编译原理

孙吉鹏　李鸣一
周　秋　袁郭苑

排版：宁睿 谢颖
版本号：V1.0
修订时间：　2018.12

山软智库-让知识回归平凡

目录

PS：

第七、十、十一章留待下个版本完成，请同学们先行展开复习。

第一章：引论

作者：周秋

要求：掌握**编译原理结构图**，能够说出它一共有哪几个部分，每部分都完成什么样的功能。

注：本章起到一个概览的作用，目的是帮助学习者从宏观上理解编译原理的工作过程，在这一章中不必纠缠于细节

首先介绍几个概念：翻译器、编译器和解释器

翻译器：又叫翻译程序，它能将一种语言程序(源语言程序)转换成另一种等价的语言程序(目标语言程序)。

编译器：又叫编译程序，能将一种计算机高级语言程序(源语言程序)转换成另一种等价的计算机低级语言程序(目标语言程序)，这里的源语言和目标语言都必须是计算机语言。

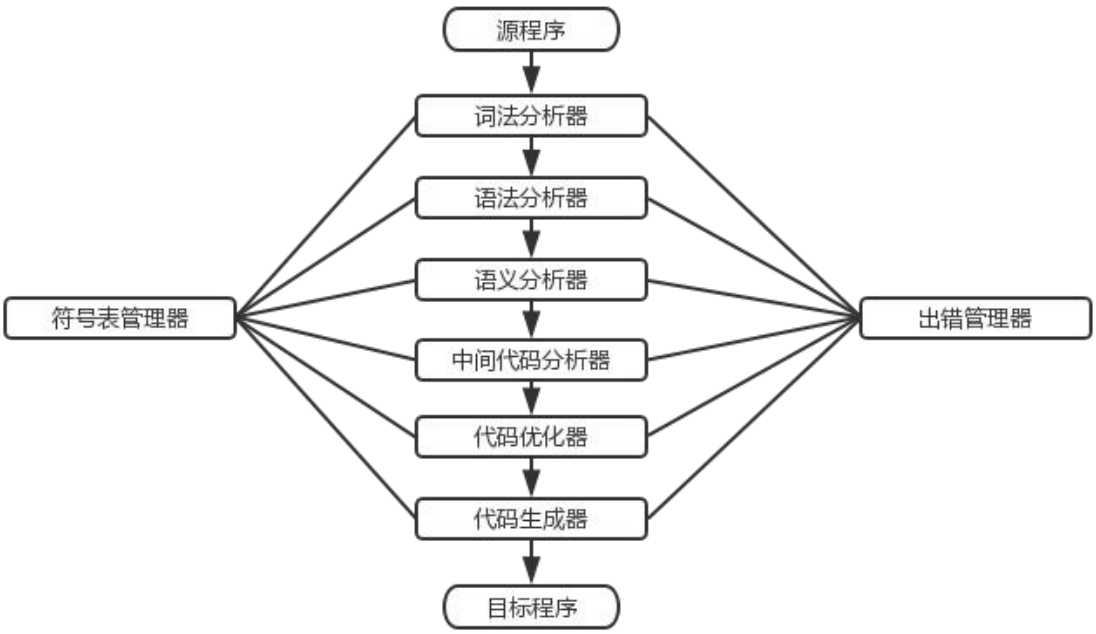
解释器：又叫解释程序，也是一种翻译程序,以一种语言写的源程序作为输入,但不产生目标代码,而是边解释边执行。

解释器与编译器的区别是：编译分成两步完成：先翻译，再运行；解释只用一步完成：边解释边执行

这门课要讲的就是**编译器（编译程序）**。

编译器从逻辑上可以分成若干阶段，每个阶段把源程序从一种表示变换成另一种表示，这门课通过描述编译器的各个阶段来介绍编译这个课题。

编译原理结构图：



下面就来介绍这个图中的几个过程，每个过程都是一个子程序，拥有自己的输入输出。

1.**词法分析器**

输入：源程序字符串　输出：单词符号表（单词类型和单词本身）

对字符串进行扫描和分解，在这个过程中要正确识别哪些内容属于同一个单词（比如大于等于号>=就是一个单词，而不是>和=）。使用正规式和有限自动机描述词法规则。

如果这个过程出错，表示源程序中有词法错误。

2.**语法分析器**

输入：单词符号表　输出：一系列语法单位（语法树）

按照早就定义好的语法，把单词符号划分为一系列语法单位，比如算术表达式、赋值语句、调用语句、子程序、总程序等。这些语法单位通常以语法树的形式呈现。

在第二章中可以看到语法树的示例。

如果这个过程出错，表示源程序中有语法错误。

3.**语义分析器**

输入：语法树　输出：包含语义（属性）的语法树

按照语义规则（通常是属性文法），检查各部分是否合乎语义，比如是否引用了未定义的量，是否调用了未定义的函数等，为生成中间代码做准备。

如果这个过程出错，表示源程序中有语义错误。

前三步都是对源程序进行严格的检查，确保没有词法、语法和语义错误，这样才能顺利进行下面的内容

4.**中间代码生成器**

输入：包含语义（属性）的语法树　输出：中间代码（四元式）

根据语法树上每个节点的属性值，结合语义规则，生成中间代码。

中间代码是一种通用的、简易的记录计算机语言的表达方式。

这一步开始才正式执行了翻译功能，即把源程序变成了意义相同的中间代码

5.**代码优化器**

输入：中间代码　输出：优化后的中间代码

这一步可以把过于冗长的中间代码进行缩减，并且保证意义不变。

6.**代码生成器**

输入：优化后的中间代码　输出：目标代码

这一步把中间代码转换成目标代码，目标代码可以直接被机器所运行。

为什么不直接把源程序翻译成目标代码：这是因为高级语言（m种）和机器语言（n种）都有很多种，两两配对的话编译器种类就太多了（m*n），借助简单的中间代码，就可以减少编译器的种类（m+n）。

至此，编译过程结束，高级语言的源程序变成了可被机器识别的目标程序

此外还有符号表管理器和出错管理器两个辅助程序

符号表管理器：记录了编译过程中的各种符号和表格。

出错管理器：用来处理错误，通常是词法、语法和语义错误。

几个上课中提到的小知识：

可变目标编译程序：产生不同于其宿主机（运行编译程序的机器）的机器代码的编译程序。

交叉编译程序：不需要重写编译程序中与机器无关的部分，就可以改变目标机（运行机器代码的机器）的编译程序。

编译前端：编译程序中与源语言有关但与目标机无关的部分，通常包括词法分析，语法分析，语义分析和中间代码产生。

编译后端：编译程序中与目标机有关的部分，通常包括与目标机有关的代码优化和目标代码生成。

遍：对源程序或者源程序的中间结果从头到尾扫描一次，做相应处理和加工，形成新的中间结果或者目标程序的过程。

第二章：高级语言及其语法描述

作者：周秋

要求掌握：

1. 掌握符号串、文法（四元式）、句型、句子和语言的定义。
2. 符号串的运算
3. 重要概念：递归、语法树、文法的二义性。
4. 文法的BNF表示和扩充的BNF范式、语法图。
5. 文法分类。

注：

了解一些之后的学习中必须用到的术语和概念，并形式化地描述高级语言的语法，如果在学习过程中感到枯燥或无法直观理解，可以在进行后面章节的学习后返回来查询相对应的概念

2.1 程序语言的定义及特性

程序语言的定义，包括语法、语义、语用三个方面的内容

语法：表示构成句子的各个记号之间的组合规律。

语义：表示各个记号的特定含义，它可以定义语言的意义。

语用：表示在各个记号所出现的行为中，它们的来源、使用和影响。

语言：特定字符集（字母表）上的一个由有限长度的字符串（句子）构成的集合，即语言是一个集合，其中每个元素都是一个句子。

程序语言：对于程序语言来说，它们也是定义在某个字母表上的句子的集合。这里的句子，就是一个源程序。

单词符号：源程序中的关键字、标识符、常数、运算符、界限符。

词法规则：单词符号的形成规则，它决定哪些单词符号是合法的。程序语言中，“<=”、“while”、“;”之所以是合法的单词符号，是因为程序语言的词法规则在起作用。

2.2 形式语言基础

形式语言：不考虑语义和语用，只从语法这一侧面来看的语言，抽象地定义为一个数学系统。

形式语言理论：对符号串集合的表示法、结构及其特性的研究，是程序设计语言语法分析研究的基础。

接下来介绍形式语言的几个概念：

字母表：以符号为元素的非空有限集合　例：Σ={a，b，c}
符号：字母表中的元素　　例：　a，b，c
符号串：由符号组成的有穷序列　　例：a, aa, ac, abc，……
空符号串：ε

符号串的形式定义：

对于给定的字母表Σ，定义：

（1）ε是Σ上的符号串；

（2）若x是Σ上的符号串，且a∈Σ，则ax或xa是Σ上的符号串；

（3）y是Σ上的符号串，当且仅当y可由（1）和（2）产生。

符号串和符号串集合的运算：

1.**符号串相等**：若x、y是集合上的两个符号串，则x＝y当且仅当组成x的每一个符号和组成y的每一个符号依次相等。

2.**符号串的长度**：x为符号串，其长度|x|等于组成该符号串的符号个数。

3.**符号串的联接**：若x、y是定义在Σ是上的符号串，且x＝XY，y＝YX，则x和y的联接 xy＝XYYX也是Σ上的符号串。

注意：一般xy≠yx，而εx＝xε

4. **符号串集合的乘积**：

令A、B为符号串集合，定义：AB＝{ xy |x∈A,y∈B}

例如：A={a,b},B={c,d},则AB={ac，ad，bc，bd}

注意：{ε}A=A{ε}=A

5. **符号串集合的幂**：有符号串集合A，定义：

A^0 ={ε},A^1=A,A^2=AA,　　A^3=AAA,……，$A^n=A^{n-1}A=AA^{n-1}$　，n>0

6.**符号串集合的闭包**：设A是符号串集合，定义：

集合A的正闭包：A^+ = A^1 ∪ A^2 ∪ A^3 ∪……∪ A^n ∪……

集合A的闭包：A* = A^0 ∪A^+

假设有一种语言，令A为它的基本字符集，B为它的单词符号集，C为它的句子集，则C⊂B*,B⊂A*

2.3 文法的直观理解

文法：也称作语法，是从形式上对语言结构的定义与描述，不涉及语义

语法规则：用来描述句子的语法结构，通常使用产生式。

比如汉语中，<句子> => <主语><谓语> 这就是一条产生式，“=>”的左边是它的左部，右边是它的右部，它规定了汉语中的一种句子类型。

产生式推导：通过下列方法可以获得句子

推导方法：

1.从一个要识别的符号开始推导

2.使用相应产生式的右部来替代左部

3.每次推导仅用一条产生式

4推导将一直进行，直到所有带< >的符号都由终结符号替代为止。

例2.1：在汉语中，“我是大学生”可由括号中的产生式推导得到

<句子> => <主语><谓语>

=> <代词><谓语> （<主语> => <代词>）

=> 我<谓语> （<代词> => 我）

=> 我<动词><宾语> （<谓语> => <动词><宾语> ）

=> 我是<宾语> （<动词> => 是）

=> 我是<名词> （<宾语> => 名词）

=> 我是大学生 （<名词> => 大学生）

上述推导可写成<句子> $\xrightarrow{+}$ 我是大学生

2.4 文法和语言的定义

根据上一节对文法的基本了解，这节来对文法进行定义

文法四元式：

文法G=（V_N，V_T，P，Z）

V_N ：非终结符集，所有非终结符的集合

V_T ：终结符集，所有终结符的集合

P：所有产生式（或规则）集合

Z：开始符号（$Z \in V_N$）

其中：

文法字汇表V：$V=V_N \cup V_T$

产生式：是一个有序对(U, x)，通常写为:U::=x或者U => x，其中$U \in V_N$，$x \in V^*$

非终结符：出现在产生式的左部，且能推出符号或符号串的符号

终结符：不出现在产生式的左部,且不能推出符号或符号串的符号

文法的BNF表示：

1.非终结符用<>括起；

2.每条规则的左部是一个非终结符，右部是由非终结符和终结符组成的一个符号串，中间一般以 ::= 隔开；

3.相同左部的产生式可以共用一个左部，把右部用 | 隔开。

直接推导：设x和y是符号串，若使用一次产生式可以从x推导出y，则称x直接推导出y，记为 $x \Rightarrow y$。

+推导：设x和y是符号串，若使用1次或若干次产生式可以从x推导出y，则称x +推导出y(或者说y是x的+推导），记为 $x \xrightarrow{+} y$。

