**《算符优先分析程序》实验报告**

1. **实验目标**

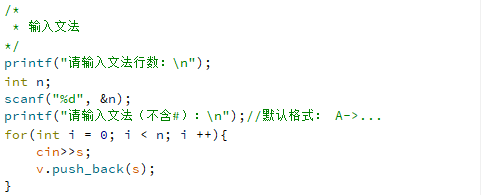
算符优先分析法是一种简单直观、广为使用的自下而上的分析方法。这种方法特别有利于分析表达式，易于手工实现。

所谓算符优先分析法，就是仿照算术式的四则运算过程而设计的一种语法分析方法。实际上，该方法最初就是为了分析和翻译程序语言中的表达式而设计的。这种方法首先是要规定运算符之间（更一般说是终结符之间）的优先关系和结合性质，然后利用这种关系，用比较相邻运算符的优先顺序来确定句型的“句柄”和进行归约。

本次实验即是编写简单的算符优先分析程序。

1. **实验要求**
2. 能够输入文法；
3. 根据文法输出相应的firstvt集和lastvt集；
4. 根据文法自动构建算符优先矩阵并输出；
5. 多次输入句子，完整的输出移近规约的过程。
6. **实验步骤**
7. **输入文法**

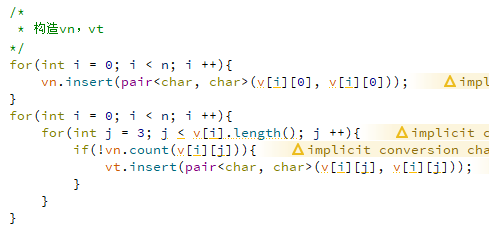
可以将每一条文法当做一个字符串来存储，由于之后会多次的遍历查找文法，所以这里用vector类型来顺序存储文法：



1. **构造Vn，Vt**

我们需要从产生式中剥离出来终结符和非终结符，由于之后会频繁判断某个字符是Vn还是Vt，所以这里可以用map类型保存Vn，Vt，且key=value=该字符，使得vn.count(ch)将直接能够判断字符ch是vn还是vt：

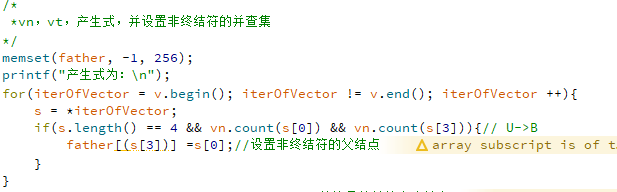




1. **构造非终结符的并查集**

在之后的规约过程中，我们需要知道规约到什么程度为止（例如，在举例文法中，字符i规约到F还是T还是E？），所以我们需要一种树状结构（类似于语法树），来判断规约的层级。这里我使用的是并查集：定义一个int数组father，由于字符的ASCII在0-256，所以该数组的大小取256即可；每当出现产生式：U:=V时，定义father[V]=U,要注意的是，开始符号的父结点为其自己：





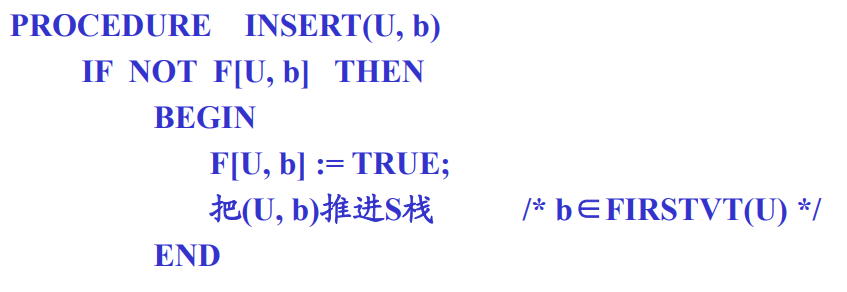


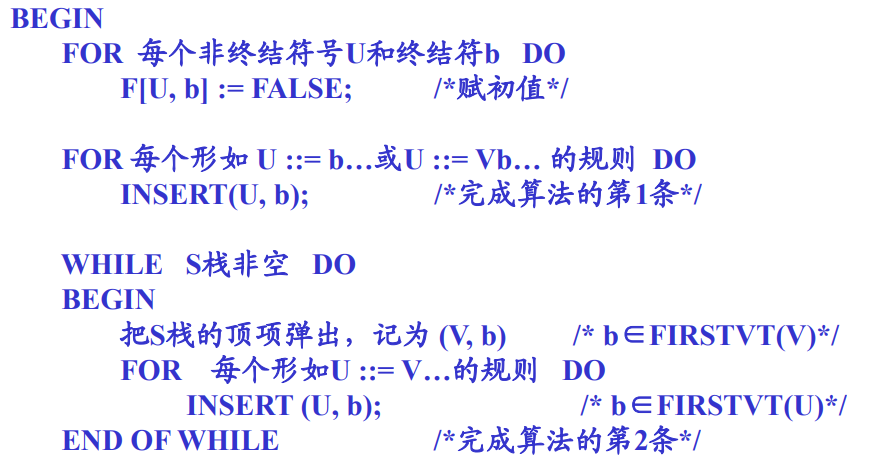
1. **构造firstvt和lastvt集**

考虑到firstvt集需要有两个下标（char，char），一个值（bool），数组会显的十分不友好，所以这里我使用multimap来存储，如果键值对<char1, char2>存在，则表明char2是char1的fristvt（lastvt同理）：



