编译原理与技术课程笔记

Course Notes on Compiling Principles and Techniques

编者

山东大学 计算机科学与技术学院
2018级同学

SDUCS 2021

目录

指导老师：王丽荣

编者（排名不分先后）：

陈晓曦，米有麦彦，徐宏涛，朱可欣，薛宇涵，尹永琪，杜雅莉，徐容，孙书镇，刘千一，黑乃磊，李博远，张倩，来苑，牛庆莹，赵子涵，王新宇，尹浩飞，张雨，王晨旭，赵雨晗，王志睿，施博凡，张芮睿，张哲，亓佳宁，陈路

正则表达式转NFA转DFA及最小化

为正则式构造NFA：

定理：对任何FA都存在一个正规式r，使L(r)=L(M).

对任何正规式r，都存在一个FA M，使L(M)=L(r).

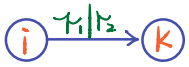
⇒ 构造Σ上的NFA M'使对给定正规式r有：L(r)=L(M)

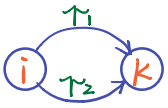
步骤：①首先，将r表示为 

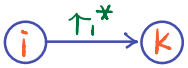
②按如下3条规则对r进行分裂.

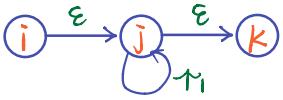
a) 

代之为 

b) 

代之为 

a) 

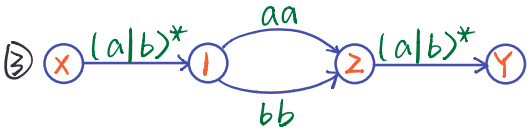
代之为 

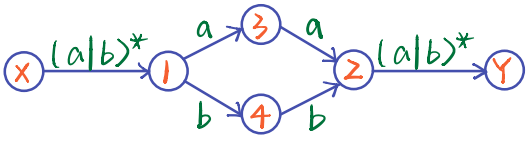
逐步把这个图转变为每条弧只标记Σ上的一个字符或ε，得一个NFA M'，显然，L(M')=L(r).

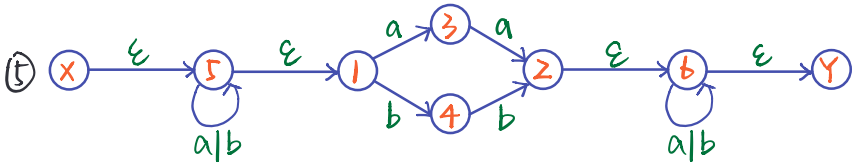
下例：(a|b)*(aa|bb)(a|b)*

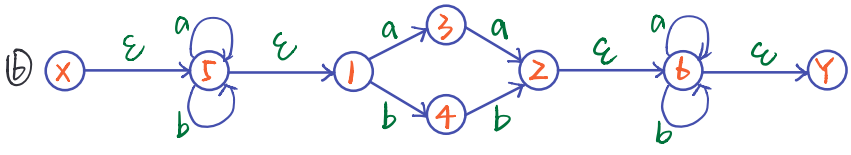
① 

② 

③ 

④ 

⑤ 

⑥ 

——BY 米有麦彦

NFA变DFA：确定化

① 假定I是状态集的子集，定义I的ε闭包 ε_CLOSURE(I)为：

i) 若 $S \in I$，则 $S \in \varepsilon\_CLOSURE(I)$；

ii) 若 $S \in I$，则从S出发经过任意条ε弧而能到达的任何状态S'都属于ε_CLOSURE(I)

即（收敛）：

$\varepsilon\_CLOSURE(I) = I \cup \{S'|$ 从某个 $S \in I$ 出发经过任意条ε弧所能到达 $S'\}$

② 假定I_a是状态集的子集，$a \in \Sigma$，定义：

$$I_a = \varepsilon\_CLOSURE(J)$$

其中J是那些可以从I的某一状态结点出发，经过一条弧a而到达的所有状态结点

③ 假定 $\Sigma = \{a_1, \cdots a_k\}$，构造一张表，每一行含k+1列

i) 置该表的首行首列为 ε_CLOSURE(X)（初态）

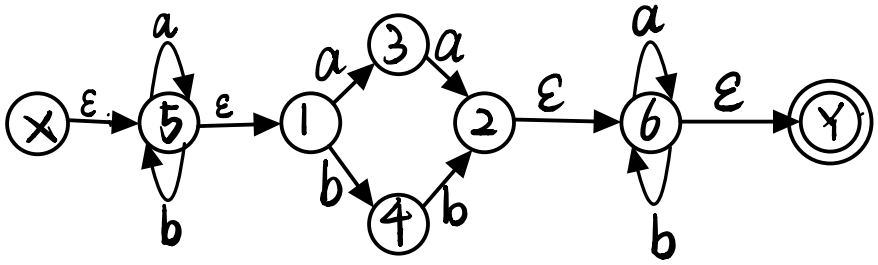
置该行的j+1列为 I_{a_j} $(j=1 \cdots k)$

ii) 然后，检查该行上的所有状态子集，若出现新状态，则增加至下一行，直至没有新状态集产生为止

例：

正规式 (a|b)*(aa|bb)(a|b)*

NFA：



I	I_a	I_b
{X, 5, 1}	{5, 3, 1}	{5, 4, 1}
{5, 3, 1}	{5, 3, 1, 2, 6, Y}	{5, 4, 1}
{5, 4, 1}	{5, 3, 1}	{5, 4, 1, 2, 6, Y}
{5, 3, 1, 2, 6, Y}	{5, 3, 1, 2, 6, Y}	{5, 4, 1, 6, Y}
{5, 4, 1, 6, Y}	{5, 3, 1, 6, Y}	{5, 4, 1, 2, 6, Y}
{5, 4, 1, 2, 6, Y}	{5, 3, 1, 6, Y}	{5, 4, 1, 2, 6, Y}
{5, 3, 1, 6, Y}	{5, 3, 1, 2, 6, Y}	{5, 4, 1, 6, Y}

可以经过a弧　可以经过b弧

初态为ε-闭包 新初态

出现新状态集

重复添加操作

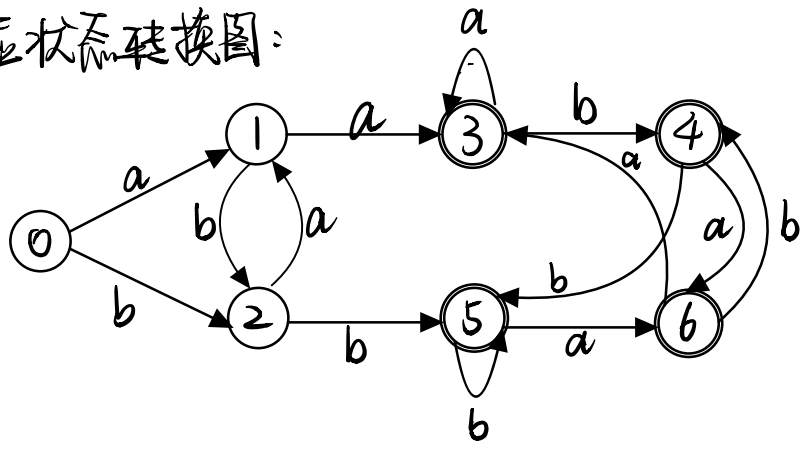
含有终态的状态集为新终态

无新状态集产生，ending~

对状态进行重命名：

S	a	b
0	1	2
1	3	2
2	1	5
3	3	4
4	6	5
5	6	5
6	3	4

画出相应状态转换图：



得到未化简的DFA

——by 陈晓曦

DFA的化简

2021年6月11日 21:26

1. 为什么进行DFA的化简
   a. 在DFA中，有些状态在识别字的时候发挥的作用是一样的。
   b. 将这些作用一样的状态合并，能够减小DFA，编写程序的时候能够节约内存，提升效率，减小编码量。
2. DFA化简的任务

   对于给定的DFA M，寻找一个状态数比M少的DFA M'，使得L(M)=L(M')
3. 状态的等价性和可区分性
   a. （状态等价性定义）假设s和t为M的两个状态，称s和t等价：如果从状态s出发能读出某个字α而停止于终态，那么同样，从t出发也能读出α而停止于终态；反之亦然。当两个状态不等价，则称他们是可区别。
   b. （可区分性定义）存在一个字α，要么s读出α停止于终态而t读出α停止于非终态，要么t读出α停止于终态而s读出α停止于非终态。
   c. 直观理解：如果一个状态到终态所有可以读出来的字都与另一个状态相同，则他们的处理能力是一样的，这两个状态就是等价的，保留一个即可。当存在一个字α，可以将两个状态的读取能力区分开来，则说明这两个状态不能完全等价，不能化简。
4. DFA化简的基本思想
   a. 把M的状态集划分为一些不相交的子集，使得任何两个不同子集的状态是可区别的，而同一子集的任何两个状态是等价的。
   b. 最后，让每个子集选出一个代表，同时消去其他状态。结果是DFA的状态数目减少了，实现了化简。

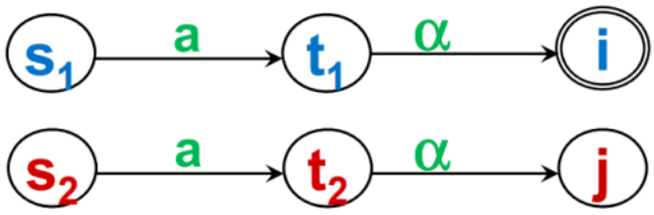
5. DFA的初始划分
   a. 按照上述原则，我们可以对DFA进行第一次划分。我们注意观察，我们可以先把DFA划分成终态和非终态。
   b. 因为空字可以将终态和非终态区别开来，所以我们可以进行这样一次的简单划分。
   c. 注意！并不是说终态都是等价的，上条只是强调了其与其他状态是可区分的。
   d. 思考！为什么初态和非初态不能简单划分。
6. DFA的化简算法
   a. 首先，把S划分为终态和非终态两个子集，形成基本划分Π。 假定到某个时候，Π已含m个子集，记为Π={I(1)， I(2)，…，I(m)}，检查Π中的每个子集看是否能进一步划分： 对某个I(i)，令I(i)={s1,s2, …,sk}，若存在一个输入字符a使得$I_a(i)$不会包含在现行Π的某个子集I(j)中，则至少应把I(i)分为两个部分。
   b. 假定状态s1和s2是I(i)={s1,s2, …,sk}中的两个状态，它们经a弧分别到达t1和t2，而t1和t2属于现行Π中的两个不同子集

i. 说明有一个字α， t1读出α后到达终态，而t2读出α后不能到达终态，或者反之

ii. 那么对于字aα， s1读出aα后到达终态，而s2读出aα不能到达终态，或者反之

iii. 所以s1和s2不等价

iv.

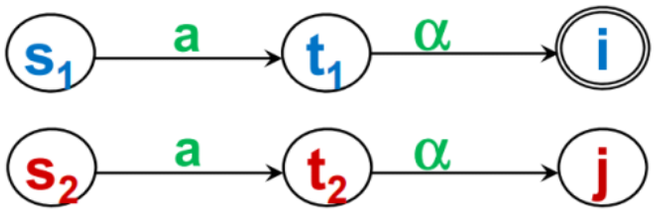


c. 将I(i)分成两半，一半含有s1，一半含有s2

i. I(i1)含有s1: I(i1) ={s|s∈I(i)且s经a弧到达t, 且t与t1属于现行Π中的同一子集}

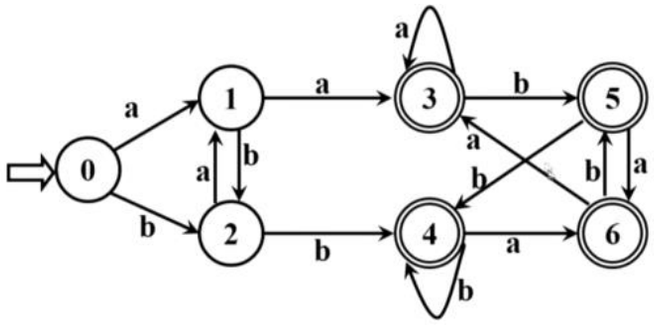
ii. I(i2)含有s2 : I(i2)=I(i)-I(i1)

iii.



d. 一般地，对某个a和I(i)，若$I_a(i)$ 落入现行Π中 N 个不同子集，则应把I(i)划分成N个不相交的组，使得每个组J的Ja都落入的Π同一子集。 重复上述过程，直到Π所含子集数不再增长。视频区域对于上述最后划分Π中的每个子集，我们选取每个子集I中的一个状态代表其他状态，则可得到化简后的DFA M’ 。若I含有原来的初态，则其代表为新的初态，若I含有原来的终态，则其代表为新的终态。

7. 例题



初步划分，将状态分成终态和非终态

$$I^{(1)}=\{0, 1, 2\} \quad I^{(2)}=\{3, 4, 5, 6\}$$

检查第一个集合，读入a时，集合I(1)进入状态1、3。而状态1、3在目前的状态划分中处于不同的集合，所以对于集合I(1)要进行划分。

$$I_a^{(1)}=\{1, 3\}$$

$$I^{(11)}=\{0, 2\} \quad I^{(12)}=\{1\} \qquad I^{(2)}=\{3, 4, 5, 6\}$$

对集合I(11)继续检查，读入a时，状态0、2都进入状态1，但读入b时，转移到状态2、4。而状

态2、4目前处于不同的划分中，所以将0、2进一步划分。

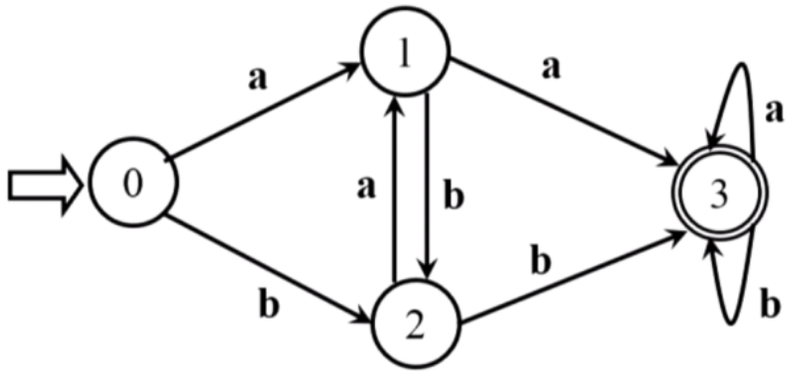
$$I_a^{(11)}=\{1\}\ I_b^{(11)}=\{2,4\}$$

$$I^{(111)}=\{0\}\quad I^{(112)}=\{2\}\qquad I^{(12)}=\{1\}\quad I^{(2)}=\{3,4,5,6\}$$

对于单个状态的集合就不用检查了，因为没有办法再划分了。

接下来检查I(2)。读入a时，进入状态3、6.目前3、6是处于同一个集合中。读入b，进入状态4、5，目前4、5处于同意个集合。

我们重新画出化简后的DFA，如下图。

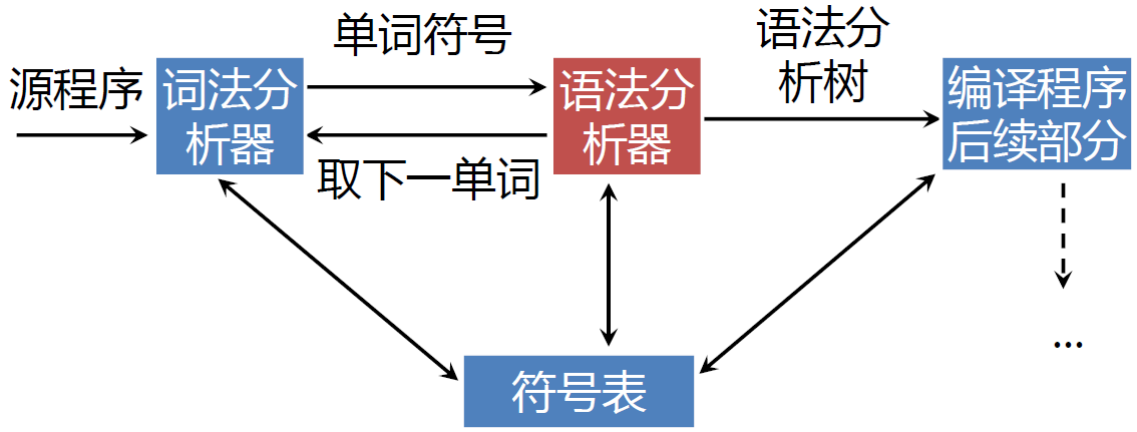


(——by 徐宏涛)

> 笔记作者 :
>
> 尹永琪与朱可欣共同完成 : 4.1 语法分析概述 4.2 自上而下语法分析 4.3 LL(1)文法
>
> 薛宇涵完成 ：4.4 递归下降分析程序构造 4.5 预测分析程序

4. 语法分析

4. 1 语法分析简介



4.1.1 概述

4.1.1.1 位置

处于编译过程的第二阶段

4.1.1.2 任务

在词法分析的基础上，根据语法规则，把单词符号串分解成各类 **语法单位(语法范畴)**，如“短语”、”子句“、”句子“（”语句“）、”程序段“和”程序“等。

tips : 词法分析是一种线性分析，而语法分析是一种层次结构分析

4.1.1.3 依循的原则

语法规则

4.1.1.4 描述工具

上下文无关文法

4.1.1.5 结果

构建一颗语法分析树

4.1.2 语法分析的方法

按照语法分析树的构建方法进行分类

4.1.2.1 自下而上

- 从输入串开始，逐步进行归约，直到文法的开始符号
- **归约**：根据文法的产生式规则，把串中出现的产生式的右部替换成左部符号 (top down parsing)
- 从树叶节点开始，构造语法树
- 常用方法：算符优先分析法、LR 分析法

4.1.2.2 自上而下

- 从文法的开始符号出发，反复使用各种产生式，寻找"匹配"的推导
- **推导**： 根据文法的产生式规则，把串中出现的产生式的左部符号替换成右部 (bottom up parsing)
- 从树的根开始，构造语法树
- 常用方法：递归下降分析法、预测分析程序

4.2 自上而下语法分析

4.2.1 概述

4.2.1.1 基本思想

从文法的开始符号(根节点)出发 , 向下推导 , 推出句子.

4.2.1.2 主旨

自上而下地建立一颗语法树.(寻找一个最左推导)

本质上是一种尽可能的试探法

4.2.1.3 实现的简单性

- 每个非终结符对应一个递归子程序 : 一个布尔过程
- 当它的某个候选与输入串相匹配 : 用它拓展语法树 , 返回1 . 否则 , 保持原有语法树和IP值不变 返回0

4.2.1.4 主要问题

1. 文法的左递归性问题 —— 无限循环

   含有左递归的文法将使得自上而下的分析过程陷入无限循环 . 因此 , 使用自上而下分析法须消除文法的左递归性.

   因此解决方法可以为： 左递归 ->右递归

2. 回溯

   如果对同一个非终结符号，存在若干个候选，如 $A \rightarrow \alpha_1|\alpha_2|\ldots|\alpha_n$，在推导过程中，如果候选选择错误，将导致回溯 . 因此，候选式的头符号不可以相交(这样就只有一个选择了)

3. 虚假匹配

   当一个非终结符用某一候选匹配成功时 , 这种成功可能仅是暂时的 .由于这种虚假现象 , 我们需要更复杂的回溯技术 .

4. 难以定位出错位置

5. 效率低下

   带回溯的自上而下分析实际上采用了一种穷尽一切可能的试探法 , 因此效率很低 , 代价极高 .

4.2.2 消除左递归

4.2.2.1 左递归文法定义

如果存在非终结符P , $P \overset{+}{\Rightarrow} P\alpha$

4.2.2.2 直接左递归的消除方法

假设关于非终结符P的规则为

$$P \to P\alpha|\beta \tag{1}$$

其中，β不以P开头，那么我们可以把P的规则改写为如下形式的非直接左递归形式

$$\begin{aligned} &P \to \beta P' \\ &P' \to \alpha P|\epsilon \end{aligned} \tag{2}$$

这种形式和原来的形式是等价的，也就是说从P推出的符号串是相同的

> 例题 4.2 文法
>
> $$\begin{aligned} &\mathrm{E} \to \mathrm{E}+\mathrm{T} \mid \mathrm{T} \\ &\mathrm{T} \to \mathrm{T} * \mathrm{F} \mid \mathrm{F} \\ &\mathrm{F} \to (\mathrm{E}) \mid \mathrm{i} \end{aligned} \tag{3}$$
>
> 经消去直接左递归后变成
>
> $$\begin{aligned} &\mathrm{E} \to \mathrm{TE}' \\ &\mathrm{E}' \to +\mathrm{TE}' \mid \varepsilon \\ &\mathrm{T} \to \mathrm{FT}' \\ &\mathrm{T}' \to *\mathrm{FT}' \mid \varepsilon \\ &\mathrm{F} \to (\mathrm{E}) \mid \mathrm{i} \end{aligned} \tag{4}$$

一般而言，假定P关于的全部产生式是

$$\mathrm{P} \to \mathrm{P}\alpha_1 \mid \mathrm{P}\alpha_2 \mid \cdots \mid \mathrm{P}\alpha_\mathrm{m} \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_\mathrm{n} \tag{5}$$

其中,每个α都不等于ϵ,而每个β都不以P开头,那么,消除P的直接左递归性就是把这些规则改写成:

$$\begin{aligned} &\mathrm{P} \to \beta_1 \mathrm{P}' \mid \beta_2 \mathrm{P}' \mid \cdots \mid \beta_\mathrm{n} \mathrm{P}' \\ &\mathrm{P}' \to \alpha_1 \mathrm{P}' \mid \alpha_2 \mathrm{P}' \mid \cdots \mid \alpha_\mathrm{m} \mathrm{P}' \mid \varepsilon \end{aligned} \tag{6}$$

使用这个办法,我们容易把见诸于表面上的所有直接左递归都消除掉,也就是说,把直接左递归都改成直接右递归。

4.2.2.3 间接左递归

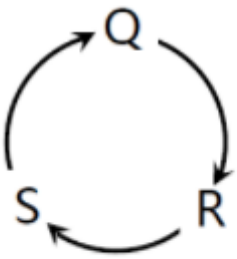
体现：产生式可以生成一个闭环。

例如文法

$$\begin{aligned} &\mathrm{S} \to \mathrm{Qc} \mid \mathrm{c} \\ &\mathrm{Q} \to \mathrm{Rb} \mid \mathrm{b} \\ &\mathrm{R} \to \mathrm{Sa} \mid \mathrm{a} \end{aligned} \tag{7}$$

虽不具有直接左递归，但S，Q，R都是左递归的，例如有

$$\mathrm{S} \Rightarrow \mathrm{Qc} \Rightarrow \mathrm{Rbc} \Rightarrow \mathrm{Sabc} \tag{8}$$



$$
\begin{aligned}
&S \rightarrow Qc \mid c \\
&Q \rightarrow Rb \mid b \\
&R \rightarrow Sa \mid a
\end{aligned}
$$

如何消除一个文法的一切左递归呢？虽然困难不少,但仍有可能。如果一个文法不含回路 (形如$P \overset{+}{\Rightarrow} P$的推导) , 也不含以$\epsilon$为右部的产生式 , 那么 , 执行下述算法将保证消除左递归 (但改写后的文法可能会含有以ϵ为右部的产生式)

消除间接左递归的基本思路：把候选中开头的非终结符替换成它的产生式，逐步减少圈中的节点，最后变成一个子圈（即直接左递归）

4.2.2.4 消除左递归的算法 :

把文法G的所有非终结符按任一种顺序排列成$P_1, P_2, \cdots, P_n$,按此顺序执行

- for i := **1** to **n** do :
  - for j := **1** to **i-1** do :
    - 把形如$P_i \rightarrow P_j\gamma$的规则改写成$\mathrm{P_i} \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_\mathrm{k}\gamma_\circ$
    - 其中 $\mathrm{P_j} \rightarrow \delta_1 \mid \delta_2 \mid \cdots \mid \delta_\mathrm{k}$ 是对于 $\mathrm{P_j}$ 的所有规则
  - 消除关于P规则的直接左递归性

化简由 2 所得的文法。即去除那些从开始符号出发永远无法到达的非终结符的产生规则。

> 例 4.3 考虑以下文法
>
> $$
> \begin{aligned}
> &\mathrm{S} \rightarrow \mathrm{Qc} \mid \mathrm{c} \\
> &\mathrm{Q} \rightarrow \mathrm{Rb} \mid \mathrm{b} \\
> &\mathrm{R} \rightarrow \mathrm{Sa} \mid \mathrm{a}
> \end{aligned} \tag{9}
> $$
>
> 令它的非终结符的排序为 $\mathbf{R}$、$\mathbf{Q}$、$\mathbf{S}$ ，求消除其左递归后的文法 .
>
> 按照上述算法可以得到下面的计算流程 :

```
i = { R , Q , S }

i == R  j = R （因为j最大到i-1） continue

i == Q  j = R 用R替换Q ， 消除直接左递归 （没有直接左递归）

i == Q  j = Q continue

i == S  j = R S不含R*的式子

i == S  j = Q 用Q替换S ， 然后消除直接左递归（有直接左递归）

i == S  j = S continue
```

对于 R , 不存在直接左递归。把 R 带入到 Q 的有关候选后,我们把 $\mathbf{Q}$ 的规则变为

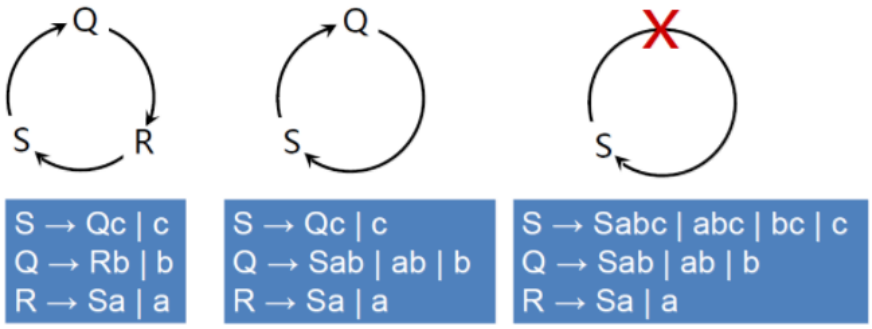
$$Q \rightarrow Sab|ab|b \tag{10}$$

现在的 Q 同样不含直接左递归,把它代入到 S 的有关候选后, S 变成

$$\mathrm{S} \rightarrow Sabc \mid abc \mid \mathrm{bc} \mid \mathrm{c} \tag{11}$$

经消除了 S 的直接左递归后,我们得到了整个文法为

$$\begin{aligned} &\mathrm{S} \rightarrow \mathrm{abcS}' \mid \mathrm{bcS}' \mid \mathrm{cS}' \\ &\mathrm{S}' \rightarrow \mathrm{abcS}' \mid \varepsilon \\ &\mathrm{Q} \rightarrow \mathrm{Sab} \mid \mathrm{ab} \mid \mathrm{b} \\ &\mathrm{R} \rightarrow \mathrm{Sa} \mid \mathrm{a} \end{aligned} \tag{12}$$



显然,其中关于 Q 和 R 的规则已是多余的(从开始符合S无法到达Q与R)。经化简后所得的文法是:

$$\begin{aligned} &\mathrm{S} \rightarrow \mathrm{abcS}' \mid \mathrm{lbcS}' \mid \mathrm{cS}' \\ &\mathrm{S}' \rightarrow \mathrm{abcS}' \mid \varepsilon \end{aligned} \tag{13}$$

注意,由于对非终结符排序的不同,最后所得的文法在形式上可能不一样。但不难证明, 它们都是等价的。

例如,若对文法(4.3)的非终结符排序选为 S、 Q、 R , 那么 , 最后所得的无左递归文法是:

$$\begin{aligned} &\mathrm{S} \rightarrow \mathrm{Qc} \mid \mathrm{c} \\ &\mathrm{Q} \rightarrow \mathrm{Rb} \mid \mathrm{b} \\ &\mathrm{R} \rightarrow \mathrm{bcaR}' \mid \mathrm{caR}' \mid \mathrm{aR}' \\ &\mathrm{R}' \rightarrow \mathrm{bca\,R}' \mid \varepsilon \end{aligned} \tag{14}$$

两个文法的等价性是显然的 (可以通过逐步替换得到上面的文法)。

4.2.3 消除回溯

消除回溯就必须保证 : 对文法的任何非终结符 , 当要它去匹配输人串时 , 能够根据它所面临的输入符号准确地指派它的一个候选去执行任务 , 并且此候选的工作结果应是确信无疑的。

假定现在轮到非终结符 A 去执行匹配任务 $\mathrm{A} \rightarrow \alpha_1 \left|\alpha_2\right| \cdots \mid \alpha_{\mathrm{n}}$ A 所面临的第一个输人符号为 a , A 能够根据不同的输人符号指派相应的候选 α_{i} 作为全权代表去执行任务 , 在这里 A 根据所面临的输人符号 a准确地指派唯一的一个候选。

4.2.3.1 FIRST 集合

令 G 是一个不含左递归的文法,对 G 的所有非终结符的每个候选 α 定义它的终结首符集 $\mathrm{FIRST}(\alpha)$ 为

$$\mathrm{FIRST}(\alpha)=\left\{\mathrm{a} \mid \alpha \stackrel{*}{\Rightarrow} \mathrm{a} \cdots, \mathrm{a} \in \mathrm{V}_{\mathrm{T}}\right\} \tag{15}$$

特别是,若 $\alpha \stackrel{*}{\Rightarrow} \varepsilon$, 则规定 $\varepsilon \in \mathrm{FIRST}(\alpha)$ 。

换句话说, $\mathrm{FIRST}(\alpha)$ 是 α 的所有可能推导的开头终结符或可能的 ε。如果非终结符 A 的所有候选首符集两两不相交, 即 A 的任何两个不同候选 α_{i} 和 α_i

$$\mathrm{FIRST}(\alpha_{\mathrm{i}}) \cap \mathrm{FIRST}(\alpha_{\mathrm{j}})=\phi \tag{16}$$

那么,当要求 A 匹配输入串时, A 就能根据它所面临的第一个输入符号 a,准确地指派某个候选前去执行任务。这个候选就是那个终结首符集含 a 的 α。

4.2.3.2 提取公共左因子

如何把一个文法改造成任何非终结符的所有候选首符集两两不相交呢？其办法是, 提取公共左因子。例如,假定关于 A 的规则是$\mathrm{A} \rightarrow \delta \beta_1 \left|\delta \beta_2\right| \cdots \left|\delta \beta_{\mathrm{n}}\right| \gamma_1 \left|\gamma_2\right| \cdots \gamma_{\mathrm{m}}$ （其中,每个 γ 不以 δ 开头)那么,可以把这些规则改写成

$$\begin{gathered}\mathrm{A} \rightarrow \delta \mathrm{A}^{\prime}\left|\gamma_1\right| \gamma_2\left|\cdots\right| \gamma_{\mathrm{m}} \\ \mathrm{A}^{\prime} \rightarrow \beta_1\left|\beta_2\right| \cdots \mid \beta_{\mathrm{n}}\end{gathered} \tag{17}$$

经过反复提取左因子,就能够把每个非终结符(包括新引进者)的所有候选首符集变成为两两不相交。我们为此付出的代价是 , 大量引进新的非终结符和 $\varepsilon-$ 产生式。

4.2.3.3 FOLLOW 集合

- 如果一个非终结符 A ，有多个候选，当面临着输入符号 a 要用 A 去匹配的时候，如果这个 a 又不在任何一个候选的首符集里面，但是，有一个候选是 ε 。当前如果 a 在某个句型中能够跟在 A 的后面时，才能选择 A 的 ε 去匹配这个 A。
- 假定S是文法G的开始符号，对于G的任何非终结符A，我们定义A的FOLLOW集合

$$FOLLOW(A)=\{a|S \stackrel{*}{\Rightarrow} \cdots Aa \cdots, a \in V_T\} \tag{18}$$

- 特别是，若 $S \stackrel{*}{\Rightarrow} \cdots A$，则规定 $\# \in FOLLOW(A)$
- 现在表述改为：当 a 要扩展 A 时，a 不出现在任何候选的 FIRST 集合里，而且有个 ε 候选，如果 a 在 A 的 FOLLOW 集合里面，就把 A 替换成 ε 。

4.3 LL(1)文法

通过上述对自上而下语法分析的一系列讨论，对于所存在问题的解决，可以找出满足**构造不带回溯的自上而下分析的文法**条件。

LL(1)文法的含义：

L	从左至右的扫描顺序
L	最左推导
(1)	每一步只向前查看一个符号

LL(1)文法应满足的条件：

1. 文法不含左递归
2. 对于文法中每一个非终结符A的各个产生式的候选首符集两两不相交。
   即 若 $A \to \alpha_1|\alpha_2|\ldots|\alpha_n$，则
   $FIRST(\alpha_i) \cap FIRST(\alpha_j) = \phi$
3. 对文法中的每个非终结符A，若它存在某个候选首符集包含ε，则
   $FIRST(A) \cap FOLLOW(A) = \phi, i = 1, 2, \ldots, n$

4.3.1 LL(1)分析法

有效的无回溯的自上而下分析

- 对于 LL(1) 文法，可以对其输入串进行有效的无回溯的自上而下分析
- 假设要用非终结符 A 进行匹配，面临的输入符号为 a，A 的所有产生式为 $A \to \alpha_1|\alpha_2|\ldots|\alpha_n$
    1. 若 $a \in FIRST(\alpha_i)$，则指派 α_i 执行匹配任务；
    2. 若 a 不属于任何一个候选首符集，则
        1. 若 ε 属于某个 $FIRST(\alpha_i)$ 且 $a \in FOLLOW(A)$， 则让 A 与 ε 自动匹配。
        2. 否则，a 的出现是一种语法错误。

4.3.2 FIRST与FOLLOW集的含义

4.3.2.1 FIRST集合的含义

FIRST(α)的含义是符号α的头符号集合：由α可推下去的所有打头的终结集。

4.3.2.2 FOLLOW集合的含义

FOLLOW(α)的含义是句型中，A出现时，其后的终结集

对于例题4.2给出LL(1)分析法所对应的判断表如下

	+	*	(	)	i	#
E			E→TE'		E→TE'	
E'	E→+TE'			E'→ ε		E'→ ε
T			T→FT'		T→FT'	
T'	T'→ ε	T'→*FT'		T'→ ε		T'→ ε
F			F→(E)		F→i	

FOLLOW 集推出

FIRST 集推出

否则报错

由上述表格我们很显然的推断，对于有推空的产生式而言，**FIRST集与FOLLOW集不可相交**（否则在某个非终结符面临终结符时，会在上述判断表中找到相应的两个操作：推空或者用某条产生式推，与我们想要构造**有效的、无回溯的自上而下分析**的目标相悖）

4.3.3 FIRST集与FOLLOW的构造

思想：把对无穷推导空间的可能的考察，转换成对有限产生式的反复扫描

4.3.3.1 构造 FIRST(α)

$$\mathrm{FIRST}(\alpha)=\left\{\mathrm{a}\mid \alpha \overset{*}{\Rightarrow} \mathrm{a}\cdots, \mathrm{a}\in \mathrm{V_T}\right\} \tag{19}$$

- 文法符号：$\alpha = X,\quad X \in V_T \cup V_N$
- 符号串：$\alpha = X_1X_2\ldots X_n,\quad X_i \in V_T \cup V_N$

构造每个文法符号的 FIRST 集合：

- 对每一 $X \in V_T \cup V_N$，连续使用下面的规则，直至每个集合 FIRST 不再增大为止：
  1. 若 $X \in V_T$，则 $FIRST(X) = X$
  2. 若 $X \in V_N$，且有产生式 $X \to a\cdots$，则把 a 加入到 $FIRST(X)$ 中；
     若 $X \to \varepsilon$ 也是一条产生式，则把 ε 也加到 $FIRST(X)$ 中
  3. - 若 $X \to Y\cdots$ 是一个产生式且 $Y \in V_N$，则把 $FIRST(Y)$ 中的所有非 ε 也加到 $FIRST(X)$ 中
     - 若 $X \to Y_1Y_2\ldots Y_{i-1}Y_i\ldots Y_k$ 是一个产生式，$Y_1, Y_2, \cdots, Y_{i-1}$ 都是非终结符
       - 对于任何 j，$1 \le j \le i-1$，$FIRST\left(Y_j\right)$ 都含有 ε （$Y_1\cdots Y_{i-1} \overset{*}{\Rightarrow} \varepsilon$），则把 $FIRST\left(Y_i\right)$ 中的所有非 ε 也加到 $FIRST(X)$ 中
       - 若所有的 $FIRST\left(Y_j\right)$ 都含有 ε，j=1,2,...,k，则把 ε 加到 $FIRST(X)$ 中

构造任何符号串的 FIRST 集合：

- 对文法 G 的任何符号串 $\alpha = X_1X_2\cdots X_n$ 构造集合 $FIRST\left(\alpha\right)$
  1. 置 $\mathrm{FIRST}(\alpha) = \mathrm{FIRST}\left(\mathrm{X_1}\right)\setminus\varepsilon$，斜杠表示集合去掉某个元素
  2. 若对任何 $1 \le j \le i-1$，$\varepsilon \in \mathrm{FIRST}\left(\mathrm{X} * \mathrm{j}\right)$，则把 $\mathrm{FIRST}\left(\mathrm{X} * i\right)\setminus\varepsilon$ 加至 $\mathrm{FIRST}(\alpha)$ 中；
     特别是，若所有的 $\mathrm{FIRST}\left(\mathrm{X_j}\right)$ 都含 ε （$1 \le j \le n$），则把 ε 也加至 $\mathrm{FIRST}(\alpha)$ 中。
     显然，若 $\alpha = \varepsilon$ 则 $\mathrm{FIRST}(\alpha) = \varepsilon$

4.3.3.2 构造 FOLLOW(A)

$$\mathrm{FOLLOW}\left(\mathrm{A}\right)=\left\{\mathrm{a}\mid \mathrm{S} \overset{*}{\Rightarrow} \cdots \mathrm{Aa}\cdots, \mathrm{a}\in \mathrm{V_T}\right\} \tag{20}$$

对于文法 G 的每个非终结符 A 构造 FOLLOW(A) 的办法是，连续使用下面的规则，直至每个 FOLLOW 不再增大为止：

1. 对于文法的开始符号 S，置 # 于 FOLLOW(S) 中；
2. 若 $A \to \alpha B\beta$ 是一个产生式，则把 FIRST(β)的非ε 加至 FOLLOW(B) 中；
3. 若 $A \to \alpha B$ 是一个产生式 或（$A \to \alpha B\beta$ 是一个产生式且$\beta \overset{*}{\Rightarrow} \varepsilon$ A≠B），则把 FOLLOW(A) 加至 FOLLOW(B) 中

若A→αB是一个产生式
∵ ∀ a ∈ FOLLOW(A), 有$S\overset{*}{\Rightarrow}$ ... Aa ...
∴ $S\overset{*}{\Rightarrow}$ ... Aa ...⇒... αBa ...
∴a ∈ FOLLOW(B)

A→αBβ是一个产生式而$\beta\overset{*}{\Rightarrow}\varepsilon$
∵ ∀ a ∈ FOLLOW(A), 有$S\overset{*}{\Rightarrow}$ ... Aa ...
∴ $S\overset{*}{\Rightarrow}$ ... Aa ...⇒... αBβa ... $\overset{*}{\Rightarrow}$ ... αBa ...
∴a ∈ FOLLOW(B)

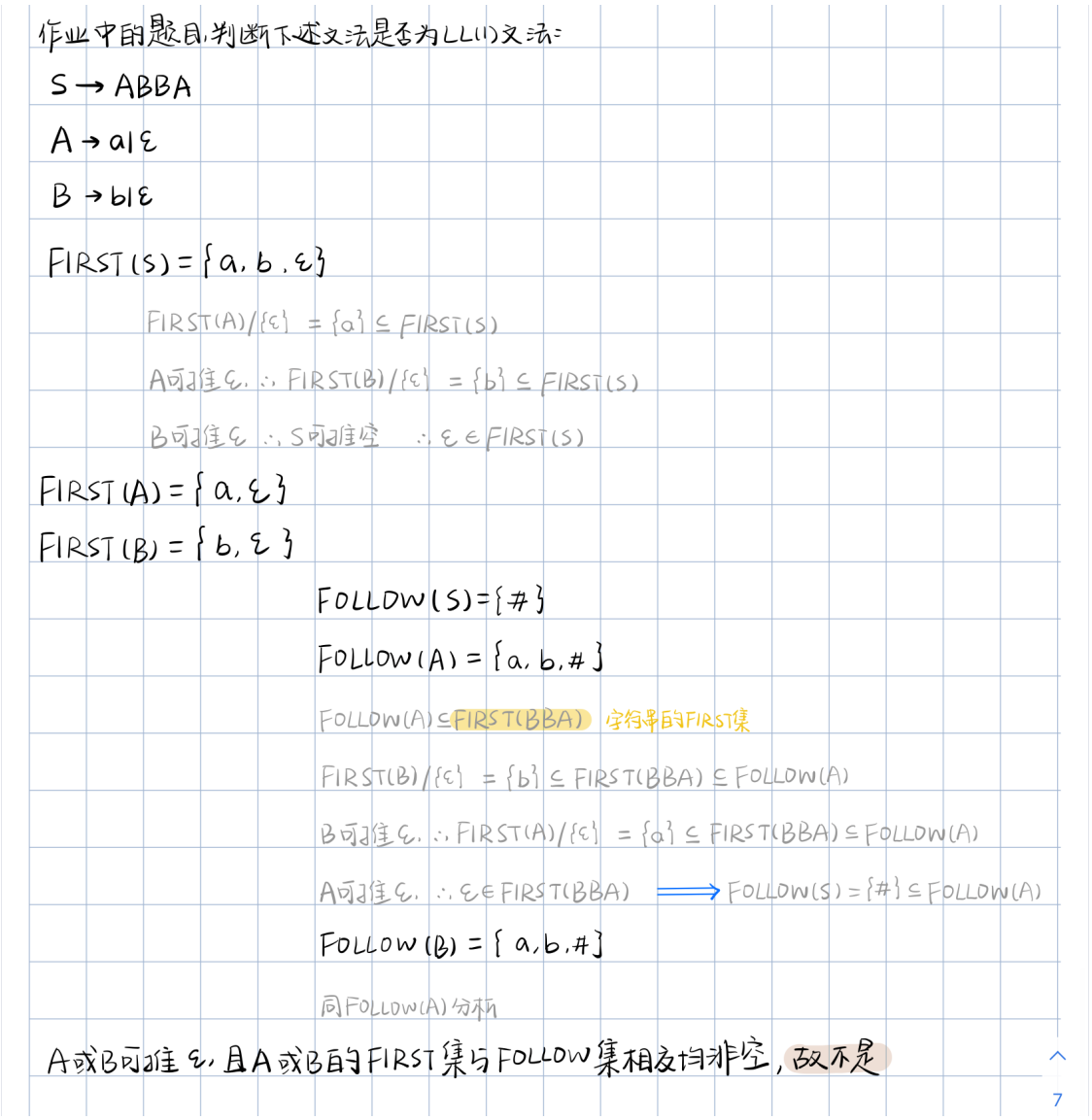
例题1：有如下文法：

$$
\begin{aligned}
&E \to TE' \\
&E' \to +TE'|\varepsilon \\
&T \to FT' \\
&T' \to *FT'|\varepsilon \\
&F \to (E)|i
\end{aligned}
$$

求每一个非终结符的FIRST和FOLLOW集：

$$
\begin{aligned}
&FIRST(E) = \{(, i\} && FOLLOW(E) = \{), \#\} \\
&FIRST(E') = \{+, \varepsilon\} && FOLLOW(E') = \{), \#\} \\
&FIRST(T) = \{(, i\} && FOLLOW(T) = \{+,), \#\} \\
&FIRST(T') = \{*, \varepsilon\} && FOLLOW(T') = \{+,), \#\} \\
&FIRST(F) = \{(, i\} && FOLLOW(F) = \{*, +,), \#\}
\end{aligned} \tag{21}
$$

例题2：



4.3.5 将非LL(1)文法改写为LL(1)文法

1. 消除左递归(直接/间接)
2. 消除回溯：反复提取公共左因子

tips : 并非所有文法都可以改造为LL(1) .

