

# 编译原理实验2

## Bison

---

计科210X 甘晴void 202108010XXX

### 实验要求

详细的实验项目文档为 [https://gitee.com/coderwym/cminus\\_compiler-2023-fall/tree/master/Documentations/lab2](https://gitee.com/coderwym/cminus_compiler-2023-fall/tree/master/Documentations/lab2)

### 实验步骤

本次实验需要在 Lab1 已完成的 **flex** 词法分析器的基础上，进一步使用 **bison** 完成语法分析器。

- 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

**Tips:** 在未编译的代码文件中是无法看到关于协同工作部分的代码，建议先编译 1.3 给出的计算器样例代码，再阅读 `/build/src/parser/` 中的 `syntax_analyzer.h` 与 `syntax_analyzer.c` 文件

### 思考题

本部分不算做实验分，出题的本意在于想要帮助同学们加深对实验细节的理解，欢迎有兴趣和余力的同学在报告中写下你的思考答案，或者在 `issue` 中分享出你的看法。

- 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）
- 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？）
- 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）
- 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。

## 1.Cminus-f语法

①Cminus-f 的所有规则分为五类

### 1. 字面量、关键字、运算符与标识符

- `id`
- `type-specifier`
- `relop`
- `addop`
- `mulop`

### 2. 声明

- `declaration-list`
- `declaration`
- `var-declaration`
- `fun-declaration`
- `local-declarations`

### 3. 语句

- `compound-stmt`
- `statement-list`
- `statement`
- `expression-stmt`
- `iteration-stmt`
- `selection-stmt`
- `return-stmt`

### 4. 表达式

- `expression`
- `var`
- `additive-expression`
- `term`
- `factor`
- `integer`
- `float`
- `call`

### 5. 其他

- `params`

- param-list
- param
- args
- arg-list

起始符号是 `program`。

## ②Cminus-f语法

```

 $\text{program} \rightarrow \text{declaration-list}$ 
 $\text{declaration-list} \rightarrow \text{declaration-list} \mid \text{declaration}$ 
 $\text{declaration} \rightarrow \text{var-declaration} \mid \text{fun-declaration}$ 
 $\text{var-declaration} \rightarrow \text{type-specifier} \mid \text{ID} \mid \text{type-specifier} \mid \text{ID} \mid \text{INTEGER} \mid \text{void}$ 
 $\text{type-specifier} \rightarrow \text{int} \mid \text{float} \mid \text{void}$ 
 $\text{fun-declaration} \rightarrow \text{type-specifier} \mid \text{ID} \mid \text{params} \mid \text{compound-stmt}$ 
 $\text{params} \rightarrow \text{param-list} \mid \text{void}$ 
 $\text{param-list} \rightarrow \text{param-list} \mid \text{param}$ 
 $\text{param} \rightarrow \text{type-specifier} \mid \text{ID} \mid \text{type-specifier} \mid \text{ID} \mid \text{void}$ 
 $\text{compound-stmt} \rightarrow \text{local-declarations} \mid \text{statement-list}$ 
 $\text{local-declarations} \rightarrow \text{local-declarations} \mid \text{var-declaration} \mid \text{empty}$ 
 $\text{statement-list} \rightarrow \text{statement-list} \mid \text{statement} \mid \text{empty}$ 
 $\text{statement} \rightarrow \text{expression-stmt} \mid \text{compound-stmt} \mid \text{selection-stmt} \mid \text{iteration-stmt} \mid \text{return-stmt}$ 
 $\text{expression-stmt} \rightarrow \text{expression} \mid \text{void}$ 
 $\text{selection-stmt} \rightarrow \text{if} \mid \text{if} \mid \text{expression} \mid \text{statement} \mid \text{if} \mid \text{expression} \mid \text{statement} \mid \text{else} \mid \text{statement}$ 

