LAB2 实验报告

学号 201908030406 姓名 陈叹

一、实验要求：

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）
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）
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）
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

二、实验难点

1.理解 bison 的工作过程
2.理解 bison 和 flex 是如何协调工作的，调用 pass_node 以及 return 将 flex 的分析结果传给 bison
3.抄文法。真的很长。

三、实验设计

(1)环境安装配置：
由于上次安装错，浪费了很多时间，这次按照规范来做。
(2)文法之前的部分，需要我们补充一些声明和定义
阅读实验手册：

- `YYSTYPE`：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。上面的例子中，

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

那么，对应我们的语法分析树的节点类型是什么呢？
去寻找头文件中包含的 syntax_tree.h 文件，找到节点定义如下：

```
 8 struct _syntax_tree_node {
 9     struct _syntax_tree_node * parent;
10     struct _syntax_tree_node * children[10];
11     int children_num;
12
13     char name[SYNTAX_TREE_NODE_NAME_MAX];
14 };
15 typedef struct _syntax_tree_node syntax_tree_node;
```

它包含有一个父指针，至多 10 个孩子，以及一个 name 数组，存储该节点下的文本。
那么，union 中包含的东西显然就只有节点类型，它已经包含了描述一个节点的全部。
代码如下：

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

另外，阅读实验手册，我们还需要 tokens。

- `%type <>` 和 `%token <>`：注意，我们上面可没有写 `$1.num` 或者 `$2.op` 哦！那么 bison 是怎么知道应该用 `union` 的哪部分值的呢？其秘诀就在文件一开始的 `%type` 和 `%token` 上。

  例如，`term` 应该使用 `num` 部分，那么我们就写

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

  这样，以后用 `$` 去取某个值的时候，bison 就能自动生成类似 `stack[i].num` 这样的代码了。

  `%token<>` 见下一条。

- `%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`，这样一行代码相当于两行代码，岂不是很赚。

- `yylval`：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。

那么很显然，我们将所有的符号分成终结符合非终结符，分别加入到 token 和 type 下，代码如下：

```
%token <node> ARRAY ELSE IF INT FLOAT RETURN VOID WHILE IDENTIFIER INTEGER FLOATPOINT ADD SUB MUL DIV LT LTE GT GTE EQ NEQ SEMICOLON
COMMA LPARENTHESE RPARENTHESE LBRACKET RBRACKET LBRACE RBRACE ASSIN

%type <node> program declaration-list declaration var-declaration type-specifier fun-declaration params param-list param compound-stmt
local-declarations statement-list statement expression-stmt selection-stmt iteration-stmt return-stmt expression var simple-expression
relop additive-expression addop term mulop factor integer float call args arg-list
```

其中 type 是从后面给出的文法的左边抄下来的。
(3)
主体部分，摘录文法。

1. \program → declaration-list\
2. \declaration-list → declaration-list declaration | declaration\
3. \declaration → var-declaration | fun-declaration\
4. \var-declaration → type-specifier **ID ;** | type-specifier **ID [INTEGER**
5. \type-specifier → **int** | **float** | **void**\
6. \fun-declaration → type-specifier **ID (** params **)** compound-stmt\
7. \params → param-list | **void**\
8. \param-list → param-list , param | param\
9. \param → type-specifier **ID** | type-specifier **ID** []\
10. \compound-stmt → **{** local-declarations statement-list**}**\
11. \local-declarations → local-declarations var-declaration | empty\
12. \statement-list → statement-list statement | empty\
13. \
    statement → expression-stmt
    | compound-stmt
    | selection-stmt
    | iteration-stmt
    | return-stmt
    \
14. \expression-stmt → expression **;** | **;**\
15. \
    selection-stmt → **if (** expression **)** statement
    | **if (** expression **)** statement **else** statement
    \

如上图，所有的文法他都已经给我们了。全部照原样抄下来即可，怎么处理等会再想。
公式真的很多，并且必须细心，抄错的话后面可能就错的莫名其妙了。
下图为部分代码。。

```
program : declaration-list

declaration-list : declaration-list declaration
                 | declaration

declaration : var-declaration
            | fun-declaration

var-declaration : type-specifier IDENTIFIER SEMICOLON
                | type-specifier IDENTIFIER LBRACKET INTEGER RBRACKET SEMICOLON
```

(4)

需要考虑对于每个产生式，如何处理节点。

- `$$` 和 `$1`, `$2`, `$3`, ...: 现在我们来看如何从已有的值推出当前节点归约后应有的值。以加法为例:

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

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

如上图，$$表示当前节点，$1,$2 等分别代表孩子。

再观察代码第 26 行如下图：

```
26 syntax_tree_node *node(const char *node_name, int children_num, ...);
```

那么，利用该函数初始化节点即可，例如：

`program : declaration-list` 产生式，即 **`program`** 节点，只有一个子节点，类型为 **`declaration-list`**

