lab2 实验报告

学号：201907040102

姓名：李平凡

实验要求

1.了解 bison 基础知识和理解 Cminus-f 语法（重在了解如何将文法产生式转换为 bison 语句）

2.阅读 /src/common/SyntaxTree.c，对应头文件 /include/SyntaxTree.h（重在理解分析树如何生成）

3.了解 bison 与 flex 之间是如何协同工作，看懂 pass_node 函数并改写 Lab1 代码（提示：了解 yylval 是如何工作，在代码层面上如何将值传给$1、$2 等）

4.补全 src/parser/syntax_analyzer.y 文件和 lexical_analyzer.l 文件

实验难点

了解 bison 和 flex 是如何协同工作的以及分析树的生成。

实验设计

1.首先需要了解 yylex，flex 词法分析器返回时（比如 yylex），返回值就是一个记号（也就是 IF，ELSE 等等），这些记号在实验一中我们有一个自己的头文件，里面定义了各种记号，但真正的头文件实际上是在与 bison 联合使用时，在.y 文件中定义的 token 然后通过 linux 下命令借助 bison 生成的，有了这些记号，yylex 在我们输入一个表达式或者用文本从中输入，它就会自动按照.l 文件中定义的正则表达式，去匹配相应的 token，如果我们相应的在正则表达式后面加入了一些操作比如 return，yylex 就会返回这个值，并且在当前返回位置停下，接着去匹配下一个值。（举一个例子）

```
%{
    enum {
        NUMBER = 258,          //这里有我们自己定义的两个记号NUMBER和ADD
        ADD = 259,
    };
    int yylval = 0;           //在之后的bison生成的头文件中会使用到，暂时先自己定义
%}
%%
"+" { return ADD; }
[0-9]+ { yylval = atoi(yytext); return NUMBER; }              //两个正则表达式
%%
int main()
{
    int token;

    while((token = yylex()) != 0) {              //yylex会有返回值，返回值会返回给token
        printf("%d", token);                     //首先打印token的值
        if (token == NUMBER)
            printf(" = %d", yylval);             //接着打印的就是我们token的实际值（就是你的输入值）
        printf("\n");
    }
    return 0;
}
```

```
lpf@LAPTOP-P8EMT48L:/mnt/c/Users/86178/Desktop/cminus_compiler-2021-fall/src/parser$ flex calc.l
lpf@LAPTOP-P8EMT48L:/mnt/c/Users/86178/Desktop/cminus_compiler-2021-fall/src/parser$ cc lex.yy.c -lfl
^[[Alpf@LAPTOP-P8EMT48L:/mnt/c/Users/86178/Desktop/cminus_compiler-2021-fall/src/parser$ ./a.out
18+12
258 = 18
259
258 = 12
```

18+12 是我们自己在 linux 下的输入的表达式，18 匹配命中了第二个正则表达式，也就是 NUMBER，所以 yylex 返回给 token 一个值也就是 NUMBER 定义的 258（NUMBER 定义为 258 是因为在与 bison 联合使用时，标记编号的 0 值意味着文件结束；而为了避免与 ASCII 码和内定值冲突，其他的标记编号要从 258 开始定义。）yylval 在我们定义的正则表达式后面的操作里，表示我们会得到这个匹配的真实值，专业一点来讲，flex 的返回值是一个记号流，记号流的定义有两个属性，一个是记号编号（就是 NUMBER）还有一个是记号值（就是 18

这个值一般由 yylval 管理）。

结合本实验计算器案例的 lexical_analyzer.l，syntax_analyzer.y 我们分析 bison 与 flex 的联合怎么体现：

flex 在生成相应的记号流之后，记号的所有属性都需要保存在一个地方包括（编号和属性值），保存这些属性就是在%union 定义的一个结构中（flex 中的正则表达式一旦匹配成功你输入的关键字等等，需要你在正则表达式后进行返回编号并且还要将属性值保存在%union 中），我们保存的方法就是在正则表达式匹配成功之后，用 yylval 来进行更改和保存，yylval 是 YYSTYPE 类型的。

