编译原理实验——词法分析与语法分析
实验报告

171860667 袁想　　171860664 谢鹏飞

一、实现功能

（1）识别 C--语言源代码中的词法错误，即出现 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）识别 C--语言源代码中的语法错误。

（3）识别“//”和“/*...*/”类型的注释。

（4）如果没有词法错误和语法错误，打印源程序的语法树结构。

二、数据结构、重要变量与函数说明

（1）语法树结构，定义在 tree.h 中：

```
typedef struct TreeNode
{
    char type[MAX_LENGTH]; //表示终结符或非终结符类型，非终结符统一为nonterminal，终结符例如int则为TYPE，>则为RELOP。MAX_LENGTH=32+1
    char contents[MAX_LENGTH]; //若是终结符，存储相应的词素。若是非终结符，存储非终结符对应的名称内容
    int linenumber; //该节点在程序中的行号位置
    struct TreeNode *parent; //该节点的父节点指针
    int childrennum; //该节点有几个子节点
    struct TreeNode *children[MAX_CHILDREN_NUM]; //存储该节点的子节点指针的数组。MAX_CHILDRE_NUM=7，因为观察到C--文法中产生式右边最多有七个符号
} TreeNode;
```

（2）创建语法树节点函数

```
TreeNode *createNode(char *type, char *contents, int linenumber);
```

第一个参数对应语法树结构中的 type，第二个参数对应语法树结构中的 contents，第三个参数对应语法树结构中的 linenumber。

（3）添加节点函数

```
void addNode(TreeNode *parent, TreeNode *child);
```

第一个参数是父节点指针，第二个参数是要插入的子节点指针。如果一个父节点有多个子节点要插入，那么就多次调用这个函数。

（4）打印语法树函数

```
void printParsingTree(TreeNode *root, int blankcount);
```

第一个参数是当前即将处理打印的语法树的节点（首次打印时这里就是语法树的根节点），第二个参数是记录输出该节点内容时需要缩进的空格数。

（5）全局变量

```
int errornum; //统计程序的错误数量（词法+语法错误）
TreeNode *root; //最终生成的语法树的根节点
```

三、程序核心内容设计思路与实现

（1）词法分析

根据 C--词法写出每一种终结符类型对应的正则表达式。一旦识别出这些表达式，

就向语法分析模块传输终结符类型并建立终结符在语法树中的节点。对于未定义的、无法匹配的字符，采用.进行匹配。如果匹配到了.表达式，那么就说明源程序发生词法错误，产生错误类型 A，通过 printf 打印出对应的词法错误，同时错误数量加 1。

（2）“//”和“/*...*/”注释识别的实现

这两种注释的识别都在词法分析中完成。对于“//”型注释，创建正则表达式"//"，即当 flex 识别到两个连续的'/'字符时触发对于该种注释的处理。处理方式就是一直从输入流读入字符（利用 flex 的 input 函数），直到读到换行符（代表该型注释下一行开始就不是注释，需要处理）或者文件末尾 EOF（该型注释可能在源代码最后一行）为止。这些字符将仅被读入，不会作正则匹配也不会向语法分析传递任何数据，即代表了注释的效果。对于“/*...*/”型注释，创建正则表达式"/*"，即当 flex 识别到'/'后面紧跟一个'*'时触发对于该种注释的处理。处理方法是一直从输入流读入字符（利用 flex 的 input 函数），直到读到一个'*'后面紧跟一个'/'时为止。这样就可以保证/*一定会匹配到第一个遇到的*/，避免允许嵌套的“/*...*/”注释。此外，如果一直读到文件末尾 EOF 还没有读到*/，那么就发生了一个语法错误，没有*/与/*匹配。在书写处理时，发现 flex 版本在 2.6.0 以上 input 函数读到 EOF 时会返回 0，flex 版本在 2.6.0 及以下的 input 函数读到 EOF 会返回-1，而在 C 语言中 EOF 符号代表的值就是-1，因此对于不同的 flex 版本判断是否读到 EOF 需要不同的书写形式。

