

编译原理实验一

一、实现功能

通过 flex 编写词法单元的正则表达式和相应的匹配动作来生成词法分析器。同时实现了词法错误的查找，包括浮点数，十六进制，八进制数，整数（标识符）错误，主要是通过预测可能的错误实现然后书写相应的正则表达式实现的，在发现相应错误的时候会打印词法错误。实现了注释功能，对/*...*/的块注释利用 input()以及 while 语句，判断了 eof 防止/*直到文件末尾产生的死循环，对//的行注释通过定义词法状态 INLINE_COMMENT 处理，注释会被直接忽略，不会返回 Token;

通过 bison 编写文法对应的生成式和对应的动作，来生成语法分析器。通过放置 error 符号，完成了语法部分的一部分错误恢复，修改了 yyerror，使得能够按照错误类型和行报错。使用#define parse.error verbose 增加了报错信息。

二、编译和测试程序

提供了 makefile 和 syn.sh。

可以直接使用 make 编译程序，提供了 make clean 用来清除编译后的程序。

如果使用 syn.sh,先使用 chmod +x syn.sh，再使用 ./syn.sh。

也可以使用下面三条命令

```
flex lexical.l
```

```
bison -d syntax.y
```

```
gcc -o parser syntax.tab.c lex.yy.c main.c tree.c -lfl
```

测试程序可以使用命令 ./parser test/测试文件名。也可以使用 test.sh,先使用 chmod +x test.sh，然后使用 ./test.sh，该命令可以直接测试 test 文件夹下所有文件，并将其输出到 test.sh 同级目录下的 test.txt。

三、实验亮点

1. 语法树

采用了二叉树的结构来实现多叉树。多叉树可以通过左孩子右兄弟的方法转换为二叉树。这样做的优点：

- 二叉树的结构相对简单，易于实现和理解。相较于复杂的多叉树结构，二叉树的操作和算法通常更加直观和容易编写。
- 可以在一定程度上减少存储多叉树所需的空间。
- 简化了语法树构建的难度，便于实现封装良好的结点构造函数，减少构造结点时的代码冗余，逻辑也更加清晰。
- 当匹配了一个合法生成式后，生成式的右值中的首个非空项作为左值（非终结符）的子结点，其余项依次作为上一项的兄弟结点。直接通过先序遍历就可正确的打印语法树。

由于以不同的产生式为根生成的树可能有不同数量的子节点，所以利用了 C 标准库 <stdarg.h>，这个库定义了一个变量类型 va_list 和三个宏，这三个宏可用于在参数个数未知（即参数个数可变）时获取函数中的参数。可变参数的函数通过在参数列表的末尾是使用省略号定义的。利用可变参数来添加语法树的非终结符节点。

```
struct node* nonterminal_node(const char* name, int line, int node_num,...); // 添加非终结符结点，说明名称，以及列值
```

node_num 是可变参数的数量。省略号的位置传参数时使用，通过 va_start 获取可变参数列表的首指针，然后调用 set_parent_brother 构建语法树的结构。

```

va_list va_list; // va_list表示可变参数列表类型, 实际上就是一个char指针
va_start(va_list, node_num); //初始化, 获取函数参数列表中可变参数的首指针 (获取函数可变参数列表)
set_parent_brother(parent_node, node_num, va_list);
va_end(va_list); //结束对可变参数的处理。

```

在 `set_parent_brother` 中使用 `va_arg` 获得可变参数。

```
node = va_arg(va_list, struct node*); //返回当前指向的参数类型, node*, 获取当前ap所指的可变参数并将ap指针移向下一可变参数
```

通过这个库可以简化语法树中非终结符节点的构造, 对任意一条产生式

```
StructSpecifier: STRUCT OptTag LC DefList RC { $$=nonterminal_node("StructSpecifier", @$, $1, $2, $3, $4, $5); }
```

只需要通过上面这样的传参方式来调用 `nonterminal_node` 就可以来构造非终结符节点。

利用 `sscanf` 函数简化节点赋值。`sscanf()` 会将参数 `str` 的字符串根据参数 `format` 字符串来转换并格式化数据。直接通过非终结符或终结符的名称写入到节点的 `name` 属性里, 灵活而且方便, 也可以很方便的将八进制和十六进制转换为十进制。

2. 错误恢复

错误恢复采用了自顶向下的匹配方式。对于涉及范围较小的语法结构, 如 `FunDec` 和 `ParamDec`, 错误匹配交给了上层进行处理, 以尽可能避免错误地移入其他合法语句块的情况。而对于存在语句块嵌套的文法结构, 如 `Stmt`, 则尽可能细致地进行匹配。此外, 针对 `Exp` 可能出现的多种错误情况, 也在其对应的文法中实现了错误恢复的处理。