```

```

$\text{iteration-stmt} \rightarrow \textbf{while} \{ \} \backslash \text{expression} \backslash \textbf{;} \backslash \text{statement} \$
$\text{return-stmt} \rightarrow \textbf{return} \{ \} \backslash \textbf{;} \backslash \backslash \textbf{return} \backslash \text{expression} \backslash \textbf{;} \$
$\text{expression} \rightarrow \text{var} \backslash \textbf{=} \backslash \text{expression} \backslash \backslash \text{simple-expression} \$
$\text{var} \rightarrow \textbf{ID} \backslash \backslash \textbf{ID} \backslash \textbf{[} \backslash \text{expression} \backslash \textbf{]} \$
$\text{simple-expression} \rightarrow \text{additive-expression} \backslash \text{relop} \backslash \text{additive-expression} \backslash \backslash \text{additive-expression} \$
$\text{relop} \rightarrow \textbf{<=} \backslash \backslash \textbf{<} \backslash \backslash \textbf{>} \backslash \backslash \textbf{>=} \backslash \backslash \textbf{==} \backslash \backslash \textbf{!=} \$
$\text{additive-expression} \rightarrow \text{additive-expression} \backslash \text{addop} \backslash \text{term} \backslash \backslash \text{term} \$
$\text{addop} \rightarrow \textbf{+} \backslash \backslash \textbf{-} \$
$\text{term} \rightarrow \text{term} \backslash \text{mulop} \backslash \text{factor} \backslash \backslash \text{factor} \$
$\text{mulop} \rightarrow \textbf{*} \backslash \backslash \textbf{/} \$
$\text{factor} \rightarrow \textbf{(} \backslash \text{expression} \backslash \textbf{)} \backslash \backslash \text{var} \backslash \backslash \text{call} \backslash \backslash \text{integer} \backslash \backslash \text{float} \$
$\text{integer} \rightarrow \textbf{INTEGER} \$
$\text{float} \rightarrow \textbf{FLOATPOINT} \$
$\text{call} \rightarrow \textbf{ID} \backslash \textbf{(} \backslash \text{args} \backslash \textbf{)} \$
$\text{args} \rightarrow \text{arg-list} \backslash \backslash \text{empty} \$
$\text{arg-list} \rightarrow \text{arg-list} \backslash \textbf{,} \backslash \text{expression} \backslash \backslash \text{expression} \$

```

## 2.Bison基础知识

Bison 是一款解析器生成器（parser generator），它的作用是将 LALR 文法转换成可编译的 C 代码。

实验文档给出了如下的示例代码，简要示范了Bison是怎么工作的。

```

%{
#include <stdio.h>

/* 这里是序曲 */
/* 这部分代码会被原样拷贝到生成的 .c 文件的开头 */

```

```

int yylex(void);
void yyerror(const char *s);
%}

/* 这些地方可以输入一些 bison 指令 */
/* 比如用 %start 指令指定起始符号, 用 %token 定义一个 token */
%start reimu
%token REIMU

%%

/* 从这里开始, 下面是解析规则 */
reimu : marisa { /* 这里写与该规则对应的处理代码 */ puts("rule1"); }
      | REIMU   { /* 这里写与该规则对应的处理代码 */ puts("rule2"); }
      ; /* 规则最后不要忘了用分号结束哦~ */

/* 这种写法表示  $\epsilon$  -- 空输入 */
marisa : { puts("Hello!"); }

%%

/* 这里是尾声 */
/* 这部分代码会被原样拷贝到生成的 .c 文件的末尾 */

int yylex(void)
{
    // 获取下一个待分析token
}

void yyerror(const char *s)
{
    // 报错处理机制
}

//main函数不一定在.y中, 可以通过链接实现
int main(void)
{
    yyparse(); // 启动解析
    return 0;
}

```

总结可知，Bison与Lex文件的编写规则类似，由%%区分的三部分构成，开头和结尾会被直接加入.c文件。

### 3.了解树的生成过程

阅读 /src/common/SyntaxTree.c，对应头文件 /include/SyntaxTree.h

首先看SyntaxTree.h，这是关于语法分析树以及相关操作的定义

```
#ifndef __SYNTAXTREE_H__
#define __SYNTAXTREE_H__

#include <stdio.h>

#define SYNTAX_TREE_NODE_NAME_MAX 30

// 这是语法分析树的节点
struct _syntax_tree_node {
    struct _syntax_tree_node * parent;
    struct _syntax_tree_node * children[10];
    int children_num;

    char name[SYNTAX_TREE_NODE_NAME_MAX];
};
typedef struct _syntax_tree_node syntax_tree_node;

// 下面是对于语法分析树进行操作的函数的定义
// 例如删除节点，增加节点，删除树，打印树
syntax_tree_node * new_anon_syntax_tree_node();
syntax_tree_node * new_syntax_tree_node(const char * name);
int syntax_tree_add_child(syntax_tree_node * parent,
syntax_tree_node * child);
void del_syntax_tree_node(syntax_tree_node * node, int recursive);

struct _syntax_tree {
    syntax_tree_node * root;
};
typedef struct _syntax_tree syntax_tree;

syntax_tree* new_syntax_tree();
void del_syntax_tree(syntax_tree * tree);
```

```
void print_syntax_tree(FILE * fout, syntax_tree * tree);

#endif /* SyntaxTree.h */
```

而SyntaxTree.c中是对应的具体实现，就不看了。

每个终结符都对应着一个叶子节点，这个叶子节点在词法分析时就可以产生。在自底向上的分析过程中，首先产生的是叶子节点，在用产生式进行规约时向上构建语法分析树。叶子节点的产生在词法分析器中的`pass_node()`函数中实现，创建一个新的节点，并将其指针赋值给`yylval`，节点名为其成分(非终结符名或终结符名)，这样语法分析器就可以使用该节点构造语法分析树。