***推导**：设x和y是符号串，若使用0次或若干次产生式可以从x推导出y，则称x *推导 出y(或者说y是x的*推导），记为 $x \xrightarrow{*} y$。

最右推导：若符号串α中有两个以上的非终结符时，对推导的每一步坚持把α中的最右非终结符进行替换。

最左推导：若符号串α中有两个以上的非终结符时，对推导的每一步坚持把α中的最左非终结符进行替换。

规范推导：最右推导。

规约：推导的逆过程，例如：x => y,可称为x直接推导出y，也可称为y直接归约成x。

规范归约：对句型中最左边的简单短语（句柄）进行的归约。

规范归约和规范推导互为逆过程。

等价文法：G和G'是两个不同的文法，若 L(G) = L(G')，（L(G)：文法G的语言，即使用文法G能推导出的全部句子）则G和G’为等价文法。

递归产生式：产生式右部有与左部相同的符号。

左递归：产生式 U → xUy，若x=ε,即U → Uy；

右递归：产生式 U → xUy，若y=ε,即U → xU；

递归文法：文法G，存在U $\in V_N$，U =>...U...，则文法G递归；

左递归文法：U==>U...；

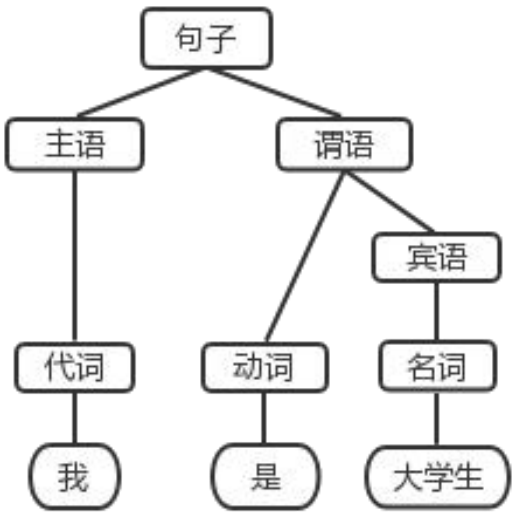
右递归文法：U==>...U；

递归文法的优点：可用有穷条产生式，定义无穷语言（包含无数个句子的语言）。

语法树：用来描述语法结构的工具，它可以记录使用产生式推导的过程。语法树上的非叶节点（语法成分）叫做非终结符，叶节点（单词符号）叫做终结符。

子树：语法树中的某个结点（子树的根）连同它向下派生的部分所组成。
简单子树：只有单层分枝的子树。

例2.1中“我是大学生”的语法树如下：



2.5 文法的类型

2.4节中所讲的形式语言，是用文法和自动机所描述的、没有语义的语言。

文法和语言分类：0型、1型、2型、3型，这几类文法的差别在于对产生式施加不同的限制。

类型	文法	语言	特征
0型	产生式的左部和右部都可以是符号串	L0：可以被图灵机接受	短语结构文法
1型	产生式P：xUy => xuy，其中U∈V_N， x、y、u∈V*	L1：这种语言可以由线性界限自动机接受	上下文有关文法，只有在x、y同时存在时才能把U改写为u
2型	产生式P：U → u，其中U∈V_N， u∈V*	L2：这种语言可以由下推自动机接受	上下文无关文法，与BNF表示相等价。
3型	左线性：产生式P：U → T或 U → wT，其中U、w∈V_N，T∈V_T* 右线性：产生式P：U → T或 U → Tw，其中U、w∈V_N，T∈V_T*	L3：又称正则语言、正则集合，这种语言可以由有穷自动机接受	正则文法，对2型文法进行进一步限制，能力较弱

根据上述讨论，L0⊂L1⊂L2⊂L3

0型文法可以产生L0、L1、L2、L3，但2型文法只能产生L2，不能产生L1。

2型文法（上下文无关文法）是我们之后主要学习的文法。

2.6 文法的二义性

句型：x是句型 ⇔ $Z \xrightarrow{*} x$，且x∈V*；

句子：x是句子 ⇔ $Z \xrightarrow{+} x$，且x∈V_T*；

语言：L(G[Z]) ={x| Z => x， x∈V_T* }；

规范句型：通过规范推导或规范归约所得到的句型。

二义性文法：若对于一个文法的某一句子存在两棵不同的语法树，则该文法是二义性文法，否则是无二义性文法。

或者说若一个文法的某规范句型的句柄不唯一，则该文法是二义性的，否则是无二义性的。

文法的二义性是不可判定的：即不可能构造出一个算法，通过有限步骤来判定任一文法是否有二义性。

文法二义性的解决：提出一些无二义性文法的充分条件。当文法满足这些条件时，就可以判定文法是无二义性的。

2.7 有关文法的限制

有害产生式:文法中有如U => U的产生式，它会引起二义性。

多余产生式:

1.在推导文法的所有句子中，始终用不到的产生式。即该产生式的左部非终结符不出现在任何句型中。

2.在推导句子的过程中，一旦使用了该产生式，将推不出任何终结符号串。即该产生式中含有推不出任何终结符号串的非终结符。

文法的其他表示法：

扩充的BNF表示：

BNF的元符号： <, >, →,|

扩充的BNF的元符号： <, >, ::=,|, {, }, [,] (,)

{}：表示可无限次追加{}中的内容

[]：表示可选[]中的内容，也可以不选

()：通常搭配|使用，(a|b|c)表示必须从a,b,c中选择一个

语法图：使用图示来表达扩充的BNF范式。（感觉不重要）

第三章：词法分析

作者：周秋

重点：

1.正规式，正规集，NFA，DFA

2.NFA的确定化

3.DFA的最小化

4.正规式与NFA的相互转化

5.正规文法与NFA的相互转化

3.1对于词法分析程序的要求

词法分析程序：实现词法分析的程序。

功能：输入源程序，输出单词符号或属性字

主要任务：从左到右扫描源字符串，逐个字符读入，按照构词规则，把源字符串切分成一个个单词，并确定其属性，再把它们转换成属性字。

属性：单词的分类。

属性字（TOKEN）：方便编译的、长度统一的标准形式，通常为(单词种别，单词符号的属性值)这样的二元式。

	含义	属性	属性值
关键字	由程序语言定义的具有固定意义的标识符。也称为保留字或基本字	一符一种	无
标识符	用来表示程序中各种名字的字符串	算作一种，ID	指向该标识符的指针
常数	常数的类型一般有整型、实型、布尔型、文字型等	算作一种，常数	指向该常数的指针
运算符	数字运算符、逻辑运算符等	一符一种	无
界限符	逗号、分号、括号等	一符一种	无

例3.1：C++代码段：while（i > =j）i - - ;
词法分析后转换为如下单词符号序列：
<while, - >
<id,指向i的符号表项的指针>
< >=, - >
<id,指向j的符号表项的指针>
<), - >
<id,指向i的符号表项的指针>
<--, - >
<;, - >

词法分析工作从语法分析工作独立出来的原因：

1. 简化设计
2. 改进编译效率
3. 增加编译系统的可移植性

词法分析器的设计:

输入子程序：输入源程序字符串放在输入缓冲区中。

预处理子程序：可以方便单词识别工作更好的进行

主要工作：

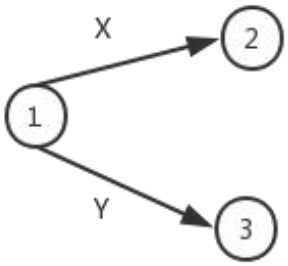
1. 剔除跳格符、回车符、换行符
2. 剔除注释
3. 把连续的空格结合成一个

识别单词符号:

超前搜索：在识别一个单词时，通常需要再读入它之后多个字符，才能确定该单词符号的种类。

状态转换图：一张有限方向图。在状态转换图中，结点代表状态,状态之间用箭弧连结。箭弧上的标记（字符）代表在射出结点状态下可能出现的输入字符或字符类。一张转换图只包含有限个状态（即有限个结点），其中有一个被认为是初态，而且实际上至少要有一个终态（用双圈表示），一个状态转换图可用于识别(或接受)一定的字符串，

例3.2：在状态1下，若输入字符为X，则读进X，并转换到状态2。若输入字符为Y，则读进Y，并转换到状态3。



例3.3：识别标识符的转换图如下图所示，其中0为初态，2为终态。

这个转换图的识别过程：

1.从初态0开始，若在状态0之下输入字符是一个字母，则读进它，并转入状态1；

2.在状态1之下，若下一个输入字符为字母或数字，则读进它，并重新进入状态1；一直重复这个过程，直到状态1发现输入字符不再是字母或数字时（这个字符也已被读进）就进入状态2；

3.状态2是终态，它意味着到此已识别出一个标识符，识别过程宣告终止。终态结上打个星号。意味着多读进了一个不属于标识符部分的字符，应把它退还给输入串；

4.如果在状态0时输入字符不为“字母”，则意味着识别不出标识符，或者说，这个转换图工作不成功。

3.2正规表达式与正规集（正规语言）

正规式：也称正则表达式，用来说明单词的模式，是定义正规集的数学工具。

正规式和它所表示的正规集的递归定义：

设字母表为Σ，辅助字母表Σ’ ={ε，ϕ，|，*，(，)}，

1.ε和ϕ都是Σ上的正规式，它们所表示的正规集分别为{ε}和ϕ；

2. 对任何a∈Σ，a是Σ上的一个正规式，它所表示的正规集为{a}；

3. 假定e1和e2都是Σ上的正规式，它们所表示的正规集分别为L(e1)和L(e2)，那么，(e1), e1·e2, e1|e2, e1*也都是正规式,它们所表示的正规集分别为L(e1), L(e1)·L(e2), L(e1)|L(e2)和(L(e1))*。

“|”读为“**或**”

“·”读为“**连接**”

“*”读为“**闭包**”，任意有限次（包括0次）的自重复连接

4. 仅由有限次使用上述三步骤而定义的表达式才是Σ上的正规式，仅由这些正规式所表示的集

合才是Σ上的正规集。

例3.4：令Σ={a，b}，Σ上的正规式和相应的正规集的例子有：

正规式	正规集
a	{a}
a|b	{a,b}
ab	{ab}
(a|b)(a|b)	{aa,ab,ba,bb}
a*	{ε,a,aa,aaa,……}

例3.5：自己试着写一下:

正规式	正规集
(a|b)*	
(a|b)*(aa|bb)(a|b)*	

正规式的等价性：若两个正规式e1和e2所表示的正规集相同,则说e1和e2等价,写作e1=e2。

例如： e1= (a|b)， e2 = b|a， e1= e2

正规式的代数规律：

1. U|V=V|U “或”交换律
2. U|(V|W)=(U|V)|W “或”结合律
3. (UV)W=U(VW) “连接”结合律
4. U(V|W)=UV|UW (V|W)U=VU|WU 分配律
5. εU=U，Uε=U ε连接恒等律

3.3有穷自动机

有穷自动机：能准确地识别正规集（即识别正规文法所定义的语言和正规式所表示的集合）的识别装置，是词法分析程序的方法和工具。

有穷自动机分为两类：确定的有限自动机DFA和不确定的有限自动机NFA。

3.3.1 确定的有穷自动机DFA

DFA定义：

一个确定的有穷自动机（DFA）M是一个五元组：

M=（K，Σ，f，s0 ，Z）,其中:

1. 状态集K：一个有穷集，它的每个元素称为一个状态；

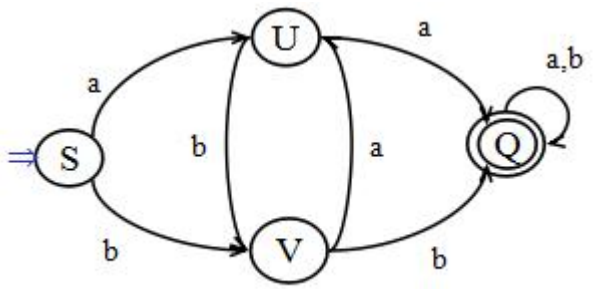
2. 输入符号表Σ：一个有穷字母表，它的每个元素称为一个输入符号；
3. 转换函数f：在K×Σ→K上的单值部分映射，即，如果 f（ki，a）=kj，（ki∈K，kj∈K）就意味着，当前状态为ki，输入符为a时，将转换为下一个状态kj，我们把kj称作ki的一个后继状态；
4. s0 ∈K：唯一的一个初态；
5. Z⊂K：一个可空终态集，终态也称可接受状态或结束状态。

一个DFA可以表示成一个状态转换图，一个状态转换图可以使用一个状态转换矩阵来表示。

例3.5：
DFA M=（{S，U，V，Q}，{a，b}，f，S，{Q}）
其中f定义为：
f（S，a）=U　　f（V，a）=U
f（S，b）=V　　f（V，b）=Q
f（U，a）=Q　　f（Q，a）=Q
f（U，b）=V　　f（Q，b）=Q
这个DFA可表示成如下转换矩阵：（初态节点带+号，终态节点带-号）

状态	a	b
+S	U	V
U	Q	V
V	U	Q
-Q	Q	Q

这个DFA还可以表示成如下状态转换图：



字符串α可为DFA M所识别（读出或接受）： 对于Σ*中的任何字符串α，若存在一条从初态结点到某一终态结点的通路，且这条通路上所有弧的标记符连接成的字符串等于α，则称α可为DFA M所识别（读出或接受）；若M的初态结点同时又是终态结点，则空字ε可为M所识别（或接受）。
例如：若t∈Σ* ，f (S，t) =P，其中S为DFA M的初态，P∈Z，Z为终态。则称t 可为DFA M所

接受（识别）。

L(M)：DFA M所能识别的字的全体的集合。

如果一个DFA M的输入字母表为Σ，则我们也称M是Σ的一个DFA。

Σ上的一个字集V∈Σ*是正规的，当且仅当存在Σ上的DFA M，使得V＝L(M)。

例3.6：证明字符串baab被例3.5中介绍的DFA所接受。

f（S，baab）=f（f（S，b），aab）

= f（V，aab）= f（ f（V，a），ab）

= f（U，ab）=f （ f（U，a），b）

= f（Q，b）=Q

Q属于终态。得证。

DFA的确定性的表现：

1. 映射f：K*Σ→ K是一个单值函数。也就是说，对任何状态s∈K和输入符号a∈Σ，f（s，a）唯一地确定了下一状态。从转换图的角度来看，假定字母表Σ含有n个输入字符，那么，任何一个状态结最多只有n条弧射出，而且每条弧以一个不同的输入字符标记。

2. DFA有且仅有唯一的一个初态s0∈K 。

3.3.2 不确定的有穷自动机NFA

NFA定义：

一个非确定的有穷自动机（NFA）M是一个五元组：

M=（K，Σ，f，S ，Z）,其中:

