**LL(1)语法分析器设计文档**

1. **开发环境**

本程序的编程环境为：基于win10-64bit操作系统，wings IDE 5的集成测试环境，语言为python2.7。

1. **任务说明**

对以下文法G进行语法分析。

文法起始符：E

非终结符：E T F

终结符： num ( ) + - \* /

文法G的产生式为：

E -> E + T | E – T | T

T -> T \* F | T / F | F

F -> ( E ) | num

要求对上述文法构造LL1分析表，并构造LL1预测分析程序。

**文件集合：**

1. LL1grammar.py：LL1分析程序的脚本
2. get\_FIRST\_SET\_AND\_FOLLOW\_SET.py：本程序衍生出的单独脚本，通用的FIRST集合FOLLOW集求解器。
3. FIRST\_AND\_FOLLOW\_SET.txt：上面脚本生成的两集合，仅供演示通用脚本的效果。
4. grammarInput.txt：程序的文法输入，有一定的格式约定以便于程序扫描，生成程序中约定的python字典格式。
5. tokens.txt：测试串的输入，默认测试串是符合语法规范的。
6. LL1Result.txt：LL1分析结果。

在本次实验中，按照以下两个主要步骤进行：

1. **得到文法G的LL1分析表**

给定输入为文法G**（从grammarInput.txt以文件形式输入）**，输出为LL1分析表TABLE，本程序中以python字典的形式存储此分析表。注意在这一步骤中文法G有被消除左递归的过程，即输出了另一个文法G’，文法G’同时也是构造LL1分析表的输入。

1. **根据LL1分析表，构造LL1语法分析程序，并输出语法分析结果**

给定输入（从tokens.txt以文件形式输入）为步骤1中得到的TABLE，待识别的token序列（在本实验中简化为python字符串）；输出为LL1语法分析过程。输出的内容被保存在LL1Result.txt中，包括LL1分析表和LL1分析器的分析过程。

请注意，为简化程序，程序中没有设计具体的错误处理模块，而是采取简单的中断机制来捕获语法错误。程序默认输入的tokens串是有效的且符合文法规范的，程序可以确保在输入串合语法的情况下，输出一个正确、完整的LL1分析过程。

1. **设计思路**

采取框图的形式说明本实验的主要思路。

程序总体分为五个主要部分，每一部分都有对应的输入输出，并在脚本中已经集成了相应的函数（模块）。

1. **消除（直接）左递归**

通过过程delLeftRecursion(VN,VT,P)完成，该过程需要一个文法本身的VN和VT集合，并需要输入原文法的所有产生式。该过程输出一个新的VN与P，VT保持不变。

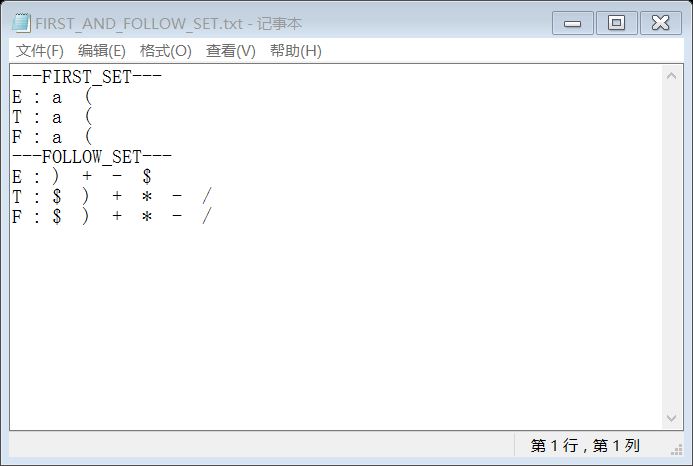
由于给定文法中存在直接左递归，我们首先要做的就是消除文法的左递归以防LL1分析器陷入无限循环。在本实验中，由于考虑到给定的文法在消除直接左递归后不会存在新引入的间接左递归，并考虑到给出一个通用的左递归消除器较为复杂，因此本算法为消除直接左递归，并不能有效地消除间接左递归。但这对于本实验来说足够了。