## 4.Bison和Flex的协同工作

在语法分析过程中，语法分析树的叶子节点是一个具体的语义值，该值的类型是`YYSTYPE`，在Bison中用`%union`指明。不同的节点对应着不同的终结符，可能为不同的类型，因此`union`中可以包含不同的数据类型。可以指明一个终结符或是非终结符的类型，以便后续的使用。可以使用`%type <>`或`%token <>`指明类型。其中`%token`是在声明词法单元名的同时指明类型，声明的`token`会由Bison导出到最终的.h文件中，让词法分析器也可以使用。

参考文档给出了一个计算器样例的实现代码，并对新出现的构造进行解释。

### ①YYSTYPE

在 `bison` 解析过程中，每个 `symbol` 最终都对应到一个语义值上。或者说，在 `parse tree` 上，每个节点都对应一个语义值，这个值的类型是 `YYSTYPE`。`YYSTYPE` 的具体内容是由 `%union` 构造指出的。例如：

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

会生成这样的代码

```
typedef union YYSTYPE {
    char op;
    double num;
} YYSTYPE;
```

使用union是为了让不同的类型读取同一片存储空间。因为不同节点可能需要不同类型的语义值。比如，上面的例子中，我们希望 `ADDOP` 的值是 `char` 类型，而 `NUMBER` 应该是 `double` 类型的。

## ②规约

```
term : term ADDOP factor
    {
        switch $2 {
            case '+': $$ = $1 + $3; break;
            case '-': $$ = $1 - $3; break;
        }
    }
```

前节点使用 `$$` 代表，而已解析的节点则是从左到右依次编号，称作 `$1`, `$2`, `$3`...

## ③%type <>

bison如何确定对于union的部分应该取哪个值？文件开始的`%type`和`%token`给出了定义。例如`term` 应该使用 `num` 部分，那么我们就写

```
%type <num> term
```

遇到`term`时，bison就会去取其`num`。

## ④%token

当我们用 `%token` 声明一个 `token` 时，这个 `token` 就会导出到 `.h` 中，可以在 C 代码中直接使用（注意 `token` 名千万不要和别的东西冲突！），供 `flex` 使用。`%token <op> ADDOP` 与之类似，但顺便也将 `ADDOP` 传递给 `%type`。

## ⑤yylval



这时候我们可以打开 `.h` 文件，看看里面有什么。除了 `token` 定义，最末尾还有一个 `extern YYSTYPE yylval;`。这个变量我们上面已经使用了，通过这个变量，我们就可以在 `lexer` 里面设置某个 `token` 的值。

## 实验原理

由 `main.c` 下调用 `parse` 函数（在 `syntax_analyzer.y`）从 `input_path` 中读取待处理的字符，使用 `syntax_analyzer.l` 中定义的词法处理规则处理 `token`，更新 `pos` 和 `lines`，将该 `token` 的 `yytext` 使用 `pass_node` 函数传递给 `yylval`，并返回该 `token` 的类型。`yylval` 负责将该 `token` 传递，并建立语法分析树。

## 实验过程

### 1.词法分析部分

目标对象： `./src/parser/lexical_analyzer.l`

#### ①void analyzer()函数

在 `Lab1` 中是因为 `main.c` 函数调用了 `void analyzer(char* input_file, Token_Node* token_stream)` 这个函数，而现在 `Lab2` 中的 `main.c` 并没有使用该函数，故这个函数可以删去，但原来在这个函数中实现的功能需要迁移。

#### ②有关others的处理（`\n`,注释，`\t`,"等）

在 `Lab1` 中也需要对于注释这些进行词法分析，并输出这些所在的位置，而在 `Lab2` 中占主导的是 `Bison`，`Lex` 主要干的事情只是辅助识别出 `token` 并将 `token` 转交给 `Bison` 构建语法分析树，故 `others` 其实并不需要返回，即识别到这些的这些东西只需要直接忽略就可以了（只是更新 `lines` 与 `pos`），故这里需要删除对于 `others` 的 `return` 操作。

包括 `ERROR` 的处理，现在也是放在 `syntax_analyzer` 中的 `yyerror` 里，故这里的 `ERROR` 也可以删去 `return` 操作。