1. 状态集K：一个有穷集，它的每个元素称为一个状态；

2. 输入符号表Σ：一个有穷字母表，它的每个元素称为一个输入符号；

3. 转换函数f：在K×Σ*→K的子集的映射，即，f: K×Σ*→2^K ；表明在某状态下对于某输入符号可能有多个后继状态；

4. S⊂K：一个非空初态集；

5. Z⊂K：一个可空终态集，终态也称可接受状态或结束状态。

像DFA一样，NFA也能表示成状态转换矩阵和状态转换图。

例3.7：

NFA M=（{S，P，Z}，{0，1}，f，{S，P}，{Z}）

其中 ：

f（S，0）={P}

f（Z，0）={P}

f（P，1）={Z}

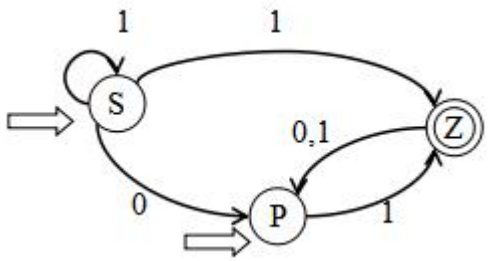
f（Z，1）={P}

f（S，1）={S，Z}

状态转换矩阵：可以看出在当前输入符号确定时，NFA指向的下一个状态不唯一

	0	1
+S	P	S,Z
P		Z
-Z	P	P

状态转换图表示如下：



字符串t可为NFA M所识别(读出或接受)：对于Σ*中的任何一个串t，若存在一条从某一初态结到某一终态结的道路，且这条道路上所有弧的标记字依序连接成的串(不理睬那些标记为ε的弧)等于t，则称字符串t可为NFA M所识别(读出或接受)。若M的某些结既是初态结又是终态结，或者存在一条从某个初态结到某个终态结的道路,其上所有弧的标记均为ε，那么空字可为M所接受。

L(M)：NFA M所能接受的符号串的全体。

Σ上一个符号串集V⊂Σ*是正规的，当且仅当存在一个Σ上的NFA M，使得V=L(M)。

3.3.3 NFA的确定化

具有ε转移的NFA可以转化为不具有ε转移的NFA：对任何一个具有ε转移（有向弧上为ε）的NFA N，一定存在一个不具有ε转移的NFA M，使得L(M)=L(N)。

DFA和NFA的等价性：对于每个NFA M，存在一个DFA M’ ，使得L(M) =L(M’)。即NFA M与DFA M’ 等价。与某一NFA等价的DFA不唯一。

NFA到相应的DFA的构造的基本思路是：DFA的每一个状态对应NFA的一组状态。DFA使用它的状态去记录在NFA读入一个输入符号后可能达到的所有状态。

构造NFA的状态K的子集的形式算法：（阅读即可）

假定所构造的子集族为C，即C= (T_1, T_2,... T_l),其中T_1, T_2,..., T_l为状态K的子集。

开始，令ε-closure(K_0)为C中唯一成员，并且它是未被标记的。

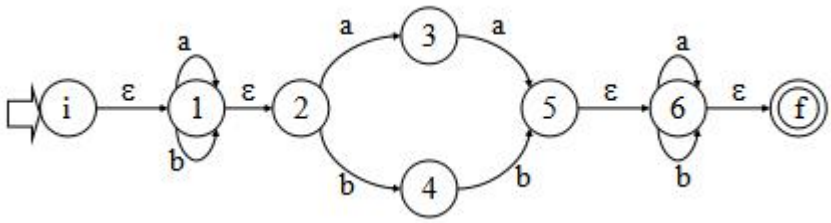
```
  while (C中存在尚未被标记的子集T)do
{   标记T;
    for 每个输入字母a   do
    {   U:=ε-closure(move(T,a));
        if   U不在C中   then
                将 U作为未标记的子集Ti加在C中
    }
}
```

ε-closure：通过ε有向弧可以到达的状态的集合

例3.8：使用上述算法

现在有如下图所示的NFA：

K={i,1,2,3,4,5,6,f}, Σ={a,b}, S_0=i, Z=f



1.把初始状态i的ε-closure：c_1={i,1,2}加入C(从i出发可由有向弧ε到达的状态号的集合)；

2.从C中取出未标记的c_1：查看字母表Σ中的字符

（1）因为从c1的状态i,1,2中经过有向弧a可以一步到达的状态有1,3(i没有出去的有向弧a，1经过有向弧a到达1，2经过有向弧a到达3)，而1,3经过有向弧ε可以到达的状态又有2（1经过有向弧ε到达2，3没有出去的有向弧ε），所以c2={1,2,3},又因为c2不在C中，所以把c2加入C；

（2）因为从c1的状态i,1,2中经过有向弧b可以一步到达的状态有1,4(1经过有向弧b到达1，2经过有向弧b到达4)，而1,4经过有向弧ε可以到达的状态又有2(1经过有向弧ε到达2)，所以c3={1,2,4},又因为c3不在C中，所以把c3加入C；

3.从C中取出未标记的c_2：查看字母表Σ中的字符

（1）因为从c2的状态1,2,3中经过有向弧a可以一步到达的状态有1,3,5(1经过有向弧a到达

1，2经过有向弧a到达3，3经过有向弧a到达5)，而1,3,5经过有向弧ε可以到达的状态又有2,6,f(1经过有向弧ε到达2，5经过有向弧ε到达6,f)，所以c4={1,2,3,5,6,f},又因为c4不在C中，所以把c4加入C；

（2）因为从c2的状态1,2,3中经过有向弧b可以一步到达的状态有1,4(1经过有向弧b到达1，2经过有向弧b到达4)，而1,4经过有向弧ε可以到达的状态又有2，所以c5={1,2,4},又因为c5=c3，所以c5不加入C；

4.从C中取出未标记的c_3：查看字母表Σ中的字符

（1）因为从c3的状态1,2,4中经过有向弧a可以一步到达的状态有1,3，而1,3经过有向弧ε可以到达的状态又有2，所以c5={1,2,3},又因为c5=c2，所以c6不加入C；

（2）因为从c3的状态1,2,4中经过有向弧b可以一步到达的状态有1,4,5，而1,4,5经过有向弧ε可以到达的状态又有2,6,f，所以c5={1,2,4,5,6,f},又因为c5不在C中，所以把c5加入C；

5.从C中取出未标记的c_4：查看字母表Σ中的字符

（1）因为从c4的状态1,2,3.5,6,f中经过有向弧a可以一步到达的状态有1,3,5,6，而1,3,5,6经过有向弧ε可以到达的状态又有2,6,f，所以c6={1,2,3.5,6,f},又因为c6=c4，所以c6不加入C；

（2）因为从c4的状态1,2,3.5,6,f中经过有向弧b可以一步到达的状态有1,4,6，而1,4,6经过有向弧ε可以到达的状态又有2,f，所以c6={1,2,4,6,f},又因为c6不在C中，所以把c6加入C；

6.从C中取出未标记的c_5：查看字母表Σ中的字符

（1）因为从c5的状态1,2,4,5,6,f中经过有向弧a可以一步到达的状态有1,3,6，而1,3,6经过有向弧ε可以到达的状态又有2,f，所以c7={1,2,3,6,f},又因为c7不在C中，所以把c7加入C；

（2）因为从c5的状态1,2,4,5,6,f中经过有向弧b可以一步到达的状态有1,4,5,6，而1,4,5,6经过有向弧ε可以到达的状态又有2,6,f，所以c8={1,2,4,5,6,f},又因为c8=c5，所以c8不加入C；

7.从C中取出未标记的c_6：查看字母表Σ中的字符

（1）因为从c6的状态1,2,4,6,f中经过有向弧a可以一步到达的状态有1,3,6，而1,3,6经过有向弧ε可以到达的状态又有2,f，所以c8={1,2,3,6,f},又因为c8=c7，所以c8不加入C；

（2）因为从c6的状态1,2,4,6,f中经过有向弧b可以一步到达的状态有1,4,5,6，而1,4,5,6经过有向弧ε可以到达的状态又有2,6,f，所以c8={1,2,4,5,6,f},又因为c8=c5，所以c8不加入C；

8.从C中取出未标记的c_7：查看字母表Σ中的字符

（1）因为从c7的状态1,2,3,6,f中经过有向弧a可以一步到达的状态有1,3,5,6，而1,3,5,6经过有向弧ε可以到达的状态又有2,6,f，所以c8={1,2,3,5,6,f},又因为c8=c5，所以c8不加入C；

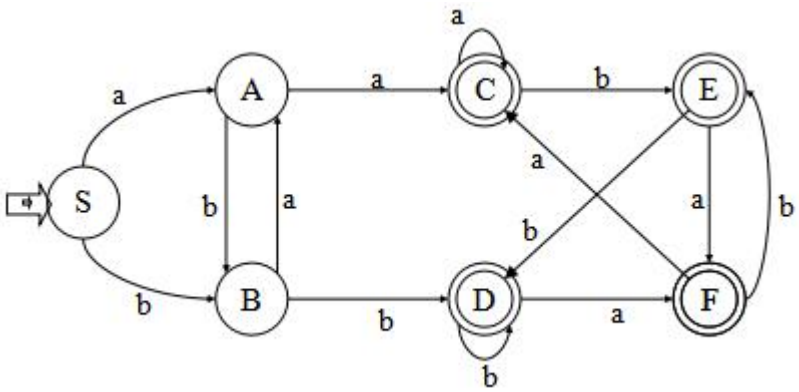
（2）因为从c7的状态1,2,3,6,f中经过有向弧b可以一步到达的状态有1,4,6，而1,4,6经过有向弧ε可以到达的状态又有2,f，所以c8={1,2,4,6,f},又因为c8=c6，所以c8不加入C；

至此，C中元素全部标记，算法停止。

下表就是刚才我们算法运算的过程，可以对照表格再理解一遍。

I	Ia	Ib
{i,1,2}	{1,2,3}	{1,2,4}
{1,2,3}	{1,2,3,5,6,f}	{1,2,4}
{1,2,4}	{1,2,3}	{1,2,4,5,6,f}
{1,2,3,5,6,f}	{1,2,3,5,6,f}	{1,2,4,6,f}
{1,2,4,5,6,f}	{1,2,3,6,f}	{1,2,4,5,6,f}
{1,2,4,6,f}	{1,2,3,6,f}	{1,2,4,5,6,f}
{1,2,3,6,f}	{1,2,3,5,6,f}	{1,2,4,6,f}

因为刚才在C中一共有7个元素，所以给DFA设置了7个状态节点，根据上表可以画出DFA状态转移图：



其中S表示c1，A表示c2,B表示c3，C表示c4,D表示c5，E表示c6,F表示c7，S因为包含原NFA的初态i，所以它在DFA中是初态；
CDEF因为包含原NFA的终态f，所以它们在DFA中都是终态；
S与A之间的有向弧a：根据上表，c1经过有向弧a的ε-closure为c2,所以S指向A。
其他有向弧的解释以此类推。

3.3.4 DFA的最小化

DFA最小化：就是寻求最小状态DFA’ ，它与原DFA等价，但是拥有最小的状态数。

最小状态DFA：

1.没有多余状态(任何输入串都无法到达的状态，又叫死状态)；

2.没有两个状态是互相等价（即从任何两个状态出发导出的串的集合都不相同）

在例3.8得到的DFA中，C和D能够导出相同的符号串集合{a,aa,ab,……},所以CD等价。

DFA的最小化算法：

形式化的算法书上或者课件上有，不再赘述。

下面写的是直观上的算法，不太严谨，但是可以帮助理解。

例3.9：以例:3.8上面得到的DFA为例：

1.把DFA中的状态按照终态、非终态分成两个集合：I1={S,A,B}，I2={C,D,E,F}；

2.随便选择一个集合，这里我们选择集合I1，考察它们：

S,A,B沿着有向弧a到达的状态分别是：A,C,A，而A和C分别属于两个不同集合I1和I2，所以应该把I1再分成两个集合{S,B}和{A}。

所以现在有三个集合：I1={S,B}，I2={A}，I3={C,D,E,F}

3.随便选择一个集合，这里我们选择集合I1，考察它们：

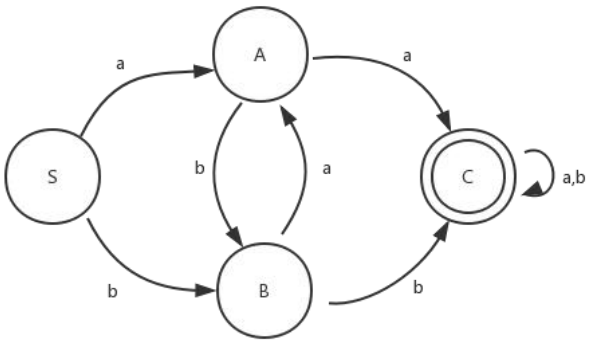
S,B沿着有向弧a到达的状态是A，而沿着有向弧b到达的状态分别是B和D，而B和D分别属于两个不同集合I1和I3，所以应该把I1再分成两个集合{S}和{B}。

所以现在有四个集合：I1={S}，I2={A}，I3={B}，I4={C,D,E,F}

4.随便选择一个集合，因为I1，I2和I3已经不可再分了，所以这里我们选择集合I4，考察它们：

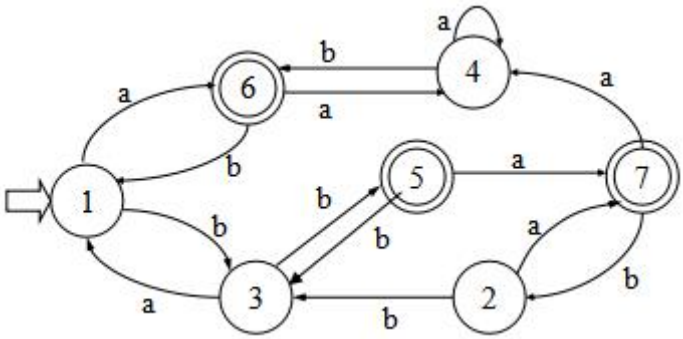
C,D,E,F沿着有向弧a到达的状态分别是C,F,F,C，而C和F属于同一个集合I4，而沿着有向弧b到达的状态分别是E,D,D,E，而E和D也属于同一个集合I4，所以I4不可再分