YYSTYPE 实际上就是 union。

.y 文件中的%union 会进行一个转换，转换如图所示

```
%union {
    char   op;
    double num;
}
```

```
union YYSTYPE
{
#line 15 "calc.y"

    char   op;
    double num;

#line 71 "calc.tab.h"

};
typedef union YYSTYPE YYSTYPE;
```

```
extern YYSTYPE yylval;
```

Bison 生成的头文件中看到的 yylval 定义

词法匹配成功之后就会进行语法的检查，判断你的输入的计算式满足哪一条，然后根据语法匹配成功的属性进行求值。（此处不放截图）

2.回到本实验结合 SyntaxTree.c 和 SyntaxTree.h 对 cminus 语法进行检查。我们的 flex 首先同样进行词法的匹配，一旦匹配成功，就会将他的属性用 yylval 进行保存，同样的需要返回记号的编号。

```
else {pos_start=pos_end;pos_end=pos_start+4;pass_node(yytext);return ELSE;}
```

```
void pass_node(char *text){
     yylval.node = new_syntax_tree_node(text);
}
%}
```

yylex 的工作就是对于匹配成功的词法为他们构建语法树的结点（并通过 yylval 传给 bison 使用），同时返回编号。

```
%union {
    syntax_tree_node *node;
    }
```

在 bison 中的 union 定义（我觉得也可以称为属性值的定义）可知是一个语法树结点

用 yylval 为我们匹配成功的 token 建立了一个结点，然后将编号返回给了 bison，等待 bison 的处理，bison 拿到输入的所有的记号（记号流之后）进行语法的匹配，以其中一条语法规则为例：

```
declaration-list
: declaration-list declaration
{
    $$=node("declaration-list",2,$1,$2);
}
```

假如匹配成功该语法规则，那么就会创建一个新父节点后面跟着一些子节点（declaration-list 和 declaration 这两个子节点（但在 node 函数中参数值用$1 和$2 表示））创建的方式就是 node 函数

```
syntax_tree_node *node(const char *name, int children_num, ...)
{
    syntax_tree_node *p = new_syntax_tree_node(name);
    syntax_tree_node *child;
    if (children_num == 0) {
        child = new_syntax_tree_node("epsilon");
        syntax_tree_add_child(p, child);
    } else {
        va_list ap;
        va_start(ap, children_num);
        for (int i = 0; i < children_num; ++i) {
            child = va_arg(ap, syntax_tree_node *);
            syntax_tree_add_child(p, child);
        }
        va_end(ap);
    }
    return p;
}
```

首先为当前（declaration-list）创建一个 p 结点，后面跟着他的孩子，由于我们不知道有多少个（但知道都属于 syntax_tree_node*类型）所以我们用 va_list ap 的方式遍历每一个孩子（ap 就相当于一个指针，通过一个 for 循环遍历每一个孩子）得到的每一个孩子都赋值给 child 结点，将 child 结点通过 syntax_tree_add_child 函数添加到以 p 父节点开头的这个语法树中。

```
int syntax_tree_add_child(syntax_tree_node * parent, syntax_tree_node * child);
```

上面是语法树.h 文件中该函数的声明，下面是语法树.c 文件中该函数的实现，首先判断父节点或者子节点是否为空，如果为空就返回-1（代表添加失败），否则（在语法树结点中定义有一个父节点，子节点最大是 10 个，通过数组来保存，并用 children_num 记录有多少个子节点（同时也当作了最后一个子节点的下标））就把孩子节点不断加入到父节点的分支中。

```
int syntax_tree_add_child(syntax_tree_node * parent, syntax_tree_node * child)
{
    if (!parent || !child)  return -1;
    parent->children[parent->children_num++] = child;
    return parent->children_num;
}
```

3.pass_node 是为了后期方便构造语法树而用到的一个函数，本质上是通过 yylval 保存结点属性（创建一个属于当前 token 的结点），$1 和$2 的分析同上。