#### ③添加pass\_node(yytext)

在识别动作中要添加 `pass_node(yytext)` 产生词法单元叶子节点，

识别出来的正常 `token` 通过 `yylval` 传递给语法分析器

%%

```

/* 运算 */
\+  {pos_start = pos_end; pos_end++; pass_node(yytext); return
ADD;}
\-  {pos_start = pos_end; pos_end++; pass_node(yytext); return
SUB;}
\*  {pos_start = pos_end; pos_end++; pass_node(yytext); return
MUL;}
\/  {pos_start = pos_end; pos_end++; pass_node(yytext); return
DIV;}
\<  {pos_start = pos_end; pos_end++; pass_node(yytext); return
LT;}
"<=" {pos_start = pos_end; pos_end+=2; pass_node(yytext); return
LTE;}
\>  {pos_start = pos_end; pos_end++; pass_node(yytext); return
GT;}
">=" {pos_start = pos_end; pos_end+=2; pass_node(yytext); return
GTE;}
"==" {pos_start = pos_end; pos_end+=2; pass_node(yytext); return
EQ;}
"!=" {pos_start = pos_end; pos_end+=2; pass_node(yytext); return
NEQ;}
\=  {pos_start = pos_end; pos_end++; pass_node(yytext); return
ASSIN;}

/* 符号 */
\;  {pos_start = pos_end; pos_end++; pass_node(yytext); return
SEMICOLON;}
\,  {pos_start = pos_end; pos_end++; pass_node(yytext); return
COMMA;}
\(  {pos_start = pos_end; pos_end++; pass_node(yytext); return
LPARENTHESIS;}
\)  {pos_start = pos_end; pos_end++; pass_node(yytext); return
RPARENTHESIS;}
\[  {pos_start = pos_end; pos_end++; pass_node(yytext); return
LBRACKET;}
\]  {pos_start = pos_end; pos_end++; pass_node(yytext); return
RBRACKET;}
\{  {pos_start = pos_end; pos_end++; pass_node(yytext); return
LBRACE;}
\}  {pos_start = pos_end; pos_end++; pass_node(yytext); return
RBRACE;}

```

```

/* 关键字 */
else {pos_start = pos_end; pos_end+=4; pass_node(yytext); return
ELSE;}
if {pos_start = pos_end; pos_end+=2; pass_node(yytext); return
IF;}
int {pos_start = pos_end; pos_end+=3; pass_node(yytext); return
INT;}
float {pos_start = pos_end; pos_end+=5; pass_node(yytext); return
FLOAT;}
return {pos_start = pos_end; pos_end+=6; pass_node(yytext); return
RETURN;}
void {pos_start = pos_end; pos_end+=4; pass_node(yytext); return
VOID;}
while {pos_start = pos_end; pos_end+=5; pass_node(yytext); return
WHILE;}

/* ID & NUM */
[a-zA-Z]+ {pos_start = pos_end; pos_end+=yyleng; pass_node(yytext);
return IDENTIFIER;}
[0-9]+ {pos_start = pos_end; pos_end+=yyleng; pass_node(yytext);
return INTEGER;}
[0-9]+\.[0-9]+ {pos_start = pos_end; pos_end+=yyleng;
pass_node(yytext); return FLOATPOINT;}
"[]" {pos_start = pos_end; pos_end+=2; pass_node(yytext); return
ARRAY;}
[a-zA-Z] {pos_start = pos_end; pos_end++; pass_node(yytext);
return LETTER;}

/* others */
// 换行操作本来是在void analyzer(char* input_file, Token_Node*
token_stream)这个函数内实现的，但是由于在Lab2该函数被删去，故需要挪至这里直接实
现。
\n {lines++; pos_end=1;}
\\\[^\*]*\*+([^\*][^\*]*\*+)*\\[ {
    for (int i=0;i<yyleng;i++){
        if (yytext[i]=='\n'){ /*换行操作*/
            lines++;
            pos_end=1;
        }
        else pos_end++;
    }
}
" " {pos_start = pos_end; pos_end+=yyleng;}

```

```
\t {pos_start = pos_end; pos_end+=yylen;}  
. {}
```

## 2. 语法分析部分

目标对象：./src/parser/lexical\_analyzer.l

### ① 完成yylval的定义

在union中只含有一个节点指针。

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

### ② 终结符(词法单元)的声明和非终结符的类型声明

它们的类型都是语法分析树的节点指针，其中终结符名要和词法分析部分中的token一致，非终结符名和Cminus-f的语法规则中一致。声明如下：

```
%start program  
%token <node> ADD SUB MUL DIV  
%token <node> LT LTE GT GTE EQ NEQ ASSIN  
%token <node> SEMICOLON COMMA LPARENTHESIS RPARENTHESIS LBRACKET  
RBRACKET LBRACE RBRACE  
%token <node> ELSE IF INT FLOAT RETURN VOID WHILE IDENTIFIER LETTER  
INTEGER FLOATPOINT ARRAY  
%type <node> type-specifier relop addop mulop  
%type <node> declaration-list declaration var-declaration fun-  
declaration local-declarations  
%type <node> compound-stmt statement-list statement expression-stmt  
iteration-stmt selection-stmt return-stmt  
%type <node> simple-expression expression var additive-expression  
term factor integer float call  
%type <node> params param-list param args arg-list program
```

