }
       | lRNUM
```

```
      { printf("PExpr -> RNUM\n"); }
      | lID
      { printf("PExpr -> ID\n"); }
      | lLP expr lRP
      { printf("PExpr -> LP Expr RP\n"); }
      | lID lLP aparams lRP
      { printf("PExpr -> ID LP ActualParams RP\n"); }
        ;
bexpr: expr lEQU expr
        { printf("BoolExpr -> Expr EQU Expr\n"); }
        | expr lNEQ expr
        { printf("BoolExpr -> Expr NEQ Expr\n"); }
        | expr lGT expr
        { printf("BoolExpr -> Expr GT Expr\n"); }
        | expr lGE expr
        { printf("BoolExpr -> Expr GE Expr\n"); }
        | expr lLT expr
        { printf("BoolExpr -> Expr LT Expr\n"); }
        | expr lLE expr
        { printf("BoolExpr -> Expr LE Expr\n"); }
        ;
aparams: expr oparams
           { printf("ActualParams -> Expr MoreExprs\n"); }
           |
           { printf("ActualParams -> \n"); }
           ;

oparams: lCOMMA expr oparams
         { printf("MoreExprs -> COMMA Expr MoreExprs\n"); }
         |
         { printf("MoreExprs -> \n"); }
         ;
%%
int yyerror(char *s)
{
    printf("%s\n", s);
    return 1;
}
```

Test run results.

test.t

```
shane@MSI:/mnt/c/Users/shane/Music/t2cHW1/t2cHW1$ ./parse test.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 SEMI
PExpr -> ID
PExpr -> ID
PExprs ->
PExprs -> TIMES PExpr PExprs
MExpr -> PExpr PExprs
PExpr -> ID
PExpr -> ID
PExprs ->
PExprs -> TIMES PExpr PExprs
MExpr -> PExpr PExprs
MExprs ->
MExprs -> MINUS MExpr MExprs
Expr -> MExpr MoreExprs
AssignStmt -> ID ASSIGN Expr SEMI
PExpr -> ID
PExprs ->
MExpr -> PExpr PExprs
MExprs ->
Expr -> MExpr MoreExprs
ReturnStmt -> RETURN Expr SEMI
block ok!
MethodDecl -> Type ID LP Formals RP Block
MethodDecls -> MethodDecl
Program -> MethodDecls
Parsed OK!
```

test1.t

```
shane@MSI:/mnt/c/Users/shane/Music/t2cHW1/t2cHW1$ ./parse test1.t
Type -> INT
Formals ->
Type -> REAL
LocalVarDecl -> Type ID SEMI
ReadStmt -> READ LP ID COMMA QSTR RP SEMI
PExpr -> ID
PExprs ->
MExpr -> PExpr PExprs
MExprs ->
Expr -> MExpr MoreExprs
WriteStmt -> WRITE LP Expr COMMA QSTR RP SEMI
block ok!
MethodDecl -> Type MAIN ID LP Formals RP Block
MethodDecls -> MethodDecl
Program -> MethodDecls
Parsed OK!
```

test2.t

```
shane@MSI:/mnt/c/Users/shane/Music/t2cHW1/t2cHW1$ ./parse test2.t
Type -> INT
Formals ->
Type -> REAL
LocalVarDecl -> Type ID SEMI
ReadStmt -> READ LP ID COMMA QSTR RP SEMI
Type -> REAL
PExpr -> RNUM
PExprs ->
MExpr -> PExpr PExprs
MExprs ->
Expr -> MExpr MoreExprs
LocalVarDecl -> Type ID ASSIGN Expr SEMI
PExpr -> ID
PExprs ->
MExpr -> PExpr PExprs
MExprs ->
Expr -> MExpr MoreExprs
WriteStmt -> WRITE LP Expr COMMA QSTR RP SEMI
block ok!
MethodDecl -> Type MAIN ID LP Formals RP Block
MethodDecls -> MethodDecl
Program -> MethodDecls
Parsed OK!
```

test3.t

