lab2 实验报告

学号：202008010415 姓名：温先武

实验要求

本次实验需要各位同学首先将自己的 lab1 的词法部分复制到 `/src/parser` 目录的 lexical_analyzer.l 并合理修改相应部分，然后根据 `cminus-f` 的语法补全 syntax_analyer.y 文件，完成语法分析器，要求最终能够输出解析树。如：

输入：

```
int bar;
float foo(void) { return 1.0; }
```

则 `parser` 将输出如下解析树：

```
>--+ program
|  >--+ declaration-list
|  |  >--+ declaration-list
|  |  |  >--+ declaration
|  |  |  |  >--+ var-declaration
|  |  |  |  |  >--+ type-specifier
|  |  |  |  |  |  >--* int
|  |  |  |  |  >--* bar
|  |  |  |  |  >--* ;
|  |  >--+ declaration
|  |  |  >--+ fun-declaration
|  |  |  |  >--+ type-specifier
|  |  |  |  |  >--* float
|  |  |  |  >--* foo
|  |  |  |  >--* (
|  |  |  |  >--+ params
|  |  |  |  |  >--* void
|  |  |  |  >--* )
|  |  |  |  >--+ compound-stmt
|  |  |  |  |  >--* {
|  |  |  |  |  >--+ local-declarations
|  |  |  |  |  |  >--* epsilon
|  |  |  |  |  >--+ statement-list
```

```
|  |  |  |  |  |  >--+ statement-list
|  |  |  |  |  |  |  >--* epsilon
|  |  |  |  |  |  >--+ statement
|  |  |  |  |  |  |  >--+ return-stmt
|  |  |  |  |  |  |  |  >--* return
|  |  |  |  |  |  |  |  >--+ expression
|  |  |  |  |  |  |  |  |  >--+ simple-expression
|  |  |  |  |  |  |  |  |  |  >--+ additive-expression
|  |  |  |  |  |  |  |  |  |  |  >--+ term
|  |  |  |  |  |  |  |  |  |  |  |  >--+ factor
|  |  |  |  |  |  |  |  |  |  |  |  |  >--+ float
|  |  |  |  |  |  |  |  |  |  |  |  |  |  >--* 1.0
|  |  |  |  |  |  |  |  >--* ;
|  |  |  |  |  |  >--* }
```

请注意，上述解析树含有每个解析规则的所有子成分，包括诸如 ; { } 这样的符号，请在编写规则时务必不要忘了它们。

2.1 目录结构

```
.
├── CMakeLists.txt
├── Documentations
│   ├── lab1
│   └── lab2
│       ├── readings.md        <- 扩展阅读
│       └── README.md          <- lab2 实验文档说明（你在这里）
├── README.md
├── Reports
│   ├── lab1
│   └── lab2
│       └── report.md          <- lab2 所需提交的实验报告（你需要在此提交实验报
告）
├── include                     <- 实验所需的头文件
│   ├── lexical_analyzer.h
│   └── SyntaxTree.h
├── src                         <- 源代码
│   ├── common
│   │   └── SyntaxTree.c       <- 分析树相关代码
│   ├── lexer
│   └── parser
│       ├── lexical_analyzer.l <- lab1 的词法部分复制到这，并进行一定改写
│       └── syntax_analyzer.y  <- lab2 需要完善的文件
```

```
└── tests                    <- 测试文件
    ├── lab1
    └── lab2                 <- lab2 测试用例文件夹
```

2.2 编译、运行和验证

- 编译

  与 lab1 相同。若编译成功，则将在 `${WORKSPACE}/build/` 下生成 `parser` 命令。

- 运行

  与 `lexer` 命令不同，本次实验的 `parser` 命令使用 shell 的输入重定向功能，即程序本身使用标准输入输出（stdin 和 stdout），但在 shell 运行命令时可以使用 `< >` 和 `>>` 灵活地自定义输出和输入从哪里来。

```
$ cd 2020fall-Compiler_CMinus
$ ./build/parser              # 交互式使用（不进行输入重定向）
<在这里输入 Cminus-f 代码，如果遇到了错误，将程序将报错并退出。>
<输入完成后按 ^D 结束输入，此时程序将输出解析树。>
$ ./build/parser < test.cminus # 重定向标准输入
<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。>
$ ./build/parser test.cminus  # 不使用重定向，直接从 test.cminus 中读入
$ ./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。>
```