### ③ 补充语法规则

规则按照给出的Cminus-f的语法编写，动作则是调用node()函数构造语法分析树的节点，参数为子节点个数和使用\$n表示的子节点的指针，当产生式为空输入时，参数为0，子节点为空串。

```
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 { $$ = node("type-specifier", 1, $1); }
               ;
fun-declaration : type-specifier IDENTIFIER LPARENTHESIS params
RPARENTHESIS compound-stmt { $$ = 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 {
$$ = node("compound-stmt", 4, $1, $2, $3, $4); } ;
local-declarations : { $$ = node("local-declarations", 0); }
```

```

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

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

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

        ;
iteration-stmt : WHILE LPARENTHESIS expression RPARENTHESIS 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 { $$ = node("mulop", 1, $1); }
    ;
factor : LPARENTHESIS expression RPARENTHESIS { $$ = 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 LPARENTHESIS args RPARENTHESIS { $$ = node("call",
4, $1, $2, $3, $4); } ;
args : { $$ = node("args", 0); }
    | arg-list { $$ = node("args", 1, $1); }
    ;
arg-list : arg-list COMMA expression { $$ = node("arg-list", 3, $1,
$2, $3); }
    | expression { $$ = node("arg-list", 1, $1); }
    ;

```

完成了以上的补充后，语法分析和词法分析就应该都可以正常进行了。尝试编译时提示缺少yyin的声明，在语法分析函数parse中使用了yyin来进行读入，yyin是词法分析Flex产生的变量，这里需要引入，因此在开头补充引入该文件指针变量。