所以现在有四个集合：I1={S}，I2={A}，I3={B}，I4={C,D,E,F}，这四个集合都不可再分，现在可以得到下图：

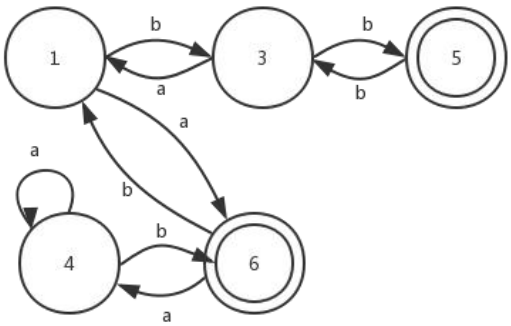


其中C代表了刚才的不可再分集合{C,D,E,F}，把原图中指向D,E,F的有向弧全部指向C；
该图即为最小状态DFA。
算法结束。

例3.10：



可以仿照刚才的算法，求出上图的小状态DFA
答案为：可得到不可再分集合为{12}，{3}，{4}，{5}，{6,7}，所以得到下图所示的最小状态DFA：

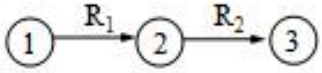
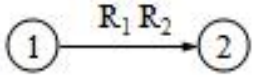


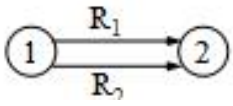
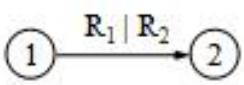
3.4正规式与有穷自动机的的等价性

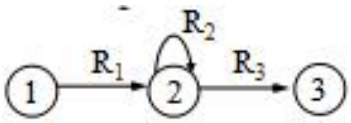
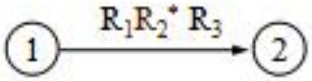
对于Σ上的NFA M，可以构造一个Σ上的正规式R，使得L（R）=L（M）。
对于Σ上的任一个正规式R ，可以构造一个Σ上的NFA M ，使得L（M）=L（R）。

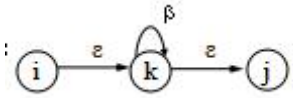
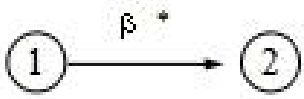
Σ上的NFA M转换为一个Σ上的正规式R：

对NFA重复下面的运算，直到只剩初态和终态节点；

1.对于  使用  来替代；

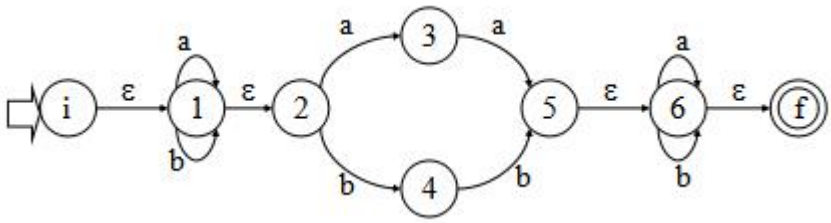
2.对于  使用  来替代；

3.对于  使用  来替代；

4.特别地，对于  使用  来替代；

最后弧上的就是所求的正规式R。

例3.11：比如例3.8中的NFA：



得到的正规式为：(a|b)*(aa|bb)(a|b)*

Σ上的正规式R转换为一个Σ上的NFA M：

对于一个正规式，使用上面的运算的逆运算即可得出NFA

3.5正规文法与有穷自动机的等价性

对于给定的正规文法G[R]，可以直接构造一个NFA M，使得L（M）=L（G）。

对于Σ上的任一个NFA M ，可以直接构造正规文法G[R] ，使得L（ R ）=L（ M ）。

把给定的正规文法G[R]转换为一个Σ上的NFA M构造规则:

设G[R]=（V_N，V_T，P，R），NFA M=（ K，Σ，f，S，Z）

1.令 Σ= V_T；

2. K = V_N，S = R；即对G中的每个非终结符生成M的一个状态（不妨取相同的名字，G的开始符号是M的初态；

3.增加一个新状态Z，作为NFA M 的终态，Z∈K；

4.对G中的形如A => tB（其中t为终结符或ε；A和B为非终结符）的产生式，

构造M的一个转换函数f（A，t）=B；

5.对G中的形如A => t（其中t为终结符或ε；A和B为非终结符）的产生式，

构造M的一个转换函数f（A，t）=Z。

把给定的Σ上的NFA M转换为一个正规文法G[R]的构造规则:

设NFA M=（ K，Σ，f，S，Z），G[R]=（V_N，V_T，P，R），

1.令 V_T = Σ ；

2.令V_N =K即对G中的每个非终结符生成M的一个状态（不妨取相同的名字，G的开始符号是M的初态；

3.令S=R（如果M有多个初态，应先拓广自动机，引入新初态x）；

4.对M 的终态Z增加一个产生式： Z => ε；

5.对M的每一个转换函数f（A，t）=B可写G的一个产生式A => tB（其中t为终结符或ε；A和B为非终结符）；

3.6词法分析程序的自动构造

把自动机表示成程序，可以进行词法分析。

第四章 自顶向下的语法分析

作者：李鸣一

章节结构

章节背景：

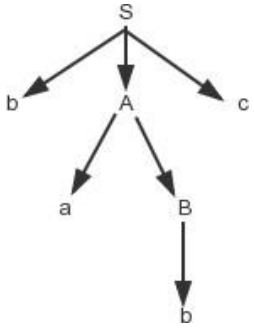
- 语法分析的主要工作：
  识别由词法分析给出的单词序列是否是给定的正确句子（程序）
- 高级程序设计语言的文法是**二型文法**，即上下文无关的文法

$$L(G(Z)) = \{x | Z^* \Rightarrow^+ x, x \in V_t^*\}$$

自顶向下分析思想

- 方法：
  从文法开始的符号（一般情况下是“<程序>”）开始进行分析，逐渐推导的往下构造语法树，使其树叶构造所给定的源程序串。
- 主要思想：从开始符号出发，先利用产生式把非终结符全部推导为终结符，再和输入符号串中的非终结符，如果全部匹配成功，则表示开始符号可以推导出给定的输入符号串，因此判定符号串是符合文法的句子

eg：

> 有文法G：
>
> $$\begin{matrix} S \to bAc \\ A \to aB \end{matrix}$$
>
> $$B \to b$$
>
> 和输入串 w = babc是否是该文法对应的语言的句子我们从文法开始符号出发推导出一个终结符的符号串
>
> 
>

而这个符号串正好和w相等，所以，w是满足该文法自顶
向下分析的关键问题：确定候选产生式的问题

- 当进行推导时，一个非终结符可能对应多个产生式，这样我们就无法事先知道应该用哪个产生式，因此必须对文法作一些限制，以便在任何情况下都能确定应该用的产生式。

  eg:

  $$A \to bAd$$
  $$A \to cAd$$

  上述两个产生式在推导过程中，对于非终结符 A 的推导都是可选的产生式，如果不对文法进行限制，那么产生式的选择成为无根据的，只好一一去试所有可能的产生式，直至成功为止。这种回溯的方法实际上是穷举的过程，效率低代价高，很少使用。

候选确定问题的解决方案

问题1

若有文法G1[S]:

$$S \to pA$$
$$S \to qB$$
$$A \to cAd$$
$$A \to a$$

和输入串 w = pccadd，判断符号串是否是文法G的符号串 推导过程为： S⇒ pA ⇒ pcAd ⇒ pccAdd ⇒ pccadd 推导过程中依次使用了产生式1、3、3、4来展开加了下划线的非终结符，最终得到的终结符串与输入串w匹配，说明w可以由文法开始符号推导出，也就是说w是文法的符号串

此时，虽然产生式A和S有两个候选，但是每个候选都是以终结符开头，那么直接选择右部开头的符号和输入符号串 中当前应该匹配的符号一致的候选即可

若有文法G2[S]:

$$S \to Ap$$
$$S \to Bq$$
$$A \to a$$
$$A \to cA$$
$$B \to b$$

在推导 A, S 时有多个候选，但是产生式的右部不在以非终结符开头，然而我们依然可以根据当前符号来判断，因为$A \Rightarrow^{+} \beta$ 中β 的最左部的终结符一定需要和当前输入符号匹配，但是这时由于β可能有多个，所以这时最左部的终结符也可能有多个，构成一个集合

- FIRST集

  设$G = (V_N, V_T, P, S)$是上下文无关文法，α是G的任一符号串，则有：

  $$FIRST(\alpha) = \{a | \alpha \Rightarrow^{*} a\beta,\ a \in V_T,\ \alpha, \beta \in V^{*}\}$$

  特别地，若$\alpha \Rightarrow^{*} \epsilon$，则规定$\epsilon \in FIRST(\alpha)$，即： FIRST(α)是从α出发推导出的所有符号串首终结符或可能的ε构成的集合。

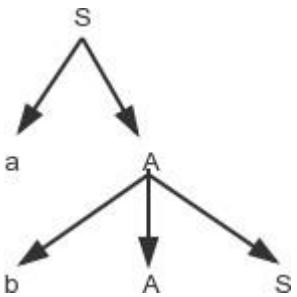
- FIRST集求法：递归构造过程

  FIRST(ε) = {ε}

  - FIRST(终结符) = {那个终结符自己}
  - $X \in V_N$，且有产生式$X \rightarrow a \ldots, (a \in V_T)$，则 $a \in FIRST(X)$
  - $X \in V_N$，且有产生式$X \rightarrow \epsilon$，则 $\epsilon \in FIRST(X)$透风墙原理：
  - 若$X, y_1, y_2, .. y_i \in V_N$，且有产生式$X \rightarrow y_1 y_2 .. y_i$，从左到右，如果一个$y_1 \Rightarrow^{*} \epsilon$，那么 $FIRST(X) = FIRST(y_2 y_3 \ldots y_i) \cup (FIRST(y_1) - \{\epsilon\})$，依次类推
    - 先看看“墙透不透风” $y_i(\Rightarrow^{*} \epsilon)$
      - 透风的话 $FIRST(X)^{+=} FIRST(y_i) - \{\epsilon\}$
      - 不透风的话 $FIRST(X)^{+=} FIRST(y_i)$，构造结束
    - 所有墙都透风时，才把 ε加入FIRST(X)

问题2

> 若有文法G3[S]
>
> $$S \rightarrow aA$$
> $$S \rightarrow d$$
> $$A \rightarrow bAS$$
> $$A \rightarrow \epsilon$$
>
> 和输入串w = abd，判断w是否满足该文法按照正常分析，我们可以做到这一步：



也就是S $\Rightarrow$ abAS。这里我们很容易判断A应该选择候选ε，因为如果想要和w匹配，那么就需要 $AS \Rightarrow d$，而我们已知S有d这个候选，那么A就理所应当地选择ε这个候选。

更一般地，我们在A各个候选的FIRST集中没找到当前的输入符号，而是在紧跟在A后面的符号串的FIRST集中找到了，所以我们判断A这个符号被跳过了，也就是选择了ε这个候选

但是我们不可以因为A所有候选的FIRST集中没有当前输入符号就判断A选择了ε候选，因为我们需要区分ε候选和语 法错误!

- FOLLOW集：“所有可能出现在 A后面的终结符”
  设$G = (V_N, V_T, P, S)$是上下文无关文法$A \in V_N$，则有：

$$FOLLOW(A) = \{a | S \Rightarrow^* \mu A \beta, a \in FIRST(\beta), \mu, \beta \in V*\}$$

  特别地，若$S \Rightarrow^* \ldots A$，那么 $\# \in FOLLOW(A)$
  - 这里S指<程序>，#是输入符号终结的标志，可理解为EOF
- FOLLOW集地构造：递归构造过程
  - 对文法开始符号S， $\# \in FOLLOW(S)$
  - 若$B \to \alpha A \beta$是一个产生式，则
    $FIRST(B) - \{\epsilon\} \subseteq FOLLOW(A)$
  - 守门员福利：
    若$B \to \alpha A$是一个产生式，或者 $B \to \alpha A \beta$中$\epsilon \in FIRST(\beta)$，则 $FOLLOW(B) \subseteq FOLLOW(A)$
    - 什么是守门员
      - 产生式右部位置在末尾的非终结符
      - 产生式右部的一个终结符，它的右边全是“透风墙”

> FIRST集和FOLLOW集构造算法的细枝末节
>
> - FIRST集的括号里是任意符号串，而FOLLOW集的括号里只能是一个非终结符
> - FIRST集的构造过程是从产生式左部到右部，“沿语法树向下”包含的过程；
>   FOLLOW集的构造过程是从产生式右部到左部，“沿语法树向上”包含的过程
> - FOLLOW集中有特殊的“#”，FIRST集中有特殊的“ε”
> - FOLLOW集的构造过程需要构造FIRST集

SELECT集

看完两个以符号或符号串为参数的算法后，我们把目光放回最初的问题—我们在语法分析中如何确定使用哪一个 候选？在编译器上下文中，我们已经掌握的知识有：

- 当前需要推导的非终结符
- 输入串中当前需要被匹配的符号
- 所有的产生式

我们可以轻松地把候选范围从“所有产生式”缩小到“以某非终结符为左部的产生式”，但是还不够，利用上面两个集合，就可以解决这个问题

对于产生式 $A \to$<一堆符号>

- 如果当前输入符号在FIRST(<一堆符号>)中，那么我们选这个产生式
- 如果FIRST(<一堆符号>)中有ε（也就是说<一堆符号>透风），那么我们需要算出FOLLOW(A)，如果当前输入符号在FOLLOW(A)中，我们也选这个产生式

