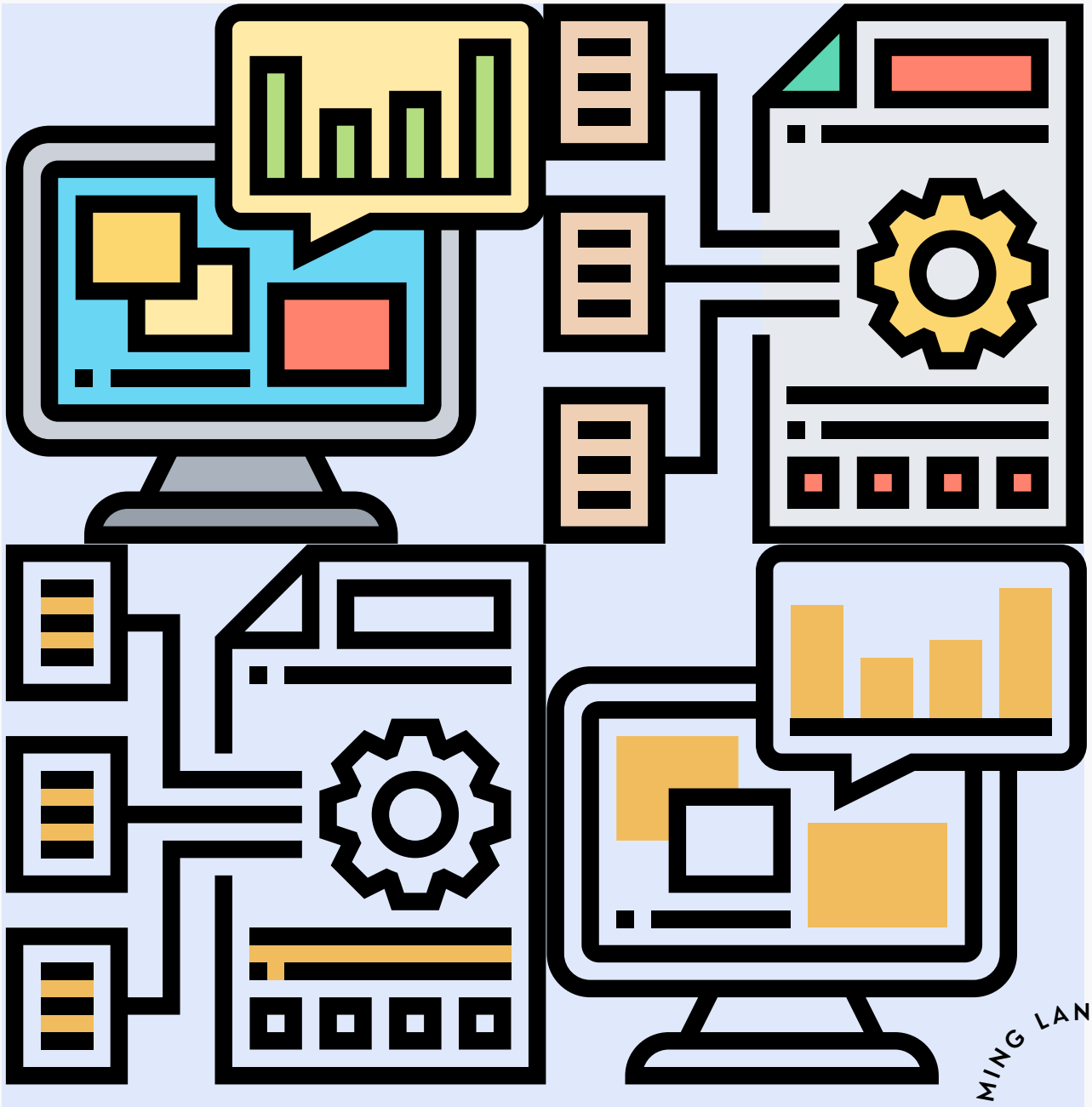
2022

using lex (or flex) and yacc (or bison))

Implementation of a Compiler Front-end



Presented by
41092308 游凱卉
41092365 陳祁鎔

PROGRAMMING LANGUAGE&COMPILER

Table of Contents

Description

Lex is a program that generates lexical analyzer. It is used with YACC parser generator. The lexical analyzer is a program that transforms an input stream into a sequence of tokens. It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

FLEX (Fast LEXical analyzer generator) is a tool for generating scanners. First, FLEX reads a specification of a scanner either from an input file *.lex, or from standard input, and it generates as output a C source file lex.yy.c. Then, lex.yy.c is compiled and linked with the "-lfl" library to produce an executable a.out. Finally, a.out analyzes its input stream and transforms it into a sequence of tokens.