一个文法经过消除左递归 提取公共左因子后 如果某非终结符存在推空式且first和follow集相交 , 则不能改为LL(1)文法

> 例如 :
> 已知文法G , 它是否可以改为LL(1)文法
> 语句->if1 | if2
> if1 ->if X then 语句 else 语句
> if2 ->if X then 语句
>
> 消除左递归 : 没有左递归
> 提取公共左因子得到下面的文法 :
>
> 语句->if X then 语句 语句'
> 语句'-> else 语句 | 空
>
> 求该文法的first集合和follow集 :
>
> 语句 first : if
> 语句' first : else 空
> 语句 follow : # else
> 语句' follow : # else
>
> 语句'存在推空式 , 语句'的first集与follow集相交 , 所以不能改为LL(1)文法

4.4 递归下降分析程序构造

4.4.1 概念

- 递归下降分析程序由一组子程序组成， 对LL(1)文法的每一个非终结符编写一个相应的子程序，识别该非终结符所表示的语法成分
- 通过子程序之间相互调用实现对输入串的识别
    - 例如，E → E + T

4.4.2 递归子程序法

- 编写文法、消除二义性
- 消除左递归和提取左因子
- 检查是不是 LL(1) 文法
    - 若不是 LL(1),说明文法的复杂性超过自顶向下方法的分析能力
- 按照文法规则，编写递归下降分析程序
    - 为每个非终结符设置一个子程序，按照候选规则编写控制结构
    - 按照规则识别终结符，调用非终结符的子程序

4.4.3 递归下降分析程序

定义全局过程和变量

- ADVANCE，把输入串指示器IP指向下一个输入符号，即读入一个单词符号
- SYM，输入串指示器IP当前所指的输入符号
- ERROR，出错处理子程序

e.g.

- 文法：

$$
\begin{aligned}
&E \to TE' \\
&E' \to +TE'|\varepsilon \\
&T \to FT' \\
&T' \to *FT'|\varepsilon \\
&F \to (E)|i
\end{aligned}
$$

- 对应的递归下降子程序

```
主程序：
PROGRAM PARSER;
BEGIN
    ADVANCE;
    E;
    IF SYM <> '#' THEN ERROR;
END;
```

```
PROCEDURE T;
BEGIN
    F;T'
END;
```

```
PROCEDURE E';
IF SYM = '+' THEN
BEGIN
    ADVANCE;
    T;E'
END;
```

```
PROCEDURE T';
IF SYM = '*' THEN
BEGIN
    ADVANCE;
    F;T'
END;
```

```
PROCEDURE F;
IF SYM = 'i' THEN ADVANCE;
ELSE
    IF SYM = '('  THEN
    BEGIN
        ADVANCE;
        E;
        IF SYM = ')' THEN ADVANCE;
        ELSE ERROR;
    END;
    ELSE ERROR;
```

4.4.4 巴科斯范式

在元符号“→”或“::=”和“|”的基础上，扩充几个元语言符号

- 用花括号 {α} 表示闭包运算 α*。
- 用表示α_0^1可任意重复 0 次至 n 次。
- 用方括号 [α] 表示α_0^1 ，即表示 α 的出现可有可无 (等价于 α|ε)。
- 用扩充的巴科斯范式来描述语法，直观易懂，便于表示左递归消去和因子提取。

e.g.

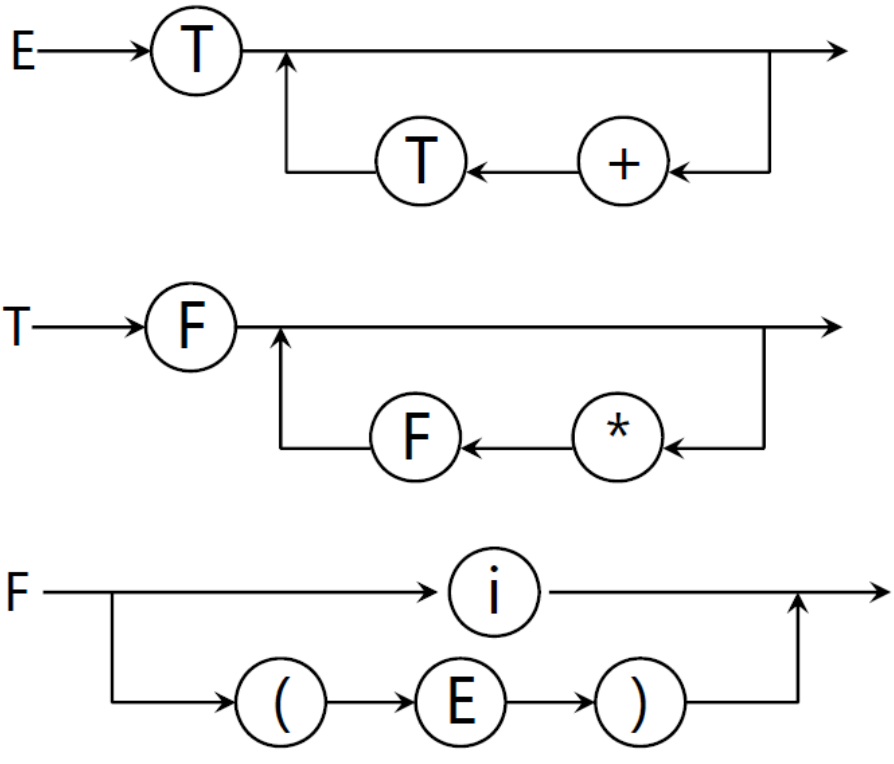
$$
\begin{aligned}
&E \to T|E+T\\
&T \to F|T*F\\
&F \to i|(E)
\end{aligned}
$$

可表示成

$$
\begin{aligned}
&E \to T\{+T\}\\
&T \to F\{*F\}\\
&F \to i|(E)
\end{aligned}
$$

4.4.5 语法图

4.4.4中第二个文法对应的语法图



4.4.6 递归子程序法优缺点分析

优点：

- 直观、简单、可读性好
- 便于扩充

缺点：

- 递归算法的实现 效率低
- 处理能力相对有限
- 通用性差，难以自动生成

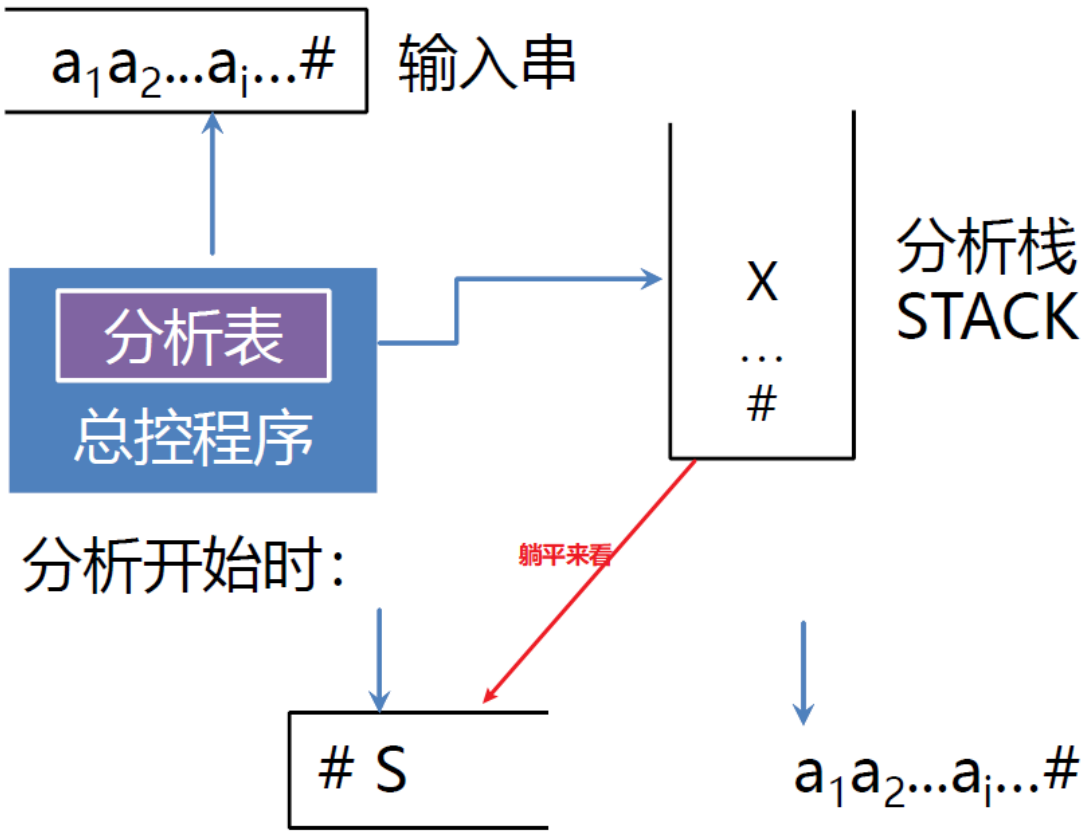
4.5 预测分析程序

4.5.1 预测分析程序法

预测分析程序构成

- 总控程序，根据现行栈顶符号和当前输入符号，执行动作
- 分析表 $M[A，a]$ 矩阵，$A \in V_N$，$a \in V_T$ 是终结符或‘#’
- 分析栈 STACK 用于存放文法符号

预测分析器模型



4.5.2 预测分析表构造

- 构造FIRST(α)和FOLLOW(A)
- 构造文法的分析表M[A，a]， 确定每个产生式A→α在表中的位置

  1)对文法 G 的每个产生式 A→α 执行第 2 步和第 3 步。

  2)对每个终结符 a∈FIRST(α)，把 A→α 加至 M[A，a] 中。

  3)若 ε∈FIRST(α)，则对任何 b∈FOLLOW(A) 把 A→α 加至 M[A，b] 中。

4)把所有无定义的 M[A，a] 标上“出错标志”。

e.g.

- 有如下文法：

$$
\begin{aligned}
&E \to TE' \\
&E' \to +TE'|\varepsilon \\
&T \to FT' \\
&T' \to *FT'|\varepsilon \\
&F \to (E)|i
\end{aligned}
$$

- 求每一个非终结符的FIRST和FOLLOW集：

$$
\begin{aligned}
&FIRST(E) = \{(, i\} && FOLLOW(E) = \{), \#\} \\
&FIRST(E') = \{+, \varepsilon\} && FOLLOW(E') = \{), \#\} \\
&FIRST(T) = \{(, i\} && FOLLOW(T) = \{+,), \#\} \\
&FIRST(T') = \{*, \varepsilon\} && FOLLOW(T') = \{+,), \#\} \\
&FIRST(F) = \{(, i\} && FOLLOW(F) = \{*, +,), \#\}
\end{aligned} \tag{22}
$$

- 构造预测分析表：

	id	$+$	$*$	$($	$)$	$\$$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T' \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to id$			$F \to (E)$		

4.5.3 预测分析过程

总控程序根据当前栈顶符号X和输入符号a，执行下列三动作之一：

- 若 X = a = ‘#’，则宣布分析成功，停止分析。
- 若 X = a ≠‘#’，则把 X 从 STACK 栈顶逐出，让 a 指向下一个输入符号。
- 若 X 是一个非终结符，则查看分析表 M。
  - 若 M[X，a] 中存放着关于 X 的一个产生式，把 X 逐出 STACK 栈顶，把产生式的右部符号串按反序一一推进 STACK 栈 (若右部符号为ε ，则意味不推什么东西进栈)。
  - 若 M[X，a] 中存放着“出错标志”，则调用出错诊察程序 ERROR。

```
BEGIN
    首先把‘#’然后把文法开始符号推进STACK栈；
    把第一个输入符号读进a；
    FLAG:=TRUE;
    WHILE FLAG DO
    BEGIN
        把STACK栈顶符号上托出去并放在X中；
        IF X∈VT THEN
```

```
            IF X= a THEN 把下一输入符号读进a
            ELSE ERROR
        ELSE IF X='#' THEN
            IF X=a THEN FLAG:=FALSE
            ELSE ERROR
        ELSE IF M[X,a]={X→X1X2…Xk}THEN
            把Xk,Xk-1,…,X1一一推进STACK栈 /* 若X1X2…Xk=ε，不推什么进栈*/
        ELSE ERROR
    END OF WHILE;
    STOP /*分析成功，过程完毕*/
END
```

e.g.

基于4.5.2中的文法和预测分析表，输入串为$i_1 * i_2 + i_3$

step	符号栈	输入串	所用产生式
0	$\#E$	$i_1 * i_2 + i_3\#$	
1	$\#E'T$	$i_1 * i_2 + i_3\#$	$E \to TE'$
2	$\#E'T'F$	$i_1 * i_2 + i_3\#$	$T \to FT'$
3	$\#E'T'i$	$i_1 * i_2 + i_3\#$	$F \to i$
4	$\#E'T'$	$*i_2 + i_3\#$	
5	$\#E'T'F*$	$*i_2 + i_3\#$	$T \to *FT'$
6	$\#E'T'F$	$i_2 + i_3\#$	
7	$\#E'T'i$	$i_2 + i_3\#$	$F \to i$
8	$\#E'T'$	$+i_3\#$	
9	$\#E'$	$+i_3\#$	$T' \to \varepsilon$
10	$\#E'T+$	$+i_3\#$	$E \to +TE'$
11	$\#E'T$	$i_3\#$	
12	$\#E'T'F$	$i_3\#$	$T \to FT'$
13	$\#E'T'i$	$i_3\#$	$F \to i$
14	$\#E'T'$	$\#$	
15	$\#E$	$\#$	$T' \to \varepsilon$
16	$\#$	$\#$	$E' \to \varepsilon$

4.5.4 预测分析法总结

- 编写文法，消除二义性
- 消除左递归、提取左因子(改写文法)
- 求 FIRST 集和 FOLLOW 集
- 检查是不是 LL(1) 文法

    - 若不是 LL(1),说明文法的复杂性超过自顶向下方法的分析能力
- 按照 LL(1) 文法构造预测分析表
- 实现预测分析器

- 5.1 自下而上分析基本问题　　制作人　徐容
- 5.2.1 算符优先分析　　制作人 杜雅莉
- 5.2.2 算符优先分析文法　　制作人 孙书镇

5.1自下而上分析基本问题

自下而上基本思想：从输入串开始，逐步进行“归约”，直到文法的开始符号，即从语法树的末端开始，一步一步向上构造语法树。

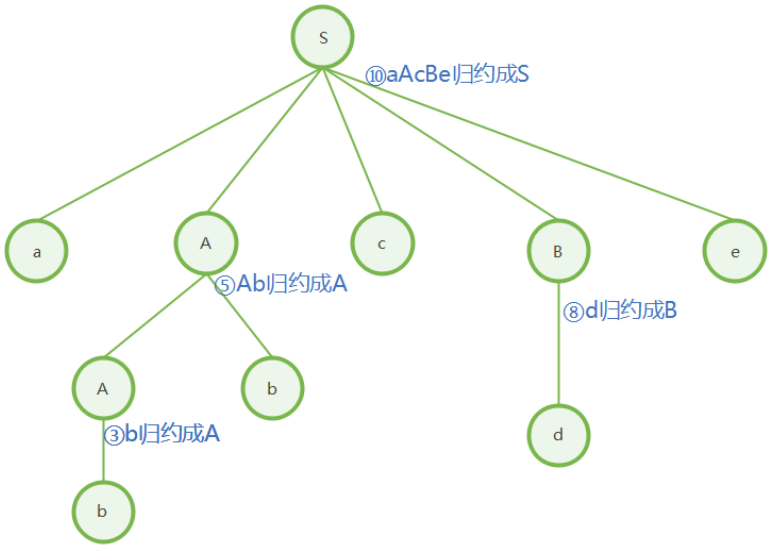
5.1.1归约

- **归约**指的是根据文法的产生式规则，把产生式的右部替换成左部符号。
- **可归约串**：归约的过程中把栈顶的一串符号用某个产生式的左部符号来代替，这样的一串符号就称为“可归约串”。
- 自下而上分析其实就是“**移进-归约**”法。使用一个寄存符号的先进后出栈，把输入串从左至右压入栈中，当栈顶形成某一个产生式的候选式，就把栈顶的这一部分替换成产生式的左部符号。
- 例1：移进归约
  文法G　(1)S->aAcBe　(2)A->b　(3)A->Ab　(4)B->d

  对符号串"abbcde"进行移进-归约。

  过程概述：把输入串“abbcde”从左至右压入栈中，当栈顶形成某一个产生式的候选式，就把栈顶的这一部分替换成产生式的左部符号，直到归约到文法的开始符号S。

用语法分析树表示上述语法分析过程



在上述语法分析过程中，步骤⑤根据产生式(3)，将栈顶的Ab归约成A，如果根据产生式(2)，将b归约成A，那么最终无法归约到文法的开始符号S。在步骤⑤中，“Ab”是可归约串和“b”是可归约串会产生不同的语法分析结果，故在自下而上分析中，需要精确地定义“可归约串”这个概念。

- **对“可归约串”不同的定义形成了不同的自下而上分析法。**在规范归约中，用“句柄”来刻画“可归约串”；在算符优先分析法中，用“最左素短语”来刻画“归约串”。

5.1.2规范归约简述

短语、直接短语、句柄

- 定义

令 G 是一个文法，S 是文法的开始符号，假定 αβδ 是文法 G 的一个句型，如果有

$$S \overset{*}{\Rightarrow} \alpha A \delta \text{ 且 } A \overset{+}{\Rightarrow} \beta$$

则称 β 是句型 αβδ 相对于非终结符 A 的短语。特别是，如果有

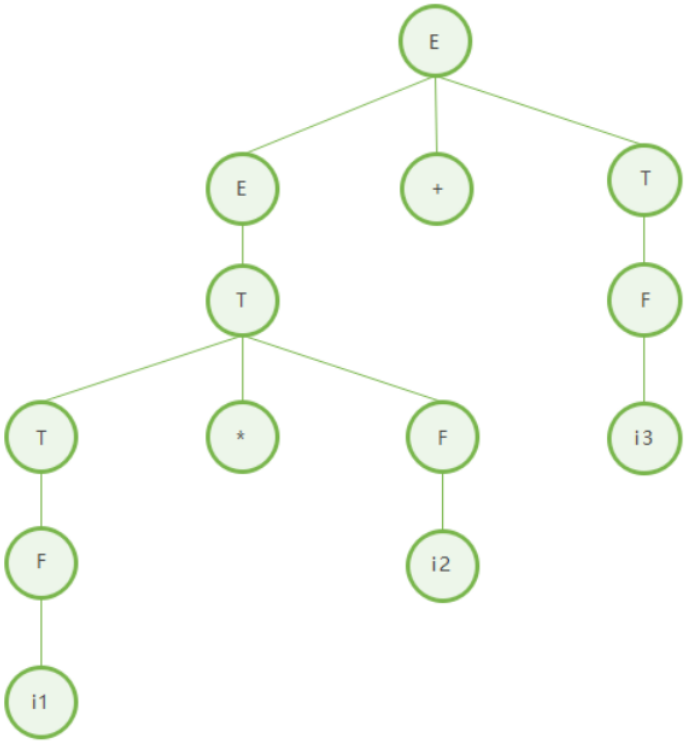
$$A \Rightarrow \beta$$

则称 β 是句型 αβδ 相对于规则 A→β 的直接短语，一个句型的最左直接短语称为该句型的句柄。

作为“短语”的两个条件均是不可缺少的。仅仅有 $A \overset{+}{\Rightarrow} \beta$，未必意味着 β 就是句型 αβδ 的一个短语。因为，还需有 $S \overset{*}{\Rightarrow} \alpha A \delta$ 这一条件。

- 从语法树角度理解：以某非终结符为根的**两代以上的子树的所有末端结点从左到右排列**就是相对于该非终结符的一个短语。如果子树只有两代，那么该短语就是直接短语。
- 例2：短语、直接短语、句柄
  文法　　E->T | E+T　　T->F | T*F　　F->i | (E)

  对于句型 i1 * i2 + i3，有以下语法树



短语：i1，i2，i3，i1 * i2，i1 * i2 +i3

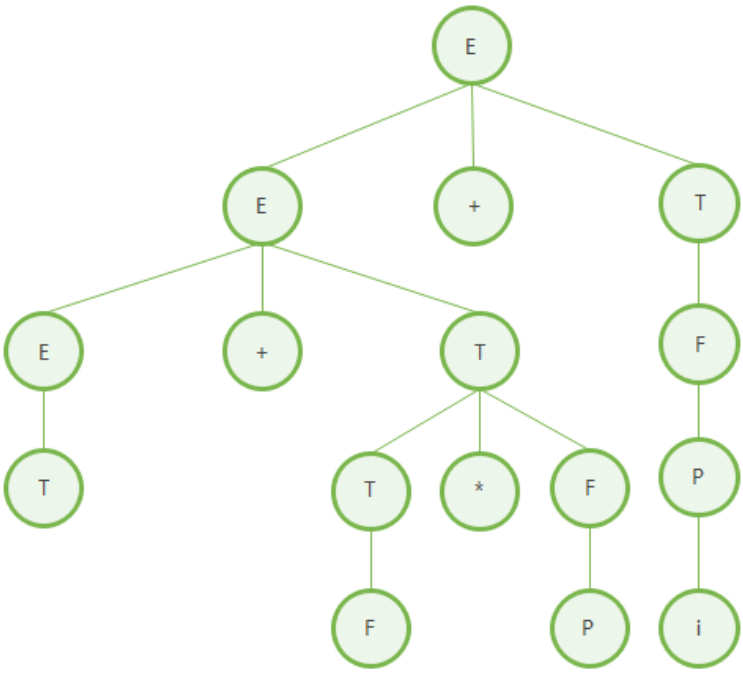
直接短语：i1，i2，i3

句柄（最左直接短语）：i1

素短语和最左素短语

- 定义
  素短语：（1）属于短语；（2）至少有一个非终结符；（3）除自身外不包含更小的素短语。
  最左素短语：处于句型最左边的素短语。
- 例3：各类短语的识别
  文法G(E)　(1) E->E+T | T　(2) T->T*F | F　(3) F->P↑F | P　(4) P->(E) | i

  对于句型T+F*P+i，有以下语法树



短语：T，F，P，i，F*P，T+F*P，T+F*P+i

直接短语：T，F，P，i

素短语：i，F*P

最左素短语：F*P

规范归约

- 规范归约是关于α的一个最右推导的逆过程。

假定 α 是文法 C 的一个句子，我们称序列

$$\alpha_n, \alpha_{n-1}, \cdots, \alpha_0$$

是 α 的一个**规范归约**，如果此序列满足：

(1) $\alpha_n = \alpha$；

(2) α_0 为文法的开始符，即 $\alpha_0 = S$；

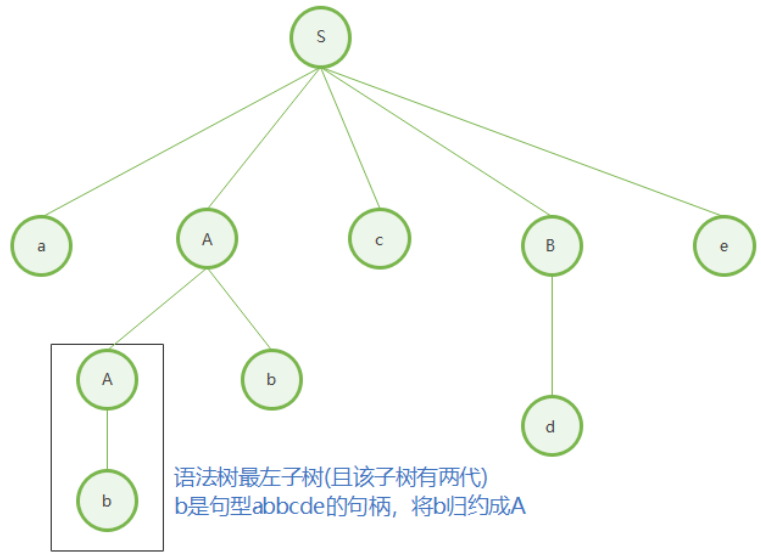
(3) 对任何 $i, 0 < i \leqslant n$，α_{i-1} 是从 α_i 经把句柄替换为相应产生式的左部符号而得到的。

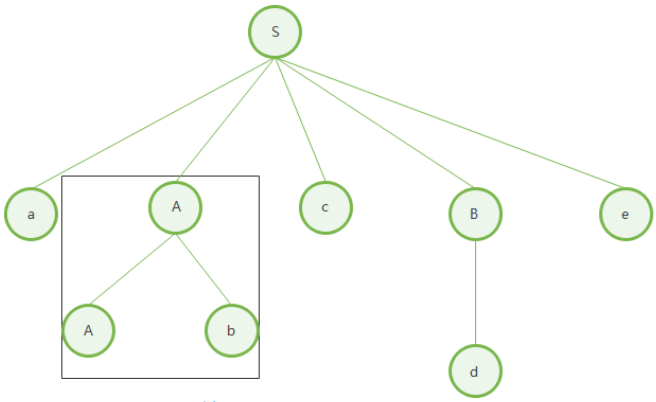
在例1的移进归约过程中，进行了四步归约，分别使用了文法中的(2)，(3)，(4)，(1)产生式。把产生式的使用顺序颠倒过来，即(1)，(4)，(3)，(2)，可以得到最右推导。

$$S \xrightarrow{(1)} aAcBe \xrightarrow{(4)} aAcde \xrightarrow{(3)} aAbcde \xrightarrow{(2)} abbcde$$

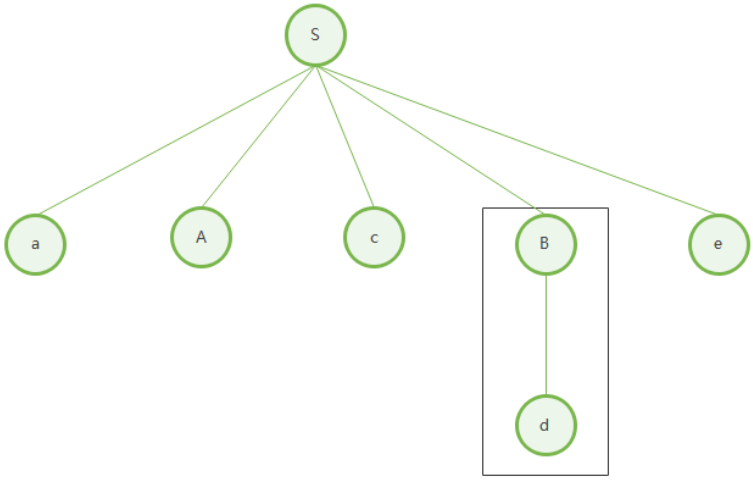
在形式语言中，最右推导常被称为规范推导。由规范推导所得的句型称为规范句型。如果文法G是无二义的，那么规范推导（最右推导）的逆过程一定是规范归约(最左归约）。

- 回顾5.1.1中的例1

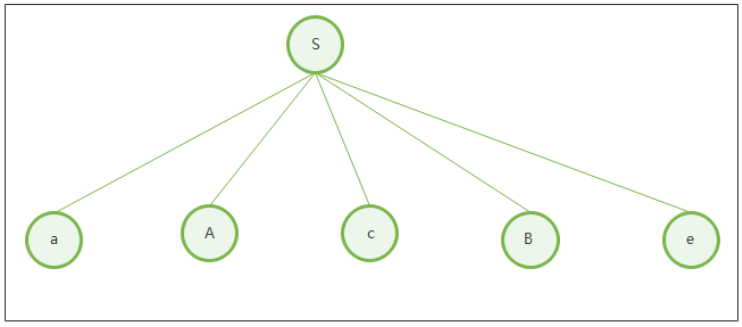




语法树最左子树(且该子树有两代)
Ab是句型aAbcde的句柄，将Ab归约成A



语法树最左子树(且该子树有两代)
d是句型aAcde的句柄，将d归约成B



语法树最左子树(且该子树有两代)
aAcBe是句型aAbcBe的句柄，将aAcBe归约成S

5.1.3 符号栈的使用与语法树的表示

- **分析过程中，符号栈和输入串的变化**

  在5.1.1的例1中使用了符号栈（用于寄存符号且先进后出）。在此后，将用一个不属于文法符号的特殊符号'#'作为栈底符，无条件地将其置在输入串之后，表示输入串的结束。

  分析开始时，符号栈和输入串ω的初始情形为

符号栈	输入串
#	ω#

分析器从左到右将输入串ω的符号依次移进符号栈中，当栈顶形成一个可归约串，就把这个串用相应的归约符号（在规范归约情况下用相应产生式左部符号）代替。这种替换可能持续多次，直到栈顶不再呈现可归约串为止。

重复上述过程，直到栈中只有'#'与文法开始符号S，输入串中只有结束符'#'，表示分析成功。如果不是下图所示格局，代表ω中含有语法错误。

- **语法分析对符号栈的四类操作**
  移进：将输入串的一个符号移进栈。
  归约：当栈顶形成一个可归约串，用相应符号去替换该可归约串。
  接受：宣布最终分析成功，是"归约"的一种特殊形式。
  出错处理：发现栈顶的内容和输入串相悖，分析工作无法正确进行，此时需要调用出错处理程序进行诊察和校正，并对栈顶内容和输入符号进行调整。
- **例4**
  文法　　E->T | E+T　　T->F | T*F　　F->i | (E)
  输入串 i1 * i2 + i3的规范归约步骤如下

步骤	符号栈	输入串	动作
0	#	$i_1 * i_2 + i_3$ #	预备
1	# i_1	$* i_2 + i_3$ #	进
2	# F	$* i_2 + i_3$ #	归，用 F→i
3	# T	$* i_2 + i_3$ #	归，用 T→F
4	# T *	$i_2 + i_3$ #	进
5	# T * i_2	$+ i_3$ #	进
6	# T * F	$+ i_3$ #	归，用 F→i
7	# T	$+ i_3$ #	归，用 T→T * F
8	# E	$+ i_3$ #	归，用 E→T
9	# E +	i_3 #	进
10	# E + i_3	#	进
11	# E + F	#	归，用 F→i
12	# E + T	#	归，用 T→F
13	# E	#	归，用 E→E + T
14	# E	#	接受

  步骤0时，符号栈里是#，输入串为i1 * i2 + i3 #
  步骤14时，符号栈里是#E（E为文法开始符号）,输入串被全部吸收，分析成功。
- **对于"归约"而言，任何可归约串的出现必然在栈的内部。规范归约是最右推导的逆过程，这种归约具有"最左"性，故可归约串必定在栈顶。**

5.2算符优先文法

5.2.1 算符优先分析

算符优先文法及优先表的构造 by dyl

1. 终结符之间的优先关系：

$a \lessdot b$　　a的优先级低于b

$a = b$　　a的优先级等于b

$a \gtrdot b$　　a的优先级高于b

2. 算符文法的定义：

一个文法，如果它的任一产生式的右部都不含两个相继（并列）的非终结符，即不含如下形式的产生式右部：…**QR**…，则我们称该文法为算符文法。

3. 算符优先文法的定义：

如果一个算符文法G中任何终结符对（a，b)**至多**只满足下述三关系之一，则称G是一个算符优先文法。

(1) $a \doteq b$ 当且仅当G中含有形如

$P \to \ldots ab \ldots$ 或 $P \to \ldots aQb \ldots$ 的规则。

(2) $a \lessdot b$ 当且仅当G中含有形如 $P \to \ldots aR \ldots$ 的规则，

且 $R \overset{+}{\Rightarrow} b \ldots$ 或 $R \overset{+}{\Rightarrow} Qb \ldots$

(3) $a \gtrdot b$ 当且仅当G中含有形如 $P \to \ldots Rb \ldots$ 的规则，

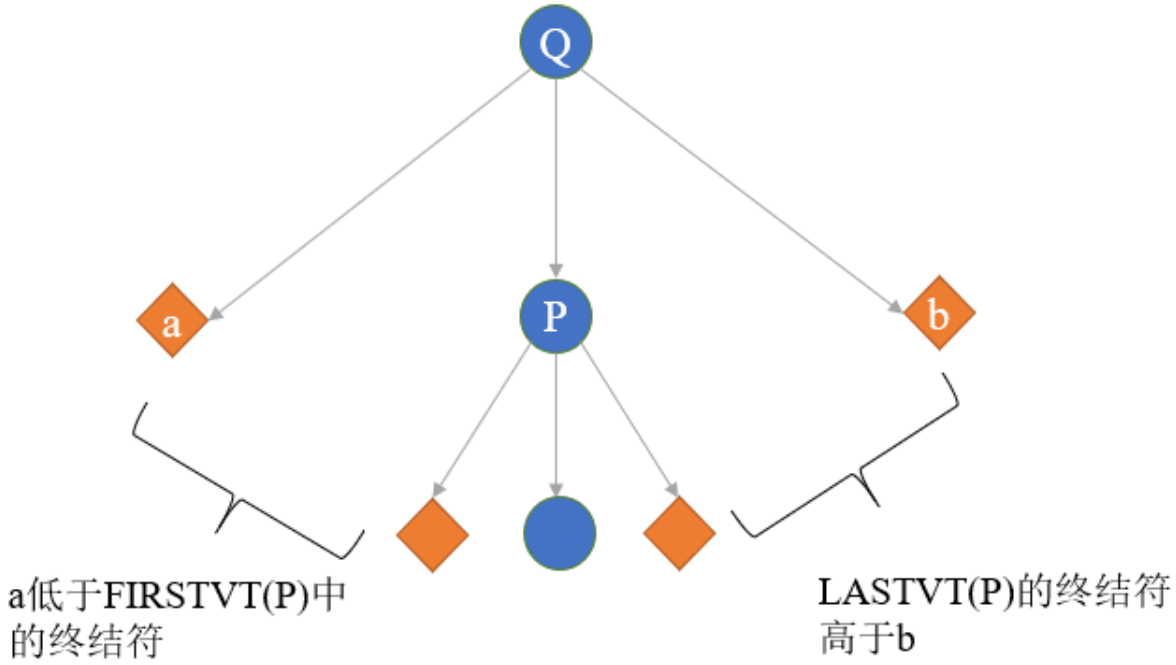
且 $R \overset{+}{\Rightarrow} \ldots a$ 或 $R \overset{+}{\Rightarrow} \ldots aQ$

4. 构造优先关系表：

1. 首先对G的每个非终结符P构造两个集合**FIRSTVT(P)**，**LASTVT(P)**：

2. 依次扫描产生式的右部，填优先关系表

- 对于终结符**a**、**b**,非终结符**P**，产生式右部出现 **…aPb…** 或 **…ab…**，则a等于b，填入优先关系表对应位置。
- 产生式右部出现 **…aP…** ，则对于任何**b**∈**FIRSTVT(P)**，有a低于b
- 产生式右部出现 **…Pb…** ，则对于任何**a**∈**LASTVT(P)**，有a高于b，高于低于的依据是先归约的高于后归约的终结符。



- 由于分析成功时，归约成句子 **#E#** ，非终结符**E**是开始符号，根据上述定义，有 **#** 等于 **#** ， **#** 低于**FIRSTVT(E)**，高于**LASTVT(E)**

5. 程序构造FIRSTVT(P):

```
根据下面两条规则构造集合FIRSTVT(P):
（1）若有产生式P→a···或P→Qa···，则a∈FIRSTVT(P)
（2）若a∈FIRSTVT(Q)，且有产生式P→Q···，则a∈FIRSTVT(P)
```

初始时建立一个布尔数组F[P,a],使得F[P,a]为真的条件是,当且仅当a∈FIRSTV(P)。开始时,按上述的规则(1)对每个数组元素F[P,a]赋初值。我们用一个栈STACK把所有初值为真的数组元素F[P,a]的符号对(P,a)全都放在STACK之中。然后,对STACK施行如下运算。
如果栈STACK不空,就将顶项逐出,记此项为(Q,a)。对于每个形如**P→→Q…** 的产生，若F[P,a]为假,则变其值为真且将(P,a)推进 STACK栈。
上述过程必须一直重复,直至栈STACK拆空为止。

如果把这个算法稍为形式化一点,我们可得如下所示的一个程序(包括一个过程和主程序)。

```
PROCEDURE INSERT(P,a);
  IF  NOT  F[P,a]  THEN
      BEGIN  F[P,a]:=TRUE;把(P,a)下推进 STACK 栈 END;
下面是主程序;
BEGIN
  FOR       每个非终结符 P 和终结符 a   DO F[P,a]:=FALSE;
  FOR       每个形如 P→a…或 P→Qa…的产生式 DO
                INSERT(P,a);
  WHILE STACK 非空 DO
      BEGIN
            把 STACK 的顶项,记为(Q,a),上托出去;
            FOR 每条形如 P→Q…的产生式 DO
                INSERT(P,a);
      END OF WHILE;
END
```

这个算法的·结果得到一个二维数组F，从它可以得到任何非终结符P的FIRSTVT
FIRSTVT(P)={ a|F[P,a]=TRUE }
同理可以构造LASTVT.