4.语法的匹配来自于实验文档

1. $ \text{program} \rightarrow \text{declaration-list} $
2. $ \text{declaration-list} \rightarrow \text{declaration-list}\ \text{declaration}\ |\ \text{declaration} $
3. $ \text{declaration} \rightarrow \text{var-declaration}\ |\ \text{fun-declaration} $
4. $ \text{var-declaration}\ \rightarrow \text{type-specifier}\ \textbf{ID}\ \textbf{;}\ |\ \text{type-specifier}\ \textbf{ID}\ \textbf{[}\ \textbf{INTEGER}\ \textbf{]}\ \textbf{;} $
5. $ \text{type-specifier} \rightarrow \textbf{int}\ |\ \textbf{float}\ |\ \textbf{void} $
6. $ \text{fun-declaration} \rightarrow \text{type-specifier}\ \textbf{ID}\ \textbf{(}\ \text{params}\ \textbf{)}\ \text{compound-stmt} $
7. $ \text{params} \rightarrow \text{param-list}\ |\ \textbf{void} $
8. $ \text{param-list} \rightarrow \text{param-list}\ \textbf{,}\ \text{param}\ |\ \text{param} $
9. $ \text{param} \rightarrow \text{type-specifier}\ \textbf{ID}\ |\ \text{type-specifier}\ \textbf{ID}\ \textbf{[]} $
10. $ \text{compound-stmt} \rightarrow \textbf{\{}\ \text{local-declarations}\ \text{statement-list}\ \textbf{\}} $
11. $ \text{local-declarations} \rightarrow \text{local-declarations var-declaration}\ |\ \text{empty} $
12. $ \text{statement-list} \rightarrow \text{statement-list}\ \text{statement}\ |\ \text{empty} $
13. $ \begin{aligned}\text{statement} \rightarrow &\text{expression-stmt}\\ &|\ \text{compound-stmt}\\ &|\ \text{selection-stmt}\\ &|\ \text{iteration-stmt}\\ &|\ \text{return-stmt}\end{aligned} $
14. $ \text{expression-stmt} \rightarrow \text{expression}\ \textbf{;}\ |\ \textbf{;} $
15. $ \begin{aligned}\text{selection-stmt} \rightarrow &\textbf{if}\ \textbf{(}\ \text{expression}\ \textbf{)}\ \text{statement}\\ &|\ \textbf{if}\ \textbf{(}\ \text{expression}\ \textbf{)}\ \text{statement}\ \textbf{else}\ \text{statement}\end{aligned} $
16. $ \text{iteration-stmt} \rightarrow \textbf{while}\ \textbf{(}\ \text{expression}\ \textbf{)}\ \text{statement} $
17. $ \text{return-stmt} \rightarrow \textbf{return}\ \textbf{;}\ |\ \textbf{return}\ \text{expression}\ \textbf{;} $
18. $ \text{expression} \rightarrow \text{var}\ \textbf{=}\ \text{expression}\ |\ \text{simple-expression} $
19. $ \text{var} \rightarrow \textbf{ID}\ |\ \textbf{ID}\ \textbf{[}\ \text{expression}\ \textbf{]} $
20. $ \text{simple-expression} \rightarrow \text{additive-expression}\ \text{relop}\ \text{additive-expression}\ |\ \text{additive-expression} $
21. $ \text{relop} \rightarrow \textbf{<=}\ |\ \textbf{<}\ |\ \textbf{>}\ |\ \textbf{>=}\ |\ \textbf{==}\ |\ \textbf{!=} $
22. $ \text{additive-expression} \rightarrow \text{additive-expression}\ \text{addop}\ \text{term}\ |\ \text{term} $
23. $ \text{addop} \rightarrow \textbf{+}\ |\ \textbf{-} $
24. $ \text{term} \rightarrow \text{term}\ \text{mulop}\ \text{factor}\ |\ \text{factor} $
25. $ \text{mulop} \rightarrow \textbf{*}\ |\ \textbf{/} $
26. $ \text{factor} \rightarrow \textbf{(}\ \text{expression}\ \textbf{)}\ |\ \text{var}\ |\ \text{call}\ |\ \text{integer}\ |\ \text{float} $
27. $ \text{integer} \rightarrow \textbf{INTEGER} $
28. $ \text{float} \rightarrow \textbf{FLOATPOINT} $
29. $ \text{call} \rightarrow \textbf{ID}\ \textbf{(}\ \text{args}\ \textbf{)} $
30. $ \text{args} \rightarrow \text{arg-list}\ |\ \text{empty} $
31. $ \text{arg-list} \rightarrow \text{arg-list}\ \textbf{,}\ \text{expression}\ |\ \text{expression} $