这个“一定选这个产生式”集合称为产生式的SELECT集，可以描述为

$$SELECT(A \to \beta) = \begin{cases} FIRST(\beta), & \epsilon \in FRIST(\beta) \\ FIRST(\beta) \cup FOLLOW(A), & \epsilon \notin FRIST(\beta) \end{cases}$$

可以看到，只要求出所有产生式的SELECT集，我们只需要看SELECT集，就可以在左部相同后选中选出正确的产生式。

LL(1)文法

我们已经知道了如何使用（穷举之外的）形式化的方法从候选中选择一个产生式进行语法分析，但是还没有涉及前面提到的**“对语法进行限制”**。

很容易想到，我们之所以可以这样的选择，必然是因为当前字符肯定包含在一个产生式的SELECT集中，但是一旦这个符号也被另一个产生式的SELECT集包含，即 $FIRST(A \to \beta_1) \cap FIRST(A \to \beta_2) \neq \phi$ 那我们上述构造集合的工作将失去意义。

所以这就是**“限制”**，进行自顶向下分析的文法一定要保证所有左部相同的产生式SELECT集候选不相交，即 $FIRST(A \to \beta_1) \cap FIRST(A \to \beta_2) = \phi$，满足这个条件的文法，我们称之为LL(1)文法。

具体而言，这个限制体现在两个方面：

- 左部相同的产生式候选的右部不能有公共前缀，即不能存在左公因子
  - 原因很简单，如果有公共前缀，那两者的FIRST集必然有重合
- 产生式不能存在左递归，例如 $A \to A\beta$
  - 根据FIRST集的构造算法，这样的产生式会使算法陷入递归

所以可以断言，只要文法中存在上述两种状况，那么该文法一定不是LL(1)文法

定义：

- 一个上下文无关文法是LL(1)文法的充分必要条件是每个非终结符 A 的两个不同产生式，$A \to \alpha, A \to \beta$，满足

$$SELECT(A \to \alpha) \cap SELECT(A \to \beta) = \phi$$

- LL(k)文法是采取确定的自左至右扫描（输入串）和自顶向下分析技术的最大一类文法。LL系指：自左向右扫描（输入串），自上而下进行最左推导。

自顶向下文法分析程序的构造

自顶向下的分析程序有两种主流的构造方法：

- 递归下降分析法
- 预测分析法

递归下降分析程序

我们知道，自顶向下的文法分析是要从文法开始符号起，选择一个候选产生式并依次分析候选产生式中的非终结符。由于文法是递归定义的，分析过程也可以看成是一个递归的过程，递归下降就采用了最为直接的思维方式构造文法分析程序

递归下降法的主要思想是：

对每个非终结符按其产生式结构写出相应语法分析子程序。因为文法递归相应子程序也递归，子程序的结构与产生式结构一致。所以称此种方法称为递归子程序法或递归下降法。

eg：对同为非终结符 A 候选产生式

$$A \to \beta_1 d$$
$$A \to \beta_2 \gamma e$$
$$\cdots$$

```
cur = {分析程序当前扫描到的符号}
function match({任意终结符t})
{
    if cur == t then:
        cur = {从输入符号串读入下一个符号}
    else:
        {报告语法错误}
}

procedure A()
{
    //对于不同候选，利用select集进行选择，然后依次处理右部符号
    if cur in SELECT(A -> β1d) then:
        call β1();
        match('d');
        return;
    if cur in SELECT(A -> β2γe) then:
        call β2();
        call γ();
        match('e');
        return;
    ...
```

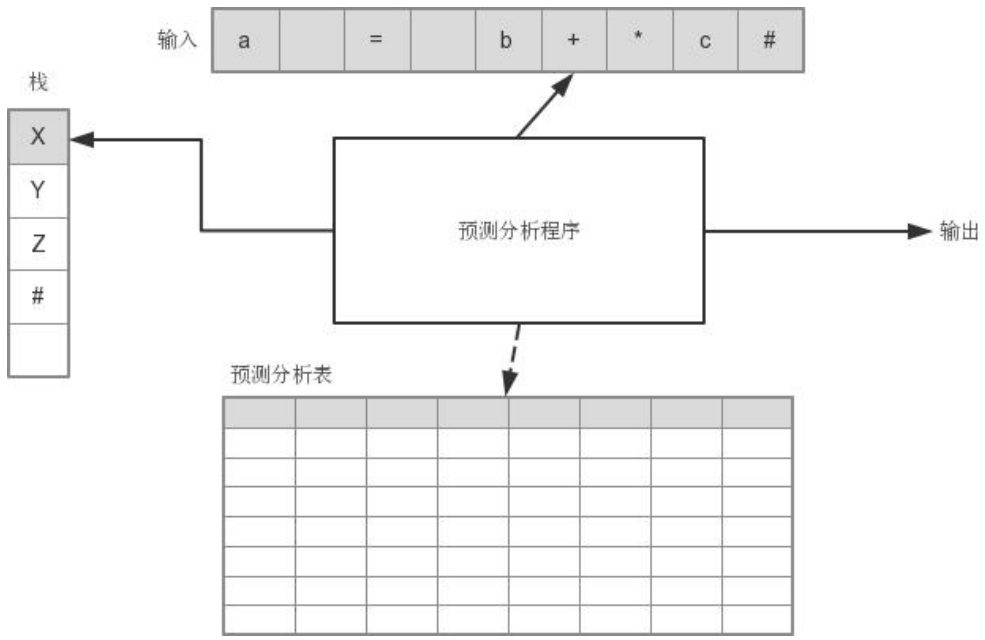
```
}

//其他非终结符同理
procedure
β1(){...}
procedure β2(){...}
procedure γ(){...}
```

这样就更容易看出我们为什么要对文法进行限制

预测分析程序的构造

这个方法就是严格按照LL文法的定义，从左到右扫描输入符号串，并从开始符号对文法进行自顶向下的最左分析。

预测分析程序结构



其中：

- **输入**就是符号串，我们使用一个游标对其进行扫描，游标的位置就是当前扫描到的符号
- **栈**是文法符号栈，分析开始的时候，栈中只有开始符号“S”和输入结束符号“#”
- **预测分析表**的行对应每一个非终结符（包括'#'），列对应每个终结符，表项的内容是产生式。预测分析表表达的意思是：一个非终结符出现在栈顶时，我们根据当前输入符号选择哪个产生式进行推导
- 分析的过程由栈顶符号和当前扫描到的符号决定，这两个位置对应的符号是分析程序的动态部分，我们用“(栈顶符号t,输入符号c)”的符号对表示当前文法分析程序的状态，并称为格局（configuration）。在 不同格局下，文法分析器将采取不同的动作：
  - 匹配：当$t \in V_T$时，我们将栈顶符号和当前输入符号进行匹配，具体操作和上面伪代码的函数一样 `match`
  - 替换：当$t \in V_N$时，我们使用(t,c)符号对查找预测分析表
    - 如果表内有产生式，就将栈顶符号出栈，并将查到的产生式的右部按倒序入栈（也就是入栈后，新栈顶是右部的第一个符号。从结果看，就是把栈顶的符号展开了）

- 如果表内无产生式，就报告语法错误
- 接受：当符号对呈现('#','#')的情况时，认为分析成功，分析程序退出

根据上面预测分析程序的结构和分析过程，我们可以看到除了预测分析表，程序的其他部分和文法一点关系也没有。换言之，定义一个新的文法，我们只要按照给定的文法构造出新的预测分析表，并替换掉分析器中原来的预测分析程序，我们就完成了一种新的语言的语法分析器，这种模块化的优势是递归向下分析程序无法比拟的。

预测分析表的构造

预测分析表的表项是当前格局（T,c）下（$T \in V_N$），应该用来继续推导非终结符的产生式，也就是说如果$c \in SELECT(T \to \beta)$时，我们就选择$T \to \beta$来继续我们的推导，体现在程序中就是用 β（指代一个符号串，而不是非终结符）替换掉栈顶的T

文法的等价变换

确定的自顶向下分析要求给定语言的文法必须是 LL(1)形式，然而，不一定每个语言都是LL(1)文法，对一个语言的非LL(1)文法是否能变换为等价的LL(1)形式以及如何变换是我们所感兴趣的。

提取左公因子

我们之前提到，存在左公因子的文法不是LL(1)文法，这样的问题比较容易修正

$$\begin{cases} A \to a\beta \\ A \to a\gamma \end{cases} \Rightarrow \begin{cases} A \to A' \\ A' \to \beta \\ A' \to \gamma \end{cases}$$

更一般地

$$A \to \alpha\beta_1|\alpha\beta_2|\dots|\alpha\beta_n \Rightarrow \begin{cases} A \to \alpha A' \\ A' \to \beta_1|\beta_2|\dots|\beta_n \end{cases}$$

注意：

- 如果$\beta_1, \beta_2, \dots, \beta_k$中也含有左公因子，可以重复上述步骤，直到没有左公因子。
- 一个文法提取了左公共因子后，只解决了相同左部产生式的右部FIRST集不相交问题。当改写后的文法不含有空产生式，且无左递归时，则改写后的文法是LL(1)文法。否则还需用LL(1)文法的判别方法进行判断才能确定是否为LL(1)文法。

消除左递归

更严格地来说，左递归的表示方法应该是 $A \Rightarrow^* A\beta$，也就是说左递归有两种形式

- 直接左递归：形如$A \to A\beta$
- 间接左递归：形如$A \to B\beta, B \to A\alpha$的多个个产生式

无论哪中情况，都会使分析程序陷入无限递归，进而不是LL(1)文法。

1. 把直接左递归改写为右递归

$$A \to A\beta \mid \gamma \Rightarrow \begin{cases} A \to \gamma A' \\ A' \to \beta A'|\epsilon \end{cases}$$

其中，$\epsilon \notin FIRST(\beta)$，γ不以A打头

一般地

$$A \to A\beta_1 | A\beta_2 | \ldots | A\beta_n | \gamma_1 | \gamma_2 | \ldots | \gamma_m$$

$$\Downarrow$$

$$\begin{cases} A \to \beta_1 A' | \beta_2 A' | \ldots | \beta_n A' \\ A' \to \gamma_1 A' | \gamma_2 A' | \ldots | \gamma_m A' | \epsilon \end{cases}$$

其中$\epsilon \notin FIRST(\beta_i)$，任何γ不以A打头

2. 消除间接左递归

   将间接左递归中所有产生式合并成为直接左递归，然后改写成右递归

   eg:

$$\begin{cases} A \to aB \\ A \to Bb \\ B \to Ac \\ B \to d \end{cases} \Rightarrow \begin{cases} A \to aB \\ A \to Acb \\ B \to d \end{cases}$$

   然后就可以直接处理$A \to Acb$了

3. 消除所有左递归的算法：

```
for i:=1 to n
do
    for j:=1 to i-1
    do
        若Aj的所有产生式为：
        Aj -> δ1| δ2 | … | δn
        替换形如Ai → Aj γ的产生式为：
        Ai -> δ1γ |δ2γ | … |δnγ
    end
    消除Ai中的一切直接左递归
end
```

这个算法的本质就是合并所有可能产生间接左递归的产生式，然后消除其中真的包含左递归的产生式

第五章 语法分析——自下而上分析

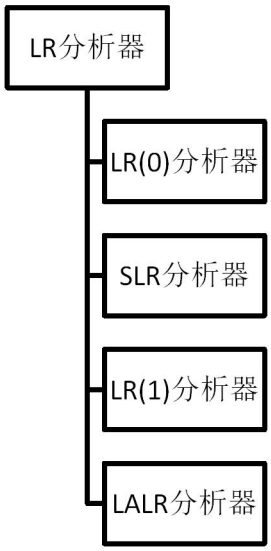
作者：袁郭苑

章节概述

这一章节中，我们讨论另一种语法分析的方法：自下而上分析。自下而上分析的核心是“归约”，其过程是不断将出现在符号栈栈顶的**产生式右边的的部分**在合理的情况下归约成为**产生式的左边项**，最终将输入的符号串归约到文法的开始符号。

在本章节，我们将一起学习LR分析法。根据遇到的不同情况，我们会学习构建LR(0)分析表、SLR分析表、LR(1)分析表和LALR分析表，运用分析表完成自下而上的语法分析过程。

章节框架



5.1 自下而上分析的一些基本知识

5.1.1 归约

我们所讨论的自下而上分析法是一种“移进-归约”法：用一个寄存符号的先进后出栈，把输入符号一个一个地移进到栈里，当栈顶形成一个产生式的某个候选式时，即把栈顶的这一部分替换成（规约为）该产生式的左部符号。

5.1.2分析表与状态

①分析表

LR分析器的核心部分是一张分析表（如图5.1所示）。它包括两部分内容，一是“动作”（ACTION）表，另一是“状态转换”（GOTO）表。

ACTION[s,a]规定了状态s面临输入符号a时应采取什么动作。

GOTO[s,X]规定了状态s面临文法符号X时下一状态是什么。

图5.1 LR分析表

②动作表的四种状态

状态	ACTION（动作）					GOTO（转换）		
	a_1	a_2	……	a_m	#	X_1	……	X_t
0	……					……		
1								
……								
n								