1. **求解FIRST集（通用的）**

通过过程getFirstSet(VN,VT,P)完成，该过程需要输入文法的VN，VT和P来作为信息，输出为文法的FIRST集。

考虑到给出一个通用的FIRST集产生器并不复杂，因此该过程被设计为通用的，即输入任何一个文法都可以得到它相应的FIRST集合。

已整合到模块get\_FIRST\_SET\_AND\_FOLLOW\_SET.py中，可以直接使用，该模块要求一个gammarInput.txt作为文件输入，输出为FIRST\_AND\_FOLLOW\_SET.txt。如本例中的程序，求解两集合的效果如下：



1. **求解FOLLOW集（通用的）**

通过过程getFollowSet (VN,VT,P,FIRST\_SET,START\_VN)完成，该过程要求输入文法的VN,VT,P，由于需要给文法的开始符提供‘$’的初始化支持，因此还需要文法的起始符号。由于求解FOLLOW集一般是基于一个已经给定的FIRST集的，因此我们也将之前求解的FIRST集作为输入。与FIRST求解器相同，该求解器也被设计为通用的，即输入任何一个文法都可以得到相应的FOLLOW集合。

已整合到模块get\_FIRST\_SET\_AND\_FOLLOW\_SET.py中。

1. **构造LL1分析表**

通过过程makeTable(VN,VT,P,FIRST\_SET,FOLLOW\_SET)完成，该过程利用了之前求解的所有信息，输出为LL1文法分析表TABLE。

1. **构造LL1分析程序**

通过过程grammerLL(test\_tokens,TABLE,START\_VN,VN,VT)完成，该过程要求一个token串作为输入，该token串理应是词法分析器的输出，但是本实验做了简化，将其简化为python的字符串类型。由于要得到LL1分析器对token串的分析过程，我们输出两个结构，其一是left\_reasons，这是由每一步得到的左句型构成的序列；其二是forms，这是每一步推导中所用到的产生式的序列。

1. **程序实现**
2. **数据结构**
   1. **文法**

本实验用‘#’符号代表‘ε’。

文法的终结符和非终结符均用python的集合表示。

VN = {'E','T','F'}

VT = {'a','+','-','\*','/','(',')'}

文法的产生式用python字典表示，键为非终结符VN，值为产生式右部构成的序列。

P = {

'E' : ['E+T' , 'E-T' , 'T'] ,

'T' : ['T\*F' , 'T/F' , 'F'] ,

'F' : ['(E)' , 'a']

}

简化程序设计，我们默认字符‘a’为代表了num的终结符，我们在测试串中用它来表示任意一个经词法分析器分析后属性为num的token。

* 1. **FIRST集与FOLLOW集**

同样用python字典表示，键为非终结符VN，值为对应VN的FIRST集或FOLLOW集构成的集合。

* 1. **LL1分析表**

采用 ( VN , VT ) -> ( VN , alpha )的形式存储，这是键和值都为二元组的字典，其中alpha表示某个产生式的右部。其实可叙述为，用一对VN和VT去索引某个产生式，注意这里面的VT包括了‘$’符号。

* 1. **tokens序列**

采用python字符串表示，例如

test\_tokens = '(a+a)\*(a-a)/a'

