作者：刘金明　16计算机1班　320160939811

引言：本次实验目的为制作一个语法分析器，承接上次的词法分析器，将词法分析器输出的字符串作为输入，输出一棵语法法分析树，并进行中间代码转换将语法树翻译成四元式。**注：由于在manjaro下使用wps编写的文档，若后面出现排版错乱的情况，请打开PDF版的实验报告继续阅读**

对于语法分析器，我写了两份代码，第一份使用递归方式，后来觉得时间复杂度过高，便使用非递归的预测分析表重新写了一份代码。后面两份代码我都会分别说明

1.总体说明

1. 编程语言:Python 3.7
2. 编程平台:manjaro
3. 编程环境:vscode
4. 完成的内容:承接上次的词法分析器，将其输出的字符表转成一个语法树，并完成四元式的转换。
5. 采用的方法：
   (1) 自上而下的递归方式（parser.py）
   (2) 非递归的预测分析表（LL.py）
6. 具体实现的语法：
   (1) 语法树支持：变量声明语句，赋值语句，输出语句，程序块
   (2) 四元式支持：变量声明语句，四则表达式的赋值语句。

2.文件结构

除去之前完成的词法分析器，本语法分析程序共涉及 1 个文件,四元式转化涉及一个文件 现将其说明如下:

- parser.py　　递归的语法分析程序
- LL.py　　非递归的预测分析语法分析
- get_predict_table　生成预测分析表
- generate.py　　中间代码生成程序

4.语法文法说明

刚开始很难自己写出一个完整的语法产生式，后来借鉴网上的语法产生式，自己对其一点一点扩展，最终形成适合自己程序的语法产生式。下面先给出一个四则运算的文法

```
Expr -> Term ExprTail
ExprTail -> + Term ExprTail
| - Term ExprTail
| null

Term -> Factor TermTail
TermTail -> * Factor TermTail
| / Factor TermTail
| null
Factor -> (Expr)
| num
```

将其用 python 代码实现，用一个字典存储该文法，每个产生式的左部作为字典的“键“，而产生式右部存储在该键值所对应的数组中中。实现如下

```
grammars = {
   "E": ["T ET"],
   "ET": ["+ T ET", "- T ET", "null"],
   "T": ["F TT"],
   "TT": ["* F TT", "/ F TT", "null",],
   "F": ["NUMBER", "BRA"],
   "BRA": ["( E )",],
   "END_STATE":r"(null)|(NUMBER)|(ID)|[+\-*/=]|(LBRA)|(RB
RA)"
}
```

可以看出，每个产生式左部键值对应的数组内容为相应的右部内容。`END_STATE`为终结符集合

之后自己对其进行扩展，改成自己想要的文法。该文法从主函数`program`开始解析，自上而下分解多条语句，声明语句，赋值语句，四则运算与输出语句

```
grammars = {
    "Program":["keyword M C Pro"],
    "C":["( cc )"],
    "cc":["null"],
    "Pro":["{ Pr }"],
    "Pr":["P ; Pr", "null"],
    "P":["keyword L", "L","printf OUT"],
    "L":["M = E", "M"],
    "M":["name"],
    "E":["T ET"],
    "ET":["+ T ET", "- T ET", "null"],
    "T":["F TT"],
    "TT":["* F TT", "/ F TT", "null"],
    "F":["number", "BRA"],
    "BRA": ["( E )"],
    "OUT":["( \" TEXT \" , V )"],
    "V":["name VV", "null"],
    "VV":[", name VV", "null"],
    "END_STATE":
    r"(null)|(number)|(name)|(keyword)|(operator)|(printf)
|(separator)|(TEXT)|[+\-*/=;,\")({}]"
}
```

5. 语法分析器说明

5.1 递归方式

词法分析器接受一个由词法分析器产生的字符表，之后从全局的

文法字典中获取第一个文法根节点，开始自上而下的递归分析，分析方法是：对于非终结符，继续在文法字典中查询相对应的关键字，并切割其所有遍历到的字符串，每个字符串被切割成的字符数组将作为该非终结符的潜在子节点，挨个进行递归并生成语法树子节点。

对于终结符，进行匹配，若终结符类型存在于终结符表中，则匹配成功，反之报错。

5.2 非递归的预测分析表

该语法分析方式分为两大步骤（生成预测表，生成语法树）

预测分析表使用嵌套的字典存储，外部字典的键值为非终结符，内容为一个内部字典，内部字典的键为终结符，内容为产生式右部

例如：

```
predict = {
    “E”：{
        “+”：“+ E F”
    }
    “F”：{
        “×”： “× T TT”
    }
}
```