移进：把(s,a)的下一状态s’=GOTO[s,a]和输入符合a推进栈，下一输入符号变成现行输入符号。在分析表中用sk表示，k为代表s’的状态号。

归约：指用某一产生式 $A \to \beta$ 进行归约。在分析表中用rn表示，n为产生式 $A \to \beta$ 的标号。

接受：宣布分析成功，停止分析器的工作。在分析表中用acc表示。

报错：发现源程序含有错误，调用出错处理程序。在分析表中用空白表示。

③LR分析器的总控程序

总控程序本身的工作是非常简单的。它的任何一步只需按栈顶状态s和现行输入符号a执行分析表上规定的动作。不管什么分析表，总控程序都是一样地工作。

④LR文法

如何从文法构造LR分析表，是我们主要关心的问题。对于一个文法，如果能够构造一张分析表，使得它的每个入口均是唯一确定的，则我们将这个文法称为LR文法。并非所有上下文无关文法都是LR文法，但对多数程序语言来说，一般都可用LR文法描述。

⑤拓广文法

为了使“接受”状态易于识别，我们总把文法G进行拓广。假定文法G是一个以S为开始符号的法，我们构造一个G’，它包含了整个G，但它引进了一个不出现在G中的非终结符S’，并加进一个新产生式 $S' \to S$，而这个S’是G’的开始符号。那么我们称G’是G的拓广文法。这样便会有一个仅含项目 $S' \to S \bullet$ 的状态，这就是唯一的“接受”态。

5.2 *LR(0)、SLR、LR(1)、LALR分析表的构造

5.2.1LR(0)分析器

这里我们举例介绍LR(0)分析表的构建。

拓广文法5.1:

0:S’->E
1:E->aA
2:E->bB
3:A->cA
4:A->d
5:B->cB
6:B->d

LR(0)项目及规范族的构造涉及到两个函数：CLOSURE(I)与GO(I,X)。

假定I是文法G’的任一项目集，定义和构造I的闭包CLOSUER(I)的办法是:

①I的任何项目都属于CLOSUER(I)

②若 $A \to \alpha \bullet B\beta$ 属于CLOSURE(I)，那么，对任何关于B的产生式 $B \to \gamma$，项目 $B \to \bullet\gamma$ 也属于CLOSURE(I)

③重复执行上述两步骤直至CLOSURE(I)不再增大为止

函数GO是一个状态转换函数。GO(I,X)的第一变元I是一个项目集，第二变元X是一个文法符号。函数值GO(I,X)定义为:

GO(I,X)=CLOSURE(J)

其中

J={任何形如 $A \to \alpha X \bullet \beta$ 的项目 | $A \to \alpha \bullet X\beta$ 属于I}

构造文法5.1的LR(0)项目及规范族:

I0:
S’->.E
E->.aA
E->.bB

I1=GO(I0,E):
S’->E.

I2=GO(I0,a):
E->a.A
A->.cA
A->.d

I3=GO(I0,b):
E->b.B
B->.cB
B->.d

I4=GO(I2,A):
E->aA.

I5=GO(I2,c):
A->c.A
A->.cA
A->.d

I6=GO(I2,d):
A->d.

I7=GO(I3,B):
E->bB.

I8=GO(I3,c):
B->c.B
B->.cB
B->.d

I9=GO(I3,d):
B->d.

I10=GO(I5,A):
A->cA.

I11=GO(I8,B):
B->cB.

在上述构建过程中，“.”代表当前分析的位置。

接下来构造LR(0)分析表，I0~I11依次为状态1~11:

表5.1 语法5.1的LR(0)分析表

状态	ACTION					GOTO		
	a	b	c	d	#	E	A	B
0	s2	s3				1		
1					acc			
2			s5	s6			4	
3			s8	s9				7
4	r1	r1	r1	r1	r1			
5			s5	s6			10	
6	r4	r4	r4	r4	r4			
7	r2	r2	r2	r2	r2			
8			s8	s9				11
9	r6	r6	r6	r6	r6			
10	r3	r3	r3	r3	r3			
11	r5	r5	r5	r5	r5			

5.2.2 SLR分析器

但是很多时候，遇到的情况不像我们设想的一样美好，按上述方式得到的LR(0)分析表的入口对应的操作不是唯一的。比如下面这个例子：

拓广文法5.2：

0:S’->E
1:E->E+T
2:E->T
3:T->T*F
4:T->F
5:F->(E)
6:F->i

我们先分析一下文法5.2的LR(0)项目集规范族：

I0:
S’->.E
E->.E+T
E->.T
T->.T*F
T->.F
F->.(E)
F->.i

I1=GO(I0,E):
S’->E.
E->E.+T

I2=GO(I0,T):
E->T.
T->T.*F

I3=GO(I0,F):
T->F.

I4=GO(I0,():
F->(.E)
E->.E+T
E->.T
T->.T*F
T->.F
F->.(E)

```
F->.i

I5=GO(I0,i):
F->i.

I6=GO(I1,+):
E->E+.T
T->.T*F
T->.F
F->.(E)
F->.i

I7=GO(I2,*):
T->T*.F
F->.(E)
F->.i

I8=GO(I4,E):
F->(E.)
E->E.+T

I9=GO(I6,T):
E->E+T.
T->T.*F

I10=GO(I7,F):
T->T*F.

I11=GO(I8,)):
F->(E).
```

在这12个项目集中，I1、I2和I9都含有“移进-归约”冲突，这里我们用SLR(1)的方法来解决冲突：对于项目 $A \to \beta \bullet$，在分析表中，我们仅对A的FOLLOW集的符号填入rj，其中j为产生式 $A \to \beta$ 的编号。

接下来构造文法5.2的SLR(1)分析表，I0~I11依次为状态1~11：

表5.2 语法5.2的SLR(1)分析表

状态	ACTION						GOTO		
	i	+	*	(	)	#	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

具有SLR表的文法G称为一个SLR(1)文法。

5.2.3 LR(1)分析器

在实际处理过程中，我们会遇到一些SLR分析器无法胜任的文法，这时我们来一起看向带有**展望符**的LR(1)分析器。

拓广文法5.3:

0:S’->S

1:S->BB

2:B->aB

3:B->b

构造LR(1)项目集族的办法和构造LR(0)项目集规范族的办法是一样的，也要用到两个函数CLOSURE和GO。

假定I是一个项目集，他的闭包CLOSIRE(I)可按如下方式构造:

①I的任何项目都属于CLOSURE(I)

②若项目[$A \to \alpha \bullet B\beta, a$]属于CLOSURE(I)， $B \to \gamma$ 是一个产生式，那么，对于FIRST(βa)中的每个终结符b，如果[$B \to \bullet\gamma, b$]原来不在CLOSURE(I)中，则把它加进去

③重复执行步骤②，直至CLOSURE(I)不再增大为止

令I是一个项目集，X是一个文法符号，函数GO(I,X)定义为:

GO(I,X)=CLOSURE(J)

其中

J={任何形如[$A \to \alpha X \bullet \beta, a$]的项目 | [$A \to \alpha \bullet X\beta, a$] $\in$ I}

构造LR(1)项目集C:

I0:

S’->.S,#

S->.BB,#

B->.aB,a/b

B->.b,a/b

I1=GO(I0,S):

S’->S.#

I2=GO(I0,B):

S->B.B,#

B->.aB,#

B->.b,#

I3=GO(I0,a):

B->a.B,a/b

B->.aB,a/b

B->.b,a/b

I4=GO(I0,b):

B->b.,a/b

I5=GO(I2,B):

S->BB.,#

I6=GO(I2,a):

B->a.B,#

B->.aB,#

B->.b,#

I7=GO(I2,b):

B->b.,#

I8=GO(I3,B):

B->aB.,a/b

I9=GO(I6,B):

B->aB.,#

接下来构造文法5.3的LR(1)分析表，I0~I9依次为状态1~9：

表5.3 语法5.3的LR(1)分析表

状态	ACTION			GOTO	
	a	b	#	S	B
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

每个SLR(1)文法都是LR(1)文法。展望符仅对归约项目有意义。

5.2.4 LALR分析器

我们称两个LR(1)项目集具有相同的心，如果除去展望符之后，这两个集合是相同的。LALR分析器就是试图将所有同心的LR(1)项目集合并为一，如果合并后构造的分析表不存在冲突（只存在“归约-归约冲突”），则称其为文法的LALR分析表。

第六章 属性文法和语法制导翻译

作者：孙吉鹏

章节导读

如果问起来编译原理最迷的一章是什么，很多同学的回答将会是这一章，课本里讲了众多的不知道为什么会有的定义和莫名其妙的附加规则，但偏偏没有讲它们是为什么来的。你如果不信，想想这几个问题，为什么会有属性文法？有了那么多语法分析方法找出了语法错误对编译器难道不够么？它跟语法分析有什么联系呢?

首先我们要清楚编译器的工作并不是仅限于我们表面上在IDE里写程序时给我们提醒各种错误，其更重要的工作是将我们的源程序翻译成“目标代码”，翻译成机器能切切实实执行的代码，这是它最初最核心的功能。也就是说编译器首先要理解我们的源程序到底是想让机器干什么，这就是“语义”！你说，学到目前的章节为止，编译器有能力知道么?

没有哦，编译器现在通过词法，语法分析只知道了“结构”，举个例子，它现在知道 “if (a > b) { i ++}”是符合语法的，“if (a > b) { i + }”是不符合语法的因为构造不出语法分析树，但真的 a > b 了，计算机应该干什么呢？它现在还不知道该i++呢！只有我们学完了这两章，通过附加语义规则让计算机理解了if E {S}这个语法结构的语义是如果E是真的，就执行S的操作，我们才有可能让计算机理解并执行我们的源程序。

第六章主要站在**方法论**的角度讨论可以用什么样的方式来对不同的属性文法进行翻译，而第七章则举出了几个常用的例子（如何翻译布尔表达式的语义，如何翻译控制语句的语义）来**应用**第六章讨论的方法，明白所处的位置了吧，让我们开始!

6.1 属性文法

拿个实在的例子，对于台式计算器的语法制导定义

产 生 式	语 义 规 则
$L \rightarrow E$ n	*print* (*E.val*)
$E \rightarrow E_1 + T$	*E.val* := E_1 *.val* + *T.val*
$E \rightarrow T$	*E.val* := *T.val*
$T \rightarrow T_1 * F$	*T.val* := T_1*.val* * *F.val*
$T \rightarrow F$	*T.val* := *F.val*
$F \rightarrow (E)$	*F.val* := *E.val*
$F \rightarrow$ digit	*F.val* := digit.*lexval*

左边的一列我们前几章都熟悉，是产生式，右边的就是导读中解释的“语义规则”，即告诉计算机看到左边产生式该干什么，比如看到 $E \rightarrow E_1 + T$这样的语法结构，语义规则告诉计算机应当做加法把E和T

的值加起来$E.val := E_1.val + T.val$。

属性文法的定义由此而来，我们把左边的产生式和右边的语义规则的结合叫属性文法，对于每个文法符号（如E），它都有对应的属性（如.value）。

6.1.1 综合属性与继承属性

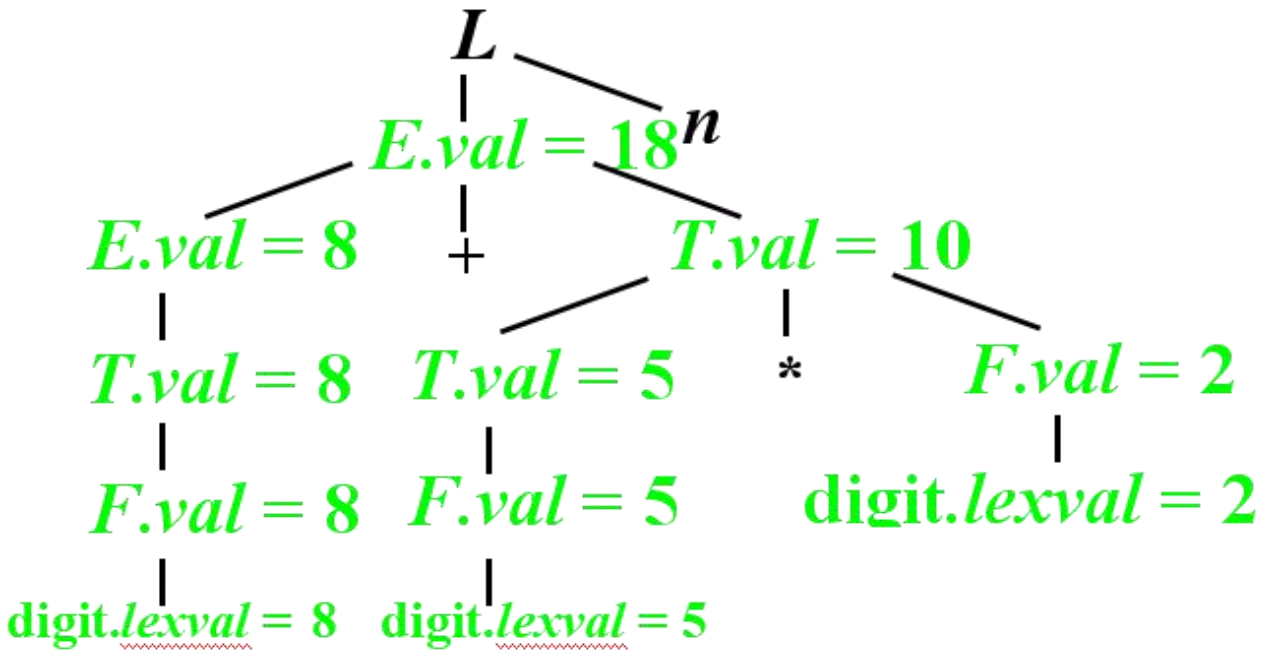
定义：

- **每个文法产生式$A \to \alpha$有一组形式为$b := f(c_1, c_2, \ldots, c_k)$的语义规则，其中$f$是函数，$b$和$c_1, c_2, \ldots, c_k$是该产生式文法符号的属性**
  - **综合属性（Synthesized Attribute）：如果b是A的属性，$c_1, c_2, \ldots, c_k$是产生式右部文法符号的属性或A的其它属性。**
  - **继承属性（Inherited Attribute）：如果b是产生式右部某个文法符号X的属性。**

综合属性是信息流流向产生式左部的属性，在语法树中体现为S属性节点被其子节点的属性值确立；继承属性是流向产生式右部的属性，在语法树中体现为被其父节点或兄弟节点的某些属性值确立的。

6.1.2 语法制导翻译方法

为了翻译解释语法结构的语义，我们自然想到的就是通过翻译语法树各节点的语义规则来解释，如解释8+5*2 n的语义（18 n呗，这个带注释的语法分析树要会画），我们可以先用上两章的方法得到的语法分析树，然后根据依赖的需要（比如算T时需要依赖T和F的值），遍历这棵树，碰见带+ ，带*的节点我们就将两个子树的节点的值算一算，最后不就得到最终的18n的语义了么。



这个计算流程就是

输入串→语法树→依赖图→语义规则计算次序

这种基于语法结构的处理方法就是语法制导翻译法。上述的方法我们注意到是先假设得到了语法分析树，之后再基于这个分析树做语义规则的计算，这样做有一个很致命的缺点就是分析现实世界编程语言中的庞大复杂的语法树实在是太费时间了，遍历一遍就够受的了，何况是遍历许多遍才能算出结果来。

能不能有这么一种方法，能压根不遍历这棵树，或者说就在构造这棵树的时候，就把这个语义分析的活儿干了呢?