根据实验文档写出语法规则（补充.y 文件）：

```
program
:declaration-list
{
    $$ = node("program", 1, $1); gt->root = $$;
}

declaration-list
: declaration-list declaration
{
    $$=node("declaration-list",2,$1,$2);
}
|declaration
{
    $$=node("declaration-list",1,$1);
}

declaration
:var-declaration
{
    $$=node("declaration",1,$1);
}
|fun-declaration
{
    $$=node("declaration",1,$1);
}

var-declaration
:type-specifier IDENTIFIER SEMICOLON
{
    $$=node("var-declaration",3,$1,$2,$3);
}
|type-specifier IDENTIFIER LBRACKET INTEGER RBRACKET SEMICOLON
{
    $$=node("var-declaration",6,$1,$2,$3,$4,$5,$6);
}

type-specifier
:INT {$$=node("type-specifier",1,$1);}
|FLOAT {$$=node("type-specifier",1,$1);}
|VOID {$$=node("type-specifier",1,$1);}

fun-declaration
: type-specifier IDENTIFIER LPARENTHESE params RPARENTHESE compound-stmt
{
    $$=node("fun-declaration",6,$1,$2,$3,$4,$5,$6);
}
```

```
params
:param-list
{
    $$=node("params",1,$1);
}
|VOID
{
    $$=node("params",1,$1);
}

param-list
:param-list COMMA param
{
    $$=node("param-list",3,$1,$2,$3);
}
|param
{
    $$=node("param-list",1,$1);
}

param
:type-specifier IDENTIFIER
{
    $$=node("param",2,$1,$2);
}
|type-specifier IDENTIFIER ARRAY
{
    $$=node("param",3,$1,$2,$3);
}

compound-stmt
:LBRACE local-declarations statement-list RBRACE
{
    $$=node("compound-stmt",4,$1,$2,$3,$4);
}

local-declarations
:local-declarations var-declaration
{
    $$=node("local-declarations",2,$1,$2);
}
|{
    $$=node("local-declarations",0);
}
```

```
statement-list
:statement-list statement
{
    $$=node("statement-list",2,$1,$2);
}
|{
    $$=node("statement-list",0);
}

statement
:expression-stmt {$$=node("statement",1,$1);}
|compound-stmt {$$=node("statement",1,$1);}
|selection-stmt {$$=node("statement",1,$1);}
|iteration-stmt {$$=node("statement",1,$1);}
|return-stmt {$$=node("statement",1,$1);}

expression-stmt
:expression SEMICOLON
{
    $$=node("expression-stmt",2,$1,$2);
}
|SEMICOLON
{
    $$=node("expression-stmt",1,$1);
}

selection-stmt
:IF LPARENTHESE expression RPARENTHESE statement
{
    $$=node("selection-stmt",5,$1,$2,$3,$4,$5);
}
|IF LPARENTHESE expression RPARENTHESE statement ELSE statement
{
    $$=node("selection-stmt",7,$1,$2,$3,$4,$5,$6,$7);
}

iteration-stmt
:WHILE LPARENTHESE expression RPARENTHESE statement
{
    $$=node("iteration-stmt",5,$1,$2,$3,$4,$5);
}

return-stmt
:RETURN SEMICOLON
{
    $$=node("return-stmt",2,$1,$2);
}
|RETURN expression SEMICOLON
{
    $$=node("return-stmt",3,$1,$2,$3);
}

expression
:var ASSIN expression
{
    $$=node("expression",3,$1,$2,$3);
}
|simple-expression
{
    $$=node("expression",1,$1);
}

var
:IDENTIFIER
{
    $$=node("var",1,$1);
}
|IDENTIFIER LBRACKET expression RBRACKET
{
    $$=node("var",4,$1,$2,$3,$4);
}

simple-expression
:additive-expression relop additive-expression
{
    $$=node("simple-expression",3,$1,$2,$3);
}
|additive-expression
{
    $$=node("simple-expression",1,$1);
}

relop
:LTE {$$=node("relop",1,$1);}
|LT {$$=node("relop",1,$1);}
|GT {$$=node("relop",1,$1);}
|GTE {$$=node("relop",1,$1);}
|EQ {$$=node("relop",1,$1);}
|NEQ {$$=node("relop",1,$1);}

additive-expression
:additive-expression addop term
{
    $$=node("additive-expression",3,$1,$2,$3);
}
|term
{
    $$=node("additive-expression",1,$1);
}

addop
:ADD {$$=node("addop",1,$1);}
|SUB {$$=node("addop",1,$1);}

term
:term mulop factor
{
    $$=node("term",3,$1,$2,$3);
}
|factor
{
    $$=node("term",1,$1);
}

mulop
:MUL {$$=node("mulop",1,$1);}
|DIV {$$=node("mulop",1,$1);}

factor
:LPARENTHESE expression RPARENTHESE
{
    $$=node("factor",3,$1,$2,$3);
}
|var {$$=node("factor",1,$1);}
|call {$$=node("factor",1,$1);}
|integer {$$=node("factor",1,$1);}
|float {$$=node("factor",1,$1);}

integer
:INTEGER {$$=node("integer",1,$1);}

float
:FLOATPOINT {$$=node("float",1,$1);}

call
:IDENTIFIER LPARENTHESE args RPARENTHESE
{$$=node("call",4,$1,$2,$3,$4);}

args
:arg-list {$$=node("args",1,$1);}
|{$$=node("args",0);}

arg-list
:arg-list COMMA expression
{
    $$=node("arg-list",3,$1,$2,$3);
}
|expression
{
    $$=node("arg-list",1,$1);
}
```