  通过灵活使用重定向，可以比较方便地完成各种各样的需求，请同学们务必掌握这个 shell 功能。

  此外，提供了 shell 脚本 `/tests/lab2/test_syntax.sh` 调用 `parser` 批量分析测试文件。注意，这个脚本假设 `parser` 在 项目目录`/build` 下。

```
# test_syntax.sh 脚本将自动分析 ./tests/lab2/testcase_$1 下所有文件后缀为 .cminus 的文件，并将输出结果保存在 ./tests/lab2/syntree_$1 文件夹下
$ ./tests/lab2/test_syntax.sh easy
  ...
```

```
  ...
  ...
$ ls ./tests/lab2/syntree_easy
  <成功分析的文件>
$ ./tests/lab2/test_syntax.sh normal
$ ls ./tests/lab2/syntree_normal
```

- 验证

  本次试验测试案例较多，为此我们将这些测试分为两类：

  i. easy: 这部分测试均比较简单且单纯，适合开发时调试。
  ii. normal: 较为综合，适合完成实验后系统测试。

  我们使用 `diff` 命令进行验证。将自己的生成结果和助教提供

  的 `xxx_std` 进行比较。

```
$ diff ./tests/lab2/syntree_easy ./tests/lab2/syntree_easy_std
# 如果结果完全正确，则没有任何输出结果
# 如果有不一致，则会汇报具体哪个文件哪部分不一致
# 使用 -qr 参数可以仅列出文件名
```

  `test_syntax.sh` 脚本也支持自动调用 `diff`。

```
# test_syntax.sh 脚本将自动分析 ./tests/lab2/testcase_$1 下所有文件后缀为 .cminus 的文件，并将输出结果保存在 ./tests/lab2/syntree_$1 文件夹下
$ ./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>
```

  请注意助教提供的 testcase 并不能涵盖全部的测试情况，完成此部分仅能拿到基础分，请自行设计自己的 testcase 进行测试。

2.3 提交要求和评分标准

- 提交要求

  本实验的提交要求分为两部分：实验部分的文件和报告，git 提交的规范性。

  - 实验部分:

        - 需要完善 `./src/parser/lexical_analyzer.l` 文件;
        - 需要完善 `./src/parser/syntax_analyzer.y` 文件;
        - 需要在 `./Report/lab2/report.md` 撰写实验报告。
            - 实验报告内容包括:
                - 实验要求、实验难点、实验设计、实验结果验证、实验反馈(具体参考 report.md);
                - 实验报告不参与评分标准，但是必须完成并提交.
        - 本次实验收取 `./src/parser/lexical_analyzer.l` 文件、`./src/parser/syntax_analyzer.y` 文件和 `./Report/lab2` 目录

    - git 提交规范：

        - 不破坏目录结构(`report.md` 所需的图片请放在`./Reports/lab2/figs/`下);
        - 不上传临时文件(凡是自动生成的文件和临时文件请不要上传);
        - git log 言之有物(不强制，请不要 git commit -m 'commit 1', git commit -m 'sdfsdf'，每次 commit 请提交有用的 comment 信息)

实验难点

第一个难点是完成 lexical_analyzer.l 文件，实验要求我们把 lab1 的 lexical_analyzer.l 文件的词法部分复制到 lab2 的 lexical_analyzer.l 文件并且要合理修改部分代码。那么，要修改什么代码呢？

首先理解一个本实验的工作原理：需要把 lexical_analyzer.l 文件按照本次实验的需求进行改写，使之可以与 bison 协同工作。对于一般的符号，要建立一个新的节点。如果语法分析树不需要建立节点，（例如空格、注释、换行和错误），语法分析树里面就不需要创建对应的节点而且需要删除返回值，所以只需要更新 pos_start 和 pos_end 的值即可。

还有就是 syntax_analyzer.y 文件，首先需要给出一个节点的定义；然后再分别声明%token 节点和%type 节点的符号，其中%token 的符号是大写，代表终结符，对应词法分析，而%type 的符号是小写，代表非终结符，对应这次的语法分析；语法分析的结构其实还是很简单的，在：前面写出产生式左部，在：后面写出产生式右部，然后再在{}里面进行赋值定义，node 的第一个参数为产生式左部名称，第二个参数代表有多少个子节点，然后再在后面的参数分别标上序号。

1. $ `\text{program} \rightarrow \text{declaration-list}` $

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

实验设计

lab1 中 yytext 是一个指向输入文本的指针，所以 pass_node(yytext)是新建一个指向输入文本的结点指针

关于 lexical_analyzer.l 文件：

lexical_analyzer.l 文件里面给出了对应的例子：

```
/* Example for you :-) */
/* \+  { pos_start = pos_end; pos_end += 1; pass_node(yytext); return ADD; }*/
/* 这个例子的意思是对于lab1中正则定义即token保持不变，所以pos_start和pos_end的定义保持不变，然后对于每个token，都创建节点 */
```