* 1. **LL1分析程序的输出**

用两个序列表示。其一为左句型序列，其二为表达式序列。表达式的构成方式与文法中的相同。

1. **模块算法实现**
   1. **消除（直接）左递归**

消除直接左递归的伪代码如下：

*def delLeftRecursion(VN,VT,P):*

*创造新的名字集合，以备新的VN使用*

*new\_P = {}*

*new\_VN = set()*

*new\_VN = new\_VN | VN*

*for 每一个vn:*

*for vn的每一个产生式:*

*if 这个产生式含有直接左递归:*

*构造新的产生式，并将新的符号加入到new\_VN中*

*return new\_VN , new\_P*

代码十分简洁有效，但代价是其很差的通用性，该过程不能解决间接左递归。消除左递归后，便可以用新产生的文法来求解FIRST与FOLLOW集。

* 1. **FIRST集**

求解FIRST集的伪代码如下：

*def getFirstSet(VN,VT,P):*

*FIRST\_SET = {}*

*for 任何一个vn:*

*FIRST\_SET[vn] = set()*

*while True:*

*FIRST\_SET\_PRE = FIRST\_SET*

*for 每一个vn的每个产生式:*

*if产生式右侧第一个符号为vt或空:*

*将这个符号加入到vn的FIRST集中*

*else:*

*向后扫描，直到遇到一个不含空的vn为止*

*将这些vn的FIRST集加入到vn的FIRST集中*

*if 一轮扫描后 FIRST\_SET == FIRST\_SET\_PRE:*

*return FIRST\_SET*

比较关键的是何时跳出FIRST求解器。我们需要提前构造一个便于比较的FIRST集合副本FIRST\_SET\_PRE，在每一次扫描完成后，对比当前的FIRST\_SET和副本FIRST\_SET\_PRE是否相同，如果相同表示此时的FIRST集合已经完备，因为它没有在这次扫描中添加任何新的成分。

* 1. **FOLLOW集**

与求解FIRST集类似，伪代码如下：

*def getFollowSet(VN,VT,P,FIRST\_SET,START\_VN):*

*FOLLOW\_SET = {}*

*for 任何一个vn:*

*FOLLOW\_SET[vn] = set()*

*添加$符号至START\_VN的FOLLOW集中*

*while True:*

*FOLLOW\_SET\_PRE = FOLLOW\_SET*

*for 每一个vn:*

*for vn的每一个产生式:*

*定位vn在每一个产生式的位置*

*if vn 之后有vt:*

*把这个vt加入到vn的FOLLOW集中*

*elif vn 之后为空:*

*把产生式左部的vn的FOLLOW集*

*并入当前vn的FOLLOW集中*

*else 之后为另一个vn’*

*把vn’的FIRST集并入vn的FOLLOW集中*

*if 一轮扫描后 FOLLOW\_SET\_PRE == FOLLOW\_SET*

*return FOLLOW\_SET*

过程与FIRST集相似，在此不再赘述。

* 1. **构造LL1分析表**

此过程十分简单，伪代码如下：

*def makeTable(VN,VT,P,FIRST\_SET,FOLLOW\_SET):*

*TABLE = {}*

*for 每一个vn:*

*vn的每一个产生式:*

*对产生式右部求FIRST集操作，得到集合K*

*for 集合K中的vt:*

*if vt不为空:*

*填表TABLE[(vn,vt)] = (vn,form)*

*if K中含有空符号*

*for对于FOLLOW\_SET[vn]中的每一个vt:*

*填表BLE[(vn,vt)] = (vn,form)*

*return TABLE*

我们希望得到一个产生式右部的FIRST集这与我们求解FIRST集的过程有些不同但简单很多，我们单独定义一个辅助过程getFiSetForForm来完成它。集合K是填表的依据，根据空符号是否存在于集合K，我们可以将所有产生式填入到LL1分析表中，也即LL1分析表的构造过程。

