

# 编译原理讲义

## (总论)

南京大学计算机系

赵建华

E-Mail: zhaojh@nju.edu.cn

HomePage: 讲义，实习题目都  
将在HomePage上。

# 课程目的

- 了解编译程序的实现原理和技术。
- 利用从本课程学习到的知识，增强编写和调试程序的能力。
- 在其它方面的应用：
  - 正文查找；
  - 正文处理；
  - 指令识别等。

# 课程内容

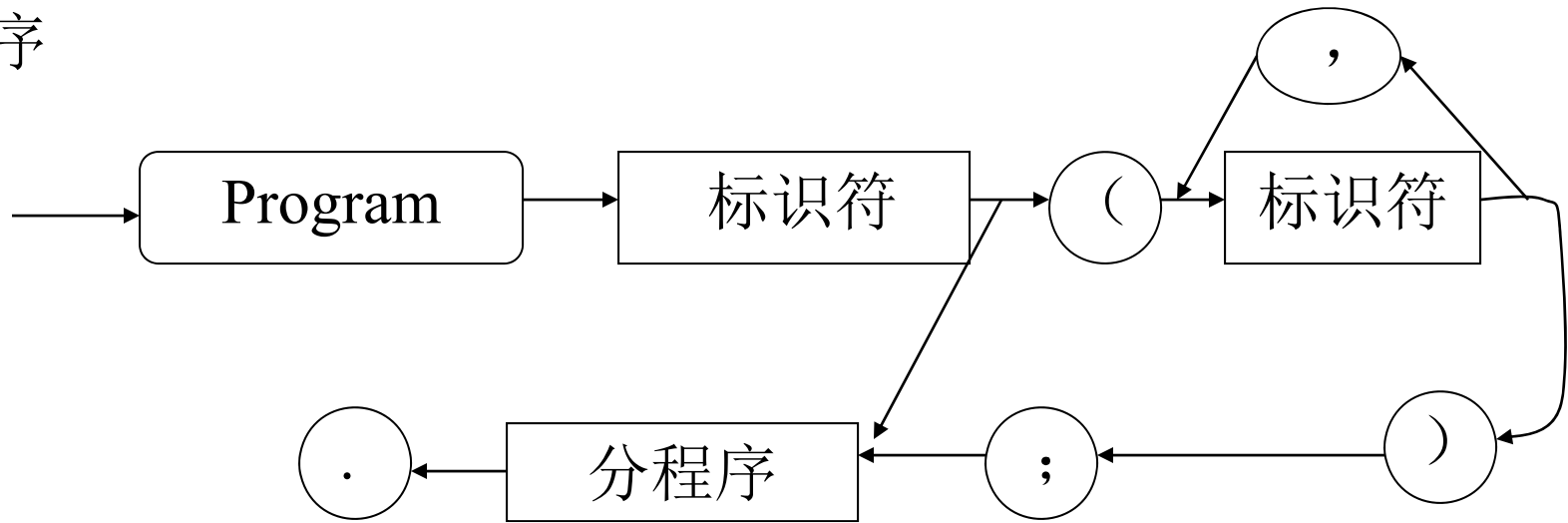
- 文法/语言/自动机
- 词法分析技术
- 语法分析技术
- 语义分析技术
- 代码生成和优化技术

# 程序设计语言的定义（语法）

- 语法：是指规定 *如何由基本符号组成一个完整的程序* 的规则。可以分为一般的语法规则和词法规则。
- 定义语法的方式
  - 语法图：直观，篇幅大。
  - BNF表示法：简洁，严谨，精确。
  - 自然语言：一般不使用在正式的文本中。

# 语法图

程序



# BNF表示方式

- $\langle \text{程序} \rangle ::= \langle \text{程序首部} \rangle ; \langle \text{程序分程序} \rangle$
- $\langle \text{程序首部} \rangle ::= \text{Program } \langle \text{标识符} \rangle ( \langle \text{程序参数} \rangle )$

# 程序设计语言的定义（语义）

- 程序设计语言中按照语法规则所构成的各个语法成分的意义。
- 定义方式：
  - 一般使用比较严格的自然语言进行描述。
  - 形式化的方法：使用数学符号以没有歧义的方式定义。



# 程序的执行

- 程序设计语言的执行基本有两种方式：
  - 解释方式：使用解释程序，对程序逐个语句进行分析，根据语句的含义进行执行。
  - 翻译方式：首先由翻译程序将程序翻译成为机器语言（或者虚拟机的语言），然后执行。
- 比较：
  - 翻译的方式可以使得一次翻译过后，多次运行。适于花较大的精力进行优化工作。

# 编译程序的构造

- 前端：完成分析工作（机器无关）
  - 词法分析：识别各个最小语法单位。
  - 语法分析：识别出各个语法结构。
  - 语义分析：确定类型，类型/运算合法性检查，识别含义和相应处理，静态语义检查。
- 后端：完成综合工作（机器相关）
  - 优化：改善目标代码质量。
  - 目标代码生成

# 编译程序的分类

- 诊断型编译程序
- 优化型编译程序
- 可重定目标型编译程序
- 交叉型编译程序
- 增量型编译程序

# 实际编译程序

- 开发支撑环境
- 预处理
- 非标准版本（方言，扩充）

# 编译原理讲义

(第二章:文法与语言)

南京大学计算机系

赵建华

# 文法与语言

- 文法被用来精确而无歧义地描述语言的句子的构成方式.
- 文法描述语言的时候不考虑语言的含义。

# 字母表

定义：字母表是有穷非空集合。

- 字母表包含了语言中所允许出现的一切符号。

# 符号串

- 定义：符号串是由字母表中的符号所组成的有穷序列。
- 一个语言的句子总是它的字母表的符号串。这个符号串的组成必须是按照文法规则组合而成的。
- 语法分析的一个重要任务就是：判断一个符号串的组成是否满足某个文法的规定，并且分析出是如何按照规则组成的。



# 关于符号串的概念和操作

- 运算：
  - 联结(并置):  $x=123, y=45$ 那么 $xy=12345$
  - 方幂:  $x$ 的 $n$ 次方幂即将 $n$ 个 $x$ 联结。
- 子符号串:  $v$ 是 $xvy$ 的子符号串。 $v$ 非空
- 头, 尾:  $x$ 是 $xy$ 的头,  $y$ 是 $xy$ 的尾。

# 符号串集合

- 定义：若集合A中的一切元素都是*同一个字母表上的集合*，那么A被称为*该字母表上的符号串集合*。
- 在本课程中，语言被认为是句子的集合。  
（外延定义？）所以，一个语言就是对应于它的字母表上的一个符号串集合。

# 符号串集合的运算

- 乘积：  $AB = \{xy \mid x \in A \text{ 且 } y \in B\}$
- 方幂： A的n次方幂就是将n个A相乘。
- 字母表的闭包与正闭包：
  - 字母表A的闭包是A上的所有符号串（包括空字符串）的集合。
  - 字母表A的正闭包是A上的所有的非空符号串的集合。

# 文法和语言的定义（重写规则）

- 重写规则：一个重写规则是一个有序对  $(U, u)$ ，通常写作  $U ::= u$ 。其中  $U$  是一个符号，称为左部； $u$  是一个符号串称为右部。有时也说这个规则是  $U$  的一个规则。
- 重写规则总是基于某个字汇表的。  $U$  和  $u$  中的符号必须都在这个字汇表内。这个字汇表中有些符号不能作为左部。
- 存在更加复杂的规则，但是这样的规则足够描述程序设计语言的文法。

# 文法和语言的定义（重写规则）

- 例如:
  - $\langle \text{标识符} \rangle ::= \langle \text{字母} \rangle$
  - $\langle \text{标识符} \rangle ::= \langle \text{字母} \rangle \mid \langle \text{标识符} \rangle \langle \text{字母} \rangle$
- 第二个规则实际使用了BNF的表示方法。  
BNF表示方法的还包括:
  - 花括号表示重复:  $\{ \langle \text{字母} \rangle \}$
  - 方括号表示可选:  $[ \langle \text{参数} \rangle ]$

# 文法和语言的定义（文法）

- 文法：文法 $G[Z]$ 是一组有穷非空的重写规则的集合。其中 $Z$ 称为识别符号。 $G$ 为文法名字
- 文法例子：P22, 例子2.10。
- 所有的规则都是基于同一个符号表 $V$ 。符号表又可分划非终结符号集合 $V_N$ 和终结符号集合 $V_T$ 。

- $\langle \text{句子} \rangle ::= \langle \text{主语} \rangle \langle \text{谓语} \rangle \langle \text{状语} \rangle$
- $\langle \text{主语} \rangle ::= \langle \text{名词} \rangle$
- $\langle \text{名词} \rangle ::= \text{Peter} \mid \text{Berry} \mid \text{River}$
- $\langle \text{谓语} \rangle ::= \langle \text{动词} \rangle$
- $\langle \text{动词} \rangle ::= \text{Swims}$
- $\langle \text{介词} \rangle ::= \text{in}$
- $\langle \text{状语} \rangle ::= \langle \text{介词} \rangle \langle \text{名词} \rangle$

# 文法和语言的定义（推导）

- 文法的作用是描述某种语言的句子的构成方式。使用文法我们可以产生对应语言的句子。
- 从识别符号开始，不断将当前符号串中的非终结符号替换为该符号的某个规则的右部。直到当前的符号串中所有的符号都是终结符号为止。



# 文法和语言的定义（推导）

- 例子：

〈句子〉  $\Rightarrow$  〈主语〉 〈谓语〉 〈状语〉

$\Rightarrow$  〈名词〉 〈谓语〉 〈状语〉

$\Rightarrow$  .....

$\Rightarrow$  Peter swims in river

# 文法和语言的定义（推导）

- 直接推导：  $v = xUy$ ,  $w = xuy$ , 并且  $U ::= u$  是文法中的一个重写规则，那么我们说  $v$  可以直接推导到  $w$ , 或者  $w$  可以直接规约到  $v$ 。记作  $v \Rightarrow w$ 。
- 例如：  
    〈主语〉  〈谓语〉  〈状语〉  
 $\Rightarrow$  〈名词〉  〈谓语〉  〈状语〉

# 文法和语言的定义（推导）

- 推导：对于符号串 $v$ 和 $w$ ，如果存在一个直接推导序列 $u_0 \Rightarrow u_1 \Rightarrow \dots \Rightarrow u_n$ ，并且 $u_0 = v$ ， $u_n = w$ ，那么我们说 $v$ 可以推导到 $w$ ，或者 $w$ 规约到 $v$ 。记作 $v \Rightarrow^+ w$ 。
- 这个推导长度为 $n$ ，且称 $w$ 为对应于 $v$ 的一个字。
- $v \Rightarrow^* w$  表示 $v = w$ 或者 $v \Rightarrow^+ w$ 。

# 文法和语言的定义（推导）

- 推导的例子：P25页例2.12。
- 文法：
  - $\langle \text{标号} \rangle ::= \langle \text{数字序列} \rangle$
  - $\langle \text{数字序列} \rangle ::= \langle \text{数字序列} \rangle \langle \text{数字} \rangle \mid \langle \text{数字} \rangle$
  - $\langle \text{数字} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid \dots \mid 9$
  - $VT = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ ,  $VN = \{\}$

# 推导的例子

<标号>  $\Rightarrow$  <数字序列>  
 $\Rightarrow$  <数字序列><数字>  
 $\Rightarrow$  <数字序列><数字><数字>  
 $\Rightarrow$  <数字><数字><数字>  
 $\Rightarrow$  <数字>23  
 $\Rightarrow$  123

# 语言的定义（句型,句子）

- 对于文法 $G[Z]$ ,  $x$ 称为 $G$ 的一个句型如果:

$$Z \Rightarrow^* x$$

识别符号是最简单的句型。

- $G[Z]$ 的一个句型 $x$ 被称为句子, 如果:

$$x \in V_T^*$$

也就是说句子是全部由终结符号组成的句型。

# 语言的定义（短语，简单短语）

- 短语：对于文法 $G[Z]$ ，如果 $Z \Rightarrow^* xUy$ ， $U \Rightarrow^+ u$ 。显然， $w=xuy$ 是一个句型。我们称 $u$ 是 *句型 $w$ 中相对于 $U$* 的短语。
- 简单短语：在上面的定义中，如果 $U ::= u$ 是 $G$ 的一个规则，那么， $u$ 是 *句型 $w$ 中相对于 $U$* 的简单短语。
- 例子：P22页例2.13。

# 语言的定义（短语，句柄）

- 注意：在寻找一个句型的短语（或简单短语）时，必须要求将这个短语规约为相应的非终结符号后所得到的符号串仍然是句型。
- 句柄：一个句型的最左简单短语称为该句型的句柄。
- 定义句柄的原因：在自底向上识别一个符号串时，总是规约这个句柄。



# 语言的定义（文法的语言）

- 文法的语言：一个文法 $G[Z]$ 的语言，用 $L(G[Z])$ 表示，定义如下：

$$L(G[Z]) = \{x \mid Z \Rightarrow^* x \text{ 并且 } x \in V_T^+\}$$

- 一个文法的语言就是该文法的所有的句子的集合。
- 文法的语言是所有终结符号串所组成的集合的子集，一般是真子集。

# 语言的定义（递归与语言）

- 递归的规则：  $U ::= \dots U \dots$
- 左右递归规则：  $U ::= U \dots$ ;  $U ::= \dots U$
- 文法的递归：  $U \Rightarrow^+ \dots U \dots$ ，称文法递归于U。
- 文法的左右递归：
- 如果文法是非递归的，那么其语言是有穷的。

# 文法与语言（例子）

- $G[A]: A ::= bA | a;$ 
  - $L(G[A]) = \{b^i a | i \geq 0\}$
- $G[Z]: Z ::= Ab; A ::= aaA \quad A ::= aa$ 
  - $L(G[Z]) = \{a^{2^n} b | n \geq 1\}$

# 语言的分类

- Chomsky文法的定义：

$$(V_N, V_T, P, Z)$$

- 该定义是我们前面讲的定义的一个更加形式化的表达。
- 在这个定义中，文法规则的左部可以是一个非空符号串。
- Chomsky文法被分为四类，我们主要用2型和3型文法。

# Chomsky文法类（0型文法PSG）

- 0型文法的规则形如： $u ::= v$ ， $u, v$ 为符号串，且 $u$ 非空。0型文法的相应语言称为0型语言，又称为递归可枚举集合。
- 0型语言是不可判定的。
- 例子：G:  $Z ::= \#A1\#$ ;  $\#A ::= \#$ ;  $A1 ::= 11A$ ;  
 $A\# ::= B\#$ ;  $1B ::= B1$ ;  $\#B ::= \#A$ 
  - $L(G) = \{\#1^i\# \mid i = 2^n, n \geq 0\}$

# Chomsky文法类（1型文法CSG）

- 1型文法的规则如下： $xUy ::= xuy$ ，其中U为非终结符号， $x, y, u$ 为符号串，且 $u$ 非空。1型文法又称为上下文相关文法。
- 1型文法也可以如下定义：所有的规则的右部都不比左部短。
- 1型文法是可判定的。但是现在没有找到有效的方法。

# Chomsky文法类（2型文法CFG）

- 2型文法的规则有如下形状： $U::=u$ ，其中U是非终结符号，u是符号串。2型文法又称为上下文无关文法。
- 一般的程序设计语言的语法都使用2型文法描述。
- 2型文法是可判定的，且又有效的判定方法。

# Chomsky文法类（3型文法RG）

- 文法规则的形状： $U::=T$ 或者 $U::=WT$ ，其中 $U$ ， $W$ 是非终结符号， $T$ 是终结符号。
- 3型文法又称为正则文法，其语言也称为正则语言。



# 语言类对运算的封闭性

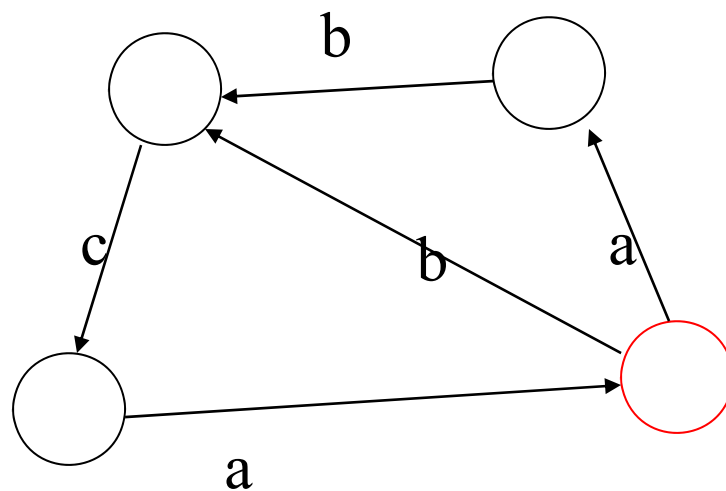
- 给定某个语言类中的语言，如果对它们进行某种运算之后得到的新语言仍旧是该类语言，那么该语言类对此运算封闭。
- 所有语言类对并，乘积，闭包运算封闭。
- CFG语言类对交，补运算不封闭。
- 正则语言类对交并补运算封闭。

# 3型语言与有穷自动机

- 任何一个3型语言都可以使用一个有穷自动机来识别。
- 有穷自动机包括一个有限控制器，和一个输入带。机器从输入带从左到右逐个读入输入符号，最终根据有限控制器的状态确定输入的符号串是否是该型语言的句子。机器的每一个动作根据当前读入的符号和当前状态确定。

# 有穷自动机例子

abcbabcaabc



# 2型语言与下推自动机

- 任何一个2型语言都可以使用一个下推自动机来识别。
- 下推自动机相当与一个有穷自动机和一个栈。自动机的每一步动作根据栈顶的符号，当前读入的符号，一个有限控制器的当前状态来确定，可以包括读入符号，压栈，出栈，和确定接受。

# 形式语言与程序设计语言

- 虽然程序设计语言的语法都使用上下文无关文法来描述，但是通常语言都是上下文相关的。
- 使用上下文无关文法描述语言的原因是：存在高效处理上下文无关文法的技术。

# 关于CFG的进一步讨论

- Chomsky范式：所有的上下文无关语言都可以用如下形式的文法产生：所有的规则都形如： $U ::= VW$  或者  $U ::= T$ ，其中 $U, V, W$ 为非终结符号， $T$ 为终结符号。
- Greibach范式：所有上下文无关语言都能由这样的文法产生： $U ::= Tu$ ，这里 $U$ 为非终结符号， $T$ 为终结符号。

# 关于CFG的进一步讨论

- 自嵌套：一个上下文无关文法为自嵌套的，如果存在一个非终结符号U满足：

$$U \Rightarrow^* xUy, \text{ 且 } x, y \text{ 非空。}$$

- 定理2.6 若一个CFG  $G[Z]$ 不是自嵌套的，那么 $L(G)$ 必然是一个正则语言。
- 但是，自嵌套的上下文无关文法也可能产生正则语言。例:P35页

# 关于推导的性质

- 定理2.7 对于CFG, 如果存在句型  $x = x_1 x_2 \dots x_n$  且  $x \Rightarrow^* y$ , 必然存在  $y_1, y_2, \dots, y_n$  使得:

$$x_i \Rightarrow^* y_i \text{ 且 } y = y_1 y_2 \dots y_n.$$

- 定理2.8 如果:  $x \Rightarrow^* y$ , 如果x的首符号是终结符号, 则y的首符号也是终结符号; 反之, 如果y的首符号是非终结符号, 那么x的首符号也是非终结符号。



# 空规则

- 定理2.9 设 $L$ 是由上下文无关文法  $G=(V_N, V_T, P, Z)$ 产生的语言， $P$ 中可能包含空规则，则 $L$ 能由这样的文法产生，在这样的文法中每一个规则或者是 $U::=u$ ，或者 $Z::=\epsilon$ .
- 这个定理表示：在语言中增加和删除一个空串，并不会改变语言的类别。

# 文法等价

- 定义： 设 $G$ 和 $G'$ 是两个文法， 如果 $L(G)$ 等于 $L(G')$ ， 那么我们说 $G$ 和 $G'$ 等价。
- 例子：
  - $G[S]$   $S ::= ABC$   $A ::= Aa|a$   $B ::= Bb|b$   $C ::= Cc|c$
  - $G'[S]$   $S ::= Sc|Bc$   $B ::= Bb|Ab$   $A ::= Aa|a$
- 两个CFG文法是否等价是不可判定的。

# 文法的等价变换

- 当有些技术不能处理一种文法时，我们可以将它处理为另外一个等价文法来处理。这就是等价变换。
  - 使文法和语言类一致。
  - 消除二义性。
  - 使文法适用于某种分析技术。
  - 文法满足某种特殊需要。

# 文法等价变换的种类

- 压缩文法等价变换
- 增广文法等价变换
- 范式文法等价变换
- 消去左递归等价变换

# 压缩文法等价变换

- 主要作用是删除文法中不可能被使用的规则，称为多余规则。包括：
  - 规则的左部不可能在句型中出现。
  - 使用了此规则之后，句型永远也不能推导得到句子。
- 一个规则 $U ::= u$ 不是多余的，当且仅当：
  - $Z \Rightarrow^* xUy$ ，且 (条件1)
  - $U \Rightarrow^+ t$ ，且 $t$ 是终结符号串。 (条件2)

# 压缩文法等价变换

- 已压缩文法定义：没有多余规则的文法称为压缩了的（或已压缩的）文法。
- 压缩算法：
- 算法**重复执行**下面两个部分，直到不能删除更多的规则：
  - 删除不满足条件一的规则。（子算法1）
  - 删除不满足条件二的规则。（子算法2）

# 压缩文法等价变换（子算法1）

- 步骤1：对规则中识别符号 $z$ 加标记；
- 步骤2：对左部非终结符号加有标记的规则，将其右部中的所有非终结符号加标记。
- 步骤3：检查是否一切非终结符号都加过标记。是，结束；否，执行步骤4。
- 步骤4：如果上一次步骤2中没有多加标记，删除所有左部没有加标记的规则，结束。否则，转向步骤2。

# 压缩文法等价变换（子算法1）

- 例子：

$Z ::= Be \quad A ::= Ae \quad A ::= e$

$B ::= Ce \quad B ::= Af \quad C ::= Cf$

$D ::= f$



# 压缩文法等价变换（子算法2）

- 步骤1：对 $u \in V_T^+$ 的规则 $U ::= u$ 的左部非终结符号 $U$ 加标记。
- 步骤2：对右部仅包含终结符号和已加标记的非终结符号的规则的左部加标记。
- 步骤3：检查是否对一切非终结符号加过标记。是，结束；否则，执行步骤4。
- 步骤4：如果上一次步骤2执行时没有多加任何标记，那么删除左部没有加标记的规则，否则，转到步骤2。

# 压缩文法等价变换（子算法2）

- 例子：

$$Z ::= Be$$

$$A ::= Ae$$

$$A ::= e$$

$$B ::= Ce$$

$$B ::= Af$$

$$C ::= Cf$$

# 增广文法等价变换

- 一般来讲，以识别符号为左部的规则有多个。在规约的时候使用的规则也时不唯一的。增广文法变换使得文法只有一个以识别符号为左部的规则。
- 变换方法： $G[Z]$ 变换为 $G[Z']$ ，且增加规则 $Z' ::= Z$ 。有时候，新的规则为 $Z' ::= Z\#$ 。此时所得到的语言有所不同。
- 是一种变换的特例：P40页。

# 消单规则等价变换

- 目的：提高分析算法的效率。注意：单规则有时是有用的，但是，太多的单规则会影响分析的效率。
- 算法的基本思想是：
  - 首先，对于每个 $U$ ，求解出所有的 $V$ ，使得  $U \Rightarrow^* V$ 。
  - 对于所有的  $U \Rightarrow^* V$ ，且  $V ::= u$ ，增加规则  $U ::= u$ ，得到的文法依旧是等价的。

# 消单规则等价变换（算法）

- 步骤1：对每个  $U \in V_N$ ，构造

$$N_U = \{V \mid U \Rightarrow^* V, V \in V_N\}$$

- 步骤2：构造

$$P' = \cup \{U_i ::= u \mid V ::= u \in P, V \in N_U, |u| > 1 \text{ 或 } u \notin V_N\}$$

- 步骤3：新的不含单规则的文法为：

$$(V_N, V_T, P', Z)$$

# 消单规则等价变换（例子）

- G2.10[E]:

$$E ::= E + T$$

$$E ::= T$$

$$T ::= T * F$$

$$T ::= F$$

$$F ::= (E)$$

$$F ::= i$$

- $N_E = \{E, T, F\}$

...

- $E ::= E + T \mid T * F \mid (E) \mid i$

...

# 习题

- $G[A] \ A ::= aAb \mid ab$ , 证明:  $L(G) = a^n b^n$
- 设计文法:  $\{a^n b^m c^m d^n\}$

# Chomsky范式范式变换

- 步骤1：消单规则。
- 步骤2：变换成为： $U ::= T$  或者  $U ::= V_1 V_2 \dots V_m$
- 步骤3：引入新的非终结符号。
- $U ::= V_1 V_2 \dots V_m$  修改为  
 $U ::= V_1 W_1 ; \quad W_1 ::= V_2 W_2 ; \quad \dots$   
 $W_{m-1} ::= V_{m-1} V_m ;$



# Chomsky范式范式变换（例子）

- 步骤1：对于文法G2.10[E]，消除单规则：  
 $E ::= E+T \quad E ::= T * F \quad E ::= (E) \quad E ::= i \dots$
- 步骤2：引入规则  $A ::= + \quad M ::= * \dots$   
原来的规则变为:  $E ::= EAT \quad E ::= TMF \dots$
- 步骤3：
- 原来的规则变为:  $E ::= EB \quad B ::= AT \dots$

# 消规则左递归等价变换

- 改写规则左递归成为右递归

将 $E ::= E+T \mid T$ 改写为:

$$E ::= TE' \quad E' ::= +TE' \mid \varepsilon$$

# 消规则左递归等价变换(例子)

- $E ::= E+T|T \quad T ::= T*F|F \quad F ::= (E)|i$

- 变换得到如下文法:

$$E ::= TE' \quad E' = +TE'|\varepsilon$$
$$T ::= FT' \quad T' = *FT'|\varepsilon$$
$$F ::= (E)|i$$

# 消规则左递归等价变换(BNF)

- 沿用扩充BNF表示法

$E ::= E+T \mid T$  改写为  $E ::= T\{+T\}$

- 步骤1：提取左因子：
  - $U ::= ux|uy|\dots|uz \Rightarrow U ::= u(x|y|\dots|z)$
- 步骤2：假定  $U ::= x|y|\dots|z|Uu$ ；替换为
  - $U ::= (x|y|\dots|z)\{u\}$

# 消规则左递归等价变换(例子2)

- $E ::= T \mid -T \mid E+T \mid E-T$
- 步骤1: 提因子
  - $E ::= (T|-T) \mid E(+T|-T)$
- 步骤2:
  - $E ::= (T|-T) \{+T|-T\}$
  - 或者  $E ::= (-|\epsilon)T \{(+|-)T\}$

# 消文法左递归(方法)

- 要求：文法不包含回路，无空规则
- 步骤1：排列非终结符号 $U_1, U_2, \dots, U_n$
- 步骤2：执行程序（见下一页）
- 步骤3：删除无用规则。
- 原理：将非终结符号排序之后，消除所有形如 $U_i ::= U_j x$ 的规则，其中 $i \geq j$ 。这样，对于任何非终结符号 $U_i$ ，如果 $U_i \Rightarrow^+ U_j x$ ，必然有 $j > i$ 。不可能有文法左递归。

# 消文法左递归(步骤2的程序)

- For(i=1; I<=n; I++)

- for(j:=1;j<= i-1; j++)

- {

- 将形如 $U_i ::= U_j r$ 的规则改写为

- $$\underline{U_i ::= x_{j1} r | x_{j2} r | \dots | x_{jk} r}$$

- 消除 $U_i$ 的规则左递归, 新的规则加入文法。

- }

# 消文法左递归(步骤2的解释)

- 在改写规则的时候, 对于每个 $i$ , 得到的规则保证:  $U_i ::= U_j x$ 时, 必然有 $j > i$ 。改写的规则也包括新加入的规则。



# 消文法左递归（例子）

- $S ::= Sa \mid Tbc \mid Td \quad T ::= Se \mid gh$
- 步骤1：排序:  $S \quad T$
- 步骤2：循环：
  - $i = 1, j = 1$ ; 消除规则左递归。
  - $i = 2, j = 1$ ; 消除  $T ::= Sx$  的情况，并消除规则左递归。
- 步骤3：

# 语法树的概念

- 定义：语法树是这样的一个语法结构，它的结点由符号组成。根结点对应于识别符号。只有非终结符号对应的结点有子结点。并且，一个结点和它的子结点分别对应于文法中的一个规则的左部和右部。
- 引入语法树的意义：作为识别句子的辅助工具，语法树可以表示句子的结构。这一点对于其后的语义分析有非常重要的意义。

# 语法树的相关概念

- 结点：每个树的结点对应于一个符号。结点的名字就是该符号。
- 边：两个结点之间的连线。
- 根结点：没有边进入的结点。
- 分支：某个结点向下射出的边和其结点称为分支。（父子结点，兄弟结点）
- 子树：语法树的某个结点和它向下射出的部分。

# 语法树的相关概念（续）

- 末端结点：没有向下射出的边的结点成为末端结点。在相对于句型的语法树中，末端结点可能是非终结符号。

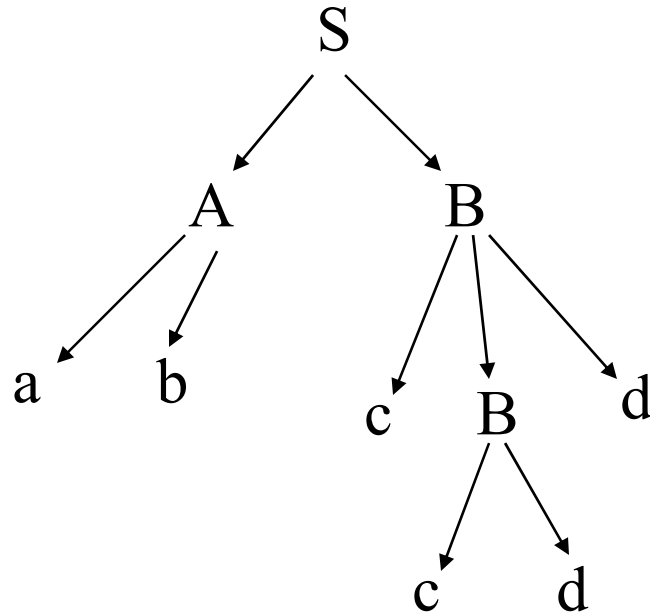
# 语法树的例子

语法 $G[S]$ :

$S ::= AB$

$A ::= aAb \mid ab$

$B ::= cBd \mid cd$



$S \Rightarrow AB \Rightarrow abB \Rightarrow abcBd \Rightarrow abccdd$

# 从推导构造语法树

- 步骤1：根结点为识别符号。
- 步骤2：对于每一次推导使用的规则  $U ::= u$ ，找出U对应的结点（此时应该是末端结点），从该结点向下画分支，子结点从左到右分别是u中从左到右的符号。
- 重复步骤2直到推导的最后一步。

# 语法树中子树的性质

- 定理2.12： 一个子树的末端结点的组成的符号串是相对于子树的根结点的短语。
- 如果这个子树的高度是1（两层）的话，其末端结点组成的符号串是根结点的简单短语。

# 从语法树构造推导

- 构造的过程是一个剪枝的过程。每剪掉一个分支，添加一步规约过程。
- 首先，任意找一个高度为1的子树，确定这个子树对应的规则。将子树的分支剪掉，得到一个新的语法树。该语法树的末端结点对应的符号串就是经过一步规约之后得到的句型。如此循环，直到语法树只剩下一个根结点。



# 从语法树构造推导（续）

- 同一棵语法树可以得到不同的推导，但是其实质上是使用规则的顺序不同。这些推导的过程实际是等价的。
- 例如：
  - $S \Rightarrow AB \Rightarrow AcBd \Rightarrow Accdd \Rightarrow abccdd$
  - $S \Rightarrow AB \Rightarrow abB \Rightarrow abcBd \Rightarrow abccdd$

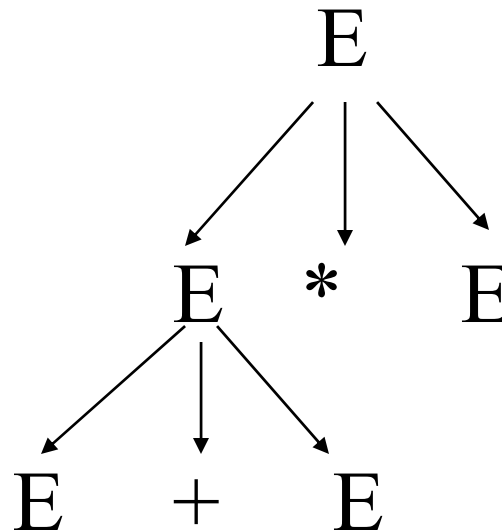
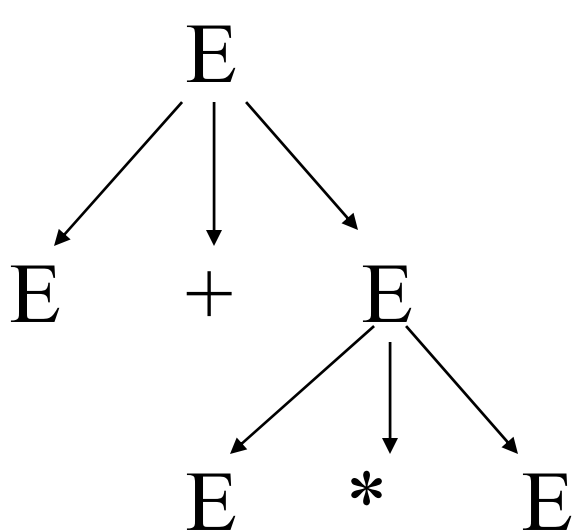
# 文法的二义性

- 定义：如果对于某文法的同一个句子存在两棵不同的语法树，则该句子是二义性的。而该文法是二义性文法。
- 这里的二义性是指语法结构上的。
- 如果一个句子具有二义性，那么对这个句子的结构可能有多种‘正确’的解释。通常情况下，句子的语义要通过其语法结构来定义，所以，二义性一般是有害的。

# 文法二义性（例子）

文法 $G[E]$ :  $E ::= E + E \mid E * E \mid (E) \mid i$

文法的句子:  $i + i * i$ , 其语法树如下:



# 二义性

- 一般来讲，二义性是针对文法而言的，说语言二义性是无意义的。
- 定义2.35：如果某个语言对应的文法都是二义性的，那么这个语言被称为先天二义性的。
- 定理2.13：文法的二义性是不可判定的。
- 即使文法是无二义性的，其句子对应的语义仍然必须有严格的定义，才可以避免语义的二义性。

# 句型分析（概念）

- 句型分析就是识别一个符号串是否是某文法的句型。它是一种推导或语法树的构造过程。对于一个给定的符号串，试图按照文法的规则构造其对应的推导，或语法树。

# 分析技术分类

- 自顶向下技术:从文法的识别符号开始,由它推导出与输入符号相同的符号串。
- 从语法树的角度看,自顶向下的分析过程就是:以识别符号作为根结点,试图从根结点开始逐步建立语法树。该语法树的末端结点组成的符号串正是输入符号串。

# 自顶向下分析例子

- 文法G2.16[S]:

–  $S ::= AB$        $A ::= aAb|ab$        $B ::= cBd|cd$

–  $S \Rightarrow AB$       ( $S ::= AB$ )

–  $\Rightarrow aAbB$       ( $A ::= aAb$ )

–  $\Rightarrow aabbB$       ( $A ::= ab$ )

–  $\Rightarrow aabbcd$       ( $B ::= cd$ )

- 在构造推导的过程中，我们总是选择正确的规则。实际上，如何正确选择规则恰恰是自顶向下分析技术的核心。

# 最左（右）推导

- 最左（右）推导定义：在一个推导的过程中，如果每一步直接推导所被替换的总是最左（右）的非终结符号，那么这种推导称为最左（右）推导。
- 一个语法树只对应与一个最左（右）推导。



# 自底向上分析技术

- 自底向上分析技术：从输入符号串出发，试图把它规约为识别符号。
- 从语法树的角度看，这个技术首先以输入符号作为语法树的末端结点，然后向根结点方向构造语法树。

# 自底向上分析例子

- 文法G2.16[S]:

–  $S ::= AB$        $A ::= aAb|ab$        $B ::= cBd|cd$

–  $aAbcd \Rightarrow aabbcd$       ( $A ::= ab$ )

–  $Acd \Rightarrow$       ( $A ::= aAb$ )

–  $AB \Rightarrow$       ( $B ::= cd$ )

–  $S \Rightarrow$       ( $S ::= AB$ )

- 在上面的例子中间，我们总是找到了正确的简单短语进行规约。实际上，如何找到正确的简单短语（句柄）是自底向上分析技术的核心。

# 最左（右）规约

- 最左（右）规约定义：在一个规约的过程中，如果每步直接规约的总是最左（右）的简单短语那么这种规约称为最左（右）规约。
- 在自底向上分析技术中，使用的一般都是最左规约。所以我们会特别地定义句柄（最左简单短语）的概念。
- 最左规约和最右推导使用的规则顺序恰好相反。

# 句型分析的基本问题

- 自顶向下过程中，如何确定使用哪个规则？
- 自底向上的过程中，如何找出简单短语？如何进行规约？
- 需要让计算机自动高效地解决上面的问题。

# 规范推导与规范规约

- 规范直接推导：  $xUy \Rightarrow xuy$  中，  $y$  不包含非终结符号。记作  $xUy \Rightarrow xuy$ 。
- 规范推导： 推导中的每一步直接推导都是规范的。记作  $v \Rightarrow^+ w$ 。
- 规范句型： 可以从识别符号开始，通过规范推导得到的句型称为规范句型。

# 编译原理讲义

## (第三章:词法分析)

南京大学计算机系

赵建华

# 词法分析与词法分析程序

- 词法分析的任务是识别源程序中具有独立含义的最小语法单位----符号或单词,如标识符, 无正负号常数与界限符等。并把源程序转换为等价的内部表示形式。
- 功能:
  - 读入源程序字符串; 识别单词 (符号);
  - 转换成属性字;
  - 一些其他的简单工作: 删除注解, 预加工处理

# 符号识别和重写规则

- 规则例子：
  - $\langle \text{标志符} \rangle ::= \text{字母} \mid \langle \text{标识符} \rangle \text{字母} \mid \dots$
  - 在上面的例子中，实际可以展开为  $\langle \text{标识符} \rangle ::= a \mid b \mid c \mid d \mid \dots$
- 如果我们规定终结符号的集合为字符，那么我们会发现上面的规则都满足正则文法的规则限制:  $U ::= T \mid UT$ 。因此，我们可以使用有限自动机来辅助完成词法分析。

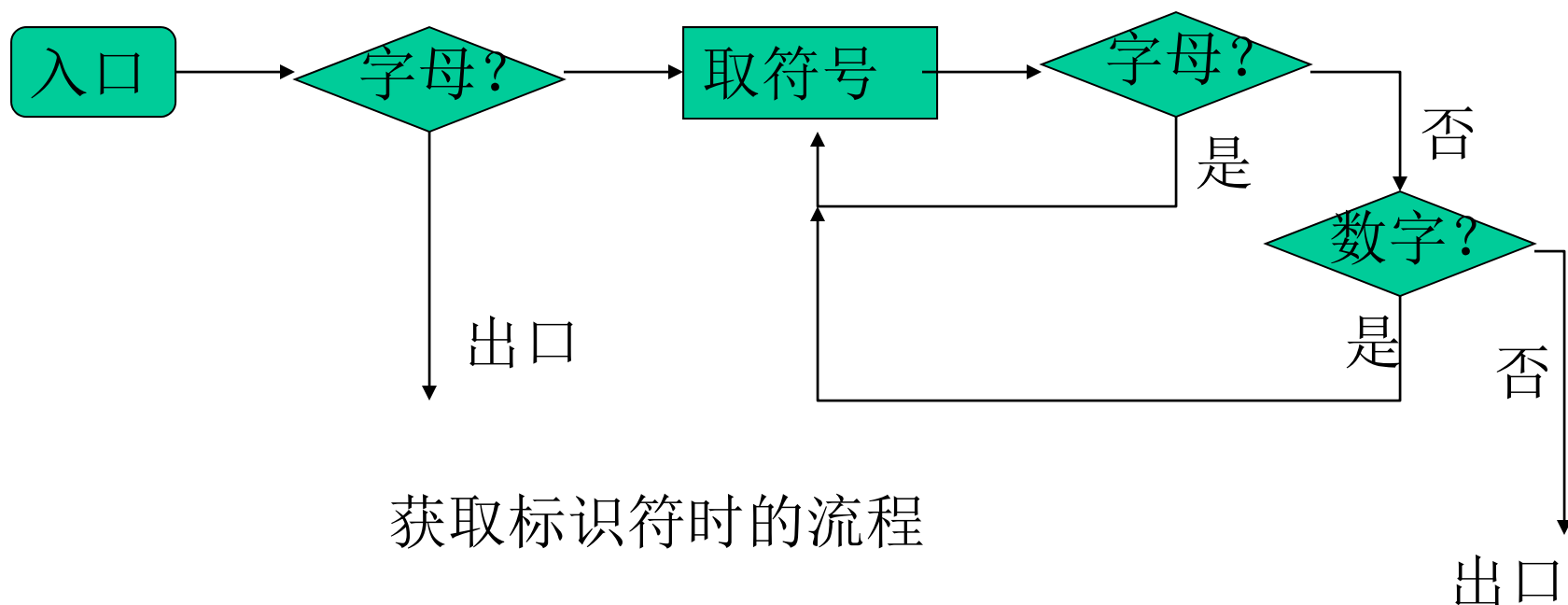


# 实现方式

- 完全融合：由于正则文法是CFG的一种特例。所以，可以把词法规则和语法规则统一处理。
- 相对独立：词法分析作为子程序。当语法分析程序需要读下一个符号的时候，调用这个子程序。
- 完全独立：词法分析程序作为单独的一趟来实现。

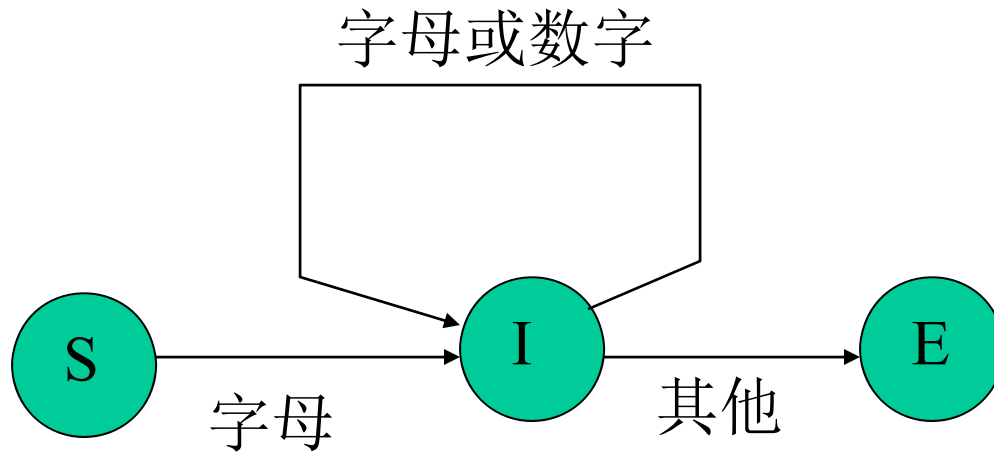
# 状态转换图与转换系统

- 在流程图中的每个边上的操作一般都是判断输入符号是否满足等于某些值。



# 状态转换图与转换系统

- 我们可以把流程图简化为一个状态转换图。状态转换图的运行由输入驱动。



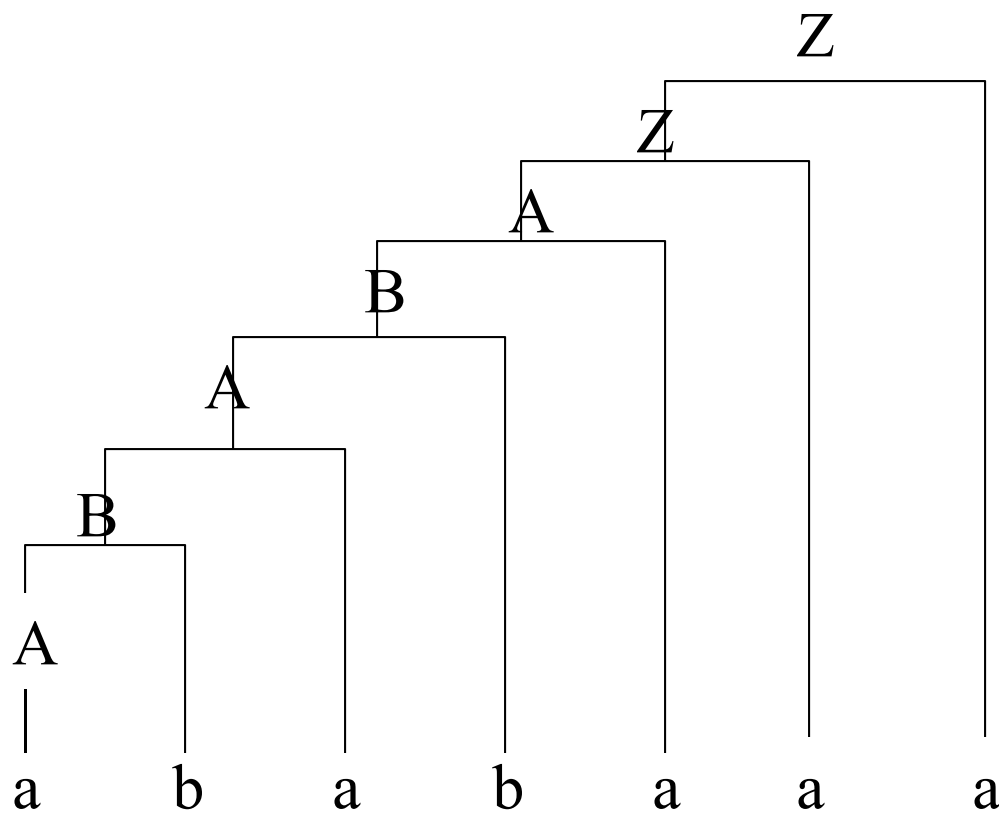
# 转换图的运行和接受

- 转换图在运行的时候，由一个当前的状态。每次输入一个符号的时候，转换图的状态改变如下：沿着从当前状态离开的，并且用输入符号标记的边，到达这个边的目标状态。
- 一个符号串被状态转换图接受当且仅当从初始状态出发，逐次输入符号串中的所有符号的时候，最终的状态是接受状态。

# 从文法构造状态转换图

- 我们可以给每个正则文法构造一个状态转换图。
- 其基本思想如下：
  - 当我们使用自底向上的方式规约一个正则文法的句子的时候，得到的句型都是形如  $Utx$  ( $tx$  为终结符号串)。而下次规约的规则总是满足  $V ::= Ut$ ，句型被规约为  $Vx$ 。如果我们把非终结符号作为状态，而把  $tx$  看作尚未读入的部分时，就得到一个状态转换图。P61 图3-4。

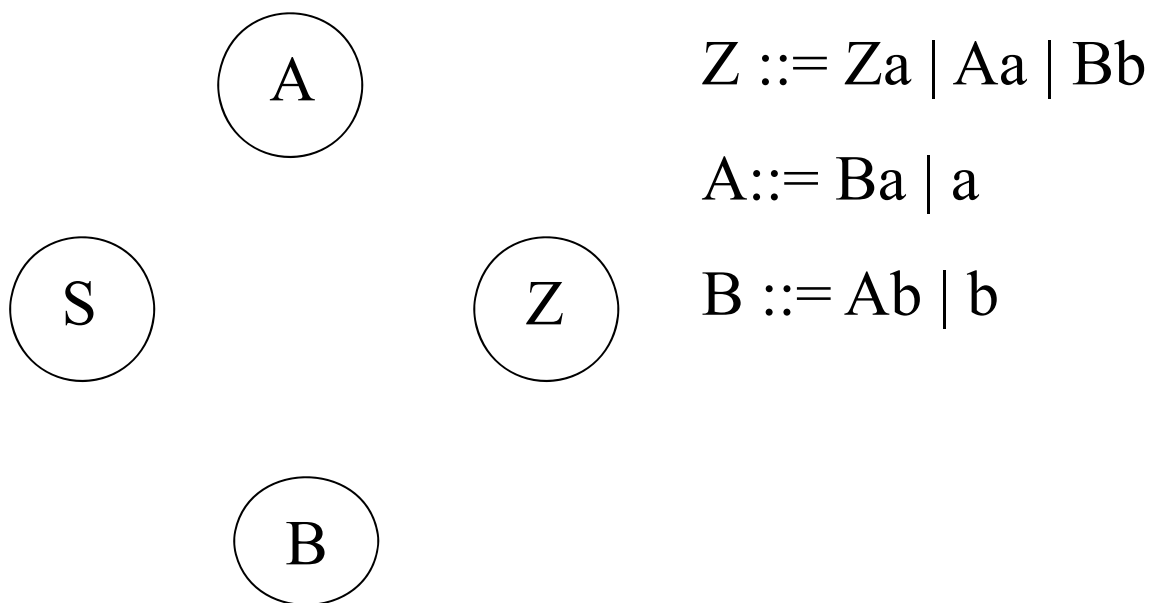
# 正则文法的语法图



# 从文法构造状态转换图(算法)

- 步骤一：引入一个开始状态S（假定S不是非终结符号。
- 步骤二：每个非终结符号作为一个结点。
- 步骤三：
  - $Q ::= T$ ：从S到Q有一条标记为T的弧。
  - $Q ::= RT$ ：从R到Q有一条标记为T的弧。
- 识别符号为接受状态。

# 从文法构造状态转换图(例子)



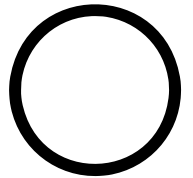
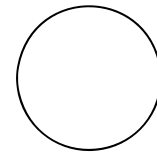
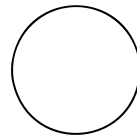
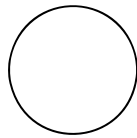


# 使用状态转换图构造正则文法

- 从构造步骤来看，正则文法和状态转换图之间具有一种对应关系。
- 状态转换图具有直观的特点，所以给定一个正则语言，构造状态转化图比较方便。

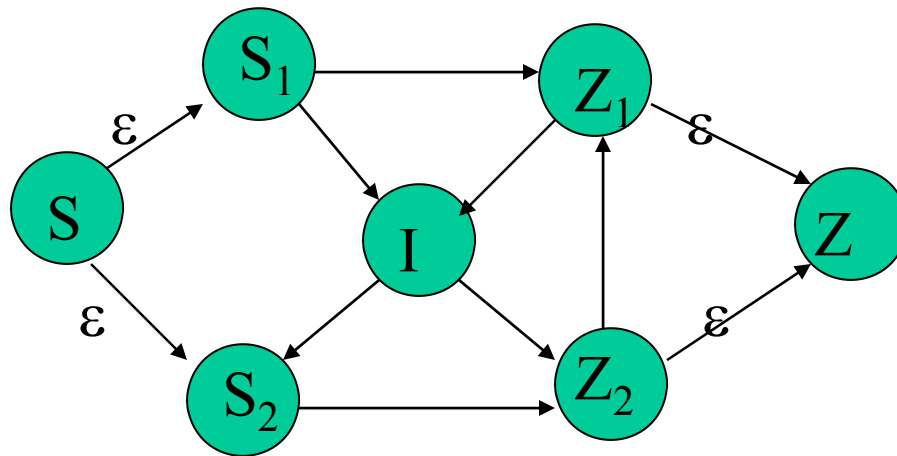
# 构造正则文法(例子)

