编译原理实验

第一次试验报告

孙楷文 学号：111100027 （5 班）

1. 代码文件

实验代码文件包含以下 lexical.l、syntax.y、main.c、Makefile。用 make 或 make analysis 命令即可编译。生产可执行程序 parser。

2. 词法分析

2.1. 对合法字符串的匹配

在定义部分定义了 digit [0-9]、letter [a-zA-Z] 和 _letter (_|[a-zA-Z]) 以辅助完成包括实验必做内容和所有选作内容（八进制十六进制、指数形式浮点数、两种注释）的词法识别。

其中，我认为“/* ... */”的 RegExp 的设计最为复杂，因为要让第一个“/*”去匹配第一个遇到的“*/”。最终，RegExp 设计为 "/*"([*]*(([^*/])+([/])*)*)*"*/" 。

2.2. 对非法字符串的匹配

对于无法成功匹配的词法单元，运行到“.”表达式，输出报错信息，printf(“Error type A at line %d: Unknown \”%s\”.\n”, yylineno, yytext); 另外，还有以下对特殊非法字符串的匹配。

2.3. 对非法八进制数字符串的特殊处理

把 8 和 9 出现在八进制数中是常见的词法错误。根据 flex 默认的优先规则，在正确的八进制正则表达式“0[0-7]+”之后，写一个非法的八进制正则表达式“0[0-9]+”，并输出错误信息。由于 flex 在长度相同且能匹配两个表达式时，会优先匹配写在前面的，所以合法的八进制数不会被误认为非法。

2.4. 对非法十六进制数字符和非法的浮点数的特殊处理

专门识别了非法的十六进制 `(0x|0X)({digit}|{letter})+` 、科学计数法 `(({digit}+\.{digit}*)|({digit}*\.{digit}+))(E|e)` 和普通的浮点数 `(\.{digit}+)|(0|([1-9]{digit}*)\.)`。

2.5. 识别了非法变量名/函数名（ID）

识别了以数字开头的非法变量名/函数名（ID）`({digit}+)(({_letter}+)({digit}*))+`

3. 语法分析

根据 C--语法产生式，完成了 syntax.y 的主体部分。

根据 C--语法的优先级和结核性，写出了下图中的代码。其中，用 LOWER_THAN_ELSE 区分 ELSE 的优先级，用 UNARY 区分符号和减号 MINUS 的优先级。

```
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE
%right ASSIGNOP
%left OR
%left AND
%left RELOP
%left PLUS MINUS
%left STAR DIV
%right UNARY NOT
%left LP RP LB RB DOT
```

4. 语法树的构建

4.1. 语法树节点的结构

因语法树的叶子节点(lexical)和内部节点(syntax)都需要用到该结构，且 syntax 的代码 include 了 lexical 的代码，故把该结构放在了 lexical.l 中。

语法树的节点结构为：

```
struct Node
{
        struct Info     info;
        int             numofchild;
        struct Node*    child[MAX_NODE_NUM];
};
```

其中，多叉树的子节点最大个数为 7，已满足 C--语法的所有合法情况；info 记录了该节点的词法/语法信息。struct Info 的数据结构如下图：

```
struct Info
{
        char    name[50];
        int     lineno;
        union   {
                unsigned int INT;
                float FLOAT;
        }       value;
        enum NodeType   nodetype;
        char    notation[MAX_NOTATION_LEN];
};
```

其中，name 为词法/语法单元的名字，如“ID”；lineno 为行号；对于整型/浮点型数值，其值记录在联合结构 value 里；enum NodeType nodetype 表明了该节点是词法单元还是语法单元，enum NodeType {ERR, LEX, STX}；notation 的作用相当于“备注”，可以以文字的形式填写任何我想填写的东西，比如词法单元 ID 的变量名，或者填写比较关系符 RELOP 的具体符号是什么。