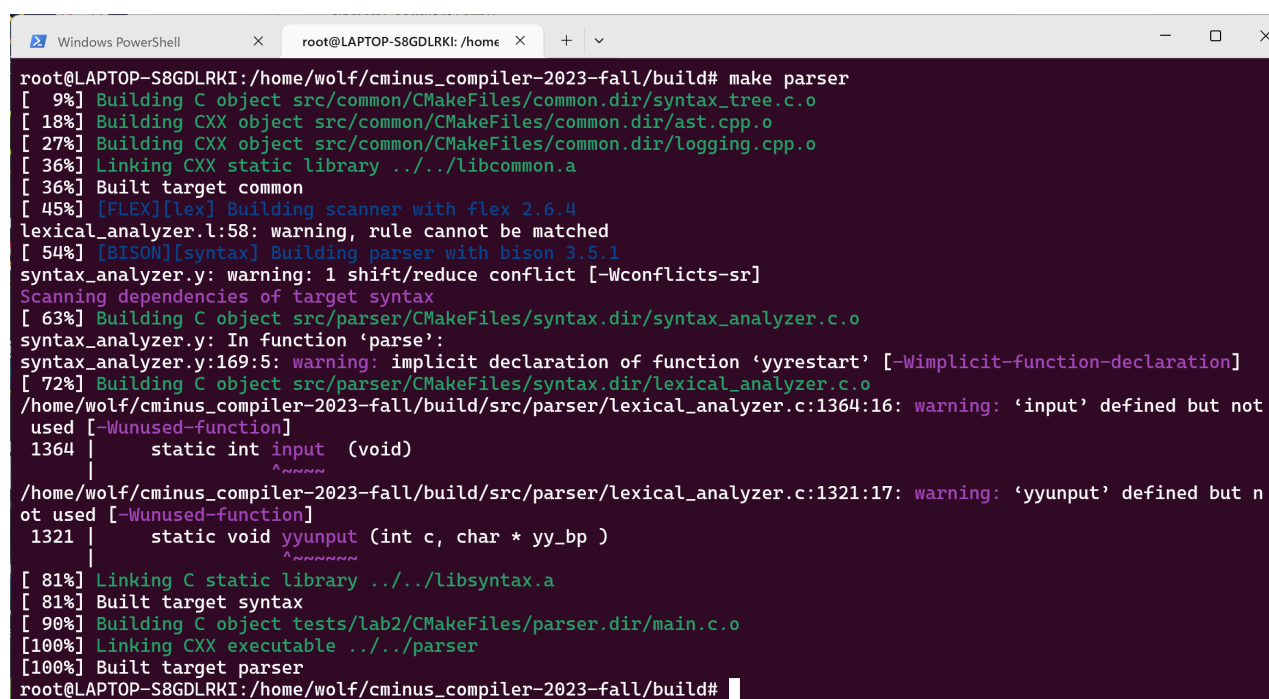
```
extern FILE *yyin;
```

## 实验结果验证

首先进行make操作，使用如下命令

```
make parser
```

若无问题，将出现如下结果



```
root@LAPTOP-S8GDLRKI: /home/wolf/cminus_compiler-2023-fall/build# make parser
[ 9%] Building C object src/common/CMakeFiles/common.dir/syntax_tree.c.o
[ 18%] Building CXX object src/common/CMakeFiles/common.dir/ast.cpp.o
[ 27%] Building CXX object src/common/CMakeFiles/common.dir/logging.cpp.o
[ 36%] Linking CXX static library ../../libcommon.a
[ 36%] Built target common
[ 45%] [FLEX][lex] Building scanner with flex 2.6.4
lexical_analyzer.l:58: warning, rule cannot be matched
[ 54%] [BISON][syntax] Building parser with bison 3.5.1
syntax_analyzer.y: warning: 1 shift/reduce conflict [-Wconflicts-sr]
Scanning dependencies of target syntax
[ 63%] Building C object src/parser/CMakeFiles/syntax.dir/syntax_analyzer.c.o
syntax_analyzer.y: In function 'parse':
syntax_analyzer.y:169:5: warning: implicit declaration of function 'yyrestart' [-Wimplicit-function-declaration]
[ 72%] Building C object src/parser/CMakeFiles/syntax.dir/lexical_analyzer.c.o
/home/wolf/cminus_compiler-2023-fall/build/src/parser/lexical_analyzer.c:1364:16: warning: 'input' defined but not
used [-Wunused-function]
1364 |     static int input (void)
      |
/home/wolf/cminus_compiler-2023-fall/build/src/parser/lexical_analyzer.c:1321:17: warning: 'yyunput' defined but n
ot used [-Wunused-function]
1321 |     static void yyunput (int c, char * yy_bp )
      |
[ 81%] Linking C static library ../../libsyntax.a
[ 81%] Built target syntax
[ 90%] Building C object tests/lab2/CMakeFiles/parser.dir/main.c.o
[100%] Linking CXX executable ../../parser
[100%] Built target parser
root@LAPTOP-S8GDLRKI: /home/wolf/cminus_compiler-2023-fall/build#
```

### ①给定案例验证



### #验证easy案例

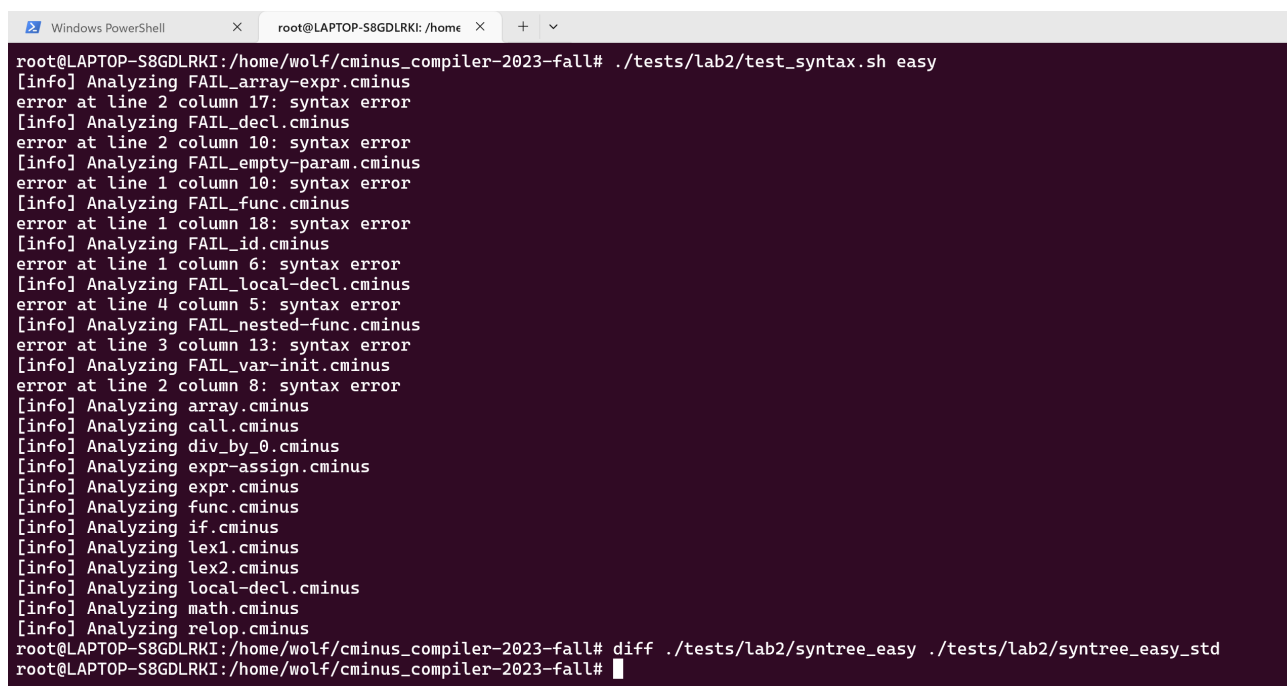
```
./tests/lab2/test_syntax.sh easy
diff ./tests/lab2/syntree_easy ./tests/lab2/syntree_easy_std

#验证normal案例
./tests/lab2/test_syntax.sh normal
diff ./tests/lab2/syntree_normal ./tests/lab2/syntree_normal_std
```