- 例子:  $\{(ab)^n b^2 \mid n \geq 0\}$



# 转换系统

- 定义：转换系统是具有下列特征的状态转换图：
  - 只有唯一的开始状态S和唯一的终止状态Z；
  - 没有弧进入S，也没有离开Z
  - 可能有空弧。



# 转换系统

- 定理3.2 对于任何一个状态图，都存在一个相应的转换系统，它们接受同样的语言。
- 转换系统对于从正则表达式转换到状态转换图起到了中间表示的作用。

# DFA的定义

- 有穷状态自动机是对状态转换图的一种形式化定义。
- 定义：一个DFA是 $(K, \Sigma, M, S, F)$ 
  - $K$  有穷非空状态集合
  - $\Sigma$  有穷的输入字母表集合
  - $M$  从 $K \times \Sigma$  到 $K$ 的映射。
  - $S$  是 $K$ 中的一个状态，称为开始状态。
  - $F$   $K$ 的子集，称为终止状态集合。

# DFA的例子

- $(\{S, Z, A, B\}, \{a, b\}, M, S, \{Z\})$ ;

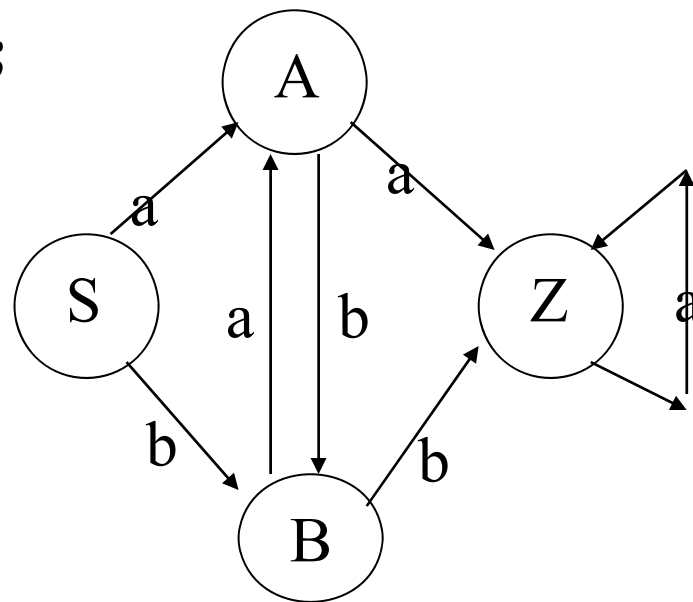
- 其中

- $M(S, a) = A$      $M(S, b) = B$

- $M(A, a) = Z$      $M(A, b) = B$

- $M(B, a) = A$      $M(B, b) = Z$

- $M(Z, a) = Z$



# 运行DFA

- 扩充后的映射M的定义：
  - $M(R, \varepsilon) = R$  表示一定要有输入符号后，DFA的状态才可能改变。
  - $M(R, Tt) = M(M(R, T), t)$  表示状态的改变是从左到右进行的。
- 定义：字符串t被DFA接受当且仅当 $M(S, t)$ 在F中。

# DFA接受的符号串集合

- 定义：(D)FA所能够接受的所有字符串的集合写成 $L(D)$ ， $L(D)$ 是正则集合，。
- 定理3.3 设 $G$ 为正则文法，如果 $x$ 属于 $L(G)$ ，那么它能被 $G$ 相应的有穷自动机所接受。
- 定理3.4 接受正则语言 $L$ 的最小状态自动机不记同构是唯一的。

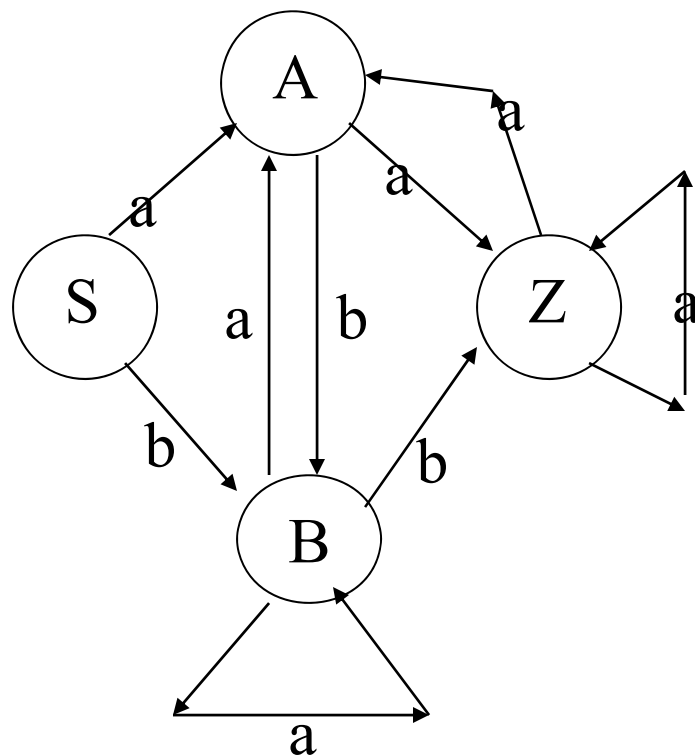


# 非确定有穷状态自动机NFA

- 在前面的从正则文法构造FA的例子中，恰好从一个状态射出的弧的标记是两两不同的。但是，如果有两个规则 $V ::= UT$ 和 $W ::= UT$ ，那么从U到V和W的弧的标记都是T。
- 此时，不能用DFA的映射来表示状态为U时，输入T时的后继状态。也就是说，当状态为U，输入为T时，这个转换图的下一步动作出现了不确定性。这个状态转换图不再是DFA。

# 不确定状态转换图的例子

- $Z ::= Za \mid Aa \mid Bb$
- $A ::= Ba \mid Za \mid a$
- $B ::= Ab \mid Ba \mid b$

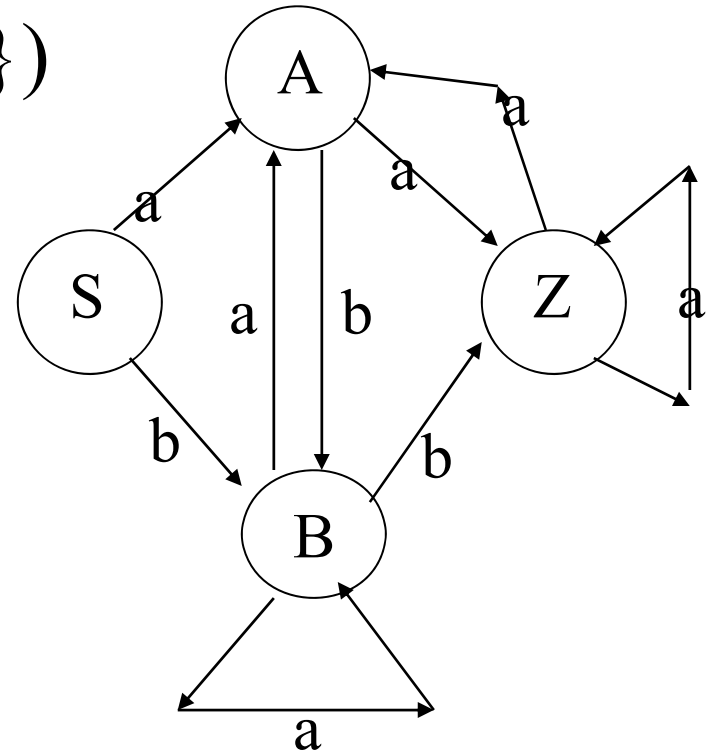


# NFA的定义

- 定义：一个NFA是 $(K, \Sigma, M, S, F)$ 
  - $K$  有穷非空状态集合
  - $\Sigma$  有穷的输入字母表集合
  - $M$  从 $K \times \Sigma$  到 $K$ 的超集的映射。
  - $S$  是 $K$ 的子集，称为开始状态集合。
  - $F$   $K$ 的子集，称为终止状态集合。
- NFA和DFA的不同在于：
  - $M$ 的值域是 $K$ 的超集。
  - 开始状态有不只一个

# NFA的例子

- $(\{S, A, B, Z\}, \{a, b\}, M, \{S\}, \{Z\})$
- $M$ :
  - $M(S, a) = \{Z\}$     $M(S, b) = \{B\}$
  - $M(A, a) = \{Z\}$     $M(A, b) = \{B\}$
  - $M(B, a) = \{A, B\}$     $M(B, b) = \{Z\}$
  - $M(Z, a) = \{A, Z\}$     $M(Z, b) = \{\}$



# 运行NFA

- 扩充的映射M
  - $M(\{R_1, R_2, \dots, R_n\}, T) = \cup_i M(R_i, T)$
  - $M(Q, \varepsilon) = Q$  表示一定要有输入符号后, NFA的状态才可能改变。
  - $M(Q, Tt) = M(M(Q, T), t)$  表示状态的改变是从左到右逐个进行的。
- $M(Q, x)$ 表示当输入符号串为x的时候, 所有可能的最后状态。

# NFA接受的语言

- 对于某个输入符号串 $x$ ，如果 $M(S,x) \cap F$ 非空，那么我们就说 $x$ 被该NFA接受。
- 直观地讲， $x$ 被一个NFA接受表示从某个开始状态，按照某种选择，可以到达某个接受状态。表3-2。
- 定理3.5：对于任何一个正则文法 $G$ ，都存在一个NFA  $A$ ，使得 $L(G)=L(A)$ 。反之亦然。

# 从NFA到DFA

- 使用NFA判定某个输入符号串的时候，可能出现不确定的情况：不知道下面选择那个状态。如果选择不好，该输入符号串可能不能到达终止状态。但是，我们不能说该输入符号串不能被该NFA接受。
- 如果通过尝试的方法，不断试探来确定输入符号串是否可被接受，那么判定的效率将降低。解决的方法是将NFA转换为等价的DFA。

# 从NFA到DFA

- 定理3.6 对于给定的NFA  $(K, \Sigma, M, S, F)$ , 可以构造一个等价的DFA  $(K', \Sigma, M', S', F')$  如下:
  - $K'$ 中的状态一一对应于 $K$ 的子集。和子集  $\{Q_1, Q_2, \dots, Q_n\}$ 对应的状态为  $[Q_1, Q_2, \dots, Q_n]$
  - $M'([Q_1, Q_2, \dots, Q_n], T) =$  对应于  $\cup_i M(Q_i, T)$  的状态。
  - $S'$ 为 $S$
  - $F'$ 为对应于所有与 $F$ 相交的子集的状态。



# 从NFA到DFA

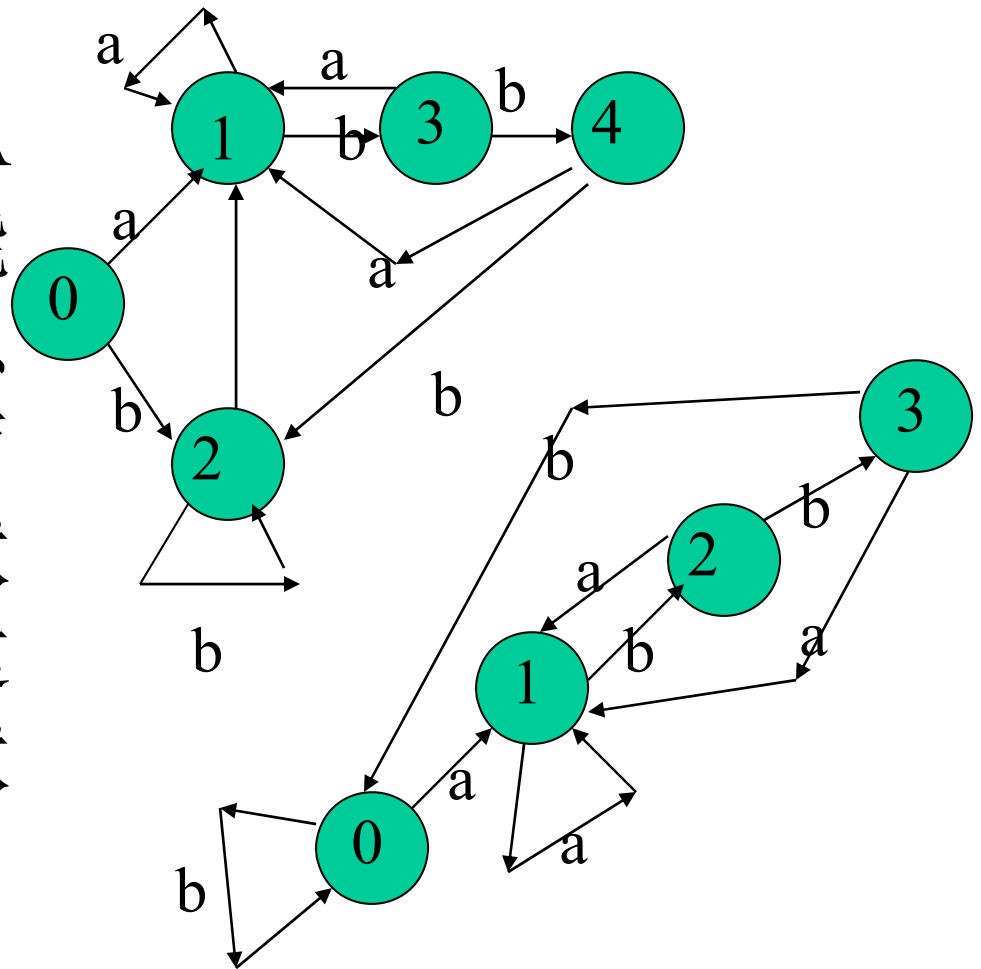
- 例子3.7
- 由于 $K$ 的子集的个数为 $K$ 的个数的指数个，所以以上面的方法得到的状态非常多。实际上不是所有的子集都可以到达的。
- 可以使用列表法来完成DFA的构造。从开始状态起，只有可以达到的状态才构造其后继。

# 自动机的等价

- 定义3.8: 如果 $L(A)=L(A')$ , 那么我们说A和A'等价。
- 定理3.8: 存在判定两个自动机是否等价的算法。

# 确定自动机的等价

- 两个等价的DFA的状态个数可能不同。DFA可以进行化简。化简不仅是去除死状态，不可能到达状态，还包括状态的合并。



# 自动机状态的区分与等价

- 定义3.9 如果从P开始输入w使得结束时候的状态的终止状态。从Q开始输入w时，结束的状态为非终止状态（或无状态），那么我们说w把P和Q区分开来。
- 定义3.10 不可区分的两个状态成为等价的状态。
- 空串把终止状态和非终止状态区分开来。
- 显然，如果P,Q等价，那么P和Q对于同一个符号的后继也等价。

# DFA化简

- 步骤1：将DFA的状态分为互不相交的子集，使得每个子集中的状态等价（见分划算法）。
- 步骤2：每个子集中选取一个状态作为子集中所有状态的代表，其余删除。这些代表构成了化简后的自动机的状态集合。对于原有的状态P和Q, 如果有P到Q的一个标记为T的边，那么有从P的代表到Q的代表有一个标记为T的弧（有可能是圈）。
- 步骤3：删除死状态和无用状态。

# DFA化简

- 得到的状态转换图仍然是DFA吗？为什么？
- 开始状态为包含有开始状态的子集的代表。
- 原来为终止状态的，现在仍然是终止状态。

# DFA化简（分划算法）

- 步骤1：初始分划：终止状态和非终止状态。
- 步骤2：重复对于每一组G都进行下列细分，直到不能再细分为止：
  - 将G分成子组，使得P,Q在一组当且仅当对于任何的输入符号，他们的后继都属于同一个小组。
  - 将子组加入到分划中替换G。
- 注意：前面发现的不能细分的小组后来可能还可以细分。所以重复步骤2的时候，需要检验所有的组，包括老的和新加入的。

# DFA化简（例子）

- 图3-9(b)
- $\{4\}$ ,  $\{0,1,2,3\}$
- $\{4\}$ ,  $\{0,1,2\}$ ,  $\{3\}$
- $\{4\}$ ,  $\{0,2\}$ ,  $\{1\}$ ,  $\{3\}$

0, 2

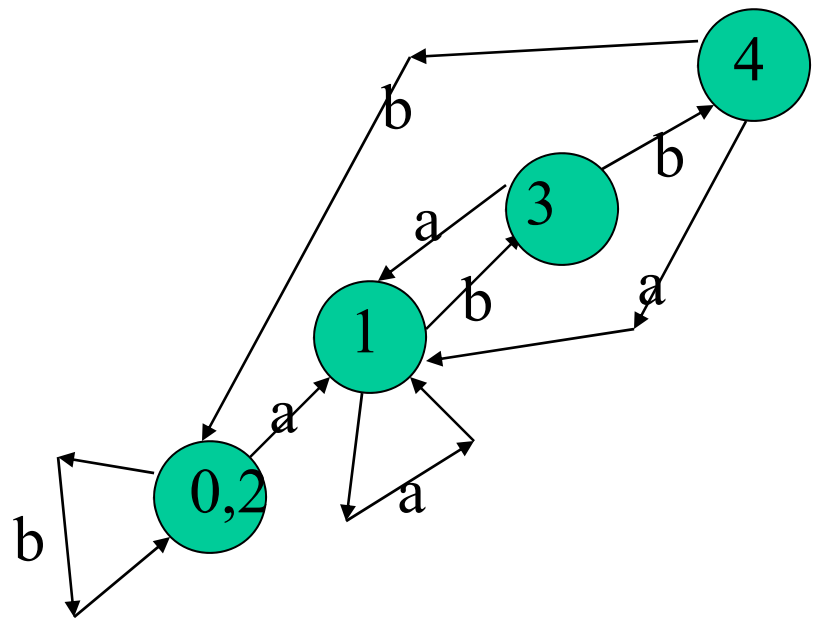
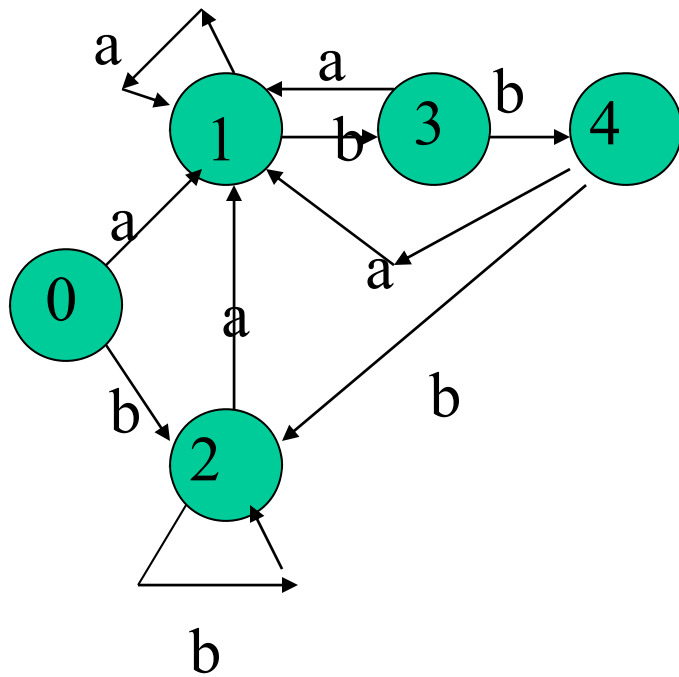
1

3

4



# DFA化简（例子）



# 正则表达式

- 引入的原因:
  - 词法规则简单，容易理解
  - 更加容易构造识别程序
  - 可以用于一些信息流的处理

# 正则表达式的定义(语法)

- 字母表 $\Sigma$ 上的正则表达式的定义如下：
  - $\varepsilon$ 与 $\Phi$ 都是 $\Sigma$ 上的正则表达式。
  - $\Sigma$  中的 $a$ 是正则表达式。
  - 如果 $e_1$ 和 $e_2$ 是 $\Sigma$ 上的正则表达式，则 $(e_1)$ ,  $e_1|e_2$ , 和  $\{e_1\}$  都是 $\Sigma$ 上的正则表达式。
- 例如：  $\Sigma_1=\{0,1\}$ ， 那么 $(0|1)\{0|1\}$ 就是 $\Sigma_1$ 上的正则表达式。

# 正则表达式的定义(语义)

- 定义3.12      正则表达式 $e$ 的值是字母表上的正则集合，用 $|e|$  表示。
  - $|\Phi| = \Phi$                        $|\varepsilon| = \{\varepsilon\}$
  - $|a| = \{a\}$                        $|(e)| = |e|$
  - $|e_1 e_2| = \{xy \mid x \in |e_1|, y \in |e_2|\}$
  - $|e_1 | e_2| = |e_1| \cup |e_2|$
  - $|\{e\}| = |e|^*$

# 正则表达式的等价

- 定义：如果两个表达式的正则集合相同，称这两个表达式等价。
- 例子：
  - $b\{ab\} = \{ba\}b$
  - $\{a|b\} = \{\{a\}\{b\}\}$

# 正则表达式的性质

- 零正则表达式 $\Phi$ :

$$- e | \Phi = \Phi | e = e \qquad e \Phi = \Phi e = \Phi$$

- 单位正则表达式:

$$- \varepsilon e = e \varepsilon = e$$

- 交换律:  $e_1 | e_2 = e_2 | e_1$

- 结合律:

$$- e_1 | (e_2 | e_3) = (e_1 | e_2) | e_3 \qquad e_1 (e_2 e_3) = (e_1 e_2) e_3$$

- 分配律:

$$- e_1 (e_2 | e_3) = e_1 e_2 | e_1 e_3 \qquad (e_1 | e_2) e_3 = e_1 e_3 | e_2 e_3$$

# 正则表达式与有穷状态自动机

- 正则表达式和有穷自动机的表达能力是相同的。具体表现在：
  - 给定一个正则表达式，可以构造出相应的有穷自动机。该自动机接受的符号串的集合恰巧是正则表达式的值
  - 给定一个有穷状态自动机，可以构造出相应的正则表达式。该正则表达式的值恰巧是该自动机接受的符号串的集合

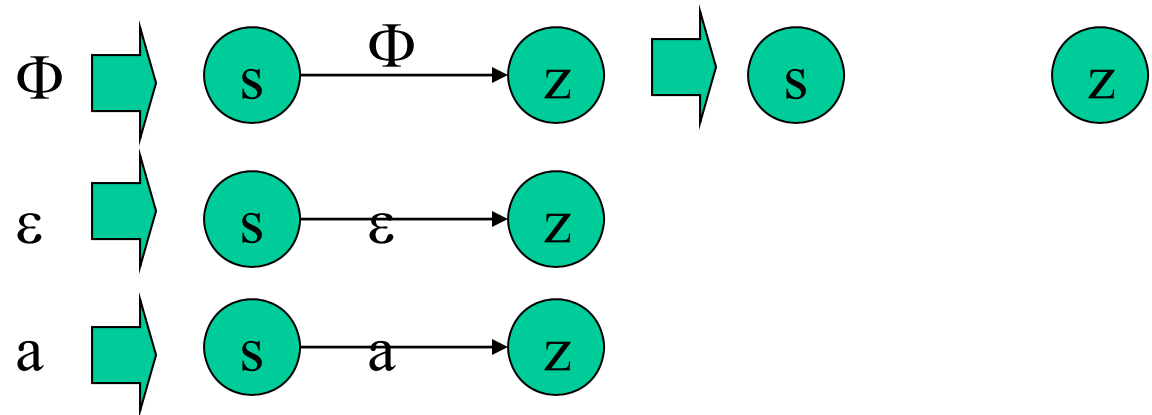
# 从正则表达式到自动机

- 步骤1：从正则表达式到转换系统。
  - 步骤2：从转换系统到状态转换图。
  - 步骤3：从状态转换图到NFA。
  - 步骤4：从NFA到DFA。
- 
- 其中，步骤3和步骤4的算法已经讲过。  
下面讲步骤1和步骤2。



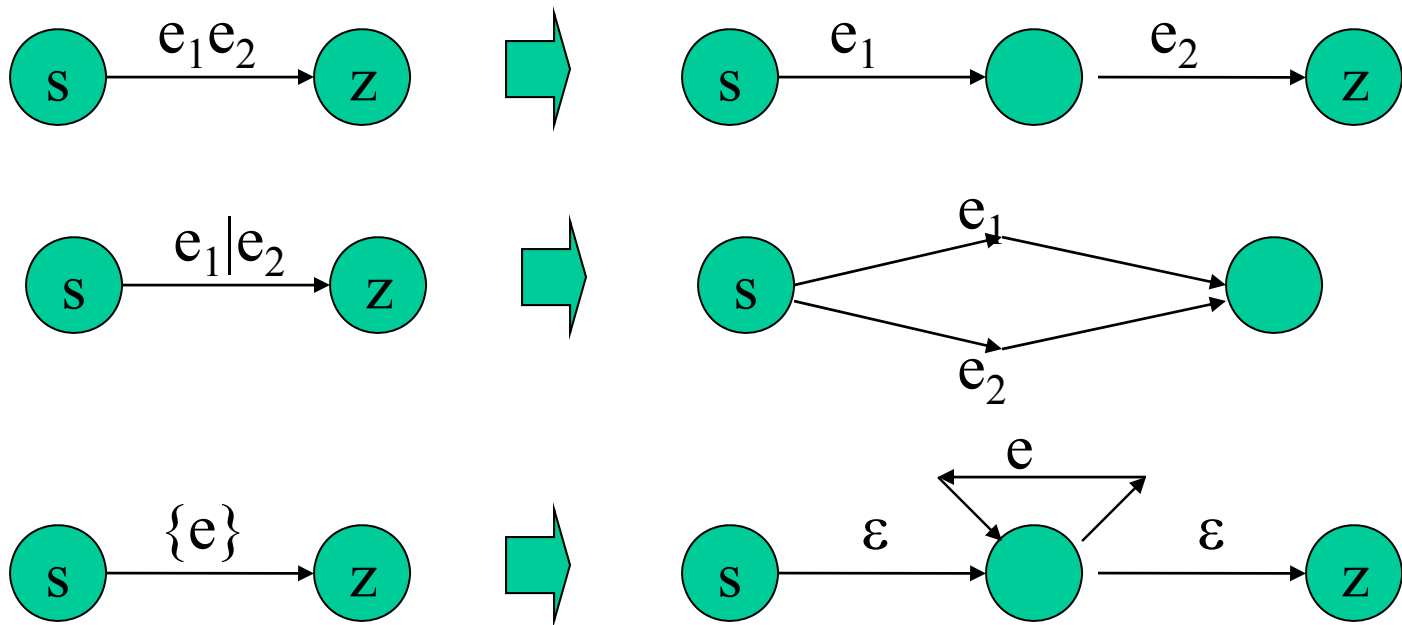
# 从正则表达式到转换系统

- 根据正则表达式的语法，使用自顶向下的构造方法。
- 基本部分：



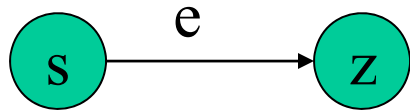
# 从正则表达式到转换系统

- 结构部分:



# 从正则表达式到自动机

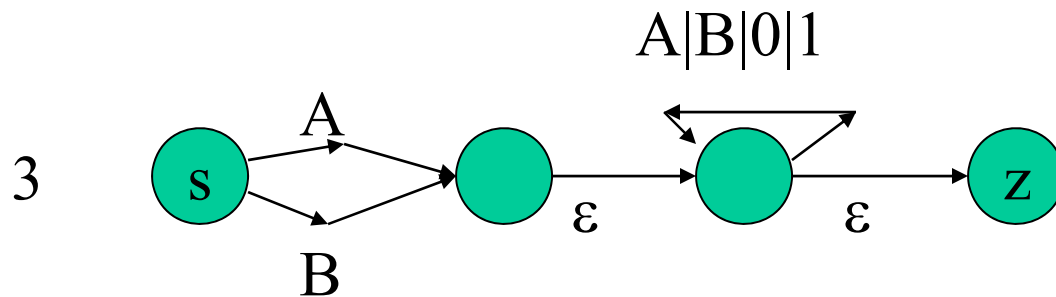
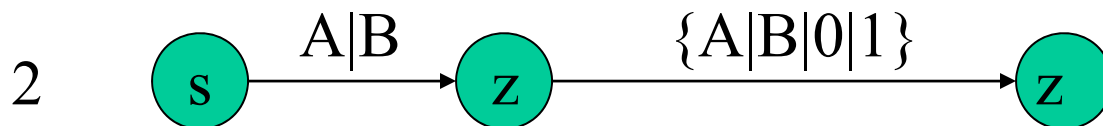
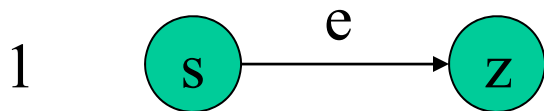
- 具体步骤：对于任意的一个正则表达式 $e$ ，从



开始，按照结构部分的规则分解，最终可以得到一个转换系统。对于引入的每一个新状态，应该赋予一个独有的名字。

# 例子

- $e = (A|B)\{A|B|0|1\}$



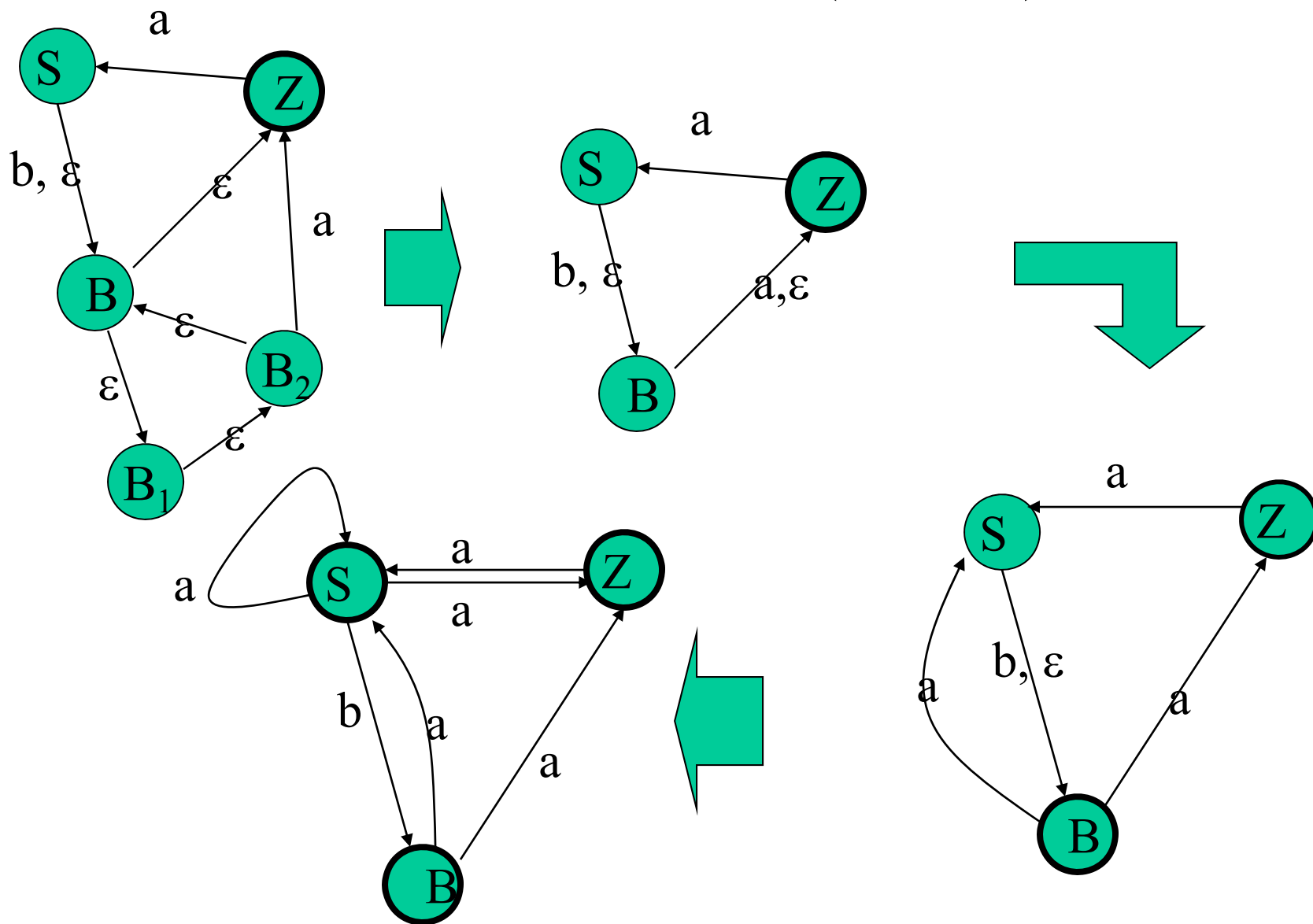
# 从转换系统到状态转换图

- 关键是消去系统中的空弧。
- 基本思想：
  - 如果存在一个空弧构成的圈，那么圈中的所有节点等价，可合并。
  - 对于任何一个如下的组合： $A - \varepsilon \rightarrow B - a \rightarrow C$ ，取代以 $A - a \rightarrow C$ 。

# 消空弧算法

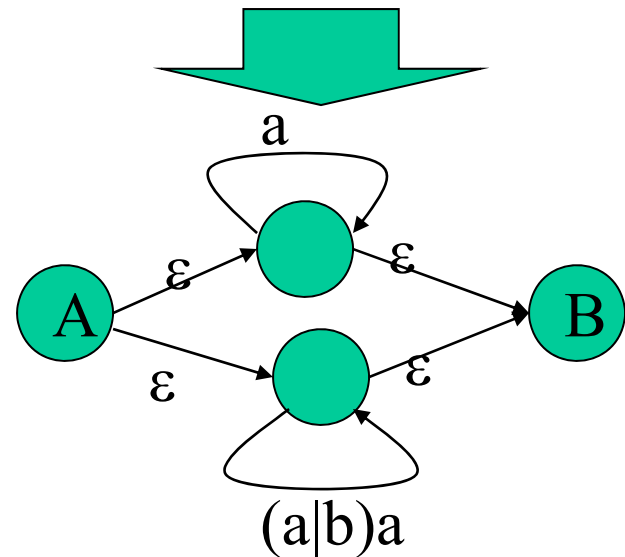
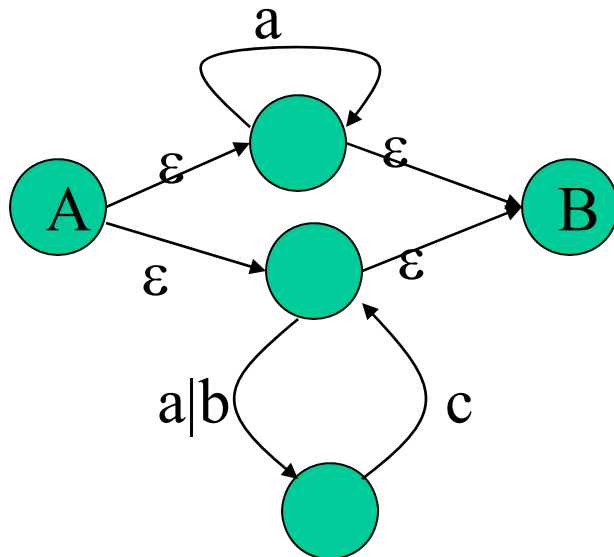
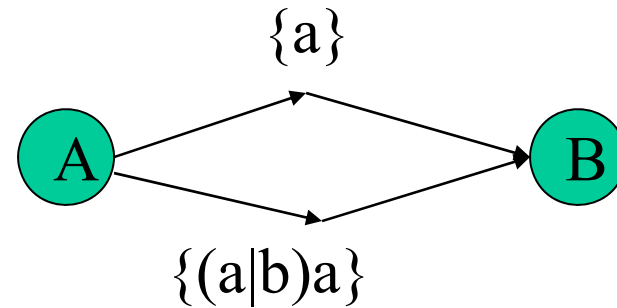
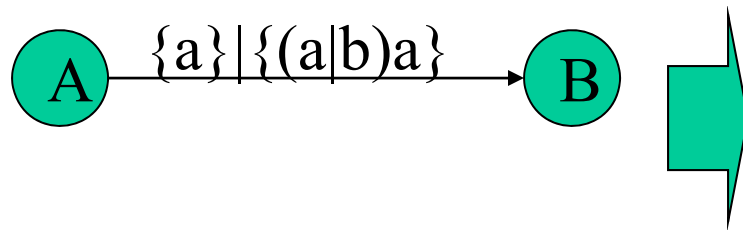
- 步骤1：从某个有空弧射出的状态开始，沿空弧前进，试图找到再没有空弧射出状态。若能找到，执行步骤2，否则到步骤3。
- 步骤2：设没有空弧射出的节点为B，而A有空弧到B。删除该空弧。对于B到C的非空弧，添加同样的A到C的弧。
- 步骤3：此时必然存在一个空弧圈，将圈中的节点合并得到新的节点。所有进入该圈的边都有相应到该点的边，所有离开该点的边都有相应离开该点的边。

# 消空弧算法(例子)



# 例子3.16

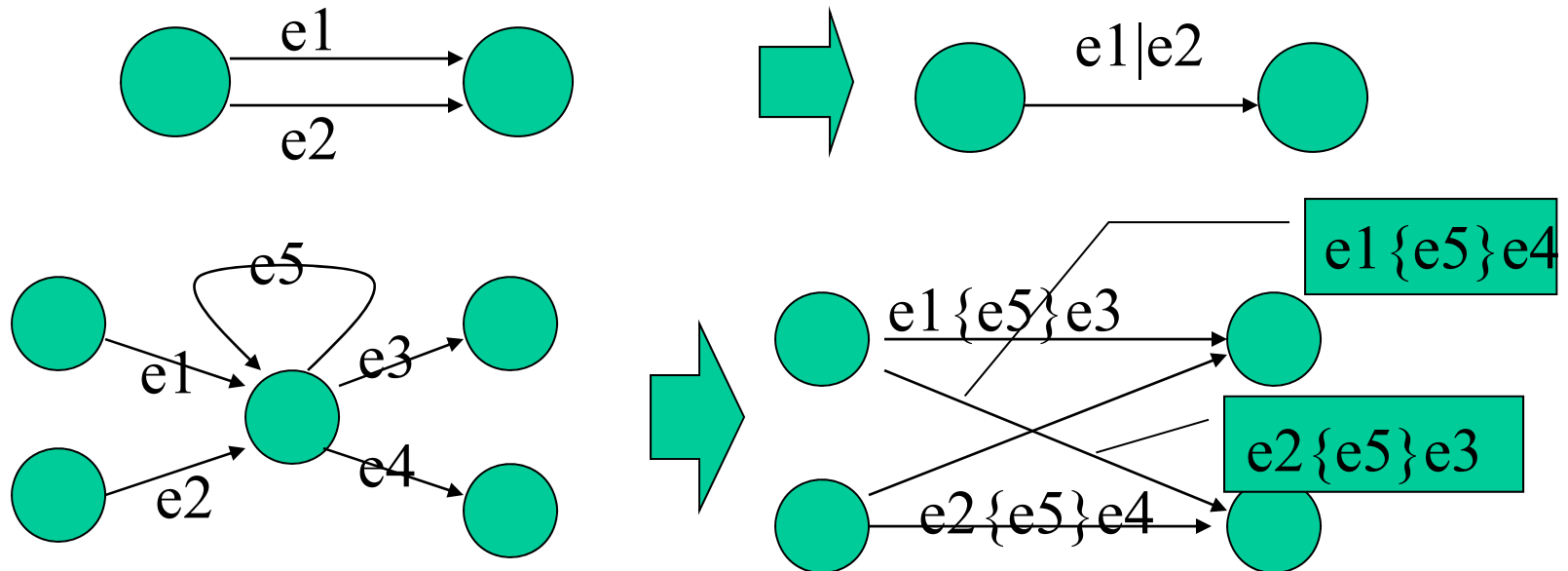
- $\{a\} | \{(a|b)a\}$





# 从自动机到正则表达式

- 首先获得一个转换系统。然后逐次按照以下规则消除节点，直到只剩下开始状态和接受状态。此时，唯一的边上的标记就是所要的表达式。



# 从自动机到正则表达式

- 在按照第二步骤进行处理时
  - 只能删除非接收状态和非开始状态的节点。
  - 对于所删除的节点，每个射入的弧和每个射出的弧要一一配对。每对这样的弧要对应一个新的弧。
- 最后得到的扩展自动机为：有一个开始状态，一个接受状态。两个状态之间只有一条边。

# 由正则文法直接构造表达式

- 使用联立方程的方法。
- 对于  $A = A(x_1 + x_2 + \dots + x_m) + (y_1 + y_2 + \dots + y_n)$  的情况，

$$A = (y_1 + y_2 + \dots + y_n) \{ (x_1 + x_2 + \dots + x_m) \}$$

- 然后，逐次将解得到的值代入。

# 直接构造表达式（例子）

- $S ::= Sc | Bc$        $B ::= Bb | Ab$        $A ::= Aa | a$
- $S = Sc + Bc$        $B = Bb + Ab$        $A = Aa + a$
- $A = a \{a\}$
- $B = Bb + a \{a\} b \Rightarrow B = a \{a\} b \{b\}$
- $S = Sc + a \{a\} b \{b\} c \Rightarrow S = a \{a\} b \{b\} c \{c\}$

## 直接构造表达式（例子2）

- $Z ::= Z0 | T1 | 1 | 0 \quad T ::= Z0 | 0$
- (1)  $Z = Z0 + T1 + 1 + 0 \quad (2) T = Z0 + 0$
- (2)  $\rightarrow$  (1)  $Z = Z0 + Z01 + 01 + 1 + 0$
- $Z = Z(0 + 01) + (01 + 1 + 0)$
- $Z = (01 | 1 | 0) \{ (0 | 01) \}$

# 词法分析程序的实现

- 属性字的一般结构:
  - 符号类:符号值
  - 要求不同的符号值有唯一的表示，一般要求有同样的长度。
- 词法分析程序分成处理源程序说明和不处理源程序说明两个类型。两种情况下的属性字有所不同。

# 不处理说明部分时

- 词法分析工作相对简单，不需要让说明信息和标识符相联系，属性字结构简单。
- 属性字由符号类和符号值组成。
  - 特定符号类中，一类符号只有一个值，因此，属性字中符号值部分是不使用的。
  - 非特定符号类中，一个类包含多个符号。比如：标识符，常整数等。这些类型的符号的属性字中的属性值部分需要加入值。
  - 由于属性字是等长的，所以，标识符的值不能直接加入到属性字中，一般使用标识符的编号。

# 符号类的编码

- 对于特定的语言，我们可以根据情况总结出相应的符号类别。在表3-4中，有符号类的编号（不需要记住）。



# 属性字序列例子

PROGRAM	exp	24 'PROGRAM'
VAR x, AB, C: integer;		1, 'exp'
BEGIN		20, ';'
x:=(AB+C**4)/8		25, 'VAR'
END		1, 'x'
		21, ', '
		1, 'AB'
		... ..

# 其它的编码方式

- 我们可以在属性字中加入一些附加信息以帮助其后的语法分析程序。
- 这些信息可以包括比如：
  - 是否说明符号
  - 是否运算符号
  - 运算优先级信息
  - 其它可以简单表示的信息。

# 词法分析中使用的数据

- 字符表：（字母表）列出源程序中所有可能的字符。
- 特定符号与机内表示表：一切特定符号与相应编码。
- 标识符表：登录一切源程序中出现的标识符。此表的序号作为属性字的值。
- 常数表：登录一切源程序中出现的常数。此表的序号作为属性字的值。

# 处理说明部分时

- 处理说明部分的时候，不仅仅要识别出各个符号，而且要将标识符与类型等信息相联系。
- 对于非特定符号的属性字，其结构大致如下：  
0 | 种类 | 特征 | 符号值
- 一般来将，一个标识符可以是：常量，简单变量，数组，记录等等。我们需要将这些信息记录在属性字的种类或特征中。
- 书中的表示方法不需要硬记，但是可以借鉴（如果你真的写编译程序的话）

# 标识符的处理

- 如果词法分析处理程序的说明的话，标识符的处理比较复杂，主要要考虑到如下问题：
  - 定义性出现与使用性出现
  - 标识符的作用域
  - 标识符符号值的确定

# 定义性出现与使用性出现

- 在第一次扫描到一个标识符的时候，需要为这个标识符构造属性字。此后遇到这个标识符的时候，只需要复制这个标识符就可以了。
- 定义性出现时：为标识符构造属性字。
- 使用性出现时：查找标识符的属性字并进行复制。

# 定义性出现与使用性出现

- 定义性出现：当源程序中某标识符在说明部分被首次说明而与类型等属性相关联时，成为该标识符的定义性出现。
- 不是定义性出现的都是标识符的使用性出现。
- 并不是标识符出现在说明部分的时候都是定义性出现。

## 例子3.21

- 标识符表在P87表3-8中。表中的属性字的符号值部分直接写了符号本身。实际上将是关于此标识符的信息，比如：A的属性字将表示其数组的信息（维数，上界...）
- 表3-9中列出了程序对应的符号属性字序列。但是其中没有说明部分。因为说明部分的信息已经包含在属性字中。



# 标识符的作用域

- 一般的现在程序设计语言都有标识符作用域的规定。PASCAL，C中有局部变量，C++中还有类的成员变量。
- 当相同拼写的标识符在内层被重新定义的时候，内外层的标识符被认为是不同的。并且，在内层的使用性说明以内层的为准。

# 标识符的作用域（处理方法）

- 在开始扫描的时候，设置标识符栈为空，包含一个间隔标记。初始时刻最外层为当前层。
- 碰到标识符的定义性出现的时候，构造属性字，下推入栈。
- 进入新的层次的时候，（C中为{）将一个间隔入栈。
- 碰到层次结束标记的时候，退栈直到将最上层的间隔。

# 标识符的作用域（处理方法续）

- 当碰到标识符的使用性出现的时候，从栈顶往下查找标识符。首先查找到的标识符记录中相应的属性字就是该标识符的属性字。如果直到栈底都没有找到，则表示程序出错。
- 按照上述的法则扫描程序，直到源程序扫描完毕。

# 标识符的作用域（例子）

- P88页程序。
- 对于函数或过程的说明中，注意函数名的标识符，以及参数标识符的层次。

间隔
exp2
x
y
f1
间隔
p1
s

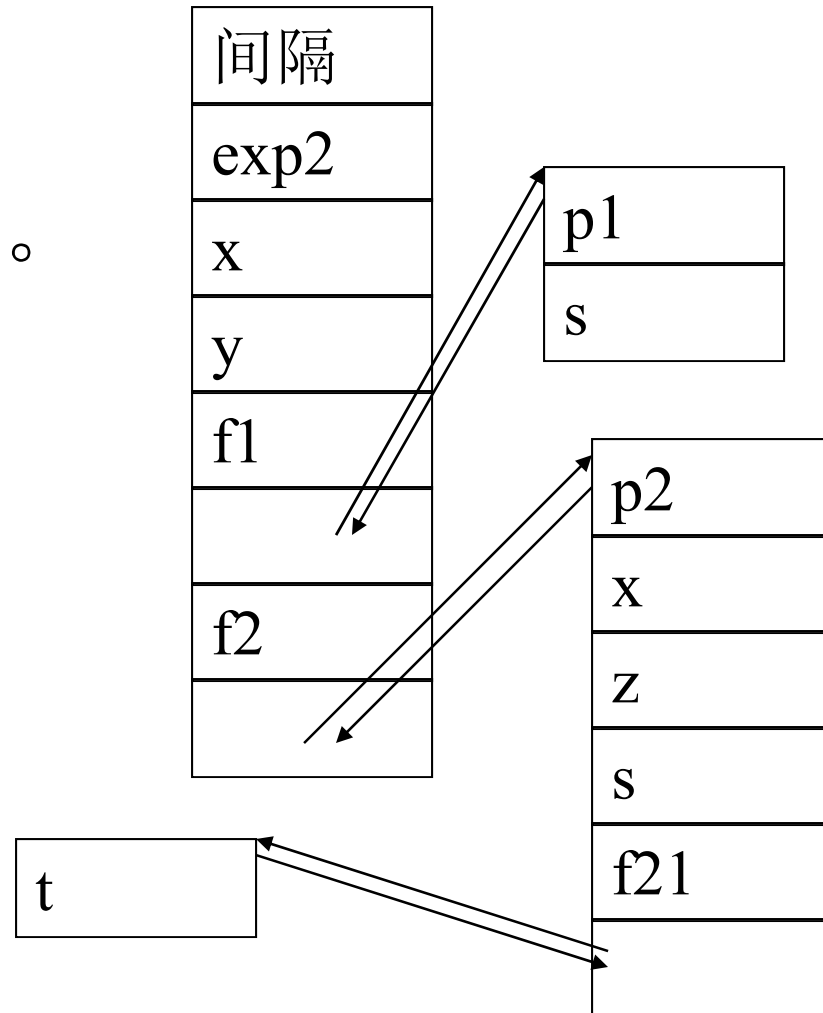
间隔
exp2
x
y
f1

间隔
exp2
x
y
f1
f2
间隔
p2

...

