# 编译原理-LL(1)分析

## 吴元守 16051730

## 一、实验目的

1. 了解 LL(1)语法分析是如何根据语法规则逐一分析词法分析所得到的单词，检查语法错误，即掌握语法分析过程。

2. 掌握LL(1)文法判别调剂和 LL(1)语法分析器的设计与调试。

## 二、实验内容

针对任意的文法，编写相应的左递归消除、左公共因子提取程序，求解相应的FIRST、FOLLOW集，构造预测分析表，并编写LL(1)语法分析程序，并给出测试句子的分析过程。（注：左递归消除和左公共因子如果在实验三里做了，可以直接拿过来用）

判断LL(1)文法部分：

1. 输入：文法
2. 处理：左递归消除、左公共因子提取，FIRST、FOLLOW等集合构造，判断LL(1)
3. 输出：是LL(1)的情况输出预测分析表，否则判断不是LL(1)

LL(1)分析程序部分：

1. 输入：诸如对应文法的符号串，以$结束。

2. 处理：基于分析表进行 LL(1)语法分析，判断其是否符合文法。

3. 输出：串是否合法。

## 三、具体实现

在这个实验中，我没有完成左递归的消去以及左公共因子的提取，实现了对于消除了左递归的文法构造FIRST集、FOLLOW集、判断文法是否为LL(1),构造LL(1)预测分析表以及判断符号串是否符合文法的函数。

#### Ⅰ.文法

文法以以下的形式给出: ::用来分割文法的左部与右部

S::a  
S::^  
S::(T)  
T::SM  
M::,SM  
M::#

首先，我从文本文件中以行为单位读出对应的文法，然后按照::进行切分，左部作为字典的字典的索引，右部添加进字典的值(为一个集合)中，这样一个左部推出的多个右部都可以放入集合中，可以存储存在多个右部的产生式。

**def** get\_grammar() -> dict:  
 **with** open(file, encoding=**'utf-8'**, mode=**'r'**) **as** fd:  
 **for** line **in** fd.readlines():  
 l, r = line.replace(**'\n'**, **''**).split(**'::'**)  
 **if** l **not in** grammer:  
 grammer[l] = set()  
  
 grammer[l].add(r)

#### Ⅱ.求解FIRST集

将文法存储进键值对之后，我们开始使用键值对存储的文法求解FIRST集:

传入函数的参数包括待求解FIRST集的串以及文法的键值对。FIRST集的求解时比较简单的,算法如下所示:

首先串的长度大于等于以一。

1.第一步我们先看待求解串的第一个字符，如果是终结符，那么它的FIRST集就是它本身，

2.如果是非终结符的话，

a.如果它对应的文法不能推出空，那么FIRST(target)包含它的产生式右部的所有FIRST集；

b.如果其中一个对应的文法推出了#,

①.如果右部只有#一个符号，将#加入FIRST集

②.否则将能推出#的符号的右面所有符号的FIRST集加入FIRST(target)，即FIRST(taget).add(get\_first(target[1:], grammar))

**def** get\_first(target, grammar):  
 *'''  
 返回* **:param** *target: 右部 如 A->Bc 种的 Bc* **:param** *grammar: 语法规则* **:return***: first集  
 '''* c = target[0]  
 one\_first = set() *# one\_first:set* **if** c.isupper():  
 *# 非终结符* **for** st **in** grammar[c]:  
 **if** st == **'#'**:  
 *# 可以推出空且空后面还有符号的情况* **if** len(target) != 1:  
 *# 递归求解* one\_first = one\_first.union(get\_first(target[1:], grammar))  
 **else**: *# A->#* one\_first = one\_first.union(**'#'**)  
 **else**:  
 *# 非终结符递归调用* recursive\_first = get\_first(st, grammar)  
 one\_first = one\_first.union(x **for** x **in** recursive\_first)  
 **else**:  
 *# 终结符* one\_first = one\_first.union(c)  
 **return** one\_first

#### Ⅲ.求解FOLLOW集

求解FOLLOW集的时候，我设计了两个函数，首先下面的这个函数需要传入语法，所有的非终结符以及开始符号(开始符号的FOLLOW集需要加入$,即$7一定可以跟在开始符号后面),

然后我需要初始化自己的数据结构，所有的非终结符作为键，值为一个集合，然后不要忘了将开始符号的FOLLOW集加入$符号。

接下来我们执行不动点算法:

当新获得的FOLLOW集follow\_now不等于原来的FOLLOW集follow\_set时:

将follow\_now赋值给follow\_set

对于每一个非终结符I:

求解I新的FOLLOW集并赋给follow\_now[I]