```
shane@MSI:/mnt/c/Users/shane/Music/t2cHW1/t2cHW1$ ./parse test3.t
Type -> INT
Formals ->
Type -> REAL
LocalVarDecl -> Type ID SEMI
ReadStmt -> READ LP ID COMMA QSTR RP SEMI
Type -> REAL
PExpr -> INUM
PExpr -> RNUM
PExpr -> ID
PExprs ->
PExprs -> TIMES PExpr PExprs
PExprs -> TIMES PExpr PExprs
MExpr -> PExpr PExprs
MExprs ->
Expr -> MExpr MoreExprs
LocalVarDecl -> Type ID ASSIGN Expr SEMI
PExpr -> ID
PExprs ->
MExpr -> PExpr PExprs
MExprs ->
Expr -> MExpr MoreExprs
WriteStmt -> WRITE LP Expr COMMA QSTR RP SEMI
PExpr -> RNUM
PExpr -> ID
PExpr -> ID
PExprs ->
PExprs -> TIMES PExpr PExprs
PExprs -> TIMES PExpr PExprs
MExpr -> PExpr PExprs
MExprs ->
Expr -> MExpr MoreExprs
WriteStmt -> WRITE LP Expr COMMA QSTR RP SEMI
block ok!
MethodDecl -> Type MAIN ID LP Formals RP Block
MethodDecls -> MethodDecl
Program -> MethodDecls
Parsed OK!
```

test4.t

```
shane@MSI:/mnt/c/Users/shane/Music/t2cHW1/t2cHW1$ ./parse test4.t
Type -> INT
Formals ->
Type -> REAL
LocalVarDecl -> Type ID SEMI
ReadStmt -> READ LP ID COMMA QSTR RP SEMI
Type -> REAL
PExpr -> INUM
PExpr -> RNUM
PExpr -> ID
PExprs ->
PExprs -> TIMES PExpr PExprs
PExprs -> TIMES PExpr PExprs
MExpr -> PExpr PExprs
MExprs ->
Expr -> MExpr MoreExprs
LocalVarDecl -> Type ID ASSIGN Expr SEMI
PExpr -> ID
PExprs ->
MExpr -> PExpr PExprs
MExprs ->
Expr -> MExpr MoreExprs
WriteStmt -> WRITE LP Expr COMMA QSTR RP SEMI
PExpr -> RNUM
PExpr -> ID
PExpr -> ID
PExprs ->
PExprs -> TIMES PExpr PExprs
PExprs -> TIMES PExpr PExprs
MExpr -> PExpr PExprs
MExprs ->
Expr -> MExpr MoreExprs
WriteStmt -> WRITE LP Expr COMMA QSTR RP SEMI
block ok!
MethodDecl -> Type MAIN ID LP Formals RP Block
MethodDecls -> MethodDecl
Program -> MethodDecls
Parsed OK!
```

test5.t

```
shane@MSI:/mnt/c/Users/shane/Music/t2cHW1/t2cHW1$ ./parse 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 SEMI
PExpr -> ID
PExpr -> ID
PExprs ->
PExprs -> TIMES PExpr PExprs
MExpr -> PExpr PExprs
PExpr -> ID
PExpr -> ID
PExprs ->
PExprs -> TIMES PExpr PExprs
MExpr -> PExpr PExprs
MExprs ->
MExprs -> MINUS MExpr MExprs
Expr -> MExpr MoreExprs
AssignStmt -> ID ASSIGN Expr SEMI
PExpr -> ID
PExprs ->
MExpr -> PExpr PExprs
MExprs ->
Expr -> MExpr MoreExprs
ReturnStmt -> RETURN Expr SEMI
block ok!
MethodDecl -> Type ID LP Formals RP Block
Type -> INT
Formals ->
Type -> INT
LocalVarDecl -> Type ID SEMI
ReadStmt -> READ LP ID COMMA QSTR RP SEMI
Type -> INT
```

```
LocalVarDecl -> Type ID SEMI
ReadStmt -> READ LP ID COMMA QSTR RP SEMI
Type -> INT
LocalVarDecl -> Type ID SEMI
PExpr -> ID
PExprs ->
MExpr -> PExpr PExprs
MExprs ->
Expr -> MExpr MoreExprs
PExpr -> ID
PExprs ->
MExpr -> PExpr PExprs
MExprs ->
Expr -> MExpr MoreExprs
MoreExprs ->
MoreExprs -> COMMA Expr MoreExprs
ActualParams -> Expr MoreExprs
PExpr -> ID LP ActualParams RP
PExprs ->
MExpr -> PExpr PExprs
PExpr -> ID
PExprs ->
MExpr -> PExpr PExprs
MExprs ->
Expr -> MExpr MoreExprs
PExpr -> ID
PExprs ->
MExpr -> PExpr PExprs
MExprs ->
Expr -> MExpr MoreExprs
MoreExprs ->
MoreExprs -> COMMA Expr MoreExprs
ActualParams -> Expr MoreExprs
PExpr -> ID LP ActualParams RP
PExprs ->
MExpr -> PExpr PExprs
MExprs ->
MExprs -> ADD MExpr MExprs
```