.l 文件仍然采用实验一的正则表达式的词法匹配但是需要增加 yylval 的操作，需要把属性保留给 yylval：

```
%%
\n {lines+=1;pos_end=1;}
\t {pos_start=pos_end;pos_end=pos_start+1;}
" " {pos_start=pos_end;pos_end=pos_start+1;}
\r {pos_start=pos_end;pos_end=pos_start+1;}
\s {pos_start=pos_end;pos_end=pos_start+1;}
else {pos_start=pos_end;pos_end=pos_start+4;pass_node(yytext);return ELSE;}
if {pos_start=pos_end;pos_end=pos_start+2;pass_node(yytext);return IF;}
int {pos_start=pos_end;pos_end=pos_start+3;pass_node(yytext);return INT;}
return {pos_start=pos_end;pos_end=pos_start+6;pass_node(yytext);return RETURN;}
void {pos_start=pos_end;pos_end=pos_start+4;pass_node(yytext);return VOID;}
while {pos_start=pos_end;pos_end=pos_start+5;pass_node(yytext);return WHILE;}
float {pos_start=pos_end;pos_end=pos_start+5;pass_node(yytext);return FLOAT;}
"[]" {pos_start=pos_end;pos_end=pos_start+2;pass_node(yytext);return ARRAY;}
\+ {pos_start=pos_end;pos_end=pos_start+1;pass_node(yytext);return ADD;}
\- {pos_start=pos_end;pos_end=pos_start+1;pass_node(yytext);return SUB;}
\* {pos_start=pos_end;pos_end=pos_start+1;pass_node(yytext);return MUL;}
\/ {pos_start=pos_end;pos_end=pos_start+1;pass_node(yytext);return DIV;}
\< {pos_start=pos_end;pos_end=pos_start+1;pass_node(yytext);return LT;}
"<=" {pos_start=pos_end;pos_end=pos_start+2;pass_node(yytext);return LTE;}
\> {pos_start=pos_end;pos_end=pos_start+1;pass_node(yytext);return GT;}
">=" {pos_start=pos_end;pos_end=pos_start+2;pass_node(yytext);return GTE;}
"==" {pos_start=pos_end;pos_end=pos_start+2;pass_node(yytext);return EQ;}
"!=" {pos_start=pos_end;pos_end=pos_start+2;pass_node(yytext);return NEQ;}
\= {pos_start=pos_end;pos_end=pos_start+1;pass_node(yytext);return ASSIN;}
\; {pos_start=pos_end;pos_end=pos_start+1;pass_node(yytext);return SEMICOLON;}
\, {pos_start=pos_end;pos_end=pos_start+1;pass_node(yytext);return COMMA;}
\( {pos_start=pos_end;pos_end=pos_start+1;pass_node(yytext);return LPARENTHESE;}
\) {pos_start=pos_end;pos_end=pos_start+1;pass_node(yytext);return RPARENTHESE;}
\[ {pos_start=pos_end;pos_end=pos_start+1;pass_node(yytext);return LBRACKET;}
\] {pos_start=pos_end;pos_end=pos_start+1;pass_node(yytext);return RBRACKET;}
\{ {pos_start=pos_end;pos_end=pos_start+1;pass_node(yytext);return LBRACE;}
\} {pos_start=pos_end;pos_end=pos_start+1;pass_node(yytext);return RBRACE;}
[a-zA-Z]+ {pos_start=pos_end;pos_end=pos_start+strlen(yytext);pass_node(yytext);return IDENTIFIER;}
[0-9]+ {pos_start=pos_end;pos_end=pos_start+strlen(yytext);pass_node(yytext);return INTEGER;}
[0-9]*\.[0-9]+ {pos_start=pos_end;pos_end=pos_start+strlen(yytext);pass_node(yytext);return FLOATPOINT;}
[0-9]+\. {pos_start=pos_end;pos_end=pos_start+strlen(yytext);pass_node(yytext);return FLOATPOINT;}
```

```
\/\*\/*([^\/]*[^\/\*]\/[\/]*)*[^\/]*\*\/ {
                pos_start=pos_end;
              pos_end=pos_start+2;
              int i=2;
                  while(yytext[i]!='*' || yytext[i+1]!='/')
                  {
                       if(yytext[i]=='\n')
                       {
                            lines=lines+1;
                            pos_end=1;
                       }
                       else
                            pos_end=pos_end+1;
                       i=i+1;
                  }
                  pos_end=pos_end+2;
                  }
. {pos_start=pos_end;pos_end=pos_start+strlen(yytext);return ERROR;}

%%
```