### #实验还给出了其他自动化验证方法

```
$ ./tests/lab2/test_syntax.sh easy yes
#分析所有 .cminus 文件并将结果与标准对比，仅输出有差异的文件名
$ ./tests/lab2/test_syntax.sh easy verbose
#分析所有 .cminus 文件并将结果与标准对比，详细输出所有差异
```

先验证easy，运行截图如下，未出现问题。



```
Windows PowerShell
root@LAPTOP-S8GDLRKI: /home
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall# ./tests/lab2/test_syntax.sh easy
[info] Analyzing FAIL_array-expr.cminus
error at line 2 column 17: syntax error
[info] Analyzing FAIL_decl.cminus
error at line 2 column 10: syntax error
[info] Analyzing FAIL_empty-param.cminus
error at line 1 column 10: syntax error
[info] Analyzing FAIL_func.cminus
error at line 1 column 18: syntax error
[info] Analyzing FAIL_id.cminus
error at line 1 column 6: syntax error
[info] Analyzing FAIL_local-decl.cminus
error at line 4 column 5: syntax error
[info] Analyzing FAIL_nested-func.cminus
error at line 3 column 13: syntax error
[info] Analyzing FAIL_var-init.cminus
error at line 2 column 8: syntax error
[info] Analyzing array.cminus
[info] Analyzing call.cminus
[info] Analyzing div_by_0.cminus
[info] Analyzing expr-assign.cminus
[info] Analyzing expr.cminus
[info] Analyzing func.cminus
[info] Analyzing if.cminus
[info] Analyzing lex1.cminus
[info] Analyzing lex2.cminus
[info] Analyzing local-decl.cminus
[info] Analyzing math.cminus
[info] Analyzing relop.cminus
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall# diff ./tests/lab2/syntree_easy ./tests/lab2/syntree_easy_std
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall#
```

再验证normal，运行截图如下，未出现问题。

```
Windows PowerShell
root@LAPTOP-S8GDLRKI: /home

root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall# ./tests/lab2/test_syntax.sh normal
[info] Analyzing You_Should_Pass.cminus
[info] Analyzing array1.cminus
[info] Analyzing array2.cminus
[info] Analyzing func.cminus
[info] Analyzing gcd.cminus
[info] Analyzing if.cminus
[info] Analyzing selectionsort.cminus
[info] Analyzing tap.cminus
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall# diff ./tests/lab2/syntree_normal ./tests/lab2/syntree_normal_std
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall#
```

## ②正确案例验证

对于自定义的案例验证，可以用重定向来进行

```
$ ./build/parser < test.cminus > out
```

#此时程序从 **test.cminus** 文件中读取输入，因此不需要输入任何内容。

#如果遇到了错误，将程序将报错并退出；否则，将输出解析树到 **out** 文件中。

或者我们不使用>out的话就会直接向终端输出结果

试试下面这个正确的程序。

```
# test1
int function(int a[], int n){
    int i;
    int sum;
    i = n;
    sum = 0;
    while(i>0){
        sum = sum+a[i];
        i--;
    }
    return ;
}
```

我们将程序保存到tests/lab2/test1.cminus，并使用代码执行运行

```
./build/parser < tests/lab2/test1.cminus
```

运行结果如下，能够正常解析。

```
Windows PowerShell  root@LAPTOP-S8GDLRKI: /h...
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall# ./build
/parser < tests/lab2/test1.cminus
>--+ program
| >--+ declaration-list
| | >--+ declaration
| | | >--+ fun-declaration
| | | >--+ type-specifier
| | | | >--+* int
| | | >--+* function
| | | >--+* (
| | | >--+ params
| | | | >--+ param-list
| | | | | >--+ param-list
| | | | | | >--+ param
| | | | | | >--+ type-specifier
| | | | | | | >--+* int
| | | | | | >--+* a
| | | | | | >--+* []
| | | | >--+* ,
| | | | >--+ param
| | | | | >--+ type-specifier
| | | | | >--+* int
| | | | >--+* n
| | | >--+* )
| | >--+ compound-stmt
| | | >--+* {
| | | >--+ local-declarations
| | | | >--+ local-declarations
| | | | | >--+ local-declarations
```