YACC (yet another compiler-compiler) is an LALR(1) (LookAhead, Left-to-right, Rightmost derivation producer with 1 lookahead token) parser generator. YACC was originally designed for being complemented by Lex. A parser generator is a program that takes as input a specification of a syntax, and produces as output a procedure for recognizing that language. Historically, they are also called compiler-compilers.

Bison is a general-purpose parser generator that converts an annotated context-free grammar into a deterministic LR or generalized LR (GLR) parser employing LALR(1), IELR(1) or canonical LR(1) parser tables.

For this homework, we used lex(or flex) and yacc(or bison) to implement the front-end(including a lexical analyzer and a syntax recognizer) of the compiler for the MiniJ programming language, which is a simplified version of Java especially designed for a compiler construction project by Professor Chung Yung.

We implemented those program rules practically and got these result as following.

Highlights
of the way we wrote the program

> Before software should be reusable, it should be usable.
>
> --Ralph Johnson

Flex and bison are tools designed for writers of compilers and interpreters.

We wrote a flex progem(.l) and a bison program(.y). Both of them have their own specification and rules. We followed those instructions and completed the codes.

For FLEX specification,
{definitions}
%%
{rules}
%%
{user routines}

For BISON specification,
{definitions}
%%
{rules}
%%
{user routines}

They might briefly seem like the same. However, their patterns, actions, or grammers are different actually.

Let's read the codes below to find differences.

Flex codes

minij_lex.l - 記事本

檔案(F) 編輯(E) 格式(O) 檢視(V) 說明

```
%{
#define YYSTYPE char *
#include <string.h>
#include <stdlib.h>
#include "minij.h"
#include "minij_parse.h"
%}

ID  [A-Za-z][A-Za-z0-9_]*
LIT [0-9][0-9]*
NONNL [^\n]
%x  COMMENT
%option yylineno

%%

class           {return CLASS;}
public          {return PUB;}
static          {return STATIC;}
String          {return STR;}
void            {return VOID;}
main            {return MAIN;}
int                     {return INT;}
if                      {return IF;}
else            {return ELSE;}
while           {return WHILE;}
new                     {return NEW;}
return          {return RETURN;}
this            {return THIS;}
true            {return TRUE;}
false           {return FALSE;}
"&&"            {return AND;}
"<"                     {return LT;}
"<="            {return LE;}
"+"                     {return ADD;}
"-"                     {return MINUS;}
"*"                     {return TIMES;}
"("                     {return LP;}
")"                     {return RP;}
"{"                     {return LBP;}
"}"                     {return RBP;}
","                     {return COMMA;}
"."                     {return DOT;}
System.Out.println {return PRINT;}
"||"        {return OR;}
"=="        {return EQ;}
"["         {return LSP;}
"]"         {return RSP;}
";"         {return SEMI;}
"="         {return ASSIGN;}
"//"        {BEGIN COMMENT;}
<COMMENT>\n {BEGIN INITIAL;}
<COMMENT>.  { }
{ID}        {yylval = strdup(yytext); return ID;}
{LIT}       {yylval = strdup(yytext); return LIT;}

[ \t\n]                 {/* skip BLANK */}
.                       {/* skip redundant characters */}

%%

int yywrap() {return 1;}
```

Definitions

Rules

(We added these rules in by using its definition)