# 标识符的作用域（例子）

- 在语法和语义分析阶段都要使用标识符表的信息。所以，上述的退栈是有问题的。具体实现的时候可以建立树型结构来记录这些标识符的作用域。



# 标识符属性字的确定

- 不处理说明的情况：属性字中的符号值就是标识符表中的序号。标识符的类型等信息在语义分析阶段获得。
- 处理说明的情况：词法分析输出的属性字序列部分没有程序说明部分。此时：
  - 简单变量：在属性字中编码表示。
  - 常量：符号值用常量表序号表示。
  - 其它类型：以相应信息表序号表示。

# 词法分析程序的结构

- 取字符子程序
- 取符号子程序，一般有如下约定：
  - 进入子程序时，已经取到当前符号的一个字符。
  - 离开子程序时，已经取到其后继字符。
- 查造标识符表子程序
- 查造常量表子程序
- 查符号机内表示对照表子程序

# 词法分析程序的自动生成（Lex）

- Lex是Unix(Linux)系统中的一个实用程序。
- Lex的输入称为Lex程序，主要定义了一些词法规则。Lex程序的输出是一个可以识别这些单词的C语言子过程。
- 输入分成说明，规则，和辅助过程三个部分。
  - 说明：用正则表达式定义了词法规则。
  - 翻译规则：表示了当程序识别到一个词型时所需要完成的动作。
  - 辅助过程用C语言书写了一些过程，被识别过程调用。



# Lex输入文件的例子（1）

- 说明部分

```
% {  
/* need this for the call to atof() below */  
#include <math.h>  
%}  
DIGIT      [0-9]  
ID          [a-z][a-z0-9]*  
%%
```

# Lex输入文件的例子（2）

- 规则部分
- `{DIGIT}+ { printf( "An integer: %s (%d)\n",  
yytext, atoi( yytext ) ); }`
- `{DIGIT}+"." {DIGIT}* {  
printf( "A float: %s (%g)\n", yytext,  
atof( yytext ) ); }`
- `if|then|begin|end|procedure|function {  
printf( "A keyword: %s\n", yytext ); }`
- `{ID} printf( "An identifier: %s\n", yytext );`
- `"+"|"-"|"*"|"/" printf( "An operator: %s\n",  
yytext );`
- `"{"[^}\n]*}" /* eat up one-line comments */`
- `[ \t\n]+ /* eat up whitespace */`
- `. printf( "Unrecognized character: %s\n",  
yytext );`
- `%%`

# Lex输入文件的例子（3）

- 辅助过程部分

```
int    yywrap() { return 1; }
main( argc, argv )
int argc;
char **argv;
{
    yyin = fopen( argv[0], "r" );
        ...      ...      ...
    yylex();
}
```

# 编译原理讲义

## (第四章:语法分析-- 自顶向下分析技术)

南京大学计算机系

赵建华

# 引言

- 在词法分析完成之后，进入语法分析阶段。
- 语法分析阶段的任务是：检查程序的语法是否正确，并生成内部中间表示形式。
- 语法分析的
  - 输入：属性字序列。
  - 输出：程序的内部中间表示形式。

# 自顶向下分析与识别算法

- 从推导的角度看，从识别符号出发，试图推导出与输入符号串相同的符号串。一般来讲，构造出的推导是最左推导。
- 从语法树的角度看，从根节点，试图向下一个语法树，其末端节点正好与输入符号串相同。

# 讨论前提

- 输入的是符号序列，不对符号构造情况感兴趣。
- 语法分析的文法是上下文无关文法。
- 自顶向下分析技术的理论基础是定理2.7:  
如果 $Z ::= X_1 X_2 \dots X_n$ 且 $y$ 为句子。那么如果 $X_1 X_2 \dots X_n \Rightarrow y$ ，必然存在 $y_1, y_2, \dots, y_n$ 使得 $X_i \Rightarrow^* y_i$ 且 $y = y_1 y_2 \dots y_n$ 。

# 要解决的基本问题

- 对于特定的终结符号，使用哪个重写规则来替换？



# 无回溯的自顶向下分析技术

- 先决条件:
- 无递归
  - 既没有规则左递归，也没有文法左递归。
- 无回溯性
  - 对于任一非终结符号 $U$ 的规则右部 $x_1|x_2|\dots|x_n$ ，其对应的字的头终结符号两两不相交。

# 递归下降分析技术（实现思想）

- 实现思想：
- 识别程序由一组过程组成。每个过程对应于一个非终结符号。
- 每一个过程的功能是：选择正确的右部，扫描完相应的字。在右部中有非终结符号时，调用该终结符号对应的过程来完成。

# 基本架构(1)

- 对于每个非终结符号 $U ::= u_1 | u_2 | \dots | u_n$ 处理的方法如下：

$U()$ {

    ch=当前符号

    if(ch可能是 $u_1$ 字的开头)          处理 $u_1$ 的程序部分

    else if(ch可能是 $u_2$ 字的开头)      处理 $u_2$ 的程序部分

    ...

    else    error()

}

# 基本架构(1)

- 对于无回溯的文法保证选择是唯一的。
- 但有某个右部的开头是非终结符号时，需要用LL(K)方法判断什么时候使用这个右部。
- 当存在空规则的时候，可以把error()替代为{return;}，这样的处理使发现错误的情况比较晚。
- 约定：进入这个过程时，对于U的第一个符号已经被读到当前符号。离开这个程序的时候，下一个符号已经被读到当前符号。

## 基本架构(2)

- 对于每个右部 $u1=x_1x_2\dots x_n$ 的处理架构如下：
  - 处理 $x_1$ 的程序；
  - 处理 $x_2$ 的程序；
  - ... ..
  - 处理 $x_n$ 的程序；
- 如果右部为空，则不处理。

# 基本架构(3)

- 对于右部中的每个符号 $x_i$
- 如果 $x_i$ 为终结符号：  
    if( $x ==$  当前的符号)  
        {NextCh(); return;}  
    else  
        出错处理
- 如果 $x_i$ 为非终结符号，直接调用相应的过程：  
     $x_i()$

# 递归下降技术（实例）

- 文法G4.3

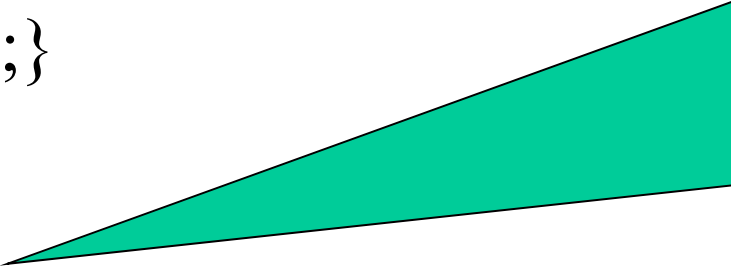
$$E ::= E + T \mid T \quad T ::= T * F \mid F \quad F ::= (E) \mid i$$

- 消左递归得到

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

# 递归下降技术（实例续）

- 对应于文法G4.3'中的每个非终结符号，都有一个过程。
- E()
  - {
  - if(当前符号可能是T的开始符号)
  - { T(); E1();}
  - else
  - error();
  - }



和书上不同的是，我们作了出错处理。一般，当只有一个右部的时候，可以不作出错处理。



# 递归下降技术（实例续）

- E1()

```
{  
    if(ch='+')  
    {  
        getnextchar(); T(); E1();  
    }  
    else  
        return;  
}
```

因为E1对应有空规则，所以其处理中，不作错误处理，而是直接return。实际表示它没有读入任何字符。

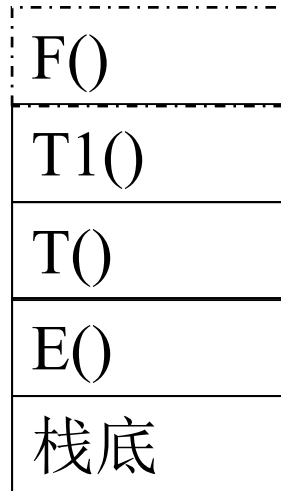
# 递归分析程序的运行

输入:  $i * i + i$



处理完\*, 当前  
符号为i

过程调用栈:



# 递归分析程序的主程序

- 假设识别符号对应的过程为Z(), 那么相应的主程序为

```
{
```

```
    GetSymbol(); //首先需要读入一个符号，以满足约定。
```

```
    Z();
```

```
    检查是否达到输入的结尾.
```

```
}
```

# 递归分析程序的优点

- 实现思想简单明了。程序结构和语法规则有直接的对应关系。
- 因为每个过程表示一个非终结符号的处理，添加语义加工工作比较方便。
- 需要书写程序的语言支持递归调用。如果递归调用机制是高效的，那么分析程序也是高效的。

# 预测分析法

- 使用下推自动机的方式实现。
- 使用一个二维分析表和栈联合进行控制来实现递归下降分析技术。
- 分析表A实际上是一个从 $V_N \times (V_T \cup \{\#\})$ 到规则的映射。当自顶向下分析过程中需要将U展开时，如果当前符号为T时，应该选择规则 $A[U, T]$ 。如果 $A[U, T]$ 为空，表示输入符号串不正确。

# 预测分析过程

- $PUSH(S, \#)$ ;  $PUSH(S, Z)$ ;
- $a = \text{下一个符号}$ ;  $X = TOP(S)$ ;
- 如果 $X$ 为终结符号且 $a == X$ 且 $a != \#$ ,  $POP(S)$ ;  
 $a = \text{下一个符号}$ 。
- 如果 $X$ 为终结符号且 $a == X$ 且 $a == \#$ , 分析结束, 接受句子。
- 如果 $X$ 为终结符号且 $a != X$ , 出错处理。

# 预测分析过程

- 如果 $X$ 为非终结符号且 $A[X,a]$ 为报错标识, 出错处理。
- 如果 $X$ 为非终结符号且 $A[X,a]='X = X_1 X_2 \dots X_{n1}$ ,
- { POP( $S$ );  
将右部 $X_n \dots X_2 X_1$ 压入 $S$ 中。
- }

# 例子

- 文法G4.3'[E]

$E ::= TE'$        $E' ::= +TE' \mid \varepsilon$        $T ::= FT'$

$T' ::= *FT' \mid \varepsilon$        $F ::= (E) \mid i$



# 例子

	i	+	*	(	)	#
E	TE'			TE'		
E'		<u>+TE'</u>			$\epsilon$	$\epsilon$
T	FT'			FT'		
T'		$\epsilon$	FT'		$\epsilon$	$\epsilon$
F	i			(E)		

#E	i+i*i#
#E'T	i+i*i#
#E'T'F	i+i*i#
#E'T'i	i+i*i#
#E'T'	+i*i#
#E'	+i*i#
#E'T+	+i*i#
#E'T	i*i#
#E'T'F	i*i#
#E'T'i	i*i#
#E'T'	*i#

# 预测分析表的生成

- 从前面的论述我们看到，预测分析过程的驱动程序时固定的。对于某个文法，分析表是分析过程的核心。
- 表中 $A[U, T] = 'U ::= X_1 X_2 \dots X_n'$ 表示对应于 $X_1 X_2 \dots X_n$ 字的首符号可以是 $T$ 。就是说 $X_1 X_2 \dots X_n \Rightarrow^* Tw$ 。我们可以通过这个方式来确定分析表中的值。

# 预测分析表的生成

- 一般来讲，对于一个符号串 $X_1X_2\dots X_n$ 的字的第一个终结符号就是 $X_1$ 对应的字的第一个终结符号。但是空规则的存在使情况有一点复杂。
- 对于 $U_1U_2\dots U_n$ ，如果 $U_1\Rightarrow^* \varepsilon$ ，那么符号串对应的字的首符号也可以是 $U_2$ 对应的字的首符号。计算一个符号串对应的字的首符号的算法也需要考虑到这些。

# FIRST(u)和FOLLOW(U)

- $\text{FIRST}(u) = \{T \mid u \Rightarrow^* T \dots, T \in V_T\}$ , 如果  $u \Rightarrow^* \varepsilon$ , 那么, 我们规定  $\varepsilon \in \text{FIRST}(u)$ 。
- $\text{FOLLOW}(U) = \{T \mid Z \Rightarrow^* \dots UT \dots, T \in V_T \cup \{\#\}\}$   
其中, 如果  $Z \Rightarrow^* \dots U$ , 那么  $\# \in \text{FOLLOW}(U)$
- 直观地讲:
  - $\text{FIRST}(u)$  包含了  $u$  对应的字的所有可能的首终结符号。
  - $\text{FOLLOW}(U)$  表示了句型中可能紧跟再  $U$  后面的终结符号

# FIRST(u) 构造算法

- 对于文法 $X$ 构造 $\text{FIRST}(X)$ 
  - 步骤1: 如果 $X \in V_T$ , 那么 $\text{FIRST}(X) = \{X\}$
  - 步骤2: 如果 $X \in V_N$ , 且有规则 $X ::= T \dots$ , 那么将 $T$ 添加到 $\text{FIRST}(X)$ 中。如果 $X ::= \varepsilon$ , 那么 $\varepsilon$ 也在 $\text{FIRST}(X)$ 中。
  - 步骤3: 对于规则 $X ::= X_1 X_2 \dots X_n$ , 把 $\text{FIRST}(X_1)$ 中的非 $\varepsilon$ 符号添加到 $\text{FIRST}(X)$ 中。如果 $\varepsilon$ 在 $\text{FIRST}(X_1)$ 中, 把 $\text{FIRST}(X_2)$ 中的非 $\varepsilon$ 符号添加到 $\text{FIRST}(X)$ 中...; 如果 $\varepsilon$ 在 $\text{FIRST}(X_n)$ 中, 把 $\varepsilon$ 添加到 $\text{FIRST}(X)$ 中。
- 对于 $\text{FIRST}(u)$ , 可是使用类似于步骤3的方法求解得到。

# FIRST的例子

- 文法G4.3'[E]:

$E ::= TE'$                        $E' ::= +TE' \mid \varepsilon$                        $T ::= FT'$

$T' ::= *FT' \mid \varepsilon$                $F ::= (E) \mid i$

- $\text{FIRST}(F) = \{ (, i \}$
- $\text{FIRST}(T) = \text{FIRST}(F) = \{ (, i \}$
- $\text{FIRST}(E') = \{ +, \varepsilon \}$      $\text{FIRST}(T') = \{ *, \varepsilon \}$

...              ...              ...              ...

# FOLLOW(U)的构造算法

- 步骤1      $\# \in \text{FOLLOW}(Z)$
- 步骤2     如果有规则  $U ::= xWy$ ，那么  $\text{FIRST}(y)$  中所有的非 $\epsilon$ 符号都在  $\text{FOLLOW}(W)$  中。
- 步骤3     如果有规则  $U ::= xW$  或则  $U ::= xWy$  且  $\epsilon \in \text{FIRST}(y)$ ，那么  $\text{FOLLOW}(U)$  中的一切符号都在  $\text{FOLLOW}(W)$  中。
- 注意：步骤3需要重复执行，直到没有哪个非终结符号的  $\text{FOLLOW}$  集合增长为止。

# FOLLOW例子

- 文法G4.3'[E]:

$$E::=TE' \quad E'::=+TE' \mid \varepsilon \quad T::=FT'$$

$$T'::=*FT' \mid \varepsilon \quad F::=(E) \mid i$$

- $\text{FOLLOW}(E) = \{\#, )\}$
- $\text{FOLLOW}(E') = \text{FOLLOW}(E) = \{\#, )\}$
- $\text{FOLLOW}(T) = \text{FIRST}(E') \cup \text{FOLLOW}(E) - \{\varepsilon\} = \{+, \#, )\}$
- $\text{FOLLOW}(T') = \text{FOLLOW}(T) = \{\varepsilon\} = \{+, \#, )\}$
- $\text{FOLLOW}(F) = \text{FIRST}(T') \cup \text{FOLLOW}(T) = \{+, \#, ), *\}$



# 预测分析表的构造

- 基本思想:
- 当我们需要将U选择某个规则展开时，如果当前的输入为a，表示我们要将U展开为以a为首符号的字。如果有规则 $U ::= u$ ，且 $a \in \text{FIRST}(u)$ ，那么表示这个规则是个好的选择。

# 分析表构造算法

- 对于每个规则 $U ::= u$ ，执行一下步骤
  - 对于每个终结符号 $a \in \text{FIRST}(u)$ ， $A[U, a] = 'U ::= u'$ 。
  - 如果 $\varepsilon \in \text{FIRST}(u)$ ，对于每个 $\text{FOLLOW}(U)$ 中的每个终结符号 $b$ 或 $\#$ ，让 $A[U, b] = 'U ::= u'$ 。
- 将其它未定义的分析表元素为ERROR。

# 分析表的例子

- 文法G4.3'[E]:

$E ::= TE'$

$E' ::= +TE' \mid \varepsilon$

$T ::= FT'$

$T' ::= *FT' \mid \varepsilon$

$F ::= (E) \mid i$

	i	+	*	(	)	#
E	TE'			TE'		
E'		TE'			$\varepsilon$	$\varepsilon$
T	FT'			FT'		
T'		$\varepsilon$	FT'		$\varepsilon$	$\varepsilon$
F	i			(E)		

# 分析表的冲突

- 文法G4.6[S]  $S ::= iCtSS' | a$      $S' ::= eS | \varepsilon$      $C ::= b$
- $FIRST(iCtSS') = \{i\}$      $FIRST(eS) = \{e\}$
- $FIRST(S) = \{i, a\}$      $FIRST(C) = \{b\}$
- $FIRST(S') = \{e, \varepsilon\}$
- $FOLLOW(S) = FOLLOW(S') = \{\#, e\}$

	a	b	e	i	t	#
S	a			iCtSS'		
S'			eS; $\varepsilon$			$\varepsilon$
C		b				

# LL(1)文法

- 定义：如果其预测分析表中没有多重定义的元素，则该文法被称为LL(1)文法。
- LL(1)文法是无二义性的。
- 定理4.1，文法G是LL(1)的，当且仅当每个非终结符号U的任何两个不同的重写规则 $U ::= x \mid y$ 满足如下条件：
  - $\text{FIRST}(x) \cap \text{FIRST}(y) = \emptyset$
  - $x \Rightarrow^* \varepsilon$  和  $y \Rightarrow^* \varepsilon$  不能同时成立
  - 如果  $y \Rightarrow^* \varepsilon$ ，那么  $\text{FIRST}(x) \cap \text{FOLLOW}(U) = \emptyset$

# 对于BNF表示法的处理

- 如果在规则的右部是使用BNF表示的，那么使用自顶向下技术进行处理时，需要作出相应的变化。
  - 预测分析法：修改文法，消除BNF表示。
  - 递归子程序法：处理规则右部的过程有所改变。

# 例子

- $E ::= T \{+T\}$
  - $F ::= i \mid (E)$
- $T ::= F \{*F\}$

# LL(1)的处理方法

- 修改方法：
  - 对于每个 $\{x\}$ ，引入新的非终结符号 $U ::= xU \mid \text{空}$ 。
  - 对于每个 $[x]$ ，引入新的非终结符号 $U ::= x \mid \text{空}$ 。
- 修改文法得到：
  - $E ::= T E'$                        $E' ::= +TE' \mid \text{空}$
  - $T ::= F T'$                        $T' ::= * F T' \mid \text{空}$
  - $F ::= i \mid (E)$
- 然后可以使用LL(1)技术来处理。



# 递归子程序法

- 理论上，可以和LL(k)方法处理BNF表达式同样处理。但是，递归子程序法可以使用更加直接的方法。
- 对于规则右部 $X_1X_2\ldots\{X_k X_{k+1}\}\ldots X_n$ 的处理方式为
  - $X_1$ 的处理;  $X_2$ 的处理; ...
  - while ( $X_k X_{k+1}$  循环继续? )
    - { $X_k$ 的处理,  $X_{k+1}$ 的处理}
  - ...;  $X_n$ 的处理
- 在判断是否继续循环的时候，判断的是，下一个符号究竟在 $\text{first}(X_k X_{k+1})$ 中，还是在 $\text{first}(X_{k+2} X_{k+3} \ldots)$ 中。

# 编译原理讲义

## (第五章:语法分析-- 自底向上分析技术)

南京大学计算机系

赵建华

# 概论

- 从输入符号出发，试图把它归约成识别符号。每一步都寻找特定得某个类型的短语（一般是简单短语）进行归约。
- 在分析过程中，每次归约的都是最左边的简单短语（或其它短语）。
- 从语法树的角度，以输入符号为树的末端结点，试图向根结点方向往上构造语法树。

# 基本问题

- 如何找出进行直接归约的简单短语？
- 将找到的简单短语归约到哪个非终结符号？

# 讨论前提

- 和自顶向下技术同样，不考虑符号的具体构成方式。
- 文法是压缩了的。
- 识别过程是从左到右，自底向上进行的。一般都采用规范归约：每一步都是对句柄进行归约（特例除外）。

# 基本方法

- 基本都采用移入-归约方法。
- 使用一个栈来存放归约得到的符号。
- 在分析的过程中，识别程序不断地移入符号。移入的符号暂时存放在一个栈中。一旦在栈中已经移入的（或者归约得到的）符号串中包含了一个句柄时，将这个句柄归约成为相应的非终结符号。

# 基本方法（续）

- 归约中的动作有4类
  - 移入：读入一个符号并把它归约入栈。
  - 归约：当栈中的部分形成一个句柄（栈顶的符号序列）时，对句柄进行归约。
  - 接受：当栈中的符号仅有#和识别符号的时候，输入符号也到达结尾的时候，执行接受动作。
  - 当识别程序觉察出错误的时候，表明输入符号串不是句子。进行错误处理。

# 例子

文法G5.1[E]:  $E ::= E + E \mid E * E \mid (E)$

输入符号串:  $i * i + i$

已处理符号	未处理符号	句型	句柄	规则
i	$*i+i$	$i*i+i$	i	$E ::= i$
E	$*i+i$	$E*i+i$		
$E*$	$i+i$	$E*i+i$		
$E*i$	$+i$	$E*i+i$	i	$E ::= i$
$E * E$	$+i$	$E * E + i$	$E * E$	$E ::= E * E$
E	$+i$	$E + i$		
$E + i$		$E + i$	i	$E ::= i$
$E + E$		$E + E$	$E + E$	$E ::= E + E$
E				

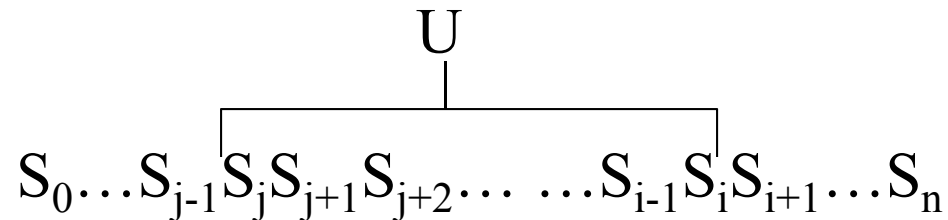


# 例子的解释

- 当栈中的符号的栈顶部分还不能形成句柄时，进行移入操作。
- 一旦发现栈顶部分形成了句柄的时候，对该句柄进行归约。将句柄出栈，然后将归约得到的非终结符号压栈。
- 如果输入是句子，则栈中的符号（从底到上）和未处理的符号组成句型。
- 在例子中，发现句柄和归约是人为干预的结果。所以移入-归约不是实际可运行的技术，而是技术的模板。

# 简单优先分析技术

- 基本思想：
  - 每次察看句型中相邻的两个符号。通过两个符号的关系判定出前一个符号是句柄的尾。然后，反向找出句柄的头。这样我们就找到了一个句柄。



# 简单优先分析技术(基本思想续)

- 我们要通过两个相邻符号 $S_i S_{i+1}$ 之间的关系来找到句柄：
  - $S_i S_{i+1}$ 在句柄内：必然有规则 $U ::= \dots S_i S_{i+1} \dots$
  - $S_i$ 在句柄内部，但是 $S_{i+1}$ 在句柄之后：必然有规则 $U ::= \dots S_i$ ，且存在规范句型 $\dots U S_{i+1} \dots$ 。
  - 如果 $S_{i+1}$ 在句柄内，而 $S_i$ 在句柄外，那么必然存在规范句型 $\dots S_i U \dots$ ，且 $U ::= S_{i+1} \dots$ 。

# 优先关系

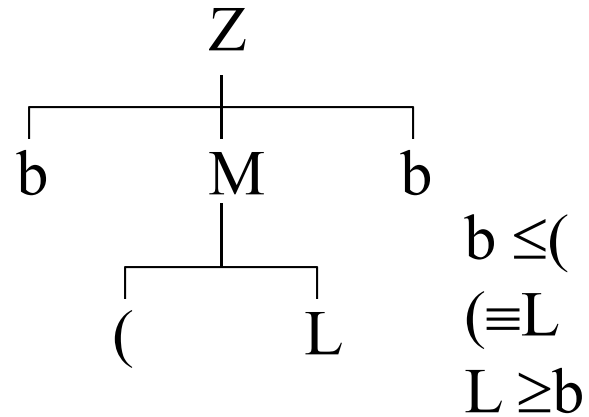
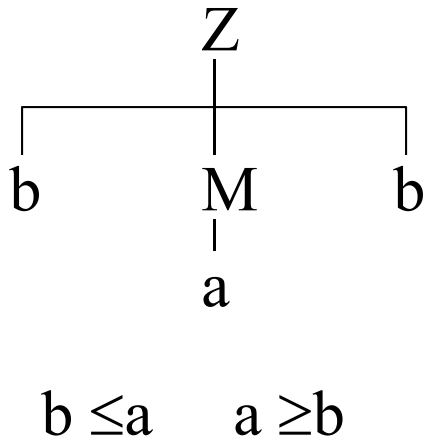
- 和书上的写法不一样，凑合用。

$$S_i \equiv S_j \quad S_i \geq S_j \quad S_i \leq S_j$$

- 注意：  $\equiv$ ，  $\geq$ ，  $\leq$  之间不同于  $=$ ，  $>$  和  $<$ 。  
由  $S_i \geq S_j$  不能导出  $S_j \leq S_i$ 。

# 优先关系的例子

- 文法:  $Z ::= bMb \quad M ::= (L|a \quad L ::= Ma)$
- 语言:  $\{bab, b(aa)b, b((aa)a)b, \dots\}$
- 可以从语法树里面导出 **部分** 优先关系。



# 优先矩阵

- 可以将优先关系填写到一个矩阵，得到优先矩阵。(将矩阵作为关系的表示形式)

	Z	M	L	a	b	(	)
Z							
M				=	=		
L				>	>		
a				>	>		=
b		=		<		<	
(		<	=	<		<	
)				>	>		

# 识别过程(例子)

- 文法: G5.2

$Z ::= bMb$

$M ::= (L \mid a \quad L ::= Ma)$

- 输入:  $b((aa)a)b$
- 过程:

# b ( ( a a ) a ) b #

< < < < >

句柄: a 归纳为M

# b ( ( M a ) a ) b #

< < < < = = >

句柄: M a) 归纳为L

# b ( ( L a ) b #

< < < = >

句柄: (L 归纳为M

# 识别过程(例子续)

# b ( M a ) b #

< < < = = >

句柄: Ma) 归纳为L

# b ( L b #

< < = >

句柄: (L 归纳为M

# b ( M a ) b #

< < < = = >

句柄: Ma) 归纳为L

# b ( L b #

< < = >

句柄: (L 归纳为M

# b M b #

< = = >

句柄: bMb 归纳为Z



# 优先关系的定义

- $S_j \equiv S_i$ : 当且仅当G中有规则  $U ::= \dots S_j S_i \dots$
- $S_j \leq S_i$ : 当且仅当  $U ::= \dots S_j V \dots$ , 且  $V$  HEAD+  $S_i$ ;
- $S_j \geq S_i$ : 当且仅当  $U ::= \dots V W \dots$ , 其中  $V$  和  $W$  分别满足  $V$  TAIL+  $S_j$  和  $W$  HEAD\*  $S_i$  且  $S_i$  为终结符号。

# HEAD和TAIL

- $U \underline{HEAD} S$  当且仅当  $U ::= S \dots$
- $U \underline{TAIL} S$  当且仅当  $U ::= \dots S$
- $\underline{HEAD}_+$ ,  $\underline{HEAD}^*$  分别是  $\underline{HEAD}$  的传递闭包和自反传递闭包。
  - $U \underline{HEAD}_+ S$  当且仅当  $U ::= U_1 \dots, U_1 ::= U_2 \dots, \dots, U_n ::= S \dots$ 。
  - $U \underline{HEAD}^* S$  当且仅当  $U = S$  或者  $U \underline{HEAD}_+ S$
- $\underline{TAIL}_+$  是  $\underline{TAIL}$  的传递闭包。
  - $U \underline{TAIL}_+ S$  当且仅当  $U ::= \dots U_1, U_1 ::= \dots U_2, \dots, U_n ::= \dots S$

# 优先关系

- 定理5.1  $S_j \equiv S_i$ 当且仅当 $S_j S_i$ 出现在某个规范句型的句柄中。
- 定理5.2  $S_j \geq S_i$ 当且仅当存在规范句型 $\dots S_j S_i \dots$ ，且 $S_j$ 是该句型的句柄的最后一个符号。
- 定理5.3  $S_j \leq S_i$ 当且仅当存在规范句型 $\dots S_j S_i \dots$ ，且 $S_i$ 是该句型的句柄的第一个符号。

# 定理5.1的证明

- 如果 $S_j S_i$ 出现在句柄中，当然有 $U ::= \dots S_j S_i \dots$
- 如果有规则 $U ::= u S_j S_i v$ ，
  - 必然有某个符号串的推导使用了该规则。（假设文法是已压缩的）
  - 根据该推导构造语法树。则必然存在一个结点标记为 $U$ ，其子结点为 $u S_j S_i v$ 。
  - 根据语法树构造最左归约（每一步都对句柄进行归约）。那么上面的结点对应的一步就是将 $u S_j S_i v$ 归约为 $U$ 。此时， $u S_j S_i v$ 为句柄。

# 定理5.2的证明(1)

- 如果  $S_j \geq S_i$ ,
  - 按照定义存在规则  $U ::= \dots VW \dots$ ,  $V \text{ TAIL}^+ S_j$ , 且  $W \text{ HEAD}^* S$ 。
  - 必然有句型  $\dots VW \dots$ , 由  $\text{TAIL}^+$  和  $\text{HEAD}^*$  的定义, 必然存在句型  $\dots vuS_j S_i \dots$ , 且  $V \Rightarrow^* vU \Rightarrow vuS_j$ ,  $W \Rightarrow S_i \dots$ 。
  - 构造相应的语法树, 并且从语法树构造相应的规范推导。 $uS_j$  必然在某一步成为句柄。而此时由于  $S_i$  在  $S_j$  的右面, 它必然还没有被归约。所以, 当  $uS_j$  成为句柄的时候, 其规范句型必然为  $\dots uS_j S_i \dots$ 。

## 定理5.2证明(2)

- 如果存在规范句型 $\dots S_j S_i \dots$ 且 $S_j$ 是某个句柄的最后符号：
  - 考虑语法树中，同时包含有 $S_j$ 和 $S_i$ 的最小子树。设其根结点为 $U$ 。而 $U$ 的子结点从左到右为 $u$ 。
  - 短语 $u$ 的形状必然为 $\dots VW \dots$ ，且 $V \Rightarrow^* \dots S_j$ 而 $W \Rightarrow^* S_i$ 。这是因为如果有某个 $u$ 中的符号 $X \Rightarrow \dots S_j S_i \dots$ ，那么以 $X$ 为根的子树才是最小的包含 $S_j$ 和 $S_i$ 的子树。
  - 显然，根据定义 $S_j \geq S_i$ 。

# 优先关系的构造

- 根据优先关系的构造性的定义（定义5.1），我们立刻可以得到构造算法。
- $\equiv$ 的构造：直接对每个规则右部处理，每个右部 $X_1X_2\dots X_n$ ，都有 $X_i \equiv X_{i+1}$ 。
- $\leq$ 的构造：由定义， $S_j \leq S_i$ 可以得到
  - 存在规则 $U ::= \dots S_j V \dots$ ，也就是 $S_j \equiv V$ 。
  - $V \text{ HEAD } S_i$ 。
  - HEAD关系可以通过检查每个规则得到。
  - 由此可以得到 $\leq$ 就是 $(\equiv)(\text{HEAD})$ 。因此 $\leq$ 可以通过计算关系 $\equiv$ 和HEAD的传递闭包。

# 优先关系的构造（续）

- $\geq$ 关系的构造：由定义， $S_j \geq S_i$ 表示：
  - 存在规则  $U ::= \dots VW \dots$                        $V \equiv W$
  - $V \underline{TAIL}_+ S_j$                        $S_j (\underline{TAIL}_+)^T V$
  - $W \underline{HEAD}^* S_i$
  - 将上面三个规则综合起来，可以得到  $\geq$ 关系就是

$$(\underline{TAIL}_+)^T \equiv \underline{HEAD}^*$$

- 从上面的算法可以看到：  $\equiv$ ,  $\underline{TAIL}$ ,  $\underline{HEAD}$ 可以直接检查每个规则得到。因此，我们只需要有对关系的联接，闭包运算就可以得到上面的三个优先关系。



# 关系的bool矩阵表示和运算

- 关系可以使用bool矩阵来表示。对某个关系 $R$ ,  $S_j R S_i$ 在矩阵中用  $B_{ij} = \text{TRUE}$ 来表示。
- 对于两个关系 $R_1$ ,  $R_2$ 对应的矩阵 $B_1$ ,  $B_2$ ,  $B_1$ 和 $B_2$ 的乘积矩阵对应的关系就是 $R_1$ 和 $R_2$ 联接运算结果。
- 关系闭包:  $S_j R^+ S_i$ 当且仅当 $S_j R S_{j1}, S_{j1} R S_{j2}, \dots, S_{jn} R S_i$ 。其中可以要求对于任何 $k$ 和 $l$ ,  $S_{jk} \neq S_{jl}$

# 关系闭包和Warshall算法

- Warshall算法是利用矩阵计算关系传递闭包的方法。计算B的传递闭包的算法伪代码如下：

$A = B;$

for ( $i = 1; i \leq n; i++$ )

for ( $j=1; j \leq n; j++$ )

{

if ( $A[j,i]==1$ )

for( $k=1; k \leq n; k++$ )

$A[j,k] = A[j,k]+A[i,k]$

}

对于外层循环，当  $i=K$  的循环结束的时候，满足：如果  $S_i$  和  $S_j$  满足  $S_i R S_{i1}, S_{i1} R S_{i2}, \dots S_{in} R S_j$ ，并且  $i_m < K$ ，那么现在

$A[i,j] = 1;$

# 使用矩阵计算优先关系 $\leq$

- 步骤1: 构造优先关系矩阵 $B \equiv$
- 步骤2: 构造关系HEAD的矩阵 $B_{\text{HEAD}}$
- 步骤3: 使用Warshall算法计算 $B_{\text{HEAD}}^+$
- 步骤4: 计算  $B \equiv B_{\text{HEAD}}^+$

# 使用矩阵计算优先关系 $\geq$

- 步骤1: 计算TAIL的布尔矩阵 $B_{TAIL}$
- 步骤2: 计算 $B_{TAIL}^+$ , 并将该矩阵转置得到 $(B_{TAIL}^+)^T$ 。
- 步骤3: 构造优先关系矩阵 $B \equiv$
- 步骤4: 构造HEAD的矩阵 $B_{HEAD}$ 。
- 步骤5: 计算 $B_{HEAD}^+$ 的矩阵, 并由此得到 $I+HEAD^+$ 的矩阵 $B_{I+HEAD^+}$ 。
- 步骤6: 计算 $(B_{TAIL}^+)^T B \equiv B_{I+HEAD^+}$ 。
- 需要在得到的矩阵中, 把非终结符号对应列中的1去掉。(因为 $\geq$ 的右边时终结符号)

# 计算优先关系的例子P136

- 文法：  $S ::= Wa$        $W ::= Wb$        $W ::= a$
- 将文法中的符号按照S, W, a, b排列。

$$B_{\text{HEAD}} = \begin{bmatrix} 0100 \\ 0110 \\ 0000 \\ 0000 \end{bmatrix}$$

$$B_{\text{HEAD}}^+ = \begin{bmatrix} 0110 \\ 0110 \\ 0000 \\ 0000 \end{bmatrix}$$

$$B_{\text{TAIL}} = \begin{bmatrix} 0010 \\ 0011 \\ 0000 \\ 0000 \end{bmatrix}$$

$$(B_{\text{TAIL}}^+)^T = \begin{bmatrix} 0100 \\ 0110 \\ 0000 \\ 0000 \end{bmatrix}$$

# 计算优先关系的例子(续)

$$B \equiv \begin{array}{|c|} \hline 0000 \\ \hline 0011 \\ \hline 0000 \\ \hline 0000 \\ \hline \end{array}$$

$$B_{I+HEAD}^{+=} \begin{array}{|c|} \hline 1110 \\ \hline 0110 \\ \hline 0010 \\ \hline 0001 \\ \hline \end{array}$$

$$B_{\leq} B \equiv B_{HEAD}^{+=} \begin{array}{|c|} \hline 0000 \\ \hline 0000 \\ \hline 0000 \\ \hline 0000 \\ \hline \end{array}$$

$$B_{\geq} (B_{TAIL}^{+})^T B \equiv B_{I+HEAD}^{+=} \begin{array}{|c|} \hline 0000 \\ \hline 0000 \\ \hline 0011 \\ \hline 0011 \\ \hline \end{array}$$

# 优先关系的冲突

- 当优先矩阵中出现值不唯一的元素时，文法不适合使用优先识别技术来识别句型。

# 简单优先文法

- 定义5.2 如果某个文法满足：
  - 字汇表中的任意两个符号之间至多有一种优先关系成立。
  - 任何两个重写规则的右部不相同。

那么称这个文法为简单优先文法。

- 第一点保证可以识别出句柄。
- 第二点保证可以确定归约到哪个非终结符号。



## 定理5.4

- 一个简单优先文法是无二义性的，且其任何一个句型 $S_m \dots S_n$ 的唯一句柄是满足条件：

$$S_{j-1} \leq S_j \equiv S_{j+1} \equiv S_{j+2} \equiv \dots \dots \equiv S_{i-1} \equiv S_i \geq S_{i+1}$$

的最左子符号串 $S_j S_{j+1} S_{j+2} \dots \dots S_{i-1} S_i$ 。

# 定理5.4的证明

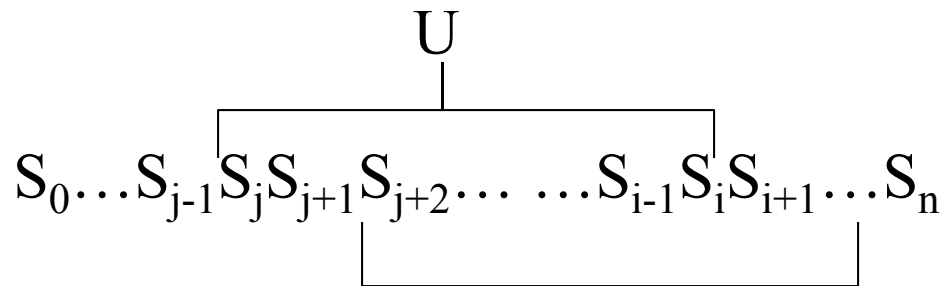
- 首先用反证法证明任何规范句型的句柄是唯一的。
  - 句型必然有句柄，且这个句柄必然满足
$$S_{j-1} \leq S_j \equiv S_{j+1} \equiv S_{j+2} \equiv \dots \dots \equiv S_{i-1} \equiv S_i \geq S_{i+1} \quad (\text{句柄1})$$
  - 如果还有另外一个语法树，那么它对应的归约（称为归约过程2）必然不是把上面的句柄作为一个整体归约的。
  - 在归约过程2中，当首次有句柄1(包括 $S_{j-1}$ 和 $S_{i+1}$ )中间的某个符号 $S_t$ 作为句柄（句柄2）的一部分被归约的时候，我们可以考虑以下的情况：（下一页）

# 定理5.4的证明(续)

- 如果 $t=j-1$ ，那么由句柄1， $S_{j-1} \leq S_j$ ；由句柄2， $S_{j-1} \equiv S_j$ 或 $S_{j-1} \geq S_j$ ；矛盾！
- 如果 $t=i+1$ ，由句柄1， $S_i \geq S_{i+1}$ ；由句柄2， $S_i \leq S_{i+1}$ 或者 $S_i \equiv S_{i+1}$ 。
- 如果 $i \leq t \leq j$ ；那么
  - $S_j$ 在句柄中：
  - $S_i$ 在句柄中：
  - $S_j$ 和 $S_i$ 都不在句柄中：

# 定理5.4的证明(续)

- 简单优先文法的无二义性是显而易见的：
  - 每个句型只有一个句柄。
  - 句柄只能归约到确定的非终结符号。
- 该证明的实质就是：句柄1和句柄2相互重叠，重叠的边缘必然有优先关系的多重定义。



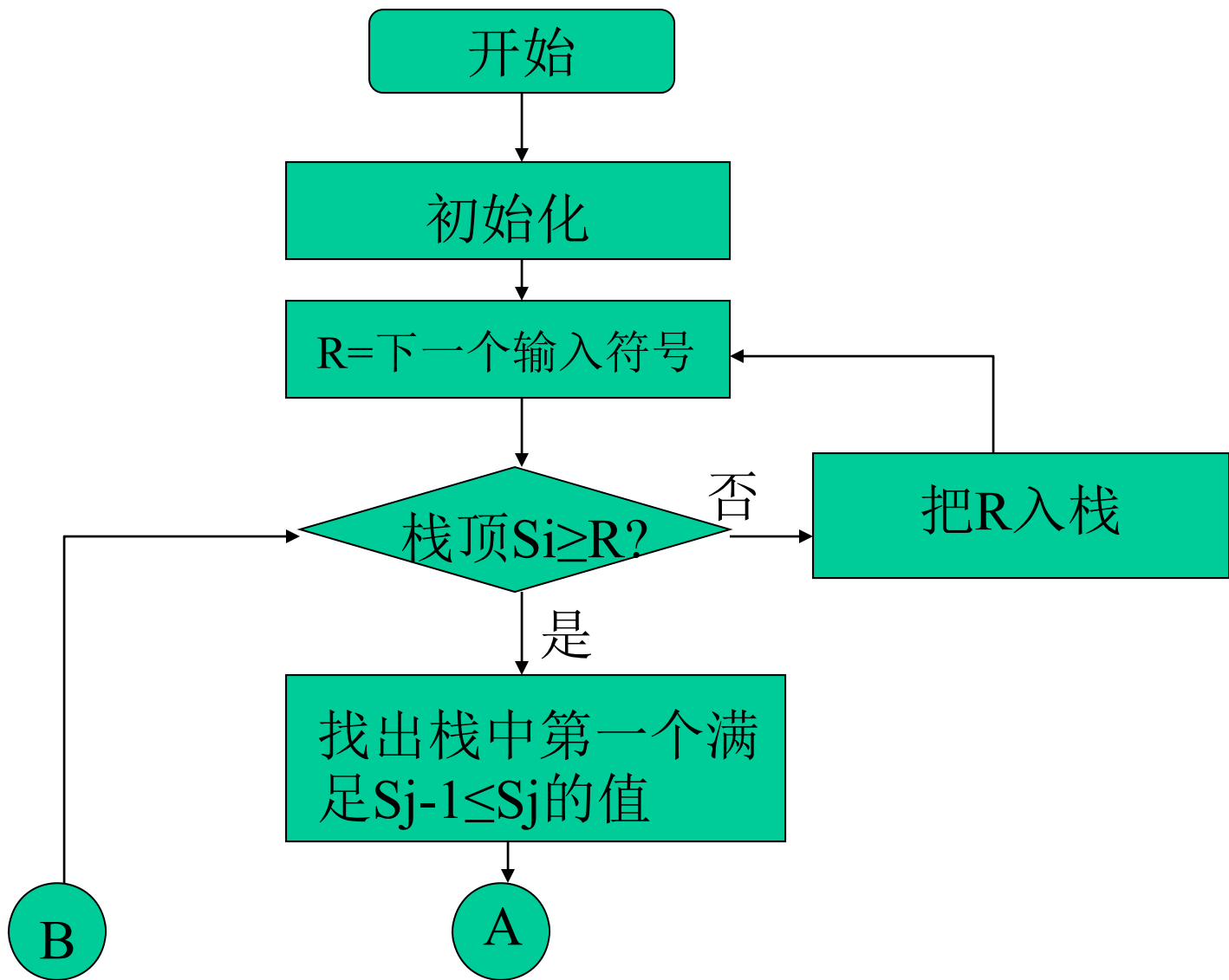
# 应用优先技术的困难与克服

- 简单优先技术只适应于简单优先文法。实际上，一般的程序设计语言的文法都不是简单优先文法。比如四则运算表达式的文法。
- 可能的解决办法是：分离法（成层法）

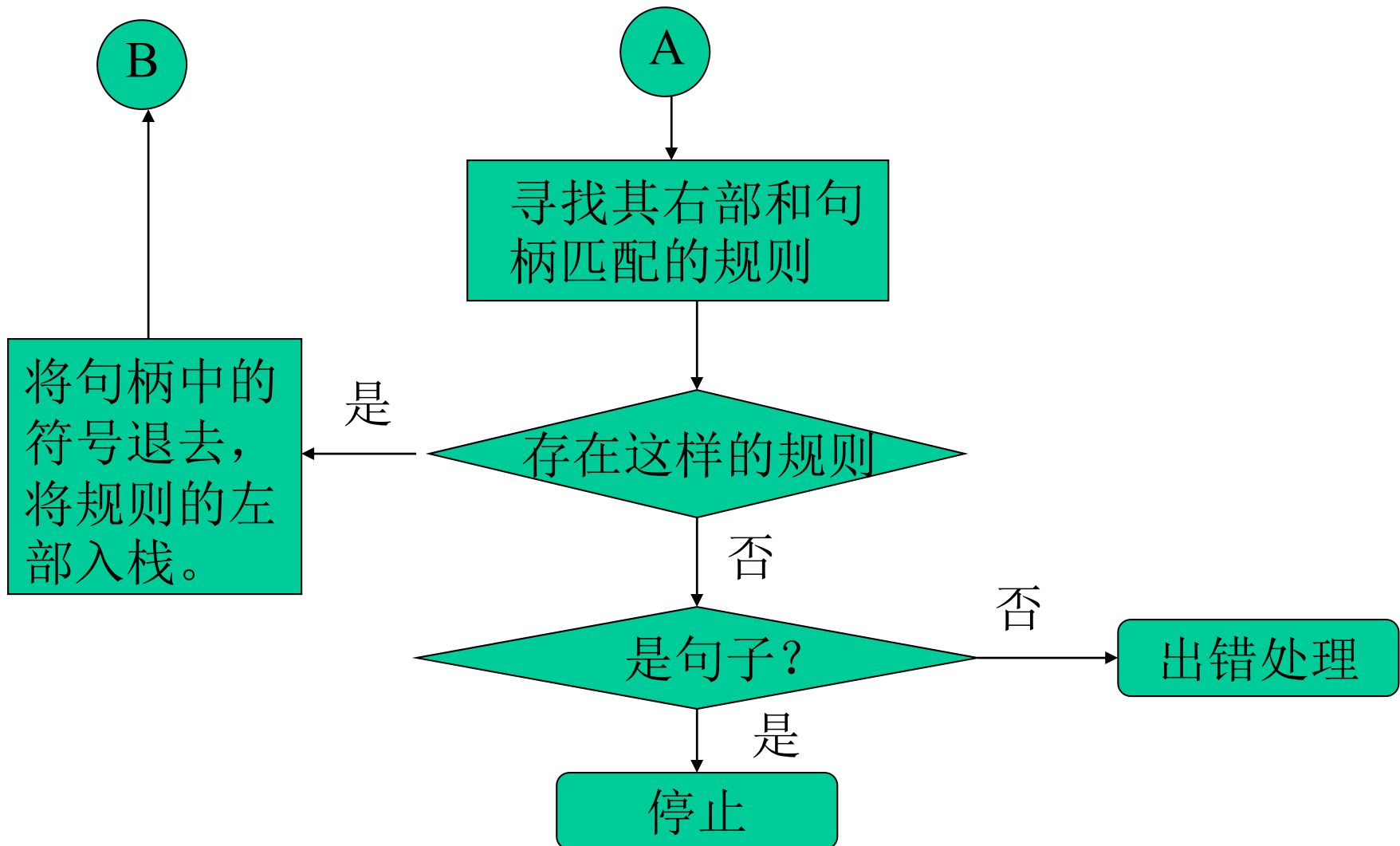
# 简单优先分析技术的实现

- 识别过程:从左到右地扫描输入符号。已经扫描或归约得到的符号被存放在一个栈中。
- 每次扫描的时候, 将当前符号 $a$ 和栈顶符号 $S$ 相比较。如果 $S \geq a$ 表示已经碰到了一个句柄的尾。然后在栈里面向前(下)找, 直到找到句柄的头。此时找到右部为该句柄的规则进行归约。

# 简单优先分析技术流程图



# 简单优先分析技术流程图(续)





# 例子

步骤	栈	关系	Next	余下部分	动作
0	#	$\leq$	b	(aa)b#	移入
1	#b	$\leq$	(	aa)b#	移入
2	#b(	$\leq$	a	a)b#	移入
3	#b(a	$\geq$	a	)b#	归约
4	#b(M	$\equiv$	a	)b#	移入
5	#b(Ma	$\equiv$	)	b#	移入
6	#b(Ma)	$\geq$	b	#	归约
7	#b(L	$\geq$	b	#	归约
8	#bM	$\equiv$	b	#	移入
9	#bMb	$\geq$	#	#	归约
10	#Z		#	#	接受

# 优先函数

- 对于一个实际的程序设计语言的文法，优先矩阵将会非常大。对于有 $n$ 个字汇的文法，矩阵的大小是 $n \times n$
- 解决的方法是引入优先函数，将矩阵线性化。这里介绍的方法称为双线性优先函数法。

## 定义5.3 优先函数

- 对于某个优先矩阵 $M$ ，如果存在两个函数 $f$ 和 $g$ ，它满足下列条件：
  - 如果 $S_j \equiv S_i$ ，那么 $f(S_j) = g(S_i)$ ；
  - 如果 $S_j \leq S_i$ ，那么 $f(S_j) < g(S_i)$ ；
  - 如果 $S_j \geq S_i$ ，那么 $f(S_j) > g(S_i)$ ；
- 那么 $f$ 和 $g$ 称为 $M$ 的线性优先函数。
- 注意：
  - 不是所有的矩阵都有优先函数的。
  - 两个没有关系的符号之间的优先函数值也有大小。

# 优先函数的例子

	Z	M	L	a	b	(	)
f	1	7	8	9	4	2	8
g	1	4	2	7	7	5	9

	Z	M	L	a	b	(	)
Z							
M				=	=		
L				>	>		
a				>	>		=
b		=		<		<	
(		<	=	<		<	
)				>	>		

# 优先函数的例子

- 下面的矩阵没有优先函数

$\equiv$	$\equiv$
$\equiv$	$\geq$

# 优先函数的构造（逐次加一法）

- 步骤1：对所有符号 $S \in V$ ，让 $f(S)=g(S)=1$
- 步骤2：对于 $S_j \leq S_i$ ，如果 $f(S_j) \geq g(S_i)$ ，让 $g(S_j)=f(S_i)+1$ ；
- 步骤3：对于 $S_j \geq S_i$ ，如果 $f(S_j) \leq g(S_i)$ ，让 $f(S_j)=g(S_i)+1$ ；
- 步骤4：对于 $S_j \equiv S_i$ ，如果 $f(S_j) \equiv g(S_i)$ ，让 $f(S_j)=g(S_i)=\max(f(S_j), g(S_i))$ ；
- 步骤5：重复步骤2-4，直到过程收敛（所有的值都不再增长）。如果有某个值大于 $2^n$ ，那么不存在优先函数。

# 逐次加一法例子

	Z	M	L	a	b	(	)
f	1	1	1	1	1	1	1
g	1	1	1	1	1	1	1

考虑关系 $\leq$

	Z	M	L	a	b	(	)
f	1	1	1	1	1	1	1
g	1	2	1	2	1	2	1

# 逐次加一法例子（例子）

考虑关系 $\geq$

	Z	M	L	a	b	(	)
f	1	1	3	3	1	1	3
g	1	2	1	2	1	2	1

.....

