# 编译原理第一次实验报告

201220187 孙赫彬

## 一、功能

1. 识别词法错误（错误类型A）
2. 识别语法错误（错误类型B）
3. 识别八进制数和十六进制数
4. 识别指数形式的浮点数
5. 识别“//”和“/.../”形式的注释
6. 构造语法树，在没有有任何词法或语法错误时，按照先序遍历的方式打印每一个结点的信息

## 二、实现思路

1. 词法分析

   首先根据Appendix_A中7.1.1 Tokens的内容构造正则表达式，并返回tokens。

   其中需要特别注意的是INT和FLOAT。INT型变量有3种情况，分为8进制、10进制和16进制，需要分别识别它们的字符串并把它们转化为int型的变量存储。这里使用strtol函数进行转化。

   ```
   digit [0-9]
   oct 0|([1-9]{digit}*)
   dec 0[0-7]+
   hex 0[xX][0-9A-Fa-f]+
   ```

   FLOAT变量有两种情况，正常的浮点数和指数形式的浮点数，其中正常浮点数是小数点前后有1或多个数字，指数形式浮点数的基数部分中，小数点前或后有一边有数字即可，指数符号为[Ee]；指数部分由[+-]?[0-9]+组成。

   ```
   com_float {digit}+\.{digit}+
   sci_float (({digit}+\.{digit}*)|({digit}*\.{digit}+))[eE][+-]?{digit}+
   ```

   为了识别注释，还需要添加两项正则表达式"//"两项"/*"，分别识别两种注释的开始。注释的结尾在函数体里面识别，不断使用input函数读取下一个输入字符，"//"中，直到读取到"\n"为止；"/*"中，直到读取到连续的"*/"为止。

   ```
   "//" {
           char tmp = input();
           while(tmp!='\n')  tmp = input();
       }
   "/*" {
           char tmp = input();
           while(1){
               if(tmp == '*'){
                   tmp = input();
                   if(tmp == '/')break;
               }
               else tmp = input();
           }
       }
   ```

这些正则表达式中，"."输出语法错误，没有返回值；注释、换行符、制表符和空格也都没有返回值。其余情况都调用语法树的init函数来创建新节点存在yylval中，并返回对应的token，比如：

`";" {yylval.node=init("SEMI",yylineno,1);return SEMI;}`

2. 语法分析

使用%token定义终结符，%type定义非终结符，根据Appendix_A中表12定义优先级和结合性。

接着根据Appendix_A中7.1 文法定义逐步构建文法分析的基本框架，其中每个产生式的语义动作包括使用init函数为左边的非终结符创建新的节点，存储在$$中，然后调用insert函数，逐个将产生式右边的符号设置为$$的孩子节点。其中Program作为初始符号，它的节点还应该存入root变量中，这样才能在之后访问语法树。

3. 语法树构造

语法树是一个多叉树，其节点结构如下所示：

```
struct Node {
    int flex;//flex变量用来存储该结点是词法单元还是语法单元
    char Type[32];//用来存储该词法单元或语法单元的类型
    union Content content;//用来存储该词法单元的内容，可以是int、float或字符串
    int line;//用来存储词法单元或语法单元的位置
    struct Node* first_child;//指向第一个孩子结点的指针
    struct Node* sibling;//指向第一个兄弟结点的指针，和first_child共同实现了多叉树的存储
    struct Node* tail_child;//指向最后一个孩子结点的指针，它的存在是为了方便进行插入新的孩子结点
};
```

其中为了在存储词法单元内容的同时节省空间，我定义了一个联合数据类型Content，它能够存储int或float或字符串类型的变量。

```
union Content {
    int my_int;
    float my_float;
    char my_char[32];
};
```

语法树的根节点保存在全局变量`struct Node* root`中。

创建结点可以使用`struct Node* init(const char* type, int line,int flex);`函数，它可以为新结点开辟空间，将参数存入结点中，然后将指针都置为NULL，最后返回新结点的指针。

要将一个已有结点插入到父结点的下方，可以使用`void insert(struct Node* parent, struct Node* child);`函数。如果该结点是父结点的第一个孩子，则将父结点的first_child和tail_child都设置为该结点，反之，则将当前父结点的最后一个孩子的sibling指针指向该结点，并且将父结点的tail_child指针指向该结点。

可以使用`void print_tree(struct Node* r,int pre);`函数来打印语法树，参数r是当前要打印的结点，pre是打印该结点类型前，应该打多少次空格*2。接下来打印结点类型，然后根据结点信息判断接下来要打印的内容。如果是词法单元，则flex=1，需要对比如果Type是ID、TYPE、INT或者FLOAT，则需要输出结点内容。如果flex!=1则输出行号。最后，递归调用print_tree函数，先输出孩子结点，再输出兄弟结点。

4. 错误恢复

首先设置一个全局变量`int ERROR=0;`用来记录错误的个数，如果该文件识别完，ERROR=0，则没有错误，输出语法树，反之，代表有错误，不需要输出语法树。

词法单元的错误检测很简单，只需要在所有的正则表达式都不匹配时，加上一个`.`用来匹配前面没有匹配的字符串，如果匹配到这一项，则输出错误提示，并把ERROR加一。

语法单元的错误检测比较复杂，需要考虑各种情况。我将error可能出现的所有情况都加入了语法分析的框架，然后根据归约冲突和错误提示逐步对不需要的产生式进行删除。

包含error的产生式会先自动调用yyerror函数，然后执行后面大括号内的语义动作。我的代码中，这部分的语义动作只有一句话，就是输出错误提示，比如：

```
ExtDef :error FunDec CompSt{printf("Error type B at Line %d: error FunDec CompSt. \n",@1.first_line);}
```

我还重写了yyerror函数，该函数只有一句代码，就是将ERROR变量加一。

## 三、编译方法

使用makefile编译，输入命令`make`

## 四、编译环境

1. GNU Linux Release: Ubuntu 20.04, kernel version 5.15.0-48-generic
2. GCC version 7.5.0
3. GNU Flex version 2.6.4
4. GNU Bison version 3.5.1