6. 程序构造优先表：

使用上述求得的FIRSTVT(P)和LASTVT(P)，就可以构造文法G的优先表，算法如下：

```
FOR 每条产生式 P→X1X2…Xn DO
      FOR  i:=1  TO  n-1 DO
           BEGIN
                IF Xi 和 Xi+1均为终结符 THEN 置 Xi = Xi+1
                IF i≤n-2 且 Xi 和 Xi+2都为终结符
                      但 Xi+1为非终结符 THEN 置 Xi = Xi+2;
                IF Xi 为终结符而 Xi+1为非终结符 THEN
                     FOR  FIRSTVT(Xi+1)中的每个 a  DO
                          置 Xi ⋖ a;
                IF Xi 为非终地符而 Xi+1为终结符 THEN
                     FOR  LASTVT(Xi)中的每个 a  DO
                          置   a ⋗ Xi+1
           END
```

7. 例子：

文法：

$S \to a \mid \wedge \mid (T)$

$T \to T, S \mid S$

① 相连FIRSTVT和LASTVT：

由 $S \to a \mid \wedge \mid (T)$，将 a、∧、(加入 S 的 FIRSTVT(S)，将 a、∧、) 加入 S 的 LASTVT(S)

由 $T \to T, S$，将 , 加入 T 的 FIRSTVT(T)，将 , 加入 T 的 LASTVT(T)

由 $T \to S$，将 S 的 FIRSTVT(S) 加入 T 的 FIRSTVT(T)．

将 S 的 LASTVT(S) 加入 T 的 LASTVT(T)

即 FIRSTVT(S)={a, ∧, (}　LASTVT(S)={a, ∧,)}

FIRSTVT(T)={, , a, ∧, (}．　LASTVT(T)={, , a, ∧,)}

② 扫描产生式，相连优先关系表．

- 首先，因为分析开始时，栈底先放入了一个'#'，同时假定输入串之后也总有一个'#'，标志输入串结束．当栈底的'#'和输入串末尾的'#'相遇的时候，则该字符串匹配成功．对于该文法，有 #S#．所以 # 低于 FIRSTVT(S) 中的终结符．LASTVT(S) 中终结符高于 #，# ≐ #
- 扫描产生式 $S \to a \mid \wedge \mid (T)$．先找到等于关系的终结符 '(' 和 ')'．

  由 S→(T) 中的'(T'，可知 (低于 FIRSTVT(T) 中的终结符．

  由 'T)' 可知，　LASTVT(T) 中的终结符高于)
- 扫描产生式 $T \to T, S$．由 'T,' 可知，LASTVT(T) 中的终结符高于 ,，由 ',S' 可知，, 低于 FIRSTVT(S) 中的终结符．

所以优先关系表为：

	a	∧	(	)	,	#
a				>	>	>
∧				>	>	>
(	<	<	<	≐	<	
)				>	>	>
,	<	<	<	>	>	
#	<	<	<			≐

5.2.2 算符优先分析文法

这一节主要讲解在利用上一节得到的优先表的基础上获得进行自下而上语法分析的过程，核心过程就是寻找**可规约串**；

首先，我们需要复习几个重要的概念：

> - 句型：假定G是一个文法,S是它的开始符号,如果 S =*> α 则称α是一个句型;
> - 短语：设有文法G，S是开始符号，设a b d是G的一个句型，若有S=*>a A d且A=+>b则称b是句型a b d关于非终结符A的短语。
> - 直接短语：在上面定义中，如果A直接推出b,即A=>b，则称b是句型a b d关于A->b的直接短语。
> - 句柄：一个句型的最左直接短语称为句柄。
> - 规范归约：规范归约是关于是一个最右推导的逆过程

值得注意的是短语的定义，不可忽视"S=*>a b d"这一条件，即使存在A可将b规约为A，但规约后的a A d无法由开始符号推出，也不能将b称为短语；

为了刻画"可规约串"，我们使用上述概念提出"素短语"和"最左素短语"的概念：

> - 素短语：一个文法G的句型的素短语是指这样一个短语，它至少含有一个终结符，并且除它自身之外不再含任何更小的素短语。
> - 最左素短语：指处于句型最左边的那个素短语。

本节的算符优先分析法，就是通过寻找最左素短语来确定可归约串的，找到一个最左素短语，就可以对其进行规约；

那么如何寻找最左素短语呢?

算符优先文法句型(括在两个 # 之间)的一般形式写成

#N1 a1 N2 a2 ... Nn an Nn+1#

其中每个ai都是终结符，Ni是可有可无的非终结符，每两个终结符之间最多只能有一个非终结符；

在以上形式下，我们可以依靠以下定理寻找该句型中的最左素短语

定理：一个算符优先文法G的任何句型的最左素短语是满足如下条件的最左子串：

$$\begin{gathered} a_{j-1} < a_j \\ a_j = a_{j+1} = a_{j+2} \ldots = a_{i-1} = a_i \\ a_i > a_{i+1} \end{gathered}$$

(注：本文中>、=、<均代表中间加点形式)

简单来说就是，这个串里：

最左边的终结符"大于"它左边的终结符；

最右边的终结符"大于"它右边的终结符；

串内终结符都"相等"；

不停地找到这样的串，将其规约，即完成自下而上语法分析；

通过一个伪代码，我们可以更清晰的理解这一过程：

```
k=1;
S[k]=‘#’;
REPEAT
    把下一个输入符号读进a中；//读入一个新符号
    若 S[k] 是终结符则 j=k //记录a左边最后一个终结符的位置j，终结符之间最多隔一个非终结符，所以k不是终结符时k-1一定是
    否则 j=k-1;
    WHILE S[j]>a          //当进入一个“小于”最后一个终结符的 终结符‘a’ 时，由前文第二条性质得此时a前面至少有一条可规约串
    DO                    //此时a左边第一个符号s[k]就是该串的最右端，我们只需要再找到最左端即可进行一次规约
        BEGIN
        REPEAT            //不停地找前面的终结符，在规约串中的非终结符都应该“等于”自己右边的非终结符
            Q=S[j];
            若 S[j-1]是终结符则 j=j-1
            否则 j=j-2
        UNTIL S[j]< Q;  //当找到“小于”自己右边的非终结符的非终结符时，就找到了规约串的最左端，可以规约了
        把S[j+1]…S[k]归约为某个N；//N并不重要，我们的分析只需要非终结符
        k=j+1;            //规约后，j+1的位置成了串的最右端，更新k
        S[k]=N            //该位置现在是刚刚规约得到的非终结符
    END OF WHILE;         //前面的可规约串可能不只一个，要等到a左边的终结符“大于”a才说明全部找完了

    IF S[j]<a OR S[j]=a THEN//结束后，a前面的最后一个终结符必须是“大于”a的
    BEGIN k:=k+1;S[k]:=a END
    ELSE ERROR          //调用出错诊察程序
UNTIL a=‘#’             //扫描到尾部的#，说明正常结束
```

在算法的工作过程中，若出现j减1后的值小于等 于0时，则意味着输入串有错。

在正确的情况下， 算法工作完毕时，符号栈S应呈现：# N #

值得注意的是，算符优先分析一般并不等价于规范归约；

由于算符优先文法跳过了许多非终结符的规约，使它比规范规约快得多；

但是也正因为忽视了非终结符在规约中的作用，使它可能将错误的输入串误认为是句子；

例题：

3.13 设有文法 G[S]: S→a|b|(A)
A→SdA|S

(1) 构造算符优先关系表；
(2) 给出句型(SdSdS)的短语、简单短语、句柄、素短语和最左素短语；
(3) 给出输入串(adb)#的分析过程。

这里省略前两步直接给出算符优先表，请结合此表对(adb)#进行分析：

	a	b	(	)	d	#
a				⩾	⩾	⩾
b				⩾	⩾	⩾
(	⩽	⩽	⩽	=	⩽	
)				⩾	⩾	⩾
d	⩽	⩽	⩽	⩾	⩽	
#	⩽	⩽	⩽			=

结果如下：

符号栈	输入串	说　明
#	(adb)#	移进
#(	adb)#	移进
#(a	db)#	用 S→a 归约
#(S	db)#	移进
#(Sd	b)#	移进
#(Sdb	)#	用 S→b 归约
#(SdS	)#	用 A→S 归约
#(SdA	)#	用 A→SdA 归约
#(A	)#	移进
#(A)	#	用 S→(A)归约
#S	#	分析成功

方法就是不断取输入字符串的栈顶和当前符号比较优先级，若是<或者=的关系就是移进的操作，如果是>的关系就是归约的关系

> 刘千一：LR文法开头——LR(0)文法中的“构造识别活前缀的DFA“中的"LR(0)项目"（含）
> 李博远：LR(0)文法中的“构造识别活前缀的DFA“中的"LR(0)项目"（含）——SLR分析表的构建(全部)
> 黑乃磊：LR(1)分析表 —— LR分析小节（全部）

LR文法

> 1965年由Knuth提出.**Donald Ervin Knuth**，1974 *ACM* A.M *Turing Award*，“For his major contributions to the analysis of algorithms and the design of programming languages, and in particular for his contributions to"*The Art of Computer Programming*" through his well-known books in a continuous series by this title. ”

LR(k)的语法分析概念

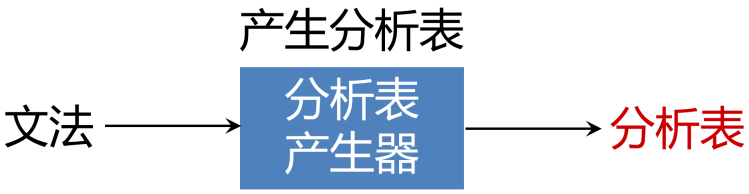
- L：从左到右扫描输入串
- R：反向构造出最右推导
- k：最多向前看k个符号

当k增大时，相应的语法分析器的规模急剧增大

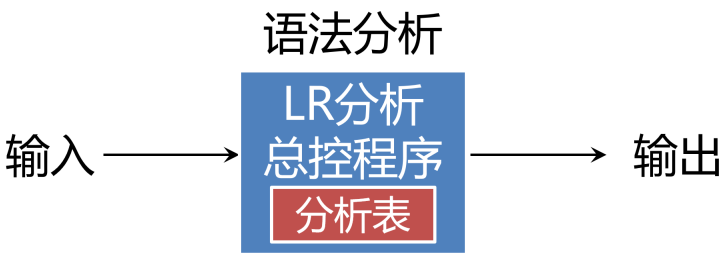
- k=2时，程序语言的语法分析器的规模通常非常庞大
- 当k=0，1时，已经可以解决很多语法分析问题，因此具有实践意义
- 我们仅考虑$k \leq 1$的情况

LR分析法的工作框架：

（1）分析表产生：文法经过分析表产生器产生分析表



（2）语法分析：输入经由LR分析总控程序得到输出



LR分析法概述

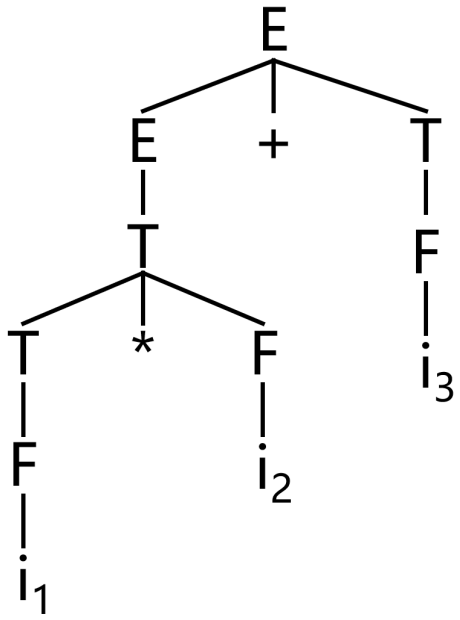
短语、直接短语和句柄

- 一个句型的最左直接短语称为该句型的句柄
- 在一个句型对应的语法树中
  - 以某非终结符为根的两代以上的子树的所有末端结点从左到右排列就是相对于该非终结符的一个短语
  - 如果子树只有两代，则该短语就是直接短语
  - 最左两代子树末端就是句柄

详细内容如下表格：

名称	概念
短语	**定义：** 若 S 为文法 G 的开始符号，αβδ 是该文法的一个句型，即 S ⇒* αβδ，且有 A ⇒+ β，则称 β 是句型 αβδ 相对于非终结符 A 的短语。**语法树：** 在语法树中表示为：以某非终结符为根的两代以上的子树，其短语即该子**树所有叶子节点**左到右排列构成的**终结符序列。注：**子树包括语法树本身，及句型本身也可以称为短语。
直接短语	**定义：** 若 S ⇒* αβδ，且文法中包含产生式 A → β，则称 β 是句型 αβδ 相对于非终结符 A 的直接短语。**语法树：** 在语法树中，某短语的子树**只有两代**，则该短语就是直接短语
句柄	“可规约串”，句柄对应某个产生式的右部，是某个，但不是任意一个。作为一种规约对象，**句柄表示最左直接短语。语法树：** 在语法树上，则表示为最左边的只包含相邻父子节点的短语（最左两代子树末端就是句柄）
素短语	**定义：** 是指一个短语至少包含一个终结符，并且除它自身之外不再包含其他素短语
最左素短语	**定义：** 最左素短语就是句型最左边的素短语，是算符优先分析法的规约对象。**语法树：** 通过语法树分析时，要注意先判断是否为素短语，再找相对最左端的素短语。

【例题】请找出下图中的短语、直接短语、句柄。



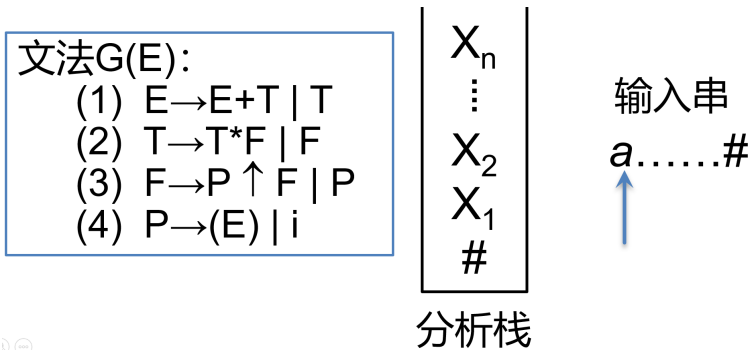
【解答】短语：i_1,i_2,i_3,$i_1 * i_2$,$i_1 * i_2 + i_3$

直接短语：i_1,i_2,i_3

句柄：i_1

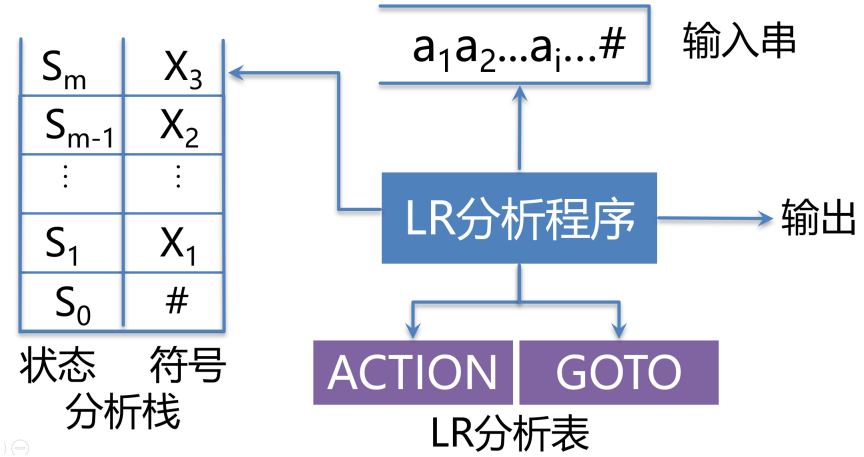
规范规约VS. 句柄

- 规范规约：假定α是文法G的一个句子，我们称序列$\alpha_n, \alpha_{n-1}, \ldots, \alpha_0$是$\alpha$的一个规范规约，如果此序列满足：
  - 1.$\alpha_n = \alpha$
  - 2.α_0为文法的开始符合，即$\alpha_0 = S$
  - 3.$\forall i, 0 \leq i \leq n$,$\alpha_{i-1}$是从$\alpha_i$经把句柄替换成为相应产生式左部符号而得到的
- 规范归约是最左归约
- 规范归约的逆过程就是最右推导
- 最右推导也称为规范推导
- 由规范推导推出的句型称为规范句型
- 规范规约的关键是寻找句柄
  - 历史：已移入符号栈的内容
  - 展望：根据产生式预测未来可能遇到的输入符号
  - 现实：当前的输入符号



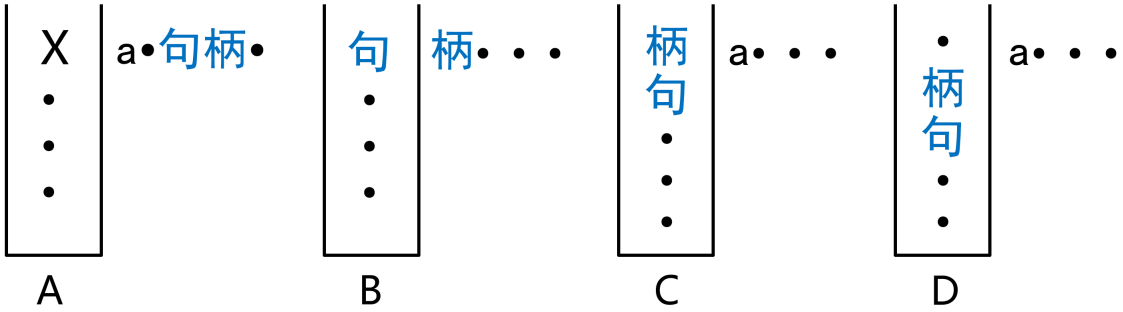
LR分析器

- LR分析方法：把"历史"及"展望"综合抽象成状态；由栈顶的状态和现行的输入符号唯一确定每一步工作



- LR分析器的性质：
  - 栈内的符号串和扫描剩下的输入符号串构成了一个规范句型
  - 一旦栈的顶部出现可归约串(句柄)，则进行归约

【例题】对于句子，在规范归约过程中，栈内的符号串和扫描剩下的输入符号串构成了一个规范句型，下面哪种格局不会出现：



【解答】D。由LR分析器的性质2可知，当栈的顶部出现可规约串则进行规约，D项不符合。由此可得关于性质2的一条推论：栈内永远不会出现句柄之后的符号。

- LR分析器的优点：
  - 由表格驱动，虽然手工构造表格工作量很大，但表格可以自动生成
  - 对于几乎所有的程序设计语言，只要写出上下文无关文法，就能够构造出识别该语言的LR语法分析器
  - 是最通用的无回溯移进-规约分析技术
  - 能分析的文法比LL(k)文法更多（关于文法之间的关系参加下文"文法类的谱系"部分）

LR分析表

LR分析器的核心是一张分析表

- `ACTION[s,a]`:当状态s面临输入符号a时，应采取什么操作
- `GOTO[s,X]`:当状态s面对文法符号X时，下一状态是什么

例如，分析表为：

	ACTION						GOTO		
状态	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			

该表中所引用的记号的意义是：

- `sj`（*shift*，移进）：把**下一状态**S_j和输入符号a分别推进状态栈和符号栈，下一输入符号变成现行输入符号.
- `rj`（*reduce*，归约）：用文法中**第`j`个产生式**$A \to \beta$进行归约。假若β的长度为r,则去除栈顶r个元素，使状态S_{m-r}变成栈顶状态，然后把下一状态$s' = GOTO[S_{m-r}, A]$和文法符号A分别推进状态栈和符号栈.
- acc（*accept*，接受）:宣布分析成功，停止分析器工作。
- 空白格：出错标志，报错

LR分析过程

- 分析开始时：

状态栈	符号栈	输入串
S_0	#,	$a_1 a_2 \dots a_n \#)$

- 以后每步的结果可以表示为：

$$(s_0 s_1 \dots s_m, \#X_1 \dots X_m, a_i a_{i+1} \dots a_n \#)$$

- 分析器根据$ACTION(s_m, a_i)$确定下一步动作

- 若$ACTION(s_m, a_i)$为移进，且s为下一状态，则格局变为：
  $(s_0 s_1 \dots s_m s, \#X_1 \dots X_m a_i, a_{i+1} \dots a_n \#)$
- 若$ACTION(s_m, a_i)$为按$A \to \beta$规约，格局变为：
  $(s_0 s_1 \dots s_{m-r} s, \#X_1 \dots X_{m-r} A, a_i a_{i+1} \dots a_n \#)$,此处$s = GOTO(s_{m-r}, A)$,r为$\beta$的长度，$\beta = X_{m-r+1} \dots X_m$。详细推导过程为：

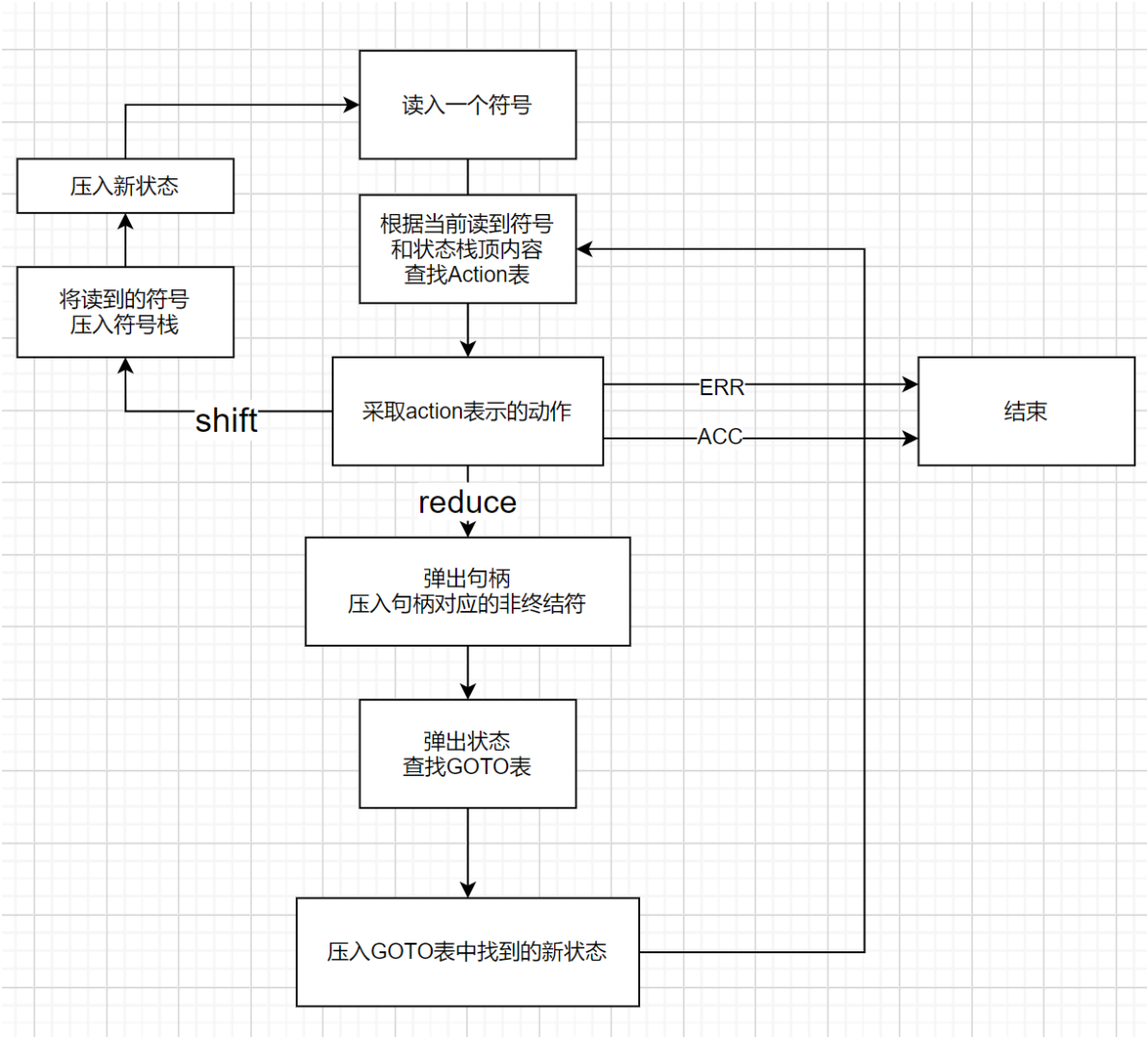
  （1）将原格局写成
  $(s_0 s_1 \dots s_{m-r} s_{m-r+1} \dots s_m, \#X_1 \dots X_{m-r} X_{m-r+1} \dots X_m, a_i a_{i+1} \dots a_n \#)$

  （2）将$\beta = X_{m-r+1} \dots X_m$以及相应的状态弹出栈，格局变为：
  $(s_0 s_1 \dots s_{m-r}, \#X_1 \dots X_{m-r}, a_i a_{i+1} \dots a_n \#)$

  （3）将符号A压入栈，格局变为$(s_0 s_1 \dots s_{m-r}, \#X_1 \dots X_{m-r} A, a_i a_{i+1} \dots a_n \#)$

  （4）由于当前状态栈顶为s_{m-r},栈顶符号为A，所以将状态$s = GOTO(s_{m-r}, A)$压入栈，格局变成$(s_0 s_1 \dots s_{m-r} s, \#X_1 \dots X_{m-r} A, a_i a_{i+1} \dots a_n \#)$
- 若$ACTION(s_m, a_i)$为“接受”，则格局变化过程终止，宣布分析成功。
- 若$ACTION(s_m, a_i)$为“报错”，则格局变化过程终止，报告错误。

- 完整分析过程可被表示成下图:



【例题】针对给出的文法G（E）：

$$(1)E \to E+T$$
$$(2)E \to T$$
$$(3)T \to T*F$$
$$(4)T \to T$$
$$(5)F \to (E)$$
$$(6)F \to i$$

其分析表已在"LR分析表"部分给出，试分析输入串$i*i+i$。

【解答】LR分析器的工作过程(即，三元式的变化过程)如下：

步骤	状态	符号	输入串
(1)	0	#	i*i+i#
(2)	05	#i	*i+i#
(3)	03	#F	*i+i#
(4)	02	#T	*i+i#
(5)	027	#T*	i+i#
(6)	0275	#T*i	+i#
(7)	02710	#T*F	+i#
(8)	02	#T	+i#
(9)	01	#E	+i#
(10)	016	#E+	i#
(11)	0165	#E+i	#
(12)	0163	#E+F	#
(13)	0169	#E+T	#
(14)	01	#E	#
(15)	接受		

LR文法

- 定义：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，则这个文法就称为LR文法。
- 定义：一个文法，如果能用一个每步顶多向前检查k个输入符号的LR分析器进行分析，则这个文法就称为LR(k)文法。

LR文法与二义文法

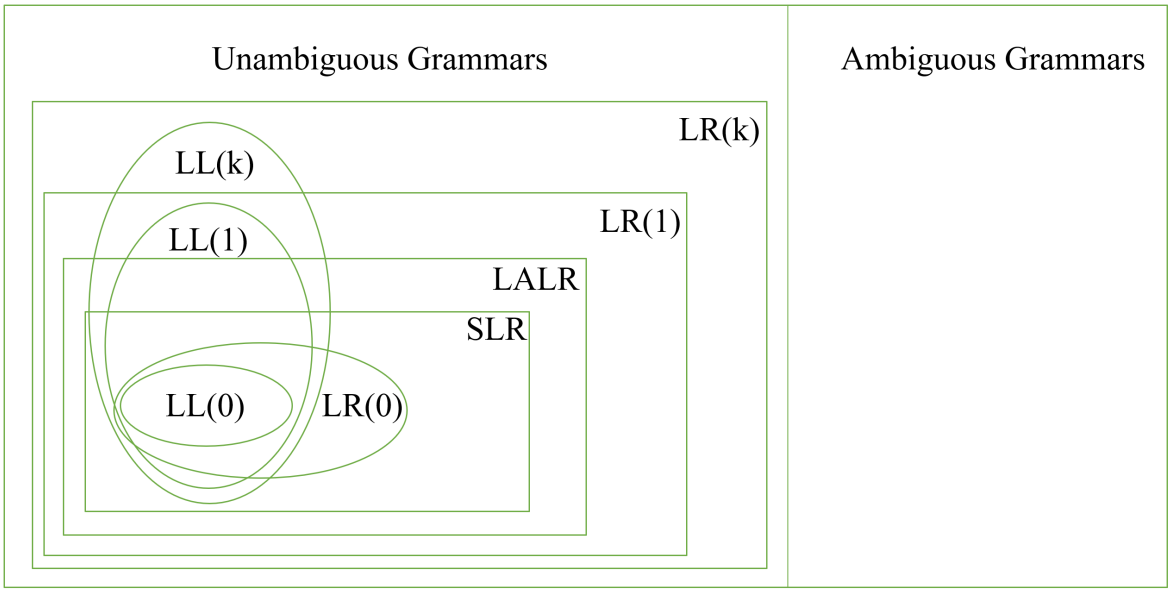
- LR文法不是二义的，二义文法肯定不会是LR的
- LR文法⊂无二义文法
- 非LR结构举例

  $s \to iCtS|iCtSeS$,假定有一个自下而上分析器，它正处于如下情形：

栈	输入
$\#\ldots iCtS$	$e\ldots\#$

此时我们无法肯定$iCtS$是否是一个句柄，无论在它之下栈所含的内容是什么。此时有两种可能：（1）把$iCtS$规约成S（2）把e移进，期待另一个S。此时不知道该采取哪个动作，因此该文法不是LR（1）的。

文法类的谱系



LR(0)文法

只概括“历史”而不“展望”。

活前缀

- 字的前缀：是指字的任意首部，如字*abc*的前缀有ϵ，*a*，*ab*，*abc*
- 活前缀：是指规范句型的一个前缀，这种前缀不含句柄之后的任何符号。即，对于规范句型$\alpha\beta\delta$，β为句柄，如果$\alpha\beta = u_1u_2\ldots u_r$，则符号串$u_1u_2\ldots u_i(1 \leq i \leq r)$是$\alpha\beta\delta$的活前缀。($\delta$必为终结符串)
- 规范归约过程中，保证分析栈中总是活前缀，就说明分析采取的移进/归约动作是正确的
- 对于一个文法G, 可以构造一个DFA,它能识别G的所有活前缀

构造识别活前缀的DFA

文法的拓广

- 将文法G(S)拓广为G'(S')
    - 构造文法G'，它包含了整个G，并引进不出现在G中的非终结符S'、以及产生式S'→S，S'是G'的开始符号
    - 称G'是G的拓广文法

LR(0)项目

- 文法的一个产生式加上在其中某处的一个点
- $A \to XYZ$有四个项目
  - $A \to \bullet XYZ$, $A \to X \bullet YZ$, $A \to XY \bullet Z$, $A \to XYZ \bullet$
- 注意，产生式$A \to \epsilon$只对应一个项目$A \to \bullet$
- 直观含义
  - 项$A \to \alpha \bullet \beta$表示已经扫描/规约到了$\alpha$，并期望在接下来的输入中经过扫描/规约得到$\beta$，然后把$\alpha\beta$规约到$A$
  - 如果β为空，表示我们可以把α规约为A
- 项也可以用一对整数表示：（i，j）表示第i条产生式，点位于右部第j个位置
- $A \to \alpha \bullet$ 称为"归约项目"
- 归约项目 $S' \to \alpha \bullet$ 称为"接受项目"
- $A \to \alpha \bullet a\beta (a \in V_T)$ 称为"移进项目"
- $A \to \alpha \bullet B\beta (B \in V_N)$称为"待约项目"

项目——NFA——DFA

1. 若状态i为$x \to x_1 \dots x_{i-1} \bullet x_i \dots x_n$，状态$j$为$x \to x_1 \dots x_i \bullet x_{i+1} \dots x_n$，那么可以从状态$i$拉一条弧到$j$，弧上跳的是$x_i$
2. 若状态i为$x \to \alpha \bullet A\beta$，（A为非终结符），则在状态$i$和$j$之间拉一条空弧。

根据以上两条规则可以构造出NFA，进一步用子集法构造出DFA（但是一般不用这种方法）

Closure——GO——DFA

以文法$G(S')$

$$
\begin{aligned}
S' &\to E \\
E &\to aA|bB \\
A &\to cA|d \\
B &\to cB|d
\end{aligned}
$$

为例

拓广文法

首先引入一个非终结符$S' \to E$，E是原文法G的开始符号，S'是**拓广文法**G'的开始符号，其目的是：**保证文法的接受态唯一。**因为原文法中可能含有多个形如$E \to X$的产生式（在例子中是$E \to aA|bB$）因此可能产生多个接受项目（根据定义），因此添加新的开始符号来将其约束成一个，相当于在原来文法的基础上进行了“拓广”.

有效项目

定义：如果存在一个规范推导$S \dot{\to} \alpha A\omega \dot{\to} \alpha\beta_1\beta_2\omega$ 则称项目$A \to \beta_1 \bullet \beta_2$对活前缀$\alpha\beta_1$是有效的。一般而言，同一个项目可能对多个活前缀有效。

活前缀的有效项目集，就是当栈内的符号是这个活前缀的时候，可能运用的下一步的最左规约的所有规则集合。

一个项目集（状态）中的所有项目，都应该是对某个活前缀有效的（类似于等价的概念）。

构造闭包（*重要!*）

就是对一个活前缀$\delta\alpha$构造出其所有的有效项目的过程。

从$S' \to \bullet E$开始：

根据有效项目的性质：

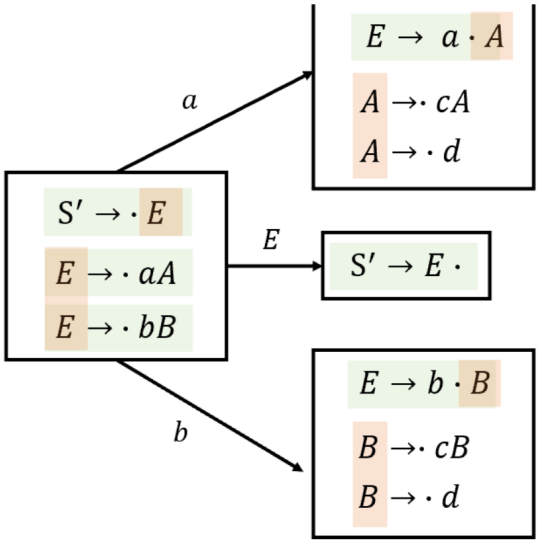
> *若项目$A \to \alpha \bullet B\beta$ 对活前缀$\eta = \delta\alpha$是有效的，且存在产生式$B \to \gamma$，则项目$B \to \bullet\gamma$ 对该活前缀也有效。*

对于某个状态的第一个项目$A \to \alpha \bullet B\beta$，检查圆点后边有无**非终结符**，若有（例如$E \to a \bullet A$），则在产生式集合里面寻找该非终结符对应的产生式$B \to \gamma$(例如$A \to cA$)，然后将其紧贴着箭头右边加上圆点$B \to \bullet\gamma$（例如$A \to \bullet cA$），将该项目加入同一闭包内。重复执行直到闭包中元素不再增加为止。

构造GO(状态转移)

$GO(I, X) = CLOSURE(J)$表示，从状态I经过符号X（可能是终结符也可能是非终结符）到状态J。例如，$E \to \bullet aA$到$E \to a \bullet A$;

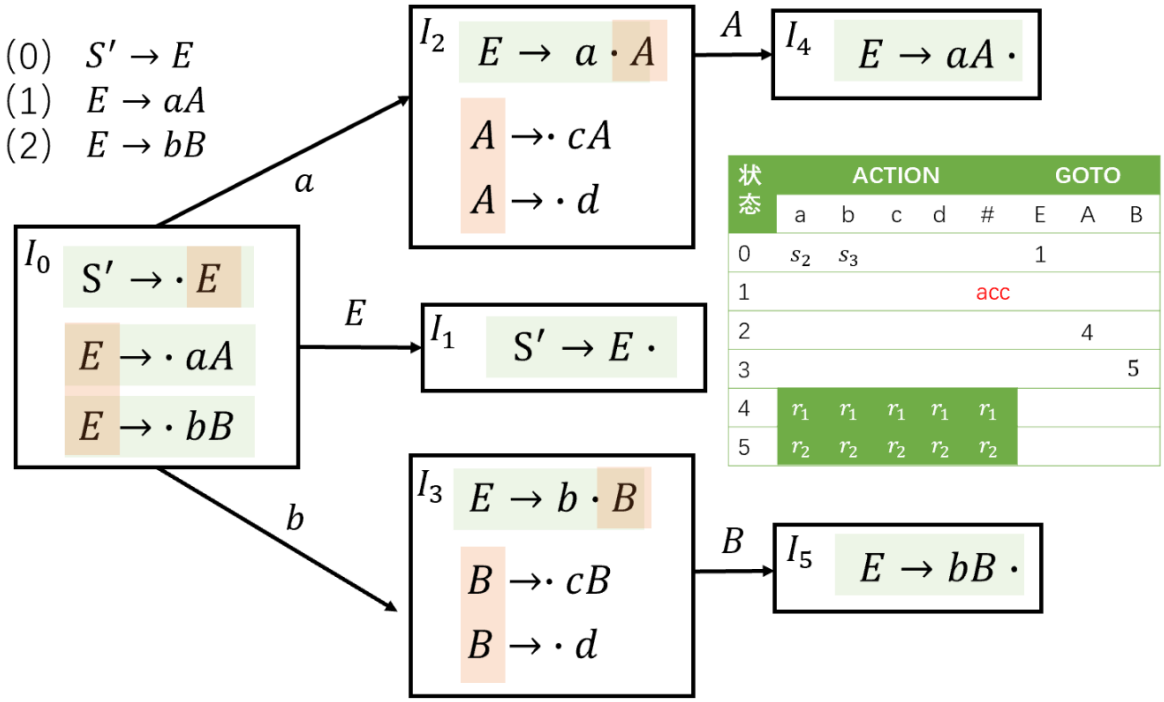
那么从$S' \to \bullet E$生成的闭包开始，对于状态I中的**每一个项目**，都向后跳一个符号（终结符或者非终结符），如果不存在就创建一个新的状态，否则将状态I连接到已经存在的状态上。



构造ACTION表和GOTO表

根据之前得到的DFA，我们得到了所有状态转移的过程，那么对于不同的弧X（可能为终结符也可能为非终结符）和状态（假设是从i到j），我们可以分为以下几种：

1. 当X为**终结符**时，则在$ACTION$表中添加项`ACTION[i,X] = sj`
2. 当X为**非终结符**时，则在$GOTO$表中添加项`GOTO[i,X] = J`
3. 当i**包含规约项目**时，则在$ACTION$表中添加项`ACTION[i,X] = rk`，其中k是产生式集合中的第k个式子，即该规约项目使用的产生式。**注意，状态`i`对应的一行都写上`rk`。**（这样就增加了发生冲突的可能性，这是后面SLR要解决的问题）
4. 特别地，若i中项目是$S' \to E\bullet$（接受项目）时，在$ACTION$表中添加项`ACTION[i,#] = acc`



状态	ACTION					GOTO		
	a	b	c	d	#	E	A	B
0	s_2	s_3				1		
1					acc			
2							4	
3								5
4	r_1	r_1	r_1	r_1	r_1			
5	r_2	r_2	r_2	r_2	r_2			

SLR(Simple LR)分析表的构建

LR(0)文法

若按照上述方法构造的DFA中的每个状态都**不**存在下述两种情况之一：

（1） 既含有**移进项目**又含有**规约项目**——移进-规约冲突

（2） 含有多个**规约项目**——规约-规约冲突

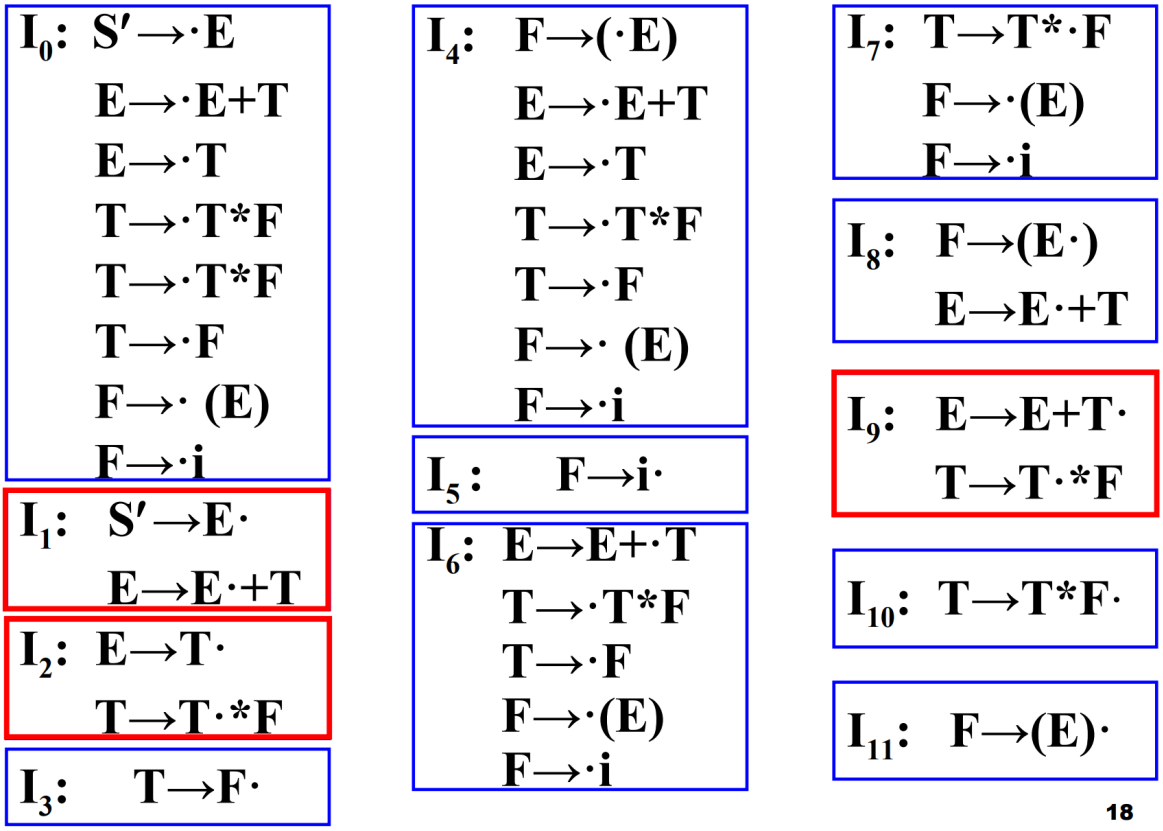
则称G是一个LR(0)文法

若存在上述两种情况?

