Lab1 词法分析和语法分析实验报告

史子凡　151180115　vivianszf9@gmail.com

一、 实验进度。

完成了实验一必做和选做部分要求的全部内容，包括词法分析、语法分析、八进制和十六进制数的识别与报错，指数形式浮点数的识别与报错以及两种注释的识别与报错。

二、 编译方式。

根目录下写有 Makefile 文件，提供以下编译方式：

1. make lab1: 即编译所有相关文件并生成所需要的 parser 可执行文件；
2. make syntax: 对 syntax.y 文件进行编译，生成 syntax.tab.h, syntax.tab.c 以及 syntax.output.；
3. make lexical: 对 lexical.l 文件进行编译，生成 lex.yy.c 文件；
4. make parser: 对 main.c, syntax.tab.c 还有 treeop.c 进行联合编译，生成可执行文件 parser；
5. make clean: 删除 parser 文件；
6. make testcN：N 用 1 到 4 的数字替代，是对 test/compulsory/下的第 N 个必做样例进行测试。比如，make testc2 表示对 test/compulsory/2.cmm 进行测试。
7. make testoN：N 用 1 到 6 的数字替代，是对 test/others/下的第 N 个选做样例进行测试。比如，make testo2 表示对 test/others/2.cmm 进行测试。

三、 实现细节。

1. 词法分析。

实现在 lexical.l 文件中。

对每个符合正则表达式、被识别出来的词法单元，调用 create_node 函数在语法树中创建相对应的结点。这些结点是语法树里的叶结点。

2. 语法分析。

实现在 syntax.y 文件中。

对于每一个非终结符，调用 add_node 函数在语法树中加入其相关的结点。这

是规约的过程。加入的结点是内部结点。

3. 语法树的相关操作。

实现在 treeop.h 和 treeop.c 中。语法树的结构定义如下：

```
};
```

ntype 记录当前结点对应的类型，共 ID/TYPE/INT/FLOAT/TOKEN/UN 六种。name 记录结点的名字，方便后面打印用。lineno 记录出现的行数。childnum 记录当前结点孩子的数量。struct Node **child 指向孩子们。因为孩子的数量不定，所以我采取了动态分配的方法，每次遇到一个新的孩子就用 realloc 函数重新分配空间以满足要求。Union 即用来记录该节点对应的值。树中的每个节点都存为 struct Node *指针类型。

除此之外，还写了三个相关的函数：

struct Node* create_node(int ntype, char* name, int lineno, char* value)用于在词法分析中生成叶节点；

struct Node* add_node(int ntype,char* name, int lineno, int num_of_c, ...)用于在语法分析中生成内部节点。num_of_c 用于记录后面'...'里有多少参数。

void preorderprint(struct Node* node, int cnt)用于按先序遍历的方式打印语法树。其中，node 表示开始结点，cnt 用于计算其深度以便于打印相对应数量的空格。

4. 识别八进制和十六进制的数。

我选择在词法分析的时候将正确的错误的八进制、十六进制的数都识别为词法单元，八进制用的正则表达式为 0[0-9]+，十六进制用的正则表达式为 0[xX][0-9a-zA-Z]+，然后在语法树中创建结点的时候（即 create_node 函数）进行判断是不是合法的八进制和十六进制数，这部分判断实现在了 treeop.c 中的 int octornot(char* value)还有 int hexornot(char* value)两个函数中。如果这个节点对应的值是合法的八进制或十六进制数，那么在存这个节点的值的时候调用 int octtodec(char* value)或 int hextodec(char* value)转成对应的十进制数存起来，以便后面打印语法树的时候用。如果不是合法的，则打印相关错误信息。

5. 识别指数形式的浮点数。

仍然是在词法分析的时候将正确的错误的指数形式浮点数都识别为词法单元，正确的对应的正则表达式为(([0-9]*\.[0-9]+)|([0-9]+\.))[eE][+-]?[0-9]+，错误的对应的正则表达式为(([0-9]*\.[0-9]+)|([0-9]+\.))[eE]。然后在语法树中创建节点的时候

（即 create_node 函数）调用 int floatornot(char* value)进行单独判断，如果是，那么就直接将字符串转为浮点数存起来。如果不是，那么打印相关错误信息。

6. 识别“//”和“/*…*/”形式的注释。

对于两类注释我都是采用了正则表达式来进行识别，行注释对应的正则表达式为\/\/.*\n，这个比较简单。正确的块注释对应的正则表达式为\/*(([^*]*(*[^\/])?)*)*\/，即必须识别出一个/*还有一个*/，并且这两者间再不允许出现*/。错误的块注释我也采用了正则表达式进行识别，(\/*(([^*]*(*[^\/])?)*))*\/((([^*]*(*[^\/])?)*)*\/)*，即/*和*/必须一一对应，多余的*/是错误的。如果有匹配了错误块注释的正则表达式的，就会输出对应的错误信息 illegal block comment.

7. 报错方法。

在 yyerror 中不进行打印错误，而只是用一个全局变量 errorlineno 记录下错误的行数，以便在错误恢复中输出更为精细的错误信息。Syntax.y 中的 mark 用于记录上一个报错的地方，是为了防止错误恢复中重复报某一行的错。errornot 用于记录是否发生词法或语法错误，只要发生其值就设为 1。最后通过其值判断是否可以打印语法树。

8. 错误恢复。

实现了多种错误恢复（具体见代码）。对于 missing+“符号”的这类错误，我采取的方式是用 error 去代替这个符号原来的位置，这样就可以发现错误是少了某个符号。但这会带来重复报错的可能，所以引入变量 mark 来记录上一个已经报错的行数，如果与当前错误恢复的行不同，就进行报错。否则进行忽略。

四、 实验总结。

通过实验一，复习了树的相关结构和操作，熟悉了 Flex 还有 Bison 两个工具，并能运用它们进行简单的词法和语法分析。不过要打印出极为精细的错误提示真的很麻烦。