算法在FOLLOW集没有收敛的时候会一直执行直到FOLLOW集不再变化，这就是我求解FOLLOW集的上层函数。

**def** all\_follow(S: str, Non\_terminal: set, grammar):  
 follow\_set = dict()  
 **for** i **in** Non\_terminal:  
 follow\_set[i] = set()  
  
 follow\_now = dict(follow\_set)  
 follow\_now[S] = follow\_now[S].union(**'$'**) *# 开始符号*  
  
 **while** follow\_set != follow\_now:  
 follow\_set = dict(follow\_now)  
 **for** i **in** Non\_terminal:  
 follow\_now = get\_follow\_set\_of\_one\_nonterminal(S, i, grammar, Non\_terminal, follow\_now)  
  
 **return** follow\_now

接下来的这个函数用于求解在一个给定的FOLLOW集中求解某一个非终结符的FOLLOW集的函数。

首先我们需要遍历所有的产生式右部，只有产生式的右部存在我们要求解FOLLOW集的非终结符B的时候，我们才会对这个产生式进行分析。

在存在B的情况下:

如果说在产生式的最右边:( A->aB) 特殊情况包括 A->B

**将FOLLOW(A)全部交给FOLLOW(B) 龙书规则3)**

**否则求解产生式中B中右边的所有串的FIRST集**

**如果 # in 求解出的FIRST集 龙书规则3)**

**则将FOLLOW(A)=添加给FOLLOW(B),也就是上面的情况的变种**

**将求解得到的FIRST集除了#加入FOLLOW(B) 龙书规则2)**

**这样一遍下来就获得了当前FOLLOW集下所能获得的一个非终结符的新的FOLLOW集**

#### Ⅳ.求解LL(1)预测分析表

语法分析表M同样是一个键值对,键为栈顶元素与当前输入串的第一个元素组成的元组

算法描述如下:

对于所有的产生式右部:

如果右部不为#

*对于First(a)中的每个终结符号a, 将推导式的右部加入table[A,a]*

*否则:*

*推出空的话，对于所有的FOLLOW集元素b,将推导式的右部加入table[A,b]*

同时，在这里还可以检测语法是否是LL(1)型的语法，只要当分析表M一个元组已经有一个对应的产生式，构造M时又出现了可以填充这个元组的情况的时候，换句话说，就是分析表的一种情况下对应两条可能的推出规则的时候，这个文法不是LL(1)文法。

**def** get\_ll1\_table(follow, grammar: dict):  
 M = {} *# 语法分析表* **for** left\_part **in** grammar:  
 **for** each\_right **in** grammar[left\_part]:  
 **if** each\_right != **'#'**:  
 *# 对于First(a)中的每个终结符号 将推导式的右部加入table[A,a]* **for** i **in** get\_first(each\_right, grammar):  
 **if** (left\_part, i) **in** M:  
 print(**"No LL1 Grammar"**)  
 **return None** M[left\_part, i] = each\_right  
  
 **else**:  
 *# #的时候 Follow集* **for** i **in** follow[left\_part]:  
 **if** (left\_part, i) **in** M:  
 print(**"Not a Legal Grammar"**)  
 **return None** M[left\_part, i] = each\_right  
 **return** M

#### Ⅴ.判断给定字符串是否符合合法

*最后，我们构造判断输入的串是否合法的函数:*

*对于这个函数，我们需要传入待处理的字符串，开始符号，以及预测分析表:*

*算法如下*

1. *将 $加入待处理的字符串的末尾*
2. *将$与开始符号压入栈中，在这里我使用列表来模拟栈的操作，指针指向输入串的开始*
3. *当栈非空时:*

*获取栈顶元素*

*如果栈顶元素与待输入的第一个串相等，弹出这个元素，指向串的指针向后加一*

*否则访问LL（1）分析表M*

*如果分析表中没有对应规则:*

*错误 将错误标志设为True*

*如果说推出符号不是空:*

*将原来的符号弹出，将分析表给出的符号逆序压入栈中(因为最左推导)*

*否则*

*推出空，直接弹出栈顶元素。*

1. *最后输出推导是否成功*

*因为这个函数算法与具体实现非常接近，详细的代码不再贴出来了。*

## 四.测试

*对于LL(1)分析，我分别使用了两组测试文法进行测试:*

### 文法一

1. *第一个文法就是这篇报告开头第二页的文法，这是LL(1)作业的文法，测试的代码如下，分别进行了First集，Follow集、LL(1)分析表以及有效字符串的判断的测试:*