最后结果

	Z	M	L	a	b	(	)
f	1	3	4	4	2	1	4
g	1	2	1	3	3	3	4



# Bell有向图法

- 利用有向图构造优先函数。
- 基本思想：
  - 用 $2n$ 个点来表示 $f_i$ 和 $g_i$ 总共 $2n$ 个值。
  - 利用有向图的弧表示各个值之间应该具有的大小关系：从点 $a$ 到点 $b$ 有一个弧表示 $a$ 代表的值不小于 $b$ 。
  - 优先函数通过计算 $f_i$ 和 $g_i$ 所能够到达的结点确定（如果 $f_i$ 能够到达结点 $g_j$ ，表示 $f_i$ 的值必须不小于 $g_j$ 的值）。

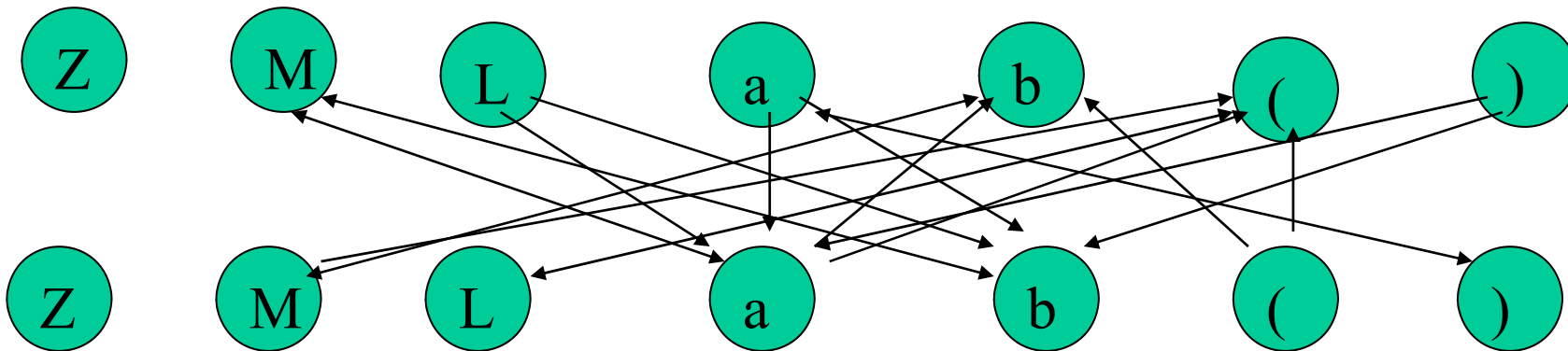
# Bell有向图算法

- 步骤1：作具有 $2n$ 个结点的有向图。标记分别为 $f_1, f_2, \dots, f_n$ 和 $g_1, g_2, \dots, g_n$ 。如果 $S_j > S_i$ 或者 $S_j = S_i$ ，那么从 $f_j$ 到 $g_i$ 画一条弧；如果 $S_j < S_i$ 或者 $S_j = S_i$ ，那么从 $g_i$ 到 $f_j$ 画一条弧。
- 步骤2：给每个结点一个数值：等于从该点出发沿弧所能到达的结点的个数。
- 步骤3：检验这样的函数是否优先函数。若否，则该优先关系不存在优先函数。

# Bell有向图的例子

f:

g:



$L \geq a, L \geq b, a \geq a, a \geq b, () \geq a, () \geq b$

$b \leq a, b \leq (, (\leq M, (\leq a, (\leq ($

$M \equiv a, M \equiv b, a \equiv (), b \equiv M, () \equiv L$

# 定理5.5

- 如果优先矩阵存在优先函数,那么使用Bell有向图方法构造得到的函数就是优先函数。
- 首先证明：如果 $S_j \equiv S_i$ ，那么 $f_j = g_i$ ；如果 $S_j \geq S_i$ ，那么 $f_j \geq g_i$ ；如果 $S_j \leq S_i$ ，那么 $f_j \leq g_i$ 。
  - 该结论显然成立。这是因为如果 $S_j \equiv S_i$ ，那么有从 $f_j$ 到 $g_i$ 和从 $g_i$ 到 $f_j$ 的弧。因此， $f_j$ 可以到达的结点， $g_i$ 也可以到达，并且 $g_i$ 的可以到达的结点， $f_j$ 也可以到达。如果 $S_j \leq S_i$ ，那么有 $g_i$ 到 $f_j$ 的弧，所以 $f_j$ 可以到达的结点， $g_i$ 也可以到达。因此 $g_i$ 的值至少不比 $f_j$ 小。同样可以证明如果 $S_j \geq S_i$ ，那么 $f_j \geq g_i$ 。

# 定理5.5的证明

- 然后证明：如果存在符号  $S_j \geq S_i$  且  $f_j = g_i$ ，那么不存在优先函数。
  - 由有向图的构造可以知道：  $g_i$  可以达到的结点，  $f_j$  必然也可以达到。又由于  $f_j = g_i$ ，所以，由  $g_i$  必然也可以到达  $f_j$ （否则  $f_j$  要大于  $g_i$ ）。因此，在  $f_j$  到  $g_i$ ，再到  $f_j$  的有一个圈。假设这个圈的结点为  $n_1 = f_j, n_2 = g_i, n_3, \dots, n_k = f_j$ 。考虑给  $n_i$  任意赋值：由弧的定义  $n_i$  的值必然不小于  $n_{i+1}$  的值，而且至少  $n_1$  的值大于  $n_2$  的值。因此，不可能有优先函数。
- 同样可以证明：如果  $S_j \leq S_i$  且  $f_j = g_i$ ，也不存在优先函数。

# Bell有向图的优点

- 非迭代过程;
- 在确定多步内可以完成;
- 改变对文法符号的排序时, 优先函数不变。

# Bell有向图法的矩阵方法

- 关系R: 如果 $f_j$ 到 $g_i$  ( $g_i$ 到 $f_j$ ) 或有一条弧, 那么 $f_j R g_i$ (或 $g_i R f_j$ )。
- 显然该关系R的矩阵如下:

	f	g
f	0	$B_{\geq \equiv}$
g	$(B_{\leq \equiv})^T$	0

# Bell有向图法的矩阵方法(续)

- 关系R传递闭包就表示了一个结点可以到达另外一个结点的关系。
- 使用Warshall算法求解上述矩阵的传递闭包 $B^+$ ，计算 $B^*=I+B^+$ 。
- 令每个结点对应的值为：该结点对应的行上的1的个数。
- 检查该赋值是否是优先函数。若是，则得到函数，否则，该优先关系不存在优先函数。



# 结点排序函数

- 后继结点：如果 $x, y \in X$ ，且存在一个弧的序列从 $x$ 到 $y$ ，那么 $y$ 是 $x$ 后继结点， $x$ 是 $y$ 的后继结点。如果弧序列长度为1，那么 $y$ 是 $x$ 的直接后继。
- 用 $\sigma(x)$ 表示 $x$ 的后继结点的集合，用 $\sigma_1(x)$ 表示其直接后继结点的集合。
- $N_D: X \rightarrow \{0, 1, \dots, |X|\}$  定义为  $N_D(x) = |\sigma(x)|$

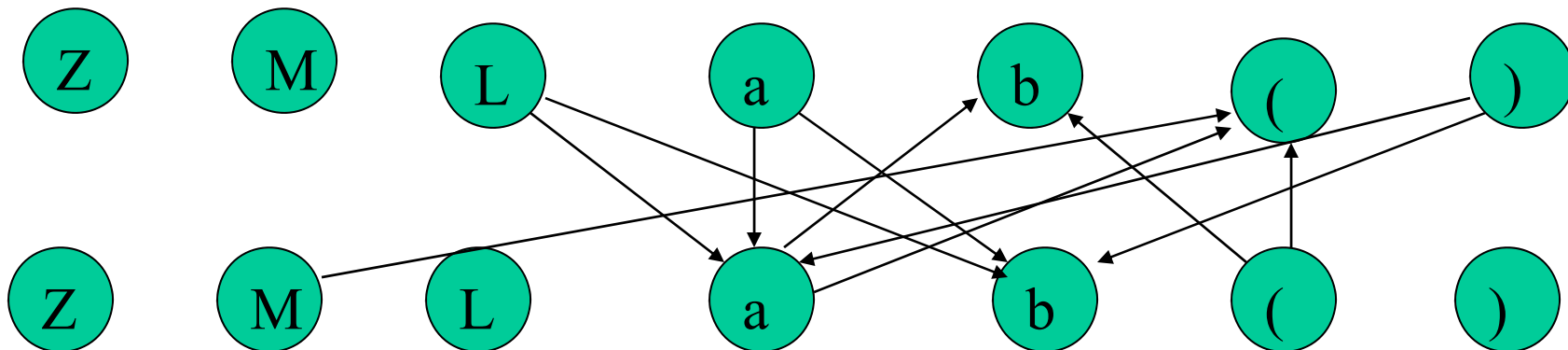
# 结点排序函数

- 任意一个函数  $N: X \rightarrow \{0, 1, \dots, |X|\}$ , 如果满足对于所有的  $x, y$ ,  $y \in \sigma(x)$  都有  $N(x) > N(y)$ , 那么称  $N$  为结点排序函数。
- 定理5.7  $N_D$  是结点排序函数, 当且仅当  $D$  中没有回路。

# Martin算法

- 步骤1：作有向图，结点为 $f_1, f_2, \dots, f_n$ ,  $g_1, g_2, \dots, g_n$ 。并且，如果 $S_j > S_i$ ，那么从 $f_j$ 到 $g_i$ 画一条弧；如果 $S_j < S_i$ ，那么从 $g_i$ 到 $f_j$ 画一条弧。
- 步骤2：如果 $S_j \equiv S_i$ ，那么从 $f_j$ 向 $g_i$ 的后继作弧，也从 $g_i$ 向 $f_j$ 的后继作弧。**重复这一个过程**，直到这个过程没有新弧加入。
- 对于最后的有向图，令 $f(S_i) = N_D(f_i)$ ，令 $g(S_i) = N_D(g_i)$ 。如果最后的有向图没有回路，那么所得到的函数就是优先函数，否则就没有优先函数。

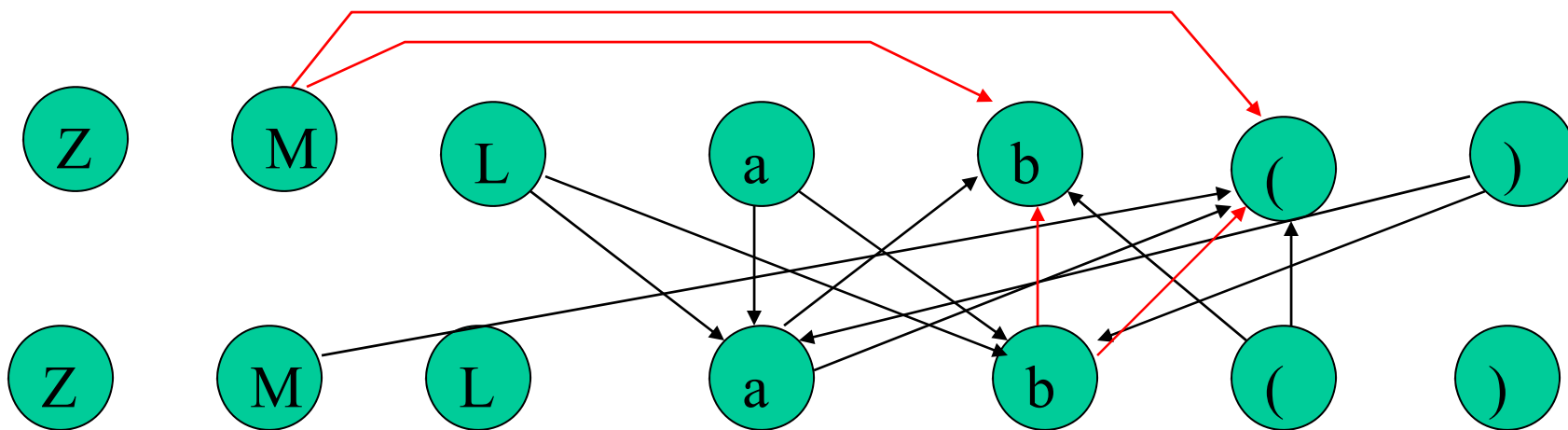
# Martin算法的例子



$L \geq a, L \geq b, a \geq a, a \geq b, ) \geq a, ) \geq b$

$b \leq a, b \leq (, ( \leq M, ( \leq a, ( \leq ($

# Martin算法例子



$M \equiv a$ ,  $M \equiv b$ ,  $a \equiv$ ,  $b \equiv M$ ,  $( \equiv L$

- 上面的图只给出了部分弧。
- 注意：对于 $\equiv$ ，必须不断重复知道没有新的弧加入为止。

# Martin算法的矩阵方法

- 步骤1: 构造布尔矩阵 $B_{\trianglelefteq}$ 。此时的矩阵相当于画图时的第一步得到的图。 $B_{\trianglelefteq}$ 为:

$$\begin{array}{cc} 0 & B_{\geq} \\ (B_{\leq})^T & 0 \end{array}$$

- 步骤2: 计算 $B = C^+ B_{\trianglelefteq}$ 。其中C为

$$\begin{array}{cc} I & B_{\equiv} \\ (B_{\equiv})^T & I \end{array}$$

注释: 点A到点B有一条弧 iff 有一个点C, 使得 $A \equiv C$   
且 $C \rightarrow B$

# Martin算法的矩阵方法

- 步骤3： 计算 $B^+$ 。
- 步骤4： 计算各个行中的1的个数。如果对于某个 $i$ ,  $B[i][i]=1$ 表示图存在回路, 优先函数不存在。

# Martin算法的正确性说明

- 在算法中，从节点A到节点B有弧表示优先关系要求A对应的优先函数值大于B对应的值。
- 如果 $S_i \equiv S_j$ ，那么 $f_i$ 的后继同时也是 $g_j$ 的后继， $g_j$ 的后继也是 $f_i$ 的后继。所以最后得到的 $S_i$ 和 $S_j$ 的后继个数相同。
- 如果存在回路，那么表示优先关系要求某个函数值大于自己，显然不行。



# 优先函数的缺点

- 信息的丢失，原来两个符号之间有4中关系（大于，小于，等于，无关），引入优先函数之后，只有3种关系。
- 信息的丢失错误的延迟发现。
- 见P188页例子。

# 简单优先技术的局限性

- 文法的适用范围小。
- 虽然使用成层法可以使一些文法变成简单优先文法，但是
  - 成层法的技术非常复杂。
  - 当两个关系既有 $\leq$ 又有 $\geq$ 时，成层法无能为力。
- 如果使用高阶矩阵，将使得算法的内存需求更加大。

# 算符优先分析技术

- 简单优先技术对字汇表中的所有符号之间建立优先关系。但是，有些情况下，不需要对所有两个符号之间建立优先关系。
- 算符优先分析技术只在部分符号（终结符号）之间建立优先关系。

# 算符优先分析技术基本思想

- 对于算术表达式，只需要按照操作符之间的优先关系，就可以确定运算的顺序。不需要考虑操作数就可以对表达式进行分析。
- 例如： $E+T * F$ 。只需要知道 $*$ 的优先级高于 $+$ ，就可以知道 $T * F$ 是句柄。
- 在一般的文法中，终结符号的地位相当于操作符号。

# 算符文法

- 定义：如果文法中没有形状如

$$U::= \dots VW\dots$$

的规则，该文法称为算符文法。

# 算符文法的性质

- 定理5.9 对于算符文法，不存在包含有相邻两个非终结符号的句型。
- 定理5.10 如果 $TU$ 出现在句型中，其中 $T$ 为终结符号， $U$ 为非终结符号，那么包含 $T$ 的短语也必然包含 $U$ 。
- 定理5.11 如果 $UT$ 出现在句型中，其中 $T$ 为终结符号， $U$ 为非终结符号，那么包含 $T$ 的短语也必然包含 $U$ 。

# 定理5.9的证明

- 只需要证明：如果 $x$ 不包含两个相邻的非终结符号，且 $x \Rightarrow y$ ，那么 $y$ 也不包含相邻的非终结符号。
- 假设 $x = wUv$ ，而 $y = wuv$ 。
  - 那么由于 $x$ 不包含两个相邻的非终结符号，那么 $w$ 和 $v$ 中没有不相邻的非终结符号。
  - 根据算符文法的定义， $u$ 中也不包含相邻的非终结符号。
  - 根据假设， $w$ 的结尾不是非终结符号（否则， $x$ 中包含有相邻的非终结符号）。
  - 同样， $v$ 的开始符号也不是非终结符号。
- 综上所述： $y$ 中不存在相邻的非终结符号。

## 定理5.10, 5.11的证明

- 假设 $w = xvy$ 是文法的句型，而 $v$ 是详相对于 $V$ 的短语。
- 那么 $xVy$ 也是句型。
- 如果 $w$ 中有两个相邻的符号 $TU$ ，且 $T$ 在 $v$ 中，而 $U$ 不在 $v$ 中。显然 $U$ 是 $y$ 的头符号。
- 因此 $xVy$ 中存在两个相邻的非终结符号 $VU$ 。
- 和定理5.9矛盾。
- 定理5.11的证明和定理5.10类似。



# 算符优先关系

- 定义5.8 设文法 $G$ 是一个算符文法,  $T_j$ 和 $T_i$ 是两个任意的终结符号, 而 $U, V, W \in V_T$ , 定义算符优先关系如下:
  - $\approx$ : 当且仅当文法 $G$ 中存在形如:  $U ::= \dots T_j T_i \dots$  或  $U ::= \dots T_j V T_i \dots$  的规则。
  - $\prec$ : 当且仅当文法 $G$ 中存在规则:  $U ::= \dots T_j V \dots$  的规则, 且  $V \Rightarrow^+ T_i \dots$  或  $V \Rightarrow^+ W T_i \dots$ 。
  - $\succ$ : 当且仅当文法 $G$ 中存在规则:  $U ::= \dots V T_i \dots$  的规则, 且  $V \Rightarrow^+ \dots T_j$  或  $V \Rightarrow^+ \dots T_i W$ 。

# 算符优先关系的直观意义

- 算符优先分析技术的基本思想是通过终结符号之间的优先关系，确定句型的句柄。

- 对于句型

$$[N_1]T_1 \dots [N_{i-1}]T_{i-1}[N_i]T_i \dots [N_j]T_j[N_{j+1}]T_{j+1} \dots [N_k]T_k[N_{k+1}]$$

- 满足关系  $T_{i-1} \prec T_i \approx T_{i+1} \approx \dots \approx T_j \succ T_{j+1}$  的最左子符号串就是要被归约的短语。

# 优先关系例子

- 文法：
 
$$\begin{array}{ll} Z ::= E & E ::= T | E + T \\ T ::= F | T * F & F ::= (E) | i \end{array}$$

- 等于关系： ( $\approx$ )

- 由推导

- $Z \rightarrow E \rightarrow E + T \rightarrow E + F \rightarrow E + i$
- $Z \rightarrow E \rightarrow E + T \rightarrow E + T * F \rightarrow E + T * (E) \rightarrow E + T * (E + T)$
- $Z \rightarrow E \rightarrow E + T \rightarrow E + T + T \rightarrow E + T + F \rightarrow E + T + (E)$

得到以下关系：

- $+ \prec i, + \prec *, * \prec (, (\prec +, + \succ ), + \succ +, + \prec ($  (等.

# 优先关系的构造

- 优先关系 $\approx$ 的构造，只需要按照定义，枚举各个规则的右部就可以得到。
- 对于关系 $\prec$ 和 $\succ$ 的构造，我们需要引入两个辅助的关系：FIRSTTERM和LASTTERM。
  - $U \text{ FIRSTTERM } T$ 当且仅当存在规则 $U::=T\dots$ 或者 $U::=VT\dots$ 。
  - $U \text{ LASTTERM } T$ 当且仅当存在规则 $U::=\dots T$ 或者 $U::=\dots TV$ 。

# FIRSTTERM<sup>+</sup>关系的构造

- FIRSTTERM<sup>+</sup>并不是FIRSTTERM的传递闭包。  
U FIRSTTERM<sup>+</sup> T表示T是U经过若干步推导得到的字的首终结符号。构造算法如下：
  - 步骤1：如果U FIRSTTERM T，那么U FIRSTTERM<sup>+</sup> T.
  - 步骤2：如果U::=V...，且V FIRSTTERM<sup>+</sup> T，那么U FIRSTTERM<sup>+</sup> T。
  - 步骤3：重复步骤2，知道过程收敛。

# LASTTERM<sup>+</sup>的构造算法

- LASTTERM<sup>+</sup>不是LASTTERM的传递闭包。U LASTTERM<sup>+</sup> S表示U经过若干步推导得到的字的尾终结符号为S。构造算法为：
  - 步骤1：如果U LASTTERM T，那么U LASTTERM<sup>+</sup> T。
  - 步骤2：如果U ::= ...V，且V LASTTERM<sup>+</sup> T，那么U LASTTERM<sup>+</sup> T。
  - 步骤3：重复步骤2直到收敛。

# 关系 $\leftarrow$ 和 $\rightarrow$ 的构造

- 关系 $\leftarrow$ 的构造:
- $\leftarrow = (\equiv)(\text{HEAD}^*)(\text{FIRSTTERM})$   
 $= (\equiv)(\text{FIRSTTERM}+)$
- $\rightarrow = ((\text{TAIL}^*)(\text{LASTTERM}))^T(\equiv)$   
 $= (\text{LASTTERM}+)^T(\equiv)$

# 算符优先文法

- 定义5.11：设有算符文法 $G$ ，如果其任意两个终结符号之间，算符优先关系最多只有一种关系成立，那么该文法 $G$ 称为算符优先文法。

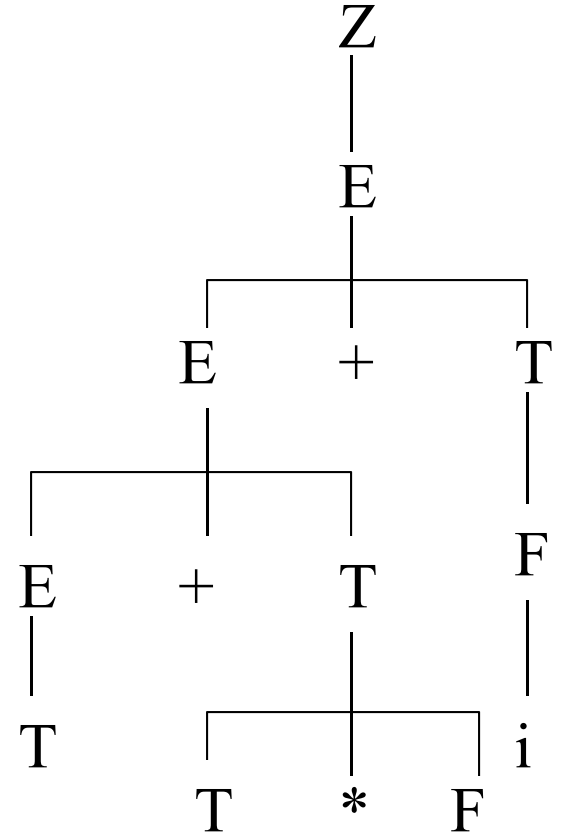


# 算符优先文法句型的识别

- 由于算符优先分析技术在分析的过程中，非终结符号是‘不可见’的。因此，对于单规则，算符优先技术无法处理。
- 定义5.12 质短语是满足下面条件的短语：
  - 至少包含一个终结符号。
  - 该短语不再包含满足第一个条件的更小的短语。

# 质短语的例子

- 短语有 $T+T^*F+i$ ,  $T+T^*F$ ,  $T^*F$ , 最左边的 $T$ ,  $i$ 。
- 其中, 质短语为 $T^*F$ ,  $i$ 。
- $T+T^*F+i$ ,  $T+T^*F$ 不是质短语, 因为它们包含了 $T^*F$ 。
- $T$ 不是质短语, 因为其中没有终结符号。



# 定理5.14

- 定理5.14: 句型

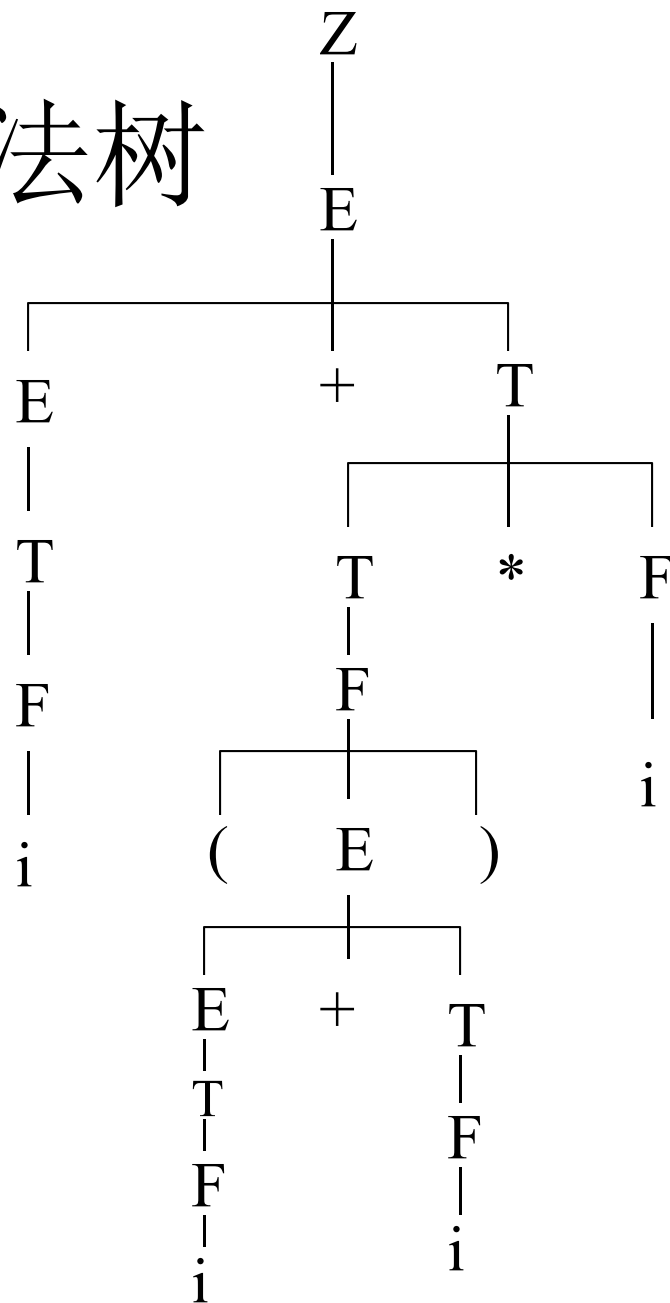
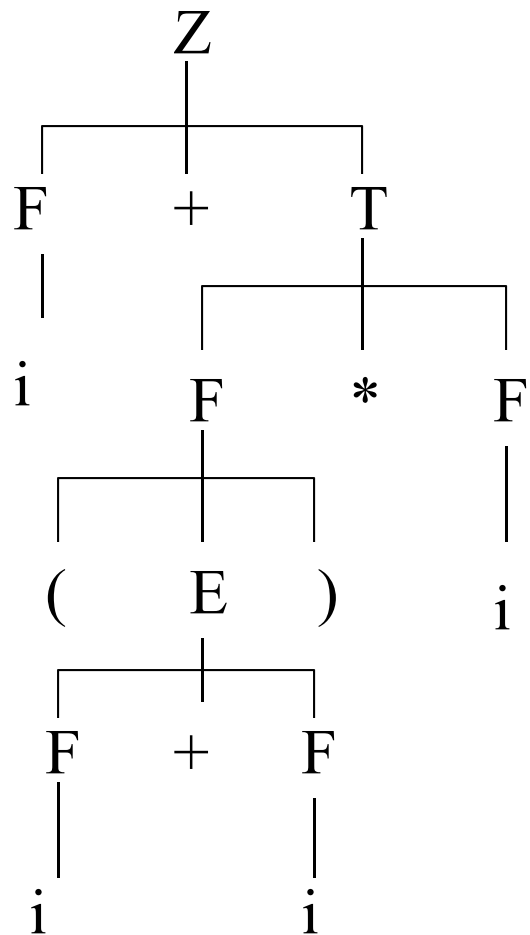
$[N_1]T_1 \dots [N_{i-1}]T_{i-1}[N_i]T_i \dots [N_j]T_j[N_{j+1}]T_{j+1} \dots [N_k]T_k[N_{k+1}]$

中满足关系  $T_{i-1} \lhd T_i \approx T_{i+1} \approx \dots \approx T_j \rhd T_{j+1}$  的最左子符号串  $[N_i]T_i \dots [N_j]T_j[N_{j+1}]$  就是句型的质短语。

# 句型识别过程

	句型	关系	质短语	符号
1	$i+(i+i)*i$	$\# \leftarrow i \rightarrow +$	$i$	F
2	$F+(i+i)*i$	$\# \leftarrow + \leftarrow (\leftarrow i \rightarrow +$	$i$	F
3	$F+(F+i)*i$	$\# \leftarrow + \leftarrow (\leftarrow + \leftarrow i \rightarrow)$	$i$	F
4	$F+(F+F)*i$	$\# \leftarrow + \leftarrow (\leftarrow + \rightarrow)$	$F+F$	E
5	$F+(E)*i$	$\# \leftarrow + \leftarrow (\approx) \rightarrow *$	$(E)$	F
6	$F+F*i$	$\# \leftarrow + \leftarrow * \leftarrow i \rightarrow \#$	$i$	F
7	$F+F*F$	$\# \leftarrow + \leftarrow * \rightarrow \#$	$F*F$	T
8	$F+T$	$\# \leftarrow + \rightarrow \#$	$F+T$	Z

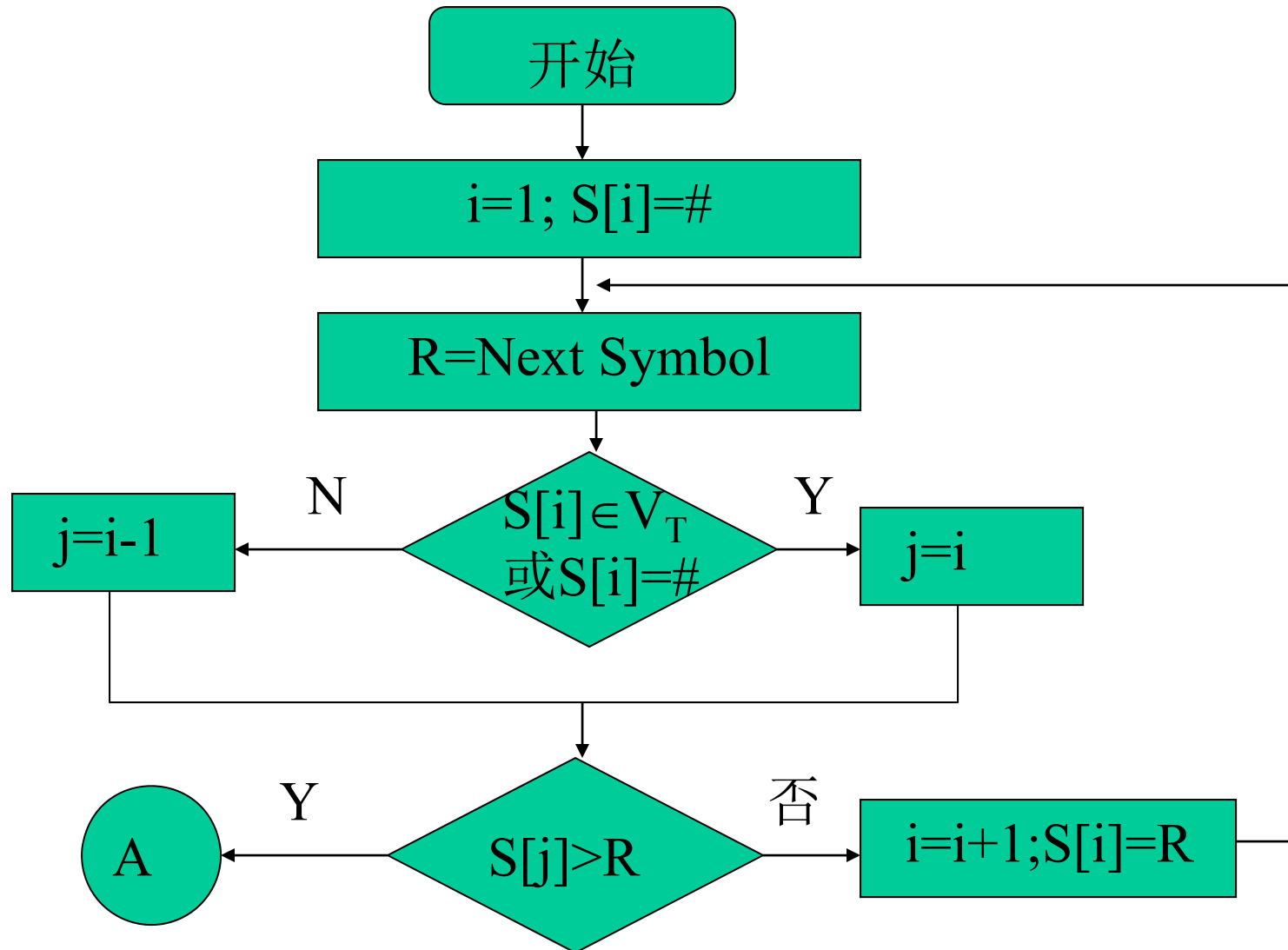
# 识别得到的语法树



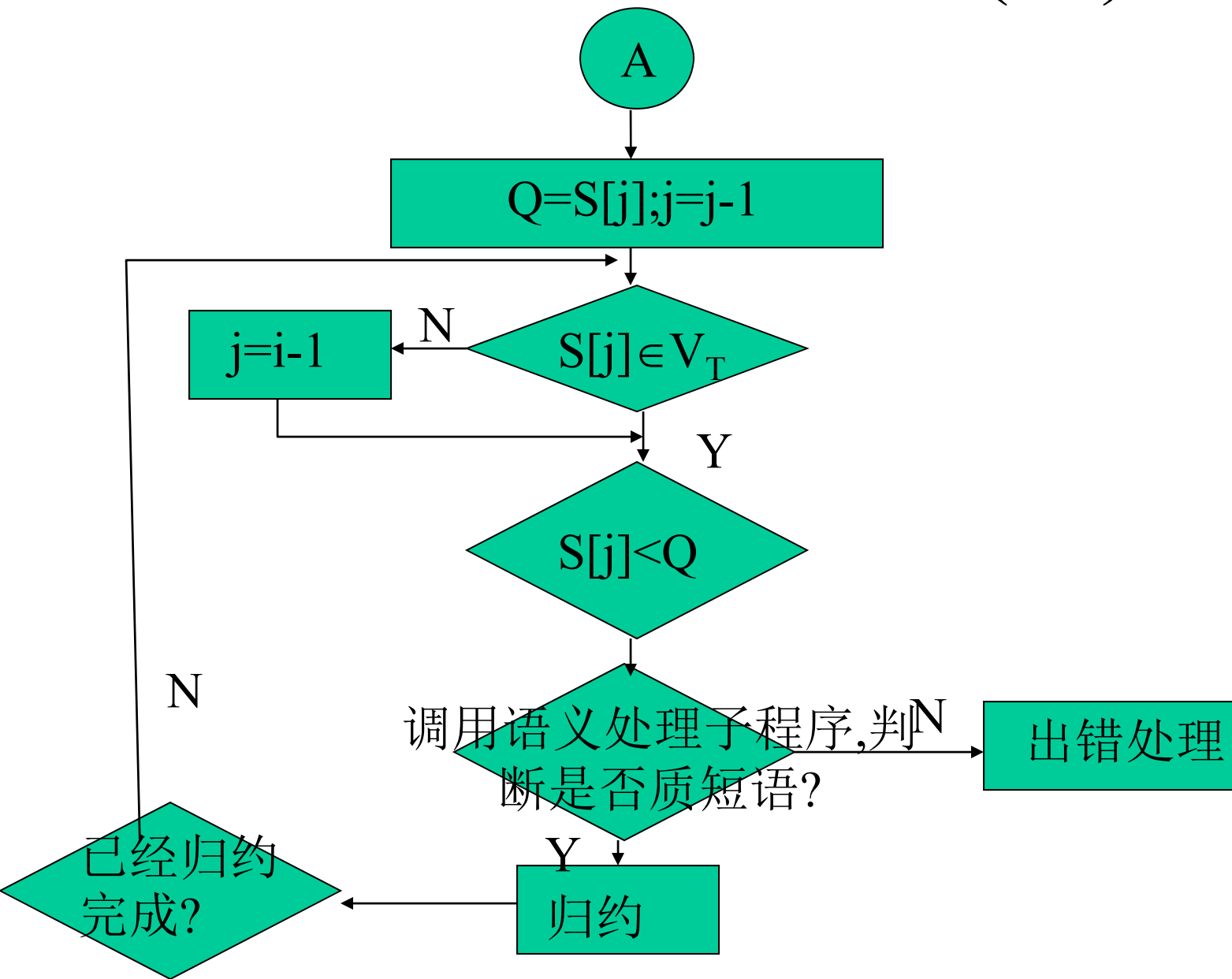
# 算符优先技术的说明

- 在算符优先技术的应用中，分析过程并不考虑非终结符号。可以认为：编译程序不考虑具体符号的名字，只考虑它的意义。
- 需要有处理质短语的语义处理子程序。
- 在使用算符优先技术的过程中，我们可以使用同一个符号N来表示归约得到的非终结符号，分析过程照样可以进行。

# 识别算法流程图



# 识别算法流程图(续)





# 语义处理子程序

- 语义处理子程序需要根据栈里面的符号（和其它信息）分析是否是一个质短语。
- 建议归约的时候，遵照以下法则：
  - 对于质短语 $w=[N_{i-1}]T_i[N_i]T_{i+1}\dots T_j[N_{j+1}]$ ，归约到如下的非终结符号 $V$ ：
  - $V \rightarrow u \rightarrow +w$ ，且在 $u$ 到 $w$ 的推导过程中，只用到了如 $U::=W$ 的单规则。 $U, W$ 为非终结符号。

# 实际应用的算符优先分析技术

- 可以使用优先函数来替代优先矩阵。优先函数的求解方法等同于简单优先矩阵的算法。（前面的算法不考虑优先关系的种类）
- 可以使用两个栈：运算符栈，运算分量栈。运算分量（非终结符号）和运算符（终结符号）将分别存放在两个栈中。

# 算符优先文法的范围

- 可以被用来处理各种表达式。
- 如果把各个关键字看作算符，这个技术也可以被用来处理程序设计语言。
- 对于实际使用的程序设计语言，只需要对文法稍微修改就可以应用算符优先分析技术。
- 对于有些情况，比如表达式，我们可以使用单优先函数来解决。（实际上即 $f(S)=g(S)$ ;

# 两种优先技术的比较

项目	简单优先技术	算符优先技术
优先关系	简单优先关系	算符优先关系
关系定义集	字汇表	终结符号集
归约方式	规范归约	‘规范’ 归约
被归约者	句柄	质短语
归约条件		
控制方式	优先矩阵或优先函数	优先矩阵或优先函数
实现工具	栈	栈
存储需求	比较大	比较小
功能	低	较高
语义子程序	要求少	要处理的多
使用范围	简单优先文法	算符优先文法

# LR(K)分析技术

- LR(K)是指:可以以k为界,从左到右地翻译。也就是说,在扫描的过程中,最多向前看K个符号,就可以确定句柄。
- LR(K)技术可以应用于几乎所有的用CFG描述的程序设计语言。并且有识别程序自动生成器。

# LR(K)文法定义

- 定义5.14 一个文法是LR(K)的，当且仅当句子的识别过程中，任何一个句柄都可以由它左边的符号和右边的K个符号确定。
- 定义5.15(句柄)设 $\alpha = X_1X_2...X_n...X_t$ 是文法的句型。假定 $X_{r+1}...X_n$ 是和第p个规则相匹配的，那么称二元组 $(n,p)$ 为 $\alpha$ 的句型。

# LR(K)文法的定义

- 定义5.16 后面跟了K个#的句型称为K句型。
- 定义5.17 (LR(K)的形式化定义) 设有文法G, 并且  $\alpha = X_1X_2...X_nX_{n+1}...X_{n+k}Y_1...Y_u$  和  $\beta = X_1X_2...X_nX_{n+1}...X_{n+k}Z_1...Z_v$  是G的两个句型,  $X_{n+1}...X_{n+k}, Y_i, Z_i$  都不是非终结符号。设 $\alpha$   $\beta$ 的句柄分别为(n,p)和(m,q)。如果G满足: 对于任何两个这样的句型,  $n=m$ 且 $p=q$ , 那么G就是LR(K)的。
- LR(K)是文法的概念。

# LR(K)文法的若干性质

- 性质1：对于任何可用一个LR(K)文法定义的上下文无关语言都能用一个确定的下推自动机以自底向上方式识别。
- 性质2：LR(K)文法无二义性。
- 性质3：当K确定的时候，一个文法是否是LR(K)文法是可判定的。
- 性质4：一个语言能够由LR(K)文法生成，当且仅当它能够由LR(1)生成。



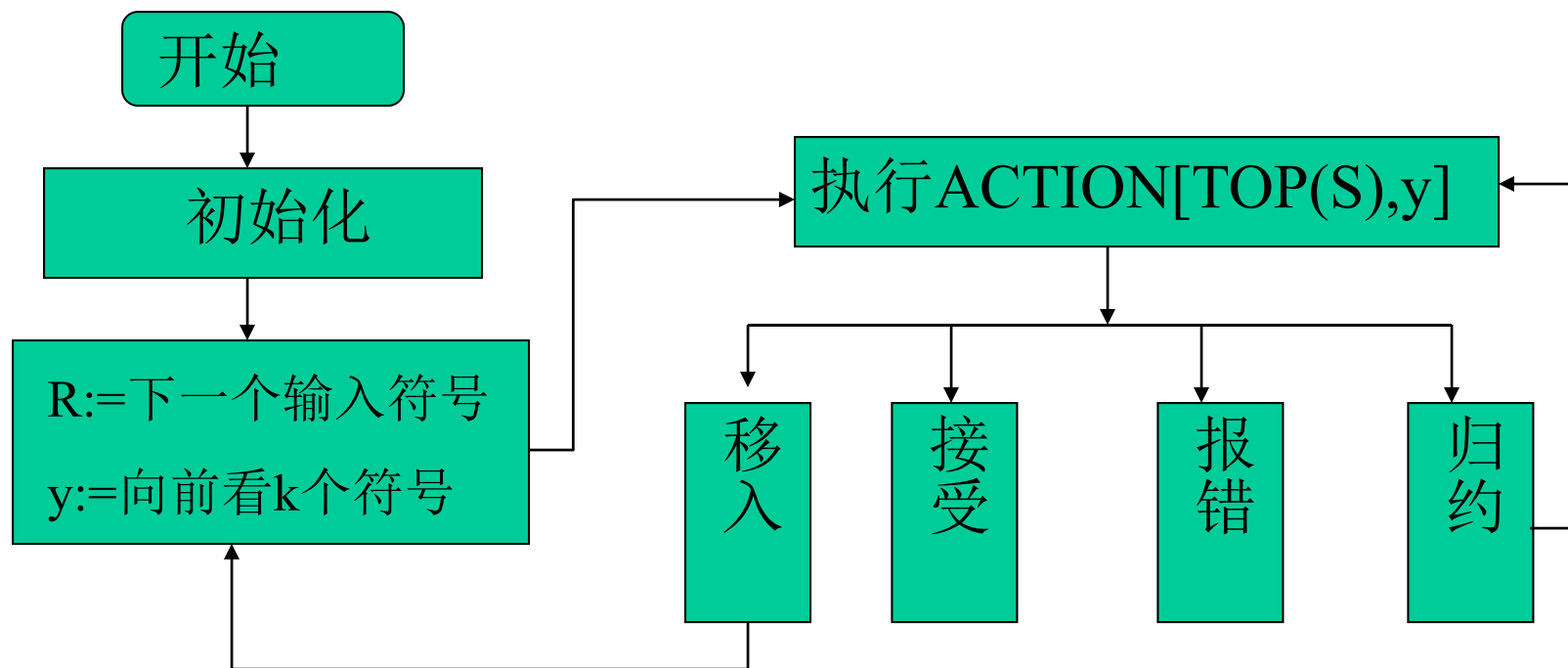
# LR(K)技术的算法框架

- 由驱动程序和分析表组成。
- 根据分析表，对栈中信息和当前输入符号做出动作决定：移入/归约/接受/报错。
- 栈中的一个元素分成两个部分：
  - 状态：综合了‘历史’和‘展望’信息。状态为如： $[Xp1...Xpj \cdot Xpj+1...Xpn; \alpha]$ 项的集合，项的意义为：试图按照规则 $p$ 归约，已经扫描和归约得到了 $Xpj$ 前的符号，希望扫描以后的符号。如果右部都扫描得到了，只有下 $K$ 个符号为 $\alpha$ 时，才按照这个规则归约。
  - 文法符号：被移入或者归约得到的符号。

# LR分析表

- LR分析表有两个部分：动作部分ACTION和状态转换GO部分。都对应于二维数组。
- ACTION[S,y]表明当状态为S，向前看符号串y时，应该采取的动作
  - 移入：将S,y的下一个状态S以及当前符号入栈。
  - 归约：对栈顶的符号串按照某个规则进行归约。
  - 接受：宣布输入符号串为一个句子。
  - 报错：宣布输入符号串不是句子。
- GO[S,U]表示当前状态S和非终结符号匹配的时候所转换到的下一个状态。

# LR驱动程序算法流程



# LR分析表例子

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			

# LR分析例子

步骤	栈	输入	动作	说明
1	0	i+i*i	S5	移入i
2	0i5	+i*i	r6	归约i
3	0F3	+i*i	r4	归约F
4	0T2	+i*i	r2	归约T
5	0E1	+i*i	S6	移入+
6	0E1+6	i*i	S5	移入i
7	0E1+6i5	*i	r6	归约i
8	0E1+6F3	*i	r4	归约F
9	0E1+6T9	*i	S7	移入*
10	0E1+6T9*7	i	S5	归约i
11	0E1+6T9*7i5	#	r6	归约i
12	0E1+6T9*7F10	#	r3	归约T*F
13	0E1+6T9	#	r1	归约E+T
14	0E1	#	acc	接受

# 例子的说明

- 分析过程的动作是由栈顶元素（状态）和当前输入符号决定的，栈中的文法符号可以忽略。原因是状态中实际包含了关于文法规则的信息。
- 只有当执行归约的时候，才需要考虑文法规则。
- 在分析开始的时候，构型为 $S_0|i+i*i\#$ 。结束的时候，构型为 $S_0ES_1|\#$
- 识别过程适用于全部的LR(1)文法。

# LR(K)分析技术的思想

- 首先猜测下一步归约要使用的规则。猜测的顺序从左边开始猜测，过程是逐步进行的。
- 逐步读入符号，根据当前符号和向前看符号串来剔除那些没有希望的猜测。同时，相应地进行归约。
- 下面，我们用文法 $E::=E+T$ ,  $E::=T$   $T::=T*F$   
 $T::=F$   $F::=(E)$   $F::=i$ 进行演示。输入为 $i+i$ 。

# LR(K)分析技术的思想(例子)

- 要归约到E（目标），必须经过规则 $E \rightarrow E+T$ ，或者 $E \rightarrow T$ 。
- 而要使用上面的两个规则，要求将输入的（某个）头首先归约到E, T（子目标）。
- 而要归约到T，必须要使用规则 $T \rightarrow F$ 或者 $T \rightarrow T * F$ 。
- 要把输入的头归约到T，首先要归约得到F。而得到F可以使用的规则为 $F \rightarrow i$ ，或者 $F \rightarrow (E)$ 。
- 现在，第一个符号为i。只有 $F \rightarrow i$ 可能归约这个符号。所以，第一次归约的规则为 $F \rightarrow i$ 。



# LR(K)分析技术的思想(例子)

- 经过归约，第一个符号为F，得到句型F+i。此时，有可能进行归约的规则为 $T \rightarrow F$ 。第一个符号为T。
- 随后，再次进行归约，第一个符号为E。
- 由于还有输入符号，我们发现，要将E(和后面的某些符号)归约，可能使用的规则为 $E \rightarrow E+T$ 。
- 我们读入+，现在要归约的符号为E+...。显然，我们要试图‘读入’一个T，然后才能按照 $E \rightarrow E+T$ 归约。这样，我们需要把余下输入的前面部分首先归约为T，然后才可以得到结果。因此，我们的下面首先尝试归约得到T。

# LR(K)分析技术的思想(例子)

- 要将余下的开始部分归约得到T，我们可能的方法是，根据规则 $T \rightarrow F$ ，或者 $T \rightarrow T * F$ 进行归约。
- 要试图按照规则 $T \rightarrow T * F$ 归约，必须首先归约得到T。最初必须归约得到F。而要得到F，我们必须按照规则 $F \rightarrow i$ 或者 $F \rightarrow (E)$ 归约。
- 下一个符号是i。所以，我们可以得到F。
- 同样，我们由F可以得到T。
- 由于，前面我们已经得到了E+，加上现在的T，我们得到了 $E \rightarrow E + T$ 。归约完成。

# 基本思想的实现

- 首先，如果我们试图按照某个规则来归约的话，我们需要记住已经归约或移入得到的规则右部的前半部分。对于第P规则 $U \rightarrow X_1X_2...X_n$ ，我们用 $U \rightarrow X_1X_2...X_j.X_{j+1}...X_n$ 表示已经读入（或归约得到）了前j个符号，现在需要试图得到后面的符号。上面的称为一个‘**项**’，可以用(P,j)表示。
- 但是，我们前面看到，在一个时刻会有多个可能猜测。所以，当前状态使用**项集**来表示。项集中的每个项都表示了一个可能的猜测。

# 基本思想的实现（续）

- 从前面的例子里面可以看到，我们尝试某个规则时，有必要先归约得到一个非终结符号。比如，我们尝试 $E \rightarrow E+T$ ，并扫描到了 $E+$ 的时候，需要首先归约得到一个 $T$ 。因此，当圆点在某个非终结符号的时候，我们需要首先尝试该非终结符号的规则。
- 定义：（项集闭包）
- $CLOSURE(S) = S \cup \{[q, 0; \beta] \mid [p, j, \alpha] \in CLOSURE(S), j < n_p, X_{p_{j+1}} = U_q, \text{ 且 } \beta \in H_k(X_{p_{j+2}} \dots X_{p_{n_p}} \alpha)\}$

# 基本思想的实现（续）

- Hk函数的说明：Hk函数的值是参数对应的字的前K个符号。
- 定义： $Hk(\alpha) = \{\beta \mid \alpha \rightarrow^* \beta, |\beta| = K, \beta \in VT^*\}$ .
- 当 $k=1$ 的时候，Hk的求法就类似于LL(1)技术中的第一个终结符号的求法。

# 项集闭包的例子

- 文法:

0 $Z ::= E\#$	1 $E ::= E+T,$	2 $E ::= T$
3 $T ::= T * F$	4 $T ::= F$	5 $F ::= (E)$
6 $F ::= i$		
- $S = \{[Z \rightarrow \cdot E\#, \#]\}$
- $CLOSURE(S)$ 还包含以下项:
- $[E \rightarrow \cdot E+T, \#], [E \rightarrow \cdot T, \#]$
- $[E \rightarrow \cdot E+T, +], [E \rightarrow \cdot T, +], [T \rightarrow \cdot T * F, \#], [T \rightarrow \cdot F, \#]$
- $[T \rightarrow \cdot T * F, +], [T \rightarrow \cdot F, +], [T \rightarrow \cdot T * F, *], [T \rightarrow \cdot F, *], [F \rightarrow \cdot i, \#], [F \rightarrow \cdot (E), \#]$
- $[F \rightarrow \cdot i, *], [T \rightarrow \cdot (E), *]$

# 向前看符号串

- 向前看符号的说明：项中的向前看符号串是用来解决冲突问题的。在前面的例子中，我们已经归约得到了第一个 $T$ ，为什么没有尝试 $T \rightarrow T * F$ 呢？原因就在于我们发现下一个符号是 $+$ 。这个 $+$ 号表明了我们可以将 $T$ 归约。
- 那么，当尝试一个规则，并移入/归约得到了它右部的所有规则之后，什么时候可以归约呢？显然，我们要考虑当初为什么要尝试这个规则。
- 如果尝试规则 $U \rightarrow u$ 的原因是：在尝试规则 $V \rightarrow \dots U r$ 时，已经扫描到了 $U$ 的前面，试图归约得到 $U$ 。所以，此时如果要进行归约，向前看符号串必须时可以由 $r$ 推导得到（或者加上 $V$ 对应的向前看符号）。

# 向前看符号串

- 再次考虑项集闭包中，对于新增加项的向前看符号串的定义：
- $CLOSURE(S) = S \cup \{[q, 0; \beta] \mid [p, j, \alpha] \in CLOSURE(S), j < n_p, X_{p_{j+1}} = U_q, \text{ 且 } \beta \in H_k(X_{p_{j+2}} \dots X_{p_{n_p}} \alpha)\}$



# 初始项集

- 我们将一个项集作为一个状态。
- 初始状态：在开始的时候，我们考虑的是最后归约到识别符号时候使用的规则。所以猜测的可能性(项集)为：CLOSURE(S)。其中S为：
- $\{[q,0;\#k] \mid \text{第}q\text{个规则为关于识别符号的规则}\}$ 。
- 显然，如果某个关于识别符号的规则的第一个符号是非终结符号的时候，我们还需要进行关于给识别符号的猜测。
- 初始项集的例子见前面

# 项集之间的转换关系(Action归约)

- 我们需要构造两个分析表：Action和Go。
- 状态对应于项集。在Action中， $A[S, \alpha]$ 表示的是栈顶状态为S的时候，向前看符号串为 $\alpha$ 的时候应该采取的动作。
- 当其中某个项的形状如： $[U \rightarrow x., \alpha]$ 的时候，表示针对该规则的扫描已经完成，并且，如果当前的符号为 $\alpha$ 的话，就可以进行归约。因此这个项主张： $A[S, \alpha]$ 的动作应该是按照这个规则进行归约。
- 如果有多个这样的规则，那么显然当栈顶符号为S而向前看符号为 $\alpha$ 的时候，我们有多多个可能的归约方式。由于我们只有一个栈，不可能同时对这个栈中的符号进行多种操作。所以这样就引起了冲突。
- 例子：当前状态（项集）为 $\{[2,1;\#], [4,0,+], [6,0,\#], \dots\}$ 而向前看符号为 $\#$ ，表示应该进行归约。