那么显然，处理方法就是：

```
$$ = node("program", 1, $1);
```

其他的处理也类似，如下图所示，为部分代码：

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

declaration-list:eclaration-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);}
```

(5)改写 lexical_analyzer.l

首先，每个正则表达式，都可以用 **`pos_start=pos_end;pos_end+=strlen(yytext);`** 来计算其长度，而不用去数它长度是多少。

另外，最后一部分的输出完全不需要了，因为只要返回字符类型，进入语法分析器即可。因此将后面的部分全部删除。

最后，最为重要的一点，需要在每个正则式后面加上 pass_node(yytext);，将当前的文本传到语法分析器中。因为语法分析器需要构建语法分析树，而每个节点存储文本，因此需要将 yytext 传回。

```
28 "float" {pos_start=pos_end;pos_end+=strlen(yytext);pass_node(yytext);return FLOAT;}
29 "return" {pos_start=pos_end;pos_end+=strlen(yytext);pass_node(yytext);return RETURN;}
30 "void" {pos_start=pos_end;pos_end+=strlen(yytext);pass_node(yytext);return VOID;}
31 "while" {pos_start=pos_end;pos_end+=strlen(yytext);pass_node(yytext);return WHILE;}
32
33 [a-zA-Z]+ {pos_start=pos_end;pos_end+=strlen(yytext);pass_node(yytext);return IDENTIFIER;}
34 [a-zA-Z] {pos_start=pos_end;pos_end+=strlen(yytext);pass_node(yytext);return LETTER;}
```

如上图所示，为部分正则表达式的代码。

仍然坚持记录位置是为了后面 debug。

(6)

debug

但是运行时，完全不对，甚至编译也有问题。

①

```
syntax_analyzer.y:162:15: error: ‘yyin’ undeclared (first use in this function)
```

yyin 未声明。

那么在上面声明部分，需要增加 `extern FILE* yyin;`，否则，似乎是 flex 无法读取文件。

②漏了~~许多~~东西导致的编译错误，一一修正。

还好前面记录的位置，不然根本找不到错误的地方。

③

```
a@ac:~/byyl/cminus_compiler-2021-fall$ ./tests/lab2/test_syntax.sh easy
[info] Analyzing array.cminus
error at line 1 column 4: syntax error
[info] Analyzing call.cminus
error at line 1 column 4: syntax error
[info] Analyzing div_by_0.cminus
error at line 1 column 4: syntax error
```

发现是由于返回了空白导致的。

空白符号存在于文本当中，但是不应该存在于语法树，因此 flex 读取到空白符时，没有必要返回，也没必要 pass_node.

修正之后代码如下：

```
"\n" {pos_start=0;pos_end=0;lines++;}
[ \f\r\t\v] {pos_start=pos_end;pos_end+=strlen(yytext);}
```

④重新编译运行：

```
[info] Analyzing FAIL_array-expr.cminus
error at line 2 column 16: syntax error
[info] Analyzing FAIL_decl.cminus
error at line 2 column 9: 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
```

发现还是有 error。

分析了半天，原来是 FAIL 开头的文件必须要识别出 error，因为这是不符合 cminus 文法规范的(因为他符合 C++的文法，因此找了很久)。

那这样 easy 部分就全部通过了,没有 error 了。

然后用 diff 指令和标准输出比较，运行结果如下图：

```
[info] Analyzing lex1.cminus
[info] Analyzing lex2.cminus
[info] Analyzing local-decl.cminus
[info] Analyzing math.cminus
[info] Analyzing relop.cminus
a@ac:~/byyl/cminus_compiler-2021-fall$ diff ./tests/lab2/syntree_easy ./tests/la
b2/syntree_easy_std
a@ac:~/byyl/cminus_compiler-2021-fall$
```

没有输出，说明全部匹配。
由于文法跟实验手册上写的一致即可，后面的孩子处理也不算难，只有第②步发现了许多因为手残眼瞎导致的 bug。
继续 normal：

```
a@ac:~/byyl/cminus_compiler-2021-fall$ ./tests/lab2/test_syntax.sh normal
[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
a@ac:~/byyl/cminus_compiler-2021-fall$ diff ./tests/lab2/syntree_normal ./tests/
lab2/syntree_normal_std
a@ac:~/byyl/cminus_compiler-2021-fall$
```

也正确
至此，实验结束。

四、实验结果验证

详见步骤三。

五、实验反馈

该实验主要让我们了解了 bison 的使用，以及语法分析树的结构。
在抄文法的过程中，我也了解到了一个复杂语言的文法是如何设计的。
但是，我希望以后实验能够让我们自己设计文法，而不是将文法全部给出，并设置更加合理的分阶段，分点给分的 testcase，让我们体会自己设计文法的乐趣，这同时也可以巩固我们在课内所学的文法。
抄文法虽然也能领略到文法设计的精妙，但终究浅显且无趣。