*这是从文本到产生式键值对的解析输出*

* 1. *S::a*
  2. *S a*
  3. *S::^*
  4. *S ^*
  5. *S::(T)*
  6. *S (T)*
  7. *T::SM*
  8. *T SM*
  9. *M::,SM*
  10. *M ,SM*
  11. *M::#*
  12. *M #*

*First集*

*{'S': {'(', 'a', '^'}, 'T': {'(', 'a', '^'}, 'M': {'#', ','}}、*

*Follow集*

*{'S': {'$', ',', ')'}, 'T': {')'}, 'M': {')'}}*

*LL(1)分析表*

*{('S', '('): '(T)', ('S', 'a'): 'a', ('S', '^'): '^', ('T', '('): 'SM', ('T', 'a'): 'SM', ('T', '^'): 'SM', ('M', ')'): '#', ('M', ','): ',SM'}*

*与当时手工构造的结果对比，可以得出都是正确的*

*最后给出字符串***'(a,a)' '(a,a))'的分析过程与结果，显而易见是正确的。**

['(', 'T', ')', '$']

['T', ')', '$']

['S', 'M', ')', '$']

['a', 'M', ')', '$']

['M', ')', '$']

[',', 'S', 'M', ')', '$']

['S', 'M', ')', '$']

['a', 'M', ')', '$']

['M', ')', '$']

[')', '$']

['$']

[]

Legal!

['(', 'T', ')', '$']

['T', ')', '$']

['S', 'M', ')', '$']

['a', 'M', ')', '$']

['M', ')', '$']

[',', 'S', 'M', ')', '$']

['S', 'M', ')', '$']

['a', 'M', ')', '$']

['M', ')', '$']

[')', '$']

['$']

Illegal!

### 文法二

对于文法grammar = {**'D'**:{**'TV'**},**'T'**:{**'i'**,**'f'**},**'V'**:{**'cW'**},**'W'**:{**',V'**,**'#'**}}，

这是另外一次作业，我对文法做了一些修改，因为本实验将单个小写字母当作一个单位，所有int缩写为i,float缩写为f，id缩写为c,

First集:

{'D': {'i', 'f'}, 'T': {'i', 'f'}, 'V': {'c'}, 'W': {',', '#'}}

Follow集:

***{'V': {'$'}, 'T': {'c'}, 'W': {'$'}, 'D': {'$'}}***

***LL(1)分析表***

***{('D', 'i'): 'TV', ('D', 'f'): 'TV', ('T', 'i'): 'i', ('T', 'f'): 'f', ('V', 'c'): 'cW', ('W', ','): ',V', ('W', '$'): '#'}***

*对比可知结果都是正确的*

*下面是*process(**'ic,c,c'**, S, M)的输出结果，可见是合法的串

*['T', 'V', '$']*

*['i', 'V', '$']*

*['V', '$']*

*['c', 'W', '$']*

*['W', '$']*

*[',', 'V', '$']*

*['V', '$']*

*['c', 'W', '$']*

*['W', '$']*

*[',', 'V', '$']*

*['V', '$']*

*['c', 'W', '$']*

*['W', '$']*

*['$']*

*[]*

*Legal!*

*下面是*process(**'icc'**, S, M)的运算结果，可见输入的字符串是不合法的。

*['T', 'V', '$']*

*['i', 'V', '$']*

*['V', '$']*

*['c', 'W', '$']*

*['W', '$']*

*Illegal!*

## 总结

LL(1)分析程序总体上还是比较复杂的，复杂型主要体现在这个程序的每一个部分都牵扯到一个具体的算法，首先需要反复复习这些算法，另一方面算法的伪代码描述转化为能够运行的代码，这两者之间需要我去自己构造合适的数据结构。

我个人感觉FIRST集的算法实现起来比较容易，FOLLOW集因为这是一个不动点算法，且需要多轮的迭代才能趋于稳定，实现起来比较复杂，而最后的构造分析表与基于栈的判断是否是合法的串的程序反而难度不算大。

当然，难度最大的反而是左递归的消除与左公因子的提取，前者我一直想实现一个更通用的版本，但是一直都没有找到思路，后者感觉需要一个高效的多字符公共前缀匹配算法。

总的来说，编译原理的课程设计的难度感觉很高，特别是涉及很多的算法，但是这个学期特别是期末考查课占据了太多的时间，导致能够花在课程设计上的时间比较有限，同时实验也确实不好写，最后挺多东西都没能实现，还是有些遗憾的。

最后，自己编码实现算法和手算结果还是有很多的不同的，很多在脑子里想起来很简单的事情编码实现起来却相当繁琐，自己实现了一遍算法的时候要反反复复的推敲算法，会发现自己对于某个算法其实并不是那么的明白；对于算法为何这样设计也会有一些感想。