示例：

$$
\begin{aligned}
&(0) S' \to E \\
&(1) E \to E + T \\
&(2) E \to T \\
&(3) T \to T * F \\
&(4) T \to F \\
&(5) F \to (E) \\
&(6) F \to i
\end{aligned}
$$

这个文法的LR(0)规范集族为：



可见，状态I_1 中，同时出现了规约项目$S' \to E\bullet$和移进项目$E \to E \bullet +T$，此时，LR(0)分析器不知道下一步应该移进`+`还是把栈中的`E`规约成`S'`.我们把这种情况称为**移进-规约冲突**；类似的，还有**规约-规约冲突**。存在这两种冲突的文法就不是LR(0)文法。

SLR(1)文法

为了解决**规约-规约冲突**，我们需要向前看一个字符来确定能不能规约，使用哪个式子规约，引入FOLLOW集合来构建一个SLR(1)文法。

> SLR(1)中，S是Simple，1表示向前展望一个字符，加起来意思就是：简单地向前展望一个字符的LR文法。

我们假设有下面这样一个状态：

$$
\begin{aligned}
&(0)X \to \alpha \bullet b\beta \\
&(1)A \to \alpha \bullet \\
&(2)B \to \alpha \bullet
\end{aligned}
$$

麻雀虽小五脏俱全，这个状态既含有**移进-规约冲突**又含有**规约-规约冲突**。

考虑到只有在**后续字符（还未入栈的字符）正好在`A`的FOLLOW集中**的情况下使用`(1)`进行规约才可能规约成功；否则，此时如果使用`B`规约，则后续分析过程将无法进行，因为`B`的**FOLLOW集中没有该字符**。

因此，我们可以根据示例给出SLR(1)文法的**定义**：

计算FOLLOW(A)与FOLLOW(B)的集合元素，如果满足：

$$
\begin{aligned}
&FOLLOW(A) \wedge FOLLOW(B) = \phi \\
&b \notin FOLLOW(B) \\
&b \notin FOLLOW(A)
\end{aligned}
$$

则该文法为SLR(1)文法。（后两个式子表示文法中不存在**移进-规约冲突**，但其实SLR的方法并不能解决这个问题）

对于构建ACTION和GOTO表的影响，则体现在：

...查看LR(0)构建方法

3. 若i**包含规约项目**时，则在$ACTION$表中添加项`ACTION[i,X] = rk`，其中k是产生式集合中的第k个式子，即该规约项目使用的产生式。**注意，~~状态`i`对应的一行都写上`rk`。~~ 产生式k的左侧非终结符的FOLLOW集合**中的元素对应的位置才标记`rk`.

...

当然，因为是"简单"地展望，只是比较粗略地使用了FOLLOW集合的内容作为展望信息，可以解决规约-规约冲突，但是**不能保证解决移进-规约冲突。**且FOLLOW集得到的超前符号集，有**可能大于实际可能出现的符号集。**

这就需要引入更强的文法。

LR(1)分析表的构建

SLR文法的限制

在SLR分析的最后，介绍了SLR依然可能存在语法冲突。为什么呢?

原因：SLR只是简单地考察下一个输入符号b是否属于与归约项目$A \to \alpha$相关联的$FOLLOW(A)$，但$b \in FOLLOW(A)$只是归约α的一个必要条件，而非充分条件　　海轰的理解：如果输入下一个字符是b，我们采用了归约操作，那么就一定可以说明b属于A的$FOLLOW$集。但是我们不能说：如果b属于A的$FOLLOW$集，那么就一定可以对A采用归约操作。

LR(1)文法的引入

在 LR(0) 分析的时候，我们引入了 LR(0) 项目。其实简单说来就是通过一个小圆点标记字符状态（圆点前字符表示已经读入，圆点后表示未读入）。但是在 LR(0) 分析的时候，我们并没有考虑下一个输入的字符。而**对于 LR(1)，我们就需要对每一个项目多考虑一项：下一个输入字符（也就是：展望符）。**

定义：将一般形式为 $[A \to \alpha \cdot \beta, \gamma_1\gamma_2 \dots \gamma_k]$ 的项称为 LR(1) 项，其中 $A \to \alpha \cdot \beta$ 是一个产生式，γ_i 是一个终结符(这里将 $ 视为一个特殊的终结符)它表示在当前状态下，**A 后面必须紧跟的终结符**，项的展望符（lookahead）。$\gamma_1\gamma_2 \dots \gamma_k$ 被称为向前搜索符串（或展望串）。γ_1 被称为向前搜索符符（或展望符）。

- LR(1) 中的 1 指的是项的第二个分量的长度。即：往后多看一个字符。
- 在形如 $[A \to \alpha \cdot \beta, \gamma]$ 且 $\varepsilon \notin FIRST(\beta)$ 的项中，展望符 γ 没有任何作用。
- 但是一个形如 $[A \to \alpha \cdot, \gamma]$ 的项在只有在下一个输入符号等于 γ 时才可以按照 $A \to \alpha$ 进行归约：这样的 γ 的集合总是 $FOLLOW(A)$ 的子集，而且它通常是一个真子集。

构造LR(1)文法的方法

项目集I的闭包CLOSURE(I)

1. I 的任何项目都属于 $CLOSURE(I)$。
2. 若项目 $[A \to \alpha \cdot \beta, \gamma]$ 属于 $CLOSURE(I)$：
    - 如果 $[B \to \cdot\gamma, \beta]$ 原来不在 $CLOSURE(I)$ 中，则把它加进去。
    - $B \to \gamma$,是一个产生式, $\beta \in FIRST(\beta\gamma)$。
3. 重复执行步骤（2），直到 $CLOSURE(I)$ 不再增大为止。

GO函数

令 I 是一个项目集，X 是一个文法符号，$GO(I,X)=CLOSURE(J)$，其中 $J=\{[A\to\alpha\cdot\beta,\gamma]|[A\to\alpha\cdot\beta,\gamma]\in I\}$

构造LR(1)分析表

1. 若项目 $[A\to\alpha\cdot a\beta,b]\in I_k$ 且 $GO(I_k,a)=I_j$
2. **若项目** $[A\to\alpha\cdot,a]\in I_k$ **，** j**是产生式** $A\to\alpha$ **的编号，置** $ACTION[k,a]$ **为** r_j
3. 若项目 $[S'\to S\cdot,\#]\in I_k$，置 $ACTION[k,\#]$为 acc。
4. 若 $GO(I_k,A)=I_j$，置 $GOTO[k,A]=j$。
5. 分析表中凡不能用规则(1)~(4)填入的空白格均置为“出错标志”。

构建LR(1)文法分析表的过程

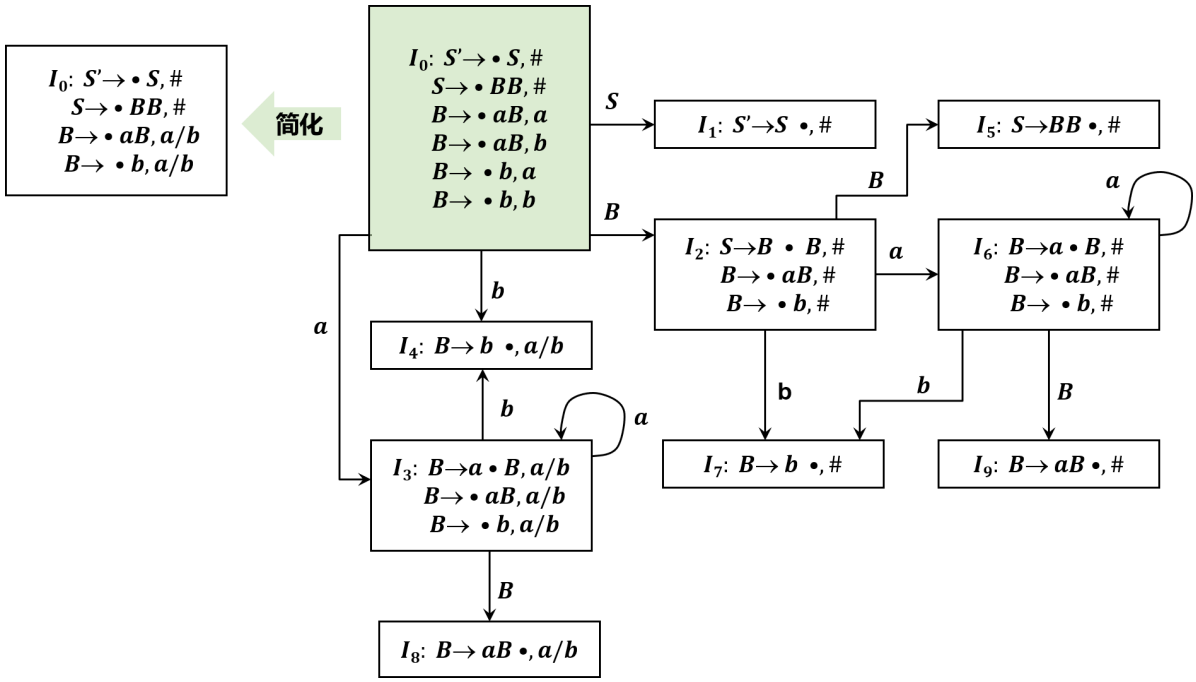
对于文法：

$$(0)S'\to S$$
$$(1)S\to BB$$
$$(2)B\to aB$$
$$(3)B\to b$$

写出FOLLOW集：

$$FOLLOW(S)$$

这个文法的 LR(1) 规范集族为：



构造ACTION表和GOTO表：

	ACTION			GOTO	
状态	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		

LR分析器的工作过程(即，三元式的变化过程)如下：

按上表对aabab进行分析

步骤	状态	符号	输入串
0	0	#	aabab#
1	03	#a	abab#
2	033	#aa	bab#
3	0334	#aab	ab#
4	0338	#aaB	ab#
5	038	#aB	ab#
6	02	#B	ab#
7	026	#Ba	b#
8	0267	#Bab	#
9	0269	#BaB	#
10	025	#BB	#
11	01	#S	#

LALR分析表的构建

LALR文法的引入

对 LR(1) 文法来说，存在着某些状态，这些状态展望串不同外，其核心部分都是相同的。

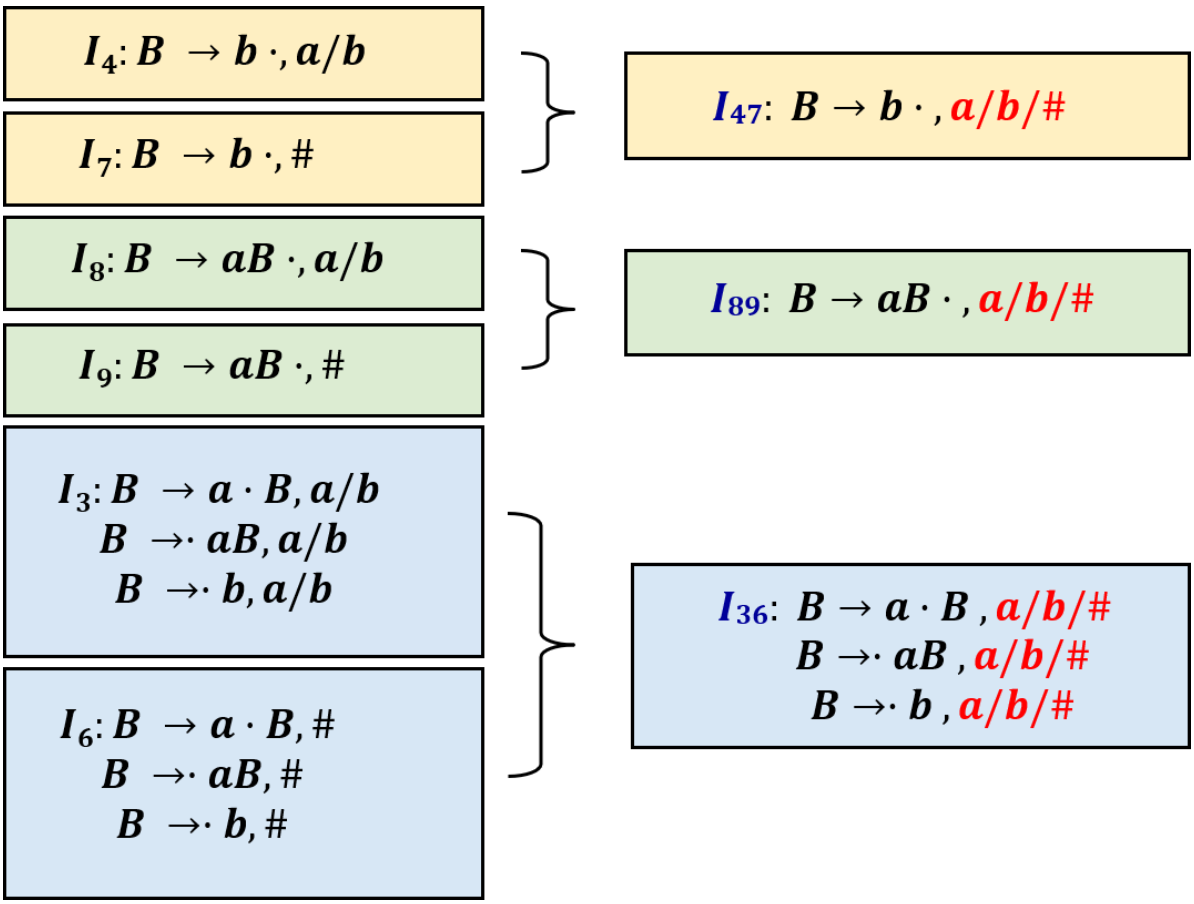
同心项：两个LR(1)项目集具有相同的心，即除去展望串之后，这两个集合是相同的。

LR(1) 到 LALR 文法的转化

- **合并同心项**，寻找具有相同核心的 LR (1) 项集，并将这些项集合并为一个项集。 所谓项集的核心就是其第一分量的集合。
- 然后根据合并后得到的项集族构造语法分析表。

构建LALR文法分析表的过程

与LR(1)文法同例，合并同心集：



查看是否会发生冲突

	ACTION			GOTO	
状态	a	b	#	S	B
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3			
5			r1		
89	r2	r2			

整理得：

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

LR分析小结

1. LR分析过程是**规范归约过程**
2. 符号栈中的符号串是**活前缀**
3. 分析决策依据**栈顶状态和现行输入符号**
4. 为构造LR分析表，可先构造**识别活前缀**的DFA
5. LR分析器的关键是**分析表的构造**
6. LR(0)、SLR(1)、 LALR(1)、LR(1) 功能逐个增强。四种LR类型的文法是真包含关系
7. LR类型文法是**无二义**的

笔记贡献者：

6.1 张倩

6.2 来苑

6.3 来苑

6.4 张倩

6.1 属性文法

本节概念

文法符号：终结符或非终结符。

属性文法：它是在上下文无关文法的基础上，为每个文法符号配备若干个相关的“值”（称为属性）。

属性：代表与文法符号相关信息，例如它的类型、值、代码序列、符号表内容等等。属性与变量一样，可以进行计算和传递。

语义规则：对于文法的每个产生式都配备了一组属性的计算规则，称为语义规则。属性加工的过程即是语义处理的过程。

属性分为两类：**综合属性和继承属性**。简而言之，综合属性用于“自上而下”传递信息，而继承属性用于“自下而上”传递信息。

在一个属性文法中，对应于每个产生式A→α都有一套与之相关联的语义规则，每条规则的形式为b:=f(c_1,c_2,…,c_k)，f为函数，且若满足下面条件之一：

（1）b是A的一个综合属性，且c_1,c_2,…,c_k是产生式右边文法符号的属性

（2）b是产生式右边某个文法符号的一个继承属性，且c_1,c_2,…,c_k是产生式中任何文法符号的属性

都说**属性b依赖于c_1,c_2,…,c_k**。

说明：
（1）终结符只有综合属性，它们由词法分析器提供（如终结符的type属性，标识符在变量表中的地址等）。

（2）非终结符即可有综合属性也可有继承属性，文法开始符号的所有继承属性作为属性计算前的初始值。

（3）对出现在产生式右边的继承属性和出现在产生式左边的综合属性都必须提供一个计算规则。属性计算规则中只能使用相应产生式中的文法符号的属性。

（4）出现在产生式左边的继承属性和出现在产生式右边的综合属性不由所给的产生式的属性计算规则进行计算，它们由其他产生式的属性规则计算或者由属性计算器的参数提供。

即对于一个产生式，我们需要关心的是：产生式左边的综合属性和产生式右边的继承属性的计算规则。

举例说明：

考虑非终结符A，B和C，其中，A有一个继承属性a一个综合属性b，B有综合属性c，C有继承属性d。产生式A→BC可能有规则：

C.d=B.c+1

A.b=A.a+B.c

此处B.c为产生式右边的综合属性，A.a为产生式左边的继承属性，在其他地方计算；

而C.d为产生式右边的继承属性，A.b为产生式左边的综合属性，应用该产生式时，我们需要考虑的是C.d和A.b的计算。

（注：为了区分一个产生式中同一个非终结符多次出现，我们对某些非终结符加了下标，以便消除对这些非终结符的属性值引用的二义性。）

示例：考虑表6.1所示的一个属性文法，它用作台式计算器程序。

对每个非终结符E、T及F都有一个综合属性----称为val的属性值。

在每个产生式对应的语义规则中，产生式左边的非终结符的属性值val是从右边的非终结符的属性值val计算出来的。

表6.1　一个简单台式计算器的属性文法

产生式	语义规则	说明
L→En	print(E.val)	打印由E产生的算术表达式的值，可以认为这条规则定义了L的一个虚属性。
$E \rightarrow E_1+T$	$E.val:=E_1.val+T.val$	
E→T	E.val=T.val	
$T \rightarrow T_1*F$	$T.val=T_1.val*F.val$	
T→F	T.val:=F.val	
F→(E)	F.val:=E.val	
F→digit	F.val=digit.lexval	符号digit有一个综合属性lexval，它的值由词法分析器提供。

综合属性

在语法树中，一个结点的综合属性的值由其子结点的属性值确定。因此，通常使用自底向上的方法在每一个结点处使用语义规则计算综合属性的值。

仅仅使用综合属性的属性文法称为S-属性文法。

例6.1　表6.1定义的属性文法说明了一个台式计算器,该计算器读人一个可含数字、括号和 + 、运算符的算术表达式,并打印表达式的值,每个输入行以n作为结束。例如,假设表达式为3*5+4,后跟一-个换行符z。则程序打印数值19。图6.1给出了输人串3*5+4n的带注释的语法树。在语法树的根打印结果,其值为根的第一个子结点E.val的值。

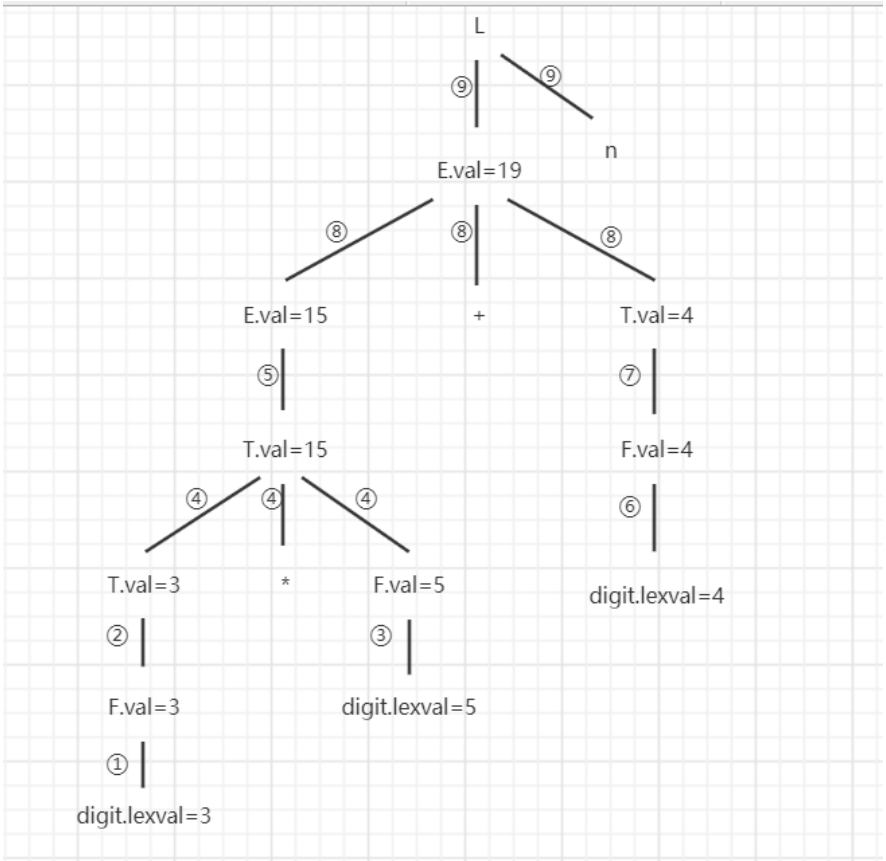


图6.1　3*5+4n的带注释的语法树

（图中序号表示规约的顺序，也即计算属性值的顺序。）

过程说明：

①第一步运用产生式F→digit对F的综合属性进行赋值，F.val=digit.lexval=3

②第二步运用产生式T→F对T的综合属性进行赋值，T.val:=F.val=3

③第三步运用产生式F→digit对F的综合属性进行赋值，F.val=digit.lexval=5

④第四步运用产生式T→T_1*F对T的综合属性进行计算，T.val=T_1.val*F.val=3*5=15

⑤第五步运用产生式E→T对E的综合属性进行赋值，E.val=T.val=15

⑥第六步运用产生式F→digit对F的综合属性进行赋值，F.val=digit.lexval=4

⑦第七步运用产生式T→F对T的综合属性进行赋值，T.val:=F.val=4

⑧第八步运用产生式E→E_1+T对E的综合属性进行计算，E.val=E_1.val+T.val=15+4=19

⑨第八步运用产生式L→En，语义规则为print(E.val)，即打印出通过E得到的表达式的值。

继承属性

在语法树中，一个结点的继承属性由此结点的父节点 和/或 兄弟结点的某些属性确定。

在下面的例子中，继承属性在说明中为各种标识符提供类型信息。

例6.2　在表6.2中给出的属性文法中：

不规范的总结：

产生式D→TL对应变量说明。（如 int a,b,c ）

由非终结符D所产生的变量说明含关键字int和real,后跟一个标识符表。

T→类型说明（如int，real）

L→标识符数组（如a,b,c）

过程：

①非终结符T有一个综合属性type,它的值由说明中的关键字确定，非终结符L有一个继承属性type，它的值由父亲或者兄弟的属性来确定。

②与产生式D→TL相应的语义规则L.type: = T. type把说明中的类型赋值给继承属性L.type.

③然后,利用语义规则把继承属性L.type沿着语法树往下传。与L的产生式相应的语义规则调用过程addtype把每个标识符的类型填人符号表的相应项中(符号在符号表由属性entry指明)。

表6.2 带继承属性L.in的属性文法

产生式	语义规则	说明
D→TL	L.type:=T.type	变量说明中的类型赋值给继承属性L.type.
T→int	T.type:=integer	变量说明中的类型关键字为int
T→real	T.type:=real	变量说明中的类型关键字为real
$L \rightarrow L_1$,id	L_1.type:=L.type	把继承属性L.type沿着语法树往下传，儿子的属性由父亲的属性确定。
	addtype(id.entry,L.type)	把每个标识符的类型填人符号表的相应项中，标识符的类型为父亲L的类型。
L→id	addtype(id.entry,L.type)	把每个标识符的类型填人符号表的相应项中，标识符的类型为父亲L的类型。

图6.2给出了句子real id_1 , id_2,id_3 的带注释的语法树。在三个L结点中L.type的值分别给出了标识符real id_1 , id_2,id_3的类型。

为了确定这三个属性值,先求出根的左子结点的属性值T. type,然后每项向下计算根的右子树的三个L结点的属性值L.in。在每个L结点还要调用过程addtype,往符号表中插入信息,说明本结点的右子结点上的标识符类型为real。

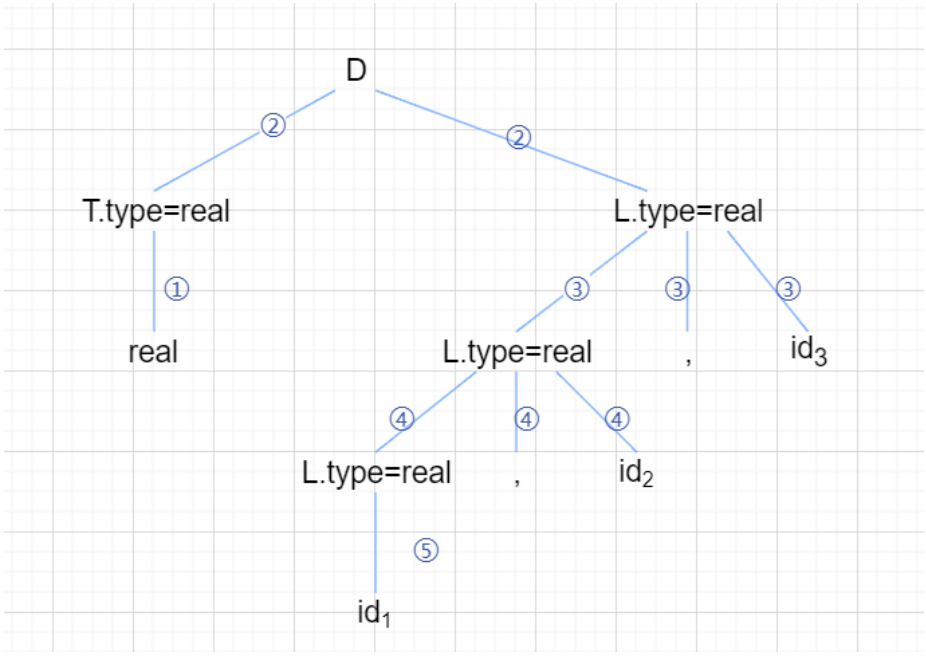


图6.2 在每个L结点都带有继承属性的语法树

说明：

①第一步运用产生式T→int给T的综合属性赋值，T.type:=real

②第二步运用产生式D→TL给L的继承属性赋值，L.type=real

③第三步运用产生式L→L_1,id 给L的继承属性赋值，L_1.type=real，同时，addtype(id_3.entry,real)

④第四步运用产生式L→L_1,id 给L的继承属性赋值，L_1.type=real，同时，addtype(id_2.entry,real)

⑤第五步运用产生式L→id ，addtype(id_1.entry,real)

6.2 基于属性文法的处理方法

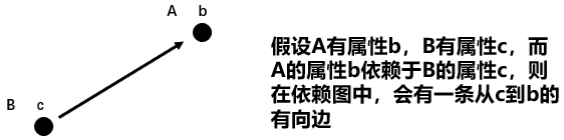
语法制导翻译法：对单词符号串进行语法分析，构造语法分析树，然后根据需要遍历语法树并在语法树的各结点处按语义规则进行计算，这种由源程序的语法结构所驱动的处理方法就是语法制导翻译法。

输入串 → 语法树 → 依赖图 → 语义规则计算次序

1.依赖图

一个有向图。依赖图为每个属性设置一个结点。

若属性b依赖于属性c，则从属性c的结点有一条有向边连接到属性b的结点。



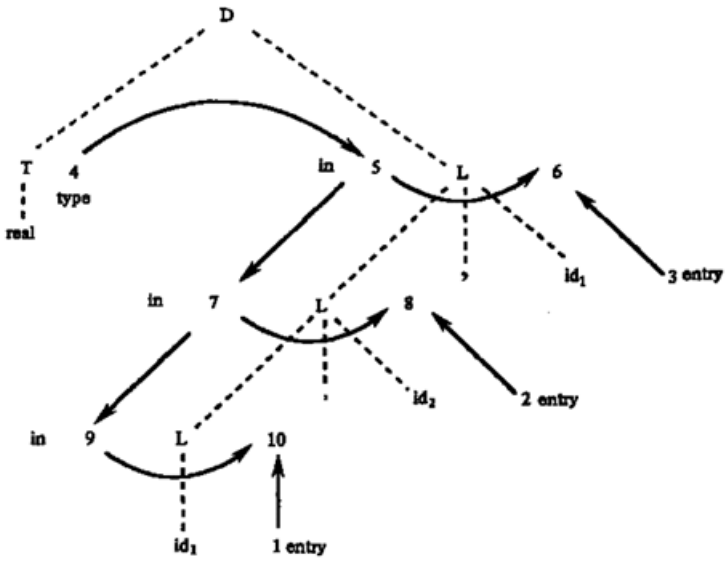
依赖图的构造步骤

```
for  语法树中每一结点 n  do
    for  结点 n 的文法符号的每一个属性 a          do
        为 a 在依赖图中建立一个结点;
for 语法树中每一个结点 n  do
    for  结点 n 所用产生式对应的每一个语义规则
        b := f(c1,c2,…,ck) do
        for  i := 1 to k do
            从 ci 结点到 b 结点构造一条有向边;
```

例：给出以下属性文法：

产生式	语义规则
D→TL	L.in:=T.type
T→int	T.type:=integer
T→real	T.type:=real
$L→L_1$,id	L_1.in:=L.in Addtype(id.entry,L.in)
L→id	Addtype(id.entry,L.in)

其依赖图为：



其中，虚线是语法树，它不是依赖图中的一部分。结点6、8、10是addtype(id.entry,L.in)产生的虚属性。

良定义：属性之间不存在循环依赖关系的属性文法是良定义的。即没有环。

属性的计算次序：一个良定义的属性文法的依赖图是一个有向无环图，所以根据拓扑排序计算属性。

2.树遍历的属性计算方法

假设语法树已经建立，并且树中带有开始符号的继承属性和终结符的综合属性。然后遍历语法树，直到计算出所有属性。

对无循环的属性文法的计算：

```
While 还有未被计算的属性 do
      VisitNode(S)          /* S是开始符号 */
procedure VisitNode (N:Node);
begin
      if  N是一个非终结符 then
          /* 假设它的产生式为 N→X1…Xm */
          for i := 1 to m do
             if not Xi∈VN then  /* 即 Xi 是非终结符 */
                    begin
                      计算 Xi 的所有能够计算的继承属性;
                      VisitNode (Xi)
                    end;
          计算 N 的所有能够计算的综合属性
end
```

只要文法属性是非循环定义的，每次扫描至少会计算出一个属性值。

3.一遍扫描的处理方法

树遍历的属性计算方法是在生成语法树后计算属性，而一边扫描的处理方法是在生成语法树的同时计算属性，且无需构造实际的语法树。另外，当一个属性值不再用于计算别的属性值时，这个属性值可以被舍弃，不保留。

一遍扫描的处理方法与①采用的语法分析方法②属性的计算次序密切相关。

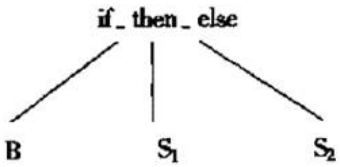
L-属性文法：用于一遍扫描的自上而下分析。产生式成功匹配输入串时，按相应的语义规则计算属性。

S-属性文法：用于一遍扫描的自下而上分析。向上规约时，按相应的语义规则计算属性。

4.抽象语法树

抽象语法树：去掉语法树中对翻译不必要的信息，从而获得的更有效的源程序中间表示。在抽象语法树中，操作符和关键字不作为叶节点出现，而是作为内部结点，即叶节点的父节点。

例如产生式S→if B then S1 else S2在抽象语法树中的表示为：



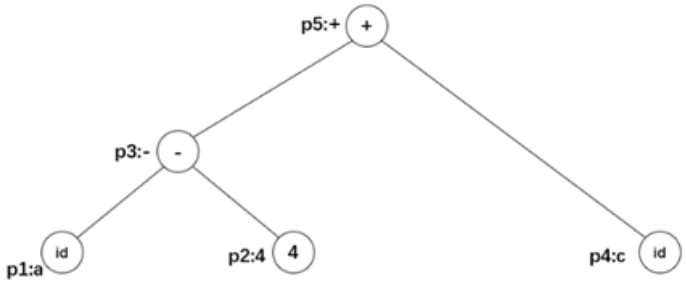
抽象语法树的建立

为每个运算分量或运算符号建立一个结点为子表达式建立子树。运算符号结点的各子节点分别是表示该运算符号的各个运算分量的子表达式组成的子树的根。

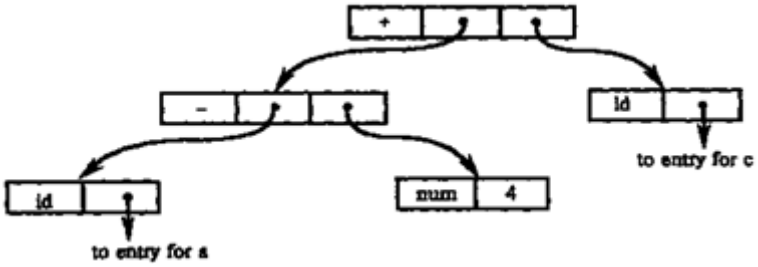
抽象语法树的每个结点可以有几个域。例如，在一个运算符号对应的节点中，一个域标识运算符号，其他域包含指向运算分量的结点的指针。运算符号通常叫做这个结点的标号。

下列是建立二目运算符表达式的结点的函数,函数返回一个指向新建立结点的指针。

> 1)mknode(op,left,right)：建立运算符号结点，标号是op，left和right分别指向两个运算表达式。
>
> 2)mkleaf(id,entry)：建立标识符结点，标号为id，entry指向标识符在符号表中的位置。
>
> 3)mkleaf(num,ral)：建立数结点，标号为num，entry存放数的值。



例：表达式a-4+c的抽象语法树。



- p1:=mkleaf(id,entrya)
- p2:=mkleaf(num,4)
- p3:=mknode('-',p1,p2)
- p4:=mkleaf(id,entryc)
- p5:=mknode('+',p3,p4)

抽象语法树的语义规则

根据前面所讲的函数，修改下面属性文法的语义规则：

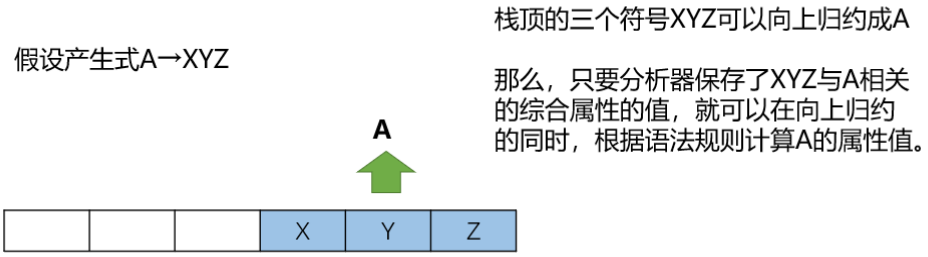
产生式	语义规则
D→TL	L.in:=T.type
T→int	T.type:=integer
T→real	T.type:=real
$L \rightarrow L_1$,id	L_1.in:=L.in Addtype(id.entry,L.in)
L→id	Addtype(id.entry,L.in)

6.3 S-属性文法的自下而上计算

1.S-属性文法

S-属性文法只含有综合属性。

综合属性可以在分析输入符号串的同时由自下而上的分析器来计算。分析器中保存与栈中文法符号有关的综合属性值，在向上归约时就可以计算新的属性值。S-属性文法的翻译器通常借助LR分析器实现。



从上图可以发现，对于一个右边带有r个符号的产生式的向上归约时，相关的就是栈顶的r个字符的属性值。

2.栈中的综合属性

自下而上的分析方法中，在分析栈中使用一个附加的域来存放综合属性值。假设栈由一对数组state和val组成，其中state隐含了文法符号，val存储了对应的属性值。

state	val
…	…
X	X.x
Y	Y.y
Z	Z.z
…	…

那么，对于下列的产生式，就会相应的有右边的代码段来实现功能。

产生式	代码段
L→En	print(val[top])
E→E+T	val[ntop]:=val[top-2]+val[top]
E→T	
T→T*F	val[ntop]:=val[top-2]*val[top]
T→F	
F→(E)	val[ntop]:=val[top-1]
F→digit	

其中，ntop=top-r+1为新栈顶。

只要在做相应的归约前执行对应的代码段，就可以计算新的属性值。

例：分析器在输入3*5+4n上的移动序列。

输入	state	val	用到的产生式
3*5+4n	-	-	
*5+4n	3	3	
*5+4n	F	3	F→digit
*5+4n	T	3	T→F
5+4n	T*	3-	
+4n	T*5	3-5	
+4n	T*F	3-5	F→digit
+4n	T	15	T→T*F
+4n	E	15	E→T
4n	E+	15-	
n	E+4	15-4	
n	E+F	15-4	F→digit
n	E+T	15-4	T→F
n	E	19	E→E+T
	En	19-	
	L	19	L→En

讨论遇到符号3时的移动序列：

①分析器把符号3（digit）的state（3）和val（3）移入到栈中

②根据产生式F→digit归约，由于没有对应的代码段，这里不更新val的值。

③根据产生式T→F归约，由于没有对应的代码段，这里不更新val的值。

……

④在栈中为T*F时，根据产生式T→*TF归约，而由于有对应的代码段val[ntop]:=val[top-2]×val[top]，执行代码段val[1]=val[1]×val[3]=3*5=15。

6.4 L-属性文法和自顶向下翻译

L-属性文法

一个属性文法成为L-属性文法，如果对于每个产生式$A \rightarrow X_1X_2...X_n$，其每个语义规则中的每个属性**或者是综合属性，或者是$X_j(1 \le j \le n)$的一个继承属性**且这个继承属性仅依赖于：

（1）产生式X_j的左边符号X_1，X_2，…，X_{j-1}的属性；

（2）A的继承属性。

由上述定义可见，S-属性文法一定是L-属性文法。因为S-属性文法只含综合属性，而（1）（2）限制只用于继承属性。

从6.2节中我们知道，可以通过深度优先的方法对语法树进行遍历，从而计算属性文法的所有属性值。

L-属性文法允许我们通过一次遍历就计算出所有属性值，诸如LL(1)这种自上而下分析方法的分析过程，从概念上说可以看成是深度优先建立语法树的过程，因此，我们可以在自上而下语法分析的同时实现L属性文法的计算。