# 项集之间的转换关系(Action移入)

- 如果项集中有项 $[U \rightarrow x.ay, T]$ ，表示对应于这个项集的猜测需要输入一个 $a$ ，并且，如果 $ay$ 可能推导出 $\alpha$ 的时候，这个项主张 $A[S, \alpha]$ 的动作应该为移入。
- 如果有多个这样的项，那么表示它们都主张下一个操作是移入。由于移入的动作是将当前符号压入栈中，所以相互之间的主张并不冲突。
- 但是，如果同时还有项主张归约，那么就可能形成冲突。
- 当没有冲突的时候，可以将主张移入的项的圆点向右移动一位，表示该猜想离开成功已经又近了一步。而其它项被删除，表示这些项对应的猜想已经失败。
- 例如：当前项集为 $\{[F \rightarrow \cdot(E), +], [F \rightarrow \cdot(E), \#], \dots\}$ ，当前符号为 $($ 。那么它的下一个状态是 **CLOSURE**  $\{[F \rightarrow ( \cdot E), +], [F \rightarrow ( \cdot E), \#], \dots\}$

# 项集之间的转换关系(GO表)

- Go表确定了如果发生归约动作，将归约得到的非终结符号入栈之后，在栈顶的状态。
- 如果在非终结符号U入栈之前，栈顶的状态中有项 $[V \rightarrow x.Uy, a]$ 。这个项表示，它对应的猜想希望输入一个U。因此在下一个状态中，必然有项 $[V \rightarrow xU.y]$ 。
- 如果原状态中有多个这样的项，并不引起冲突。所有这样的项都应该将圆点后移，添加到后继状态中。其余的项对应的猜想已经失败。
- 后继状态为所有添加到表中的项的项集闭包。
- Go表中的动作不会有冲突的情况。

# LR(K)分析表的构造

- 构造初始项集闭包，这个项集在状态集合中。
- 对于每个项集 $S$ ，和每个终结符号串 $\alpha$ ， $S$ 对应于 $\alpha$ 首符号的移入\_后继的闭包也在状态集合中。相应的分析表项 $A[S, \alpha]$ 中为移入，并且后继状态为该后继项集。
- 对于每个项集 $S$ ，如果项集中有项 $[U \rightarrow x., \alpha]$ ，那么分析表中有 $A[S, \alpha] = ri$ 。其中 $i$ 为 $U \rightarrow x$ 对应的规则编号。
- 对于每个项集 $S$ ，和每个非终结符号 $U$ ，它们的移入\_后继的闭包也在项集中。并且，分析表 $G_0$ 中有对应项为 $G[S, U] = \text{后}$ 即对应的集合。

# LR(K)文法的判定

- 如果按照上述算法得到的LR(K)分析表中每一项最多只有一个值，那么这个文法就是LR(K)的。

# LR(K)技术的评价

- 适用面广：几乎可以适用于所有用上下文无关文法描述的程序设计语言，且一般只需要 $K=1$ .
- 效率高：适用确定的下推自动机来实现。
- 错误发现及时：一旦语法有错误，立刻可以发现。便于进行其它出错处理。
- 便于识别程序的自动构造。

# LR(0)方法和SLR(K)技术

- 在LR(K)技术中，每次都需要通过向前看K个符号来确定下一步的动作。导致的结果是，状态非常多。
- 有的时候，不需要向前看就可以确定下一步的动作（归约，还是移入）。因此一个可能的解决办法是：当不能确定下一步动作的时候才向前看K个符号。这个方法称为SLR(K)方法。
- 首先，我们构造不向前看的LR(0)分析表。



# LR(0)分析

- 首先，LR(0)技术不需要向前看符号就可以确定是归约（到哪个符号）还是移入。所以，LR(0)的分析表没有ACTION部分。
- 在LR(0)状态中，只有状态根据当前输入符号的转换关系。这些转换关系由特征自动机(CFSM)表示。
- CFSM的状态是LR(0)项集闭包。

# 特征自动机(CFSM)

- 完备项:  $U \rightarrow u.$ 称为完备项。包括接受项和归约项。如:  $Z \rightarrow E\#.$ ,  $E \rightarrow E+T.$
- 不完备项:  $E \rightarrow E.+T.$
- 接受项:  $Z \rightarrow u.$ 称为接受项。  $Z \rightarrow E\#.$
- 移入项:  $U \rightarrow u.Tv$ , 其中 $T$ 为终结符号。  $E \rightarrow E.+T.$
- 待约项:  $U \rightarrow u.Vv$ , 其中 $v$ 为非终结符号。  $E \rightarrow E+.T.$

# 特征自动机(CFSM)

- 初始项集：  $Z \rightarrow \cdot E \#$  的闭包。
- 后继项： 项集中  $U \rightarrow u \cdot A v$  项对应的后继项为  $U \rightarrow u A \cdot v$ 。
- $\#U \rightarrow u$  后继项集： 项集中有  $U \rightarrow u \cdot$  项时，该项集对应于这个项的后继为：  $\#U \rightarrow u$  后继，简称为规约后继。
- 项集和项集闭包：
- LR(0)项集规范族的构造：
  - 初始项的闭包在规范族里面。
  - 如果  $S_i$  在规范族里面，那么其后继也在。

# 项集规范族的例子

0: 初始项:  $Z \rightarrow .E\#$ , 闭包集合:  $E \rightarrow .E+T$ ;  $E \rightarrow .T$ ;  $T \rightarrow .T*F$ ;  $T \rightarrow .F$ ;  $F \rightarrow .(E)$ ;  $F \rightarrow .i$ 。

0  $\xrightarrow{E}$  1: 基本项:  $Z \rightarrow E.\#$ ;  $E \rightarrow E.+T$ ;

0  $\xrightarrow{T}$  2: 基本项:  $E \rightarrow T.$ ;  $E \rightarrow T.*F$ ;

0  $\xrightarrow{F}$  3: 基本项:  $T \rightarrow F.$

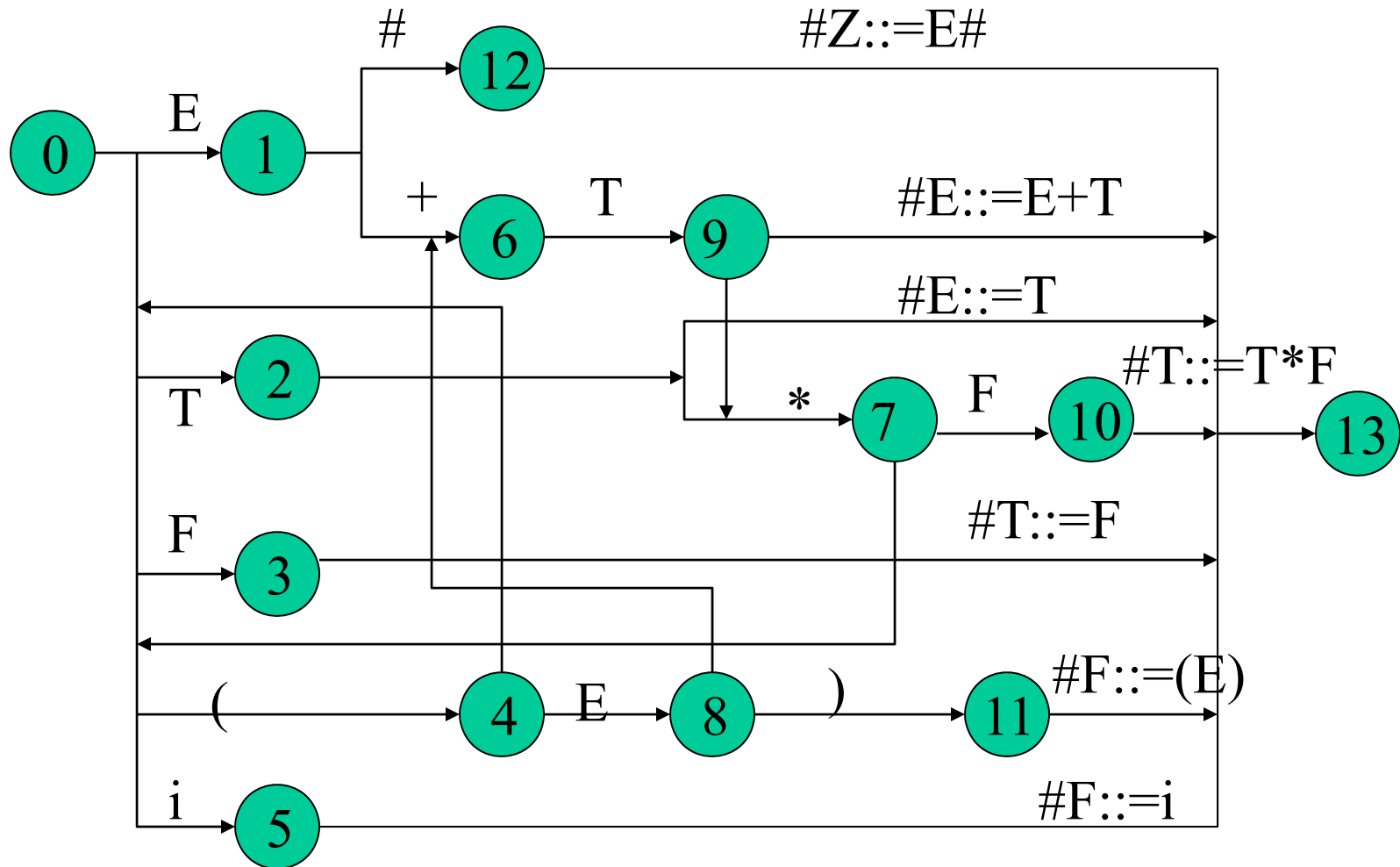
0  $\xrightarrow{(}$  4: 基本项:  $T \rightarrow (.E)$ ; 闭包集合:  $E \rightarrow .E+T$ ;  $E \rightarrow .T$ ; ...

0  $\xrightarrow{i}$  5: 基本项:  $F \rightarrow i.$ ;

# 特征自动机

- 项集规范族和它们之间的转换关系可以用特征自动机描述。方法如下：
  - 将LR(0)项集规范族中的集合作为自动机的状态。
  - 将状态之间的转换关系对应于后继关系。
  - 和初始项集对应的状态为初始状态。和#归约\_后继项集对应的状态作为该FSM的终止状态。

# 特征自动机的例子



# LR(0)项集的分类

- 归约状态：项集中只有一个完备项。这个状态只有一条到归约后继的出弧。
- 读状态：项集中包含的全部是不完备状态。没有到归约后继的出弧。
- 不适定状态：即有完备项，又有不完备项的状态称为不适定状态。这个状态既有到归约后继的出弧，还有到其它状态的弧。

# 利用CFSM判断LR(0)文法

- 一个文法是LR(0)的，当且仅当该文法的CFSM中间没有不适定状态。



# 利用CFSM识别LR(0)句子

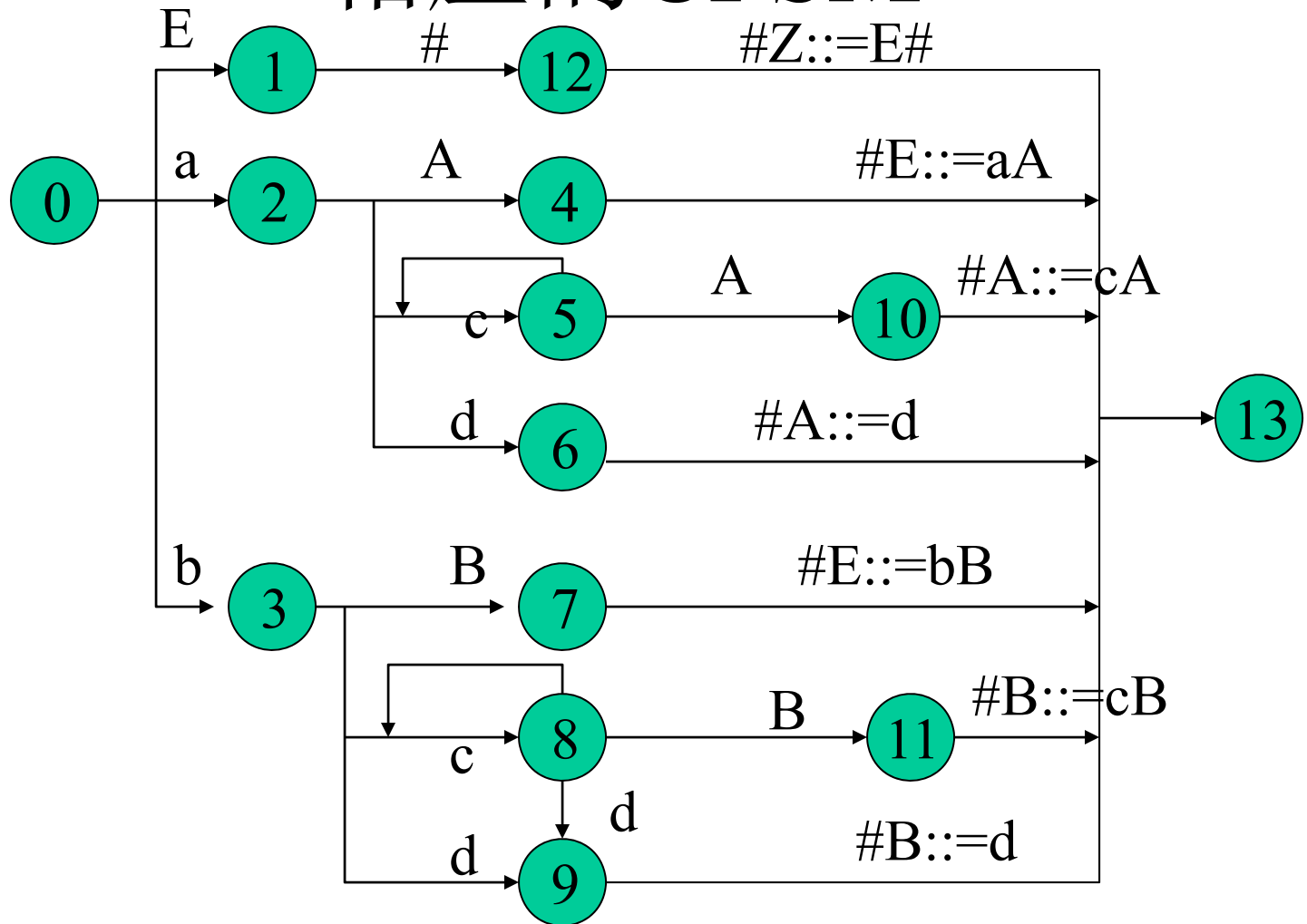
- 将状态0入栈。
- 读入下一个符号，入栈。由CFSM确定栈顶状态，如果不能确定，出错。
- 如果当前的栈顶状态为读状态，goto步骤2；如果是归约状态，根据相应的规则进行归约（此时可以进行相应的语义工作）。归约时，将相应的右部弹出。如果是归约到识别符号，那么句子接受，否则将非终结符号作为输入符号，转步骤2。

# 利用CFSM识别LR(0)文法的句子

- 文法:  $Z ::= E\#$     $E ::= aA \mid bB$     $A ::= Ca \mid d$     $B ::= cB \mid d$

0: 初始项: $Z \rightarrow .E\#$ ; 闭包: $E \rightarrow .aA$ $E \rightarrow .bB$ E_后继: 1      a_后继: 2      b_后继: 3		
1: $Z \rightarrow E. \#$ #_后继: 12		
2: $Z \rightarrow a.A$ ; 闭包: $A \rightarrow .cA$ $A \rightarrow .d$ A_后继: 4      c_后继: 5      d_后继: 6		
3: $E \rightarrow b.B$ ; 闭包: $B \rightarrow .cB$ $B \rightarrow .d$ B_后继: 7      a_后继: 8      b_后继: 9		
4: $E \rightarrow aA.$ #E::=aA_后继: 13		
5: $A \rightarrow c.A$ ; 闭包: $A \rightarrow .cA$ $A \rightarrow .d$ A_后继: 10      c_后继: 5      d_后继: 6		

# 相应的CFSM



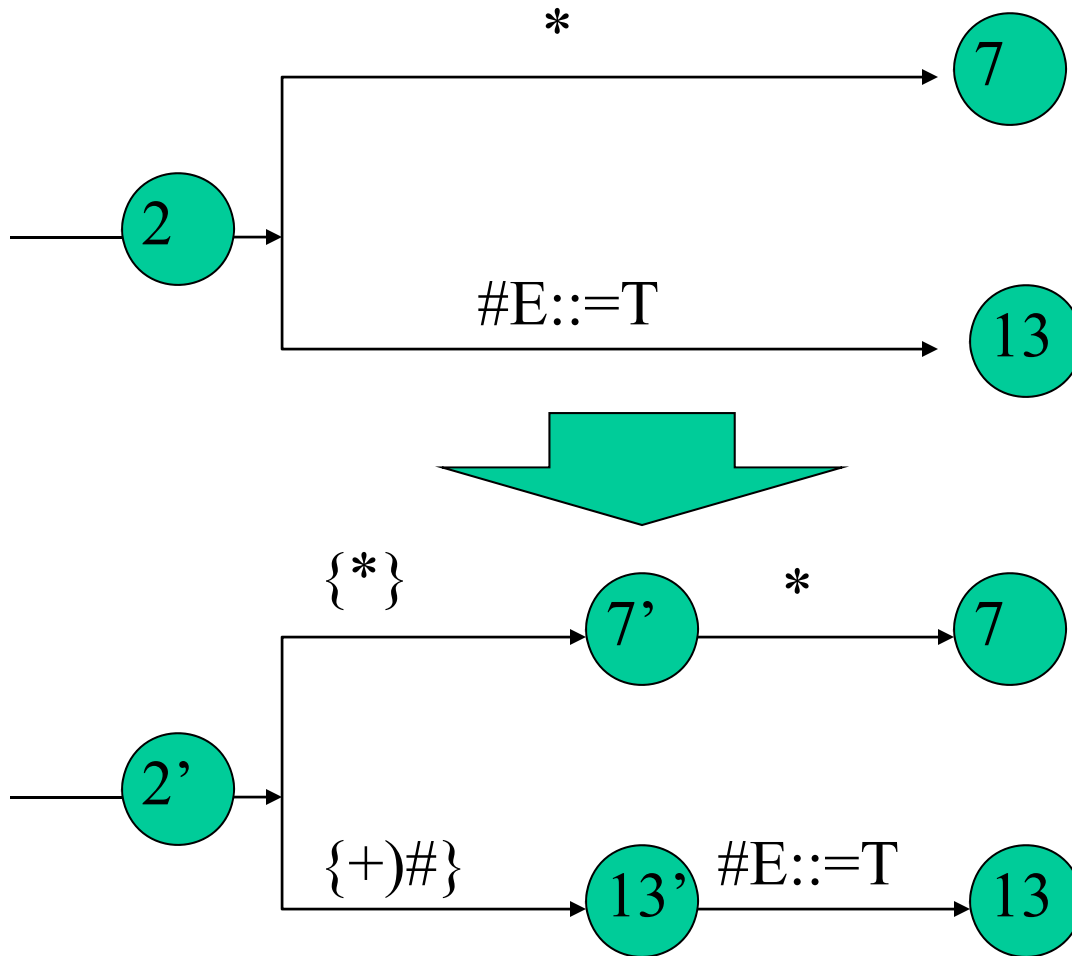
# 识别过程

步骤	栈	其余输入部分
0	0	accd#
1	0a2	ccd#
2	0a2c5	cd#
3	0a2c5c5	d#
4	0a2c5c5d6	#
5	0a2c5c5A10	#
6	0a2c5A10	#
7	0a2A4	#
8	0E1	#

# SLR(K)文法

- 如果某个文法不是LR(0)的，它的特征自动机就存在不适定状态。
- 在识别的过程中，如果栈顶出现了不适定状态，那么它的下一步动作就需要依靠向前看K个符号来确定。
- SLR(K)技术就是依靠简单向前看k的符号来确定是否进行归约。
- 我们要求掌握K=1的情况。

# 简单向前看1集合（示例）



# 简单向前看1集合

- 定义5.25：一个简单向前看1集合是某些文法符号组成的集合，它和CFSM中的一个不适定状态的各个转换相关。
  - 对于文法符号下的转换， $X$ \_转换的简单向前看1就是 $\{X\}$ 。
  - 对于 $\#U::=u$ 转换，简单向前看1集合 $F(U)$ 就是 $FOLLOW(U)$ 。其中 $FOLLOW(U)=\{T|T\in VT\cup\{\#\}, Z\rightarrow\ldots UT\ldots\}$

# 简单向前看1集合(构造例子)

- 对于前面的CFSM，状态2是不适定状态，对于它的简单向前看1集合，存在两个转换： $*\_$ 转换和 $\#E::=T$ 转换。
  - 对于 $*\_$ 转换，简单向前看集合就是 $\{*\}$ 。
  - 对于 $\#E::=T$ 转换，简单向前看1集合就是 $\text{FOLLOW}(E)=\{(+, \#)\}$ 。



# SLR(1)文法的定义

- 一个文法是SLR(1)的，当且仅当其CFSM的每个不定状态的各个 $T$ 转换和 $\#U::=u$ 转换的简单向前看1集合不相交。

# SLR(K)文法

- 对于CFSM中的不适定状态N，出自N的T\_转换和#U::=u转换相关联的各个简单向前看K集合互不相交。
- 简单向前看K集合：
  - 对于#U::=u转换， $F(U) = \{x | Z \Rightarrow^* uUv, x \text{ 是 } v \text{ 的长度 } \leq k \text{ 的头符号串, 且 } x \in (VT + \{\#\})^*\}$
  - 对于T转换， $\{Xv \in VT^* \mid X \text{ 是到状态 } N \text{ 的转换, } v \text{ 属于一个简单向前看 } k-1 \text{ 集合, 这个集合和某个从 } N \text{ 来的 } T_ \text{ 转换或 } \# \text{ 转换相关, } T \in VT\}$

# SLR(1)识别程序的构造(简单描述)

- 对于CFSM的修改：把不适定状态修改为向前看状态，使得对于每个在符号 $X$ 下的状态从 $N$ 到 $M$ 的转换，以及从相联系的简单向前看1集合 $L$ ，都存在有从 $N'$ 到 $M'$ 的一个转换，并且从 $M'$ 到 $M$ 就是原来的转换。
- 识别算法的修改：当栈顶符号为一个不适定状态的时候，算法查看下一个符号，从而确定进入哪一个状态（新状态不入栈）。然后由新的状态确定下一步的动作。

# 活前缀和有效项

- 活前缀：规范句型的一个头符号串，并且不包含句柄右部的任何符号。
  - 比如：句型 $E+T*i+i$ 的活前缀有 $E$ ,  $E+T$ ,  $E+T*$ ,  $E+T*i$ 。
  - $E+T*i+$ 不是活前缀，因为句柄为 $i$ ，并且 $+$ 在 $i$ 的右面。

# 活前缀和有效项

- LR识别过程中，栈里面的符号就是一个活前缀。栈里面的符号添加上适当的终结符号串就可以得到一个句型。
- 有效项：设有文法 $G[Z]$ 的增广文法。如果存在一个 $Z'$ 到 $uUv$ 再到 $uu_1u_2v$ 的规范推导，那么项 $U \rightarrow u_1.u_2$ 对于活前缀 $uu_1$ 是有效的。

# CFSM和活前缀，有效项

- 定理5.16 对于一个活前缀 $r$ 的有效项集合恰好是从初始状态出发，沿着CFSM中标号为 $r$ 的路径到达的状态所对应的项集。
- 在使用LR系列识别文法的句子的时候，其栈顶状态中项主干部分（不包含向前看符号串）的集合就是其所有有效项的集合。

# SLR(1)分析表的构造

- 步骤1：扩充成为增广文法。
- 步骤2：构造CFSM。首先构造项集规范族作为状态集合，然后构造状态之间的转换(GO函数)。
- 构造SLR(1)分析表：
  - 如果有移入项 $U \rightarrow u.av \in S_i$ ，且 $GO(S_i, a) = S_j$ ，那么  $A[i, a] = S_j$ 。
  - 如果有归约项 $U \rightarrow u \in S_i$ ，那么对于每个 $a \in F(U)$ ， $A[i, a] = r_j$ 。
  - 如果 $Z' \rightarrow Z.\# \in S_i$ ， $A[i, \#] = acc$ 。
  - $GO(S_j, U) = S_j$ ，那么 $G[i, U] = j$
  - 其余的为error。
- 如果分析表中有多值元素，那么该文法不是SLR(1)的。

# 例子5.29

- $G[10]: G ::= E\# \quad E ::= E+T \quad E ::= T \quad T ::= T*F \quad T ::= F \quad F ::= (E)$   
 $F ::= i$
- 项集规范族
- $S_0 = \{Z \rightarrow .E\#, E \rightarrow .E+T, E \rightarrow .T, T \rightarrow .T*F, T \rightarrow .F, F \rightarrow .(E), F \rightarrow .i\}$
- $S_1 = \{Z \rightarrow E.\#, E \rightarrow E.+T\}$
- $S_2 = \{E \rightarrow T., T \rightarrow T.*F\}$
- $S_3 = \{T \rightarrow F.\}$
- $S_4 = \{F \rightarrow (.E), E \rightarrow .E+T, E \rightarrow .T, T \rightarrow .T*F, T \rightarrow .F, F \rightarrow .(E), F \rightarrow .i\}$
- $S_5 = \{F \rightarrow i.\}$
- $S_6 = \{E \rightarrow E+.T, T \rightarrow .T*F, T \rightarrow .F, F \rightarrow .(E), F \rightarrow .i\}$
- $S_7 = \{T \rightarrow T*.F, F \rightarrow .(E), F \rightarrow i\}$
- $S_8 = \{F \rightarrow (E.), E \rightarrow E.+T\}$
- $S_9 = \{E \rightarrow E+T., T \rightarrow T.*F\}$
- ... ..



## 例子5.29 (续)

- $GO(S0, E) = S1,$                        $GO(S0, T) = S2$
- $GO(S0, F) = S3,$                        $GO(S0, ( ) ) = S4$
- $GO(S0, i ) = S5,$                        $GO(S1, + ) = S6$
- $GO(S2, * ) = S7,$                        $GO(S4, E) = S8$
- $GO(S4, T) = S2,$                        $GO(S4, F) = S3$
- $GO(S4, ( ) ) = S4,$                        $GO(S4, i ) = S5$
- $GO(S6, T) = S9,$                        $GO(S6, F) = S3$
- $GO(S6, ( ) ) = S4,$                        $GO(S6, i ) = S5$
- $GO(S7, F) = S10,$                        $GO(S7, ( ) ) = S4$
- $GO(S7, i ) = S5,$                        $GO(S8, , ) = S11$
- $GO(S8, + ) = S6,$                        $GO(S9, * ) = S7$

## 例子5.29（续）

- 构造分析表：
  - $A[0,[]]=S4, \quad A[0,i]=S5$
  - $G[0,E]=1 \quad G[0,T]=2$
  - $A[1,\#]=acc$
  - $A[2,+]=A[2,.)]=A[2,\#]=r2$
  - $A[3,+]=A[3,*]=A[3,.)]=r4$
  - ...
- 也可以不写出GO函数而直接构造分析表。

# LALR(K)分析技术

- SLR(K)技术的优点在于：状态个数比较少。因此，分析表也比较小。因此，使用SLR(K)技术使用的内存比较小。
- SLR(K)技术的能力比LR(K)技术弱。原因在于：简单向前看集合不考虑归约的上下文关系。

# SLR(1)技术不能分析的例子

- 文法: 1)  $S ::= L = R$       2)  $S ::= R$     3)  $L ::= *R$     4)  $L ::= i$   
5)  $R ::= L$

0: $Z \rightarrow .S\#$ 闭包: $S \rightarrow .L = R$ $S \rightarrow .R$ $L \rightarrow .*R$ $L \rightarrow .i$ $R \rightarrow .L$ $S\_后继: 1$ ; $L\_后继: 2$ ; $R\_后继: 3$ ; $*\_后继: 4$ ; $i\_后继: 5$ ;
---

1: $Z \rightarrow S. \#$ $\#\_后继: 10$
---------------------------------------

2: $S \rightarrow L. = R$ $R \rightarrow L.$ $=\_后继: 6$ 归约\_后继 11
---

... ..

6: $S \rightarrow L = .R$ 闭包: $R \rightarrow .L$ $L \rightarrow .*R$ $L \rightarrow .i$ ... ..
--

... ..

11: 归约
--------

# SLR(1)技术不能分析的例子(续)

- 项集2显然是一个不适定状态。
- 而且，由于 $= \in F(R)$ ，而且另外一个转换( $=$ \_转换)的简单向前看1符号集合也是 $\{=\}$ 。因此，分析表 $A[2,=]$ 中会有两个值。所以，这个文法不是SLR(1)文法。
- 而且，对于任意的 $K$ ，这个文法不可能是SLR( $K$ )文法。原因在于： $S \rightarrow L=R \rightarrow *R=R$ 。因此，不管往前面看多少个， $=R$ 对应的字的前 $K$ 个符号在 $FOLLOW_k(R)$ 中，同时，它也在 $=$ \_转换的简单向前看 $k$ 集合中。

# LALR技术

- 目标是得到一个比较好的分析表，使得它的状态比较少，但是有比SLR更加强大的分析能力。
- 显然，对于每个移入项，它的向前看符号集合是不变的。
- SLR技术的缺点在于，它生成归约项的向前看符号集合的时候，不考虑上下文。假设某个LR(0)项集中有归约项 $U \rightarrow u.$ ，那么它对应的将 $u$ 归约后的活前缀为 $\dots U$ 。
- 在活前缀 $\dots U$ 对应的规范句型中，可能允许跟在 $U$ 后面的符号是在所有句型中跟在 $U$ 后面的符号的子集。
- 因此，我们可以考虑求出这样的子集。从而减少归约项的向前看符号集合和其它的向前看符号集合之间的冲突可能性。这就是LALR技术。

# 非SLR文法例子的分析

- 在状态2中，有两个项 $S \rightarrow L.=R$ 和 $R \rightarrow L.$ 。
- 从这个图中，我们可以看到：对应于状态2的活前缀只有符号串L。
- 而按照项 $L \rightarrow R.$ 进行归约后，得到的活前缀为R。以R为活前缀时，其后只可能跟#。
- 而=根本不可能跟在活前缀R后面。
- 因此，我们发现，实际上，如果下一个符号为=时，相应的动作应该时移入。

# LALR的基本步骤

- 首先生成LR(1)项集规范族。
- 合并同心项集。并且相应地建立合并后的项集之间的转换关系。合并后的项集的个数和LR(0)项集规范族中的项集个数相同。
- 根据这个项集族，得到分析表。



# LR(1)项集规范族的例子

$$I_0 = \{[Z \rightarrow \cdot S, \#], [S \rightarrow \cdot L = R, \#], [S \rightarrow \cdot R, \#], [L \rightarrow \cdot *R, =], [L \rightarrow \cdot i, =| \#], [R \rightarrow \cdot L, \#]\}$$

$$I_1 = GO(I_0, S) = \{[Z \rightarrow S \cdot, \#]\}$$

$$I_2 = GO(I_0, L) = \{[S \rightarrow L \cdot = R, \#], [R \rightarrow L \cdot, \#]\}$$

$$I_3 = GO(I_0, R) = \{[S \rightarrow R \cdot, \#]\}$$

$$I_4 = GO(I_0, *) = \{[S \rightarrow * \cdot R, =| \#], [R \rightarrow \cdot L, =| \#], [L \rightarrow \cdot *R, =| \#], [L \rightarrow \cdot i, =| \#]\}$$

...      ...      ...      ...

$$I_{11} = GO(I_6, *) = \{[S \rightarrow * \cdot R, \#], [R \rightarrow \cdot L, \#], [L \rightarrow \cdot *R, \#], [L \rightarrow \cdot i, \#]\}$$

# 同心项集

- 对于两个LR(1)项集，如果除了搜索符（向前看符号）外，其它完全相同，那么这两个项集被成为同心项集。
- 同心项集的T\_后即仍然是同心项集。
- 合并同心项集：
  - 合并后的项集的基本部分保持不变。基本部分相同项的搜索符被合并。
  - 原来的同心项集之间的转换关系依然不变。

# 同心项集合并的例子

- I4和I11是同心项集。
  - $I4 = \{[L \rightarrow *.R, =| \#], [R \rightarrow .L, =| \#], [L \rightarrow .*R, =| \#], [L \rightarrow .i, =| \#]\}$
  - $I11 = \{[L \rightarrow *.R, \#], [R \rightarrow .L, \#], [L \rightarrow .*R, \#], [L \rightarrow .i, \#]\}$
- 合并后为：
  - $I4' = \{[L \rightarrow *.R, =| \#], [R \rightarrow .L, =| \#], [L \rightarrow .*R, =| \#], [L \rightarrow .i, =| \#]\}$
- 转换关系为：
  - $GO(I4', R) = I7'$  (I7, I13合并后的同心项)
  - $GO(I4', L) = I8'$  (I8, I10合并)
  - $GO(I4', *) = I4'$
  - $GO(I4', i) = I5'$  (I5和I12合并)

# 合并可能引起的冲突

- 合并引起的冲突是指：本来的LR(1)项集没有冲突，而合并同心项集后有冲突。
- 不可能引入归约-移入型冲突。
  - 假定归约后的有移入\_归约冲突，就是说：有项  $[U \rightarrow u., a]$  和项  $[V \rightarrow v.aw, b]$ 。显然，原来的项集中都有  $[V \rightarrow v.aw, ?]$ 。而  $[U \rightarrow u., a]$  必然在某个原来的项集中。这样， $[U \rightarrow u., a]$  所在的项集必然已经冲突了。
- 但是可能引起归约\_归约冲突。

# 引入归约\_归约冲突的例子

- $G_{5.13}[S]: S ::= aAd|bBd|aBe|bAe \quad A ::= c \quad B ::= c。$
- $I_{58} = \{[A \rightarrow c., de], [B \rightarrow c., de]\}$

0:  $\{[S \rightarrow .aAd, \#] [S \rightarrow .bBd, \#] [S \rightarrow .aBe] [S \rightarrow .bAe]\}$

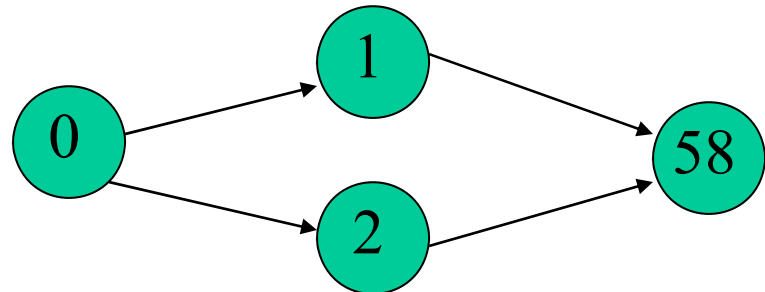
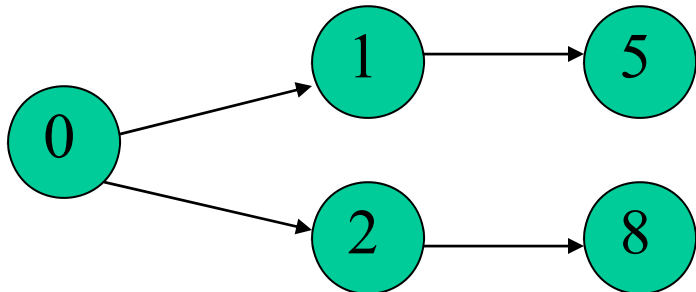
a\_后继: 1, b\_后继: 2

0:  $\{[L \rightarrow a.Ad, \#] [S \rightarrow a.Be] [A \rightarrow .c, d] [B \rightarrow .c, e] \}$  c\_后继: 5。

2:  $\{[L \rightarrow b.Bd, \#] [S \rightarrow b.Ae] [A \rightarrow .c, e] [B \rightarrow .c, d]\}$  c\_后继: 8。

5:  $\{[A \rightarrow c., d] [B \rightarrow c., e]\}$

8:  $\{[A \rightarrow c., e] [B \rightarrow c., d]\}$



# LALR分析表构造算法

- 构造增广文法。
- 构造LR(1)项集规范族 $C=\{I_0, I_1, I_2, \dots, I_n\}$ 。
- 合并同心项集，并用合并后的项集替代 $C$ ，得到 $C'=\{J_0, J_1, \dots, J_n\}$ 。
- 查看 $C'$ 中有没有冲突。如果有冲突，文法不是LALR(1)文法。
- 给 $J$ 中的每个项集编号，作为状态。并建立分析表如下：
  - $[U \rightarrow u.av, ?] \in J_i, a \in V_t, Go[J_i, a] = J_j, A[i, a] = S_j$
  - $[U \rightarrow u., a] \in J_i, a \in V_t, A[i, a] = r_j$
  - $[Z' \rightarrow Z., \#]$ 在 $J_i$ 中，则 $A[i, \#] = acc.$
- 建立GOTO表。

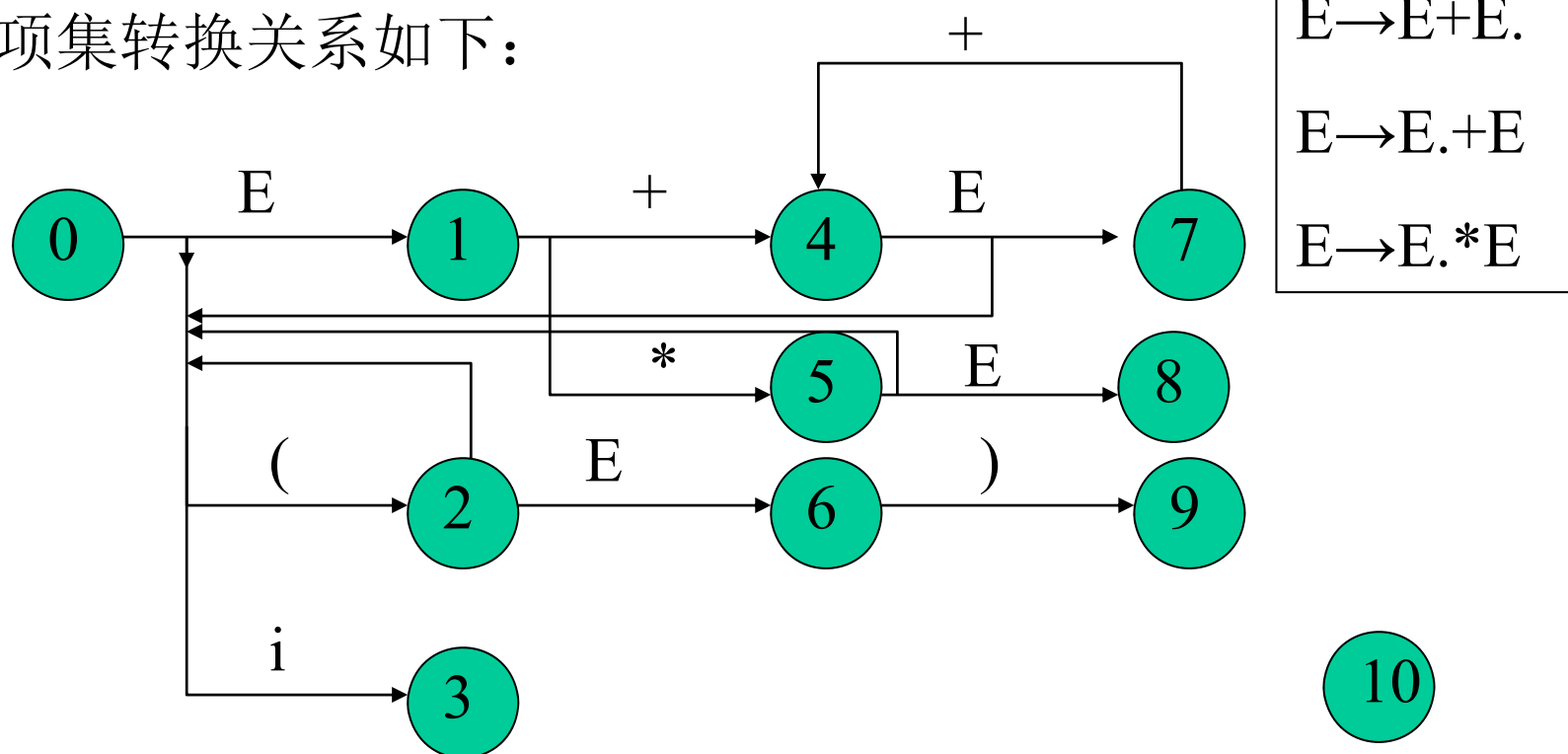
# LR(k)分析表的修改

- 由于LR文法不是二义性的，因此，任何二义性文法都不是LR文法，SLR文法或者LALR文法。
- 所以在相应的分析表中，必然有冲突项。
- 在适当的时候，我们可以修改这些冲突项目，来消除分析的‘二义性’。比如删除不适定状态中的某些项目，或者修改向前看符号集合。
- 修改某个状态的时候，必须理解该项集的意义。

# LR(k)分析表的修改（续）

- 文法： $Z ::= E\#$                        $E ::= E + E$                        $E ::= E * E$   
 $E ::= (E)$                        $E ::= i$

- 项集转换关系如下：



修改如下：在状态7，如果下一个符号为 $+$ ，归约。否则移入。





# 编译原理讲义

## (第六章:语义分析和 目标代码生成)

南京大学计算机系

赵建华

# 概念

- 编译程序的作用是：将源程序转换为具有 **相同效果** 的可运行程序。
- 所谓 **相同效果** 就是程序的语义。
- 并不是所有满足语法规则的程序都是有意义的(well-form)的。
- 所谓语义分析，就是确定程序是有意义的，分析程序的含义，并做出相应的处理。
- 程序的含义包括：
  - 数据结构含义：和标识符相关联的数据对象。
  - 控制结构的含义：由语言定义。规定了每个语句的执行效果。

# 基本功能

- 确定类型：确定标识符所关联的数据对象的数据类型。
- 类型检查：按照语言的类型规则，对运算及分量进行类型检查，必要时做出相应类型转换。
- 识别含义：根据程序设计语言的语义定义，确定各个构造部分组合后的含义，做出相应处理（生成中间代码或者目标代码）。
- 其它静态语义检查：比如控制流检查，嵌套层数检查。

# 语义分析的输入输出

- 输入为前面语法分析的结果。
- 输出为中间表示代码(抽象语法树，三元式...).
  - 把生成中间代码和代码生成分开，可以使难点分解。
  - 对中间代码的分析比较容易，便于进行优化。
  - 可以把编译程序分成机器独立部分和机器相关部分。
  - 便于任务的分解，和开发人员的分组开发。

# 属性文法

- 对于某个压缩了的文法，当把每个文法符号和一组属性相关联，并把重写规则附加以语义规则的时候，就得到属性文法。
- 语法制导的翻译过程：由于属性文法的规则和重写规则是一一对应的关系。所以，由属性文法确定的语义分析可以在语法分析的过程中进行。这个过程称为语法制导的翻译。
- 语法制导的定义/翻译方案：语法制导的定义比较抽象，隐藏了实现细节，无需指明语义规则的计算次序。翻译方案则指明的计算次序。

# 语法制导定义的例子

$L ::= E_n$	$\text{Print}(E.\text{val})$
$E ::= E_1 + T$	$E.\text{val} = E_1.\text{val} + T.\text{val}$
$E ::= T$	$E.\text{val} = T.\text{val}$
$T ::= T_1 * F$	$T.\text{val} = T_1.\text{val} * F.\text{val}$
$T ::= F$	$T.\text{val} = F.\text{val}$
$F ::= (E)$	$F.\text{val} = E.\text{val}$
$F ::= \text{digit}$	$F.\text{val} = \text{digit.lexval}$

- 对于语法符号E,T,F，都确定了属性val，表示它们的值。
- 如果一个重写规则中有两个相同的符号，需加足标区别。
- 在归约过程中，每个归约得到的符号都对应于输入符号串中的某一段。

## 语法制导定义的例子（2）

$D ::= TL$	$L.in = T.type$
$T ::= int$	$T.type := integer$
$T ::= real$	$T.type := integer$
$L ::= L_1, id$	$L_1.in := L.in; addtype(id.entry, L.in)$
$L ::= \text{空}$	

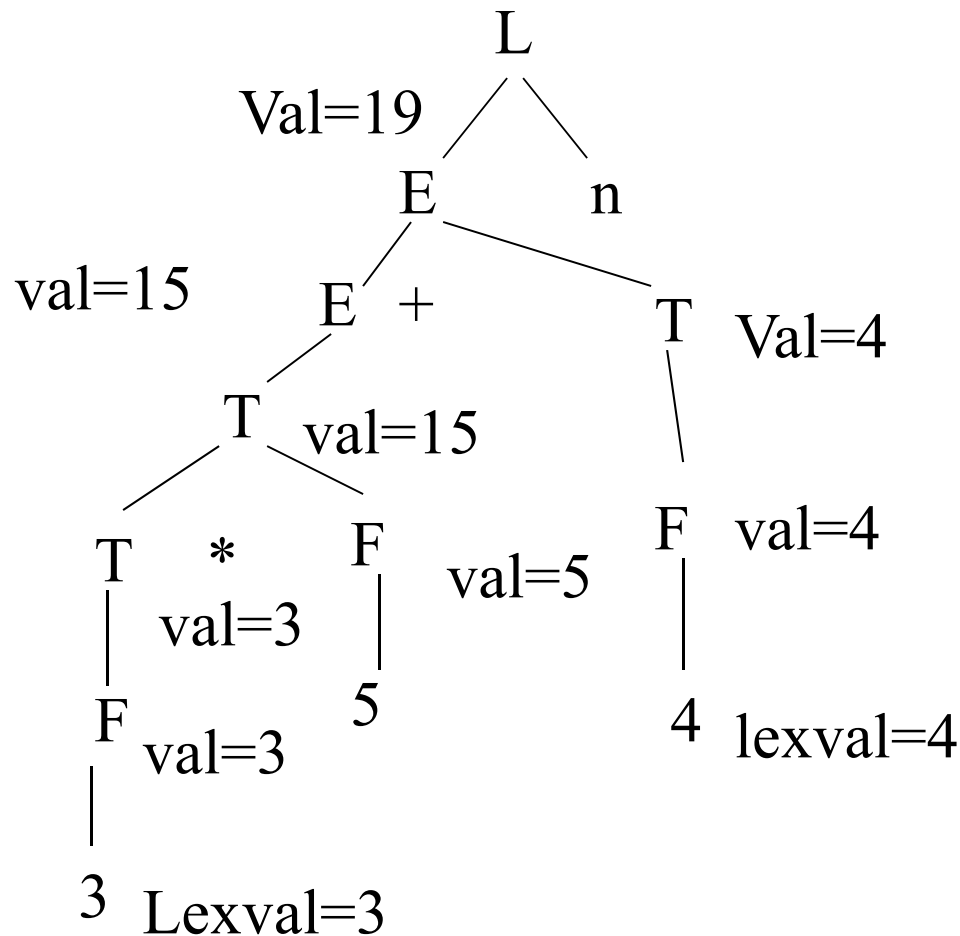
- L的属性包括:in; T的属性值为type
- addtype表示将id对应的类型信息加入到标识符表中。



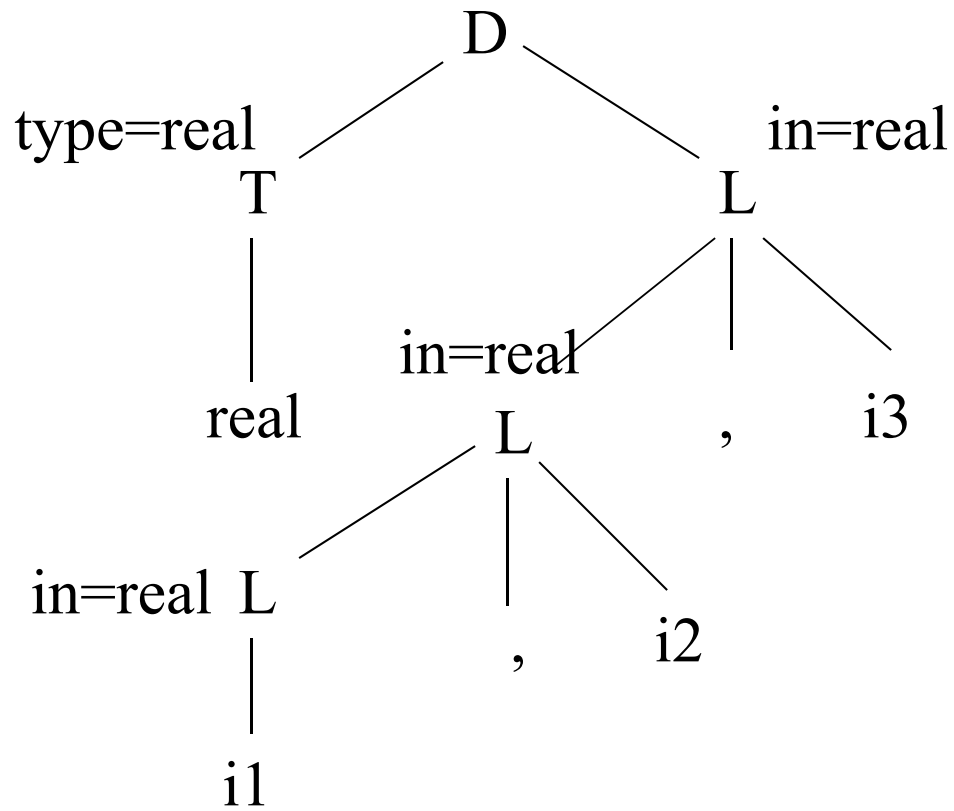
# 属性的计算和注释分析树

- 在语法制导分析过程中，我们不仅生成了一棵语法树，还可以得到各个结点的属性值。把这些值加入到语法树上面之后，就得到了注释分析树。
- 结点的属性值的计算过程有两种类型：
  - 结点的属性由其子结点确定，称为综合属性。
  - 结点的属性由兄弟结点和父结点的属性确定，称为继承属性。
- 在概念上，翻译过程如下：
  - 1：分析输入符号，建立语法树
  - 2：从语法分析树得到描述结点属性之间的依赖关系，得到计算次序。
  - 3：按照语义规则计算属性值。

# 分析过程和注释分析树例子(1)



# 分析过程和注释分析树例子(2)



# 属性文法

- 一个属性文法AG是一个四元组 $(G, A, R, C)$ ，其中
  - $G$ 是已经压缩的文法。
  - $A$ 是有穷属性集合，对于每个文法符号 $X$ ，都有其属性集合 $A(X)$ 对应。
  - $R$ 是有穷属性规则集合。对于规则 $p: X_0 = X_1 X_2 \dots X_k$ ，有规则 $X_i.a = f(X_j.b, \dots, X_{pk}.c)$ 。定义了在使用这个规则进行归约的时候， $X_i.a$ 的求值。
  - $C$ 是有穷条件集合。 $C(X_i.a, \dots, X_j.a)$ 表示规则 $p$ 必须满足的条件。只有这些条件满足的时候，输入的符号串才是有意义的。

# 属性分类

- 综合属性：由相应语法分析树中的子结点的属性计算得到。
- 继承属性：由相应语法分析树中结点的兄弟结点或者父结点的属性来计算而得到。

# 语法制导定义的分类

- S\_属性定义：所有的文法符号的属性都是综合属性。
  - 实际上是说：一个语法结构的属性由它的组成部分的属性确定。例如：对于表达式的结果类型；
  - 基于S\_属性定义的语法分析树可以用自底向上的方法得到。
  - 规则 $E ::= E_1 + T$ 对应的语法结构， $E.type$ 的值可以如下确定：如果 $E_1$ 的类型和 $T$ 的类型都是`int`，那么结果也是`int`，如果有一个是`real`，那么结果类型是`real`。...