实验结果验证

```
lpf@LAPTOP-P8EMT48L:/mnt/c/Users/86178/Desktop/cminus_compiler-2021-fall$ diff ./tests/lab2/syntree_easy ./tests/lab2/syntree_easy_std -qr
lpf@LAPTOP-P8EMT48L:/mnt/c/Users/86178/Desktop/cminus_compiler-2021-fall$ diff ./tests/lab2/syntree_easy ./tests/lab2/syntree_easy_std
lpf@LAPTOP-P8EMT48L:/mnt/c/Users/86178/Desktop/cminus_compiler-2021-fall$ diff ./tests/lab2/syntree_normal ./tests/lab2/syntree_normal_std
lpf@LAPTOP-P8EMT48L:/mnt/c/Users/86178/Desktop/cminus_compiler-2021-fall$ diff ./tests/lab2/syntree_normal ./tests/lab2/syntree_normal_std -qr
```

刚开始验证发现所有样例都是错误的，于是发现 sublime 默认换行 line_ending 是 windows 下的所以不一样，后来在 sublime 里面修改设置，最终实验结果验证成功。

请提供部分自己的测试样例

```
int main(void){int attay[10];}
>--+ program
  >--+ declaration-list
    >--+ declaration
      >--+ fun-declaration
        >--+ type-specifier
        |  >--* int
        >--* main
        >--* (
        >--+ params
        |  >--* void
        >--* )
        >--+ compound-stmt
        |  >--* {
        |  >--+ local-declarations
        |  |  >--+ local-declarations
        |  |  |  >--* epsilon
        |  |  >--+ var-declaration
        |  |  |  >--+ type-specifier
        |  |  |  |  >--* int
        |  |  |  >--* attay
        |  |  |  >--* [
        |  |  |  >--* 10
        |  |  |  >--* ]
        |  |  |  >--* ;
        |  |  >--+ statement-list
        |  |  |  >--* epsilon
        |  |  >--* }
```

```
lpf@LAPTOP-P8EMT48L:/mnt/c/Users/86178/Desktop/cminus_compiler-2021-fall$ ./build/parser
int main(void){int array[];}
error at line 1 column 25: syntax error
```

```
lpf@LAPTOP-P8EMT48L:/mnt/c/Users/86178/Desktop/cminus_compiler-2021-fall$ ./build/parser
int gcd(int u,int v){return 1;}
int main(void){int x;int y;  gcd(x,y);}
```

```
lpf@LAPTOP-P8EMT48L:/mnt/c/Users/86178/Desktop/cminus_compiler-2021-fall$ ./build/parser
int main(void){}
void main(void){}
int main(void){}
int main(){}
error at line 4 column 10: syntax error
```