```
MExprs -> ADD MExpr MExprs
Expr -> MExpr MoreExprs
AssignStmt -> ID ASSIGN Expr SEMI
PExpr -> ID
PExprs ->
MExpr -> PExpr PExprs
MExprs ->
Expr -> MExpr MoreExprs
WriteStmt -> WRITE LP Expr COMMA QSTR RP SEMI
block ok!
MethodDecl -> Type MAIN ID LP Formals RP Block
MethodDecls -> MethodDecl
MethodDecls -> MethodDecl MethodDecls
Program -> MethodDecls
Parsed OK!
```

test7.t

```
shane@MSI:/mnt/c/Users/shane/Music/t2cHW1/t2cHW1$ ./parse test7.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 SEMI
PExpr -> ID
PExpr -> ID
PExprs ->
PExprs -> TIMES PExpr PExprs
MExpr -> PExpr PExprs
PExpr -> ID
PExpr -> ID
PExprs ->
PExprs -> TIMES PExpr PExprs
MExpr -> PExpr PExprs
MExprs ->
MExprs -> MINUS MExpr MExprs
Expr -> MExpr MoreExprs
AssignStmt -> ID ASSIGN Expr SEMI
PExpr -> ID
PExprs ->
MExpr -> PExpr PExprs
MExprs ->
Expr -> MExpr MoreExprs
ReturnStmt -> RETURN Expr SEMI
block ok!
MethodDecl -> Type ID LP Formals RP Block
Type -> INT
Formals ->
Type -> INT
LocalVarDecl -> Type ID SEMI
ReadStmt -> READ LP ID COMMA QSTR RP SEMI
Type -> INT
LocalVarDecl -> Type ID SEMI
syntax error
```

Discussion.

自從學習程式語言以來，一直接觸的都是高階語言，例如 C/C++等，但將這些高階語言轉換為機器碼的編譯器一直在使用，但卻不知道它的原理，只知道按下編譯的按鈕，程式執行的結果就會跑出來，然後就可以收工了，但在這之後的幕後功臣我卻一直沒有去了解它。

編譯器這堂課程帶我們從零開始了解編譯器的設計原理與功能，並且透過這樣作業，讓我有機會親手接觸簡單的 scanner 以及 parser 的開發，雖然只是編譯器設計中最簡單的部分，代也因為初次接觸，對很多規則與語法以及環境的建置等等都不熟悉，因此也耗費了我數個假日。

而在完成這份作業的過程中，我也遭遇了許多困難，從開發環境的建置開始，由於我是使用 windows 系統的關係，出現了許多不可控的錯誤，例如無法使用 Bison 來編譯 t_parse.y、報錯是一連串問號、不會輸出文字等等，我試過了重新安裝、更改 IDE、修改電腦字型等，最後透過使用 Linux 系統解決了環境建置的問題，也藉著這個機會安裝並接觸到了 Linux 系統，學會了一些基本常用的指令，讓我能夠在 Linux 系統上面做開發，當環境好不容易建置好後，原本覺得上課聽起來也沒這麼困難，但當真的自己實作起來時卻困難重重，於實我乖乖地回去看原文書，搭配教授的講義與網路上有關編譯器的文章，才慢慢理解這個作業的目標，其中我認為最困難的部分是這項作業的檔案數較多，需要互相配合最後鏈結出來的 parse 才能工作，因此只要有某個檔案出現錯誤，就會需要花費很多的時間 debug，再加上對 flex 以及 bison 的不熟悉，除錯的任務更是難上加難。

透過這次的作業練習，並沒有讓我成為編譯器方面的專家，但卻引領我初步的認識了編譯器，也接觸到了 flex 與 biosn。往後，當我在撰寫高階語言的時候，也能了解編譯的過程與原理，也知道自己撰寫的程式碼是如何轉變為讓電腦可以執行的機器碼，讓自己對於整個電腦軟體的運作有更深的認識。