# 继承属性

- 语法分析树中，一个结点的属性由其父结点和兄弟结点确定。
  - 反映了文法结构的属性对上下文的依赖特性。
  - 比如：对于C语言中变量的申明可以是如下方式，  
TYPE v1, v2; 那么变量v1, v2的属性值‘类型’是由TYPE确定的，如果TYPE是float，那么它们的类型就是实数。
  - 在前面的例子6.2中，结点T的属性type是综合属性，但是D::=TL中，L的属性in就是继承属性，这个属性的值是由T的属性type确定的。

# 依赖图

- 在语义规则中，假设某个属性的定义是 $b=f(c_1, c_2, \dots, c_n)$ ，那么要求得属性 $b$ 的值，首先要计算出属性 $c_1, c_2, \dots, c_n$ 的值。显然它们的计算顺序有一种依赖关系。
- 通常，我们可以在语法分析树中添加有向的箭头表示这种依赖关系。（对于过程调用，可以添加虚拟的属性）
- 在语法分析树中，如果属性 $b$ 依赖于 $c$ ，那么从 $c$ 对应的结点到 $b$ 对应的结点画一个箭头。

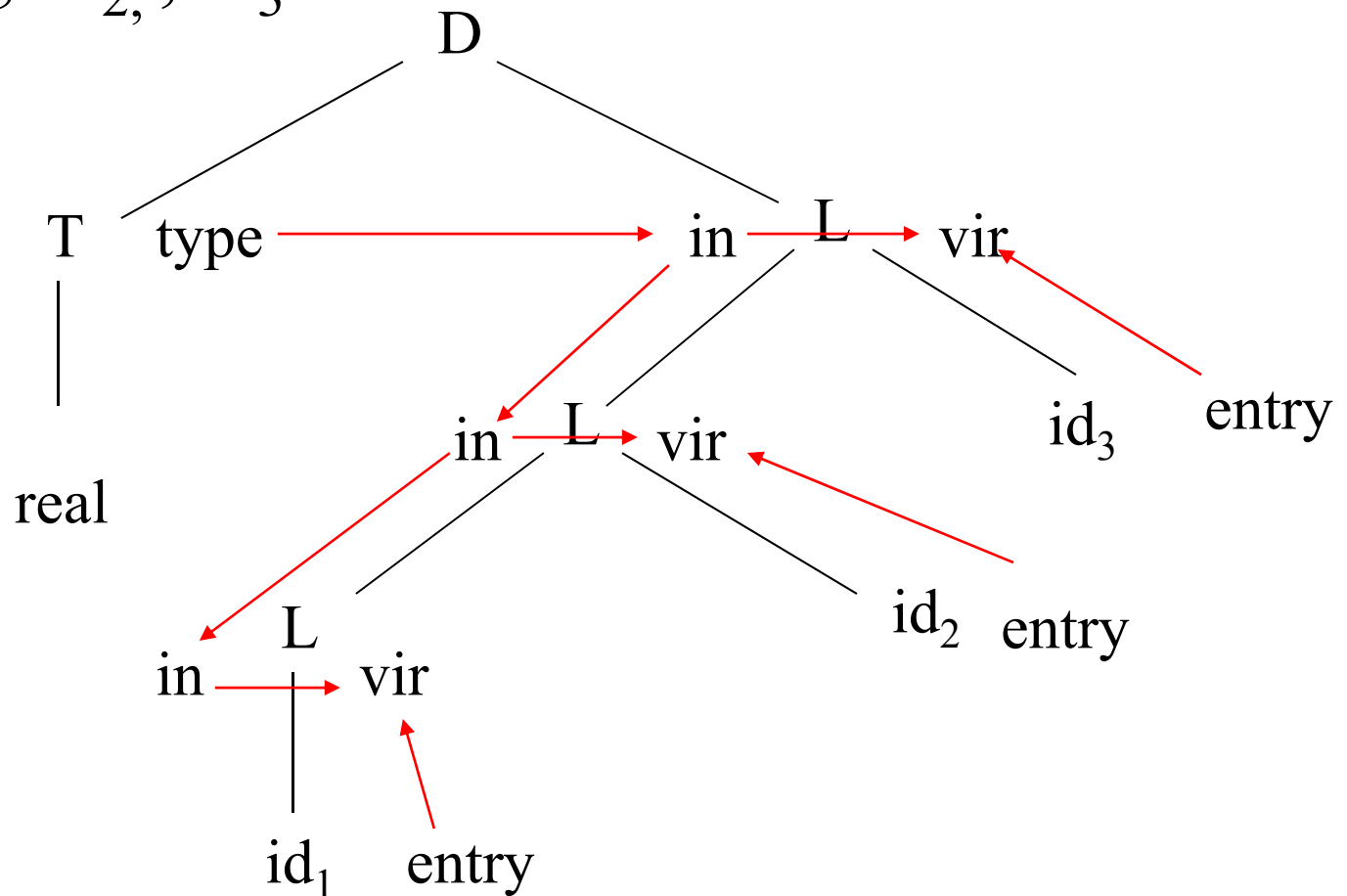


# 依赖图构造算法

- 步骤1：构造语法树，对于语法树中的每个结点的每个属性，构造一个结点。
- 步骤2：对每个树的结点 $n$ ，对该结点所相关的每个语义规则 $b:=f(c_1, c_2, \dots, c_n)$ 都如下添加有向弧：
  - 从 $c_1, c_2, \dots, c_n$ 的结点向 $b$ 对应的结点画有向弧。
- 实际上是对所有在建立语法树中所使用的规则都建立依赖关系。

# 依赖图的例子

- real  $id_1, id_2, id_3$



# 计算次序

- 显然，依赖图中指定了各个属性的计算顺序。
- 各个属性的计算顺序可以有多种，但是必须满足依赖图规定的拓扑次序。以保证在计算某个属性的值的时候，这个属性所依赖的属性值已经被计算出来。
- 用语法制导定义说明的翻译步骤如下：
  - 构造语法分析树。
  - 构造依赖图。
  - 对依赖图的各个结点进行拓扑排序，得到计算次序。
  - 按照上述次序计算属性的值。
- 比如，上面的图的计算顺序如下：
  - `a4=real, a5=a4, addtype(id3.entry, a5), ...`

# 其它得到计算次序的方法

- 避免在分析的过程中显式地构造分析树。
- 忽略规则：按照语法分析的归约（或者）推导程序完成属性的计算。
  - 比如，当定义中只有S\_属性定义的时候，可以按照自底向上归约过程进行。
- 基于规则的计算次序：通过对重写规则和语义规则进行分析，确定和每个重写规则相关联的语义规则的计算次序。

# 翻译方案

- 语法制导定义基于属性文法，其计算次序被隐藏。
- 如果在进行语义定义的时候，陈述了一些实现的细节，我们称为翻译方案。
- 在翻译方案中，语义定义部分是用{}包含的语义动作。并且语义动作可以出现在规则右部的任何地方。
- 这些语义动作的执行顺序是这样的。当建立了一个语法分析树之后（把语义动作也当作规则右部的结点加入到语法树中），我们按照深度优先的方法遍历这个分析树。每次遍历到一个语义动作对应的结点的时候，执行这个动作，计算相应的属性值。
- L\_属性定义是可以使用翻译方案计算的一类语义定义方式。

# L\_属性定义

- L\_属性定义：如果每个重写规则 $U ::= X_1 X_2 \dots X_n$ 对应的语义规则中的每个属性：
  - 或者是综合属性，
  - 或者是这样的继承属性：
    - $X_j.a$ 或者仅仅依赖于 $U$ 。
    - $X_j.a$ 依赖于 $X_j$ 左边的符号的属性。
- S\_属性定义也是L\_属性定义。
- 显然，对于L\_属性定义，其属性的值可以按照深度优先的次序计算。也就是说，可以直接地使用翻译方案实现。
- L\_属性定义的例子可以看关于变量声明的定义。

# L\_属性定义的属性计算

L\_dfvisit(n)

{

for m=从左到右的n的每个子节点 do

{

计算m的继承属性;

L\_dfvisit(m);

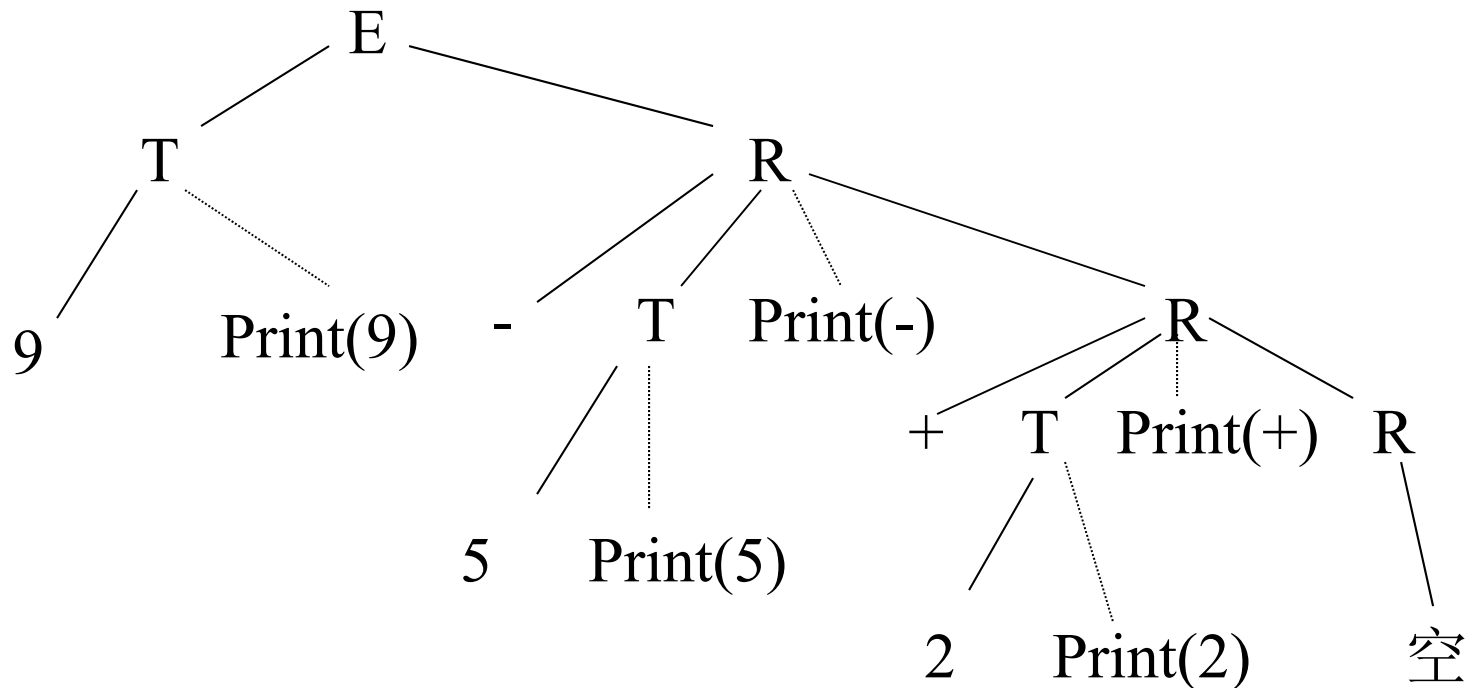
}

计算n的综合属性。

}

# 翻译方案的例子

- $E ::= TR$
- $R ::= \text{addop } T \{ \text{print}(\text{addop.lexeme}) \} R1 \mid \text{空}$
- $T ::= \text{num} \{ \text{print}(\text{num.lexval}) \}$
- 同时，翻译方案设计需要满足L\_属性定义。





# 翻译方案的设计

- 对于综合属性，只需要把属性的定义修改称为赋值语句，放在规则的最末尾就可以了。
- 存在继承属性的时候，必须满足以下条件：
  - 重写规则右部符号的属性必须先于这个符号的语义动作中计算。
  - 一个语义动作不能引用右边符号的综合属性。
  - 重写规则左部的符号的综合属性必须在它所引用的所有属性都计算完成之后才可以计算。
- 错误的例子： $S ::= A_1 A_2 \{A_1.in=1, A_2.in=2\}$   $A ::= a$   
 $\{print(A.in)\}$ .应该把 $\{A_1.in=1\}$ 移到最前面。
- L\_属性定义总存在相应的翻译方案。

# 翻译方案的实现(自底向上)

- S\_属性定义的翻译方案可以在分析输入符号串的时候完成。
- 通过在栈中的每个符号添加一个属性项来完成。
- 在移入的时候，把文法符号的属性同时压栈。终结符号的属性在词法分析的时候获得。
- 归约的时候，除了进行原来的工作之外，需要根据重写规则和语义规则，根据栈顶部的符号属性计算出被归约得到的符号的属性值。
- 比如，如果需要按照 $U ::= XYZ$ 归约，而相应的语义动作为 $\{U.a = f(X.x, Y.y, Z.z)\}$ ，那么栈顶部的符号为ZYX，从这些符号的属性部分得到X.x, Y.y和Z.z就可以计算得到U.a。当把U压入栈的时候，同时把U.a的值压入属性栈。

# 实例

步骤	栈	输入	规则
1	(#,-)	3*5+4n	
2	(#,-)(3,3)	*5+4n	
3	(#,-)(F,3)	*5+4n	F::=digit
4	(#,-)(T,3)	*5+4n	T::=F
5	(#,-)(T,3)(*,-)	*5+4n	
6	(#,-)(T,3)(*,-)(5,5)	5+4n	
7	(#,-)(T,3)(*,-)(F,5)	+4n	F::=digit
8	(#,-)(T,15)	+4n	T::=T*F
9	(#,-)(E,15)	+4n	E::=T
10	(#,-)(E,15)(+,-)	输入	

# 非S\_属性定义的例子

- $D ::= T \{L.in = T.type\} L$
- $T ::= \text{int} \{T.type = \text{integer}\}$
- $T ::= \text{real} \{T.type = \text{real}\}$
- $L ::= \{L1.in = L.in\} L1, \text{id} \{\text{addtype}(\text{id.entry}, L.in)\}$
- $L ::= \text{id} \quad \{\text{addtype}(\text{id.entry}, L.in)\}$
- L\_属性定义翻译方案不能直接用自底向上的方法实现。

# 自底向上的方法

- 在YACC中通过添加空规则来实现。比如第一个规则修改为 $D ::= TAL$ ,  $A ::= \text{空}$ , 但是问题是: 可能变成非LR(K)文法, 并且, 不能直接使用上面的定义。
- 比如:  $D ::= T \{L.in := T.type\} L$ , 实现的方式实际是:  $D ::= TAL$ ,  $A ::= \text{空} \{????\}$ 。
- 但是, 值的传递不是那么方便。

# 自顶向下的实现方法

- 可以按照前面L\_属性定义的计算算法计算的方法完成。
- 但是，需要消除左递归。消除左递归的时候，牵涉到语义规则的修改。

# 自顶向下的实现方法（例子）

- $E ::= E+T | E-T | T$        $T ::= (E) | \text{num}$
- 翻译方案如下：
  - $E ::= E+T$        $\{E.\text{val} = E1.\text{val} + T.\text{val}\}$
  - $E ::= E-T$        $\{E.\text{val} = E1.\text{val} - T.\text{val}\}$
  - $E ::= T$        $\{E.\text{val} = T.\text{val}\}$
  - $T ::= (E)$        $\{T.\text{val} = E.\text{val}\}$
  - $T ::= \text{num}$        $\{T.\text{val} = \text{num.lexval}\}$
- 这个翻译方案中，所有的属性都是综合属性。

# 自顶向下的实现方法（例子）

- 由于自顶向下的方法不能处理左递归的情况，所以必须修改相应的重写规则。而语义规则也需要相应的修改。
- $E ::= T \{R.i = T.val\} R \{E.val = R.s\}$
- $R ::= +T \{R1.i = R.i + T.val\} R1 \{R.s = R1.s\}$
- $R ::= -T \{R1.i := R.i - T.val\} R1 \{R.s = R1.s\}$
- $R ::= \text{空} \{R.s = R.i\}$
- $T ::= (E \{T.val = E.val\})$
- $T ::= \text{num} \{T.val = \text{num.lexval}\}$



# 实现程序 (1)

- $E ::= T \{R.i = T.val\} R \{E.val = R.s\}$

```
S_attribute E(L_attribute)
{
  attribute E_val;
  attribute T_val = T();
  attribute R_i = T_val;
  attribute R_s = R(R_i);
  E_val = R_s;
}
```

## 实现程序 (2)

- $R ::= +T \{R1.i = R.i + T.val\} R1 \{R.s = R1.s\}$

$R(attribute\ in)$

{

    GetSym();

    attribute T\_val = T();

    attribute R1\_i = in + T\_val;

    attribute R\_s = R(R1\_i);

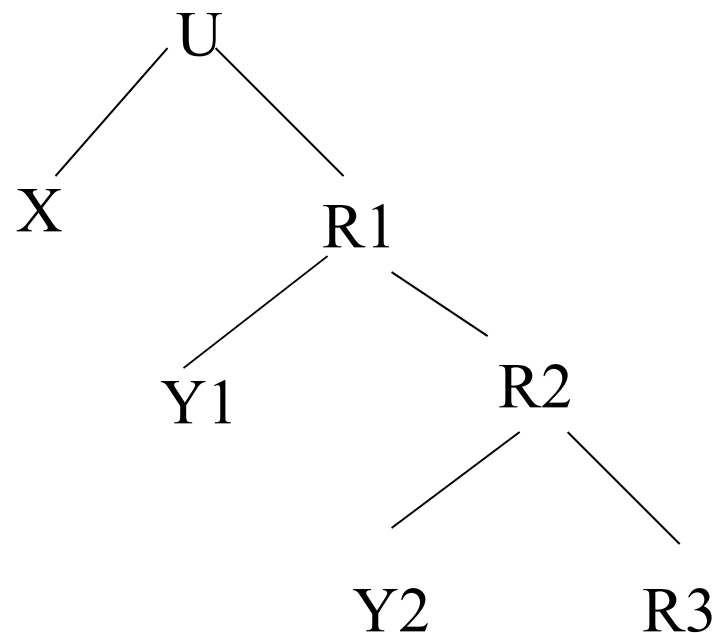
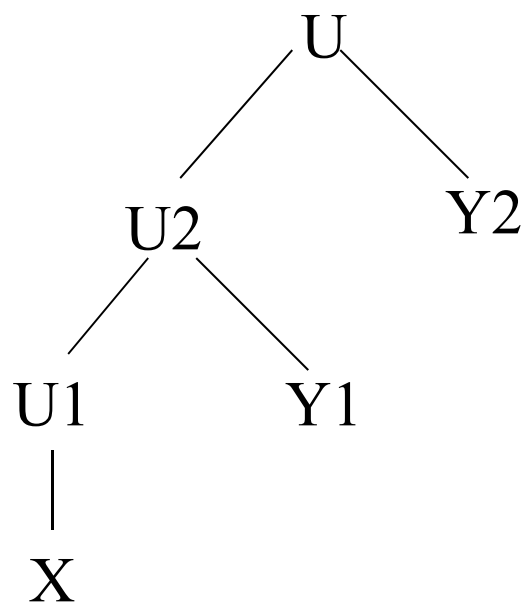
    return R\_s;

}

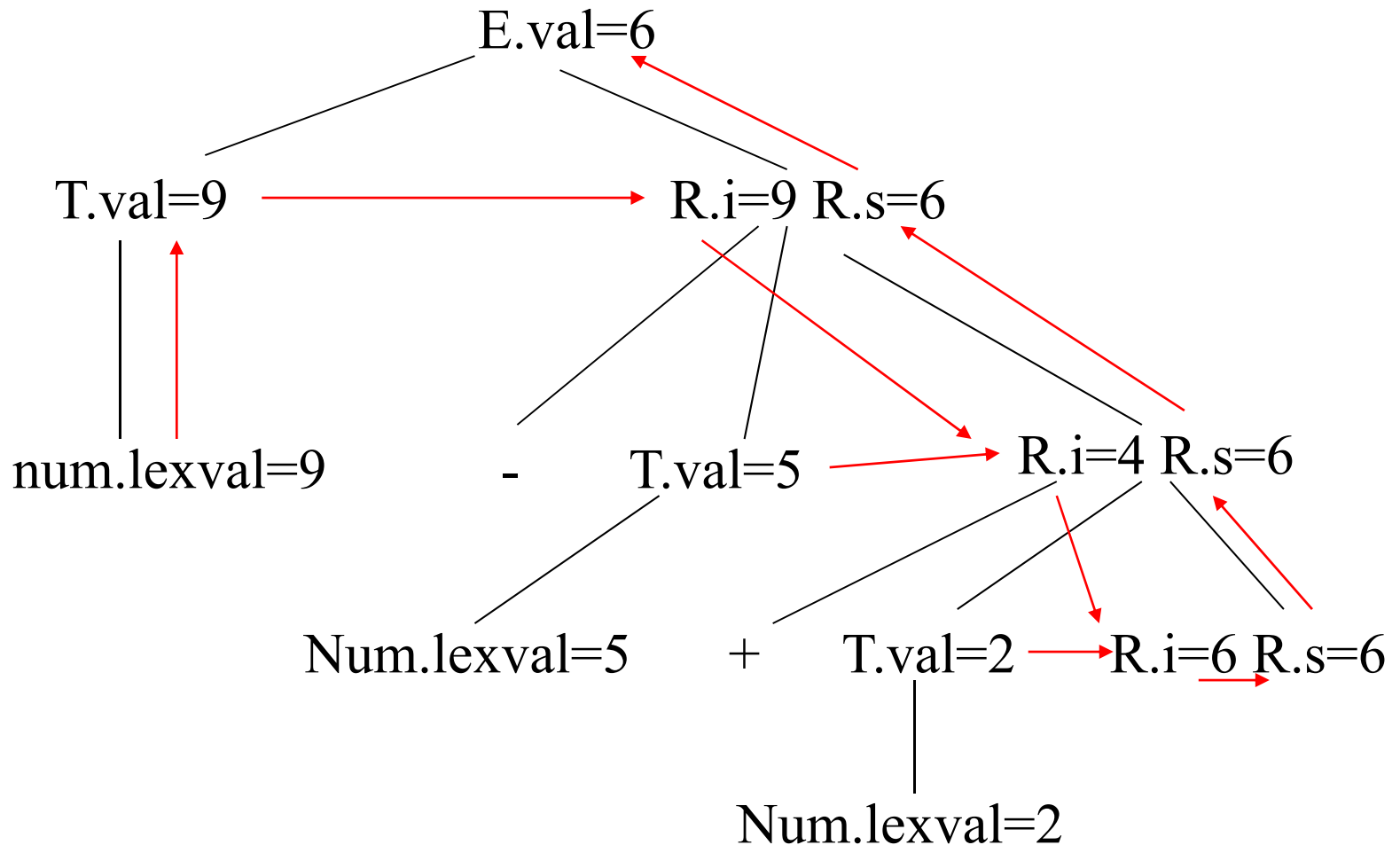
# 消除左递归时翻译方案的修改

- $U ::= U1Y \quad \{U.a = g(U1.a, Y.y)\}$
- $U ::= X \quad \{U.a = f(X.x)\}$
- 修改为:
  - $U ::= X \quad \{R.i = f(X.x)\} \quad R \{U.a = R.s\}$
  - $R ::= Y \quad \{R1.i = g(R.i, Y.y)\} \quad R1 \{R.s = R1.s\}$
  - $R ::= \text{空} \quad \{R.s = R.i\}$
- 引入新的符号R，对于R有继承属性i。
- R.i纪录了前面扫描过的U的a属性值。

# 关于语义定义的解释



# 修改后的翻译方案的计算过程



# 类型体系

- 指明了程序中，运算的合法型和运算分量类型的一致性（相容性）；类型转换的规则等。
- 本部分说明如何使用翻译方案来实现类型的检查。
- 类型的获得（变量说明）由变量申明部分确定。

# 类型表达式

- 使用类型表达式来表示程序中可能出现个各种类型。
- 定义：
  - 基本类型就是类型表达式。
  - 对类型表达式命名的类型名是类型表达式。
  - 类型构造符作用于类型表达式的结果是类型表达式。
    - 数组：  $\text{array}(I, T)$
    - 卡氏积：  $T_1 \times T_2$
    - 纪录：  $\text{record}((N_1 \times T_1) \times (N_2 \times T_2) \times \dots \times (N_n \times T_n))$
    - 指针：  $\text{pointer}(T)$
    - 函数：  $D_1 \times D_2 \times \dots \times D_n \rightarrow R$
  - 类型表达式可以包含变量（如果允许类型重载）

# 类型表达式例子

- `int a[200];      array(0..199, int)`
- `struct {int      a[10]; float f} st;`  
    `record((a × array(0..9, int) ) × (f × real))`
- `int *a;    pointer(int)`
- `int f(int a, float f)`  
    `(int × float) → int`



# 表达式类型的等价性

- 类型等价的定义可以分为结构等价和名字等价。
- 结构等价：两个类型表达式结构等价iff它们是相同的基本类型，或者它们是从同样的类型构造符作用于结构等价的类型。
- 名字等价：（如果允许类型命名）当且仅当它们相同。

# 结构等价确定

BOOL testeq(s,t)

{ if(s和t是相同的基本类型) return true;

if(s ==array(s1,s2) && t==array(t1,t2))

return testeq(s1,t1) && testeq(s2,t2);

if(s==s1 ×s2 && t==t1 ×t2)

return testeq(s1,t1) && testeq(s2,t2);

... ..

return false;

}

# 名字等价的例子

- 变量和类型申明(pascal):
  - TYPE link = ^cell; np = ^cell; neq=^cell;
  - VAR next:link, last:link, p:np, q:nqr; r:nqr;
- 所有的变量在结构等价的意义下面都相同。但是在名字等价的情况下:
  - next和last相同; q,r类型相同;
  - p和next类型相同。

# 类型强制

- 一般的程序设计语言中都规定了某些类型之间的转换关系：比如说整数量可以被当作实数量参与运算，并且不需要程序员显式说明。
- 不同类型的常数在计算机中有不同的表示。当一个值需要转换成为其它类型使用的时候，需要使用某些代码进行转换。
- 因此，编译程序要识别需要进行类型转换的地方，并相应地生成代码。
- 程序设计语言的设计者需要考虑什么情况下需要和可以进行转换。

# 类型强制转换的语义规则

- 首先，要确定强制转换的地方，首先需要确定参加运算的分量的类型，而这个分量本身就可能是一个表达式。

重写规则	语义规则
$E ::= \text{num}$	$E.\text{type} := \text{integer}$
$E ::= \text{num.num}$	$E.\text{type} = \text{float}$
$E ::= \text{id}$	$E.\text{type} = \text{gettype}(\text{id.entry})$
$E ::= E_1 \text{ op } E_2$	$\text{if } (E_1.\text{type} = \text{int} \ \&\& \ E_2.\text{type} = \text{int})$ $\quad E.\text{type} = \text{integer};$ $\text{if } (E_1.\text{type} = \text{int} \ \&\& \ E_2.\text{type} = \text{float})$ $\quad E.\text{type} = \text{float}; (\text{E1的结果需要强制转换})$ $\quad \dots \dots$

# 变量类型的确定

- 类型确定就是确定标识符代表对象的数据类型。在标识符的定义性出现的时候，要把标识符和类型相关联；在标识符的使用性出现时，需要获取得到标识符相关联的类型。
- 假设说明部分的文法如下：
  - $P ::= D; E \quad D ::= D; D \mid id:T$
  - $T ::= char \mid integer \mid ARRAY[num] \text{ OF } T \mid ^T$
- 处理说明部分的时候，需要做的语义处理是：对于每个说明 $id:T$ ，把 $id$ 和 $T$ 的类型表达式相关联。就是说，将 $id$ 的类型为 $T$ 的信息纪录到标识符表中。所以，每次识别到 $D ::= id:T$ 的情况时，需要调用 $addtype(id, T.type)$

# 变量类型的确定

- D有一个虚拟的综合属性。T有属性type,
- 以后的作业和考试中, 对于标识符有属性entry, 常量有属性lexval

$D ::= D; D$	
$D ::= id:T$	$\{addtype(id.entry, T.type)\}$
$T ::= char$	$\{T.type := char\}$
$T ::= integer$	$\{T.type = integer\}$
$T ::= ^T_1$	$\{T.type = pointer(T_1.type)\}$
$T ::= ARRAY[num] \text{ of } T$	$\{T.type = array(num.lexval, T.type)\}$

# 类型检查

- 类型检查可以分为动态和静态两种。
  - 动态检查在运行时刻完成。功效很低。但是如果语言允许动态确定类型，动态检查是必须的。
  - 静态检查在编译时刻完成。静态检查是高效的。
- 如果一个语言能够保证所有经过静态检查之后，程序没有运行时刻的类型错误，则称为强类型的。
- 类型检查的内容包括：
  - 表达式
  - 语句
  - 函数



# 表达式的类型检查

- 对于表达式的类型检查，主要的工作是检查参与运算的操作数是否可以进行相应操作。

重写规则	语义动作
$E ::= \text{literal}$	$\{E.type = \text{char}\}$
$E ::= \text{num}$	$\{E.type = \text{integer}\}$
$E ::= \text{id}$	$\{E.type = \text{gettype}(\text{id.entry})\}$
$E ::= E_1 \text{ mod } E_2$	$\{E.type = \text{if } (E_1.type = \text{integer} \ \&\& \ E_2.type == \text{integer}) \text{ then integer else type\_error}\}$
$E ::= E_1 [E_2]$	

# 语句的类型检查

- 语句的类型检查主要包括：赋值语句类型的相容性，控制表达式的结果类型检查。
- 先面的例子中，赋值语句要求左右两边类型相同；在C语言中，赋值类型相容性在表达式部分处理。

重写规则	语义动作
$S ::= id := E$	$\{S.type = \text{if } id.type == E.type$ $\text{then void else type\_error}\}$
$S ::= \text{if } E \text{ then } S1$	$\{S.type = \text{if } E.type = \text{boolean}$ $\text{then } S1.type \text{ else type\_error}\}$
$S ::= \text{while } E \text{ do } S1$	$\{S.type = \text{if } E.type = \text{boolean then}$ $S1.type \text{ else type\_error}\}$

# 函数的类型检查

- 简化了的函数调用语法为 $E ::= E(E)$ ;
- 类型检查所需要做的工作是：检查参数是否符合条件，并且确定函数结果的类型。
- $E ::= E1(E2)$
- $\{E.type = \text{if } E2.type=s \text{ and } E1.type=s \rightarrow t \text{ then } t \text{ else type\_error}\}$
- 当函数有多个参数的时候，需要检查多个参数的类型。
- 实际的翻译方案还需要考虑类型的相容性问题。

# 说明部分的翻译

- 对于说明部分的翻译，需要考虑的问题包括：
  - 标识符的类型纪录。
  - 标识符对应的数据对象的地址分配。
  - 标识符的作用域问题。

# 常量定义的翻译

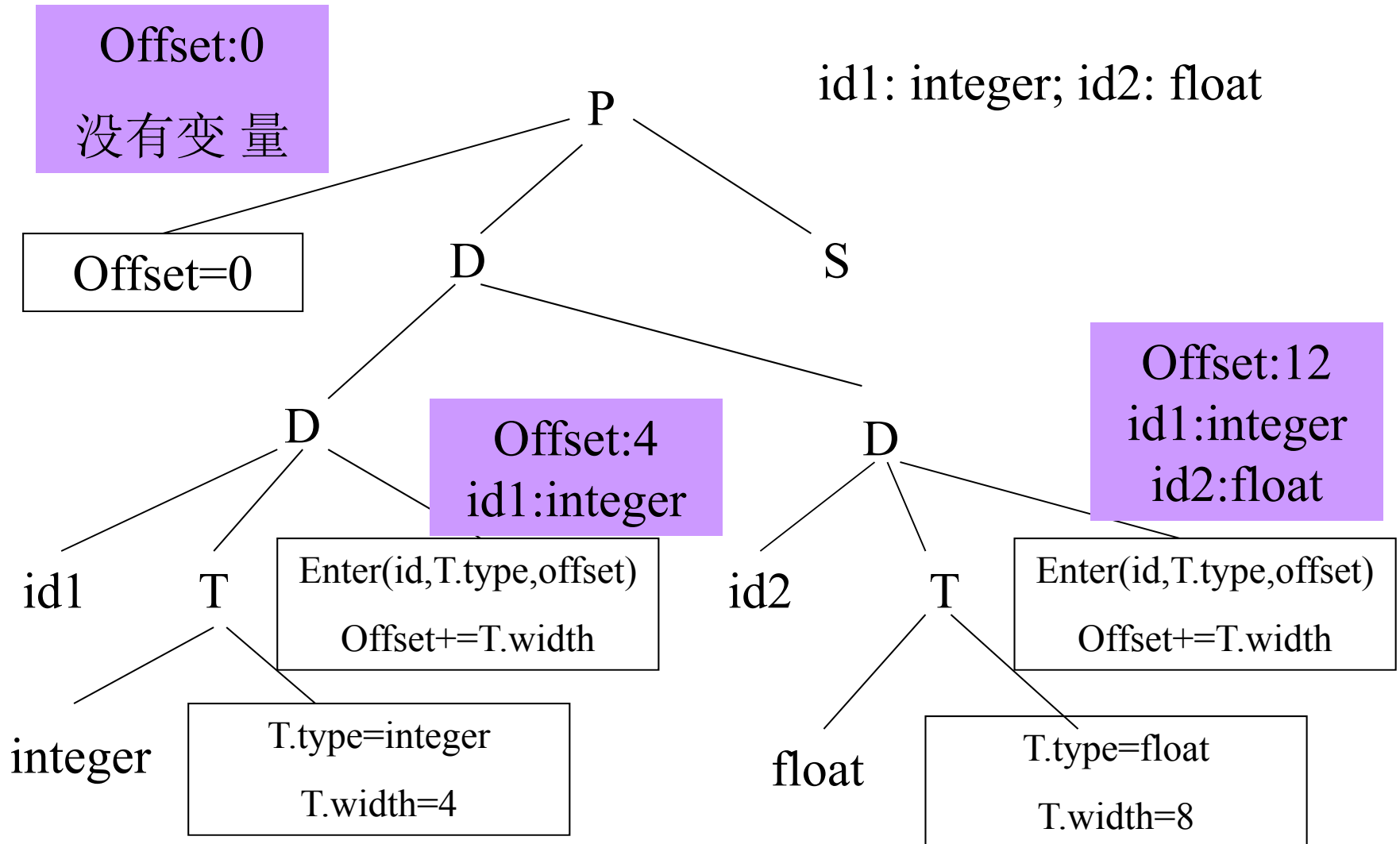
- 常量定义的翻译主要工作是将常量标识符的信息记录下来。
- Add(...)函数在全局的标识符表中记录相关信息

重写规则	语义动作
CDT::=CDT;CD	
CDT::=CD	
CD::=id=num	{num.ord=lookCT(num.lexval);id.ord=num.ord; id.type=integer; id.kind=CONST; add(id.entry,id.kind, id.type, id.ord)}

# 变量说明的翻译

- 变量说明的翻译工作包括：确定变量类型，分配地址空间。
- 地址空间的分配方式如下：有一个全局变量`offset`，纪录了当前可用的空间的开始地址。每次分配一个变量的空间，将`offset`增加相应的值。
- 有`static`类变量时，必须在全局地址空间中分配。
- 规则如下：
  - $P ::= \{\text{offset} = 0\} D; S \qquad D ::= D; D$
  - $D ::= \text{id} : T \quad \{\text{enter}(\text{id.name}, T.\text{type}, \text{offset}); \text{offset} += T.\text{width}\}$
  - $T ::= \text{integer} \quad \{T.\text{type} = \text{integer}, T.\text{width} = 8\}$
  - $T ::= \text{real} \quad \{T.\text{type} = \text{real}; T.\text{width} = 8\}$
  - $T ::= \text{ARRAY}[\text{num}] \text{ of } T1 \quad \{T.\text{type} = \text{array}(\text{num.lexval}, T1.\text{type}); T.\text{width} = \text{num.lexval} * T1.\text{width}\}$

# 变量说明的处理过程



# 函数说明的翻译

- 函数说明的翻译工作主要包括：
  - 为函数建立标识符表，存放在函数定义中局部说明的标识符。
- 主要方法：
  - 每个函数有一个标识符表；
  - 全部变量也有标识符表。
  - 每个函数都有自己的地址空间，相对地址从0开始。用栈offset的顶存放当前过程可用的初始地址。



# 函数定义的翻译方案（1）

重写规则	语义动作
$P ::= MGL$	$\{ \text{addwidth}(\text{top}(\text{tblptr}), \text{top}(\text{offset}));$ $\text{pop}(\text{tblptr}); \text{pop}(\text{offset}) \}$
$M ::= \text{空}$	$\{ \text{t} = \text{mktable}(\text{NIL});$ $\text{push}(\text{t}, \text{tblptr}); \text{push}(0, \text{offset}) \}$
$G ::= D1$	
$G ::= \text{空}$	
$D ::= D1; D2$	
$D ::= T \text{ id}$	$\{ \text{enter}(\text{top}(\text{tblptr}), \text{id.name}, T.\text{type}, \text{top}(\text{offset}));$ $\text{Top}(\text{offset}) = \text{top}(\text{offset}) + T.\text{width} \}$
$D ::= T \text{ id}[\text{num}]$	$\{ \text{enter}(\text{top}(\text{tblptr}), \text{id.name}, \text{array}(\text{num}.\text{lexval},$ $T.\text{type}), \text{top}(\text{offset})); \text{Top}(\text{offset}) =$ $\text{top}(\text{offset}) + T.\text{width} * \text{num}.\text{lexval} \}$

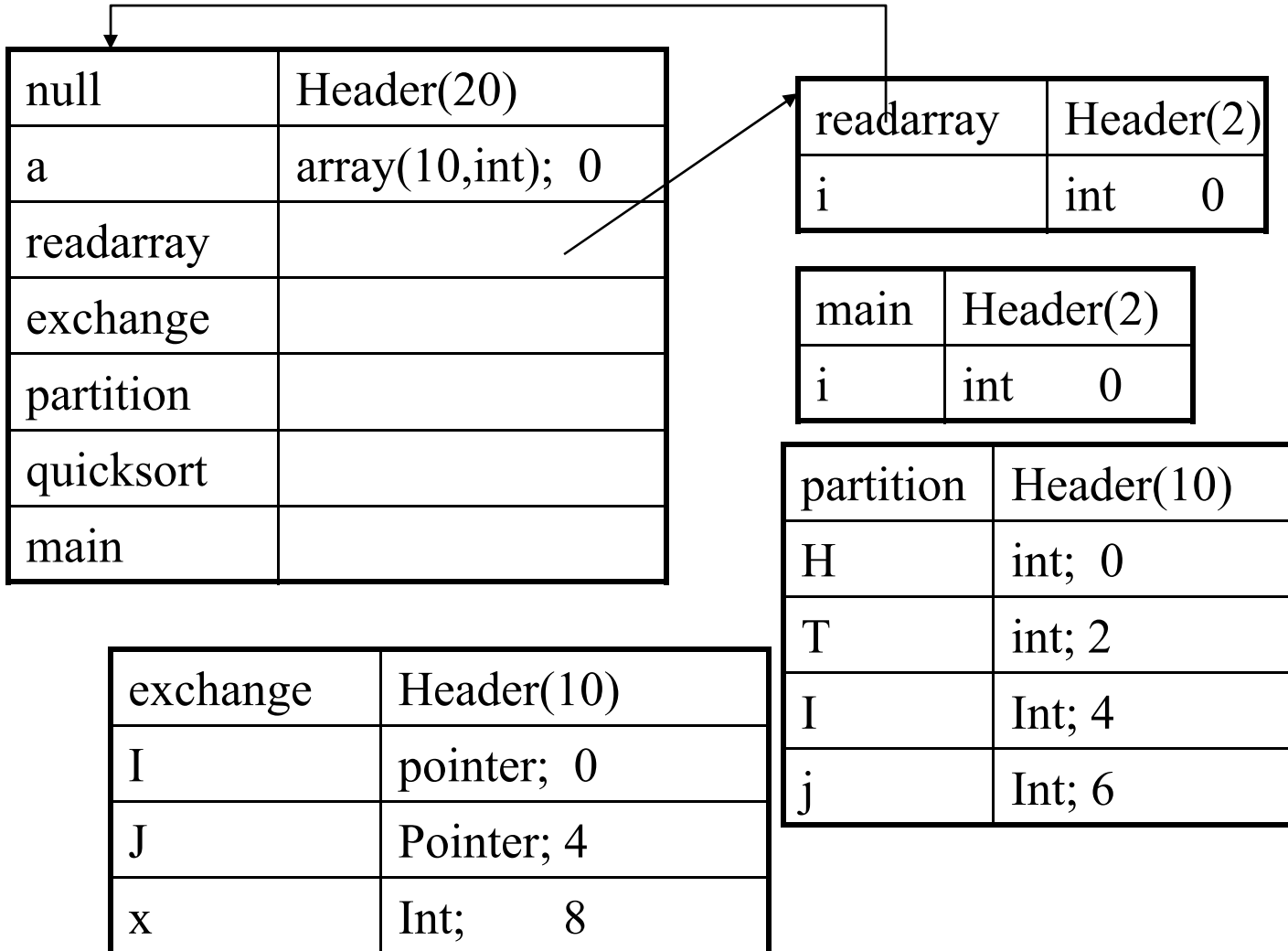
# 函数定义的翻译方案（2）

$F ::= N \text{ id}() \{D; S\}$	$\{t := \text{top}(\text{tblptr}); \text{addwidth}(t, \text{top}(\text{offset}));$ $\text{pop}(\text{tblptr}); \text{pop}(\text{offset});$ $\text{enterproc}(\text{top}(\text{tblptr}), \text{id.name}, t)\}$
$N ::= \text{空}$	$\{t := \text{mktable}(\text{top}(\text{tblptr})); \text{push}(t, \text{tblptr});$ $\text{put}(0, \text{offset})\}$

# 函数说明翻译方案的解释

- 在进行扫描之前，建立初始的tblptr栈和offset栈。即 $M ::= \text{空}$  对应的语义动作。
- 进入每个函数说明的时候，建立一个新的标识符表，存放函数的局部定义。即 $N ::= \text{空}$  对应的语义动作。
- 每个函数定义扫描结束的时候，消除空栈，并且将函数名加入到全局说明的标识符表中去。

# 过程说明符号表的例子



# 结构类型说明的翻译

- 在处理记录类型的说明的时候，首先需要知道各个域的名字，类型，和该域的偏移量。
- 如果考虑到记录类型的嵌套，那么可以和过程的说明类似处理。都建立符号表。

$T ::= \text{struct } L \{ D \}$	$\{ T.type := \text{record}(\text{top}(\text{tblptr}));$ $T.width := \text{top}(\text{offset});$ $\text{pop}(\text{tblptr}); \text{pop}(\text{offset}) \}$
$L ::= \text{空}$	$\{ t := \text{mktable}(\text{null});$ $\text{push}(t, \text{tblptr}); \text{push}(0, \text{offset}) \}$

# 结构类型说明翻译的例子

- `Struct {char name[10]; int number} R`
- 在扫描到`struct`之后，执行L的语义动作：创建一个空的标识符表。并压入栈。
- 然后，在扫描结构中的域定义的时候，可以使使得域名和类型都存放到前面创建的这个表里面。
- 在扫描完全部的结构说明的时候，指明其类型属性为`record`，并且各个域名/类型对存放在栈顶的标识符表中。

# 目标代码的生成

- 代码生成程序
  - 输入是源程序的中间表示形式，输出为和源程序等价的的目标代码。
- 目标代码可以有不同的形式：
  - 目标机器代码形式；
  - 汇编语言程序形式；
  - 虚拟机目标代码形式；

# 目标代码生成的有关问题（1）

- 目标机器语言的确定：确定目标代码的形式，目标机，和目标语言。也可以使用一些中间表示方式，比如三元式，四元式等。
- 语言结构目标代码的确定：确定语言中的各类语言结构和目标代码结构之间的对应关系。
- 运行时刻存储管理：程序运行的时刻，需要为变量分配存储空间。在编译时刻可以确定变量所需要的空间，和偏移地址。而实际的地址要等到运行时刻才可以确定。



# 目标代码生成的有关问题（2）

- 寄存器分配：合理利用寄存器可以使得程序运行效率比较高。
- 求值顺序的选择：通过改变表达式求值的顺序，可以提高效率。但是必须保证程序的正确性。
- 代码生成程序的设计：产生正确的代码，代码易于优化。

# 虚拟机

- 虚拟机的设立包含了一般计算机的特征，但是比较简单。方便学习编译程序的书写。
- 按照字节编址，4个字节为一个字。
- 有n个通用寄存器。
- 指令形式为            OP        源， 目标， 表示将源和目标地址中的数据计算后存放到目标地址中。

绝对地址	M	M	1
寄存器	R	R	0
变址	D(R)	D+contents(R)	2
间接寄存器	*R	contents(R)	1
间接变址	*D(R)	Con(D+con (R))	2
立即数	#C	常数C	1

# 虚拟机的指令集合

- NEG T       $-\text{con}(T) \Rightarrow T$
- SUB S T     $\text{con}(T) - \text{con}(S) \Rightarrow T$
- MPY S T     $\text{con}(T) * \text{con}(S) \Rightarrow T$
- DIV S T     $\text{con}(T) / \text{con}(S) \Rightarrow T$
- CMP S T    比较 $\text{con}(T)$ 和 $\text{con}(s)$
- Cjrelop T   按relop为true转向T
- ITOF S T   整型 $\text{con}(S)$ 转换为float后存放到T。
- GOTO T    转向T
- RETURN    返回
- CALL P, N            以N个参数调用子程序P

# 语句的翻译（赋值）

- 语法：  $V=e$
- 语义：计算 $e$ 的值之后把值赋给变量 $V$ 。
- 执行的过程：
  - 计算 $V$ 的变量地址；
  - 计算右部表达式 $e$ 的值；
  - 如果需要，对 $e$ 进行强制类型转换；
  - 把转换过的 $e$ 值赋予 $V$ 的地址；
- 显然，一个赋值语句（赋值表达式）对应的代码由对应于上述4个部分的代码组成。
- 对于每个表达式，它对应的属性应该包括：代码`code`，结果存放的地址`place`。

# 赋值语句的翻译方案

$S ::= id := E$	$P := \text{lookup}(id.name); // \text{获取} id \text{信息}$ $\text{if}(p \neq \text{NIL}) \text{ then}$ $S.code = E.code \parallel \text{gencode}(\text{MOV}, E.place, p \rightarrow place)$ $\text{else}$ $\text{error}$
$E ::= E1 + E2$	$E.place = \text{newtemp};$ $E.code = E1.code \parallel E2.code \parallel \text{gencode}(\text{MOV}, E1.place, E.place)$ $\parallel \text{gencode}(\text{ADD}, E2.place, E.place)$
$E ::= E1 * E2$	$E.place = \text{newtemp};$ $E.code = E1.code \parallel E2.code \parallel \text{gencode}(\text{MOV}, E1.place, E.place)$ $\parallel \text{gencode}(\text{MPY}, E2.place, E.place)$
$E ::= id$	$p := \text{lookup}(id.name);$ $\text{if}(p \neq \text{null}) \{ E.place = p \rightarrow place; E.code = \text{空} \}$

# 赋值语句中的类型强制

- 当赋值语句的左右两边类型不同却相容的时候，应该进行类型强制。

```
{p=lookup(id.name);  
if(p!=null) then  
{ S.code=E.code;  
  if(p->type ==int and E.type==int)  
    S.code=S.code||gencode(MOV, E.place,p->place)  
  else if (p->type=real and E.type=int)  
    {u=newtemp; S.code= S.code||gencode(ITOF,  
      E.place,u)||gencode(MOV,u,p->place)}  
  else if      ...      ...
```

# 变量的存取

- 当赋值语句的左部不是简单变量的时候，代码中计算左部地址的部分比较复杂。
- 主要需要讨论的是：
  - 下标变量的翻译方案
  - 域变量的翻译方案

# 下标变量的计算方法

- 对于一维数组  $T\ a[num]$ ，其第  $i$  (从0开始计算) 个元素的地址是  $base + i * sizeof(T)$
- 对于二维数组  $T\ a[num1][num2]$ ，元素  $a[i][j]$  的地址和数组的分配相关。假设，数组元素按行存放，那么  $a[i][j]$  的地址是  $base + (i * num2 + j) * sizeof(T)$ 。
- 对于多维数组  $T\ a[n1][n2] \dots [nm]$ ，其元素  $A[i1, i2, \dots, im]$  的地址是：
  - $base + (i1 * n2 * \dots * nm + i2 * n3 * \dots * nm + \dots + im) * sizeof(T)$ 。
- 赋值左部的语法修改成：  $V ::= Elist \mid id$ ，  
 $Elist ::= Elist, E \mid id[E]$ 。



# 下标变量的计算方法(续)

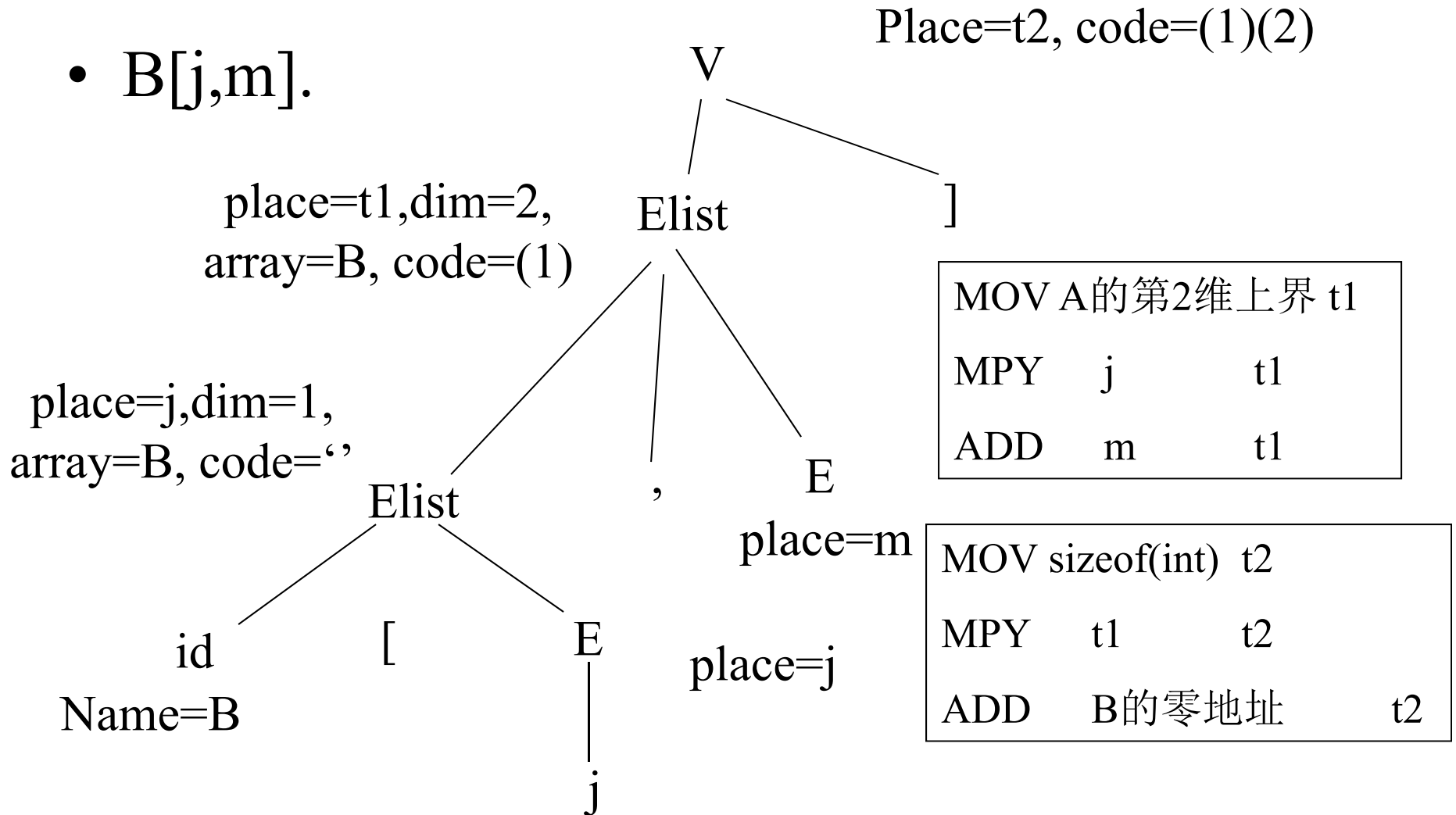
- $E::=V$   
    {if (V.atag==0)/\*不是数组\*/ .....  
      else{ E.place = newtemp;  
          E.code = V.code || gencode('MOV 0(V.place),E.place)}  
    }
- $V::=Elist$  {V.place=newtemp; V.tag = 1; V.code =  
Elist.code || 'MOV', sizeof(T)存放处, V.place || MPY  
Elist.place, V.place) || ADD Elist.array的零地址 V.place
- $V::=id$  {p=lookup(id.name);V.place=p;V.atag=0}
- 注：类型T和零地址存放处在Elist.array中可以得到。

# 下标变量的计算方法(续)

- $Elist ::= Elist1, E \{ t = newtmp; i := Elist1.dim + 1; Elist.code := Elist1.code \parallel E.code \parallel MOV, Elist1.array \text{第} i \text{维上界存放处}, t \parallel MPY Elist1.place, t \parallel ADD E.place, t; Elist.array = Elist1.array; Elist.place = i; Elist.dim = i; \}$
- $Elist ::= id[E \{ p = lookup(id.name); Elist.place = E.place; Elist.dim = 1; Elist.code = E.code; Elist.array = p; \}]$
- 属性说明：
  - $Elist.array$ 包含了这个数组的信息。
  - $Elist.dim$ 表示当前已经扫描到了第几维。
  - $Elist.place$ 最终存放了偏移量（以 $sizeof(T)$ 为单位），在其它时刻，扫描到第 $m$ 维时生成的代码使 $((i1 * num2 + i2) * num3 + \dots) * numm$ 存放在 $Elist.place$ 中。

# 下标变量计算的例子

- B[j,m].