表6-7 非L-属性文法例子

产生式	语义规则	说明
A→LM	L.i := l(A.i)	语义规则中的l,m,r,q,f可以视为函数。
	M.i := m(L.s)	
A→QR	R.i := r(A.i)	
	Q.i := q(R.s)	因为文法符号Q的继承属性Q.i依赖于**右边**的文法符号R的属性s，所以该属性文法文**不是L-属性文法。**
	A.s := f(Q.s)	

6.4.1 翻译模式

属性文法可以看作是关于语言翻译的高级规范说明,其中隐去实现细节,使用户从明确说明翻译顺序的工作中解脱出来。

下面我们讨论一种适合语法制导翻译的另一种描述形式,称为**翻译模式**(Translation schemes)。翻译模式给出了使用语义规则进行计算的次序,这样就可把某些实现细节表示出来。

在翻译模式中,和文法符号相关的属性和语义规则(这里我们也称语义动作),用花括号{}括起来,插入到**产生式右部**的合适位置上。这样*翻译模式给出了使用语义规则进行计算的顺序。*

下面是一个简单的翻译模式例子,它把带加号和减号的中缀表达式翻译成相应的后缀表达式。

E→TR

R→addop T {print(addop.lexeme)} R1 | ε

T→num {print(num,val)}

图6.10表示的是关于输入串9-5+2的语法树，**每个语义动作都作为相应产生式左部符号的结点的儿子。**即把**语义动作看作是终结符号**，表示在什么时候应该执行哪些动作。

图中用实际的数字和加减法运算代替了单词num和addop。

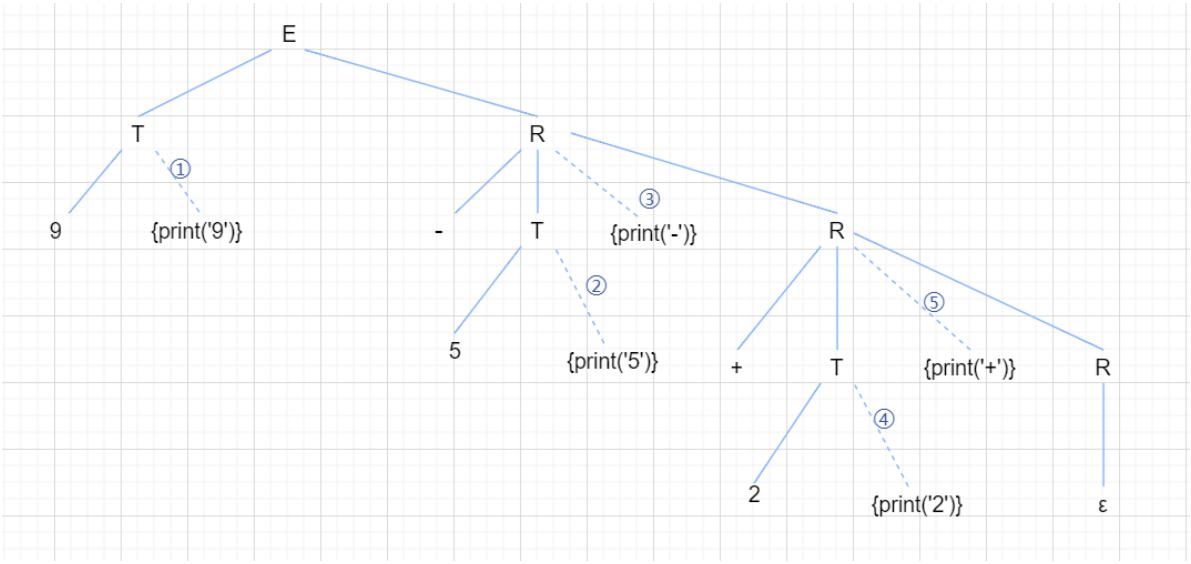


图6.10 9-5+2的说明动作的语法分析树

说明：图中的序号为语义动作执行的顺序。

当按**深度优先次序**执行图6.10中的动作后，打印输出：95-2+

设计翻译模式要注意：

设计翻译模式时,我们必须注意某些限制以保证当某个动作引用一个属性时它必须是有定义的（属性必须有定义性）。L-属性文法本身就能确保每个动作不会引用尚未计算出来的属性。

当只需要综合属性时,情况最为简单。在这种情况下,我们可以这样来建立翻译模式:为每一个语义规则建立一个包含赋值的动作,并把这个动作放在相应的产生式右边的末尾。

例如,假设有下面的产生式和语义规则:

产生式	语义规则
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$

我们建立产生式和语义动作：

产生式	语义动作
$T \rightarrow T_1 * F$	$\{T.val := T_1.val * F.val\}$

如果**既有综合属性又有继承属性**，在建立翻译模式时就必须特别小心：
(1)产生式右边的符号的继承属性必须在这个符号以前的动作中计算出来。

(2)一个动作不能引用这个动作右边的符号的综合属性。

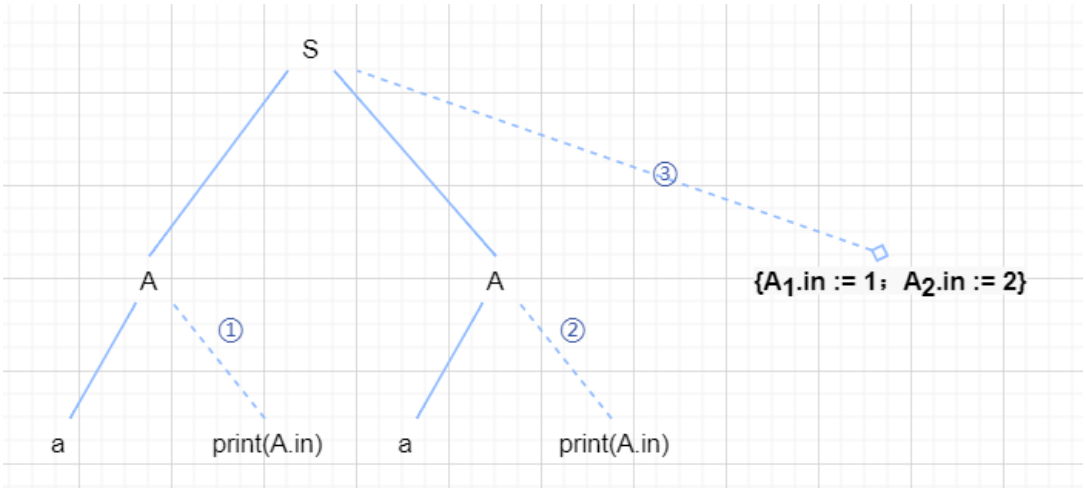
(3)产生式左边非终结符的综合属性只有在它所引用的所有属性都计算出来以后才能计算。计算这种属性的动作通常可放在产生式右端的末尾。

后面我们将看到满足这三个条件的翻译模式是如何在一般的自上而下和自下而上分析器中实现的。

下面的翻译模式不满足上述三个条件中的第一个条件：

$S \rightarrow A_1 A_2$	**{A_1.in := 1；A_2.in := 2}**
A→a	{print(A.in)}

说明：如按深度优先遍历输入串aa的语法树，语法树如下：



说明：图中的序号为语义动作执行的顺序。

可以看到①②对应的语义动作：当要打印第二个产生式里继承属性A.in时的值时，该属性还没有定义。

也就是说，从S开始按深度优先遍历A_1和A_2子树之前A_1.in和A_2.in还未赋值。即该翻译模式不满足上述三个条件中的第一个条件。

改动语义动作使其满足要求：

$S \rightarrow \{A_1.in := 1\}\ A_1\ \{A_2.in := 2\}\ A_2$	
A→a	{print(A.in)}

上述文法满足上述三个条件。（先得到A的继承属性A.in再print)

6.4.2 自顶向下翻译

在第四章我们知道,为了构造不带回溯的自顶向下语法分析,必须消除文法中的左递归。现在我们把前面讨论过的消除左递归的算法加以扩充,当消除一个翻译模式的基本文法的左递归时同时考虑属性。这种方法适合带综合属性的翻译模式。这样,许多属性文法可以使用自顶向下分析来实现。下面举一个例子。

由于大多数算术运算符号都是左递归的,因此,我们很自然地用左递归文法来产生算术表达式。关于算术表达式的左递归文法相应的翻译模式形式如图6.13所示。

$E \rightarrow E_1 + T \quad \{E.val := E_1.val + T.val\}$

$E \rightarrow E_1 - T \ \{E.val := E_1.val - T.val\}$

E→T　{E.val := T.val}

T→(E) {T.val := E.val}

T→num {T.val := num.val}

图6.13 带左递归的文法的翻译模式

对图6.13消除左递归，构造新的翻译模式R如图6.14所示。

E→T　{R.i := T.val}　R {E.val := R.s}

R→+ T　{R_1.i := R.i + T.val}　R_1　{R.s := R_1.s}

R→-T {R_1.i := R.i - T.val} R_1　{R.s := R_1.s}

R→ε　{R.s := R.i}

T→(E) {T.val := E.val}

T→num {T.val := num.val}

图6.14 消除左递归后的翻译模式

在图6.14给出的翻译模式中，每个数都是由T产生的，并且T.val的值就是由属性num.val给出的数的词法值。

新的翻译模式产生的表达式9-5+2的带注释的语法树如图6.15所示。图中实线箭头指明了对表达式计算的顺序，图中序号表示语义动作的执行顺序。

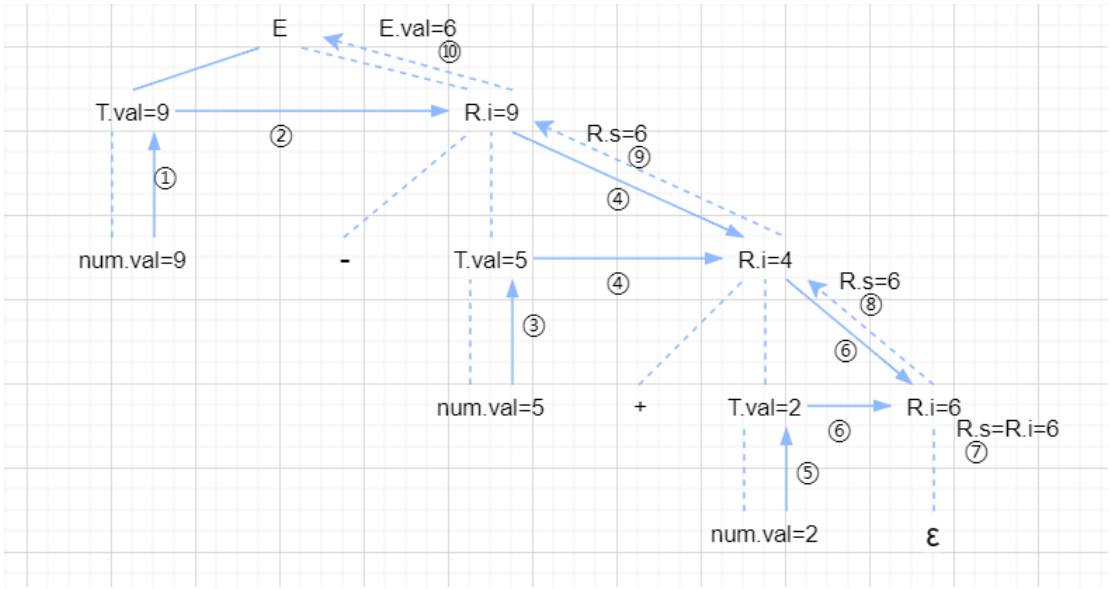


图6.15 计算表达式9-5+2

子表达式9-5中的数字9是由最左边的T生成的,但是减号和5是由根的右子结点R生成的。继承属性R.i从T.val得到值9。计算9-5并把结果4传递到中间的R结点,这是通过产生式中嵌入的下面动作实现:

{R_1.i := R.i - T.val}

类似的动作把2加到9-5的值上,在最下面的R结点处产生结果R.i=6。这个结果将成为根结点处E.val的值(如⑦⑧⑨⑩的语义动作所实现）；

图6.13中的第二个产生式中,第一个动作(对R.i赋值)是在T被完全展开成终结符号后执行的,第二个动作是在R被完全展开成终结符号后执行的。正如前面我们所讨论的，一个符号的继承属性必须由出现在这个符号之前的动作来计算,产生式左边非终结符的综合属性必须在它的所依赖的所有属性都计算出来以后才能计算。

下面我们把转换左递归翻译模式的方法推广到一般,以便进行自顶向下分析。假设我们有下面的翻译模式:

$A \rightarrow A_1 Y \quad \{A.a := g(A_1.a, Y.y)\}$

$A \rightarrow X \quad \{A.a := f(X.x)\}$

它的每个文法符号都有一个综合属性，用小写字母表示，g和f是任意函数。

利用第四章消除左递归的算法，可将其转换成下面的文法：

$A \rightarrow XR$

$R \rightarrow YR \mid \varepsilon$

再考虑语义动作，翻译模式变为：

$A \rightarrow X \quad \{R.i := f(X.x)\} \quad R \quad \{A.a := R.s\}$

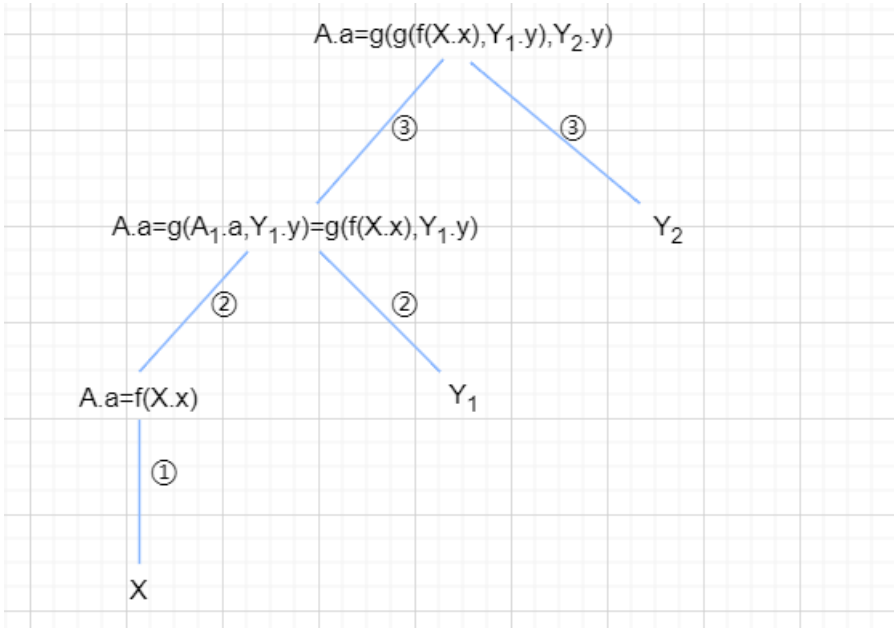
$R \rightarrow Y \quad \{R_1.i := g(R.i, Y.y)\} \quad R_1 \quad \{R.s := R_1.s\}$

$R \rightarrow \varepsilon \quad \{R.s := R.i\}$

经过转换的翻译模式与图6.14中一样使用R的继承属性i和综合属性s。

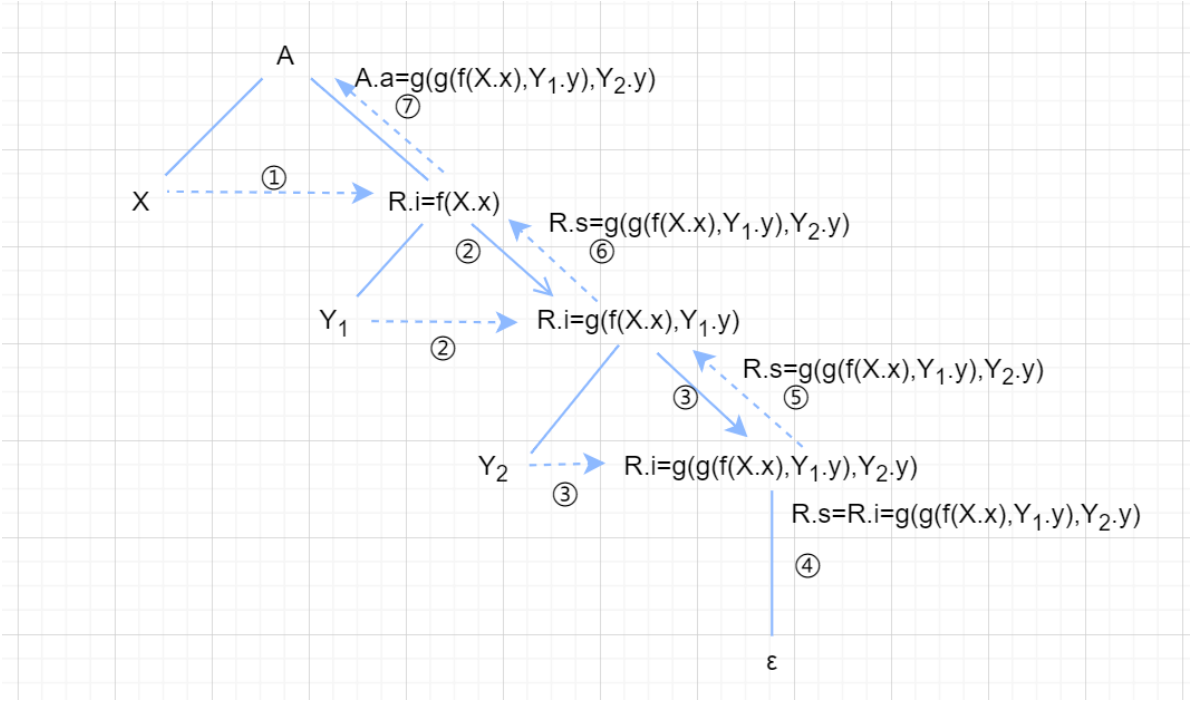
为了说明为什么翻译模式(6.1)和翻译模式(6.3)的结果是一样的,我们考虑图中两棵带注释的语法树。图6.16(a)中A.a的值是根据翻译模式(6.1)自下而上计算的。图6.16(b)中包含了根据翻译模式(6.3)自上而下计算R.i。

图6.16 计算属性值的两种方法



(a)自下而上计算属性值；

说明：图中序号表示语义动作的执行顺序，该文法所涉及的属性均为综合属性。



(b)自上而下计算属性值

说明：图中序号表示语义动作的执行顺序，箭头表示数据流向。

图(b)中最下面的R.i的值不变地传递到上面作为R.s的值,并作为根结点A的A.a值。

如果把构造抽象语法树的属性文法定义(见表6.4)转化成翻译模式，当从该翻译模式中消除左递归时，翻译模式如图6.17所示。有关T的产生式和语义动作与表6.4中原有定义相似。

E→T {R.i := T.nptr} R {E.nptr := R.s}

R→+ T {R_1.i := mknode('+',R.i,T.nptr)} R_1 {R.s := R_1.s}

R→-T {R_1.i := mknode('-',R.i,T.nptr)} R_1 {R.s := R_1.s}

R→ε {R.s := R.i}

T→(E) {T.nptr := E.nptr}

T→id {T.nptr := mkleaf(id,id.entry)}

T→num {T.nptr := mkleaf(num,num.val)}

图6.17 构造抽象语法树的翻译模式

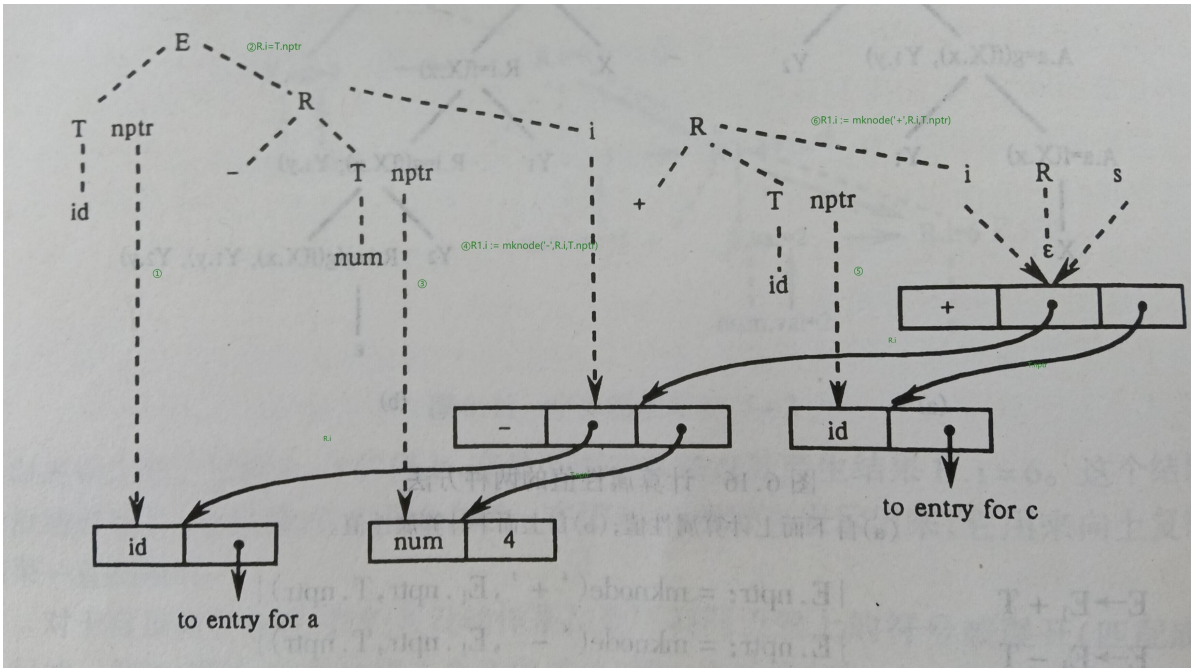


图 6.18　使用继承属性构造语法树

图6.18表示了怎样用图6.17中的动作构造a-4+e的语法树。像例6.8中一样,抽象语法树中的叶结点由产生式T→i和T→num对应的语义动作建立。最左边的T结点的属性T.nptr指向叶结点a。然后E→TR中的R.i := T.nptr。

当产生式R→-TR_1在根的右子结点处应用时,R.i指向结点a,且 T.rptr指向结点4。对于减号和这些指针应用mknode来构造与a-4相应的结点。

最后,当应用产生式R→ε时,R.i向上指向整个语法树的根结点。或说,整个语法树通过代表R的结点的s属性返回(图6.18中没有表示出来),直到R.s的值成为E.nptr的值。

6.4.3 递归下降翻译器的设计

对给定的适合于自顶向下翻译的翻译模式,下面给出设计递归下降翻译器的方法。

1.对每个非终结符A构造一个函数过程，对A的每个继承属性设置一个形式参数，函数的返回值为A的综合属性(作为记录,或指向记录的一个指针,记录中有若干域,每个属性对应一个域)。为了简单，我们假设每个非终结只有一个综合属性。A对应的函数过程中，为出现在A的产生式中的每一个文法符号的每一个属性都设置一个局部变量。

2.非终结符A对应的函数过程中，根据当前的输入符号决定使用哪个产生式候选。

3.每个产生式对应的程序代码中，按照从左到右的次序,对于单词符号(终结符)非终结符和语义动作分别作以下工作。

(1)对于带有综合属性x的终结符X，把x的值存入为X.x设置的变量中。然后产生一个匹配X的调用，并继续读入一个输入符号。

(2)对于每个非终结符B，产生一个右边带有函数调用的赋值语句c = $B(b_1,b_2,...,b_k)$，其中,$b_1,b_2,...,b_k$是为B的继承属性设置的变量，c是为B的综合属性设置的变量。

(3)对于语义动作,把动作的代码抄进分析器中，用代表属性的变量来代替对属性的每一次引用。

例6.12图6.17中的文法是LL(1)的，因此适合于自顶向下分析。根据文法非终结符的属性，得到关于非终结符E,R、T的函数及其参数的类型，具体如下:

function E:↑AST-node;

function R(in:↑AST-node): ↑AST-node;

function T:↑AST-node;

因为E和T没有继承属性，所以没有参数。

我们在新的产生式中用addop代表+和-。

R→oddop T {R_1.i := mknode(addop.lexme, R.i, T.nptr)}

R→ε {R.s := R.i}

关于产生式R的代码以图6.19的分析过程为基础。如果输入符号为addop,则使用产生式R→addop T R,通过advance读人addop之后的下一输入符号,然后再调用T和R的函数。否则,按产生式R→ε,该过程什么也不做。

```
producer  R:
begin
  if sym = addop then
    begin
      advance;T;R
    end
  else begin/*do nothing*/
         end
end;
```

图6.19产生式R→addop TR/ε的分析过程

在图6.19的基础上构造对应R的函数过程，如图6.20所示。函数中包含计算属性的代码。

```
function R(in:↑AST-node): ↑AST-node;
  var  nptr,i1,s1,s:↑AST-node;
  addoplexeme:char;
begin
  if sym = addop then begin
    /*产生式R→addop TR*/
    addoplexeme := lexval;
    advance;
    nptr := T;
    i1 := mknode(addoplexeme,in,nptr);
    s1 := R(i1)
    s := s1
  end
  else s := in;
```

```
    return s;
end;
```

把单词符号addop的词法值 lexval存人addoplexeme中；匹配addop；调用函数T，把结果存人nptr中，变量i_1对应于翻译模式中的继承属性R_1.i，S_1对应于综合属性R_1.s。返回语句在控制离开函数以前返回s的值。

类似地,我们可构造出E和T的函数。

> - 7.1-7.2：赵子涵
> - 7.3：牛庆莹
> - 7.4-7.5：席腾霄

第七章 语义分析和中间代码生成

静态语义检查和翻译紧接在词法分析和语法分析之后。

语义分析：

1. 类型检查
2. 控制流检查
3. 一致性检查
4. 相关名字检查
5. 名字的作用域分析

中间代码：易于翻译成目标程序的源程序的等效内部表示代码，复杂性介于源语言和目标语言之间。

> 源程序可以直接翻译成目标代码，但使用中间代码的好处：
>
> 1. 便于进行与机器无关的代码优化工作
> 2. 使编译程序改变目标机更容易
> 3. 使编译程序的结构在逻辑上更简单明确

7.1 中间语言

常见的形式

- 后缀式（逆波兰表示法）
- 图表示法：
  - 抽象语法树(6.2.4节-P144)
  - DAG图表示
- 三地址代码：
  - 三元式
  - 四元式
  - 间接三元式

7.1.1 后缀式（逆波兰表示法）

- 基本思想：把运算量（操作数）写在前面，算符写在后面。是语法分析树的线性表达（后序遍历）。
- 举例：
  - `a*(b+c)`：`abc+*`
  - `(a+b)*(c+d)`：`ab+cd+*`
- 语义规则描述：

产 生 式	语 义 规 则
$E \rightarrow E_1\ op\ E_2$	$E.code := E_1.code \parallel E_2.code \parallel op$
$E \rightarrow (E_1)$	$E.code := E_1.code$
$E \rightarrow id$	$E.code := id$

7.1.2 图表示法

每个子表达式都有一个结点：一个内部节点代表一个操作符，它的孩子代表操作数。

- 抽象语法树：公共子表达树表示为重复的子树（可参见6.4.2节-P144）
- DAG图表示：公共子表达树有多个父节点

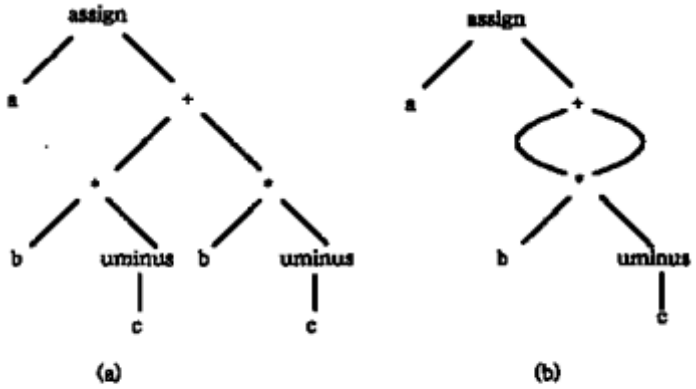


图 7.3　a:=b*−c+b*−c 的图表示法
(a)语法树;(b)DAG。

7.1.3 三地址代码

三地址代码：`x := y op z`

- 每个语句右边只能有一个运算符
- 是图表示法的一种线性表示
- 是中间代码的一种抽象形式

> 例：`x + y * z`可以转换为如下三地址代码：
>
> - `T1 := y * z`
> - `T2 := x + T1`

三地址语句：

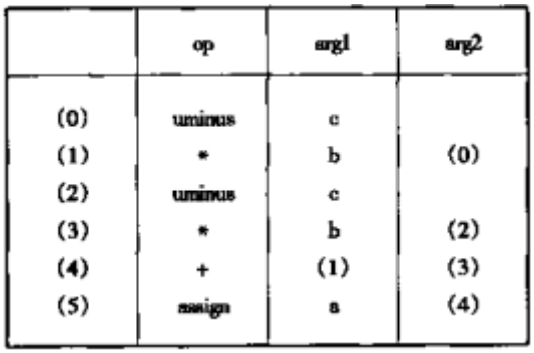
- 种类：
  - x:=y op z
  - x:=op y
  - x:=y
  - goto L
  - if x relop y go L或if a goto L
  - 传参、转子：param x、call p,n；返回语句：return y
  - 地址和指针赋值：x:=&y、x:=*y*、x:=y
  - 索引赋值：x:=y[i]、x[i]:=
- 具体实现：四元式、三元式、间接三元式

三地址语句具体实现

下面说明三地址代码与各实现方式的对应关系：

1. 四元式`(op, arg1, arg2, result)`
   - 计算结果：result域
   - `x := y op z`可表示为`(op, y, z, x)`
2. 三元式`(op, arg1, arg2)`
   - 计算结果：引用该语句的位置（编号）

- 样例：`a := b * -c + b * -c`：
  - `T0 := -c`
  - `T1 := b * T0`
  - `T2 := -c`
  - `T3 := b * T2`
  - `T4 := T1 + T3`
  - `a := T4`



	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

3. 间接三元式：三元式表+间接码表
   - 目的：
     - 便于优化。调整运算顺序时，只需重新安排间接码表。
     - 避免在三元式表中重复填入项。例如上述三元式表中计算b*-c的部分，有两对重复：(0)(2)与(1)(3)，可去除。
   - 间接码表：使用三元式表中的位置表示三元式。即按照三元式出现顺序，引用三元式表位置表达。
   - 样例：继续使用上例`a := b * -c + b * -c`
     - 三元式表：

序号	op	arg1	arg2
0	uminus	c	
1	*	b	(0)
2	+	(1)	(1)
3	:=	a	(2)

     - 间接码表：

间接代码
(0)
(1)
(0)
(1)
(2)
(3)

> 三种方式的对比：

四元式	按编号次序计算	计算结果存于result	方便移动，计算次序容易调整	大量引入临时变量
三元式	按编号次序计算	由编号代表	不方便移动	在代码生成时进行临时变量的分配
间接三元式	按编号次序计算		方便移动，计算次序容易调整	在代码生成时进行临时变量的分配

7.2 说明语句

说明语句的翻译：

- 一般不产生代码
- 将变量名字及其相关属性插入符号表，产生表项。
  - 符号表信息：`name  type  offset`
    - name：名字
    - type：类型
    - offset：在该过程中的相对地址

      > 例如声明`int a, b;`。若int在目标机中占4个字节，则a的offset是0，b的offset是4
  - 对符号表的操作：
    1. `mktable(previous)`：创建一张新符号表。previous表示表头信息
    2. `enter(table, name, type, offset)`：插入新表项
    3. `addWidth(table, width)`：记录总域宽
    4. `enterproc(table, name, newtable)`：在外围过程符号表中建立内嵌过程的新表项

7.2.1 过程中的说明语句（过程中的局部声明）

- 仅涉及符号表操作 `enter(table, name, type, offset)`
- 各种说明语句的type、offset表项信息计算：

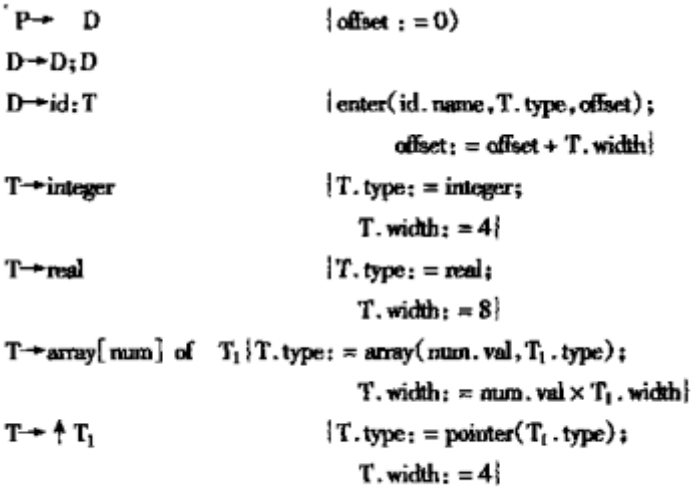


图 7.6 计算说明语句中名字的类型和相对地址

7.2.2 保留作用域信息（嵌套过程声明）

- 每个过程有一个独立的符号表；局部名字信息使用上述方式计算，并添加到对应符号表
  - 每个过程局部声明开始的第一个offset=0，与外部无关
- 每个符号表可以添加一个表头header表示其他信息，如嵌套深度等等。即mktable的参数previous，可以用于找到定义该过程名字的过程符号表。
- 关于`enterproc(table, name, newtable)`：
  - 在外围符号表table插入声明的过程名name
  - 该过程name对应的符号表为newtable

> 嵌套过程符号表即可通过header信息和过程名表项的newtable链接起来，相互找到：

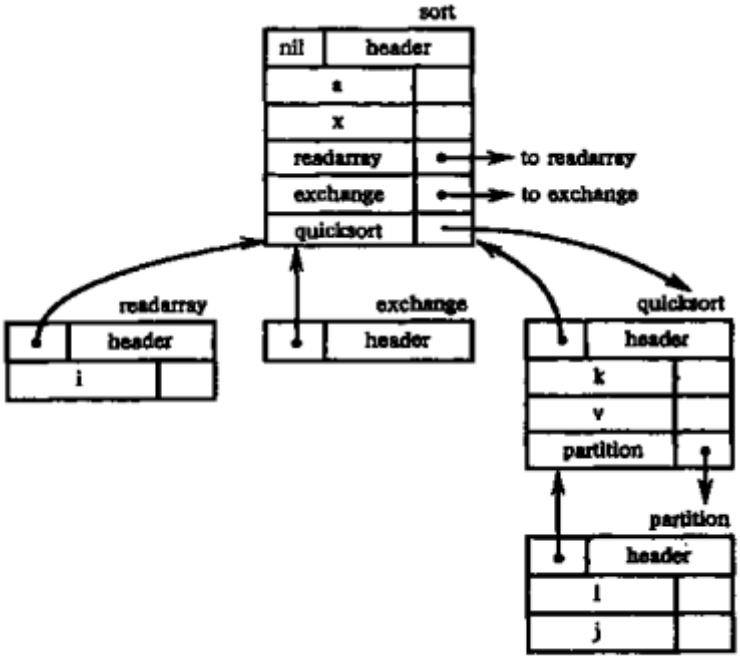


图 7.7　嵌套过程的符号表

- 嵌套过程中的说明语句处理
  - tblptr栈：保存各外层过程的符号表指针

offset栈：保存相对地址。栈顶即为当前被处理过程的下一个局部名字的相对地址

```
P→M D          { addwidth(top(tblptr), top(offset));
                   pop(tblptr);  pop(offset)  }
M→ε            { t: = mktable(nil);
                 push(t, tblptr); push(0, offset)  }
D→D1 ; D2
D→proc id; N D1 ; S   { t: = top(tblptr);
                        addwidth(t, top(offset));
                        pop(tblptr); pop(offset);
                        enterproc(top(thlptr), id.name, t) }
D→id:T         { enter(top(tblptr), id.name, T.type,
                   top(offset));
                   top(offset); = top(offset) + T.width }
N→ε            { t: = mktable(top(tblptr));
                     push(t, tblptr); push(0, offset) }
```

图 7.8　处理嵌套过程中的说明语句

7.3 赋值语句的翻译

> 赋值语句中表达式的类型：整型、实型、数组和记录。
> 讨论如何在符号表中查找名字及如何存取数组和记录的元素。

7.3.1 简单算术表达式及赋值语句

- 下图给出了把简单算术表达式及赋值语句翻译为三地址代码的翻译模式。该翻译模式中还说明了如何查找符号表的入口。属性 **id.name**表示id所代表的名字本身。过程**lookup(id. name)**检查是否在符号表中存在相应此名字的入口。如果有,则返回一个指向该表项的指针,否则,返回 nil 表示没有找到。

$S \rightarrow id := E$	{p := lookup(id.name); if p ≠ nil then emit(p ':=' E.place) else error}
$E \rightarrow E_1 + E_2$	{E.place := newtemp; emit(E.place ':=' E_1.place '+' E_2.place)}
$E \rightarrow E_1 * E_2$	{E.place := newtemp; emit(E.place ':=' E_1.place '*' E_2.place)}
$E \rightarrow E_1$	{E.place := newtemp; emit(E.place ':=' 'uminus' E_1.place)}
$E \rightarrow (E_1)$	{E.place := E_1.place}
$E \rightarrow id$	{p := lookup(id.name); if p ≠ nil then E.place := p else error}

图 7.10　产生赋值语句三地址代码的翻译模式

- 由 S 所产生的赋值语句中的名字必须或者是在 S 所在的那个过程中已被说明，或者是在某个外层过程中已被说明。
- 当应用到 name 时,新的 lookup 过程先通过 top(tblptr) 指针在当前符号表中查找，看是否 name 在表中。若未找到，lookup 就利用当前符号表表头的指针找到该符号表的外围符号表，然后在那里查找名字name，一直到查找出 name 为止。如果所有外围过程的符号表中均无此 name，则 lookup 返回 nil，表明查找失败。

7.3.2 数组元素的引用

- 若数组 A 的元素存放在一片连续单元里，则可以较容易地访问数组地每个元素。假设数组 A 每个元素宽度为 w ， 则 **A[i]这个元素的起始地址为**

  `base + (i - low) x w`

其中 low 为数组下标的下界并且base是分配给数组的相对地址,即 base 为A的第一个元素 A[low] 的相对地址。

- 上式可以整理为：

`i * w + (base - low x w)`

其中，子表达式`c = base - low x w`可以在处理数组说明时计算出来。我们假定C值存放,在符号表中数组A的对应项中,则A[i]相对地址可由`i x w + c`计算出来。