5.2.1 生成预测表

预测表的生成过程又分为三小步

- 生成 first 集合
- 生成 follow 集合
- 生成预测表

其中生成 follow 集合的过程较为复杂（我不知道正常情况是不是这样）非终结符的后继非终结符的 first 集合可能存在如下情况

1．A -> BC

2．C -> D | null

3．D -> (A) | i

那么在一次遍历过程中，因为 C 的 first 集合存在 null，所以需要将 follow（A）加入 follow（B）（重点）**但是**！此时的 follow（A），并不是完整的 ,它可能在后续的遍历中会继续更新自身的 follow 集合所以此时 follow(B) 中加入的 follow(A) 并不是完整的 follow（A）

为了解决这种情况，我加入了订阅者模式，一种实时更新的机制，订阅者为一个字典 ,字典键值为产生式左部 ,字典内容为产生式右部。

简而言之：follow（A）发生了更新，那么曾经将 follow（A）加入自身的 B，C 也更新其 follow。并且，这是一个递归过程。详细说明见代码。

6.四元式产生器说明

简而言之，对一个语法树进行遍历，遍历过程中遇到相应节点进行相应的处理。我的四元式产生器任然使用递归方法实现，目前仅实现了声明语句，赋值语句和四则混合运算。

7. 代码说明

7.1 paraser.py

```
def build_ast(tokens):
  root = Node("Program")
  # 建立根节点，自上而下分析
```

```
class Node: # 语法树节点
  def match_token(self, token): #字符匹配
  def __init__(self, type): # 初始化函数
  # 建立抽象语法树
  def build_ast(self,tokens: list,token_index=0):
```

详细说明见源码注释

7.2 generate.py

```
class Mnode:  # 四元式节点
def view_astree(root, ft=None): # 生成四元式
```

8. 运行代码

运行环境：python3（若使用 python2 会发生编码错误）

1. 查看词法分析生成的单词表

```
python lexer.py
```

2. 查看生成的语法树

```
python LL.py
```

3. 若要使用递归方式

```
python parser.py
```

4. 查看生成的四元式

```
python generate.py
```

5. 查看预测分析表生成过程

```
python get_predict_table.py
```

```
➜ python实现编译器—词法分析 python parser.py
(Program, None)
        (keyword, int)
        (M, None)
                (name, main)
        (C, None)
                ((, ()
                (cc, None)
                        (null, None)
                (), ))
        (Pro, None)
                ({, {)
                (Pr, None)
                        (P, None)
                                (keyword, int)
                                (L, None)
                                        (M, None)
                                                (name, a)
                                        (=, =)
                                        (E, None)
                                                (T, None)
                                                        (F, None)
                                                                (number, 1)
                                                        (TT, None)
                                                                (null, None)
                                                (ET, None)
                                                        (+, +)
                                                        (T, None)
                                                                (F, None)
                                                                        (BRA, None)
                                                                                ((, ()
```

```
➜ python实现编译器—词法分析 python generate.py
(+,4,2,T2)
(*,T2,3,T1)
(+,1,T1,T0)
(=,T0,0,a)
(=,0,0,b)
(*,3,7,T6)
(+,2,T6,T5)
(/,T5,6,T4)
(+,1,T4,T3)
(=,T3,0,b)
```

```
➜ python实现编译器 git:(master) python get_predict_table.py

first集合如下

Program ['type']
C ['(']
cc ['null']
Pro ['{']
Pr ['type', 'name', 'printf', 'null']
P ['type', 'name', 'printf']
L ['name']
LM ['=', 'null']
M ['name']
E ['number', '(']
ET ['+', '-', 'null']
T ['number', '(']
TT ['*', '/', 'null']
F ['number', '(']
BRA ['(']
OUT ['(']
V ['name', 'null']
VV [',', 'null']

follow集合如下

Program []
```

9. 收获与感悟

使用自上而下的递归方法做抽象语法树相对来说还是比较简单的，然而对于性能来说确实十分糟糕，虽然目前编译的小代码不会有太大延迟，但是代码量一多肯定还是会有很大的时间代价。

另外四元式的递归生成感觉代码写的还是比较笨重，为了达成目的而写的代码，不优雅，不美观，可读性差，，自己写的时候也把自己绕晕了，递归使用的太多，逻辑上的可读性非常差，需要仔细读才能看懂代码意思，可能过段时间自己也看不懂了。

预测分析表比想象的要简单，写起来还挺方便，除了 follow 集合求着麻烦，但是时间复杂度要比递归方法好很多！