发现可以定义多个主函数（但多个主函数明显是错误的），但是每个主函数的参数不能少。

实验反馈

本次实验我了解到了 bison 和 flex 之间的联系，开始的时候我并不知道 bison 和 flex 之间如何联系，但后来明白 flex 是分析每一个 token 是否合乎规范，同时保留每个合乎规范的 token 的编号和值，然后传给 bison 进行处理，bison 会分析这些 token 组合起来是否是一个合乎规范的语法，也可以说是语句，但语句不一定有逻辑。同时利用 flex 传过来的属性值进行语法树的构建，或者做一个计算器。（不禁让我再次联想到上课讲到的词法分析就是"羊"，"老虎"，"吃"。语法分析就是分析"羊吃老虎","老虎吃羊"符合语法规范和词法规范。传给 bison 的属性值就是使得"羊吃老虎"或者"老虎吃羊"构建出了一个分析树或者语法树（文件中的英文名这样写的））这是本次实验最大的收获（加强了对词法分析和语法分析的理解）。Cminu 语法中有一个地方还不是太懂，好像并未用到，附上截图：

15. $ \begin{aligned}\te

思考题：

1. 在 1.3 样例代码中存在左递归文法，为什么 bison 可以处理？（提示：不用研究 bison 内部运作机制，在下面知识介绍中有提到 bison 的一种属性，请结合课内知识思考）

Bison 通过 LALR 文法自底向上规约语法规则构造出语法树，不用避免左递归，但是如果是自顶向下的语法分析的算法就需要避免左递归。

2. 请在代码层面上简述下 yylval 是怎么完成协同工作的。（提示：无需研究原理，只分析维护了什么数据结构，该数据结构是怎么和$1、$2 等联系起来？）

词法分析匹配相应的 token，将 token 的编号返回给 bison，bison 通过符号栈将其存储，同时，每个符号的属性值（比如$1 和$2）通过传给 yylval 进行更改，然后放入到另一个栈中，

也就是属性栈。

3. 请尝试使用 1.3 样例代码运行除法运算除数为 0 的例子（测试 case 中有）看下是否可以通过，如果不，为什么我们在 case 中把该例子认为是合法的？（请从语法与语义上简单思考）

```
lpf@LAPTOP-P8EMT48L:/mnt/c/Users/86178/Desktop/cminus_compiler-2021-fall/src/parser$ ./calc
3/0
 = inf
```

可以通过，结果是无穷大，从语法上是符合的，我们可以找到相应的语法规则来规约这一条表达式，这个表达式相当于“羊吃老虎”，语法分析只会分析符合除法的形式，但不会对于这条表达式的含义进行分析是否正确，分析含义是语义分析的任务，它来判断除 0 是一个错误的表达式。

4. 能否尝试修改下 1.3 计算器文法，使得它支持除数 0 规避功能。

文法方面我觉得我并没有修改，但我可能是从程序角度思考，我将%union 中的 num，double 类型改成了 int 类型，便支持了除数 0 规避功能，结果如下：

```
%union {
    char   op;
    int num;
}
```

```
lpf@LAPTOP-P8EMT48L:/mnt/c/Users/86178/Desktop/cminus_compiler-2021-fall/src/parser$ bison -d calc.y
lpf@LAPTOP-P8EMT48L:/mnt/c/Users/86178/Desktop/cminus_compiler-2021-fall/src/parser$ flex calc.l
lpf@LAPTOP-P8EMT48L:/mnt/c/Users/86178/Desktop/cminus_compiler-2021-fall/src/parser$ gcc lex.yy.c calc.tab.c driver.c -o calc
lpf@LAPTOP-P8EMT48L:/mnt/c/Users/86178/Desktop/cminus_compiler-2021-fall/src/parser$ ./calc
3/0
Floating point exception
```