4.2. 与节点相关的函数和变量

对于 Info 和 Node 结构，在 lexical.l 代码中实现了它们的内存空间申请、初始化、空间释放等函数，并实现了为 Node 节点插入子节点的函数 void InsertNode(Node* parent, Node* child)。

指向语法树的根的指针 syntaxTreeRoot 被定义在 lexical.l 中。

4.3. 语法分析的返回值（yylval 的返回值）类型

在 syntax.y 中定义下图中的数据结构。真正使用到的其实只有 Node* type_node，但因为 union 中必须至少有两个成员，故添加了 char 型成员。

```
%union{
        struct Node* type_node;
        char type_double;
}
```

4.4. lexical.l 中的动作

识别出字符串后，建立节点，填写节点 info，给 yylval.type_node 赋值，返回 token。下图是识别十进制字符串的例子。

```
0|([1-9][digit]*)       {                       //dec-integer
        struct Node* node = getNewNode();//printf(%s",yytext);
        strcpy(node->info.name,"INT");
        node->info.lineno = yylineno;
        node->info.value.INT = (unsigned int)atoi(yytext);
        node->info.nodetype = LEX;
        strcpy(node->info.notation,yytext);
        yylval.type_node = node;
        return INT;
}
```

4.5. syntax.y 中的动作

识别出语法单元后，建立节点，填写节点 info，将（1 个或多个）子节点挂到父节点上，对$$赋值，返回 token。下图是识别 StructSecifier 的例子。

```
StructSpecifier:STRUCT OptTag LC DefList RC
                {
                        struct Node* node = getNewNode();
                        strcpy(node->info.name,"StructSpecifier");
                        node->info.lineno = $1->info.lineno;
                        node->info.nodetype = STX;
                        InsertNode(node,$1);
                        InsertNode(node,$2);
                        InsertNode(node,$3);
                        InsertNode(node,$4);
                        InsertNode(node,$5);
                        $$ = node;
                }
```

5. 错误恢复

5.1. 总结了三种底层的错误

在写错误恢复部分之前，我现总结出了三种可以进行错误恢复的类型。

第一种的正确形式是：多个非终结符+终结符。对应的错误恢复文法是：error 终结符。这种错误恢复即以最后的一个终结符为同步符号。

第二种的正确形式是：中间有终结符。如 DecList → Dec COMMA DecList，可能是遗忘了 DecList，恢复文法设计为 DecList → error COMMA。

第三种的正确形式是：终结符+其他符号。如 Exp → FLOAT，可能是在文法符号的前面有多余的东西，其恢复文法设计为 Exp → error FLOAT。该类的特征为首个文法符号为终结符。

对于第三种，比如 Specifier → TYPE | StructSpecifier。在我为其添加了 Specifier → error TYPE 后，面对下面的这个测试用例，我的程序可以非常准确的指明错误的具体原因：int 的前面多东西了！“Extra thing before TYPE”

```
int main()
{
    /*
      comment
       /*
         nested comment
       */
    */ <-- 多余的符号
    int i = 1;
}
```

程序输出：

```
Error type B at line 8: Syntax error. Illegal Specifier (Extra thing before TYPE)
```

最后，根据实际的语法含义、错误是否常见、是否会产生语义冲突、实验测试结果等，去掉不合适的错误恢复语句。比如，当发生冲突时，抛弃第三种猜测，因为文法符号前面多了东西的可能性较小。

5.2. 另一种错误恢复

对于叫高层的错误，由于高层产生式中非终结符较多，难以对 error 做同步，所以可以根据整个语法来考察产生式最后一个非终结符可能产生的（子）语法树最底层的终结符中最右端的终结符是哪个，并在该终结符前加 error，作为高层的错误恢复产生式。比如，DecList 可产生 Dec 和 Dec COMMA DecList，而 Dec 常以“]”、“)”和“;”结尾，故错误恢复产生式设计为 error RB、error RP 和 error SEMI。但因：warning: rule useless in parser due to conflicts：DecList error RB”，故只保留 error RP 和 error SEMI。