采用自顶向下的考虑方式使得不必陷入对较小子文法的琐碎细节和可能的错误类型的思考。同时, 在含有较多语句块的文法中进行错误匹配, 使得错误恢复能够较快地结束, 尽可能减少对较大语句块的影响。

尽管经过了测试, 实现的错误恢复效果还不错, 但仍然在验收时存在一些疏漏, 比如括号不匹配可能会导致错误的弹栈。比如下面的测试文件 `testcase_3`。

```

// testcase_3.c
11 {
12     exit(-1);
13 }
14 st.kind = ADDR;
15 st.u.ret_addr = code;
16
17 while (i < extra_d-1)
18 {
19     stack = stack.next;
20     array[4+i] = array[5 * / 12];
21     i = i + 1;
22 }
23
24 st.next = stack.next;
25 stack.next = st;
26
27 return original_stack;
28 }
29
// testcase_3.c
12 {
13     exit(-1);
14 }
15 st.kind = ADDR;
16 st.u.ret_addr = code;
17
18 while (i < extra_d-1)
19 {
20     stack = stack.next;
21     array[4+i] = array[5 * / 12];
22     i = i + 1;
23 }
24
25 st.next = stack.next;
26 stack.next = st;
27
28 return original_stack;
29 }

```

```

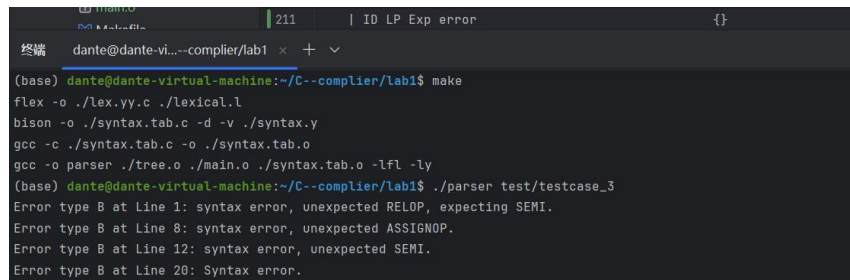
bison -o ./syntax.tab.c -d -v ./syntax.y
gcc -o ./syntax.tab.o -o ./syntax.tab.o
gcc -o parser ./tree.o ./main.o ./syntax.tab.o -lfl -ly
(base) dante@dante-virtual-machine:~/C--compiler/lab1$ ./parser test/testcase_3
Error type B at Line 1: syntax error, unexpected RELOP, expecting SEMI.
Error type B at Line 8: syntax error, unexpected ASSIGNOP.
Error type B at Line 12: syntax error, unexpected SEMI, expecting RP.
Error type B at Line 18: syntax error, unexpected LC.
Error type B at Line 20: Syntax error.
Error type B at Line 29: syntax error, unexpected end of file.
(base) dante@dante-virtual-machine:~/C--compiler/lab1$ ./parser test/testcase_3
Error type B at Line 1: syntax error, unexpected RELOP, expecting SEMI.
Error type B at Line 8: syntax error, unexpected ASSIGNOP.
Error type B at Line 20: Syntax error.

```

可以看到, 由于在 12 行未闭合的右括号, 第 18 行错误的报错了, 如果把第 12 行的括号手动闭合上, 则不会误报第 18 行的语法错误。可以补充下面这条产生式和动作, 让该程序正确报错。

Exp: ID LP Exp error

}



```
终端 dante@dante-vi...-compiler/lab1 x + v
(base) dante@dante-virtual-machine:~/C--compiler/lab1$ make
flex -o ./lex.yy.c ./lexical.l
bison -o ./syntax.tab.c -d -v ./syntax.y
gcc -c ./syntax.tab.c -o ./syntax.tab.o
gcc -o parser ./tree.o ./main.o ./syntax.tab.o -lfl -ly
(base) dante@dante-virtual-machine:~/C--compiler/lab1$ ./parser test/testcase_3
Error type B at Line 1: syntax error, unexpected RELOP, expecting SEMI.
Error type B at Line 8: syntax error, unexpected ASSIGNOP.
Error type B at Line 12: syntax error, unexpected SEMI.
Error type B at Line 20: Syntax error.
```

四、实验感悟

本次实验我通过使用 flex 和 bison 编写了 C--的词法分析器和分析器,并且完成了实验的附加部分,把实验内容和上课所讲的各部分知识点结合,让我对正则表达式、文法的使用有了更深的见解,对语法分析树的构建也是复习了一些数据结构的知识,也对词法分析过程和语法分析的过程更加的熟悉。