因此，对于普通的 token 符号，我们只需要在返回值前加入 pass_node(yytext)表示加入阶段即可。

代码如下：

```
\* {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;}
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;}
float {pos_start=pos_end;pos_end=pos_start+5; pass_node(yytext); return FLOAT;}
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;}
[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;}
"[]" {pos_start=pos_end;pos_end=pos_start+2; pass_node(yytext); return ARRAY;}
[0-9]+\. {pos_start=pos_end;pos_end=pos_start+strlen(yytext); pass_node(yytext); return FLOATPOINT;}
```

对于换行、空格、注释和错误，只需要更新 pos_start 和 pos_end 的值：

```
\n { lines+=1;pos_start=1;pos_end=1; }//这是换行符，语法分析树不考虑，所以不用创建节点
\/\*([^\*]|(\*)*[^\*\/])*(\*)*\*\/ {
        pos_start = pos_end;
        int num = count_num_enter(yytext);
        if ( num != 0 ) { //kua
                lines += num;
                pos_end = strlen(strrchr(yytext, '\n'));
        }
        else  {pos_end += strlen(yytext);}
}//这是注释符，语法分析树也不考虑，不用创建节点
" " {pos_start = pos_end; pos_end += 1;}//这是空格符，语法分析树也不考虑，不创建节点
\t {pos_start = pos_end; pos_end += 1;}//制表符，不创建
. {return 0;}//错误，不创建节点
```

关于 syntax_analyzer.y 文件：

syntax_analyzer.y 文件其实就是 Bison 文件，它由%%分为三部分，第一部分是序曲，就是说明你要解析的 token 符号和起始符号，也就是定义节点，例如：

```
/* 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
```

我们要识别的符号如下（其实就是 cminus-f 语法规则里面的符号）：

```
33 /* TODO: Your tokens here. */
34 %token <node> ELSE IF INT RETURN VOID WHILE FLOAT ARRAY ADD SUB
35 %token <node> MUL DIV LT LTE GT GTE EQ NEQ ASSIN SEMICOLON COMMA
36 %token <node> LPARENTHESE RPARENTHESE LBRACKET RBRACKET LBRACE RBRACE
37 %token <node> IDENTIFIER INTEGER FLOATPOINT ERROR
38 %type <node> program type-specifier relop addop mulop declaration-list
39 %type <node> declaration var-declaration fun-declaration local-declarations
40 %type <node> compound-stmt statement-list statement expression-stmt
41 %type <node> iteration-stmt selection-stmt return-stmt expression
42 %type <node> var additive-expression term factor integer float call
43 %type <node> params param-list param args arg-list simple-expression
44 %start program
```

%token 的符号是大写，代表终结符，对应词法分析，而%type 的符号是小写，代表非终结符，对应这次的语法分析。

然后是 Bison 文件的第二部分，我们需要根据实验中所给出的 Cminus-f 语法，在此部分中补充语法解析。如下：

```
/* 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 ε —— 空输入 */
marisa : { puts("Hello!"); }

%%
```

例如：对于 declaration-list，查看 Cminus-f 语法：

2. $ \text{declaration-list} \rightarrow \text{declaration-list}\ \text{declaration}\ |\ \text{declaration} $

其第一条解析所得到的是两个解析符号 declaration-list declaration，所以对于该解析要执行的操作是{$$ = node("declaration-list", 2, $1, $2);}，$$ 表示当前节点，解析的 2 个节点从左到右依次编号，称作 $1、 $2，node 函数则是创建 declaration-list 的两个孩子节点

```
50 declaration-list
51 : declaration-list declaration
52 {
53         $$=node("declaration-list",2,$1,$2);
54 }
```