User routines

Bison codes

minij_parse.y - 記事本

檔案(F) 編輯(E) 格式(O) 檢視(V) 說明

```
%{
        #include <stdio.h>
        #include <stdlib.h>
        #include "minij.h"
        #include "minij_parse.h"
%}

%token CLASS PUB STATIC
%left  AND OR
%left  LT LE EQ
%left  ADD MINUS
%left  TIMES
%token LBP RBP LSP RSP LP RP
%token INT BOOLEAN
%token IF ELSE
%token WHILE PRINT
%token ASSIGN
%token VOID MAIN STR
%token RETURN
%token SEMI COMMA
%token THIS NEW DOT
%token ID LIT TRUE FALSE NOT

%expect 32

%%
prog    :       mainc cdcls
                { printf("Program -> MainClass ClassDecl*\n");
                  printf("Parsed OK!\n"); }
        |
                { printf("Parsing failed!\n"); }
        ;

mainc   :       CLASS ID LBP PUB STATIC VOID MAIN LP STR LSP RSP ID RP LBP stmts RBP RBP
                { printf("MainClass -> class id lbp public static void main lp string lsp rsp id rp lbp Statemet* rbp rbp\n"); }
        ;

cdcls   :       cdcl cdcls
                { printf("(for ClassDecl*) cdcls : cdcl cdcls\n"); }
        |
                { printf("(for ClassDecl*) cdcls : \n"); }
        ;

cdcl    :       CLASS ID LBP vdcls mdcls RBP
                { printf("ClassDecl -> class id lbp VarDecl* MethodDecl* rbp\n"); }
        ;

vdcls   :       vdcl vdcls
                { printf("(for VarDecl*) vdcls : vdcl vdcls\n"); }
        |
                { printf("(for VarDecl*) vdcls : \n"); }
        ;

vdcl    :       type ID SEMI
                { printf("VarDecl -> Type id semi\n"); }
        ;

mdcls   :       mdcl mdcls
                { printf("(for MethodDecl*) mdcls : mdcl mdcls\n"); }
        |
                { printf("(for MethodDecl*) mdcls : \n"); }
        ;

mdcl    :       PUB type ID LP formals RP LBP vdcls stmts RETURN exp SEMI RBP
                { printf("MethodDecl -> public Type id lp FormalList rp lbp Statements* return Exp semi rbp\n"); }
        ;

formals :       type ID frest
                { printf("FormalList -> Type id FormalRest*\n"); }
        |
                { printf("FormalList -> \n"); }
        ;
```

Definitions

C Declarations

define tokens, and precedence of tokens here

expect declaration for shift/reduce conflicts

Rules Grammar rules

Bison codes

```
frest   :       COMMA type ID frest
                { printf("FormalRest -> comma Type id FormalRest\n"); }
        |
                { printf("FormalRest -> \n"); }
        ;

type    :       INT LSP RSP
                { printf("Type -> INT LSP RSP \n"); }
        |
                BOOLEAN
                { printf("Type -> BOOLEAN \n"); }
        |
                INT
                { printf("Type -> INT \n"); }
        |
                ID
                { printf("Type ->\n"); }
        ;

stmt    :       LBP stmts RBP
                { printf("stmt -> lbp Statement* rbp \n"); }
        |
                IF LP exp RP stmt ELSE stmt
                { printf("stmt -> if lp Exp rp Statement else Statement \n"); }
        |
                WHILE LP exp RP stmt
                { printf("stmt -> while lp Exp rp Statement \n"); }
        |
                PRINT LP exp RP SEMI
                { printf("stmt -> print lp Exp rp semi \n"); }
        |
                ID ASSIGN exp SEMI
                { printf("stmt -> id assign Exp rp semi \n"); }
        |
                ID LSP exp RSP ASSIGN exp SEMI
                { printf("stmt -> id lsp Exp rsp assign Exp semi \n"); }
        |
                vdcl
                { printf("stmt -> VarDecl \n"); }
        ;

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

exp     :       exp ADD exp
                { printf("exp -> Exp add Exp \n"); }
        |
                exp MINUS exp
                { printf("exp -> Exp minus Exp \n"); }
        |
                exp TIMES exp
                { printf("exp -> Exp times Exp \n"); }
        |
                exp AND exp
                { printf("exp -> Exp and Exp \n"); }
        |
                exp OR exp
                { printf("exp -> Exp or Exp \n"); }
        |
                exp LT exp
                { printf("exp -> Exp lt Exp \n"); }
        |
                exp LE exp
                { printf("exp -> Exp le Exp \n"); }
        |
                exp EQ exp
                { printf("exp -> Exp eq Exp \n"); }
        |
                ID LSP exp RSP
                { printf("exp -> id lsp Exp rsp \n"); }
        |
                ID LP expls RP
                { printf("exp -> id lp Explist rp \n"); }
        |
                LP exp RP
                { printf("exp -> lp Exp rp \n"); }
```