# 下标变量的另外表示

- 如果采用文法
  - $V \rightarrow id \mid id[Elist] \quad Elist \rightarrow Elist \mid E \mid E$
  - 在扫描Elist的时候，id的类型信息必须作为继承属性传递。比较难以用尾动作翻译方案实现。
- 可以考虑这么做：
  - Elist有属性code和indexPlaces。code表示计算各个下标的值的代码。而indexPlaces记录了所有下标表达式的值的存放位置。
  - $V \rightarrow id[Elist] \{ V.code = Elist.code \parallel \text{按照} Elist.indexPlaces \text{ 计算下标偏移量的代码} \}$

# 域变量的翻译方案

- 在说明记录（结构）类型的时候，已经记录了各个域的偏移量，因此，域变量的地址的计算只需要查找各个域的偏移量就可以完成。
- $V ::= E.\text{domainname} \{ V.\text{place} = \text{newtmp};$   
     $\text{offset} = \text{LOOKUP\_st}(E.\text{type}, \text{domainname})$   
     $V.\text{code} = E.\text{code} \parallel \text{MOV } E.\text{place}, V.\text{place}$   
     $\parallel \text{ADD const}(\text{offset}) V.\text{place}$

# 选择语句的翻译

- 语法:  $S ::= \text{if}(E) S1 \text{ else } S2; \mid \text{if}(E) S1;$
- 语义: 如果E的值为true, 那么执行S1; 否则执行S2 (情况1) 或者不执行 (情况2)
- 代码模式: (E为真的时候跳转)
  - E的目标代码
  - 判别E值的代码
  - E值为true时转向E.true的代码
  - 转向E.false的代码
  - E.true: S1的目标代码
  - 转向E.next的代码
  - E.false: S2的代码
  - S.next: 后继语句的代码

# 代码的例子

- If ( $a < b$ )  $\text{max} = b$  else  $\text{max} = a$ ;
- MOV      a,      R0;   CMP R0      b;
- CJ<      L1
- GOTO    L2;
- L1:MOV b      R1;   MOV R1      max;
- GOTO    L3;
- L2: MOV a      R1;   MOV R1      max;
- L3:

# 语法制导的语义定义

- $S ::= \text{if } (E) \text{ } S1 \text{ then } S2$
- $\{E.true = \text{newlabel}();$   
 $E.false = \text{newlabel}(); S1.next = S.next;$   
 $S2.next = S.next;$  (翻译方案中斜体部分应该在if后面)  
 $S.code = E.code \parallel \text{CMP } E.place \text{ \#1 } \parallel \text{CJ=, \#E.true}$   
 $\parallel \text{GOTO } E.false \parallel \text{\#E.true:} \parallel S1.code \parallel \text{GOTO}$   
 $S1.next \parallel \text{\#E.false:} \parallel S2.code \parallel \text{\#S.next:}$   
 $\}$
- 用#var表示变量var的值。



# 布尔表达式的翻译（基本方式）

- $E ::= E1 \text{ OR } E2 \{E.place = newtmp; E.code = E1.code \parallel E2.code \parallel \text{MOV } E1.place, E.place \parallel \text{OR } E1.place \ E.place\}$
- $E ::= id1 \text{ relop } id2 \{E.place = newtemp; t := newtemp; E.code = \text{MOV } id1.place, t \parallel \text{CMP } t, id2.place \parallel \text{Cjrelop } t, *+8 \parallel \text{GOTO } *+12 \parallel \text{MOV } \#1 \ E.place \parallel \text{GOTO } *+8 \parallel \text{MOV } \#0 \ E.place\}$
- 这里假定每个指令长度为4，所以goto \*+8表示跳转到指令下面2条指令。

# 基本方式的代码例子

- $a < b$  OR  $c > d$  代码如下:
- (1) MOV a, t2; (2) CMP t2, b;
- (3) CJ< \*+8; (4) GOTO \*+12;
- (5) MOV #1 t1; (6) GOTO \*+8;
- (7) MOV #0 t1; (8) MOV c t4
- (9) CMP t4, d; (A) CJ> \*+8
- (B) GOTO \*+12; (C) MOV #1, t3
- (D) GOTO \*+8; (F) MOV #0 t3;
- (G) MOV t1 t5; (H) OR t3 t5;

# 布尔表达式的翻译（高效方式）

- 显然，对于一个布尔表达式来说，有时不需要计算全部表达式就可以得到值。比如E1 OR E2，如果E1的值为真，那么整个表达式的值也是真。
- 可以仅仅使用一个MOV #1 (#0) E.place语句，当表达式的值为真（假）的时候，转向这个语句。而对于用于控制的表达式，直接转向‘真’的分支。
- 注意：因为表达式可能有副作用，布尔表达式怎么翻译需要根据语言的语义来决定。比如
  - if (E1 OR f()) S1，使用基本方式的时候，f()一定会被执行，但是使用高效方式的时候，f()可能不执行。
  - 一般语言的语义定义都使用高效方式。

# 布尔表达式的语义规则

- $E ::= E1 \text{ OR } E2$ 
  - $E1.true = E.true; E1.false = newlabel;$
  - $E2.true = E.true; E2.false = E.false;$
  - $E.code = E1.code || gencode(E1.false, ':') || E2.code$
- $E ::= id1 \text{ relop } id2$ 
  - $t = newtemp;$
  - $E.code = MOV \ id1.place, t || CMP \ t, id2.place || Cjrelop$   
 $E.true || GOTO \ E.false$
- E的属性true和false分别表示如果E的值为真和假的时候，程序执行的下一步(继承属性)。
- 这是语义规则，所以E1的继承属性true在后面定义。

# 例子

- $a < b$  OR  $c > d$ 的代码:
- `MOV       a,       t1;       CMP               t1,       b;`
- `CJ< E.true;`
- `GOTO       L1;`
- `L1: MOV   c,       t2;`
- `CMP>       E.true; GOTO E.false`
- 对于if ( $a < b$  OR  $c > d$ ) S1 else S2, E.true和E.false分别是S1和S2的标号。
- 对于id =  $a < b$  OR  $c > d$ , E.true和E.false分别是MOV #1 id和MOV #0 id的标号。

# 回填技术

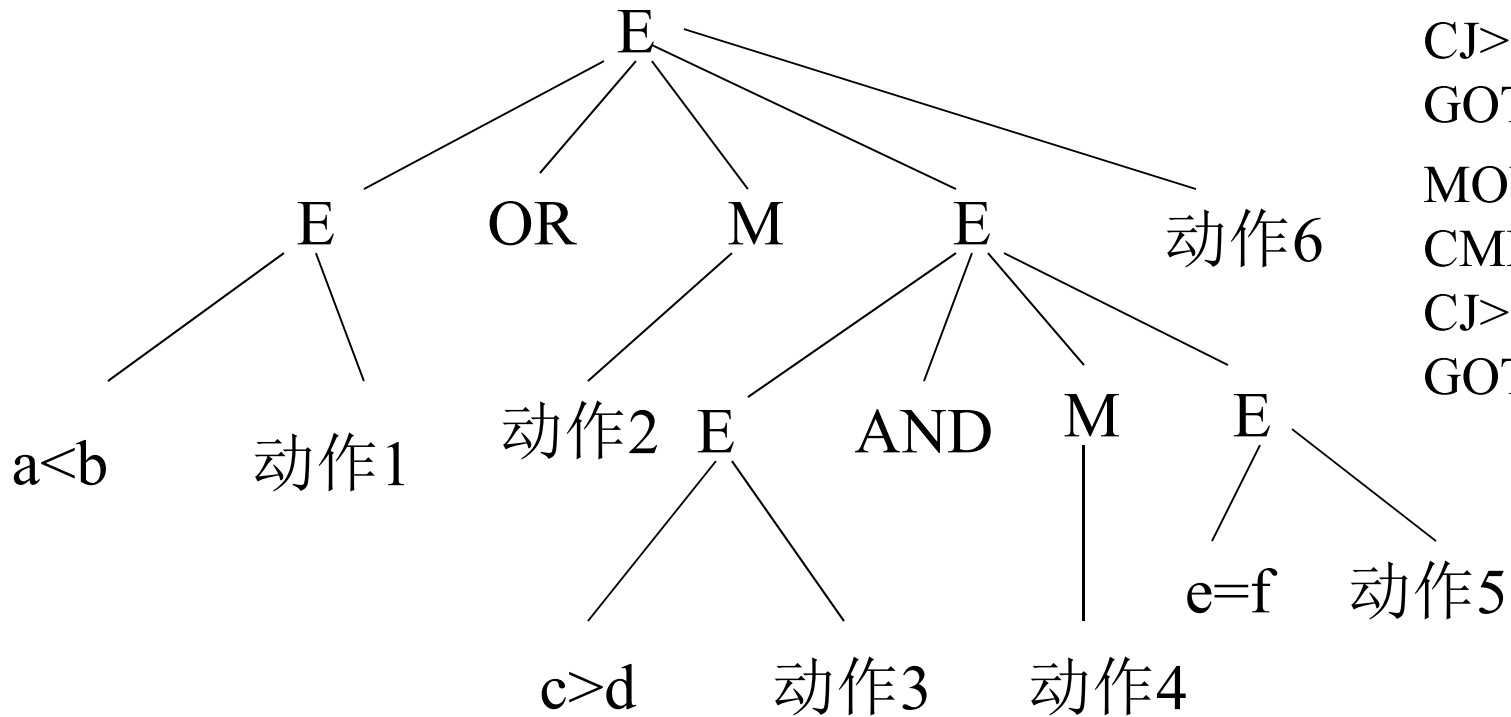
- `if (a<b) max=b; else max = a;`
- `MOV a R0; CMP R0 b; CJ< ?`
- 如果使用一趟扫描方式，显然，当要生成CJ< ?指令的时候，不知道该指令需要转向哪里。
- 可以使用回填技术解决：
  - 先生成一个指令坯，不填入转移目标；把这个指令的相关信息保存好，当可以得到转移目标的时候，在把目标填入这个位置。
- Backpatch函数：把第一个参数里面的所有地址对应的指令中的转移目标都设置为第二个参数。

# 条件语句的翻译方案（回填）

- $S ::= \text{if } (E) \text{ M1 S1 N ELSE M2 S2}$ 
  - $\{\text{backpatch}(E.\text{truelist}, M1.\text{pos}); \text{backpatch}(E.\text{falselist}, M2.\text{pos}); S.\text{nextlist} = \text{merge}(S1.\text{nextlist}, S2.\text{nextlist}); S.\text{nextlist} = \text{merge}(S.\text{nextlist}, N.\text{nextlist});\}$
- $M ::= \text{空} \{M.\text{pos} = \text{nextpos}\}$
- $N ::= \text{空} \{N.\text{nextlist} = \text{makelist}(\text{nextpos}); \text{emit}(\text{'GOTO'}, \text{'_'});\}$
- $E ::= E1 \text{ OR } M \text{ E2} \{\text{backpatch}(E1.\text{falselist}, M.\text{POS}), E.\text{truelist} = \text{merge}(E1.\text{truelist}, E2.\text{truelist}); E.\text{falselist} = E2.\text{falselist};\}$

# 回填的例子

- $a < b \text{ OR } c > d \text{ AND } e = f$



```
MOV a t1
CMP t1 b
CJ< _
GOTO _

MOV c t2
CMP t2 d
CJ> _
GOTO _

MOV e t3
CMP t3 f
CJ> _
GOTO _
```



# 情况语句(switch?)

语法:

CASE E of

case C1: S1;

case C2: S2;

....

Default        Sn

end

- 语义：根据E的值，执行相应的分支；如果E的值等于Ci，那么执行Si。

# 情况语句的执行步骤

- 步骤1： 计算表达式E的值
- 步骤2： 在情况表中顺序寻找和E的值相同的值。如果找不到，则和缺省值匹配。
- 步骤3： 执行和该值相关的语句，然后执行后继语句。

# 情况语句的目标代码设计 (方案一)

- 计算E的值并存放在t的代码
- 如 $t \neq C1$ 则转向L1的代码。
- 语句S1的代码。
- 转向后继语句的代码。
- L1: 如 $t \neq C2$ 则转向L2的代码。
- 语句S2的代码。
- 转向后继语句的代码。
- L2: ... ..
- ... ..
- $L_{n-1}$ : 语句 $S_n$ 的目标代码。
- Next: 后继语句的目标代码。

# 情况语句的目标代码设计 (方案二)

- 计算E的值并存于t的目标代码。
- 转向测试(test)的目标代码。
- L1: 语句S1的目标代码。
- 转向后继语句的目标代码。
- L2: 语句S2的目标代码。
- 转向后继语句的目标代码。
- L3: ... ..
- ... ..
- test: 如 $t=C_i$ 则转向 $S_i$ 的代码。
- Next: 后继语句的代码。

# 翻译方案的设计

- $S ::= \text{switch } E \{ \text{Clist. StateNextPos} = S.\text{NextPos} \}$
- $\text{CList DefaultS end}$
- $\{ \text{APPENDCODE}(\text{CaseTestCode}(\text{CList.PosList}, \text{CList.ValueList}, \text{DefaultS.Pos}));$
- $\}$
- $\text{CList1} ::=$ 
  - $C : \{ \text{pos} = \text{nextPos}(); \text{Clist1.PosList} += \text{pos};$   
 $\text{Clist1.ValueList} += C.\text{lexvalue}; \}$
  - $S \{ \text{APPENDCODE}(\text{“goto” Clist.StateNextPos}) \}$   
 $\{ \text{Clist.StateNextPos} = \text{CList1.StateNextPos}; \}$   $\text{Clist}$
- 上面的方案只是表示一个大概思路。

# 迭代语句的翻译

- 语法：while E do S
- 语义：当E的值为真的时候，重复执行语句S，直到E的值为假。
- 执行步骤：
  - 步骤1：计算E的值。
  - 步骤2：判断E的值，如果为true，则执行S，然后返回步骤1。
  - 步骤3：如果E的值为false，转向执行当语句的后续语句。

# 迭代语句的代码设计

**L:**    计算表达式E的目标代码。  
        判别E的值，并且当其值为false时，  
            转向后继语句(NEXT)的目标代码。  
        语句S的目标代码  
        转向标号L的目标代码。

**NEXT:** 后继语句的目标代码。

# 代码例子

- While  $n < 10$  do  $n = n + 1$ ;

```
L1:  MOV  n,    r0
      CMP           r0,    #10
      CJ<          L2
      GOTO L3
L2:  MOV  n,    r1
      ADD           #1,    r1
      MOV  r1,    n
      GOTO L1
```

L3:



# 迭代语句的语义规则

- $S ::= \text{while } E \text{ do } S$
- 规则如下：
  - $S.\text{begin} = \text{newlabel}();$
  - $E.\text{true} = \text{newlabel}(); E.\text{false} = S.\text{next};$
  - $S1.\text{next} = S.\text{begin};$
  - $S.\text{code} = S.\text{begin}: \| E.\text{code} \| E.\text{true}: \| S1.\text{code} \| \text{goto } S1.\text{next};$
- 请注意E的代码是按照高效的方案实现的，所以E的代码后面不需要判断转向的代码。

# 迭代语句的翻译方案（回填）

- $S ::= \text{while } M1 \text{ E do } M2 \text{ } S1$   
    {backpatch(S1.nextlist, M1.pos);  
    backpatch(E.truelist, M2.pos);  
    S.nextlist = E.falselist;  
    emit(goto, M1.pos)}
- $M ::= \text{空 } \{M.\text{pos} = \text{nextpos}\}$

# 复合语句的回填方案

- $S ::= \text{begin } L \text{ end} \{S.\text{nextlist} = L.\text{nextlist}\}$
- $L ::= L1; M \ S \ \{\text{backpatch}(L1.\text{nextlist}, M.\text{pos}); L.\text{nextlist} := S.\text{nextlist};\}$
- $L ::= S \ \{L.\text{nextlist} = S.\text{nextlist}\}$

# 过程调用

- 过程调用时，程序转向执行另外一段代码，执行完毕后，返回原来的执行点继续往下执行。正确的过程调用必须保证：
  - 程序的控制能够正确地转移到被调用的过程。
  - 被调用者能够正确地调用者哪里获取数据。
  - 被调用者能够把处理结果返回给调用者。
  - 同时，控制能够返回到调用的地方继续执行。
- 上述4条中，1，4是控制联系，2，3是数据联系。

# 数据联系

- 数据联系的内容包括：
  - 寄存器的内容在调用前后不被破坏。
  - 实在参数的正确传递。
  - 被调用过程局部变量的分配。
  - 非局部变量的正确引用。

# 活动记录

- 过程（函数）体的每次执行称为该过程（函数）的活动。
- 活动记录是为了管理过程的一次执行（活动）中所需要信息的连续存储区域。
- 所有的活动记录被放在一个栈里面。活动记录在过程被调用的开始建立并压栈，在过程执行结束之后出栈。

# 活动记录的结构

- 返回值：用来存放被调用过程返回给调用者的值。
- 实在参数：存放调用者传递给被调用者的实在参数。
- 访问链：存放用于引用其它活动记录中的非局部数据。
- 机器状态：保存临调用时的机器状态。
- 局部数据：过程的局部变量存放地点。
- 临时数据：存放执行时产生的中间结果。
- 活动记录的大小根据各个被调用过程决定。一般来说，在编译时刻就可以确定活动记录的大小。

返回值
实在参数
控制链
访问链
机器状态
局部数据
临时数据

# 调用者和被调用者的衔接

- 调用者：
  - 计算实在参数；把返回地址和自己的活动记录指针存放 to 被调用过程的活动记录。
- 被调用过程：
  - 保存寄存器和其它的机器状态。
  - 初始化局部数据，开始执行过程。
  - 把返回值存放在活动记录中。
  - 使用活动记录中的相应信息，恢复top\_sp和其它寄存器，返回控制。
- 调用者：
  - 把返回值复制到自己的活动记录中去。继续执行。



# 过程（函数）调用语句

- 语法：
  - call id(Elist)
- 语义：
  - 建立形式参数和实在参数的对应关系后，执行被调用过程的过程体。

# 过程语句的执行

- 为被调用者的活动记录分配存储区域
- 计算过程调用中各个实在参数的值，并把值存放在到相应的位置。
- 设置环境指针（访问链）等，使得被调用过程可以访问外围数据。
- 保护交用过程的机器状态，包括寄存器状态。
- 初始化被调用过程的局部数据。
- 控制转移到被调用过程的入口处。

# 目标代码设计

- 过程调用入口和出口部分的处理可以调用入口子程序和出口子程序来完成。
- 代码模式如下：
  - 计算实在参数 $p_1$ 的值的代码
  - param  $p_1$
  - ... ..
  - 计算实在参数 $p_n$ 的值的代码
  - param  $p_m$
  - call P m

# 翻译方案

- $S ::= \text{CALL id}(\text{Elist})$ 
  - $\{S.\text{code} = \text{Elist.code} \parallel \text{'CALL id.place, Elist.number}\}$
- $\text{Elist} ::= \text{Elist1}, E$ 
  - $\{\text{Elist.code} = \text{Elist1.code} \parallel E.\text{code} \parallel \text{PARAM } E.\text{place}; \text{Elist.number} = \text{Elist1.number} + 1;\}$
- $\text{Elist} ::= E$ 
  - $\{\text{Elist.code} = E.\text{code} \parallel \text{PARAM } E.\text{place}; \text{Elist.number} = 1\}$
- 缺陷:
  - $\text{call } P(\dots, f(x), \dots)$
  - 由于PARAM的功能时将数据拷贝到被调用者的活动区域。在上述的例子中， $f(x)$ 对应的PARAM指令会和P对应的指令混淆。

# 改进后的翻译方案

- 申请临时空间存放实在参数，用队列数据结构记录各个参数的存放位置。在计算完成之后一次性复制到新的活动记录。
- $S ::= \text{CALL id}(\text{Elist})$ 
  - {count = 0; while not emptyQ(Elist.q) do
  - { t=headQ(Elist.q); S.code = S.code || PARAM, t    Count:=count +1;}
  - S.code = S.code || CALL id.place, Count;
  - }
- $\text{Elist} ::= \text{Elist1}, \text{E} \{ \text{EnterQ}(\text{E.place}, \text{Elist.q}) \}$
- $\text{Elsit} ::= \text{E} \{ \text{Elist.q} = \text{CreateQ}(); \text{Enter}(\text{E.place}, \text{q}) \}$

# 参数的传递

- 值调用
  - 值参数的形式参数被当作过程的局部变量看待，存储区域在被调用者的活动记录中。
  - 调用者计算实在参数的值，并把该值送入相应位置。
- 引用调用
  - 如果参数是变量，传送给形式参数的实际上是参数的地址。
  - 如果参数是常量或者表达式，首先给这些值分配临时存储区域，然后把这个区域的地址传送给被调用者。
  - 在被调用的过程中，对于参数的使用通过间接的方式实现。比如`swap(x,y)`中，`x,y`为引用调用时，`x=1`的操作为向`x`中的值存放

# 非局部数据的存取

- 过程内部的语句可以使用到非局部的变量。在pascal语言中，这种情况比较复杂。

F()

```
{int    x,y
```

```
    f1()
```

```
    {int        i,j;    j=x;}
```

```
    f2()
```

```
    {
```

```
        call f1();
```

```
    }
```

```
    call f2();
```

```
}
```

F的活动记录
F2的活动记录
F1的活动记录

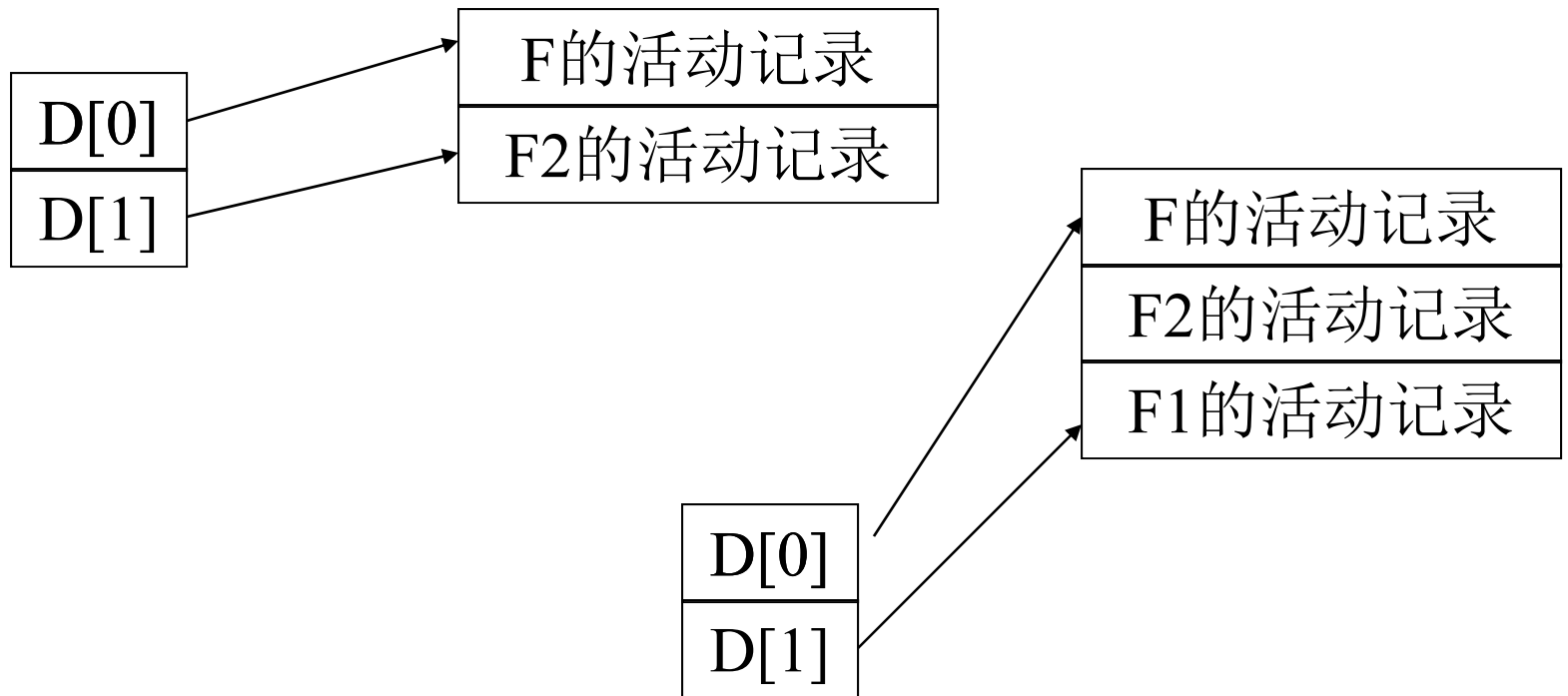
F1中调用x时，对应的是在F的活动记录中的x.需要根据访问链向上寻找。

# 显示表

- 显示表是活动记录的指针数组 $d$ ，第 $i$ 个元素表示嵌套层次为 $i$ 的过程对应的活动记录的位置。
- 显示表的维护：当为（正文静态）嵌套深度为 $i$ 的过程的活动记录的时候，把 $d[i]$ 保存在活动记录中，并把 $d[i]$ 设置为当前活动记录。退出某个活动记录的时候，恢复 $d[i]$ 为原来保存的值。
- 显示表的使用：当要使用嵌套层次为 $i$ 的变量时，由 $d[i]$ 得到该活动记录。根据该变量的偏移量，就可以得到该变量的地址。
- 显示表的内容是根据正文嵌套关系定的，和具体的调用关系没有多大关系。
- 在C语言中，由于不需要考虑函数的嵌套定义，所以非局部变量就是全局变量。不需要考虑显示表。



# 显示表的例子



# 源程序的中间表示

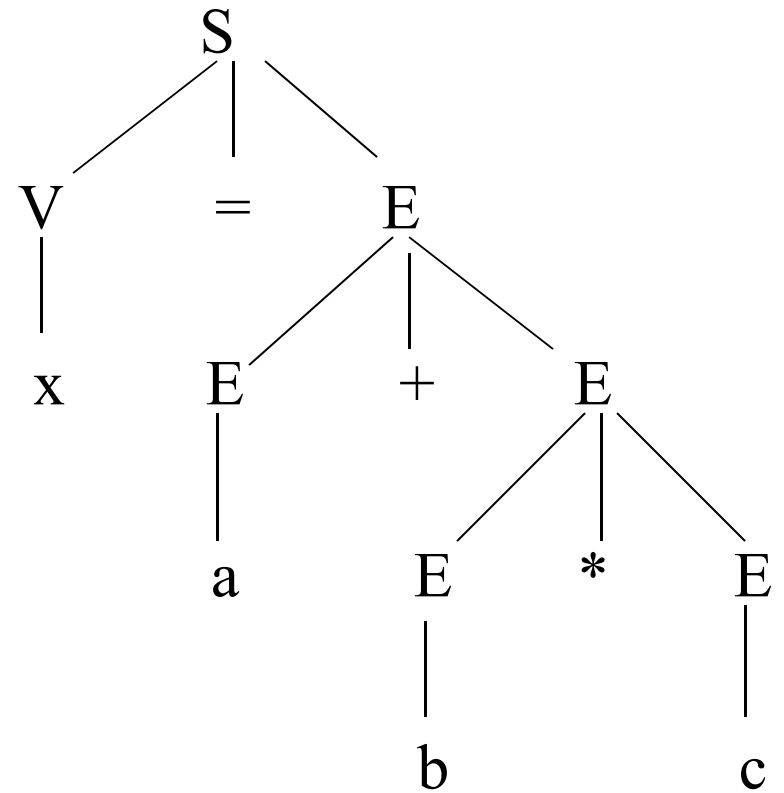
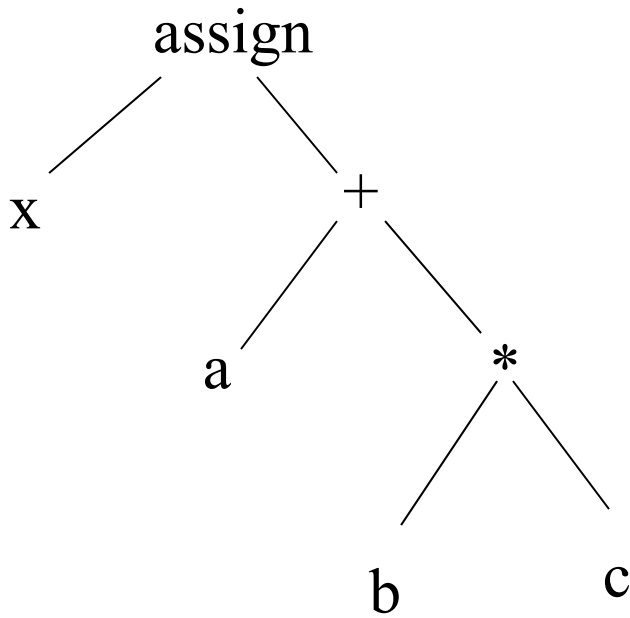
- 优点：
  - 和具体的机器特性无关，有利于重定目标。
  - 可以对中间表示进行优化，提高代码质量。
  - 可以分解程序的复杂性。

# 抽象语法树

- 通常的语法规则中，有些符号起到了解释作用。比如 $S ::= \text{if } (E) S1 \text{ else } S2$ 中，`if`, `else`等的作用就类似于标点符号。一般，关键字就类似于标点符号。
- 如果我们抛弃掉这些标点符号，就得到了抽象的语法规则：
  - 选择语句      表达式      语句    语句

# 抽象语法树的例子

- $x = a + b * c$



# 产生抽象语法树的语法制导定义

重写规则	语义规则
$S ::= id = E$	$S.nptr := mn(assign, ml(id, id.entry), E.nptr)$
$E ::= E1 + T$	$E.nptr = mn('+', E1.nptr, T.nptr)$
$E ::= T$	$E.nptr = T.nptr$
$T ::= T1 * F$	$T.nptr = mn('*', T1.nptr, F.nptr)$
$T ::= F$	$T.nptr = F.nptr$
$F ::= (E)$	$F.nptr = E.nptr$
$F ::= id$	$F.nptr = ml(id, id.entry)$
$F ::= num$	$F.nptr = ml(num, num.lexval)$

# 逆波兰表示方法

- 逆波兰又称为后缀表示方法，是表达式的一种表示形式。
- 在逆波兰表达式中，运算符写在分量后面。
- $a+b*c/(d+e)$ 的逆波兰表示为 $abc*de+ / +$
- 定义：设E是表达式，那么
  - 如果E是变量或者常量，E的逆波兰表示为E
  - $E1 \text{ OP } E2 \implies E1' E2' \text{ OP}$ ，其中 $E1', E2'$ 为E1, E2的逆波兰表示。
  - $(E) \implies E'$

# 逆波兰表示的生成（翻译）

- $E ::= E1 + T \quad \{\text{printf}(+)\}$
- $E ::= T$
- $T ::= T1 * F \quad \{\text{printf}(*)\}$
- $T ::= F$
- $F ::= (E)$
- $F ::= \text{id} \quad \{\text{printf}(\text{id.name});\}$

# 逆波兰表示方法的扩充

- 逆波兰表示可以被扩充到表达式以外的地方。
- 赋值语句:  $V=e \implies V'e' =$
- 赋值语句的例子:
  - $t=(a+b)*c/(d-e)$
  - $tab+c*de-/=$



# 条件语句的逆波兰表示

- if (e) St else Sf
- e' N1 GOF St' N2 GO Sf'
- N1和N2是逆波兰表示中符号的序号。
- GOF表示按假转，GO表示无条件转。
- 例子：if (a<b) max=b else max=a

1	2	3	4	5	6	7	8	9	10	11	12	13
a	b	<	11	GOF	max	b	=	14	GO	max	a	=

# 扩充后的逆波兰表示

- $\text{GOTO } N \implies N \text{ GOL}$
- GOL表示无条件转向标号
- 复合语句:  $S1;S2 \implies S1'S2'$
- 例子:
  - $\{i=1; \text{next: } f=f+1; \text{if } (i>f) \text{ } i=2 \text{ else goto next;}\}$
  - $\underline{i} \ 1 = \underline{\text{next:}} \ f \ f \ 1 \ + = \ i \ f > \underline{20} \ \underline{\text{GOF}} \ i \ 2 = \underline{22} \ \underline{\text{GO}} \underline{\text{next}} \ \underline{\text{GOL}}$

# 迭代语句的逆波兰表示

- $\text{while } (e) \text{ S}; \implies e' \text{ N1 GOF S' 1 GO}$
- 例子:
  - $\text{while } (i < 100) \{j = j + i\}$
  - $i \text{ 100 < 13 GOF j j i + = 1 GO}$

# 从逆波兰表示生成代码

- 算法框架：
  - 设立运算分量栈。
  - 从左到右扫描逆波兰表示。扫描到运算分量栈的时候，将该分量入栈。扫描到运算符的时候，根据该运算符的个数，对运算分量栈中相应个数的栈顶元素生成相应的目标代码；从分量栈中退出相应个数的分量；把存放运算结果的临时变量压栈。
  - 重复第二步，直到扫描完表达式中的所有符号。

# 从逆波兰表示生成代码（例子）

- If ( $a < b$ )  $\max = b$  else  $\max = a$
- $a \ b < 11 \ \text{GOF} \ \max \ b = 14 \ \text{GO} \ \max \ a :=$

MOV	a	R0	GOTO	_____
CMP	R	0 b	MOV	a R2
CJ<	*+8		MOV	R2 max
CJ>=	_____			
MOV	b	R1		
MOV	R1	max		

# 四元式序列

- 表示方法:
  - 运算符          运算分量    运算分量    结果
  - 运算分量可以是变量，常量，或者由编译程序引进的临时变量。
- 例子:
  - $a+b*c$
  - \*          b          c          t1
  - +          a          t1          t2

# 四元式

- 单目运算符  $op$ 

$$\begin{array}{ccccc} & & x & & \\ - & OP & X & - & T \end{array}$$
- 赋值语句:  $x = y;$ 

$$\begin{array}{ccccc} - & := & y & - & x \end{array}$$
- 间接赋值:  $*x = y;$ 

$$\begin{array}{ccccc} - & \&= & y & x & x \end{array}$$
- 转向指令:  $GOTO L$  (L为标号)
 
$$\begin{array}{ccccc} - & GOL & L & - & - \end{array}$$

# 四元式

- 按照关系relop比较按真转： (L为四元式序号)
  - relop    x        y        L
- 无条件转移到序号L(L为四元式序号)
  - GO        L        —        —
- 过程调用： p(x1,x2,...,xn)
  - PARAM        x1        —        —
  - PARAM        x2        —        —
  - ...
  - PARAM        xn        —        —
  - CALL        p        n        —



# 四元式

- 分程序复合语句的开始与结尾:
  - BLOCK
  - BLOCKEND
- 将数组A的第i个元素的地址存放到t:
  - $[] = A[i]$  t
- 将数组A的第i个元素存放到t:
  - $=[] A[i]$  t
- $A[i] = B[j]$ 
  - $[] = A[i]$  t1
  - $=[] B[j]$  t2
  - $\&= t2$  t1 t1

# 四元式序列的例子

- `max=a[1]; i=2;`
- `while(i<=n)`
- `{ if(a[i]>max) max = a[i]; i=i+1; }`

=[]	a	1	t1
=	t1		max
=	2		i
>	i	n	(12)
=[]	a	i	t2
<=	t2	max	(9)

=[]	a	i	t3
=	t3		max
+	i	1	i
=	t4		i
GO	(4)		

# 生成四元式的翻译方案

- 生成四元式的方案和前面的直接生成代码的方案是类似的。
- $S ::= id = E \{ p = \text{loopup}(id.name);$   
     $\text{if}(p \neq \text{NULL}) \text{genquad}('=', E.place, ' ', p \rightarrow place)$   
     $\text{else error}();$
- $E ::= E1 + E2$   
     $\{ E.place = \text{newtmp}; \text{genquad}('+', E1.place, E2.place, E.place) \}$
- $E ::= id \{ p = \text{lookup}(id.name); \text{if}(p \neq \text{null}) E.place = p \rightarrow place;$   
     $\text{else error}(); \}$

# 从四元式到代码

- 一般来讲，四元式的运算符都有对应的机器指令，或者对应的子程序，因此从四元式生成指令代码是容易的。
- 主要考虑的问题是运算分量和计算结果的存取问题。主要考虑：运算分量在内存中还是寄存器中，在寄存器中时以后是否可以被使用等。
- 例如：-  $x - y = z$ 对应的代码：
  - 如果 $x, y$ 的值分别在 $R_i$ 和 $R_j$ 中且以后不需要引用 $x$ ，那么指令为：sub  $R_j$   $R_i$ ，寄存器 $R_i$ 中包含 $z$ 的值。
  - 如果 $x$ 在 $R_i$ 中，而 $y$ 在内存单元：SUB  $y$   $R_i$
  - 如果 $x$ 的值以后还需要被使用，或者 $x, y$ 都在内存中，或者其中有常数，指令的生成还要作相应的变化。

# 基本块

- 定义：基本块是这样的一一个四元式序列，其控制流从第一个四元式进入，而从最后一个四元式离开，期间没有停止，也没有分支。
- 例如：

— *	a	z	t1
— *	a	t1	t2
— *	b	b	t3
— *	b	t3	t4
— -	t2	t4	t5

# 基本块划分算法

- 算法6.1
- 步骤1：确定入口四元式：
  - 第一个四元式是入口四元式。
  - 由条件转移或无条件转移四元式转移到的四元式是入口四元式；
  - 紧跟在条件转移或者无条件转移四元式后面的四元式是入口四元式。
- 步骤2：对于每个入口四元式，它所对应的基本块是从该四元式（含）开始，到下一个入口四元式（不含）或者四元式结束的全部四元式组成。

# 基本块划分的例子

=	1		i
=	1		f
=	0		a
>=	i	10	(12)
=	f		b
+	f	a	t1

=	t1		f
=	b		a
+	i	1	t2
=	t2		i
GO	(4)		
=	f		fib

- 入口四元式：1，4，5，12
- 基本块：1-3，4，5-11，12

# 代码生成算法

- 假定：每个四元式运算符都由相应的指令代码，并要求计算结果尽可能保留在寄存器中（为提高效率）。
- 算法输入：四元式；输出：目标代码
- 数据结构：
  - 寄存器描述符：记录寄存器当前存放了哪个变量的值。一个寄存器可以存放一个或者多个的值。
  - 地址描述符：记录每个名字的当前值的存放处所，可以是寄存器，内存地址，或者它们的集合（当值被赋值传输的时候）。



# 寄存器描述符和地址描述符的例子

- 四元式

– +	a	b	t1
– =	t1		x

- 代码:

– MOV	a	r0		
– ADD	b	r0	r0包含t1	t1在r0中
– MOV	r0	x	r0包含t1,x	x在r0和存储字中。

# 算法

- 对于每个四元式  $op \quad x \quad y \quad z$  完成下列动作：
  - 调用 `getreg()`，确定  $x \ op \ y$  的值的存放地址  $L$ ， $L$  通常是寄存器。
  - 查看地址描述符，确定  $x$  的当前的一个位置  $x'$ 。如果  $x$  的当前值既在内存，又在寄存器中，选择寄存器作为  $x'$ 。如果  $x'$  不是  $L$ ，生成指令  $MOV \quad x' \quad L$ 。
  - 生成指令：  $op \ y' \quad L$ ，其中  $y'$  是  $y$  的当前值所在（寄存器优先）。修改地址描述符，指明  $z$  在  $L$  中。如果  $L$  是寄存器，修改寄存器描述符号，指明  $L$  中包含了  $z$  的值。

# 算法（续）

- 如果x和/或y的值此后不再使用，并且在寄存器中，则修改寄存器描述符，只是这些寄存器中不再包含x和/或y的值。
- 处理单目运算的四元式时，和上面的方法类似。但是对于  $x \_ y$  四元式，如果此时x的值在某个寄存器中，那么不需要生成代码，只需要用寄存器描述符和地址描述符指明y的值也在该寄存器中即可。
- 当基本块处理完毕之后，如果有些值可能在其他基本块被使用，并且地址描述字表示这些值仅仅存放在寄存器中。那么我们需要生成将这些值存放到内存中的指令。

# 算法(getreg)

- 如果x的值在寄存器中，此寄存器不包含其他名字的值，并且op x y z之后，没有对x的引用，则将该寄存器作为L。修改地址描述符。
- 如果1失败，并且有空闲寄存器的话，回送一个空闲寄存器作为L
- 如果2失败，并且z在基本块中有下次引用，或者OP必须使用寄存器，那么寻找一个被占用的寄存器R，用MOV R M指令将R的值存入相应的内存单元，修改地址描述符。如果R中存放了多个值，需要生成多个MOV指令。
- 把z的单元作为L。

# 例子

- $X = (a - b) * (a - c) + (a - c)$
- 1) - a          b          t1                                  2) -          a          c          t2
- 3) \* t1          t2          t3                                  4) +          t3          t2          t4
- 5) = t4                                  x
- MOV a, R0; SUB b R0 (R0包含t1, t1在R0中)
- MOV a R1; SUB c R1 (R1包含t2, t2在R1中)
- MPY R1 R0 (R0包含t3, R1包含t2 t2在R1中, t3在R0中)
- ADD          R1          R0 (R0包含t4, t4在R0中)
- MOV          R0          x (R0包含t4, x x在R0和内存中, t4在R0中)

# 不同的寻址方式

- 对于变址寻址和间接寻址，考虑四元式：
  - $=[] \quad A \quad i \quad t$
  - $= \quad t \quad \_ \quad a$
- $i$ 所在的位置确定了生成哪些指令。
  - 当 $i$ 在寄存器 $R_i$ 中的时候，
    - $MOV \ A(R_i), \quad R$
    - $MOV \ R, \quad a$
  - 当 $i$ 在内存中的时候，需要在前面添加指令
    - $MOV \quad M_i, \quad R_i.$

## 寻址方式 (2)

- 考虑四元式:
  - $[] = B \quad i \quad t$
  - $\& = b \quad t \quad t$
- 假定有获取地址的指令  $\text{ADDR } A, R$ 
  - $\text{ADDR } B(Ri) \quad R$
  - $\text{MOV } b \quad *R.$
- 同样,  $i$  在内存的时候需要先生成指令
  - $\text{MOV } Mi \quad Ri.$

# 三元式

- 如果我们不明显给出四元式的结果部分，而是用四元式的编号来表示结果，那么我们可以得到三元式。形式如下：

运算符

运算分量

运算分量

- 和四元式的区别：

- 用序号表示结果：ADD      (3)      t
- 赋值运算是双目运算符：=    x      y
- 按假转中有两个运算参数：GOF    (14)    (4)



# 三元式例子

- `max=a[1]; i=2;`
- `while(i<=n)`
- `{ if(a[i]>max) max = a[i]; i=i+1; }`

(1)	=[]	A	1
(2)	=	(1)	max
(3)	=	2	i
(4)	<=	i	n
(5)	GOF	(14)	(4)
(6)	=[]	A	i
(7)	>	(6)	max

(8)	GOF	(11)	(7)
(9)	=[]	A	i
(10)	=	(9)	max
(11)	+	i	1
(12)	=	(11)	i
(13)	GO	(4)	

# 第七章 运行环境

- 在得到目标程序之后，程序的运行还需要一些运行时刻支持程序包的支持。
- 本章内容包括：
  - 运行时刻存储分配以及支持。
  - 符号表的管理。
  - 运行时刻支持系统。

# 名字到存储字的结合

- 程序设计语言是基于冯.诺伊曼结构的。变量仿效的是存储字，而赋值语句模拟的是计算机存取，运算操作。
- 在目标程序运行的时刻，必须要对源程序中的每一个量都分配相应的存储字。
- 分配是隐式进行的。在编译时刻就决定了这些变量应该如何分配，在运行时刻决定究竟分配到哪里。

# 环境和状态

- 环境是名字到存储字的映射，就是说：在程序的运行中的某个时刻，某个名字的变量究竟存放在什么内存的什么地方。
- 在不同的时刻，变量的存放位置不变。
- 一般在进入分程序和退出分程序的时刻，环境会改变。
- 状态是存储字到值的映射函数。
- 一般执行赋值语句的时候，状态会改变。但是环境不变。

# 例子

- `main()`
- `{ int x,y,t; t=x; x=y; y=t; }`
- 当进入`main()`函数的时候，`x,y,t`的存储字在活动记录中分配。
- 环境为：  $\{(x, mx), (y, my), (t, mt)\}$ ;
- 假设开始的时刻`x,y,t`的值为5,1,0，那么相应的状态为  $\{(mx, 5), (my, 1), (mt, 0)\}$ 。
- 在执行了三个赋值语句之后，状态变成  $\{(mx, 1), (my, 5), (mt, 5)\}$

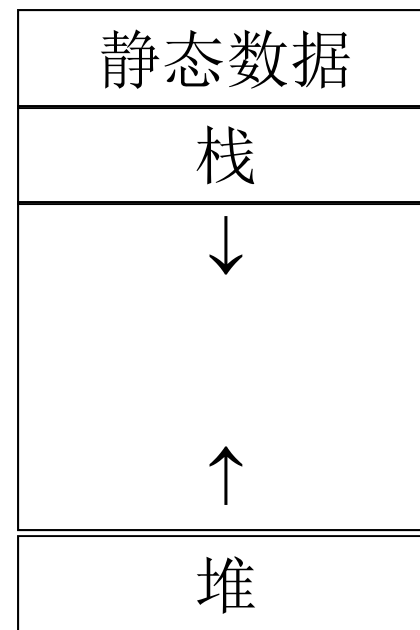
# 作用域和生存周期的问题

- 作用域是根据静态的源程序文本确定的。但是程序的运行是动态的。每个函数的调用的层次关系和正文的嵌套是不一致的。我们需要一个方法来确定在特定的运行时刻，某个名字对应的值存放在什么地方。
- 每个值都有生存周期。
  - 全局变量的值在整个运行期间有效。
  - 局部变量在函数退出之后就没有意义了。
  - 由new, 或者malloc确定的存储字中的值的意义直到相应的delete和free的语句执行过后才消失。

# 程序运行时刻的存储区域

- 分为代码段，和数据段。
- 数据段中分为静态数据，栈和堆。
- 堆和栈的增长方向相反。

代码
----



# 存储分配策略

- 静态分配策略
- 动态分配策略
  - 栈式分配策略
  - 堆式分配策略



# 静态存储分配

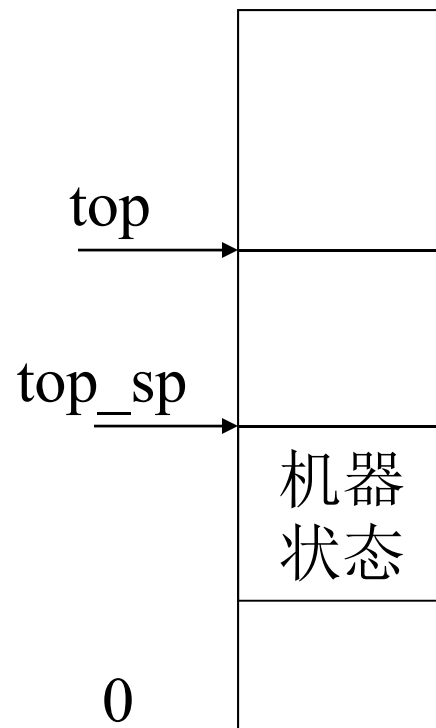
- 对于某些变量，在编译时刻就可以由编译程序为它们分配存储字。在运行时刻，这些变量和存储字的结合不变。
- 全局变量和静态变量。
- 不能处理的情况：
  - 在编译时刻不能确定大小的变量。
  - 要支持递归过程实现。
  - 动态建立的数据结构。

# 栈式存储分配

- 在内存中开放一个栈区，当过程的一次活动开始的时候，把活动记录入栈。过程活动结束的时候，活动记录出栈。
- 过程的局部变量在活动记录入栈的时候和存储字结合，活动记录出栈的时候结合消失。
- 栈式存储的方式可以分成活动记录大小可以确定和活动记录大小不可确定两种。

# 栈式存储分配（记录大小确定）

- 引入两个指针（一般使用特定的寄存器存放）：
  - **top**: 指向运行栈的栈顶。
  - **top\_sp**: 指向活动记录中机器状态域的末端。
- 在过程 $p$ 执行之前，把 $top$ 加上 $l$ ，在过程 $p$ 执行结束后，把 $top$ 减去 $l$ 。 $Top\_sp$ 的值可以相应确定。
- 对于过程中的局部名字 $x$ ，假设它的偏移量为 $dx$ ，那么 $x$ 的地址可以表示为 $-dx(R_{top})$ 。也可以写成 $dx'(R_{top\_sp})$ 。

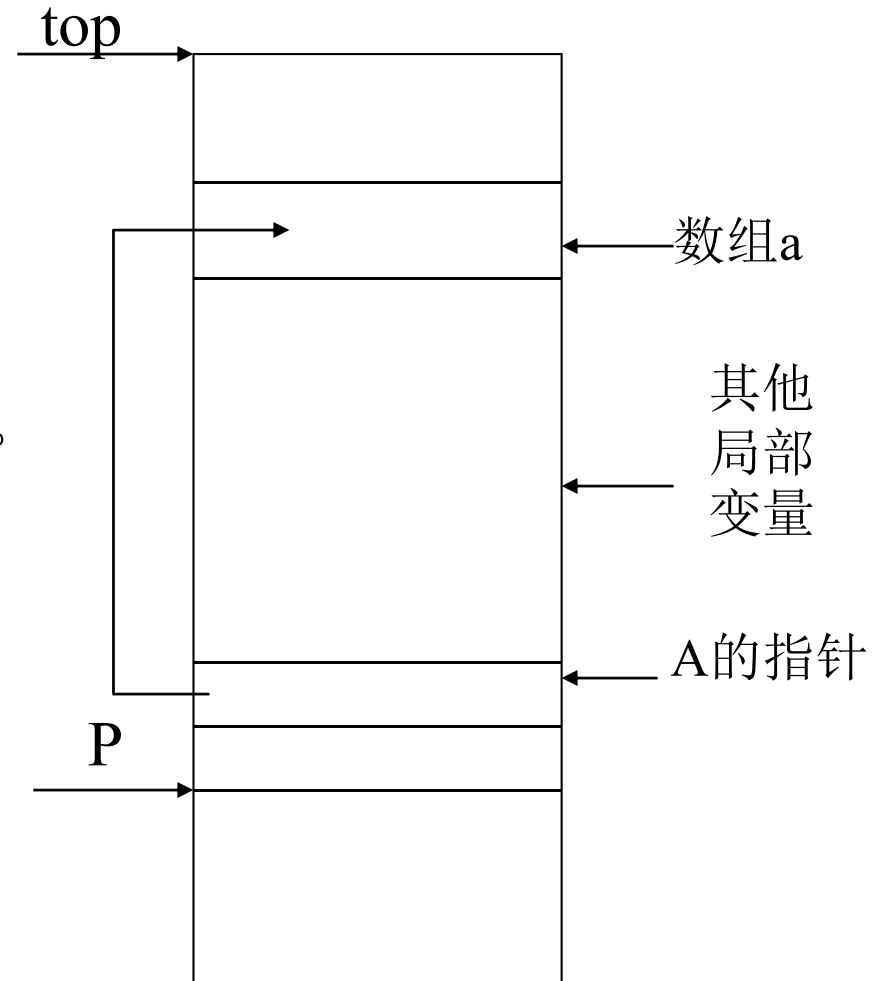


# 出/入口子程序

- 使用一个寄存器sp来存放栈顶指针。
- 入口子程序：
  - ADD #caller.recordsize SP
  - MOV #here+8, \*SP//保存控制返回地址
  - GOTO callee.code\_area
- 出口子程序：
  - GOTO 0(sp);//callee执行。
  - SUB #caller.recordsize SP//退栈，由caller执行。

# 可变长度变量的存放

- 在安排活动记录的时候，首先安排的是在编译时刻就可以确定的域，比如：控制链，访问链，机器状态域等。这样可以使得安排可变长度的变量的时候可以不影响到这些域的位置。
- 可变长数组被安排在活动记录的顶部。并且在活动记录的局部数据区中另外存放可变长度数据的指针（长度可以预先确定）。
- 调用其他函数的时候，SP的增加值需要动态确定。



# 堆式存储分配

- 有些数据对象是不适合安排在静态区或者栈区里面的。它们的生存期是难以确定的。比如：由new生成的对象。这些对象生存期在new语句执行后开始，到delete语句执行后结束。以下任何一种情况都必须使用堆式分配：
  - 数据对象随机创建和消亡。
  - 过程活动结束后，数据对象依旧需要保留。
  - 被调用过程活动的生存期比调用过程更加长。
- 堆式分配中,只有当程序在运行中,正式要求撤销数据对象的时候,该对象才被释放.