（3）语法分析

在词法分析中，对于每一个识别到的 C--文法的终结符，将在词法分析处理识别到的终结符的代码中，创建该终结符在语法树的节点，同时返回这个终结符的类型供语法分析使用。例如，对于整型数，在词法分析识别到这种的数的正则表达式时作上述处理：

```
{INT} {yylval.node = createNode("INT", yytext, yylineno); return INT;}
```

其中 yylval 的类型在 syntax.y 中被定义为一个只包含 struct TreeNode* node 的 union 类型。在语法分析中，所有的终结符和非终结符的 type 都被定义为 struct TreeNode*，在发生规约动作时，例如一个 A -> B C 的产生式，那么我们在该产生式规约的时候就会进行如下处理：此时说明 B、C 已经完成了语法树节点的创建，那么我们只需要利用 createNode 创建 A 对应的节点，然后调用两次 addNode 函数分别将 B 和 C 对应的节点（利用$1 和$2 访问）加到它们的父节点 A 节点（利用$$访问）下面即可。如果进行了空串产生式的规约，如 A -> ε，那么 A 对应的节点就是一个空指针。在语法分析时还需要注意运算符的左结合、右结合和优先级。通过查询 C--文法可以得到这些信息，并将运算符性质在 syntax.y 中利用%left、%right 指示，同时利用指示性质声明位置不同来表示它们的优先级（优先级高的在优先级低的符号声明下面）。由于负号和减号对应的都

是 MINUS 这一终结符类型，但是负号的优先级比减号高，因此为了体现这一点，就需要模仿对于实验指导中 IF-ELSE 语句移入-规约冲突的处理，定义一个比 MINUS 优先级更高的算符 NEG，在 syntax.y 中书写如下：

```
%right NOT
%nonassoc NEG
```

之后将产生式 Exp : MINUS Exp 改进为：Exp : MINUS Exp %prec NEG，就使得负号的优先级将比减号（对应文法 Exp MINUS Exp）要高。

对于语法分析的错误匹配，即语法分析当前状态调用 yylex 从词法分析中获取词法单元，如果当前状态没有针对这个词法单元的动作，就发生了语法错误。为了能够检查出文件中的所有错误，需要利用保留字 error 来进行错误恢复。error 可以实现再同步，使得可以继续后面的语法分析，从而可以发现源程序多个语法错误。因此核心问题就是将 error 放在产生式的哪些位置。在我们的实现中，我们设计的大部分 error 产生式都是放在括号、分号结尾之前的位置，即";"、")"、"]"、"}"这一类符号前面。有一些 error 产生式也将放在'('、'['、'{'的前面，有一些 error 将放在两个成对的括号"()"、"[]"之间，例如 ID LP error RP，VarDec LB error RB。为了覆盖尽可能多的语法错误情况，我们还在一些非常有标志性的终结符的前面进行 error 匹配，例如 IF error ELSE Stmt。同时，我们也在一些高层产生式中添加适当的 error 产生式匹配，这样再同步就更容易成功，例如我们设计了高层 error 产生式 error FunDec CompSt。此外，我们也注意了高层 error 产生式与底层 error 产生式之间的平衡，否则会出现较多的移入-规约冲突。在匹配到一个 error 产生式后，我们在其处理代码中将会把源程序错误数量加 1，并且通过 printf 根据 error 所在的行报出这一行的语法错误。同时我们重写了 yyerror，在里面不写任何输出，因为一旦发现一个语法错误 bison 就会自动调用 yyerror，避免了重复输出同一行的语法错误。

如果源程序没有任何的语法错误，意味着错误数量 errornum 为 0，那么就可以打印语法树。printTree 函数的思路是从 root 开始，先序遍历语法树，如果当前遍历到的节点为空，直接返回；按照 blankcount 输出空格。之后判断，如果当前节点子节点数非 0，那么这个节点一定不是终结符，因此输出该节点的信息，之后递归调用 printTree 打印其子节点信息。如果子节点数为 0，且其类型为 INT，那么利用 atoi 将 contents 的内容转为数值，类型为 FLOAT 则调用 atof 函数。

四、编译说明

利用实验网站上给定的 Makefile 文件，放在与源代码同样的路径下，终端中输入 make 即可编译。或者终端中依次输入以下指令：flex lexical.l　bison -d -v syntax.y gcc main.c syntax.tab.c tree.c -lfl -ly -o parser。