Rules Grammar rules

Bison codes

```
        |
                exp DOT exp
            { printf("exp -> Exp dot Exp \n"); }
        |
                LIT
            { printf("exp ->  \n"); }
        |
                TRUE
            { printf("exp -> true \n"); }
        |
                FALSE
            { printf("exp -> false \n"); }
        |
                ID
            { printf("exp -> \n"); }
        |
                THIS
            { printf("exp -> this \n"); }
        |
                NEW INT LSP exp RSP
            { printf("exp -> new int lsp exp rsp \n"); }
        |
                NEW ID LP RP
            { printf("exp -> new id lp rp \n"); }
        |
                NOT exp
            { printf("exp -> Exp \n"); }
        ;

expls   :       exp exprest
            { printf("expls -> Exp ExpRest* \n"); }
        |
            { printf("expls -> \n"); }
        ;

exprest : COMMA exp exprest
            { printf("exprests -> comma Exp \n"); }
        |
            { printf("exprests -> \n"); }
        ;

%%

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

Rules Grammar rules

User routines helper functions or the main function

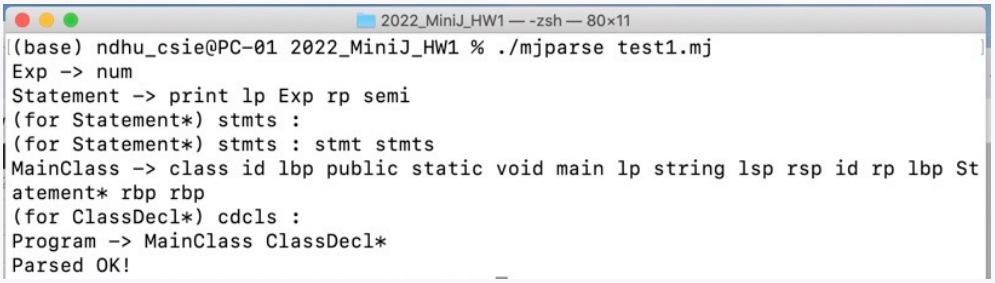
Problem Listing

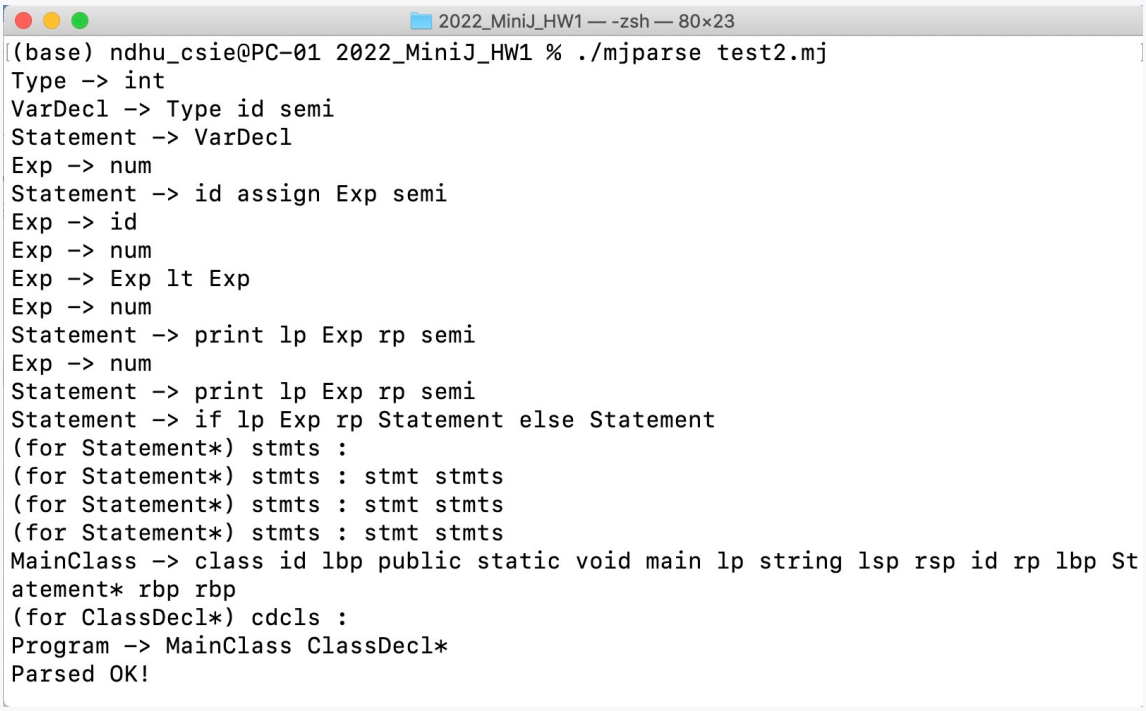
We did encounter lots of problems during this homework.

It was our very first time using flex and bison to buld a compiler. Although professor had already taught us and gave us detailed instructions, we still face problems. There are several problems. We listed the most significant three out as below. By solving these, we learned more indeed, not only those key concepts covered in this homework, and the skills to deal with problems.

Problem	Description	Solutions
Unfamiliar grammar rules	Program couldn't recognize and give out error messages.	• Check PPT notes and search examples online. • Rethink. • Rewrite.
Unable to debug	We spent the most time debuggung errors and reading error messages	• Trace everything again and again. • Get help from others.
Lack of common sense to computers	Couldn't execute cmd correctly at first	• Internet. • Ask others.

Test Run Results





Test Run Results