- 对于多维数组也可以做类似处理。
- FORTRAN采用按列存放，Pascal采用按行存放。

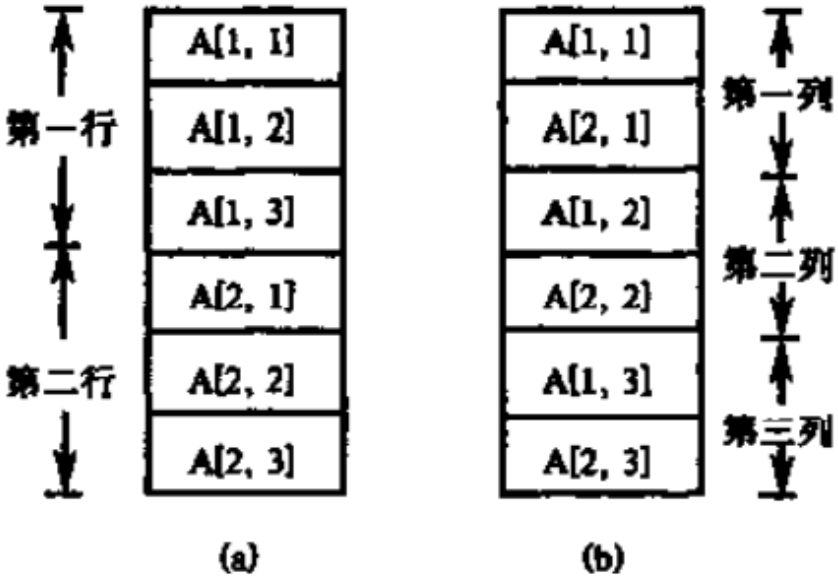


图 7.11　二维数组的存放方式

(a)按行存放；(b)按列存放。

- 对于一个二维数组若按行存放，则可用如下公式计算：

$A[i_1, i_2]$ 的地址：

$$base + ((i_1 - low_1) * n_2 + i_2 - low_2) * w$$

$$= base - ((low_1 * n_2) + low_2) * w + ((i_1 * n_2) + i_2) * w$$

$A[1,2] = base(1*3+1)+(1*3+2) = base+1$

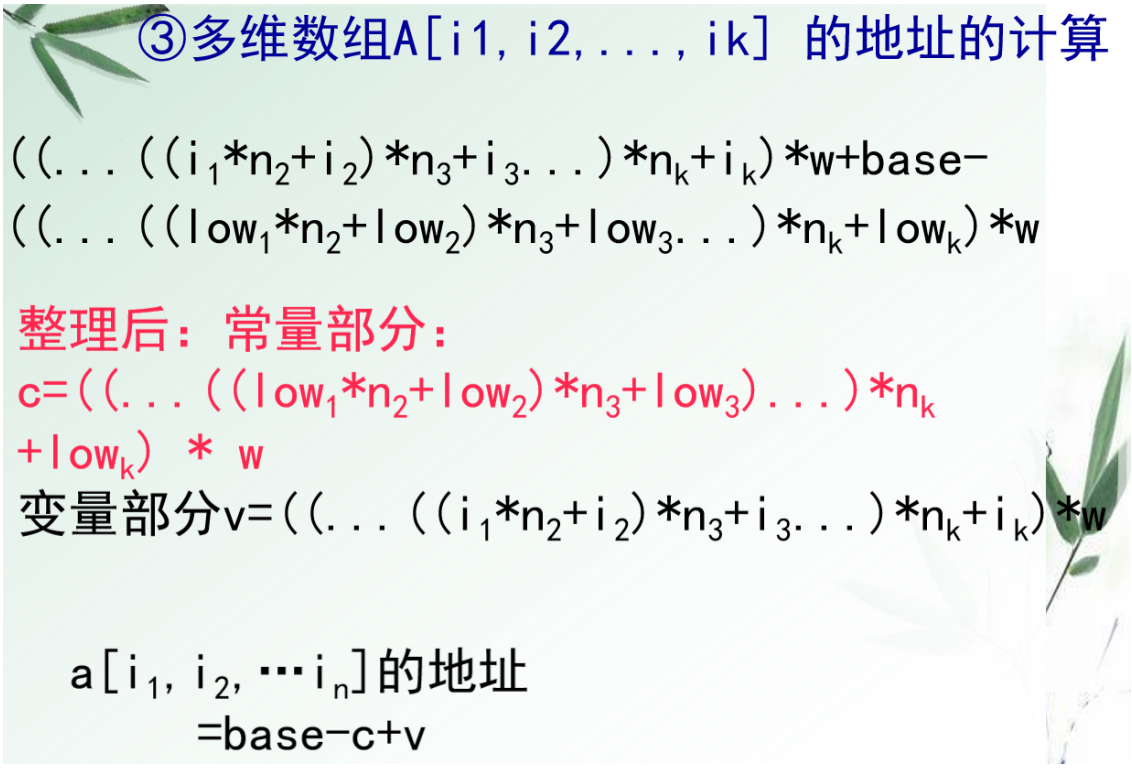
令 $c = ((low_1 * n_2) + low_2) * w$

则常量部分 $= a[low_1, low_2] - c$

A[1, 1]
A[1, 2]
A[1, 3]
A[2, 1]
A[2, 2]
A[2, 3]

A[2, 3]按行存放

- 多维数组地址计算



- 引用数组元素的文法

```
L→ id [Elist] | id
Elist→Elist,E|E
```

为了便于语义处理，上述方法改写为：

```
L→Elist]| id
Elist→Elist,E|id[ E
```

- **一个 Elist 可以产生一个 k－维数组引用 $A[i_1, i_2, \cdots, i_k]$的前 m 维下标，并将生成计算下面式子的三地址代码：**

$$(\cdots((i_1 n_2 + i_2)n_3 + i_3)\cdots)n_m + i_m \qquad (7.8)$$

利用如下的递归公式进行计算：

$$e_1 = i_1,\ e_m = e_{m-1} \times n_m + i_m \qquad (7.9)$$

- 在赋值语句中加入数组元素之后地翻译模式，把语义动作加入到如下文法中：

```
（1）S→L: = E
（2）E→E＋E
（3）E→（E）
（4）E→L
（5）L→Elist]
（6）L→id
（7）Elist→Elist, E
（8）Elist→id[ E
```

若 L 是一个简单名字，将生成一般的赋值；否则，若 L 为数组元素引用，则生成对 L 所指示地址的索引赋值：

1. S→L: = E

```
{if L.offset = null then /* L是简单变量 */
    emit(L.place ': = ' E.place)
 else  emit(L.place '[' L.offset ']' ': = ' E.place)}
```

对于算术表达式的代码完全与图 7.10 相同:

2. $E \rightarrow E_1 + E_2$

```
{E.place: = newtemp;
    emit(E.place ': = ' E1.place '+' E2.place)}
```

3. $E \rightarrow (E_1)$

```
{E.place: = E1.place}
```

当一个数组引用 L 归约到 E 时,我们需要 L 的右值。因此我们使用索引来获得地址 L.place[L.offset]的内容:

4. E→L

```
{if L.offset = null then
    E.place: = L.place
 else begin
    E.place: = newtemp;
    emit(E.place ': = ' L.place '[' L.offset ']')
 end}
```

L.offset 是一个新的临时变量,存放着 w 与 Elist.place 的值的乘积。因此 L.offset 等价于式(7.6)的第一项:

5. L→Elist]

```
{L.place: = newtemp;
emit(L.place ': = ' Elist.array '-' C);    {C 的定义见式(7.7)}
 L.offset: = newtemp;
emit(L.offset ': = ' w '*' Elist.place)}
```

一个空的 offset 表示一个简单的名字:

6. L→id

```
{ L.place: = id.place;   L.offset: = null }
```

每当扫描到下一个下标表达式时,我们应用递归公式(7.9)。在下列语义动作中,Elist1.place 与式(7.7)中的 e_{m-1} 对应,Elist.place 与式(7.9)中的 e_m 对应。注意若 Elist1 有 m-1 个元素,则产生式左部的 Elist 有 m 个元素。

7. Elist→ Elist, E

```
{t: = newtemp;
    m: = Elist.ndim + 1;
    emit(t ': = ' Elist1.place '*' limit(Elist1.array,m));
    emit(t ': = '  t  '+' E.place);
    Elist.array: =  Elist1.array;
    Elist.place: = t;
    Elist.ndim: = m}
```

E.place 保存表达式 E 的值,以及当 m = 1 时式(7.8)之值。

8. Elist→id [E

```
{Elist.place: = E.place;
Elist.ndim: = 1;

Elist.array: = id.place}
```

- 例题：

例 7.1　设 A 为一个 10×20 的数组，即 n1 = 10，n2 = 20，并设 w = 4。对赋值语句 x：= A[y,z]的带注释的语法分析树见图 7.12。该赋值语句被翻译成如下三地址语句序列：

$T_1 := y * 20$

$T_1 := T_1 + z$

$T_2 := A - 84$

$T_3 := 4 * T_1$

$T_4 := T_2[T_3]$

$x := T_4$

其中每个变量，我们用它的名字来代替 id.place。

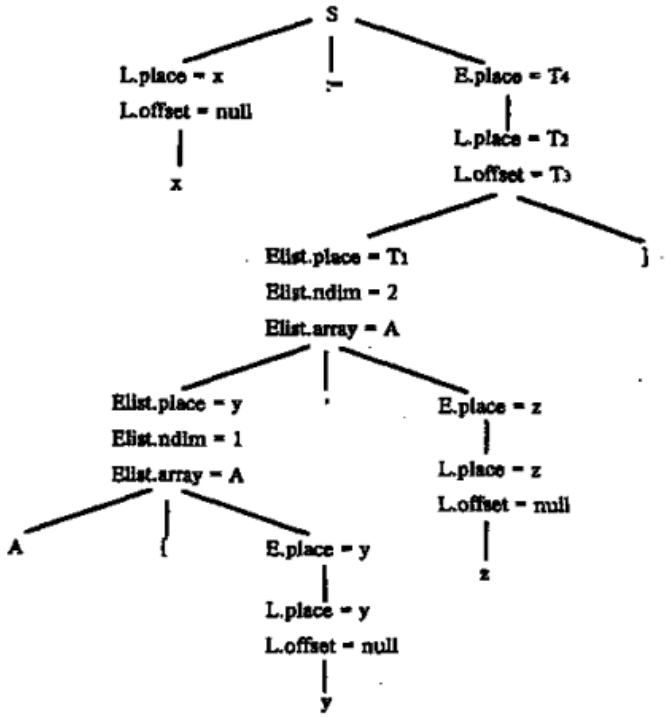


图 7.12　关于 x：= A[y,z]的带注释的分析树

~~7.3.3 记录中域的引用~~

7.4 布尔表达式

产生布尔表达式的文法:

```
E→E or E | E andE | not E | (E) | i rop i | i
```

7.4.1 数值表示法（及其翻译模式）

用1表示真，用0表示假，用这种方式，布尔表达式将从左到右按类似算术表达式的求值方式来计算，例如对于布尔表达式：

```
a<b
```

可以等价表示为：

```
    if a<b then 1 else 0
```

可以翻译成如下三地址代码：

```
100:    if a<b goto 103
101:    T:=0
102:    goto 104
103:    T:=1
104:
```

布尔表达式的数值表示法的翻译模式

•过程emit将三地址代码送到输出文件中

•nextstat给出输出序列中下一条三地址语句的地址索引

每产生一条三地址语句后，过程emit便把nextstat加1

具体的翻译模式如下：

```
E→E1  or E2     {E.place:=newtemp;  emit(E.place ‘:=’ E 1.place ‘or’ E2.place)}
E→E1 and E2     {E.place:=newtemp;  emit(E.place ‘:=’ E 1.place ‘and’ E2.place)}
E→not E1        {E.place:=newtemp;  emit(E.place ‘:=’ ‘not’ E 1.place)}
E→(E1)          {E.place:=E1.place}
E→id1  relop id2   { E.place:=newtemp;
                     emit(‘if’  id1.place  relop. op  id2. place ‘goto’
nextstat+3);
                     emit(E.place ‘:=’ ‘0’);
                     emit(‘goto’ nextstat+2);
                     emit(E.place‘:=’ ‘1’) }
E→id            { E.place:=id.place }
```

例题：

例 7.2　根据图 7.13，对布尔表达式 a < b or c < d and e < f 可以生成图 7.14 中的三地址代码。　即之前所述的翻译模式

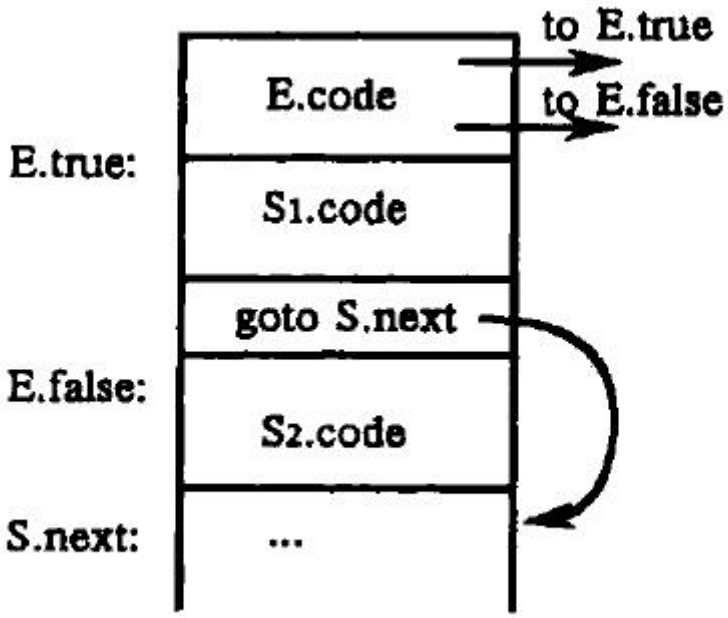
100:	if a < b goto 103	107:	$T_2 := 1$
101:	$T_1 := 0$	108:	if e < f goto 111
102:	goto 104	109:	$T_3 := 0$
103:	$T_1 := 1$	110:	goto 112
104:	if c < d goto 107	111:	$T_3 := 1$
105:	$T_2 := 0$	112:	$T_4 := T_2$ and T_3
106:	goto 108	113:	$T_5 := T_1$ or T_4

7.4.2 作为条件控制的布尔表达式翻译

对于条件语句：

```
if  E  then S1 else S2
```

其中，需要赋予 E 两种出口:一真一假， 真出口指向S1, 假出口指向S2，代码结构示意图：



我们可以把条件控制的布尔表达式翻译成一串跳转指令, 例如:

```
if  a>c or b <d then  S1 else S2
```

可以翻译为:

$$
\begin{aligned}
&\text{if } a > c \text{ goto } L_2\\
&\text{goto } L_1\\
L_1:\quad &\text{if } b < d \text{ goto } L_2\\
&\text{goto } L_3\\
L_2:\quad &\text{(关于 } S_1 \text{ 的三地址代码序列)}
\end{aligned}
$$

$$
\begin{aligned}
&\text{goto Lnext}\\
L_3:\quad &\text{(关于 } S_2 \text{ 的三地址代码序列)}\\
&\text{Lnext:}
\end{aligned}
$$

这其中L2就是真出口，L3是假出口。

基于以上特点，在翻译条件控制的布尔表达式，应该满足如下特点（结合之前小结所述）：

1. 设计两个出口：E.true 和 E.false，E.true 指向S1， E.false指向S2；
2. 一遍扫描，在进行语法规约的同时，进行翻译

3. 中间代码采用四元式形式
4. 拉链反填技术：四元式转移地址有可能无法立即知道,把这个未完成的四元式地址作为E的语义值保存,待机"回填"。具体做法为：

   1）为非终结符E赋予两个综合属性E.truelist和E.falselist。它们分别记录布尔表达式E所应的四元式中需回填“真”、“假”出口的四元式的标号所构成的链表

   2）引入变量nextquad，它指向下一条将要产生但尚未形成的四元式的地址(标号)。nextquad的初值为1，每当执行一次emit之后，nextquad将自动增1。

   3）函数makelist(i)，它将创建一个仅含i的新链表，其中i是四元式数组的一个下标(标号)；函数返回指向这个链的指针。

   4）函数merge(p1,p2)，把以p1和p2为链首的两条链合并为一，作为函数值，回送合并后的链首。

   5）过程backpatch(p, t)，其功能是完成“回填”，把p所链接的每个四元式的第四区段都填为t

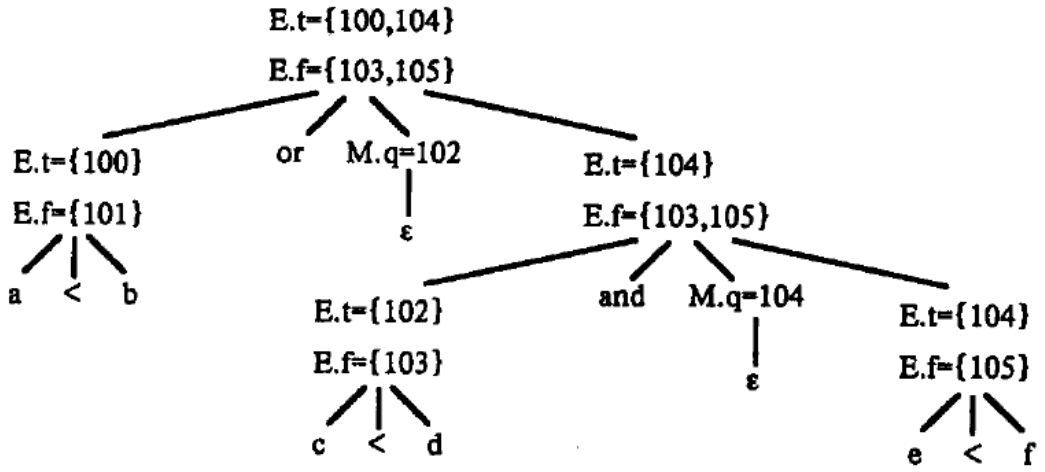
条件控制的布尔表达式翻译模式

```
(1) E→E1 or M E2
    { backpatch(E1.falselist, M.quad); //回填，因为E1为假，就会去E2的首地址
      E.truelist:=merge(E1.truelist, E2.truelist);//合并
      E.falselist:=E2.falselist }

(2) E→E1 and M E2
    { backpatch(E1.truelist, M.quad); //E1为真，还要再去判定E2
      E.truelist:=E2.truelist;// E2为真，则整个表达式为真
      E.falselist:=merge(E1.falselist,E2.falselist) //任何一个为假，都是假
}
(3) E→not E1
    { E.truelist:=E1.falselist;
      E.falselist:=E1.truelist}

(4) E→(E1)
    { E.truelist:=E1.truelist;
      E.falselist:=E1. falselist}
(5) E→id1 relop id2    {
    E.truelist:=makelist(nextquad);
      E.falselist:=makelist(nextquad+1);
      emit(‘j’ relop.op ‘,’ id 1.place ‘,’ id 2.place‘,’  ‘-’);
        emit(‘j, —, —, -’) }

(6) E→id
    { E.truelist:=makelist(nextquad);
      E.falselist:=makelist(nextquad+1);
      emit(‘jnz’ ‘,’ id .place ‘,’ ‘—’ ‘,’  -);
        emit(‘ j, -, -, -) }
(7) M→空
        { M.quad:=nextquad }
```

例题：

```
a<b or c<d and e<f
```

根据翻译模式，可在语义分析时，向语法树加入如下注释：



最终的代码：

100 (j<, a, b, -)

101 (j, -, -, 102)

102 (j<, c, d, 104)

103 (j, -, -, -)

104 (j<, e, f, -)

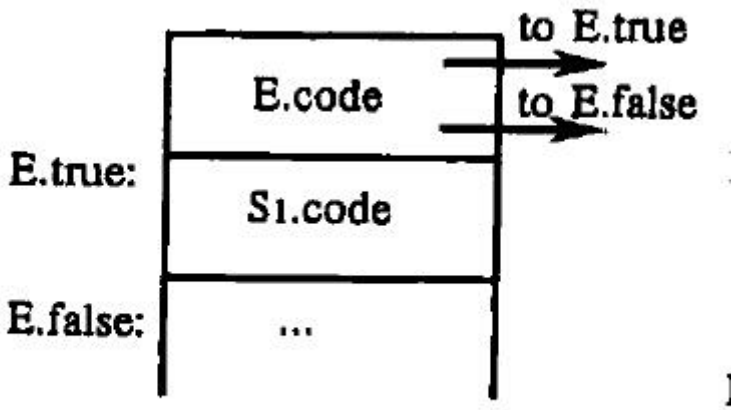
105 (j, -, -, -)

7.5 控制语句的翻译

7.5.1 控制流语句-if-then

```
S→if E then S1
```

对于E来说，同样需要真假两个出口，其中真出口指向S1，代码结构示意图：



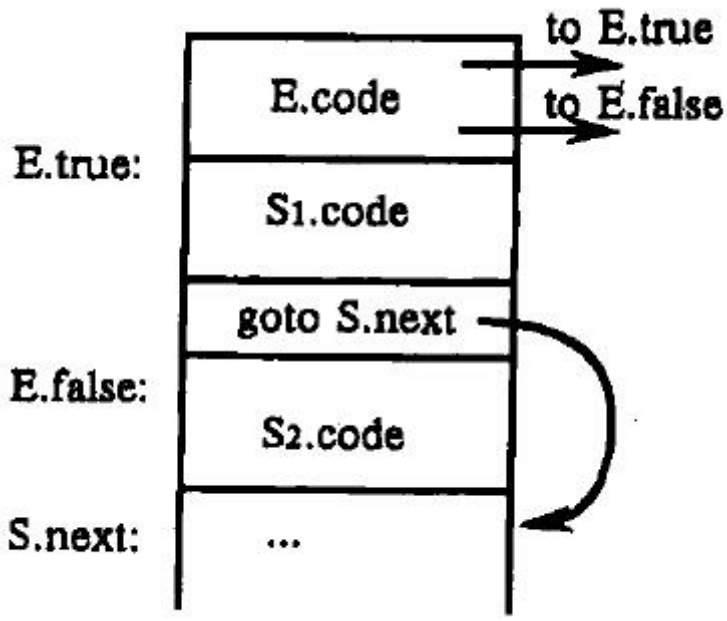
可以用如下语义规则来描述：

```
E.true:=newlabel;
E.flase:=S.next;
S1.next:=S.next
S.code:=E.code ||gen(E.true ‘:’) || S1.code
```

7.5.1 控制流语句 if-then-else

```
S→if E then S1 else S2
```

E的真出口指向S1, 假出口指向S2，S1,S2的出口都是S.next, 代码结构示意图：



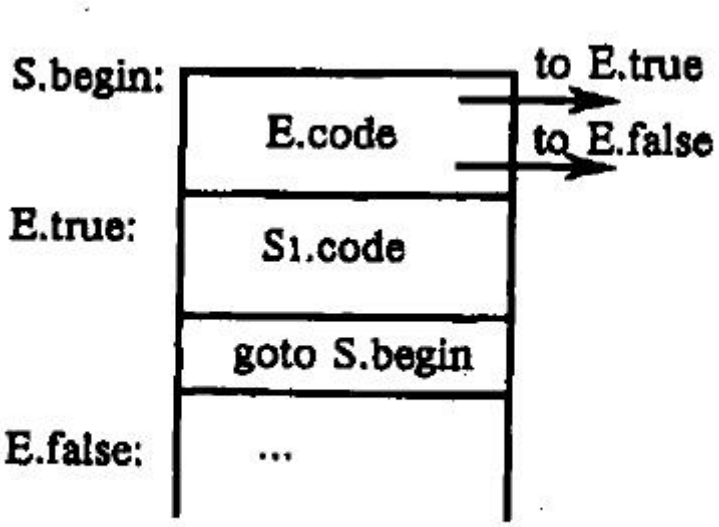
语义规则：

```
E.true:=newlabel;
E.false:=newlabel;
S1.next:=S.next
S2.next:=S.next;
S.code:=E.code || gen(E.true ‘:’) || S1.code
        || gen(‘goto’  S.next) || gen(E.false ‘:’) || S2.code
```

7.5.1 控制流语句 while-do

```
S→while E do S1
```

E的真出口指向S1,S1的出口，需要再指回S的入口 代码结构：



语义规则：

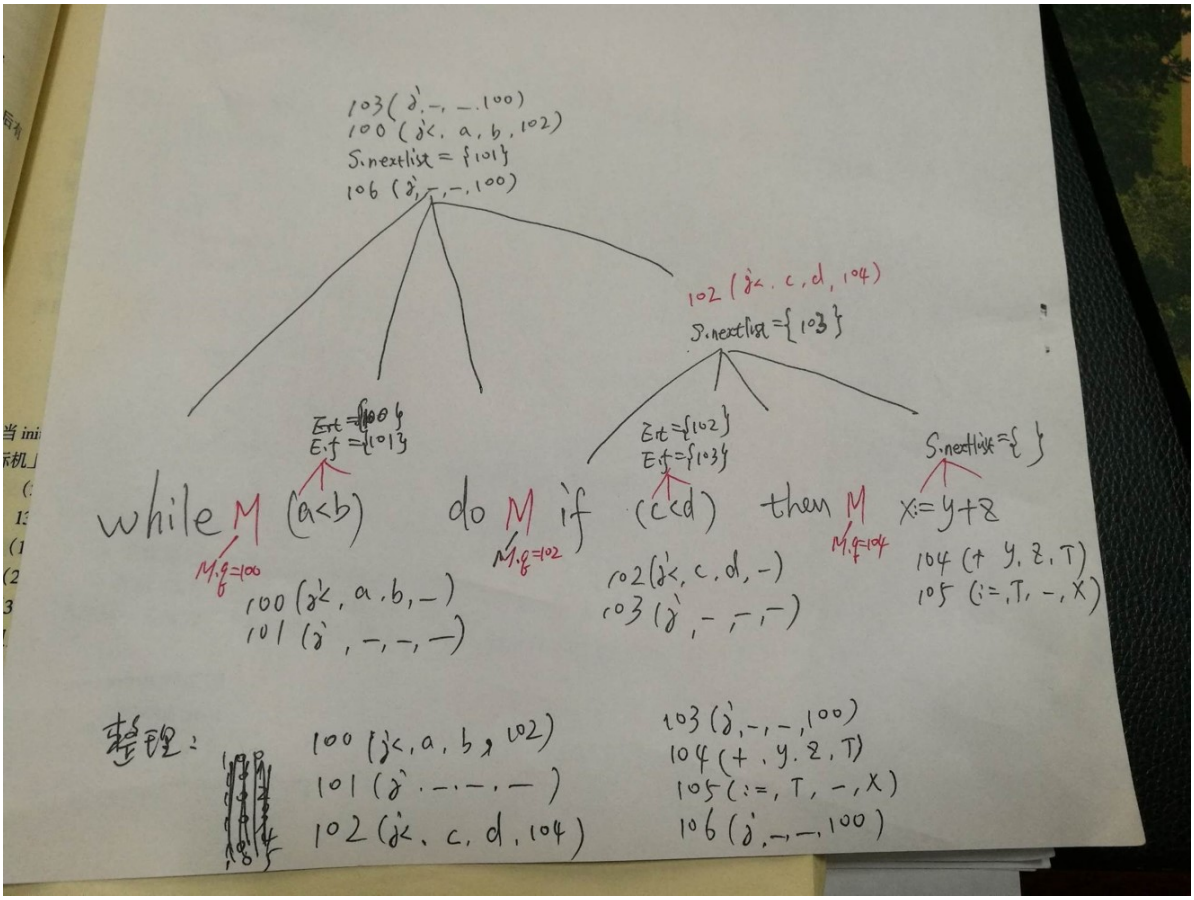
```
S.begin:=newlabel;
E.true:=newlabel;
E.false:=S.next;
S1.next:=S.begin;
S.code:=gen(S.begin ‘:’) || E.code || gen(E.true ‘:’) || S1.code || gen(‘goto’
S.begin)
```

7.5.1 控制流语句 翻译模式

```
1. S→if  E  then  M  S1
 { backpatch(E.truelist, M.quad);
    S.nextlist:=merge(E.falselist, S1.nextlist) }
2. S→if  E  then  M1  S1  N  else  M2  S2
 {  backpatch(E.truelist, M1.quad);
    backpatch(E.falselist, M2.quad);
    S.nextlist:=merge(S1.nextlist, N.nextlist, S2.nextlist) }
3. M→空  { M.quad:=nextquad }
4. N→空  { N.nextlist:=makelist(nextquad);  emit(‘j,－,－,－’) }
5. S→while M1 E do M2 S1
   { backpatch(S1.nextlist, M1.quad);
    backpatch(E.truelist, M2.quad);
    S.nextlist:=E.falselist
    emit(‘j,－,－,’ M1.quad)
    }
```

例题1：

```
while (a<b) do
      if (c<d) then x:=y+z;
```



注意，课本上的答案有误，101指令第四个参数不确定，不能确定跳转的位置。

9.1 参数传递（尹浩飞）

过程的定义

过程或函数是模块程序设计的主要手段，也是节省程序代码和扩充语言的主要途径。

- 过程的定义

```
procedure add(x, y : integer; var z : integer)
begin
    z := x + y;
end;
```

其中，x, y 为值参数，z 是变量参数。

过程的调用

- 过程调用

```
add(a, b, c);
```

a, b, c 都是实参，过程调用时，需要把实参传递给形参。

参数的传递

- 参数传递的方式
  - 传地址（PASCAL 变量参数，C 传引用）
    - 把实参的地址传递给被调用过程的**形式单元**中
    - 在过程调用中，形式参数的引用或复制，全部处理成对形式单元的间接访问（访问实参的地址，修改其对应的值）
  - 得结果（不重要）
  - 传值（不重要）
  - 传名（不重要）

9.2 运行时存储器的划分（尹浩飞）

运行时所需的存储空间

- 一个目标程序运行时所需的存储空间包括
  - 存放目标代码的空间
  - 存放数据项目的空间
  - 存放程序运行时控制或连接数据所需的单元

编译程序需要考虑的问题

- 编译程序组织存储空间时需要考虑的问题：
  - 过程调用是否允许递归?
    - 决定采用动态分配（允许递归）还是静态分配（不允许递归）

- 从一个过程返回时，对局部名称的值如何处理？
  - 一个过程结束后，该过程的局部值删除或可以继续访问。
- 过程是否允许引用非局部的名字？
  - 引用父局部的名字
  - 引入全局的名字
- 过程如何传递参数？过程是否可以作为参数被传递或作为结果被返回？
  - 9.1 中描述的参数传递方式
  - 如何描述过程，例如C语言中，函数作为过程有函数指针，可以被传递或返回。
- 存储空间是否可以在程序控制下动态分配？
  - Fortran 语言无法动态分配
  - C/C++ 语言可以动态申请内存
- 存储空间是否需要显式的释放？
  - C++语言，new的内存需要手动delete
  - Java语言，new的内存，虚拟机自动回收

存储分配的策略

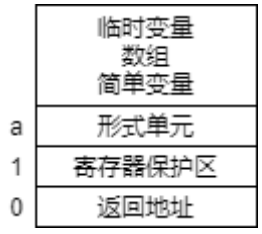
- 存储的分配策略
  - 静态分配策略
    - 在编译时可以确定分配空间的大小
  - 动态分配错误
    - 允许递归，允许动态申请内存
    - 分类
      - 栈式分配
      - 堆式分配

9.3 静态存储分配（尹浩飞）

> 下面的例子都以 Fortran 语言为例

静态的存储分配有以下特点：

- 程序所需要的空间总量，在编译时就可以确定。
  - 不允许直接或间接的递归调用
  - 不允许动态申请与释放内存。
- 编译程序可以确定每一个数据对象在内存中的地址。
- 每个程序段可以独立的进行编译，之后由链接装配程序把各段连接成可运行的整体。
- 按数据区为单位组织存储。
- 每个程序段在同一时刻的内存中，只会有一个活动，所以为每个程序段分配一个局部数据区即可。



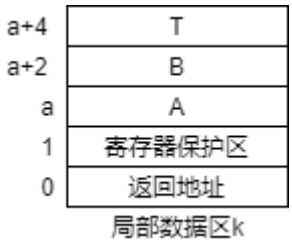
以下述的 Fortran 代码为例，以下代码要执行SWAP的过程。

```
SUBROUTINE SWAP(A,B)
T=A
A=B
B=T
RETURN
END
```

其参数列表如下图所示：

名字	性质	地址	
		DA	OFFSET
SWAP	子程序,二目		
A	哑元,实型	k	a
B	哑元,实型	k	a+2
T	实型变量	k	a+4

在编译时就可以确定的地址空间可以表示为：



9.4 简单的栈式存储分配（王新宇）

程序语言模型

简单语言

本节的内容是简单的栈式存储分配，这里的简单表示一种简单的程序语言模型。这个程序语言过程定义不能嵌套，不过我们允许过程的递归调用。这种语言与C语言类似，而实验要求的PL0出现了过程定义嵌套因此不属于这种语言。下面给出两个样例，解释简单语言。

下面的样例符合简单语言的要求

```
void R()
{
    // 不能在其中定义其他函数
}

void P()
{
    // 可以递归调用 自己或其他函数
    Q();
```

```
    P(); //递归调用
}
int main()
{
    P();
}
```

下面的样例不满足简单语言定义

```
void R()
{
    void P()// 不允许嵌套定义
    {
        void Q() // 嵌套定义的情况请看下一节
        {

        }
    }
}
int main()
{
    R();
}
```

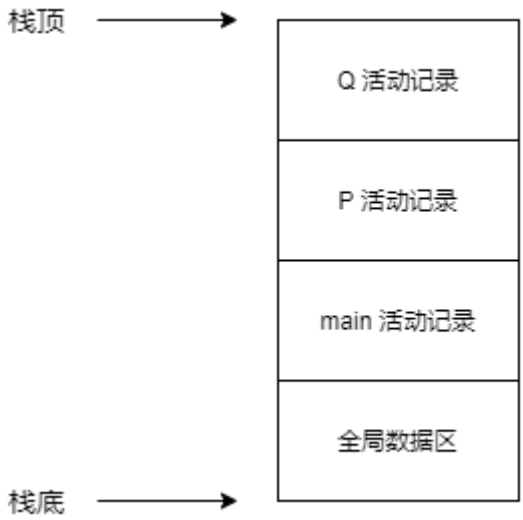
为什么是简单语言?

简单语言与允许过程嵌套的语言之间差异最大的就是变量的访问范围，后者一个过程不仅能访问自己内部定义的变量和全局变量还能访问自己外层过程中的变量，而前者仅能访问过程内部定义的变量和全局变量，这就极大程度上方便了变量的访问。

栈的存储原理

栈式存储分配意味着我们要把整个内存空间看作是一个大的栈，栈底部首先放入全局数据，然后每调用一个过程就将该过程的活动记录压入到栈中，当过程执行结束后，就弹栈，这样我们能保证栈顶的数据就是当前正在执行的过程的活动记录。

```
void Q()
{
    // some code
}
void P()
{
    Q();
}
int main()
{
    P();
}
```

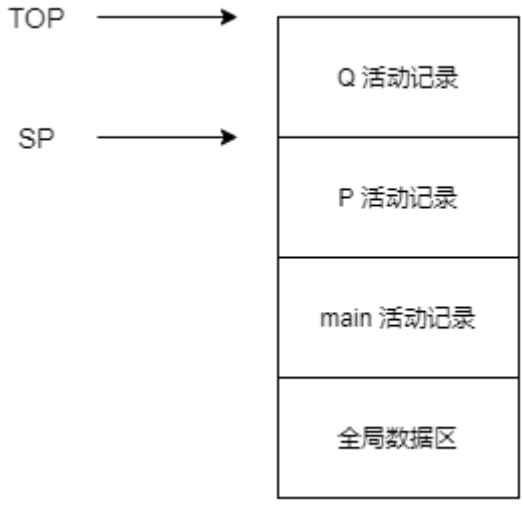


上面给出了一个程序和他执行时栈的情况。此时程序在main中调用了P，然后在P中调用了Q，我们可以看到，main首先入栈，然后P入栈，最后Q入栈，此时正在执行Q的相关代码，栈顶也是Q的活动记录。

变量访问

存储分配最终的要的是让我们能访问到需要的变量，在简单语言中，我们仅能访问全局变量和过程内的局部变量。全局变量是相对固定的，所以我们直接记录它在全局数据区的绝对地址，在指令中填写绝对地址就能访问了。对于局部变量，我们会它存放在活动记录中，我们在指令中记录它在活动记录中的相对地址(可以静态算出)，然后在执行时记录活动记录的地址，最后将两个地址相加即可得出绝对地址。

我们在处理器中设置两个寄存器SP(stack pointer) 和 TOP ，分别记录栈顶的活动记录的基址和栈顶位置，如下图所示

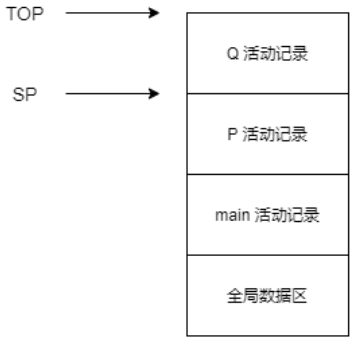


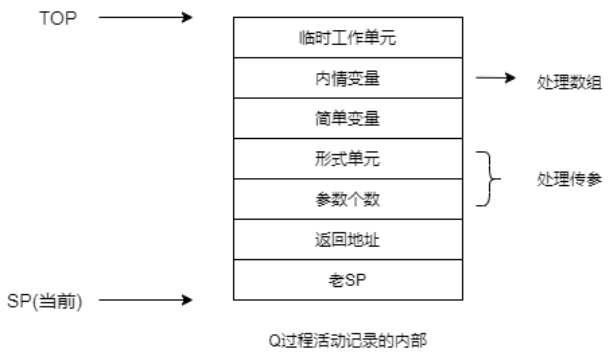
这样做，我们编译时确定一个局部变量的地址只需要关注该变量在活动记录中的偏移就可以了。在执行时当新的活动记录入栈，只需要知道TOP的值就可以了。

活动记录

> 活动记录是什么?
>
> 在课本中我们叫的活动记录就是我们通常理解的函数栈，函数在执行时会对许多变量进行操作，这些变量就存储在函数栈中，此外为了程序的正常运行，函数栈中还会额外记录一下数据，告诉函数执行完后应该恢复成什么状态以及去哪里执行下一条指令等等。

活动记录虽然被作为一个整体被压入栈，但是活动记录内部也是一个类似与栈的结构。





Q过程活动记录的内部

两个连接数据，分别是老SP和返回地址，老SP是上一个过程活动记录的SP(也就是活动记录的基址)，在活动记录入栈时记录上一个过程的SP，当发生弹栈时要将SP寄存器恢复为老SP的值（同理，TOP也要恢复，我们可以把TOP恢复为当前SP的值，也可以额外开一个老TOP来记录）。返回地址给出当前过程的返回地址，当过程执行完毕后返回到该地址继续执行。

参数个数和形式单元用于记录调用时向该过程传入的参数的信息。简单变量存储的是过程内部的局部变量，我们只需要知道一个变量相较于SP的偏移量Offset，就可通过SP+Offset(可以表示为Offset[SP])来获得该变量的绝对地址进行访问了。这种寻址方式是一种基址寻址，SP充当了基址寄存器。内情变量是处理数组访问使用的，因为不要求掌握，因此不做过多的介绍。临时工作单元用于存放临时的计算结果和一些中间结果。

过程调用与返回

过程调用中含有传入参数的相关部分，但是传参数的内容并不是教学的主要内容，因此不过多介绍有关参数传递的内容，仅仅讲述过程调用和调用结束返回的内容。

我们假设过程调用的指令是call xxx，xxx是新过程代码的地址。此时我们要构建新过程的活动记录，并将活动记录压入栈。当前的TOP寄存器是原过程的顶部，也是新过程底部，因此通过TOP就可以访问到新过程的活动记录。此时的SP是旧过程的SP，在新过程看来就是老SP，PC+1为旧过程中call指令的下一条指令，也就是新过程应该的返回地址。所以我们令1[TOP] = SP ; 2[TOP] = PC+1 便在新过程的活动记录中记录老SP和返回地址。之后设置SP 为 TOP+1，也就是新活动记录的基址，TOP为 TOP+L，L为新过程的活动记录大小，可以静态算出，PC设置为call xxx 中的xxx。

函数返回是一个逆过程，TOP恢复为SP-1，SP恢复为老SP，PC指向活动记录中的返回地址即可。

9.5 嵌套过程语言的栈式实现

> 实验使用

程序语言模型（王新宇）

上一节我们讨论的是简单的栈式存储分配，其使用的程序语言不允许出现过程的嵌套，这里我们要打破整个限制，允许出现嵌套的过程定义。下面简单给出一个嵌套过程的语言样例和过程变量访问原则。

```
void P()
{
    int a;
    void Q()
    {
        int b;
        a = 3; // 允许访问包含本过程的外部过程定义的变量
        void T()
        {
            b=1;
```

```
            a=4;
            T();// 允许递归调用
            Q();// 调用自己的外层
        }
        T();// 可以调用自己内部定义的过程
    }
    void S()
    {
        a = 3;
        Q() ; // 允许调用同级之前的过程
        //T() ; // 该调用不被允许，不能访问其他过程内部的过程
    }
    Q();
    S();
    //T(); // 只能访问到自己内部一级的，内部过程中定义的不能访问。
}
int main()
{
    P();
}
```