还真有。

只要属性文法符合一定的规则特点，一遍扫描就可以实现，6.2—6.4我们将介绍几类可以一遍扫描的属性文法及其处理方法，就是在语法分析的同时计算属性值，而不是先构造语法树后进行计算。由于这种方法与语法分析器的动作紧密相连，它与输入串所采用的语法分析方法和属性的计算次序有关。

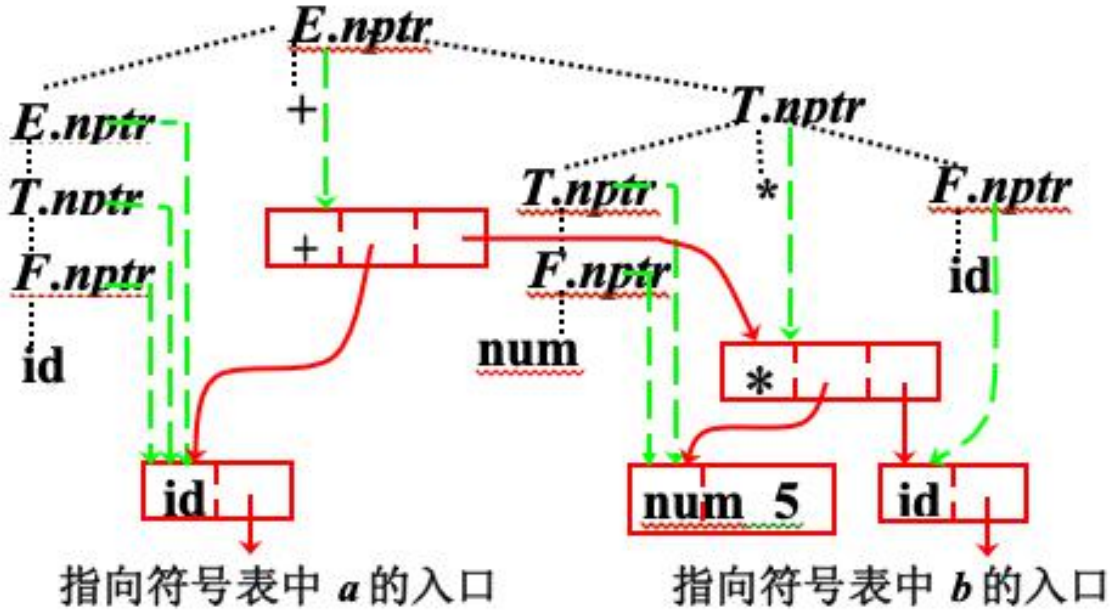
在自上而下语法分析中，若一个产生式匹配输入串成功，或者，在自下而上分析中，当一个产生式被规约时，此产生式相应的语义规则就被计算。由此可见，语法分析工作和语义规则的计算是穿插进行的。

6.1.3 抽象语法树

适用于翻译的语法结构不只是语法分析树，抽象语法树将操作符和关键字都不作为叶节点而是把它们当作内部节点，使用如下产生规则构造，可以看出它的核心就在于只为id，num，运算符创造节点（mkleaf（），mknode（）），T到E，F到T的规约，E到F的规约只是子树的转移。

产 生 式	语 义 规 则
$E \rightarrow E_1 + T$	$E.nptr := mknode(\text{'+'}, E_1.nptr, T.nptr)$
$E \rightarrow T$	$E.nptr := T.nptr$
$T \rightarrow T_1 * F$	$T.nptr := mknode(\text{'*'}, T_1.nptr, F.nptr)$
$T \rightarrow F$	$T.nptr := F.nptr$
$F \rightarrow (E)$	$F.nptr := E.nptr$
$F \rightarrow \mathbf{id}$	$F.nptr := mkleaf(\mathbf{id}, \mathbf{id}.entry)$
$F \rightarrow \mathbf{num}$	$F.nptr := mkleaf(\mathbf{num}, \mathbf{num}.val)$

最终对a+5*b构造抽象语法树的结果如下图所示，画出如何通过属性文法边分析语法结构（绿线）边构造抽象语法树（红线）也是一个很重要的考察点。

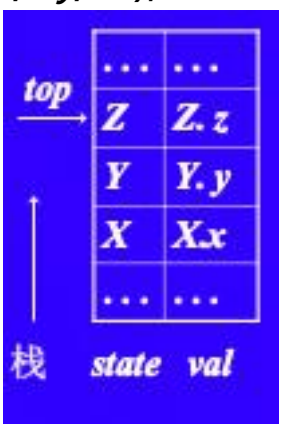


6.2 S属性文法的自下而上计算

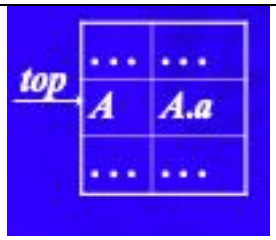
S属性文法是仅使用综合属性的语法制导的文法，如前面的台式机算器的例子，所有产生式左部符号的属性都来自于产生式右部的属性。由于综合属性的特点是产生式右边的属性传递到产生式左边符号的属性，这与规约从右向左的过程是一致的，所以S属性文法都可以采用自下而上的计算方法—LR分析器实现。

具体做法是将LR分析器的栈增加一个域来保存该符号的综合属性值（没有则不定义），当一个符号被规约出栈时，我们就可以拿到它所附加的综合属性值，从而进行运算操作，之后将规约后的符号压栈时，我们把计算得到的该符号的综合属性值也压入其中。

产生式$A \rightarrow XYZ$的语义规则是$A.a := f(X.x, Y.y, Z.z)$，



那么上图的情况归约后，Z,Y,X出栈，取得了综合属性值Z.z, Y.y, X.x，运用 $A.a := f(X.x, Y.y, Z.z)$，计算得出A.a的值，与A一起，压入栈顶，得到下图，从而在没有构造语法分析树的前提下完成了语义翻译的工作



给一个例题，将台式计算器的语法制导定义改成栈操作代码

产 生 式	语 义 规 则
$L \rightarrow E$ n	print (E.val)
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	E.val := T.val
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	T.val := F.val
$F \rightarrow (E)$	F.val := E.val
$F \rightarrow$ digit	F.val := digit.lexval

分析第一个产生式，打印E的值，由于移进时后移入的是n，所以现在栈顶元素是n，E是栈顶第二个元素，所以打印val[top-1]，分析第二个产生式，需要T出栈，E1出栈，算出E后压栈，所以val[top-2] = val[top-2]+val[top]。分析第三个产生式，出一个，没有任何计算后进一个，栈没有改变，所以可以不翻译。同理可得全部答案如下

产 生 式	代 码 段
$L \rightarrow E$ n	print (val [top−1])
$E \rightarrow E_1 + T$	val [top −2] := val [top −2]+val [top]
$E \rightarrow T$	
$T \rightarrow T_1 * F$	val [top −2] := val [top −2]×val [top]
$T \rightarrow F$	
$F \rightarrow (E)$	val [top −2] := val [top −1]
$F \rightarrow$ digit	

6.3 L属性文法的自上而下计算

6.3.1 L属性文法

定义

- **如果每个产生式$A \rightarrow X_1\ X_2 \ldots X_n$ 的每条语义规则计算的属性是A的综合属性；或者是X_j 的继承属性，$1 \le j \le n$，但它仅依赖：**
  - **该产生式中X_j左边符号$X_1, X_2, \ldots, X_{j-1}$的属性；**
  - **A的继承属性。**

通俗的讲，或者计算的是左边的综合属性，或者要计算的右部中的继承属性只依赖其左边符号的文法属

性，无论其在产生式右部还是左部。

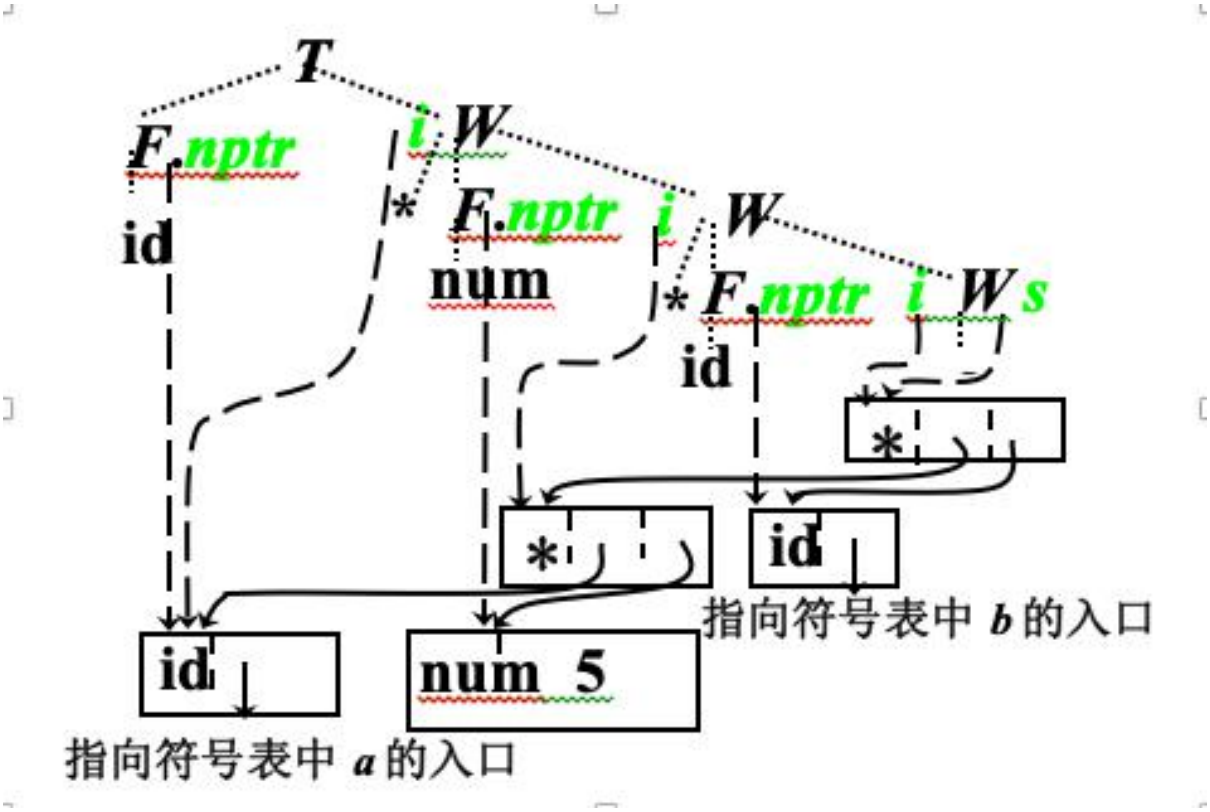
6.3.2 自顶向下翻译

回忆自顶向下的语法分析方式LL（1），分析器从开始符号根据读入的字符的select集选择产生式进行进行推导，读入字符的方式是从左向右；对于L属性文法，因为右边的符号属性依赖左边，所以必须要求左边的符号属性需要提前算出来，这跟自顶向下的分析方式的信息流动方向是吻合的。我们只需要将已推导出（位于待推导展开的符号左边）的符号属性的信息全部写给右边待推导符号的继承属性域里让它携带（同时埋下一个引子将父节点的综合属性域指向目前待推导的符号的综合属性域里，虽然为空），当右边的符号被推导展开时就可以从继承属性域里拿到其左边符号的信息，再进行运算，继续存到待展开的符号的继承属性域里并且埋下综合属性域的引子，直到右边的符号推导出ε，这时将其所有信息存到右边符号的综合属性域里，通过之前埋下的综合属性域的“引子”，我们可以从根节点顺藤摸瓜得到最终的结果。

依旧拿构造抽象语法树a*5*b举个例子

```
T → F          {W.i := F.nptr}
    W              {T.nptr := W.s}
W → *
      F              {W1.i := mknode ( '*', W.i, F.nptr)}
    W1         {W.s := W1.s}
W → ε          {W.s := W.i }
F → id         {F.nptr := mkleaf (id, id.entry)}
F → num        {F.nptr := mkleaf (num, num.val)}
```

