编译原理实验一：词法和语法分析

小组成员	学号	邮箱
韩庭轩	181860027	hantx@smail.nju.edu.cn
易舒舒	181860131	1397179783@qq.com

1. 程序功能

(1) 检测词法错误（错误类型 A）：出现 C- -词法中未定义的字符以及任何不符合 C–词法单元定义的字符；

(2) 检测语法错误（错误类型 B）：出现 C- -文法未定义的语句；

(3) 选做 1.2：识别指数形式的浮点数；

(4) 若检测到错误则报出错误行号和错误类型，支持多行报错，但每行最多报一个 A 错误和一个 B 错误；

(5) 若未检测到词法和语法错误，则根据要求打印语法分析树（先序遍历）；

2. 功能实现

(1) 检测词法错误的功能主要在 **lexical.l** 中完成，主要是定义 C- -的各个词法单元并给出相应规则。需要自行定义的只有 INT、FLOAT 和 ID 的词法单元，其正则表达式定义如下所示：

```
digit [0-9]
digits [0-9]+
INT ([1-9][0-9]*)|0
FLOAT ({digits}\.{digits})
ID [_a-zA-Z][_a-zA-Z0-9]*
```

在规则部分，为了防止将形如 **if,else,while** 等的保留字识别为 **ID**，我们将 **ID** 放到规则的倒数第二条，其余规则正常处理。最后一行我们对未能识别的词法单元进行报错，报错函数定义在 **main.c** 中。其中 **LexCurrentLineno** 来记录上次报错的行号从而保证同一行最多报一

个 A 错误。函数 **lexerror** 负责打印错误并将错误数 **ErrorNum** 加一，函数具体定义如下：

```
int LexCurrentLineno = 0;
void lexerror(char* msg){
    if(yylineno != LexCurrentLineno){
        ++ErrorNum;
        printf("Error type A at Line \%d: Mysterious
                characters \'\%s\'\n",yylineno, msg);
        LexCurrentLineno = yylineno;}}
```

(2) 检测语法错误的功能主要在 **syntax.y** 中完成，其文法定义以及结合性、优先级基本与附录 A 相同。通过接收 **lexical.l** 中规则返回的词法单元进行语法分析，即将 **lexical.l** 中的规则定义为如下格式：

```
{XXX} {return XXX;} //XXX指定义的词法单元名，如INT,COMMA等
```

下面主要阐述一下在 **syntax.y** 中的错误处理，通过定义含有关键词 **error** 的产生式来实现检测多行错误的功能。这些错误恢复产生式主要定义在**; })]** , 等符号周围。实际的一个例子如下：

```
VarDec: ID
  | VarDec LB INT RB
  | VarDec LB INT error RB {synerror("syntax error, near 'RB'");}
```

这个就是在一个 **']'** 附近进行错误恢复的例子。其中函数 **synerror** 与函数 **lexerror** 类似，都是定义在 **main.c** 中用以进行格式化报错的。

(3) 实现指数形式的浮点数只要在 **lexical.l** 中将 FLOAT 的正则表达式修改为如下即可：

$FLOAT \quad (digitsdigits) \mid ((digit*?digits)|digits)[Ee][+-]?digits$

(4) 根据上面提到的函数 **lexerror** 和 **synerror** 的定义易知已实现多行报错且每行最多一个 A 错误和一个 B 错误的功能。

(5) 最后是语法分析树的构造，为此我们新建两个文件 **tree.h** 和 **tree.c** 来定义树的节点以及相关函数。节点格式如下所示：

```
typedef struct Node{
    char name[MAX_LENGTH]; //词法/语法单元名
    char text[MAX_LENGTH]; //词法单元对应的词素/数值
```

```
    int lineno;                     //行号
    int childSum;                   //孩子节点总数
    struct Node* children[MAX_CHILD];
} Node;
```

相关函数有三个，分别实现创建节点、添加父子关系、先序打印语法分析树的功能。函数 **Node* NewNode(char* name, char* text);** 通过传入参数对新节点进行初始化并将新节点作为返回值传出，其中要注意的是首先要为新节点 **malloc** 一块大小为 **Node** 的空间。函数 **void AddChild(Node* parent, Node* child);** 将 **child** 挂到 **parent** 的子节点数组中。函数 **void PreOrder(Node* node, int num);** 对语法分析树进行先序遍历并打印，函数开头首先打印 **num** 数目的空格。遇到语法单元则进行递归，并将 **num** 加 **2**，遇到词法单元则结束递归，如果遇到 INT、FLOAT 等则按要求打印数值等即可。

接着将 **lexical.l** 的规则格式修改如下，即为每个词法单元创建一个新节点：

```
{XXX}       {yylval.node = NewNode("XXX",yytext);return XXX;}
```

然后将 **syntax.y** 中正确产生式的规则格式修改如下，下面分别是不为空和为空的形式：

```
OptTag: ID {$$ = NewNode("OptTag", ""); AddChild($$, $1);}
  | /* empty */ {$$ = NULL;}
```

产生式不为空则为产生式的头创建一个新节点并以其为父节点将产生式的体作为子节点依次挂上去；

产生式为空则返回一个 NULL 节点。

3. 编译方法

使用 **Makefile** 进行编译，指令如下，在 **Code** 文件夹的命令行中依次执行即可进行测试：

```
make clean
make parser
make test
```