总结一下访问规则：

> 这个原则也是实验中要求的 名词：函数=过程

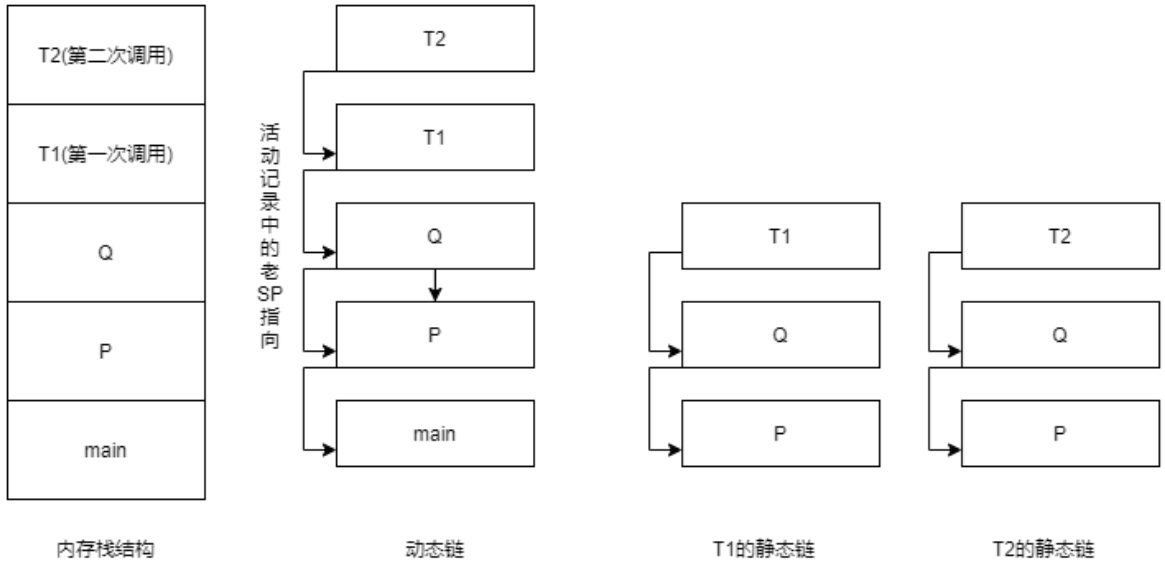
1. 函数可以调用比自己先声明的同层次函数，被自己直接包含的函数，函数本身，以及包含自己的高层次函数。
2. 变量访问总是访问本层或者层次低于此层的变量，如果出现同名变量则取层次最靠近自己的那个变量。

一个过程的外层指的是包含它的过程，比如T的外层有Q和P。S不是T的外层，因为T不在S中声明。直接外层指的是直接包含它的过程，T的直接外层是Q。

要支持嵌套定义麻烦的是支持嵌套情况下的变量访问。如果访问过程内部自己定义的局部变量，我们可以按照上一节的模型，通过SP的基址寻址实现，但是访问外层的变量较为麻烦，比如我们要在上例中从Q过程中访问P的变量a，我们在执行Q时SP指向的是Q的活动记录的地址，但我们想访问a必须知道P的活动记录的地址，这就是访问的难题所在。

静态链与动态链（王新宇）

要解决外部变量的访问，我们要借助静态链。首先我们先明确什么是静态链什么是动态链。动态链是指函数在执行时不断压栈构成的链，反应的是实际执行的调用关系。静态链则与函数的执行无关，是为了访问外层变量，与编写时静态的层次关系有关。假设一个过程要访问他任意一个外层过程的变量，那么**他必须知道它所有外层过程的最新活动记录地址**，这个地址就是通过静态链维护。我们还是使用上面的例子，main调用P，P调用Q，Q调用了T，T在内部递归调用了T。此时链接的情况如下图



动态链就是栈的实际情况，即便在T1中递归调用了T2，新的T2也会指向旧的T1. T过程在编写时嵌套在Q和P内部，则无论T递归多少次，都应该指向编写时的外层Q和P，所以T1，T2的静态链是一样的。

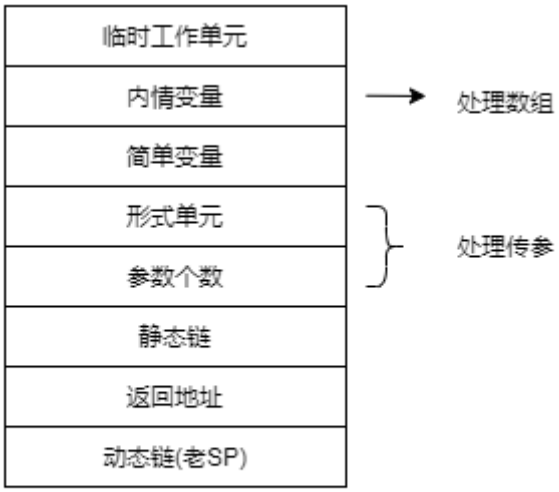
静态链本质上是记录的是编写时静态的层次关系，每一个过程都指向最新的直接外层，最终实现外层变量的访问。

静态链的实现方式

> 关于静态链给出两种实现方式

单指针（王新宇）

我们可以在活动记录中添加一个单元记录直接外层的活动记录地址，新的活动结构如下：

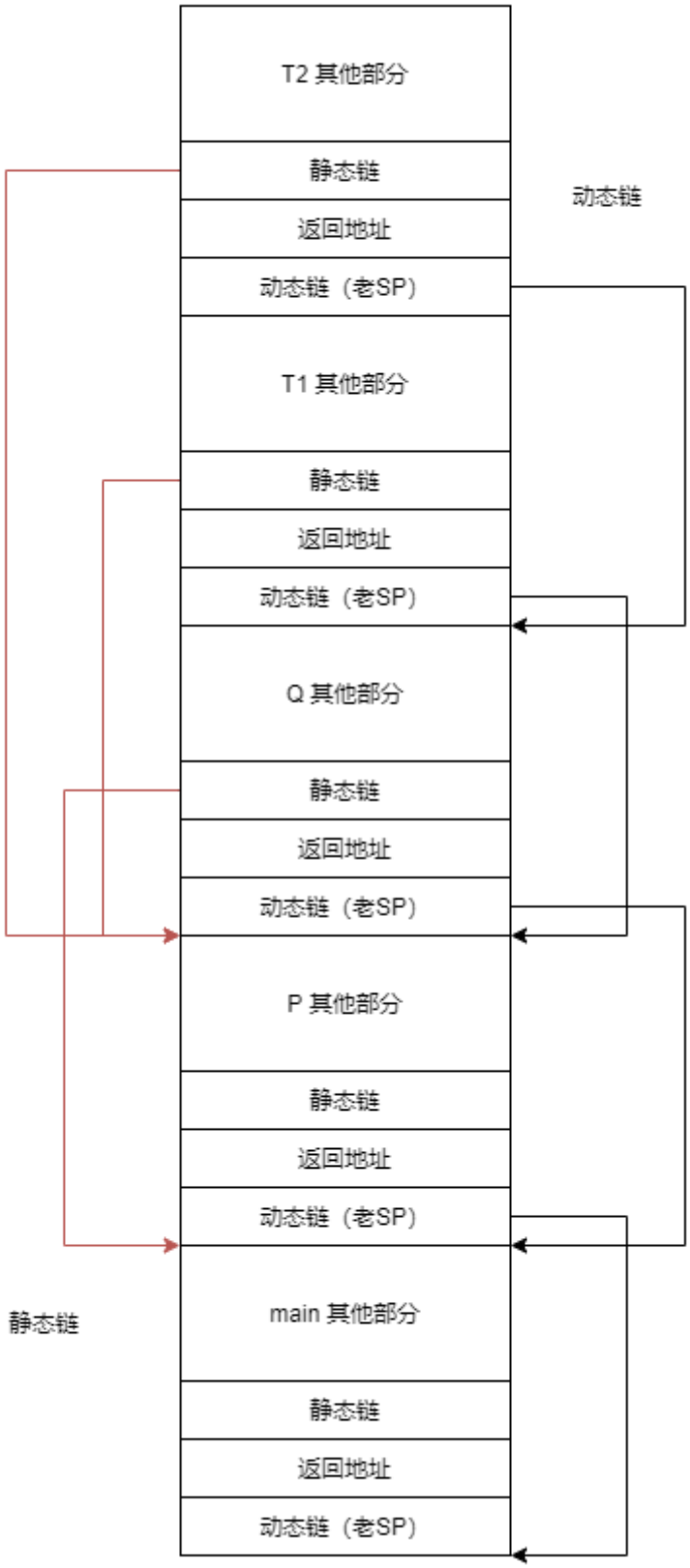


活动记录

变量访问

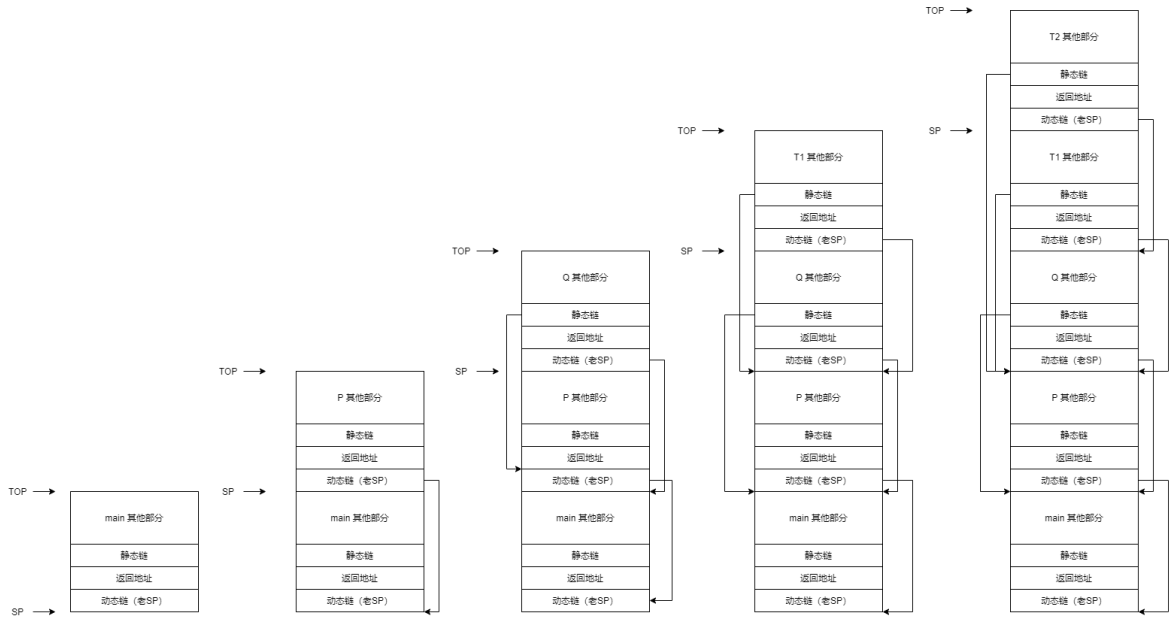
新添加的静态链单元(**后面的"静态链"理解为地址吧，课本起名太奇怪了**)存储的是一个地址，是最新的直接外层的活动记录地址。如果这时我们想访问直接外层的变量，只需要知道其offset即可，offset+静态链得出的就是该变量的绝对地址。当访问外层的外层时，只需要两次静态链访问即可。下面给出一个例子，讲解静态链的结构。

我们函数的调用结构还是参考上面，main->P->Q->T1->T2



可以看到T2和T1的静态链是相同的，都指向Q的活动记录地址，Q的静态链则指向P的活动记录，注意P没有外层所以P的静态链为空。

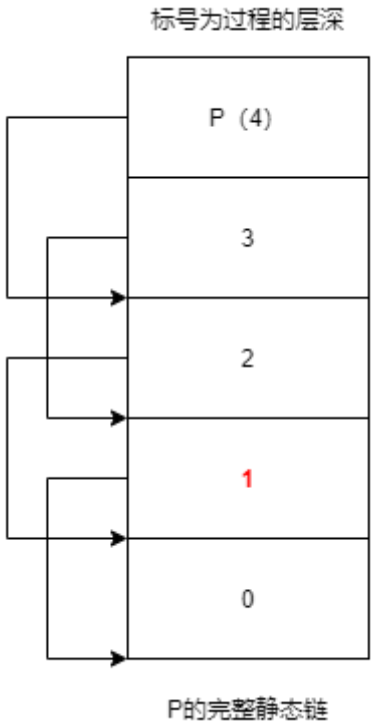
我们还可以模拟一下构建的过程



代码实现

我们现在已经很清楚静态链的结构了，难题在于如何维护静态链，我们调用一个新的过程时会压入一个新的活动记录，在压入时我们就要构建好该活动记录的静态链。下面给出一种简单的方式：

我们假设有一个过程P，静态的层深是4，P的某个外层过程Q层深是2，在P中调用了Q，假设当前正在执行Q活动记录的压栈过程。我们要找到Q的静态链需要借助P的帮忙。P的层深是4，那么他的静态链肯定能跳4次达到层深为0的最外层(这个是根据调用规则保证的，外层只能调用直接内层，不能调用更深的内层)。因为Q是P的外层，因此Q的外层必然是P的外层，那么Q需要的静态链必然在P静态链向下跳的过程中被找到。

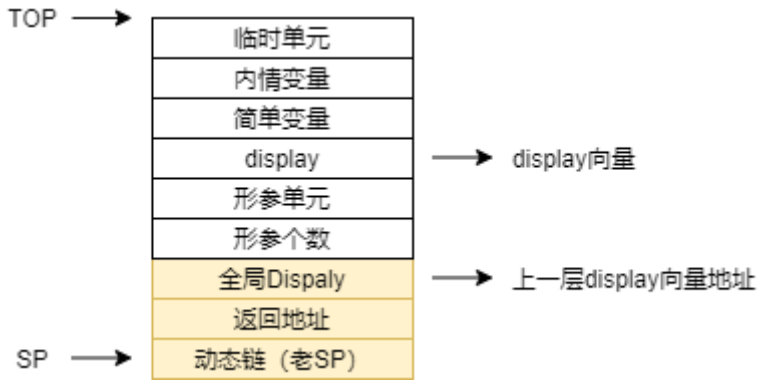


看上图，1是静态层深为1的过程，它是P的外层同时也层深为2的Q的外层，且正好是Q的直接外层，所以Q的活动记录中静态链指针指向1的活动记录地址即可，具体的值通过P的静态链3次跳转即可。

当然上述只是一种情况，其他情况大家可以自行分析，这个思路与下面的DISKPLAY表方法有异曲同工之妙。

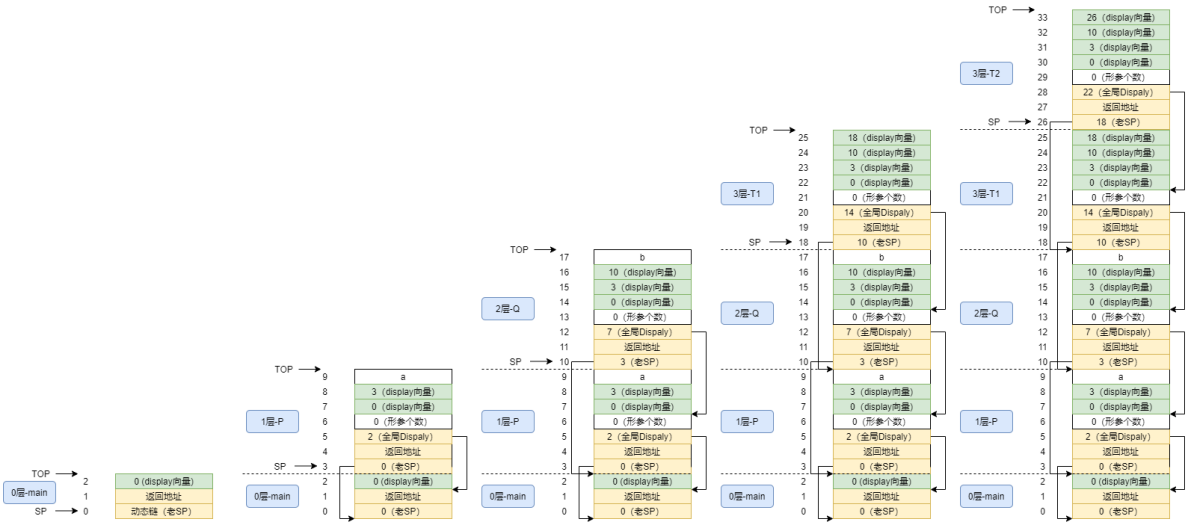
嵌套层次显示表(display)（尹浩飞）

在上一节的单指针的部分可以看到，若果要访问一个上层的变量，需要逐层向上寻找，所以，为了提高访问非局部变量的速度，可以引用一个指针数组，该数组名为**嵌套层次显示表**。这种方法的思想就是把之前所有外层的 SP 指针的值，全部存在当前层的栈空间当中，其结构可以表示为下图：



其中底色为黄色的三个值为连接数据，均由调用当前过程的部分负责填写。其中，全局Display存储了上一层的display向量的首地址，以方便当前层在构建display向量的时候，可以直接从上层进行复制。

我们函数的调用结构还是参考上面，main->P->Q->T1->T2，其构建过程为：



变量访问

基于Display表进行的变量访问十分容易，只要我们知道要访问的变量所在的层次，就可以直接在当前层的display向量中直接访问到对应成的SP的值，再根据要访问变量在所在层的相对位置，就可以直接获取到所需要的值了。例如，已知当前层的基地址为 SP，display表在当前层中的相对位置为 d，已知要访问的变量所在的层是 k，变量在所在层中的相对位置为 x，则可以通过 $(d+k)[SP]$ 得到变量所在层的 SP' 值，然后在通过 $x[SP']$ 就可以获取到所需要的值了。

如上图所示，若要在 T2 中访问变量 a，只需要知道，a 位于 1 层的 $+6$ 位置。首先我们从当前的 $SP \to 26$ 开始，先获取当前活动单元中 display 向量的偏移（$+4$），然后找到 1 层 SP 指针 SP_1 的位置，即：$SP_1 = (+4+1)[SP] = 3$，然后根据在 P 中的变量位置偏移，找到 $a = (+6)[SP_1]$。

代码实现

相较于维护动态链，display表要多维护的一个量为：

- 当前过程的层数
- display向量在当前活动单元中的偏移（由于参数是固定的，所以该值在编译时也是固定的）

在处理连接数据时，需要多传递一个量：全局display指针。该值可以直接选取当前活动单元中，display向量的首地址的位置（通过上面维护的偏移量获得）。

在构造当前display向量时，首先维护当前的位置为display向量的偏移，然后根据当前过程的层数（设当前层为 I），在当前活动单元全局display指针指向的位置开始，复制 $I-1$ 个数据到当前的display向量，然后再将当前的活动单元的 SP 的值放在最后一个display向量的位置上即可。

第十章 优化

10.1 概述 - 张雨

概念：

对程序进行各种等价变换，使得从变换后的程序出发，能生成更有效的目标代码。

- 等价：不改变程序的运行结果
- 有效：目标代码运行时间短，占用存储空间小

目的：

产生更高效的代码

遵循的原则：

- 等价原则：优化不应改变程序运行的结果
- 有效原则：使优化后所产生的目标代码运行时间较短，占用的存储空间较小
- 合算原则：应尽可能以较低的代价取得较好的优化效果

优化的级别：

- 局部优化
- 循环优化
- 全局优化

优化的种类：

- 删除公用子表达式

  如果一个表达式E在前面已计算过,并且在这之后E中变量的值没有改变,则称E为公共子表达式。对于公共子表达式,我们可以避免对它的重复计算,称为删除公共子表达式(有时称删除多余运算)。

- 复写传播

  复写传播的目的是使对某些变量的赋值变为无用。

- 删除无用代码

  对于变量及临时变量,如果这些变量的值在整个程序中不再被使用,因此,这些变量的赋值对程序运算结果没有任何作用。我们可以删除对这些变量赋值的代码。我们称之为删除无用赋值或删除无用代码。

- 代码外提

  对于循环中的有些代码,如果它产生的结果在循环中是不变的,就可以把它提到循环外来,以避免每循环一次都要对这条代码进行运算。

- 强度消弱

  将部分运算 / 操作用复杂度较低的运算 / 操作等价替换，以达到提升效率的目的。例如，图1中的内循环B3。每循环一次,j的值减1;T4的值始终与j保持着T4=4 * j的线性关系。每循环一次 ,T4的值减少4。因此,我们可以把循环中计算T4的值的乘法运算,变换为在循环前面进行一次循环乘法运算,而在循环进行减法运算。如图2所示(在图2中我们省略了B2, B5, B6的内容)。因为加减法运算一般要比乘除法快,所以称这种变换为强度削弱。

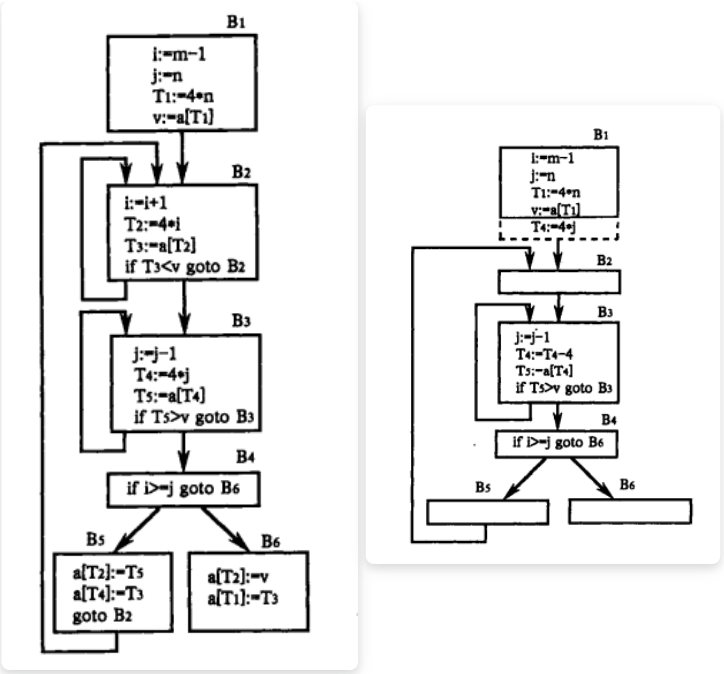


图1（左） 图2（右）

- 删除归纳变量

  在图1中我们看到,在B2中每循环一次,i增加1,T2的值与i保持着 T4=4 * i 的线性关系;而在B3,中每循环一次j减少1,T4与j保持着T4=4 * j的线性关系。这种变量我们称之为归纳变量。对T2: =4 * i和T4: =4 * j进行强度削弱后,i和j除了在条件判断if i >= j goto B6之外,其它地方不再被引用。因此,我们可以像图3一样把条件判断变换为if T2 >= T4 goto B6。

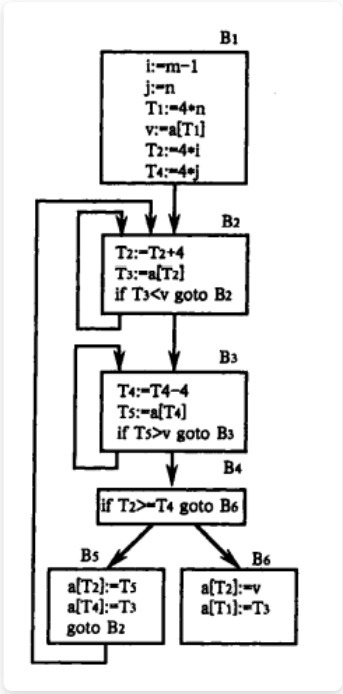


图3

10.2 局部优化

10.2.1 基本块及流图 - 赵雨晗

基本块：程序中一顺序执行的语句序列，只有一个入口和一个出口，入口是第一条语句，出口时最后一条语句。对于一个程序在各个基本块内进行优化则称为**基本块内的优化**或**局部优化。**

划分基本块的算法

1. 求出出口语句，分别是
    1. 程序的第一个语句，或者
    2. 能够由条件转移语句或无条件转移语句转移到的语句，或者
    3. 能够紧跟在条件转移语句后面的语句
2. 对求出的每一入口语句，构造其所属的基本快，它是由该入口语句到另一入口语句（不包括），或到一转移语句（包括在内），或到一停语句（包括在内）之间的语句序列组成的
3. 未纳入任一一基本块的语句是不会到达的，可以直接删除

示例在流图部分

局部优化方法

除了10.1中提到的**删除公共子表达式**和**删除无用赋值**这两种优化外，还有以下几种方式：

1. 合并已知量，如

$$
\begin{aligned}
&T_1 := 2\\
&\cdots\\
&T_2 := 4 * T_1
\end{aligned}
$$

   T_2 是常量可直接变为 $T_2 = 8$
2. 临时变量改名，可以把一个基本块中的临时变量改成定义的新的临时变量，这样在某些时候可以简化运算。
3. 交换语句的位置，如果两个语句互不相关，通过交换语句顺序可能产出更高效的代码
4. 代数变换，如

$$
\begin{aligned}
&x := x + 0\\
&x := x * 1
\end{aligned}
$$

   都没有改变 x 的值，可直接删除，又如

$$x := y * * 2$$

   为乘方运算，通常需要使用函数实现，可以直接改为

$$x := y * y$$

   用代数上等价的形式，用简单的运算替换。

流图

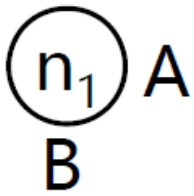
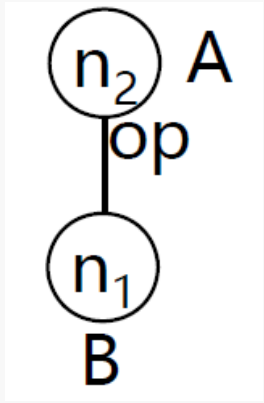
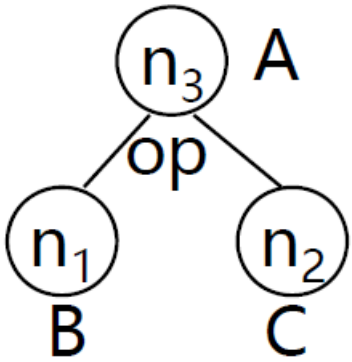
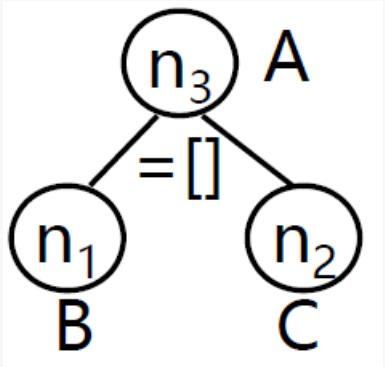
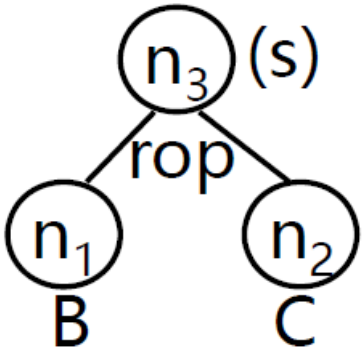
通过构造一个有向图，称之为**流图**，我们可以将控制流的信息增加到基本块的集合上来表示一个程序，每个流图以基本块为节点，如果一个节点的基本快的入口程序是程序的第一条语句，则称此节点为首结点，如果在某个执行顺序中，基本块 B_2 紧接在基本块 B_1 后执行，则从 B_1 到 B_2 有一条有向边。如

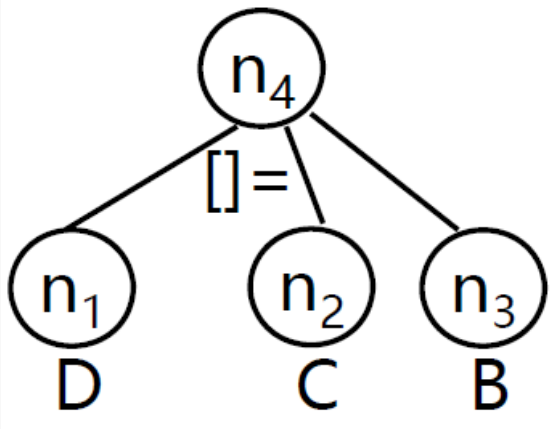
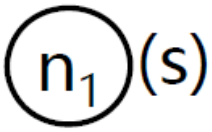
10.2.2 基本块的DAG表示及其应用 - 王晨旭

基本块的DAG图表示

- 图的叶结点以一标识符或常数作为标记，表示该结点代表该变量或常数的值。
    - 特别地，如果叶节点用来代表某变量A的地址，则用addr(A)作为该节点的标记。
    - 通常把叶节点上作为标记的标识符加上下标0，以表示他是该变量的初值。如果叶子节点上的标识符是某句代码的左值，则应该要将叶节点的标识符加上下标0，以进行区分。
- 图的内部结点以一运算符作为标记，表示该结点代表应用该运算符对其后继结点所代表的值进行运算的结果。
- 各个结点上可能附加一个或多个标识符 (称附加标识符) 表示这些变量具有该结点所代表的值。

如何绘制DAG图?

代码（四元式）	DAG图节点
0型: A:=B (:=，B，-，A)	
1型: A:=op B (op，B，-，A)	
2型: A:=B op C (op，B，C，A)	
2型: A:=B[C] (=[]，B，C，A)	
2型: if B rop C goto (s) (jrop，B，C，(s))	

代码（四元式）	DAG图节点
3型: D[C]:=B ([]=，B，D，C)	
0型: goto (s) (j，-，-，(s))	

课本算法

- 一个基本块，可用一个DAG来表示
- 对基本块中每一条四元式代码，依次构造对应的 DAG 图，最后基本块中所有四元式构造出来 DAG 连成整个基本块的 DAG
- 步骤：
  - 1.准备操作数的结点
    如果 NODE(B) 无定义，则构造一标记为 B 的叶结点并定义 NODE(B) 为这个结点;
    如果当前四元式是 0 型，则记 NODE(B) 的值为 n，转 4。
    如果当前四元式是 1 型，则转 2(1)
    如果当前四元式是 2 型，则 (i) 如果 NODE(C) 无定义，则构造一标记为 C 的叶结点并定义 NODE(C)为这个结点；(ii) 转2(2)
  - 2.合并已知量
    (1) 如果 NODE(B) 是标记为常数的叶结点，则转 2(3)；否则，转 3(1)
    (2) 如果 NODE(B) 和 NODE(C) 都是标记为常数的叶结点，则转2(4)；否则，转3(2)
    (3) 执行 op B (即合并已知量)。令得到的新常数为 P。如果 NODE(B) 是处理当前四元式时新构造出来的结点，则删除它。如果 NODE(P) 无定义，则构造一用P作标记的叶结点 n。置 NODE(P)=n，转4
    (4)执行 B op C (即合并已知量)。令得到的新常数为P。如果 NODE(B) 或 NODE(C) 是处理当前四元式时新构造出来的结点，则删除它。如果 NODE(P) 无定义，则构造一用P作标记的叶结点n。置 NODE(P)=n，转4
  - 3.删除公共子表达式
    (1) 检查 DAG 中是否已有一结点，其唯一后继为 NODE(B) 且标记为 op(即公共子表达式)。如果没有，则构造该结点 n，否则，把已有的结点作为它的结点并设该结点为 n。转4。
    (2) 检查 DAG中是否已有一结点，其左后继为 NODE(B)，右后继为 NODE(C)，且标记为 op(即公共子表达式)。如果没有，则构造该结点n，否则，把已有的结点作为它的结点并设该结点为 n。转4。
  - 4.删除无用赋值
    如果 NODE(A) 无定义，则把 A 附加在结点 n 上并令 NODE(A)=n; 否则，先把 A 从 NODE(A) 结点上的附加标识符集中删除(注意，如果 NODE(A) 是叶结点，则其 A 标记不删除)。把 A 附加到新结点 n 上并置 NODE(A)=n。转处理下一四元式。

特别地

1. 对任何一个代码，如果其中参与运算的对象都是编译时的已知量，那么，算法的步骤(2)并不生成计算该结点值的内部结点，而是执行该运算，用计算出的常数生成一个叶结点。所以步骤(2)的作用是实现合并已知量。
2. 如果某变量被赋值后，在它被引用前又重新赋值，那么，算法的步骤(4)已把该变量从具有前一个值的结点上删除，也即算法的步骤(4)具有删除前述第二种情况无用赋值的作用。
3. 算法的步骤3的作用是检查公共子表达式，对具有公共子表达式的所有代码，它只产生一个计算该表达式值的内部结点,而把那些被赋值的变量标识符附加到该结点上。

以课本例题为例

例 10.4　试构造以下基本块 G 的 DAG

(1) $T_0 := 3.14$

(2) $T_1 := 2 * T_0$

(3) $T_2 := R + r$

(4) $A := T_1 * T_2$

(5) $B := A$

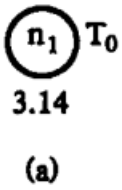
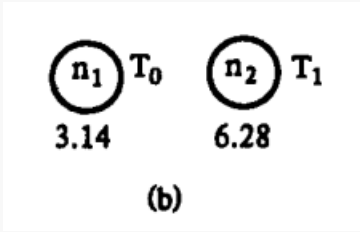
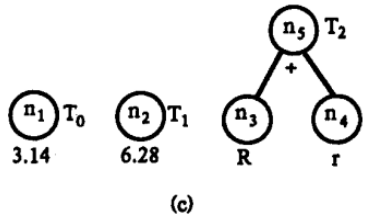
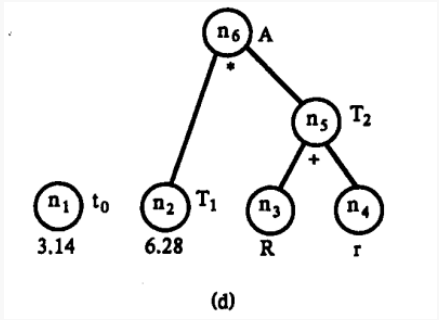
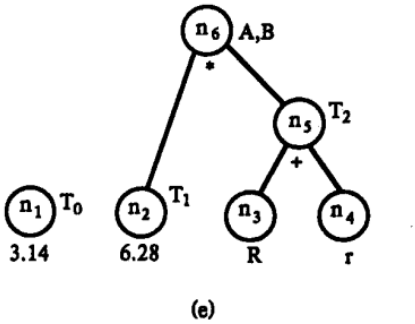
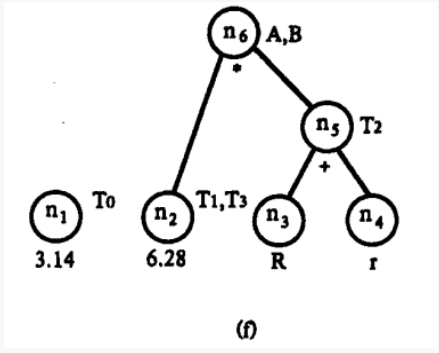
(6) $T_3 := 2 * T_0$

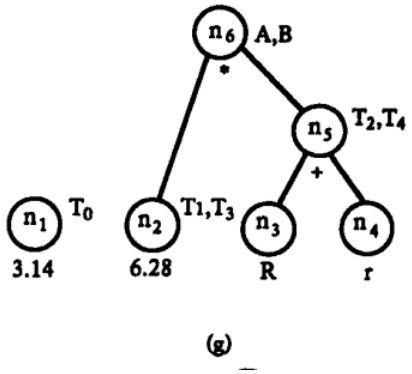
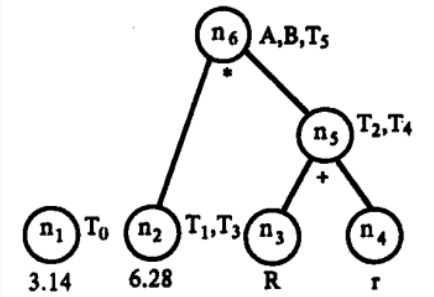
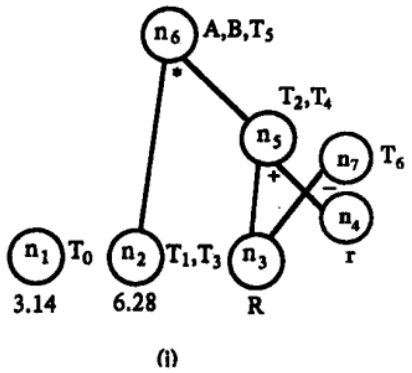
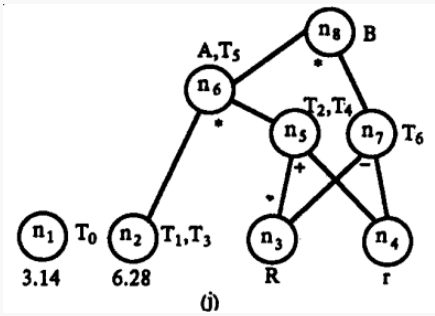
(7) $T_4 := R + r$

(8) $T_5 := T_3 * T_4$

(9) $T_6 := R - r$

(10) $B := T_5 * T_6$

代码	当前DAG图	解释
(1) $T_0 := 3.14$		0型代码， 形成的DAG图节点如左图所示， 只有一个节点。
(2) $T_1 := 2 * T_0$		由于2和T_0（参与运算的对象都是编译时的已知量）， 因此这里直接计算生成新的已知量， 生成一个叶节点， 而不是像普通2型式那样生成3个节点的DAG图。 **实现了合并已知量。**
(3) $T_2 := R + r$		按照普通2型式的DAG图生成方式生成。
(4) $A := T_1 * T_2$		按照普通2型式的DAG图生成方式生成。
(5) $B := A$		按照普通0型式生成DAG图， 由于A对应的节点已经生成， 这里直接将B加到A的右侧， 表示两者共享同一个节点。
(6) $T_3 := 2 * T_0$		同（2）， 进行**合并已知量**， 又由于该已知量对应的节点已经存在， 则直接将T_3加到该节点右侧。

代码	当前DAG图	解释
(7) $T_4 := R + r$	 (g)	如果按照普通2型式的DAG图生成方式生成， 则需要生成新的节点n_7， 并分别连接n_2和n_5； 但是由于该节点的表达式已经存在（**公共子表达式**）， 所以直接将左值T_4加到已存在节点n_5上。
(8) $T_5 := T_3 * T_4$	 (h)	同（7）， 由于计算$n_2 * n_5$的节点已经存在， 所以进行删除公共子表达式， 直接将左值T_5加到节点n_6。
(9) $T_6 := R - r$	 (i)	按照普通2型式的DAG图生成方式生成。
(10) $B := T_5 * T_6$	 (j)	按照普通2型式的DAG图生成方式生成n_8， 但会发现，n_6和n_8均有标识符B， 并且满足：某变量被复制后在他被引用前又重新赋值。 这种情况叫**无用赋值**，需要进行删除。 删除方法就是去掉赋值时间早的n_6上的B， 在赋值时间晚的n_8上添加B。

关于无用赋值：什么时候对标识符进行删除？

某变量被复制后在他被引用前又重新赋值，需要删除前赋值节点上的该变量。

如果该变量在引用之后被重新赋值呢？不允许DAG图中相同变量出现在两个不同的节点上。

- 如果该变量是来自基本块之前，也即在叶子节点处被引用（作为操作数），则后续如果要对其进行赋值，则需要之前的叶子节点出的变量添加下标0，已进行区分。
- 如果该变量是在本基本块中产生的，如：

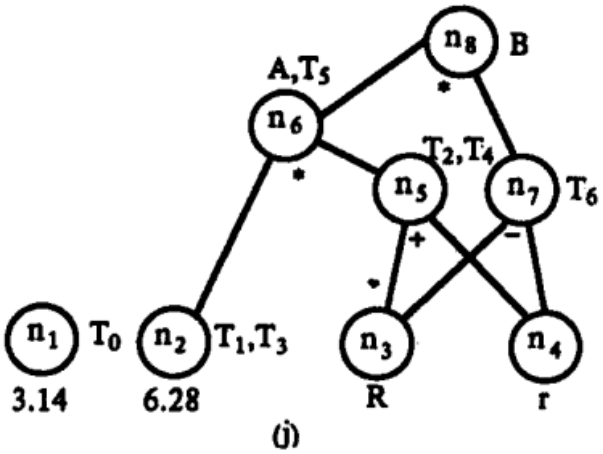
$$
\begin{aligned}
T_1 &:= a + b; \\
T_2 &:= a + T_1 \\
T_1 &:= b + T_2
\end{aligned}
$$

则将前一个赋值的T_1重命名（如命名为S），而将T_1名称的使用权交给后一个赋值。这样相当于代码变为：

$$
\begin{aligned}
S &:= a + b; \\
T_2 &:= a + S \\
T_1 &:= b + T_2
\end{aligned}
$$

从DAG图转换为优化后的基本块代码

以课本例子为例：



按照构造DAG图节点的顺序，重新写成中间代码G'：

$$
\begin{aligned}
(1) \quad T_0 &:= 3.14 \\
(2) \quad T_1 &:= 6.28 \\
(3) \quad T_3 &:= 6.28 \\
(4) \quad T_2 &:= R + r \\
(5) \quad T_4 &:= T_2 \\
(6) \quad A &:= 6.28 * T_2 \\
(7) \quad T_5 &:= A \\
(8) \quad T_6 &:= R - r \\
(9) \quad B &:= A * T_6
\end{aligned}
$$