第二天解析是 declaration，所以要加上：

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

以下是编写的代码：

```
49 program : declaration-list { $$ = node("program", 1, $1); gt->root = $$; }
50 declaration-list
51 : declaration-list declaration
52 {
53         $$=node("declaration-list",2,$1,$2);
54 }
55 |declaration
56 {
57         $$=node("declaration-list",1,$1);
58 }
59
60 declaration
61 :var-declaration
62 {
63         $$=node("declaration",1,$1);
64 }
65 |fun-declaration
66 {
67         $$=node("declaration",1,$1);
68 }
69
70 var-declaration
71 :type-specifier IDENTIFIER SEMICOLON
72 {
73         $$=node("var-declaration",3,$1,$2,$3);
74 }
75 |type-specifier IDENTIFIER LBRACKET INTEGER RBRACKET SEMICOLON
76 {
77         $$=node("var-declaration",6,$1,$2,$3,$4,$5,$6);
78 }
```

```
80 type-specifier
81 :INT {$$=node("type-specifier",1,$1);}
82 |FLOAT {$$=node("type-specifier",1,$1);}
83 |VOID {$$=node("type-specifier",1,$1);}
84
85 fun-declaration
86 : type-specifier IDENTIFIER LPARENTHESE params RPARENTHESE compound-stmt
87 {
88         $$=node("fun-declaration",6,$1,$2,$3,$4,$5,$6);
89 }
90
91 params
92 :param-list
93 {
94         $$=node("params",1,$1);
95 }
96 |VOID
97 {
98         $$=node("params",1,$1);
99 }
100
101 param-list
102 :param-list COMMA param
103 {
104         $$=node("param-list",3,$1,$2,$3);
105 }
```

```
106 |param
107 {
108         $$=node("param-list",1,$1);
109 }
110
111 param
112 :type-specifier IDENTIFIER
113 {
114         $$=node("param",2,$1,$2);
115 }
116 |type-specifier IDENTIFIER ARRAY
117 {
118         $$=node("param",3,$1,$2,$3);
119 }
120
121 compound-stmt
122 :LBRACE local-declarations statement-list RBRACE
123 {
124         $$=node("compound-stmt",4,$1,$2,$3,$4);
125 }
126
127 local-declarations
128 :local-declarations var-declaration
129 {
130         $$=node("local-declarations",2,$1,$2);
131 }
```

```
|{
        $$=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);
}

```

实验结果验证

请提供部分自己的测试样例

首先是助教的测试样例：

```
click@click:~/桌面/cminus_compiler-2022-fall$ ./tests/lab2/test_syntax.sh easy yes
[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 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 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
[info] Comparing...
[info] No difference! Congratulations!
```

```
click@click:~/桌面/cminus_compiler-2022-fall$ ./tests/lab2/test_syntax.sh normal yes
[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
[info] Analyzing You_Should_Pass.cminus
[info] Comparing...
[info] No difference! Congratulations!
```

可以看到，没有差异，实验成功！

自己编写的测试样例：

```
1 int main(void)
2 {
3         int num;
4         int a;
5         num = 0;
6         while(a <= 10)
7         {
8                 a = a + 1;
9                 num = num + a;
10        }
11 }
```

从 1.txt 读取输入并将输出重定向到 1.out 中：

```
click@click:~/桌面/cminus_compiler-2022-fall$ ./build/parser < 1.txt > 1.out
click@click:~/桌面/cminus_compiler-2022-fall$
```

```
>--+ program
|  >--+ declaration-list
|  |  >--+ declaration
|  |  |  >--+ fun-declaration
|  |  |  |  >--+ type-specifier
|  |  |  |  |  >--* int
|  |  |  |  >--* main
|  |  |  |  >--* (
|  |  |  |  >--+ params
|  |  |  |  |  >--* void
|  |  |  |  >--* )
|  |  |  |  >--+ compound-stmt
|  |  |  |  |  >--* {
|  |  |  |  |  >--+ local-declarations
|  |  |  |  |  |  >--+ local-declarations
|  |  |  |  |  |  |  >--+ local-declarations
|  |  |  |  |  |  |  |  >--* epsilon
|  |  |  |  |  |  |  >--+ var-declaration
|  |  |  |  |  |  |  |  >--+ type-specifier
|  |  |  |  |  |  |  |  |  >--* int
|  |  |  |  |  |  |  |  >--* num
|  |  |  |  |  |  |  |  >--* ;
|  |  |  |  |  |  >--+ var-declaration
|  |  |  |  |  |  |  >--+ type-specifier
|  |  |  |  |  |  |  |  >--* int
|  |  |  |  |  |  |  >--* a
|  |  |  |  |  |  |  >--* ;
|  |  |  |  |  >--+ statement-list
|  |  |  |  |  |  >--+ statement-list
```

可以看到，没有报错，识别成功（文件太长，此处仅截取前面一部分）

再设计一个错误的代码：

```
int main(void)
{
        int num;
        int a;
        num = 0;
        while(a <= 10)
        {
                a++;
                num = num + a;
        }
}
```

```
click@click:~/桌面/cminus_compiler-2022-fall$ ./build/parser < 1.txt > 1.out
error at line 8 column 5: syntax error
```

可以看到，第 8 行出现了错误，也就是 a++。这是因为我们没有定义++这个运算符号的语义，导致无法进行语义分析，我们只定义了+这个运算符号的语义，因此上图中 a = a + 1 可以识别分析。

实验反馈

这个实验其实就是在 lab1 的基础上加上了词法分析，lab1 的功能是识别出我们所输入的一个一个单词，这里就是在 lab1 的基础上修改 lexical_analyzer.l 文件，使之能够生成语法分析树的节点，然后完成 syntax_analyzer.y 文件，根据 cminus-f 语法规则，给出语法分析树节点的语法分析，从而构建出语法分析树。难点是根据 cminus-f 的语法规则写出相应节点的语法分析代码。