Lab1 实验报告

161220117 唐诗美

一、功能

能够识别出 C 源代码中的词法错误；

能够识别出合法的八进制、十六进制；

能够识别指数形式的浮点数；

能够查出 C 语言中的语法错误；

能够正确处理两种注释风格；

在没有词法语法错误时能够正确打印出语法树；

二、实现方法

词法错误是通过正则表达式匹配进行判断，如果所有正则表达式都不能匹配当前的词语，则通过排在最后的“.”来处理，把 is_error 的 flag 设置为 1，表示发生错误，打印词法错误的信息。对于整形数范围的确定，是通过分成很多种情况分别进行匹配最后排除不在范围内的数字匹配。比如到了 2147483647，就进行每一位的特判。具体可以看./code/lexical.l 文件。

```
        {
        is_error = 1;
        printf("\033[45;37mError type A at Line %d: Mysterious characters \'%s\'\033[0m\n", yylineno, yytext);
    }
```

八进制和十六进制的匹配也是通过正则表达式，输出的时候不能直接把 yytext 通过 atoi()转化为整形数，是通过定义在/code/gtree.h 中的 int OcttoDec(char* s)和 int HextoDec(char* s)，转化为符合正确八进制和十六进制。指数形式的浮点数是通过“[0-9]*(\.)?[0-9]*[Ee][+-]?[0-9]+”来匹配的。

对于 C 语言中的语法错误，是通过附录中给的文法定义去写的。主要是错误处理，添加了几个含有 error 的表达式：ExtDef:error SEMI；CompSt:error RC；Stmt:error SEMI；Stmt:Exp error；Def:Specifier DecList error SEMI；Exp:Exp LB error RB；这几个错误处理。

对于注释的处理，是仿照实验讲义中对于“//”的处理写的。

```
while (1) {
    //printf("c1c2: %c%c %d %d\n", c1, c2, c1, c2);
    if((c1=='*')){
        char c2 = input();
        if(c2 == '/') {
            #ifdef LEX
            printf("\033[42;35mthis is /**/annotation\033[0m\n");
            #endif
            break;
        }
    }
    if(c1 == -1) {
        #ifdef LEX
        printf("this is read end\n");
        #endif
        is_error = 1;
        yyerror("SYNTAX ERROR");
        break;
    }
    c1 = input();
}
```

对于语法树的打印，运用了树的数据结构，这是树的节点的结构：

```
struct node {
    char name[20];
    int lineno;
    union {
        char str[53];
        int type_int;
        float type_float;
    };
    //struct node* gparent;
    int child_cnt;
    struct node* gchild[10];
};
```

通过访问到每个符号时进行创建插入节点，把所有符号都加入到这棵树中，主要是运用这四个函数：struct node* CreateIntGNode(int int_num, int lineno);struct node* CreateFloatGNode(float float_num, int lineno);struct node* CreateIdGNode(char* id, int lineno); extern struct node* CreateTypeGNode(char* type, int lineno); extern struct node* CreateGNode(char* name, int lineno, int child_cnt, ...);最后通过 int tran(struct node* r, int layer)这个函数进行遍历打印出这棵语法树。

三、编译运行方法

本次实验是在 ubuntu16.04 中进行的。

通过在 code 文件夹中输入“make”指令，得到../中的 parser，运行 parser 进行解析即可。“make debug”可以编译出 syntax.output 文件，带调试信息地运行 parser。

四、实验总结

本次实验对于词法语法分析一块有了较深刻的认识，对于 Bison 的一些特性也了解。这次实验的 bug 主要是一些在一些细节部分，比如一开始没有声明符号的类型的错误。还有关于优先级问题，后来才发现优先级和声明的顺序是反的，越靠后声明符号优先级越高，还有对于后面规则的顺序也是很重要的，如果先声明了一些规则就会导致 bison 报错。

一开始在 flex 文件中定义了 yylineno，却怎么也改变不了行号，最后发现是没有添加’\n’词法单元；关于 yylineno 在 bison 中的使用，一开始打印行号的时候我直接使用了

yylineno，后来通过查阅资料才发现是需要用到.first_line 才能正确打印出一开始的行号，否则就是打印出最后文法结构结束的行号。

对于错误处理部分，一开始加上了一些错误处理后，shift/reduce conflict 增加了很多，通过对于开启了 debug 模式后，就会发现是哪个状态有冲突从而得出，从而对文法进行修改得到没有冲突的文法。还可以通过编译出的.output 文件更好地理解自底向上的语法分析过程。