### ③错误案例验证

编写一个存在语法错误的程序，cminus语法中变量不可以在一个声明语句声明多个同类型变量。同时，还有一个问题，--和++并没有在cminus语法中定义。

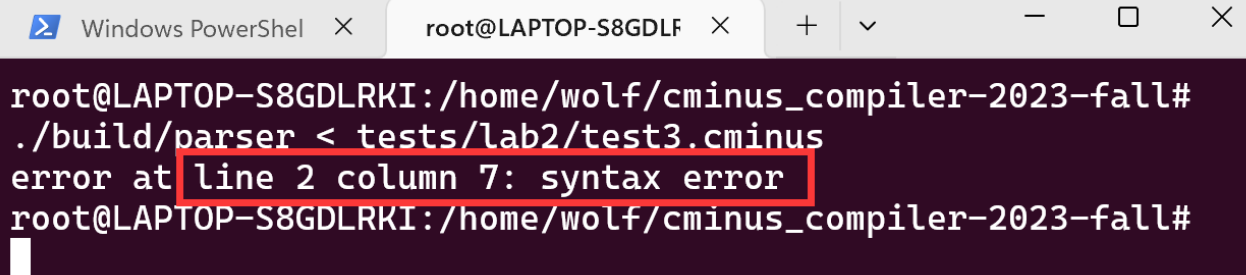
```
# test3
int function(int a[], int n){
    int i,sum;
    i = n;
    sum = 0;
    while(i>0){
        sum = sum+a[i];
        i--;
    }
    return ;
}
```

进行验证如下：

```
./build/parser < tests/lab2/test3.cminus
```

运行截图如下，能够找到问题所在

```
# test3
int function(int a[], int n){
    int i,sum;
    i = n;
    sum = 0;
    while(i>0){
        sum = sum+a[i];
        i--;
    }
    return ;
}
```



```
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall#
./build/parser < tests/lab2/test3.cminus
error at line 2 column 7: syntax error
root@LAPTOP-S8GDLRKI:/home/wolf/cminus_compiler-2023-fall#
```

## 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）

因为**bison**是使用**LALR(1)**将文法转为解析器的，**LALR**使用了前看符号（在归约时通过**FOLLOW(N)**选择性归约），所以通过前看符号可以解决左递归文法出现的冲突；

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？）

**flex**通过正则表达式读到匹配的字符串后，将字符串转为对应非终结符的语义值，然后将这个语义值放在全局变量**yylval**中，**yylval**相当于一个栈，栈的类型可以由**%union**定义。

**Bison**维护一个栈（这个栈中的每一个元素的值，都是由**yylval**所指定）来保存文法符号的语义值，当最后**n**个被移进的记号和语义值匹配某个语法规则时，就将它们依次弹出栈，再将规则的左部压栈（归约）。

**bison**定义**\$**和 **和**和**n**来引用栈中的元素：**\$\$**表示规则左部，即归约之后被重新压入栈中的元素；**\$n**表示规则左边第**n**个部件的语义值，即归约之前栈中距离栈顶编号为**i**的元素；

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）

可以通过；语法分析器认为除数为0是合法的，因为“**2/0**”可以由上面规定的文法推导出来，所以从语法上来说它是合法的，由于语法分析使用的是上下文无关文法，所以它不能判断语义是否合法；

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。

词法分析器在读到非终结符**NUMBER**时，先判断**yytext**获取到的值是否为0，不为0才将它的语义值压入到**yylval.num**中，否则不将其传到语法分析器中：

修改之后，若除数为0，则直接报错，支持除数0规避功能：

## 实验反馈

通过实现一个cminus-f语法分析器，我大致了解了bison的分析过程：

调用函数yyparse开始进行分析；

用词法分析器读取记号：yylex从输入流中识别记号并将记号类型的正值数字码（数字码用来确定需要解析的token类型）返回给语法分析器（数字码在bison编译.y文件时生成的.h文件里），并将这些记号和它们的语义值压入栈中（移进）；

当最后n个被移进的记号和组匹配某个语法规则时，可以由那个规则将它们结合起来（归约），这些记号被规则的左部取代。动作是处理归约的一部分，因为动作会计算这个组的语义值；

当yyparse遇到输入结束或者不能恢复的错误就会返回；

## 参考文献

- A橙: <https://blog.csdn.net/Aaron503/article/details/128324964>
- [https://blog.csdn.net/weixin\\_45428457/article/details/123095236](https://blog.csdn.net/weixin_45428457/article/details/123095236)