* 1. **构造LL1分析程序**

有了LL1分析表，LL1分析程序只是查表、操作的过程，也十分简单，伪代码如下：

*def grammerLL(test\_tokens,TABLE,START\_VN,VN,VT):*

*初始化符号栈，将$压栈，并将文法起始符START\_VN压入栈中*

*将test\_token + ‘$’作为输入，ip指针指向第一个字符*

*while 栈不空:*

*X是栈顶符号，a是ip所指符号*

*if X是终结符号或者$：*

*if X == a：*

*栈顶弹出X*

*ip++*

*else：*

*报错*

*else：*

*if TABLE[X,a]是存在的索引为A->B*

*栈顶弹出A并把B串反向压入栈中*

*输出这个产生式*

*else：*

*报错*

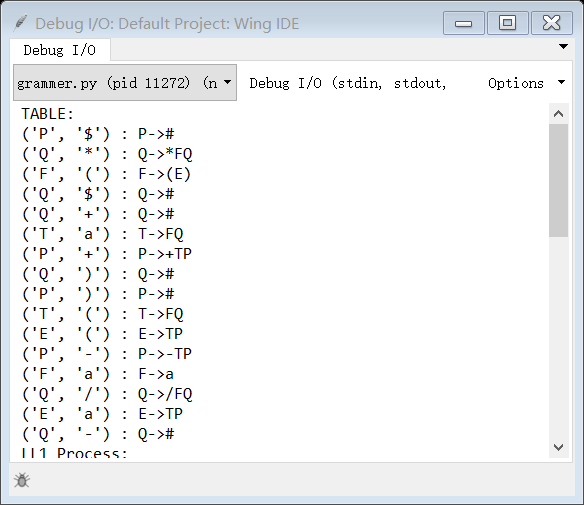
*return 左句型变化信息以及产生式*

这与书上算法4.1是类似的，该过程返回一连串的左句型和产生式。

1. **测试用例**

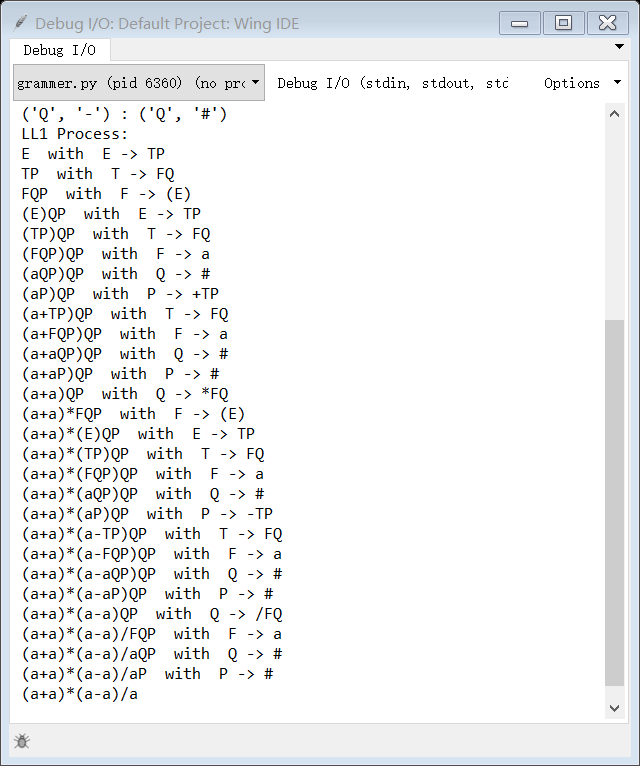
**1、(a+a)\*(a-a)/a**

输出的LL1分析表（TABLE）为：



按照程序约定的数据结构，LL1分析表是足够清晰的。

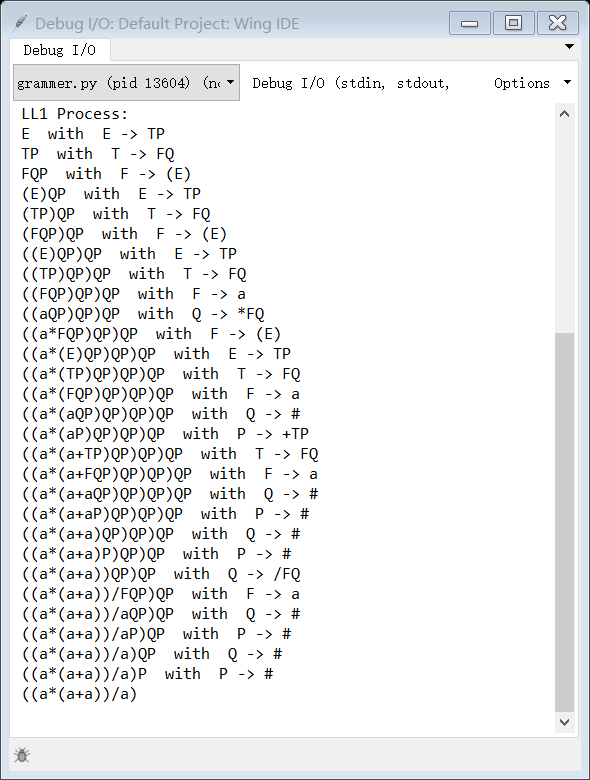
输出的LL1分析过程为：



可以看到，左侧为分析步骤中的左句型，右侧是该句型之后所用到的产生式，输出为下一个左句型。可以看出左侧的左句型序列构成了一个完整的最左推导过程，这个过程也是LL1分析程序的执行过程。从输出文件LLIResult.txt观察到的结果是相同的。

**2、((a\*(a+a))/a)**

省略分析表，直接输出分析过程。



很容易看出其分析的正确性。