读入a识别其为id，建立一个抽象树的叶子结点，使用$T \rightarrow F\ W$，将W左边F的树的信息存入W的继承属性域，T的综合属性域当作引子指向W的综合属性域，之后读入*，选择使用$W \rightarrow * FW1$ 推导，当前是F，读入5识别其为num，建立叶子结点，根据之前W和F的两个叶子结点的信息，生成a*5结点的子树，后存入W1的继承属性域，将W的综合属性域引子指向W1的综合属性域，进入W1，读入*，选择用$W \rightarrow * FW1$推导，当前为F，读入b识别为id，使用$F \rightarrow id$，建立b的叶子结点，根据当前W的继承属性域信息构造子节点a*5和b的节点，存入W1的继承属性域，同时W的综合属性域引子指向W1的综合属性域，当前为W，读入终结符选择$W \rightarrow \varepsilon$，这时将W的综合属性域指向W的继承属性域，综合属性域获得了最终结果的值。



6.4 自下而上地计算继承属性

我们看到这里会有疑惑，为什么继承属性这个产生式中从左向右流动，语法树上从上到下或者平级流动的属性，也能跟着从下而上的语法分析过程一遍翻译？之前我们看到的规约过程的一遍翻译可都是针对综合属性而言的。

让我们抛开课本深入地思考一下，本质上，是什么制约了继承属性的翻译？

是继承属性从左向右的信息流决定了它必须保证计算右边的属性值时，我左边的属性一定要已知了，否则没法计算。

那这么说如果我有种神奇的方法，它就是能知道左边符号的属性值是什么，或者从规约栈的角度看，左边的符号一定已经进栈了，它就是能知道栈里左边符号在栈里的位置，那么无论你到底是从上到下还是从下而上的计算过程，又有什么关系呢？

这一部分本质上，就是在用改写文法的方法，达到这个目的。

先看个简单的例子

翻译int p, q, r ，文法规则如下

D → ***T*** {***L.in*** := ***T.type***}

L

T → int {***T. type*** := ***integer***}

T → real {***T. type*** := ***real***}

L → {L_1.***in*** := ***L.in*** }

L_1, id {***addtype*** (id.***entry***, ***L.in***)}

L → id {***addtype*** (id.***entry***, ***L.in***)}

输入串	栈的状态	所用产生式
Int p,q,r		
P,q,r	int	
P,q,r	T	***T*→ int**
,q,r	Tp	
,q,r	TL	***L*→ id**
Q,r	TL,	
,r	TL,q	
,r	TL	***L*→L1, id**
r	TL,	
	TL,r	
	TL	***L*→L1, id**
	D	***D*→TL**

从属性文法规则中我们看到，***L*→L1, id** 规约时，需要执行{***addtype* (id.*entry*, *L.in*)}，但L.in这个属性的值我们实际上需要从***D*→TL 的{*L.in* := *T.type*}**这个继承属性里得到，也就是我们每次规约***L*→L1, id都**需要知道*T.type*这个值，想取得这个值，我们只有知道T在栈内的位置才可以，那我们观察这个栈，是不是很巧，每次***L*→L1, id 或者*L*→ id** 规约时T恰恰都是在L的下面，要取得这个值，我们只需要盲目取L下面的一个元素即可，因为我们知道那个地方一定是T，因此我们可以得到以下的翻译：

产 生 式	**代 码 段**
D* → *TL	
***T*→ int**	***val*[*top*] := *integer***
***T*→ real**	***val*[*top*] := *real***
***L*→ L_1, id**	***addtype*(*val*[*top*], *val*[*top*–3])**
***L*→ id**	***addtype*(*val*[*top*], *val*[*top*–1])**

你看，表面上

***L*→ L_1, id**	***addtype*(*val*[*top*], *val*[*top*–3])**

这个规则因为没有出现T所以看不出T的位置来，但我们就是因为有了那个T下面就是L的保证，所以让我们就可以无忧无虑地找到T的位置，从而完成属性文法的计算。

但如果像下面这种情况，

S* → *aAC	***C.i* := *A.s***
S* → *bABC	***C.i* := *A.s***
C* → *c	***C.s* := *g*(*C.i*)**

我们在规约c时需要用到C.i的值，C.i的值却依赖A的位置，可此时A既有可能出现在C的下面一个符号（***S* → *aAC***），也可能出现在C下两个符号的位置（***S* → *aAC***），这种情况下，我们该怎么拿到A.s的值呢?

我们如果能永远保证A.s的值在C下面一个的位置就好了，三十六计里有李代桃僵，偷梁换柱的思想，我们不妨加一个稻草人***M*** → ε跟在C的前面充数，让M一个属性的值等于A.s不就好了！那如何让M的值等于A.s呢？我们不妨在带M的表达式规约时加一条规则，把A.s的值赋给M，因为M在A的右边，所以从左到右传值这是M的一个继承属性，就记为M.i 把！那如何将M.i的值留下来呢，因为栈里的综合属性域只能有M.s的值，M.i是继承属性，那就在***M*** → ε规约时让它执行***M.s*** := ***M.i***的语义动作就好啦！不就把继承属性存到综合属性里了么！所以改写后的文法，应该是这样的：

S → ***aAC*** ***C.i*** := ***A.s***

S → ***bABMC*** ***M.i*** := ***A.s***; ***C.i*** := ***M.s***

C → ***c*** ***C.s*** := ***g***(***C.i***)

M → ε ***M.s*** := ***M.i***

总结一下，我们处理此类问题的方法就三步

1. 观察属性文法左部依赖哪个右部符号，是不是相对位置总相等
2. 不相等就李代桃僵，偷梁换柱
3. 运用复写规则存起来那个继承属性的值

当我们遇见如下的文法，怎么办？

Pascal的声明，如m, n : integer

D → ***L*** : ***T***

T → **integer | char**

L → ***L***, **id | id**

此时L的属性依赖右边T的值！关键T因为在L右边，不断算L时T还算不出来，即使通过继承属性第一个L从右边的T那里拿到T.type,因为从右向左，这也不是L属性文法了。

这真的无能为力了，只能像当年消除左递归一样，改文法了！

改成从右向左归约

D → **id** ***L***

L → **,** **id** ***L*** **| :** ***T***

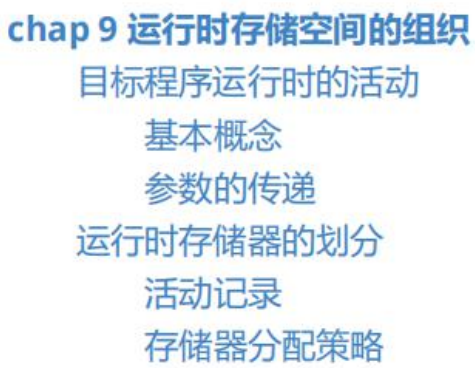
T → **integer | char**

这样T规约时，一定所有的id已经规约成L了，T就可以直接传给L了。

第九章 运行时存储空间的组织

作者：李鸣一

章节结构



章节背景：

根据我们以往的经验，程序运行时的存储空间管理是操作系统的工作；而编译器是负责将静态的源程序翻译成静态的可执行程序，按理说和程序运行时应该一点关系都没有。

但是实际上，程序各个段的划分，函数调用的传参和返回，局部变量的空间分配这些仅用一条指令根本实现不了的行为，都是由编译器生成指令实现的。

操作系统进行的运行时内存管理，其实仅仅是以进程为对象的存储空间分配与回收，加上虚拟内存，对整个用户空间进行管理。而编译器负责的是进程自身的存储空间管理行为，如栈空间的申请与释放等。

目标程序运行时的活动

基本概念

- 活动：过程的一次执行称为过程的一次活动
- 活动记录：过程的活动需要的可执行代码，和存放信息所需要的存储空间（活动记录）
- 活动的生存期：过程P一次活动的生存期，指的是从执行该过程体第一步操作到最后一步操作之间的操作序列，包括执行P时调用的其他过程所花费的时间

> *注意：*
>
> *活动描述的对象是过程，不一定是进程，过程是一段相对独立的代码的集合，可以理解为我们熟悉的函数、方法、子程序等，过程的执行，实际上就是过程被调用。*

参数的传递

按照传递内容，可以分为3种传递方式：

- 传值：

  把参数的值计算出来，放到一个被调用过程可以拿得到的地方。被调用过程开始工作时，首先把这些值抄到自己的活动记录中，然后像使用局部变量一样使用它们。

- 传地址：

  把参数当前所存储的地址传给被调用过程，被调用过程像对待传值参数一样把这些地址放到自己的活动记录中，但是访问参数时按照地址使用。

- 传名：

  把被调用过程扣掉参数，整个替换掉调用它的语句，然后用实际参数填补空缺。行为和C/C++的含参宏一样，原则上讲并不是编译器的行为。

另外，参数传递的途径也有多种：

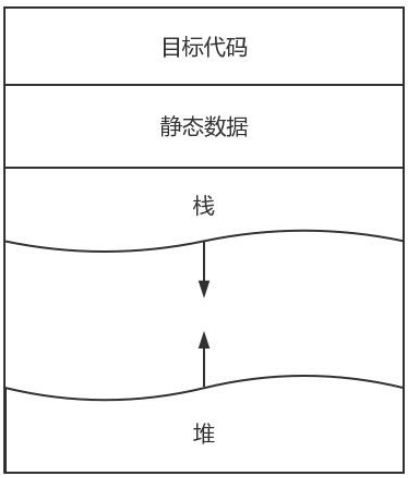
- 利用专用寄存器存放参数
- 利用通用寄存器存放参数
- 将参数直接放入子过程的地址空间

这个不重要，了解即可

运行时存储器的划分

这里的划分不是指操作系统的存储器管理，而是指一个进程的地址空间内，空间如何划分给各个段。空间中应该包括生成的目标代码、数据对象和跟踪过程活动的控制栈。

- 目标代码的大小在编译时可以确定，所以编译程序可以把它放在一个静态确定的区域。
- 堆栈空间的实际用量很难在运行时得出，这里编译器就需要一套完备的动态存储管理方式



活动记录

为了管理过程在一次执行中所需要的信息，使用一个连续的存储块，这样的一个连续存储块称为**活动记录**，一般包括：

- 局部变量：源代码中在过程体中声明的变量
- 临时单元：为了满足表达式计算要求而不得不预留的临时变量
- 内情向量：内情向量是静态数组的特征信息，用来描述数组属性信息的一些常量，包括数组类型、维数、各维的上下界及数组首地址
- 动态链&返回地址：
- 都是和过程返回有关的，返回地址存储调用指令的下一条指令的地址；而动态链存储"父过程"的栈空间基址，用于支持基址寻址的平台上基址的恢复。返回地址我们必须存储，而动态链根据目标平台而定。
- 静态链：对于某些老式的高级程序语言，它们支持过程嵌套，这对于编译器来说是非常麻烦的事情，因为内部过程必须要具备访问外部过程变量的能力，而外部过程的变量也是动态变量。

同样，动态链也是在基址寻址的背景下发展出来的技术。在过程调用的时候，我们将"外层过程"在栈中首个活动记录的栈基址存储为被调用"内层过程"的静态链，这样，"内层过程"想要访问"外层过程"的的变量时可以直接通过"静态链+偏移量"实现

> 注：
>
> 这个部分是过时的技术，作为编译性语言，C/C++经久不衰的原因正是因为它们不支持过程嵌套。因为过程嵌套会延伸出许多语法细节层面的问题（例如，内层过程能不能调用外层过程，兄弟过程、表兄弟过程之间是否能够相互调用），而编译器设计者不得不明确这些问题—或是实现相关文法特性，或是在语言手册中指出这种操作是非法的，无论是哪种情况，都会使文法变得十分复杂并难以掌握。
>
> PS：
>
> 并不是说过程嵌套是落后的，javascript过程嵌套的便捷是受业界公认的，但是其语法的复杂性可见一斑，如此复杂的语法的实现突出了解释性语言的优势
>
> Java和C++支持的lambda和匿名内部类可能会和过程嵌套产生混淆，首先，前者是建立在面向对象的基础上的，对象的生命周期是独立于过程的，另一方面，匿名内部类也不能随意访问局部变量，如Java中想要被匿名类访问的变量必须以final声明。

请注意，活动记录并不是活动唯一可以访问的数据。全局变量，堆，对象的成员也可以被过程访问

存储器分配策略

1.静态分配策略：在编译时对所有数据对象分配固定的存储单元，且在运行时始终保持不变。

- 面向全局变量，静态变量，常量等

2.动态分配策略

- 栈式动态分配策略：在运行时把存储器作为一个栈进行管理，运行时，每当调用一个过程，它所需要的存储空间就动态地分配于栈顶，一旦退出，它所占空间就予以释放

○堆式动态分配策略：在运行时把存储器组织成堆结构，以便用户关于存储空间的申请与归还（回收），凡申请者从堆中分给一块，凡释放者退回给堆。

两种策略不互斥，各有各的特点和必要性

栈式策略是面向过程的程序设计的基础，没有栈式策略，过程调用就会像进程间通信一样复杂

堆式策略是面向对象的基础，作为对面向过程的程序设计的一种补充，堆式策略使得堆中的数据从过程的活动中独立出来，具备了独立的，可控的生命周期。进而才能发展出以对象交互为核心的程序设计思想。

3.影响存储分配策略的语言特征

（仅作了解）

- 过程能否递归
- 当控制从过程的活动返回时，局部变量的值是否要保留
- 过程能否访问非局部变量
- 过程调用的参数传递方式
- 过程能否作为参数被传递
- 过程能否作为结果值传递
- 存储块能否在程序控制下动态地分配
- 存储块是否必须显式地释放

==========

感谢对智库的支持，如发现本知识见解有错误或对结构内容有你的观点，欢迎扫码给我们留言。