注：当某个节点上附加了多个标识符时，使用第一个标识符进行计算或被引用，而其他表示符则直接使用代码：$T_{other} := T_{first}$来赋值。

考虑基本块内有关变量在基本块后面被引用的情况，进行优化

以课本相关例题为例：

假设上面的例子中，$T_0, T_1, T_2, T_3, T_4, T_5, T_6$在基本块后面都不会被引用。

从后往前检查优化后的基本块G'：

$$
\begin{array}{lll}
(1) & T_0 := 3.14 & T_0\text{不引用，删除} \\
(2) & T_1 := 6.28 & T_1\text{不引用，删除} \\
(3) & T_3 := 6.28 & T_3\text{不引用，删除} \\
(4) & T_2 := R + r & \text{保留} \\
(5) & T_4 := T_2 & T_4\text{不引用，删除} \\
(6) & A := 6.28 * T_2 & A\text{活跃} -> T_1\text{保留} \\
(7) & T_5 := A & T_5\text{不引用，删除} \\
(8) & T_6 := R - r & \text{保留} \\
(9) & B := A * T_6 & B\text{活跃} -> T_6\text{保留}
\end{array}
$$

重写为：

$$
\begin{array}{ll}
(1) & T_2 := R + r \\
(2) & A := 6.28 * T_2 \\
(3) & T_6 := R - r \\
(4) & B := A * T_6
\end{array}
$$

也可以像课本那样，对T_2和T_6重新命名：

$$
\begin{array}{l}
(1)\ S_1 := R + r \\
(2)\ A := 6.28 * S_1 \\
(3)\ S_2 := R - r \\
(4)\ B := A * S_2
\end{array}
$$

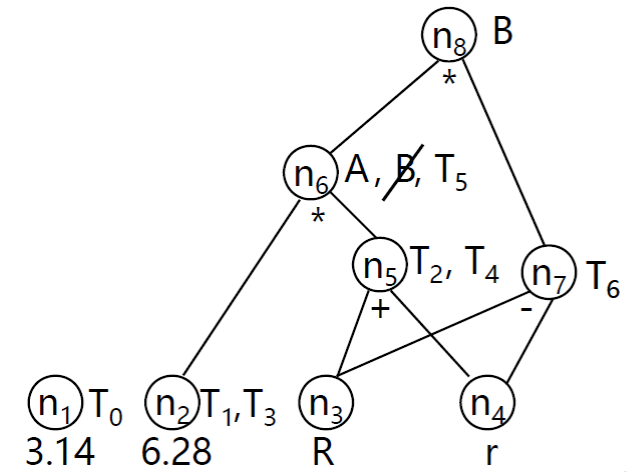
综上，整个过程为：

(1) T_0:=3.14
(2) T_1:=2*T_0
(3) T_2:=R+r
(4) A:=T_1*T_2
(5) B:=A
(6) T_3:=2*T_0
(7) T_4:=R+r
(8) T_5:=T_3*T_4
(9) T_6:=R-r
(10) B:=T_5*T_6

(1) T_0:=3.14
(2) T_1:=6.28
(3) T_3:=6.28
(4) T_2:=R+r
(5) T_4:=T_2
(6) A:=6.28*T_2
(7) T_5:=A
(8) T_6:=R-r
(9) B:=A*T_6

若只有A和B是出基本块之后活跃的

(1) T_2:=R+r
(2) A:=6.28*T_2
(3) T_6:=R-r
(4) B:=A*T_6



从 DAG 中得到的优化信息：

- 在基本块外被定值并在基本块内被引用的所有标识符，就是作为叶子结点上标记的那些标识符
- 在基本块内被定值并且该值在基本块后面可以被引用的所有标识符，就是 DAG 各结点上的那些标记或者附加标识符

§11 目标代码生成

Part 1（施博凡）

任务

把分析、翻译、优化后的中间代码变换成目标代码

输入

- 源程序的中间表示，以及符号表中的信息
- 类型检查

输出

- **绝对指令代码**：能够立即执行的机器语言代码，所有地址已经定位；
- **可重新定位指令代码**：待装配的机器语言模块，执行时，由连接装配程序把它们和某些运行程序连接起来，转换成能执行的机器语言代码；
- **汇编指令代码**：需要经过汇编程序转换成可执行的机器语言代码；

解决的问题

- 如何充分利用计算机的指令系统的特点
- 如何充分利用计算机的寄存器，减少目标代码中访问存贮单元的次数
    - 在寄存器分配期间，为程序的某一点选择驻留在寄存器中的一组变量
    - 在随后的寄存器指派阶段，挑出变量将要驻留的具体寄存器

代码生成

要设计一个好的代码生成器，必须预先熟悉目标机器及其指令系统。以下采用一简单模型机为例，讲解代码生成过程。

目标机器模型

- 具有多个通用寄存器，可用作累加器和变址器
- 运算必须在某个寄存器中进行
- 含有四种类型的指令形式

类　型	指令形式	意义(设 op 是二目运算符)
直接地址型	op R_i,M	(R_i) op (M) $\Rightarrow R_i$
寄存器型	op R_i, R_j	(R_i) op $(R_j) \Rightarrow R_i$
变址型	op R_i, c(R_j)	(R_i) op $((R_j)+c) \Rightarrow R_i$
间接型	op R_i, * M	(R_i) op $((M)) \Rightarrow R_i$
	op R_i, * R_j	(R_i)op $((R_j)) \Rightarrow R_i$
	op R_i, * c(R_j)	(R_i)op $(((R_j)+c)) \Rightarrow$ Ri

其中包括部分常用指令说明如下：

指　令	意　　义	指　令	意　　义
LD R_i, B	把 B 单元的内容取到寄存器 R,即(B)$\Rightarrow R_i$。	J < X	如 CT = 0　　转 X 单元
ST R_i, B	把寄存器 R_i 的内容存到 B 单元,即$(R_i) \Rightarrow$B。	J≤X	如 CT = 0 或 CT = 1 转 X 单元
J　X	无条件转向 X 单元。	J = X	如 CT = 1　　转 X 单元
CMP A, B	把 A 单元和 B 单元的值进行比较,并根据比较情况	J≠X	如 CT≠1　　转 X 单元
	把机器内部特征寄存器 CT 置成相应状态。CT 占两	J > X	如 CT = 2　　转 X 单元
	个二进位。根据 A < B 分别置 CT 为 0 或 1 或 2	J≥X	如 CT = 2 或 CT = 1 转 X 单元

一个简单代码生成器

已经获取中间代码，以基本块为单位生成目标代码，要在基本块的范围内考虑如何充分利用寄存器：

- 当生成计算某变量值的目标代码时，尽可能地让该变量的值保留在寄存器中，直到该寄存器必须用来存放别的变量值或者已到达基本块出口为止；
- 后续的目标代码尽可能地引用变量在寄存器中的值，而不访问主存.

不考虑代码效率，可以简单地把每条中间代码映射成若干条目标指令；但此种做法通常是很冗余的.

理想状况下，希望：

- 进入基本块时，所有寄存器空闲
- 离开基本块时，把存在寄存器中的现行的值存回主存中，释放所有寄存器
- 不特别说明，所有说明变量在基本块出口之后均为非活跃变量

因此，要尽可能做到：

- 在生成计算某变量值的目标代码时，尽可能让该变量保留在寄存器中
- 后续的目标代码尽可能引用变量在寄存器中的值，而不访问内存
- 在离开基本块时， 把存在寄存器中的现行的值放到主存中

为此，引入**待用信息**、**寄存器描述数组**和**变量地址描述数组**用以记录代码生成时所需收集的信息。

待用信息计算

待用信息：

如果在一个基本块内，四元式i对A定值，四元式j要引用A值，而从i到j之间没有A的其他定值， 那么，我们称j是四元式i的变量A的**待用信息**。

二元组(x, x)：

表示变量的待用信息和活跃信息。第1元：i表示待用信息， ^表示非待用；第2元：y表示活跃，^表示非活跃。

(x，x)→(x，x)：

表示待用信息和活跃信息的变化，用后者更新前者。

待用信息表生成算法：

（1）开始时，把基本块中各变址的符号表登记项中的待用信息栏填为“非待用”，并根据该变量在基本块出口之后是不是活跃的，把其中的活跃信息栏填为”活跃”或“非活跃 "；

（2）从基本块出口到基本块入口由后向前依次处理各个中间代码。对每一中间代码i：*A:=B op C*，依次执行下述步骤：

①把符号表中变量A的待用信息和活跃信息附加到中间代码i上；

②把符号表中A的待用信息和活跃信息分别置为“非待用”和“非活跃”；

③把符号表中变量B和C的待用信息和活跃信息附加到中间代码i 上；

④把符号表中B和C的待用信息均置为i，活跃信息均置为“活跃"。

过程示例

有基本块：

1. T:=A-B
2. U:=A-C
3. V:=T+U
4. W:=V+U

其中，W为基本块出口之后的活动变量；

初始化变量状态表和待用信息表：

变量名	初始状态→信息链
T	(^,^)
A	(^,^)
B	(^,^)
C	(^,^)
U	(^,^)
V	(^,^)
W	(^,y)

序号	四元式	左值	左操作数	右操作数

处理完4号四元式：

变量名	初始状态→信息链
T	(^,^)
A	(^,^)
B	(^,^)
C	(^,^)
U	(^,^) → (4,y)
V	(^,^) → (4,y)
W	(^,y) → (^,^)

序号	四元式	左值	左操作数	右操作数
(4)	W:=V+U	(^,y)	(^,^)	(^,^)

处理完3号式：

变量名	初始状态→信息链
T	(^,^) → (3,y)
A	(^,^)
B	(^,^)
C	(^,^)
U	(^,^) → (4,y) → (3,y)
V	(^,^) → (4,y) → (^,^)
W	(^,y) → (^,^)

序号	四元式	左值	左操作数	右操作数
(3)	V:=T+U	(4,y)	(^,^)	(4,y)
(4)	W:=V+U	(^,y)	(^,^)	(^,^)

处理完2号式：

变量名	初始状态→信息链
T	(^,^) → (3,y)
A	(^,^) → (2,y)
B	(^,^)
C	(^,^) → (2,y)
U	(^,^) → (4,y) → (3,y) → (^,^)
V	(^,^) → (4,y) → (^,^)
W	(^,y) → (^,^)

序号	四元式	左值	左操作数	右操作数
(2)	U:=A-C	(3,y)	(^,^)	(^,^)
(3)	V:=T+U	(4,y)	(^,^)	(4,y)
(4)	W:=V+U	(^,y)	(^,^)	(^,^)

处理完1号式：

变量名	初始状态→信息链
T	(^,^) → (3,y) → (^,^)
A	(^,^) → (2,y) → (1,y)
B	(^,^) → (1,y)
C	(^,^) → (2,y)
U	(^,^) → (4,y) → (3,y) → (^,^)
V	(^,^) → (4,y) → (^,^)
W	(^,y) → (^,^)

序号	四元式	左值	左操作数	右操作数
(1)	T:=A-B	(3,y)	(2,y)	(^,^)
(2)	U:=A-C	(3,y)	(^,^)	(^,^)
(3)	V:=T+U	(4,y)	(^,^)	(4,y)
(4)	W:=V+U	(^,y)	(^,^)	(^,^)

以上，完成了待用信息表的构造。

Part 2（王志睿）

寄存器描述和地址描述

前言

回顾本章节Part 1部分的内容，要在一个基本块的范围内考虑充分利用寄存器需要知道：

- [x] 四元式指令：每条指令中各变量在将来会被使用的情况
- [ ] 变量：每个变量现行值的存放位置
- [ ] 寄存器：每个寄存器当前的使用状况

其中***“每条指令中各变量在将来会被使用的情况”***已通过上节中所介绍的***“计算待用信息和活跃信息的算法”***解决，本节则针对***后两点***，提出解决方案。

变量地址描述

建立一个**变量地址描述数组AVALUE**来动态记录各变量现行值的存放位置。

> 例如 AVALUE[A]={R1, R2, A}，表示变量A的现行值存放在R1, R2以及内存单元A中

寄存器描述

建立一个编译用的**寄存器描述数组RVALUE**来动态记录各寄存器的使用信息。

> 例如 RVALUE[R]={A,B}，表示寄存器R正被变量A,B所使用

补充说明

- 对于四元式A:=B，如果B的现行值在某寄存器Ri中，则：
  - 无需生成目标代码
  - 只需在RVALUE(Ri)中增加一个A（即把Ri同时分配给B和A）
  - 并把AVALUE(A)改为Ri
- 上述处理思路体现了**尽可能引用变量在寄存器中的值而不访问内存**的优化思想。

简单代码生成算法

主程序框架

程序功能

- 输入：中间代码序列
- 输出：目标代码序列

流程描述

对每个四元式: i: A:=B op C，依次执行：

1. 以四元式: i: A:=B op C 为参数，调用函数过程GETREG(i: A:=B op C)，返回一个寄存器R，用作存放A的寄存器；
2. 利用AVALUE[B]和AVALUE[C]，确定B和C现行值的存放位置B'和C'（如果其现行值在寄存器中，则把寄存器取作B'和C'）；
3. 如果B'≠R，则生成目标代码：

   LD R, B'
   op R, C'

   否则生成目标代码：

   op R, C'

   如果B'或C'为R，则删除AVALUE[B]或AVALUE[C]中的R。
4. 令AVALUE[A]={R}, RVALUE[R]={A}。
5. 若B或C的现行值在基本块中不再被引用，也不是基本块出口之后的活跃变量，且其现行值在某寄存器Rk中，则删除RVALUE[Rk]中的 B或C以及AVALUE[B]或AVALUE[C] 中的Rk，使得该寄存器不再为B或C占用。

流程解析

第一步：调用GETREG(i:A:=B op C)，为A分配寄存器；

第二步：根据变量地址描述数组AVALUE寻址操作数B和C，若现行值存放在寄存器中则优先访问寄存器；*（体现了"尽可能用：后续的目标代码尽可能引用变量在寄存器中的值，而不访问内存"）*

第三步：生成目标代码并修改操作数的变量地址描述数组；

第四步：修改与A相关的变量地址描述与寄存器描述；

第五步：为了方便后续的寄存器分配过程，在这一步中根据待用信息和活跃信息，释放将来不会被引用到的变量所占用的寄存器。

伪码表示

```
// Intermediate_Code_List :  中间代码序列
function CODE_GENERATION(Intermediate_Code_List){
    for each i:A:=B op C in Intermediate_Code_List do
        // Step 1
        R = GETREG(i:A:=B op C);
        // Step 2
        for address in AVALUE[B] do
            B' = address;
            if address is Register then
                break;
        for address in AVALUE[C] do
            C' = address;
            if address is Register then
                break;
        // Step 3
        if B' != R then
            GEN(LD R, B');
            GEN(op R, C');
        else
            GEN(op R, C');
```

```
        if B' == R then
            delete R from AVALUE[B];
        if C' == R then
            delete R from AVALUE[C];
        // Step 4
        AVALUE[A] = {R};
        RVALUE[R] = {A};
        // Step 5
        for Rk in Registers do
            if B in RVALUE[Rk] and
            B will not be used in the future then
                delete B from Rk
            if C in RVALUE[Rk] and
            C will not be used in the future then
                delete C from Rk
}
```

寄存器分配算法分程序

程序功能

- 输入：四元式：i:A:=B op C
- 输出：返回一个用来存放A的值的寄存器

算法思想

- 尽可能用左操作数独占的寄存器
- 尽可能用空闲寄存器
- 抢占非空闲寄存器

流程描述

1. 如果B的现行值在某个寄存器 Ri 中，RVALUE[Ri] 中只包含 B，此外，若满足以下条件之一：
    - B 与 A 是同一个标识符；
    - B 的现行值在执行四元式 A:=B op C 之后不会再引用；

    则选取Ri为所需要的寄存器 R，并转4；
2. 如果有尚未分配的寄存器，则从中选取一个 Ri 为所需要的寄存器 R，并转4；
3. 从已分配的寄存器中选取一个 Ri 为所需要的寄存器 R，最好使得 Ri 满足以下条件之一：
    - 占用 Ri 的变量的值也同时存放在该变量的贮存单元中，
    - 在基本块中要在最远的将来才会引用到或不会引用到。

    为 Ri 中的变量生成必要的存数指令。
4. 给出 R，返回。

流程解析

第一步（”尽可能用 B 独占的寄存器“）：若B与A是同一个标识符，则直接使用该寄存器即可（体现了”尽可能留：在生成计算某变量值的目标代码时，尽可能让该变量保留在寄存器中“），若 B 之后不会被引用到，则继续存放该值对后续处理没有任何意义，直接抢占该寄存器分配给 A 即可；

第二步（”尽可能用空闲寄存器“）：分配一个空闲的寄存器给 A；

第三步（”抢占非空闲寄存器“）：以上两种情况均不满足，则考虑抢占非空闲寄存器，若该寄存器中的变量的现行值同时存放在该变量的贮存单元中或之后不会被引用到，则直接抢占即可，否则还要为其生成必要的存数指令写回内存（因为后面还要引用到该现行值）；

第四步：将分配到的寄存器 R 返回给代码生成主程序。

伪码表示

```
function GETREG(i:A:=B op C){
    // Step 1
    for Rk in Registers do
        if RVALUE[Rk] == {B} and
        (B == A or B will not be used in the future) then
            return Rk;
    // Step 2
    for Rk in Registers do
        if RVALUE[Rk] == {} then
            return Rk;
    // Step 3
    Ri = NULL;
    for Rk in Registers do
        if M in RVALUE[Rk] also in AVALUE[M]
           or M  will not be used in the future then
                Ri = Rk;
                break;
    if Ri == NULL then
        Ri = Rk in Registers which is closest to the condition above;
    STORE_VARIABLE(M, Rk);
    return Rk;
}
```

生成存数指令分程序

程序功能

把被抢占寄存器且后续还会被引用到的变量的现行值写回内存。

流程描述

对 RVALUE[Ri] 中每一变量 M，如果 M 不是 A，或者如果 M 是 A 又是 C，但不是 B 并且 B 也不在 RVALUE[Ri] 中，则

1. 如果 AVALUE[M] 不包含 M，则生成目标代码 ST Ri, M
2. 如果 M 是 B，或者 M 是 C 但同时 B 也在 RVALUE[Ri] 中，则令 AVALUE[M]={M， Ri}，否则令 AVALUE[M]={M}
3. 删除 RVALUE[Ri] 中的 M

流程解析

第一步：M 的现行值只存在于寄存器中，则为了后续引用，需要将其现行值写回内存；

第二步：令 AVALUE[M]={M} 的分支操作很好理解，但是

> **另一个分支中将AVALUE[M]置为{M，Ri}，为什么将Ri依旧留在AVALUE[M]中呢?**

> 原因是在代码生成主程序中对于变量的引用是根据AVALUE数组来寻址的，这里将Ri留在AVALUE[M]中是为了让主程序尽量引用寄存器中的值，避免访问主存，从而产生优化的效果。

第三步：更新RVALUE数组；

伪码表示

```
function STORE_VARIABLE(M, Ri){
    for M in RVALUE[Ri] do
        if M != A or (M == A and M == C
            and M != B and B not in RVALUE[Ri]) then
            if M not in AVALUE[M] then
                GEN(ST Ri, M)
            if M == B or (M == C and B in RVALUE[Ri]) then
                AVALUE[M] = {M, Ri};
            else
                AVALUE[M] = {M};
            delete M from RVALUE[Ri];
}
```

简单代码生成算法模拟

> 例：基本块
>
> 1. T:=A-B
> 2. U:=A-C
> 3. V:=T+U
> 4. W:=V+U
>
> 设W是基本块出口之后的活跃变量，只有R0和R1是可用寄存器，请利用简单代码生成算法生成目标代码。

Step 1 计算待用信息和活跃信息

已在 Part 1 完成计算，这里就不再赘述，直接给出结果。

序号	四元式	左值	左操作数	右操作数
(1)	T:=A - B	(3, y)	(2, y)	(^,^)
(2)	U:=A -C	(3, y)	(^, ^)	(^,^)
(3)	V:= T + U	(4, y)	(^, ^)	(4, y)
(4)	W:= V + U	(^, y)	(^, ^)	(^,^)

Step 2 简单代码生成算法

中间代码	目标代码	RVALUE	AVALUE
T:=A - B	LD R0, A SUB R0, B	R0含有T	T在R0中
U:=A -C	LD R1, A SUB R1, C	R0含有T R1含有U	T在R0中 U在R1中
V:= T + U	ADD R0, R1	R0含有V R1含有U	V在R0中 U在R1中
W:= V + U	ADD R0, R1	R0含有W	W在R0中
	ST R0, W		

过程解析

1. T:=A - B，此时两个寄存器都空闲，选取 R0 作为存放 T 的寄存器，生成目标代码后，更新 RVALUE 和 AVALUE；
2. U:=A - C，左操作数 A 不在寄存器中，因此选取空闲的 R1 作为存放 U 的寄存器，生成目标代码后，更新 RVALUE 和 AVALUE；
3. V:=T + U，左操作数 T 在 R0 中且 T 在之后不会再被引用，因此选取 R0 作为存放 V 的寄存器且不必将 R0 中的值写回 T 的贮存单元，生成目标代码后，删除AVALUE[T] 中的 R0 以及 RVALUE[R0] 中的 T，置 AVALUE[V]={R0}，RVALUE[R0]={V}；
4. W:=V+U，左操作数 V 在 R0 中且 V 在之后不会再被引用，因此选取 R0 作为存放 W 的寄存器且不必将 R0 中的值写回 V 的贮存单元，生成目标代码后，删除AVALUE[V] 中的 R0 以及 RVALUE[R0] 中的 V，置 AVALUE[V]={R0}，RVALUE[R0]={V}；又因为 U 不是基本块出口之后的活跃变量，所以删除 RVALUE[R1] 中的 U 以及 RVALUE[U] 中的 R1，释放寄存器；
5. 由于 W 在基本块出口之后是活跃变量，所以还要生成一条目标代码 ST R0, W，将现行值写回 W 的贮存单元。

附图（各中间代码对应的目标代码）

序号	中间代码	目标代码	备　注
1	A: = B op C	LD R_i, B op R_i, C	(1)其中 R_i 是新分配给 A 的寄存器 (2)如果 B 和/或 C 的现行值在寄存器中,则目标中 B 和/或 C 用寄存器表示。但如果 C 的现行值在 R_i 中,而 B 的现行值不在 R_i 中,则 C 要用其主存单元表示 (3)如果 B 的现行值也在 R 中,则不生成第一条目标代码
2	A: = op_1 B	LD R_i, B op1 R_i, R_i	(1)同 1 中备注(1) (2)同 1 中备注(3) (3)op_1 指一目运算符
3	A: = B	LD R_i, B	(1)同 1 中备注(1) (2)如果 B 的现行值在某寄存器 R_i 中,则如前所述,不生成目标代码
4	A: = B[I]	LD R_j, I LD R_i, B(R_j)	(1)同 1 中备注(1) (2)如果 I 的现行值在某寄存器 R_j 中,则第一条目标可省去,否则 R_i 是分配给 I 的寄存器

5	A[I]: = B	LD R_i, B
		LD R_j, I
		ST R_i, A(R_j)
			(1)同 1 中备注(3)
			(2)同 4 中备注(2)

6	goto X	J X′	(1)X′是标号为 X 的中间代码的目标代码的首地址

7	if A rop B goto X	LD R_i, A
		CMP R_i, B
		J rop X′
			(1)X′的意义同 6 中备注(1)
			(2)若 A 的现行值在寄存器 R_i 中,则第一条目标代码可省去
			(3)如果 B 的现行值在某寄存器 R_k 中,则目标代码中的 B 就是 R_k
			(4)rop 指 <、≤、=、≠、>或≥

8	A: = P↑	LD R_i, * P	(1)同 1 中备注(1)
9	P↑: = A	LD R_i, A ST R_i, * P	(1)同 1 中备注(1) (2)如果 A 的现行值原来在某寄存器 R_i 中,则不生成第一条目标代码

DAG目标代码优化

优化思想

利用基本块的DAG，给基本块的中间代码序列重新排序（计算表达式时采用从右往左算的计算次序），尽可能地减少访问内存的次数，以便生成较优的目标代码。

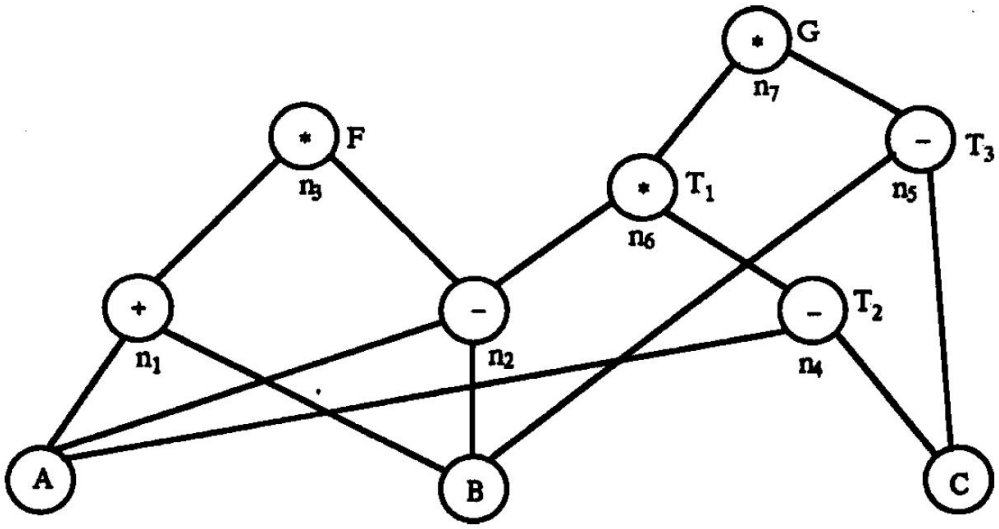
算法流程

设 DAG 有 N 个内部结点，T 是一个线性表，它共有 N 个登记项，算法的步骤如下：

```
置初值：
FOR k:  = 1 TO N DO T[k]: = null;
i:= N;
WHILE 存在未列入T 的内部结点 DO
BEGIN
    选取一个未列人 T 但其全部父结（即前驱）均已列入T 或者没有父结的内部结点n;
    T[i]:= n;
    i:= i - 1;
    WHILE n的最左子结m不为叶结且其全部父结均已列入T中 DO
    BEGIN
        T[i]: =m;
        i: =i-1;
        n: =m
    END
END;
最后T[1], T[2],...,T[N]即为所求的结点顺序。
```

算法模拟

> 例题　考察下面中间代码序列G1:
>
> T1 : = A + B
>
> T2 : = A - B
>
> F := T1 * T2
>
> T1 := A - B
>
> T2 := A - C
>
> T3 := B - C
>
> T1 := T1 * T2
>
> G := T1 * T3
>
> 其对应的DAG如下图所示：

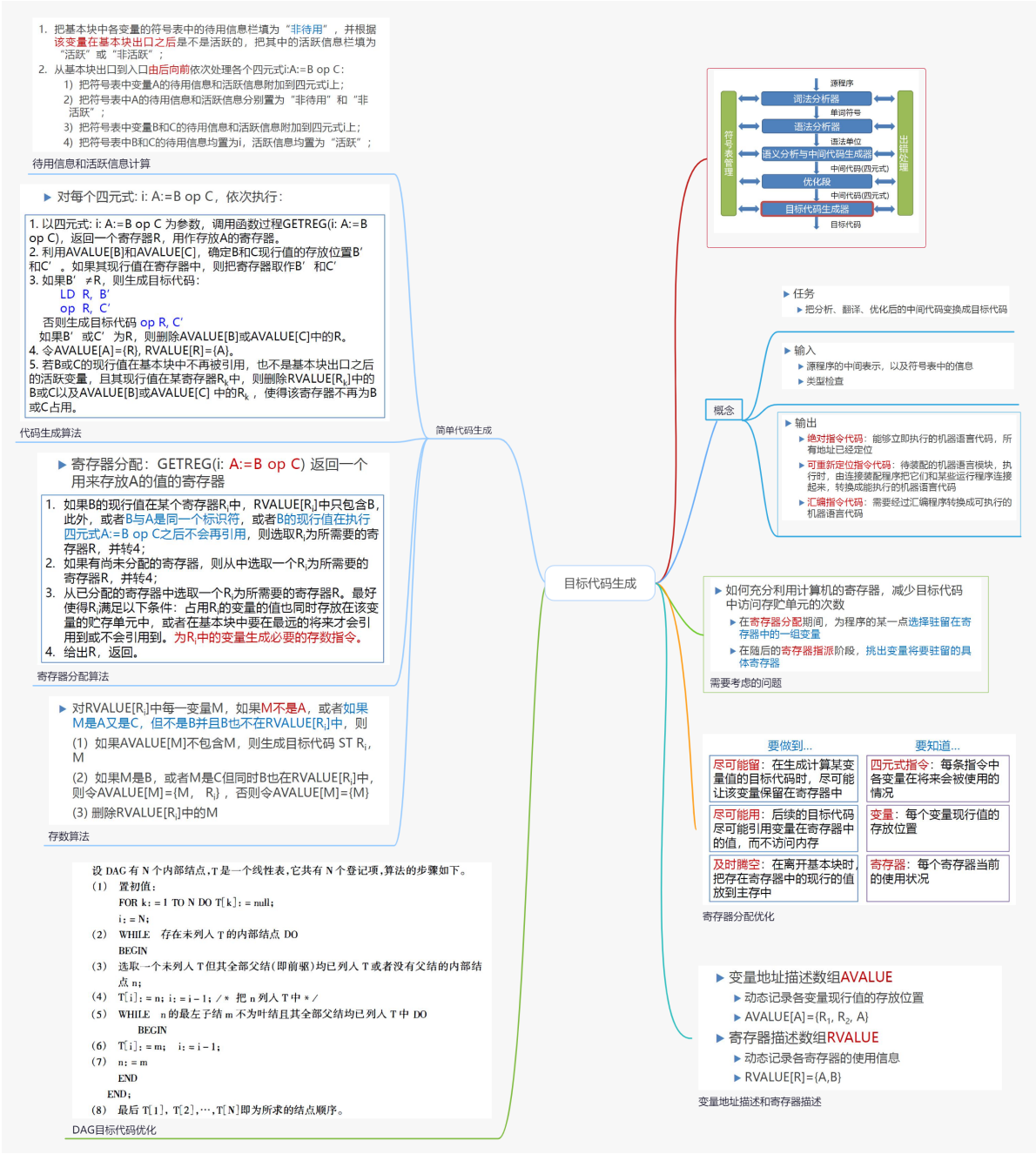


请应用前述算法对该DAG进行排序。

模拟步骤如下：

- 置初值：i = 7，T 的所有元素全为 null;
- 内结n3和n7均满足第 3 步的要求，假定选取T[7]为n3；
- 结点n3的最左子结n1满足第5步的要求，因此，按第6步，T[6] =n1;
- 但n1的最左子结A为叶结，不满足第5步的要求。现在只有n7满足第3步的要求，于是T[5] = n7;
- 结点n7的最左子结n6满足第5步的条件，因此，T[4] = n6;
- 结点n6的最左子结n2同样满足第5步的要求，因此，T[3] = n2;
- 目前，满足第3步要求的结点尚有n4和n5 ，假定选取 T[2] = n4;
- 当最后把n5列入T[1]后，算法工作结束;
- 至此，我们所求出内结点顺序为：**n5, n4, n2, n6, n7, n1, n3**

章节总结



典型例题（王志睿、施博凡）

例题一

1. 对以下中间代码序列 G：

$T_1 := B - C$

$T_2 := A * T_1$

$T_3 := D + 1$

$T_4 := E - F$

$T_5 := T_3 * T_4$

$W := T_2 / T_5$

假设可用寄存器为 R_0 和 R_1，W 是基本块出口的活跃变量，用简单代码生成算法生成其目标代码，同时列出代码生成过程中的寄存器描述和地址描述。

答案解析

（1）计算待用信息和活跃信息

变量名	初始状态->信息链
A	(^,^)->(2,y)
B	(^,^)->(1,y)
C	(^,^)->(1,y)
D	(^,^)->(3,y)
E	(^,^)->(4,y)
F	(^,^)->(4,y)
T1	(^,^)->(2,y)->(^,^)
T2	(^,^)->(6,y)->(^,^)
T3	(^,^)->(5,y)->(^,^)
T4	(^,^)->(5,y)->(^,^)
T5	(^,^)->(6,y)->(^,^)
W	(^,y)->(^,^)

序号	四元式	左值	左操作数	右操作数
(1)	T1:=B-C	(2,y)	(^,^)	(^,^)
(2)	T2:=A*T1	(6,y)	(^,^)	(^,^)
(3)	T3:=D+1	(5,y)	(^,^)	(^,^)
(4)	T4:=E-F	(5,y)	(^,^)	(^,^)
(5)	T5:=T3*T4	(6,y)	(^,^)	(^,^)
(6)	W:=T2/T5	(^,y)	(^,^)	(^,^)

（2）代码生成

序号	中间代码	目标代码	RVALUE	AVALUE
(1)	T1:=B-C	LD R0, B SUB R0, C	RVALUE[R0]={T1}	AVALUE[T1]={R0}
(2)	T2:=A*T1	LD R1, A MUL R1, R0	RVALUE[R1]={T2}	AVALUE[T2]={R1}
(3)	T3:=D+1	LD R0, D ADD R0, 1	RVALUE[R1]={T2} RVALUE[R0]={T3}	AVALUE[T2]={R1} AVALUE[T3]={R0}

(4)	T4:=E-F	ST R1, T2
		LD R1, E
		SUB R1, F
			RVALUE[R1]={T4}
			RVALUE[R0]={T3}
				AVALUE[T2]={T2}
				AVALUE[T3]={R0}
				AVALUE[T4]={R1}

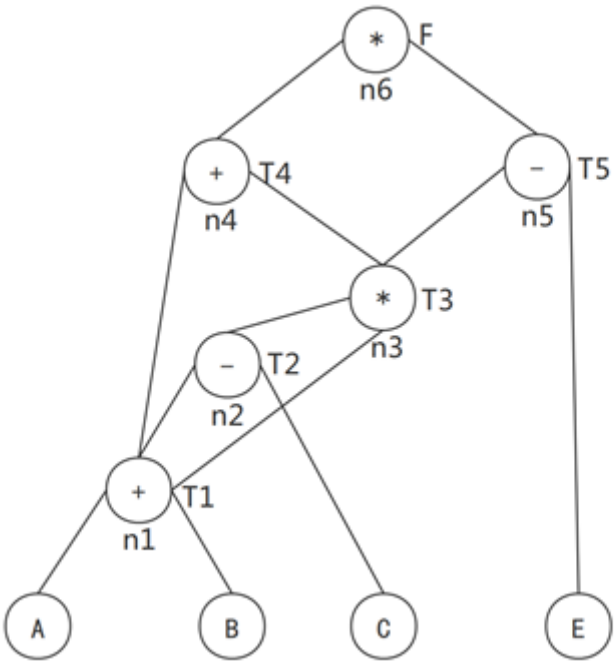
(5)	T5:=T3*T4	MUL R0, R1	RVALUE[R0]={T5}	AVALUE[T2]={T2} AVALUE[T5]={R0}
(6)	W:=T2/T5	LD R1, T2 DIV R1, R0	RVALUE[R1]={W}	AVALUE[T2]={T2} AVELUE[W]={R1}
		ST R1, W		

例题2

> 2. 对以下中间代码序列：
>
> $T_1 := A + B$
>
> $T_2 := T_1 - C$
>
> $T_3 := T_2 * T_1$
>
> $T_4 := T_1 + T_3$
>
> $T_5 := T_3 - E$
>
> $F := T_4 * T_5$
>
> （1）应用 DAG 结点排序算法重新排序；
>
> （2）假设可用寄存器为 R_0，F 是基本块出口处的活跃变量，应用简单代码生成算法分别生成排序前后的中间代码序列的目标代码，并比较其优劣。

答案解析

（1）中间代码序列对应的DAG如下：



应用DAG结点排序算法得顺序为：n1, n2, n3, n5, n4, n6

重新排序后的中间代码序列为：

T1:=A+B

T2:=T1-C

T3:=T2*T1

T5:=T3-E

T4:=T1+T3

F:=T4*T5

（2）生成目标代码

排序前：

① 计算待用信息和活跃信息

变量名	初始状态->信息链
T1	(^,^)->(4,y)->(3,y)->(2,y)->(^,^)
T2	(^,^)->(3,y)->(^,^)
T3	(^,^)->(5,y)->(4,y)->(^,^)
T4	(^,^)->(6,y)->(^,^)
T5	(^,^)->(6,y)->(^,^)
A	(^,^)->(1,y)
B	(^,^)->(1,y)
C	(^,^)->(2,y)
D	(^,^)
E	(^,^)->(5,y)
F	(^,y)->(^,^)

序号	四元式	左值	左操作数	右操作数
(1)	T1:=A+B	(2,y)	(^,^)	(^,^)
(2)	T2:=T1-C	(3,y)	(3,y)	(^,^)
(3)	T3:=T2*T1	(4,y)	(^,^)	(4,y)
(4)	T4:=T1+T3	(6,y)	(^,^)	(5,y)
(5)	T5:=T3-E	(6,y)	(^,^)	(^,^)
(6)	F:=T4*T5	(^,y)	(^,^)	(^,^)

② 代码生成

序号	中间代码	目标代码	RVALUE	AVALUE
(1)	T1:=A+B	LD R0, A ADD R0, B	RVALUE[R0]={T1}	AVALUE[T1]={R0}
(2)	T2:=T1-C	ST R0, T1 SUB R0, C	RVALUE[R0]={T2}	AVALUE[T1]={T1} AVALUE[T2]={R0}
(3)	T3:=T2*T1	MUL R0, T1	RVALUE[R0]={T3}	AVALUE[T1]={T1} AVALUE[T3]={R0}

(4)	T4:=T1+T3	ST R0, T3
		LD R0, T1
		ADD R0, T3
			RVALUE[R0]={T4}	AVALUE[T1]={T1}
				AVALUE[T3]={T3}
				AVALUE[T4]={R0}

(5)	T5:=T3-E	ST R0, T4
		LD R0, T3
		SUB R0, E
			RVALUE[R0]={T5}	AVALUE[T1]={T1}
				AVALUE[T3]={T3}
				AVALUE[T4]={T4}
				AVALUE[T5]={R0}

(6)	F:=T4*T5	ST R0, T5
		LD R0, T4
		MUL R0, T5
			RVALUE[R0]={F}	AVALUE[T1]={T1}
				AVALUE[T3]={T3}
				AVALUE[T4]={T4}
				AVALUE[T5]={T5}
				AVALUE[F]={R0}

		ST R0, F		

排序后：

① 计算待用信息和活跃信息

变量名	初始状态->信息链
T1	(^,^)->(5,y)->(3,y)->(2,y)->(^,^)
T2	(^,^)->(3,y)->(^,^)
T3	(^,^)->(5,y)->(4,y)->(^,^)
T4	(^,^)->(6,y)->(^,^)
T5	(^,^)->(6,y)->(^,^)
A	(^,^)->(1,y)
B	(^,^)->(1,y)
C	(^,^)->(2,y)
D	(^,^)
E	(^,^)->(4,y)
F	(^,y)->(^,^)

序号	四元式	左值	左操作数	右操作数
(1)	T1:=A+B	(2,y)	(^,^)	(^,^)
(2)	T2:=T1-C	(3,y)	(3,y)	(^,^)
(3)	T3:=T2*T1	(4,y)	(^,^)	(5,y)
(4)	T5:=T3-E	(6,y)	(5,y)	(^,^)
(5)	T4:=T1+T3	(6,y)	(^,^)	(^,^)
(6)	F:=T4*T5	(^,y)	(^,^)	(^,^)

② 代码生成

序号	中间代码	目标代码	RVALUE	AVALUE
(1)	T1:=A+B	LD R0, A ADD R0, B	RVALUE[R0]={T1}	AVALUE[T1]={R0}
(2)	T2:=T1-C	ST R0, T1 SUB R0, C	RVALUE[R0]={T2}	AVALUE[T1]={T1} AVALUE[T2]={R0}
(3)	T3:=T2*T1	MUL R0, T1	RVALUE[R0]={T3}	AVALUE[T1]={T1} AVALUE[T3]={R0}
(4)	T5:=T3-E	ST R0, T3 SUB R0, E	RVALUE[R0]={T5}	AVALUE[T1]={T1} AVALUE[T3]={T3} AVALUE[T5]={R0}

(5)	T4:=T1+T3	ST R0, T5
		LD R0, T1
		ADD R0, T3
			RVALUE[R0]={T4}	AVALUE[T1]={T1}
				AVALUE[T3]={T3}
				AVALUE[T5]={T5}
				AVALUE[T4]={R0}

(6)	F:=T4*T5	MUL R0, T5	RVALUE[R0]={F}	AVALUE[T1]={T1} AVALUE[T3]={T3} AVALUE[T5]={T5} AVALUE[F]={R0}
		ST R0, F		