求firstvt集的伪代码如下：





要注意的是，在对出栈的（V，b）进行向上分析后，仍需将新出现的（U，b）压入栈中，代码如下（lastvt同理）：



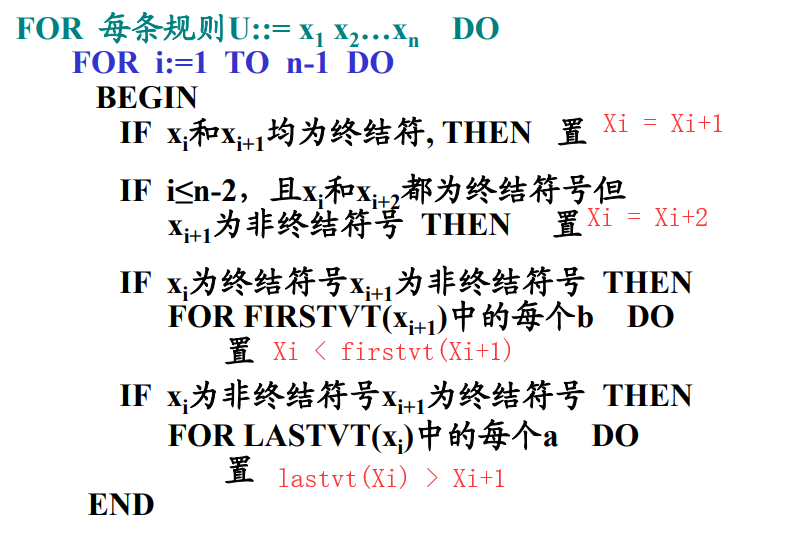
1. **构造算符优先矩阵**

类似于firstvt集的存储结构，这里依旧用multimap来存储，由于共有三种比较符号：<,>,=,所以需要用两个multimap存储：



如果vtOfStack中包含<char1, char2>,则表示 栈内终结符char1 > 栈外终结符 char2；

构造算符优先矩阵的伪代码如下：



注意在multimap中对同一key的键值对进行遍历时，可先声明一个迭代器：





此时It中first保存着键为key的第一条记录的迭代器，second保存着键为key的最后一条记录的迭代器，通过：





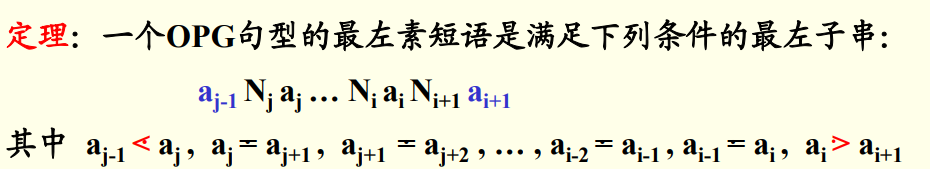
即可遍历；

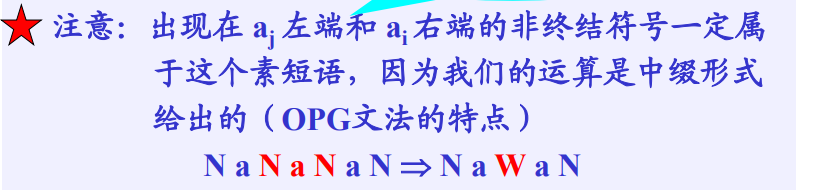
代码如下：



1. **获取当前待规约的最左素短语**

当需要对字符栈中的字符串进行规约时，我们首先需要找到其的最左素短语。伪代码如下：





由入栈出栈的规定可知，ai为栈最上面的终结符，所以我们从s1=ai开始向左递进，遇到终结符s2时则判断：如果s2=s1，则令s1=s2，继续向左递进；否则只能是s2<s1,这时s1即为aj，代码如下：



1. **规约最左素短语**

需要根据产生式来规约。通过循环来判断没个产生式的右部与最左素短语的比较情况：对每个对应位置的字符来说，如果都是终结符，则需要相等；如果都是非终结符，那么需要其根结点（最上层父结点）相同，这样判断是为了消除规约的层次问题；否则移近到下一个产生式。代码如下：