```
管理者: C:\WINDOWS\system32\cmd.exe
arse.exe test3.mj
exp -> new id lp rp
exp ->
exprests ->
expls -> Exp ExpRest*
exp -> id lp Explist rp
exp -> Exp dot Exp
stat -> print lp Exp rp semi
stats ->
stats -> Statement*
MainClass -> class id lbp public static void main lp string lsp rsp id rp lbp Statenet* rbp rbp
(for VarDecl*) vdcls :
Type -> INT
Type -> INT
FormalRest ->
FormalList -> Type id FormalRest*
Type -> INT
VarDecl -> Type id semi
(for VarDecl*) vdcls :
(for VarDecl*) vdcls : vdcl vdcls
exp ->
exp ->
exp -> Exp lt Exp
exp ->
stat -> id assign Exp rp semi
exp ->
exp -> this
exp ->
exp ->
exp -> Exp minus Exp
exprests ->
expls -> Exp ExpRest*
exp -> id lp Explist rp
exp -> Exp dot Exp
exp -> lp Exp rp
exp -> Exp times Exp
stat -> id assign Exp rp semi
stat -> if lp Exp rp Statement else Statement
stats ->
stats -> Statement*
exp ->
MethodDecl -> public Type id lp FormalList rp lbp Statements* return Exp semi rbp
(for MethodDecl*) mdcls :
(for MethodDecl*) mdcls : mdcl mdcls
ClassDecl -> class id lbp VarDecl* MethodDecl* rbp
(for ClassDecl*) cdcls :
(for ClassDecl*) cdcls : cdcl cdcls
Program -> MainClass ClassDecl*
Parsed OK!
```

Discussion

Here's the summation. To build an compiler, we could first construct a .l program and a .y program, which could be lexed by Flex and be parsed by Bison. Next, we could get the .c and .h files through Flex and Bison. Then, we could use GCC to let those .c and .h become .o programs. After several executions by using GCC, we would get the .exe program for compiling at the end.
Last but not least, write your code to try your own compiler and enjoy!!

To be honest,
I've never thought of
that I could deal with
more than one programs
at the same time.

I felt like
nothing is impossible
after I finished this project,
especially after
the loooong and hard time
debugging errors.

Really learn more than knowledge!