1. **分析规约输入的句子**

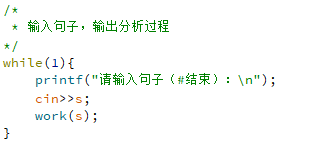
对输入句子的每个输入终结符进行判断：如果该终结符的优先级大于栈顶的终结符，则将其压入栈中，开始分析下一个输入终结符；如果该终结符优先级等于栈顶的终结符，则将其压入栈中，开始分析下一个输入的终结符；如果该终结符小于栈顶终结符，则寻找栈中的最左素短语并规约。注意，当栈中仅剩下一个非终结符且输入串分析完毕时，分析未必完成：如果该非终结符不是开始符号的子节点（即该非终结符不能规约到开始符号），则句子规约失败；否则，将该非终结符规约到开始符号，分析成功并结束。代码如下：





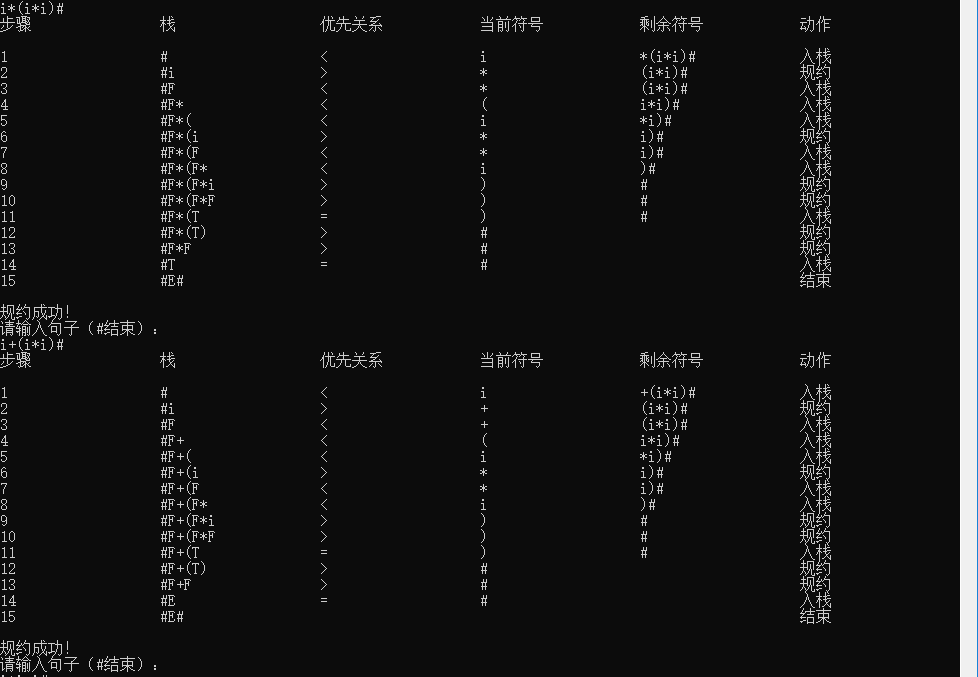
1. **输入句子**

这里注意一点：程序中将字符‘#’当做开始和结束符，所以文法中不能把‘#’当做终结符，并且在输入的句子末尾添加‘#’以表示句子结束：



1. **运行结果**







1. **实验问题与解决**
2. 以字符为下标的数据存储问题？

由于查找的需求，需要以字符为下标进行存储数据，简单的数组无法满足。STL中的一些类型可以很好的解决这个问题。Map可以存储唯一键值对<key,value>,multimap则可以同一key对应多个value，却允许重复键值对，vector也比常规的数组好用的多，并且可以通过他的push\_back()和pop\_back()函数，将其当做栈来使用。

1. Multimap查找键值对问题？

Multimap的键值对查找类似于二维数组的查找：先定义一个迭代器对：，用It去存储键为key的一系列键值对，且其中It.first为键为key的第一个迭代器，It.second为键为key的最后一个键值对，通过一层for循环即可遍历。

1. 最左素短语的规约层级问题？

当我们在规约时，如果我们规约方式为全等（即产生式的右部字符串等于最左素短语），当遇到有U:=V类型的产生式，下层规约到V还是U？在例题文法中，句子（i），素短语i如果规约到F，则（F）无法继续向上层规约（因为没有产生式的右部为（F））。所以需要有一个非终结符的父子节点概念，即规约时，对于产生式右部和最左素短语的对应位置字符，如果都为终结符则要求相等；如果都为非终结符，则要求根节点（最上层父节点）相同；否则换下一个产生式。这样（F）继续向上层规约为（E）、F等等。

1. **实验感想与收获**

**这次实验比词法分析程序更难一些，更多的考察一些逻辑操作。不过，在数据存储方面，用一些高级的结构实在是舒服太多了，STL之前一直不是太懂，借着这次机会好好地重学了一边，才发现原来我用简单的数组是多么傻。其次就要将书上的伪代码翻译好。**