# 堆式存储分配

- 堆管理的基本功能:
  - 分配空间
  - 释放空间
- 指针(指引)的管理:
  - 当程序申请了一个空间之后，一般得到一个指针。该指针同时也是表示该空间的句柄。
  - 如果已经释放了该空间，那么这个指针就没有意义了，称为悬空指针。

# 指针空间

- 对于每个申请的空间，都有长度限制。如果超出这个限制，会产生不可预测的错误。
- C语言中的堆空间的分配一般如下：





# 垃圾回收系统

- 有些程序设计语言（一般是面向对象的程序设计语言）没有显式的空间回收机制（函数或者语句），而是使用垃圾回收系统。便于编写程序，但是效率较低下。
- 当没有指针指向某段空间的时候，系统就回收该空间。
- 数据对象的指针的拷贝时，需要记录指向该对象的指针数量。

# 4 运行时刻支持系统

- 运行时刻子程序是为了支持目标程序的运行而开发的一系列子程序。这些子程序的全体被称为运行时刻支持程序包。
- 面向用户的运行子程序
  - 用户可以显式调用这些子程序。
- 对用户隐匿的运行子程序：
  - 活动记录的存储分配，参数传递机制，机器状态的保护。

# 编译原理讲义

## (第八章 代码优化)

南京大学计算机系

赵建华

# 优化的概念

- 编译时刻为改进目标程序的质量而进行的各项工作。
  - 空间效率
  - 时间效率
- 空间效率和时间效率有时是一对矛盾，有时不能兼顾。
- 优化的要求：
  - 必须是等价变换
  - 为优化的努力必须是值得的。
  - 有时优化后的代码的效率反而会下降。

# 优化的分类

- 机器相关性：
  - 机器相关优化：寄存器优化，多处理器优化，特殊指令优化，无用指令消除等。
  - 无关的优化：
- 优化范围：
  - 局部优化：当个基本块范围内的优化，合并常量优化，消除公共子表达式，削减计算强度和删除无用代码。
  - 全局优化：主要是基于循环的优化：循环不变式优化，归纳变量删除，计算强度削减。
  - 优化语言级：
- 优化语言级：针对中间代码，针对机器语言。

# 代码优化程序的结构



- 控制流分析的主要目的是分析出程序的循环结构。循环结构中的代码的效率是整个程序的效率的关键。
- 数据流分析进行数据流信息的收集，主要是变量的值的获得和使用情况的数据流信息。
  - 到达-定值分析；活跃变量；可用表达式；
- 代码变换：根据上面的分析，对内部中间代码进行等价变换。

# 基本块和流图

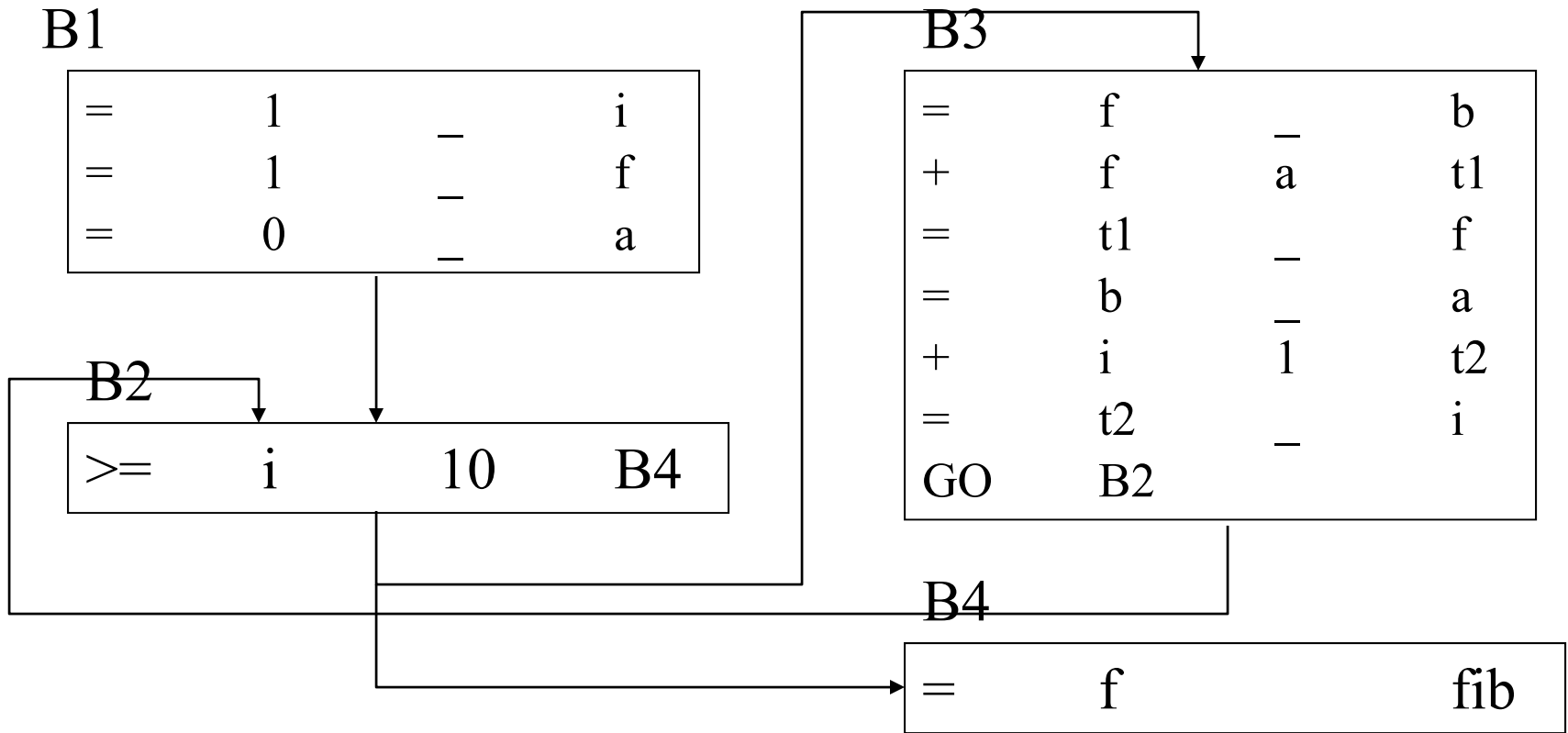
- 基本块中，控制流是由第一个四元式进入，到达最后一个四元式离开。
- 流图：把一个程序的中间表示中所有的基本块作为节点集合。有边从节点 $n$ 到节点 $n'$ 当且仅当控制流可能从 $n$ 的最后一个四元式到达 $n'$ 的第一个四元式。
- 首节点：对应的基本块的第一个四元式是程序的第一个四元式。

# 流图的构造

- 以所有的基本块为节点集合。
- 有B1到B2的边（B2是B1的后继）当且仅当：
  - B1的最后一个四元式有条件或无条件地转移到B2的第一个四元式。
  - B2是紧紧跟随在B1后面的四元式，且B1的最后四元式不是无条件转向语句。



# 流图的例子



- 在转移语句中，目标标号转变称为基本块的编号，可以避免因为四元式的变动而引起的麻烦。

# 基本块的优化

- 合并常量计算
- 消除公共子表达式
- 削减计算强度
- 删除无用代码

# 合并常量计算

- 例子:  $1 = 2 * 3.14 * r$

— \*        2        3.14   t1

— \*        t1        r        t2

— =        t2                    1

- $2 * 3.1415926$  的值在编译时刻就可以确定。

— \*        6.28    r        t2

— =        t2                    1

# 消除公共子表达式

- $+ \quad b \quad c \quad a \quad - \quad a \quad d \quad b$
- $+ \quad b \quad c \quad c \quad - \quad a \quad d \quad d$
- 显然，第2和4个四元式计算的是同一个值，所以第四个四元式可以修改称为  $= \quad b \quad - \quad d$ 。
- 对于第1和3个四元式，虽然都是计算 $b+c$ ，但是他们的值其实是不同的，所以不能完成处理。
- 公共表达式：如果某个表达式先前已经计算，且从上次计算到现在，E中的变量的值没有改变。那么E的这次出现称为公共子表达式。
- 利用先前的计算结果，可以避免对公共子表达式的重复计算。

# 例子

- $x + y * t - a * (x + y * t) / (y * t)$

- $* \quad y \quad t \quad t1 \quad + \quad x \quad t1 \quad t2$

- $* \quad y \quad t \quad t3 \quad + \quad x \quad t3 \quad t4$

- $* \quad a \quad t4 \quad t5 \quad * \quad y \quad t \quad t6$

- $/ \quad t5 \quad t1 \quad t7 \quad - \quad t2 \quad t7 \quad t8$

- 消除公共子表达式之后:

- $* \quad y \quad t \quad t1 \quad + \quad x \quad t1 \quad t2$

- $* \quad a \quad t2 \quad t5 \quad / \quad t5 \quad t1 \quad t7$

- $- \quad t2 \quad t7 \quad t8$

# 削減計算強度

- 实现同样的运算可以有多种方式。用计算较快的运算代替较慢的运算。
- $X^2$  变成  $x*x$ 。
- $2*x$ 或 $2.0*x$  变成  $x+x$
- $x/2$  变成  $x*0.5$
- $a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$  变成  
–  $((\dots(a_n x + a_{n-1})x + a_{n-2})\dots)x + a_1)x + a_0$

# 删除无用代码

- 如果四元式  $op \quad x \quad y \quad z$  之后， $z$  的值再也没有被使用到，那么这个四元式是无用的。
- 无用的四元式往往意味着程序的错误，一般不会出现再正确的程序里面。
- 多数无用四元式是由优化引起的。
- $= \quad t1 \quad \quad \quad t3$ ，如果我们尽量用  $t1$  替代  $t3$ ，可能使  $t3$  不被使用，从而这个四元式称为无用的四元式。

# 等价变换的分类

- 保结构等价变换

- 删除公共子表达式和删除无用代码，重新命名临时变量和交换独立四元式的顺序等。

– +	x	y	t	变成	+	x	y	u
– +	a	b	t1	+	x	y	t2	变成
– +	x	y	t2	+	a	b	t1	

- 代数等价变换利用了代数恒等性质，

- 削减计算强度。  $2x = x + x$ ,  $B \text{ and true} = B$ .
  - 需要考虑双目运算符的可交换特性。

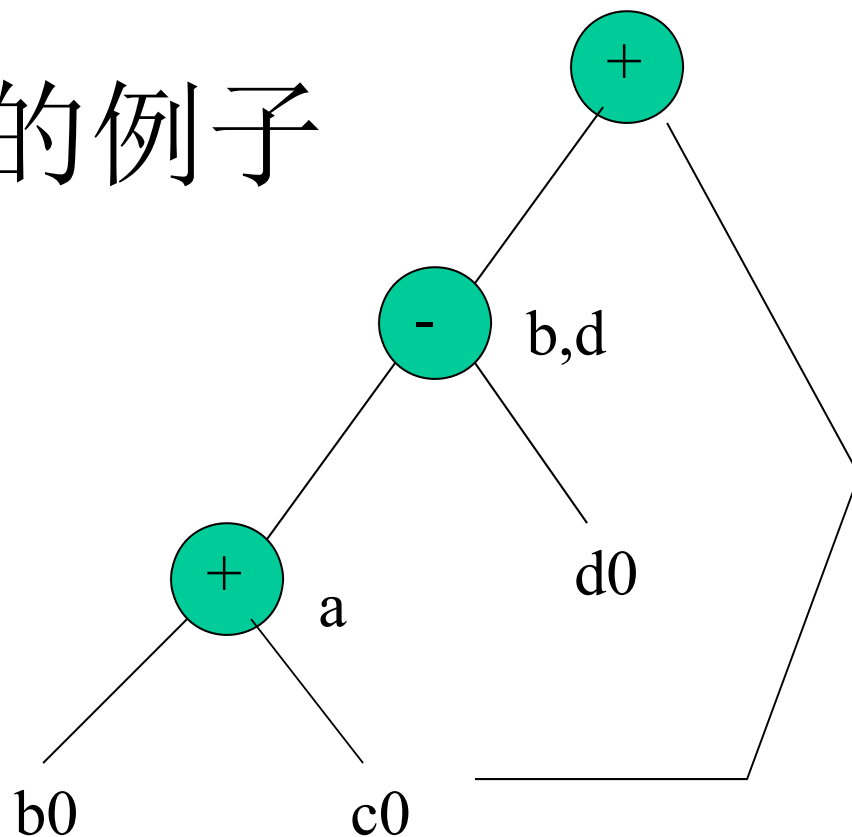


# 基本块优化的实现

- 基本块内部优化的实现主要工具为DAG图。
- 用DAG图表示各个值的计算/依赖关系。
- 图中的标记：
  - 叶子节点的标记为标识符（变量名）或常数作为唯一的标记。叶子节点是标识符时，用下标0表示它时初值。
  - 内部节点用运算符号作为标记，表示计算的值。每个节点的值都可以用关于变量初始值的表达式表示。
  - 各节点可能附加有一个或者多个标识符。同一个节点的标识符表示相同的值。

# DAG图的例子

- + b c a
- - a d b
- + b c c
- - a d d



# 四元式的分类

- 0型:  $= \quad x \quad \_ \quad y$
- 1型:  $op \quad x \quad \_ \quad y(\text{单目运算})$
- 2型:  $op \quad x \quad y \quad z$   
 $relop \quad x \quad y \quad z(z\text{是序号})$

# 基本块DAG图构造算法

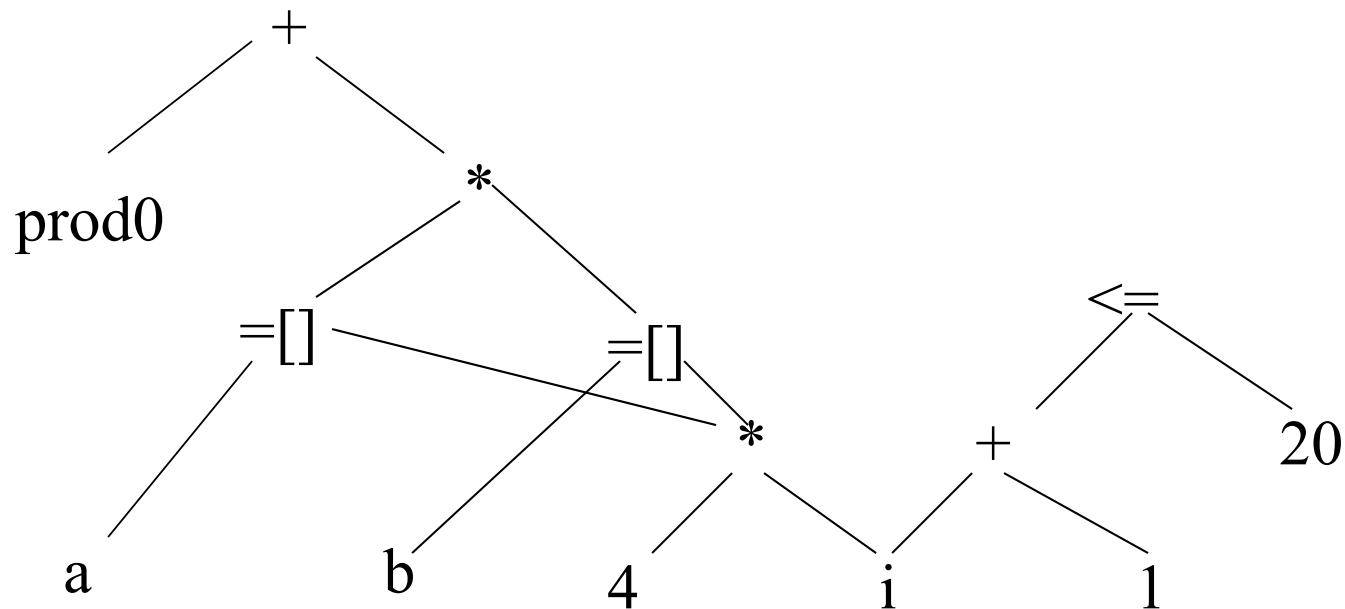
- 输入：一个基本块                      输出：相应DAG图
- 算法说明：
  - 通过逐个扫描四元式来逐渐建立DAG图。
  - 函数 $\text{node}(x)$ 表示和标识符 $x$ 相应的最近建立的节点。他代表扫描到当前的四元式的时候，标识符 $x$ 的值对应的节点。
- 步骤1：初始化：无任何节点， $\text{node}$ 对任何标识符无定义。
- 步骤2：依次对基本块中的每个四元式 $\text{op } x \ y \ z$ 执行下面的步骤。
  - 如果 $\text{node}(x)$ 没有定义，建立叶子节点，标记为 $x$ ，让 $\text{node}(x)$ 等于这个节点。如果 $\text{node}(y)$ 没有定义，为 $y$ 建立节点。
  - 如果四元式为0型， $n=\text{node}(x)$ ;
  - 如果四元式为1型，寻找标记为 $\text{op}$ 且子节点为 $\text{node}(x)$ 的节点，如果找不到，建立这样的节点。

# 基本块DAG图构造算法（续）

- 对于2型四元式，查看是否存在标记为 $op$ 的节点，且其左右子节点分别为 $node(x)$ 和 $node(y)$ 。如果找不到，建立这样的节点。
- 步骤3：如果 $z$ 为标识符，从 $node(z)$ 中删除标识符 $z$ ，并把 $z$ 加入到步骤4所找到或者建立的节点 $n$ 的标识符表中，并设置 $node(z)$ 为 $n$ 。
- 说明：
  - 处理2型四元式的时候，如果 $op$ 是可交换的运算符，可以其左右节点可以互换。

# 生成DAG图的例子

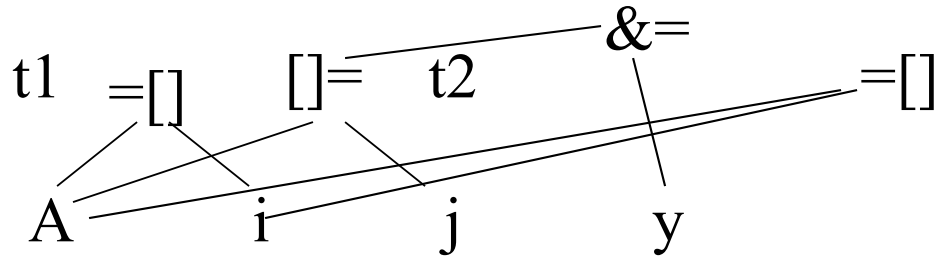
- \* 4 i t1 =[] a t1 t2
- \* 4 i t3 =[] b t3 t4
- \* t2 t4 t5 + prod t5 t6
- = t6 prod + i 1 t7
- = t7 i <= i 20 (3)



# DAG图的应用

- 公共子表达式：构造中，寻找是否有标记为 $op$ 且子节点为 $node(x)$ ,  $node(y)$ 的节点时，自然完成了公共子表达式的寻找。
- 在基本块中，其值被引用的标识符：构造了叶子节点的标识符。
- 结果能够在基本块外被引用的四元式 $op\ x\ y\ z$ 。设它对应的节点为 $n$ ，如果DAG图构造结束的时候， $n$ 的标志符表不为空。

# []=和&=运算符的处理



- 对数组的赋值需要特别的处理，这是因为数组的下标是变量。对于数组元素的赋值可能改变数组中任何一个元素的值。
- $[] =$  A      i      t1       $[] =$       A      j      t2
- $\& =$  y      t2      t2       $[] =$       A      i      t3
- A[i]并不是公共子表达式。
- 在处理对数组A的元素的赋值四元式的时候，应该注销所有以  $[] =$  为标记，A为左节点的节点。从而不可能在此节点的标识符表中再附上其他的标识符。
- 处理对指针所指空间的赋值的时候，同样要注销相应的节点。如果不能确定指针指向的范围，那么，需要注销所有的节点。



# 从DAG图到四元式序列

- 在DAG图中，有些运算已经进行了合并。
- 如果不考虑[]=和&=算符，可以依照DAG图中的拓扑排序得到的次序进行。但是，有了[]=和&=算符之后，计算的次序必须修正。
- 实际上，我们可以按照各个节点生成的顺序来从DAG图生成四元式序列。

# 从DAG重建四元式序列算法

- 按照DAG图中的各个节点生成的次序，对于每个节点作如下处理：
  - 若是叶子节点，且附加标识符表为空，不生成四元式。
  - 若是叶子节点，标记为x，附加标识符为z，生成= x z。
  - 若是内部节点，附加标识符为z，根据其标记op和子节点数目，生成下列4种形式的四元式。
    - Op不是=[]或者[]=，也不是relop，有两个子节点，生成
$$\text{op} \quad x \quad y \quad z$$
    - 如果是=[]或者[]=，生成
$$\text{op} \quad x \quad y \quad z。$$
    - 如果是relop，生成
$$\text{relop} \quad x \quad y \quad z，z\text{是基本块序号。}$$
    - 只有一个子节点，生成
$$\text{op} \quad x \quad \_ \quad z。$$

# 从DAG重建四元式序列算法(续)

- 若是内部节点，且无附加标识符，则添加一个局部于基本块的临时性附加标识符，按照上一情况生成。
- 如果节点的标识符重包含多个附加标识符 $z_1, z_2, \dots, z_k$ 时：
  - 若是叶子节点，标记为 $z$ ，生成一系列四元式

– =             $z$              $z_1$

– =             $z$              $z_2$

– ...            ...            ...

– =             $z$              $z_n$

- 不是叶子节点，生成四元式序列：

– =             $z$              $z_2$

– ...            ...            ...

– =             $z$              $z_n$

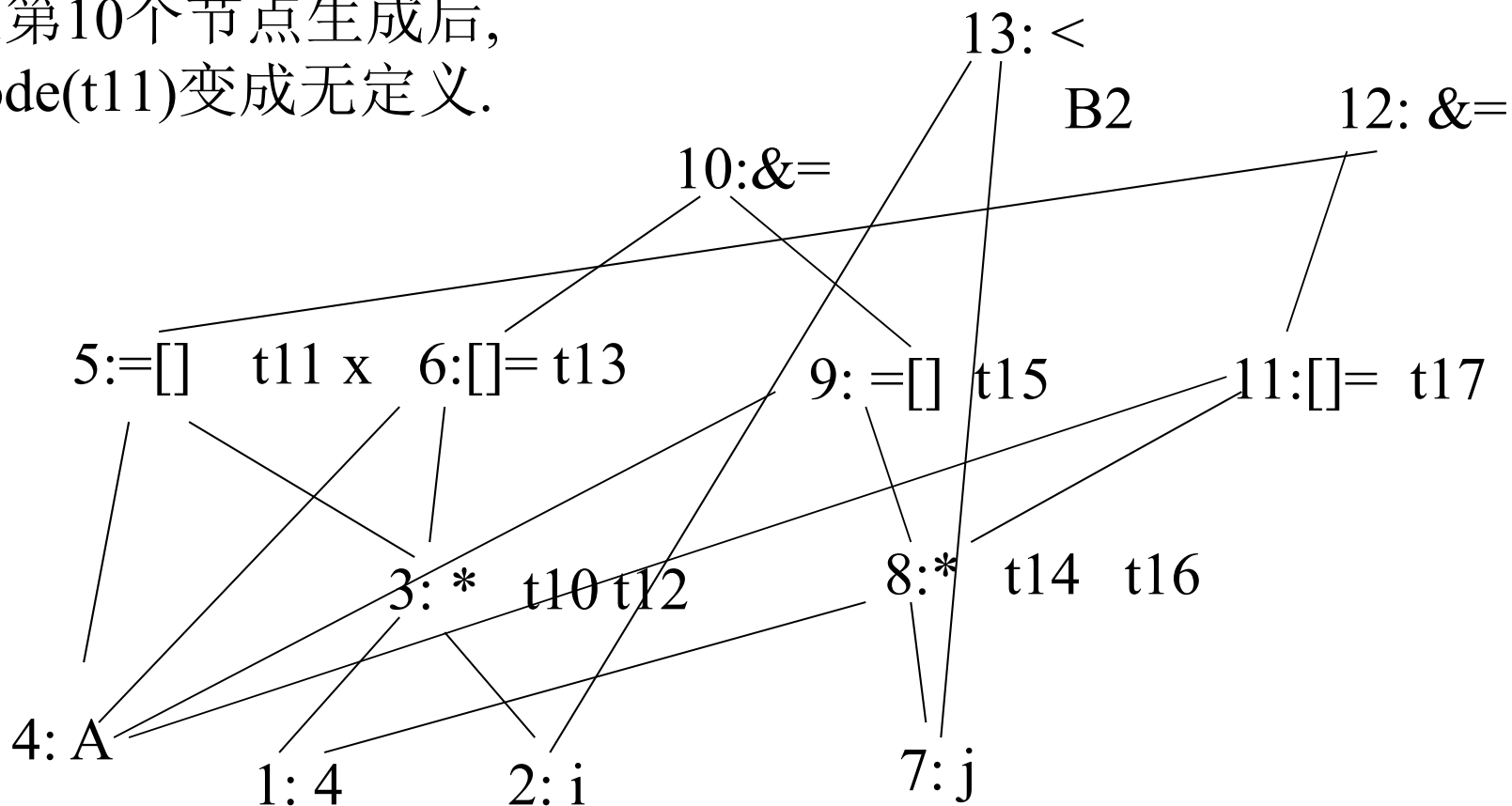
# 使用DAG图进行优化的例子 (四元式序列)

- 四元式序列片断:

- $* \quad 4 \quad i \quad t10 \quad =[] \quad A \quad t10 \quad t11$
- $= \quad t11 \quad x \quad * \quad 4 \quad i \quad t12$
- $[] = A \quad t12 \quad t13 \quad * \quad 4 \quad j \quad t14$
- $=[] A \quad t14 \quad t15 \quad \&= \quad t15 \quad t13 \quad t13$
- $* \quad 4 \quad j \quad t16 \quad [] = A \quad t16 \quad t17$
- $\&= x \quad t17 \quad t17 \quad < \quad i \quad j \quad B2$

# 使用DAG图进行优化的例子 (DAG图)

- 在第10个节点生成后,  
node(t11)变成无定义.



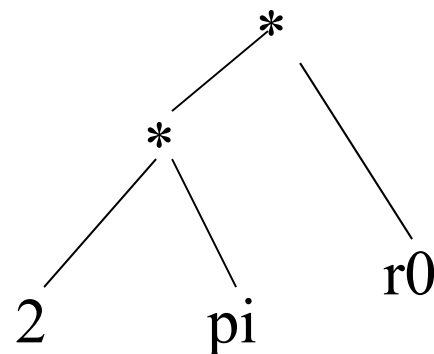
# 从DAG图到四元式序列

– *4	i	t10	(3)	
– =[]	A	t10	t11	(5)
– :=	t11		x	(5)
– []=	A	t10	t13	(6)
– *	4	j	t14	(8)
– =[]	A	t14	t15	(9)
– &=	t15	t13	t13	(10)
– []=	A	t14	t17	(11)
– &=	x	t17	t17	(12)
– <	i	j	B2	(13)

# DAG的其他应用

- 合并常量计算:

— *	2	pi	t1
— *	t1	r	t2
— =	t2	1	



- 无用代码的删除:

- 对于 = t10 t12, 如果t12不需要使用, 那么, 这个四元式不需要生成。

# 与循环有关的优化

- 循环不变表达式外提。
- 归纳变量删除。
- 计算强度削减



# 循环不变式外提

- 有些表达式位于循环之内，但是该表达式的值不随着循环的重复执行而改变，该表达式被称为循环的不变表达式。
- 如果按照前面讲的代码生成方案，每一次循环都讲计算一次。
- 如果把这个表达式提取到循环外面，该计算就只被执行一次。从而可以获得更加好的效率。

# 循环不变式的例子

- 计算半径为 $r$ 的从10度到360度的扇形的面积：
  - `for(n=1; n<36; n++)`
  - `{S:=10/360*pi*r*r*n; printf(“Area is %f”, S); }`
- 显然，表达式 $10/360 \cdot \pi \cdot r \cdot r$ 中的各个量在循环过程中不改变。可以修改程序如下：
  - `C= 10/360*pi*r*r*n;`
  - `for(n=1; n<36; n++)`
  - `{S:=C*n; printf(“Area is %f”, S); }`
- 修改后的程序中， $C$ 的值只需要被计算一次，而原来的程序需要计算36次。

# 四元式的循环不变式

- (1)= 1 n (2)> n 36 (21)
- (3)GOF (4) (4)/ *10* *360* *tl*
- *(5)\* tl pi t2* *(6)\* t2 r t3*
- *(7)\* t3 r t4* (8)\* t4 n t5
- (9)= t5 S ... ... ...
- (18)+ n 1 t9 (19)= t9 n
- (20)GO (4) (21)
- 其中，四元式4,5,6,7是循环不变四元式。

# 循环不变四元式的相对性

- 对于多重嵌套的循环，循环不变四元式是相对于某个循环而言的。可能对于更加外层的循环，它就不是循环不变式。

- 例子：

```
for(i = 1; i<10; i++)
```

```
    for(n=1; n<360/(5*i); n++)
```

```
        {S:=(5*i)/360*pi*r*r*n;...}
```

- $5*i$ 和 $(5*i)/360*\pi*r*r$ 对于n的循环（内层循环）是不变表达式，但是对于外层循环，它们不是循环不变表达式。

# 循环不变表达式优化需要解决的问题

- 如何识别循环中的不变表达式？
- 把循环表达式外提到什么地方？
- 什么条件下，不变表达式可以外提？

# 归纳变量的删除（例子）

- 例子：

```
Prod=0; i = 1;
```

```
for(i = 1; i<= 20; i++)
```

```
    prod += prod+A[i]*B[i];
```

- i作为计数器。每次重复，i的值增加1，而A[i], B[i]对应的地址t1, t3增加4。
- 我们可以删除i，而使用t1或者t3进行循环结束条件的测试。

# 归纳变量的删除

- 在循环中，如果变量 $i$ 的值随着循环的每次重复都固定地增加或者减少某个常量，则称 $i$ 为循环的归纳变量。
- 如果在一个循环中有多个归纳变量，归纳变量的个数往往可以减少，甚至减少到1个。减少归纳变量的优化称为归纳变量的删除。

# 归纳变量的删除（四元式例子）

=	0		prod
=	1		i

*	4	i	t1
=[]	a	t1	t2
*	4	i	t3
=[]	b	t3	t4
*	t2	t4	t5
+	prod	t5	t6
=	t6		prod
+	i	1	t7
=	t7		i
<=	i	20	B2

=	0		prod
=	0		t1

+	4	t1	t1
+	4	t3	t3
=[]	a	t1	t2
=[]	b	t3	t4
*	t2	t4	t5
+	prod	t5	t6
=	t6		prod
<=	t1	80	B2



# 归纳变量的删除

- 归纳变量的删除一方面可以删除变量，减少四元式，另外，删除归纳变量同时也削减了计算强度。
- 为了进行归纳变量删除优化，必要的是找出归纳变量。

# 计算强度削减

- 在删除归纳变量的过程中,已经将一些乘法运算转换为加法运算。
- 还有一类经常可以被应用的是对于下标变量地址的计算。

# 计算强度削减（下标变量）

- 对于数组T  $a[n1][n2]...[nm]$ ，其下标变量  $a[i1][i2][i3]...[im]$  的地址计算如下：
  - $base + d$ ；其中  $base$  为  $a[0][0]...[0]$  的地址。
  - $d = (((i1 * n2 + i2) * n3 + i3) ... ) * nm + im) * \text{sizeof}(T)$ ;
- 当满足某些情况的时候，地址的计算可以使用加法来代替乘法。

# 下标变量计算强度的削减（例子）

- $\text{for}(v1=v10; v1<v1f; v1++)$   
     $\text{for}(v2=v20; v2<v2f; v2++)$   
         $\{ \dots A[i1][i2][i3] \dots \}$
- $i1, i2, i3$  都可以表示成为:  
     $Ck0 + Ck1 * V1 + Ck2 * V2 (k=1,2,3);$
- $A[i1][i2][i3]$  的地址为  $\text{base} + d; d = (i1 * n2 * n3 + i2 * n3 + i3);$
- 将  $i1, i2, i3$  的表达式代入  $d$  的表达式, 可以得到  
     $d = C0' + C1' * V1 + C2' * V2.$

# 下标变量计算强度的削减（例子）

- 显然，在上面的例子中，每次内循环d的值增加C2'；每次外循环, d的值增加C1'（但是V2被重置）。
- 显然我们可以这样计算A[i1][i2][i3]的地址：
  - 在循环开始的时候，设置初值 $d1 = (\text{base} + C0') + C1' * V10$ ;
  - 在进入外层循环后，进入内存循环前，设置 $d2 = d1 + C2' * V20$
  - 在内存循环，使用d2作为地址获取A[i1][i2][i3]的值。
  - 内存循环体每次运行结束之前，将d2的值增加C2'。
  - 每次外层循环体运行结束之前，将d1的值增加C1'。
- 显然，对于A[i1][i2][i3]的地址计算变成了加法运算。

# 下标变量计算强度的削减结果

```
D1 = base+C0+C1'*V10;  
for(v1=v10; v1<v1f; v1++)  
{  
    D2 = D1+C2'*V20;  
    for(v2=v20; v2<v2f; v2++)  
    {  
        ... *D2...;  
        D2+=C2';  
    }  
    D1+= C1';  
}
```

# 下标地址优化计算的条件

- 相应的数组是常界数组：数组的上下界都是常量。
- 下标变量中的下标表达式是循环控制变量的线性表达式。
- 满足上述条件的称为可优化下标变量。
- 在C语言中，要求循环控制变量每次循环的变动是常数。

# 循环优化的实现

- 循环结构的识别
- 数据流分析
- 代码转换



# 循环结构的识别

- 对于源程序来说，识别循环是比较方便的。但是代码的优化是针对四元式序列的，所以识别循环必须针对流图进行。
- 定义8.3 如果流图中，从某个初始节点出发，每一条到达节点 $n$ 的路径都必须经过 $m$ ，那么称 $m$ 是节点 $n$ 的必经节点。记为 $m \text{ dom } n$ 。
  - 任何节点都是自己的必经节点。
  - $m$ 为 $n$ 的前驱， $n$ 为 $m$ 的后继。
- 直接必经节点：从初始节点到达 $n$ 的所有路径上，节点 $n$ 的最后一个必经节点称为直接必经节点。

# 循环满足的条件

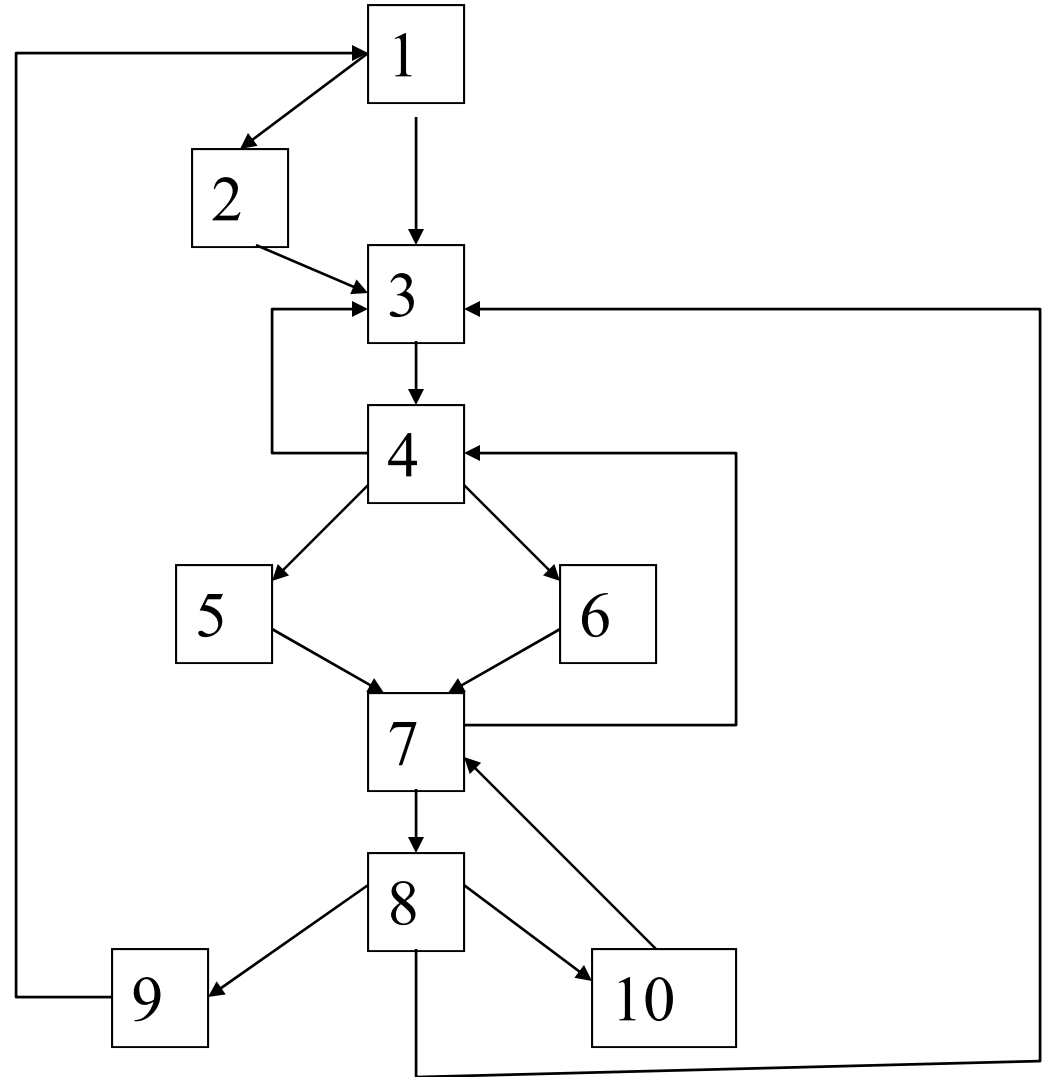
- 循环必须有唯一的入口节点，称为首节点。
- 对于循环中任何一个节点，必定至少有一个路径回到首节点。

# 回边和自然循环

- 定义8.4 假定流图中存在两个节点M和N满足 $M \text{ dom } N$ 。如果存在从节点N到M的有向边 $N \rightarrow M$ ，那么这条边称为回边。
- 定义8.5 在流图中，给定一个回边 $N \rightarrow M$ ，对应与这个回边的自然循环为：M，以及所有可以不经过M而到达N的节点。M为该循环的首节点。
- 用节点的集合表示自然循环。

# 自然循环的例子

- 3 dom 4
  - 回边4->3
- 4 dom 7
  - 回边7->4
- 10->7的自然循环  
{7, 8, 10}
- 7->4的自然循环  
{4,5,6,7,8,10}
- 4->3, 8->3的自然循环  
{3,4,5,6,7,8,10}



# 回边寻找算法

- 首先列出所有从首节点开始，不带圈的路径。
- 节点N的必经节点的集合为满足一下条件的节点M：
  - 所有包含N的路径P, M都在N的前面出现。
- 回边集合如下：
  - $\{N \rightarrow M \mid N \text{ 是一个节点, } M \text{ 在 } N \text{ 的必经节点集合中}\}$

# 寻找自然循环的算法

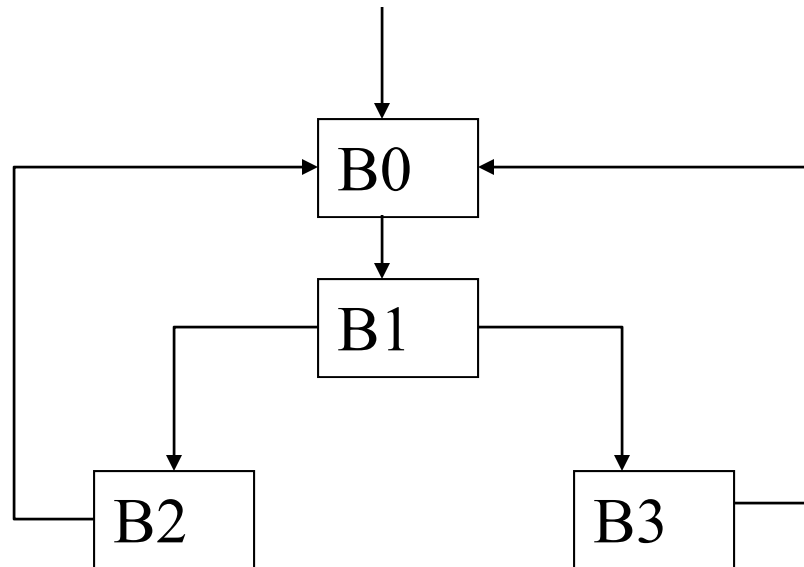
- 输入:回边  $\{N \rightarrow M\}$ ; 输出: 回边对应的自然循环.
- 算法:  
    设置  $\text{loop} = \{N, M\}$ ;  
    push(stack, N);  
    while non-empty(stack) do  
        {m = top(stack); pop(stack);  
        for m的每个前驱节点p  
            {if p is\_not\_in loop then  
                {loop += p; push(stack,p);}  
        }  
    }

# 算法的说明

- 节点M在初始时刻已经在loop中所以, M的前驱不可能被加入到loop中。
- 如果 $N \rightarrow M$ 不是回边, 那么, 首节点会被加入到loop中。此时算法不能得到自然循环。

# 相关概念

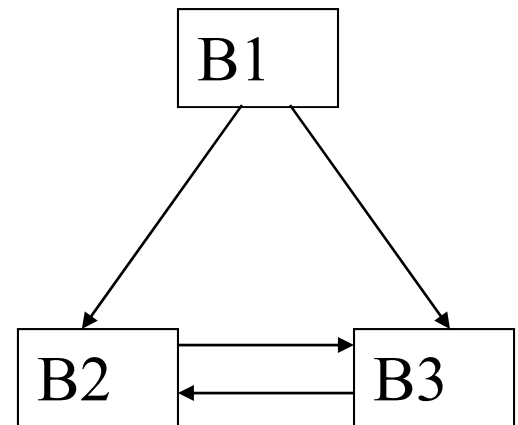
- 通常，循环互不相交，或者一个在另外一个里面。
- 内循环：不包含其他循环的循环称为内循环。
- 如果两个循环具有相同的首节点，那么很难说一个包含另外一个。此时把两个循环合并。





# 可归约流图

- 可归约流图：删除了其中回边之后，可以构成无环有向图的流图。
  - 特性：不存在循环外向循环内部的转移，进入循环必须通过其首节点。
- 实际的程序对应的流图一般都是可归约的流图。
- 没有goto语句的结构化程序的流图总是可归约的。一般使用goto语句的程序也是可归约的。



# 数据流分析相关概念

- 变量获得值的方式：
  - 通过赋值语句；
  - 通过输入语句；
  - 通过过程形式参数；
- 点：流图基本块中的位置，包括：第一个四元式之前，两个相邻四元式之间，和最后的四元式之间。
- 定值：使变量 $x$ 获得值的四元式称为对 $x$ 的定值，一般用四元式的位置表示。
- 引用点：引用某个变量 $x$ 的四元式的位置称为 $x$ 的引用点。

# 数据流分析的种类

- 到达-定值数据流方程
- 活跃变量数据流方程
- 可用表达式数据流方程

# 到达-定值数据流方程

- 到达-定值：假定 $x$ 有定值 $d$ ，如果存在一个路径，从紧随 $d$ 的点到达某点 $p$ ，并且此路径上面没有被注销，则称 $x$ 的定值 $d$ 到达 $p$ 。这表明，在 $p$ 点使用变量 $x$ 的时候， $x$ 的值可能是由 $d$ 点赋予的。
- 到达-定值链：设变量 $x$ 有一个引用点 $u$ ，变量 $x$ 的所有能过到达 $u$ 的一切定值称为 $x$ 在 $u$ 点处的引用-定值链，简称 $ud$ 链。
- 显然，通过变量 $x$ 在引用点 $u$ 的 $ud$ 链，可以判断 $x$ 是否循环不变的。

# 到达定值数据流方程（记号）

- $IN[B]$ : 表示基本块 $B$ 的入口点处各个变量的定点集合。
  - 如果 $B$ 中点 $p$ 之前有 $x$ 的定值点 $d$ , 且这个定值能够到达 $p$ , 则点 $p$ 处 $x$ 的ud链是 $\{d\}$ 。
  - 否则, 点 $p$ 处 $x$ 的ud链就是 $IN[B]$ 中关于 $x$ 的定值点的集合。
- $P[B]$ :  $B$ 的所有前驱基本块的集合。
- $GEN[B]$ : 各个变量在 $B$ 内定值, 并能够到达 $B$ 的出口点的所有定值的集合。
- $OUT[B]$ : 各个变量的能够到达基本块 $B$ 的出口点的所有定值的集合。
- $KILL[B]$ : 是各个变量在基本块 $B$ 中重新定值, 因此才块内部被注销的定值点的集合。

# 到达定值数据流方程

- $IN[B] = \bigcup OUT[p]$  where  $p \in P[B]$
- $OUT[B] = GEN[B] \cup (IN[B] - KILL[B])$
- 其中：
  - $GEN[B]$ 可以从基本块中求出：使用DAG图就可以得到。
  - $KILL[B]$ 中，对于整个流图中的所有 $x$ 的定值点，如果 $B$ 中有对 $x$ 的定值，那么该定值点在 $KILL[B]$ 中。

# 方程求解算法

- 使用迭代方法。

初始值设置为:  $IN[B_i] = \text{空}$ ;  $OUT[B] = GEN[B_i]$ ;

$change = \text{TRUE}$ ;

$\text{while}(change)$

{

$change = \text{FALSE}$ ;

    for each B do

$\{IN[B] = \bigcup OUT[p] \text{ where } p \text{ is in } P[B];$

$OUT[B] = GEN[B] \cup (IN[B] - KILL[B]);$

$oldout = OUT[B];$

$\text{if}(OUT[B] \neq oldout) change = \text{TRUE};\}$

}

# 算法例子

B1

d1: -	m	1	j
d2: =	n		j
d3: =	u2		a

B2

d4: +	i	1	i
d5: -	j	1	j

B3

d6: =	u2	a
-------	----	---

B4

d7: =	u3	i
-------	----	---

- $GEN[B1] = \{d1, d2, d3\}$
- $KILL[B1] = \{d4, d5, d6, d7\}$
- $GEN[B2] = \{d4, d5\}$
- $KILL[B2] = \{d1, d2, d7\}$
- $GEN[B3] = \{d6\}$
- $KILL[B3] = \{d3\}$
- $GEN[B4] = \{d7\}$
- $KILL[B4] = \{d1, d4\}$



# 计算过程

- 初始化:
  - $IN[B1] = IN[B2] = IN[B3] = IN[B4] = \text{空}$
  - $OUT[B1] = \{d1, d2, d3\}$ ,  $OUT[B2] = \{d4, d5\}$
  - $OUT[B3] = \{d6\}$ ,  $OUT[B4] = \{d7\}$ .
- 第一次循环:
  - $IN[B1] = \text{空}$ ;  $IN[B2] = \{d1, d2, d3, d7\}$ ;  $IN[B3] = \{d4, d5\}$ ;
  - $IN[B4] = \{d4, d5, d6\}$ ;  $OUT[B1] = \{d1, d2, d3\}$ ;
  - $OUT[B2] = \{d3, d4, d5\} \dots$
- 结果:
  - $IN[B1] = \text{空}$ ;  $OUT[B1] = \{d1, d2, d3\}$ ;
  - $IN[B2] = \{d1, d2, d3, d5, d6, d7\}$ ;  $OUT[B2] = \{d3, d4, d5, d6\}$ ;
  - $IN[B3] = \{d3, d4, d5, d6\}$ ;  $OUT[B3] = \{d4, d5, d6\}$ ;
  - $IN[B4] = \{d3, d4, d5, d6\}$ ;  $OUT[B4] = \{d3, d5, d6, d7\}$ ;

# 活跃变量数据流方程

- 判断在基本块出口之后，变量的值是否还被引用的这种判断工作称为活跃变量分析。
- 消除复写四元式的依据就是对活跃变量的分析。如果某个变量的值在以后不被引用，那么该复写四元式可以被消除。
- 对于变量 $x$ 和流图上的某个点 $p$ ，存在一条从 $p$ 开始的路径，在此路径上在对 $x$ 定值之前引用变量 $x$ 的值，则称变量 $x$ 在点 $p$ 是活跃变量，否则称 $x$ 在点 $p$ 不活跃。
- 无用赋值：对于 $x$ 在点 $p$ 的定值，在所有基本块内不被引用，且在基本块出口之后又是不活跃的，那么 $x$ 在点 $p$ 的定值是无用的。

# 记号

- $L\_IN[B]$ : 基本块 $B$ 的入口点的活跃变量集合。
- $L\_OUT[B]$ : 是在基本块 $B$ 的出口点的活跃变量集合。
- $L\_DEF[B]$ : 是在基本块 $b$ 内的定值，但是在定值前在 $B$ 中没有被引用的变量的集合。
- $L\_USE[B]$ : 表示在基本块中引用，但是引用前在 $B$ 中没有被定值的变量集合。
- 其中， $L\_DEF[B]$ 和 $L\_USE[B]$ 是可以从基本块 $B$ 中直接求得的量，因此在方程中作为已知量。

# 活跃变量数据流方程

- $L\_IN[B] = L\_USE[B] \cup (L\_OUT[B] - L\_DEF[B])$
- $L\_OUT[B] = \bigcup L\_IN[s]$ ，其中s遍历B的所有后继。
- 变量在某点活跃，表示变量在该点的值在以后会被使用。
- 第一个方程表示：
  - 如果某个变量在B中没有定值就使用了，显然，变量在入口处的值会被使用。
  - 如果这个变量在B的出口处活跃，并且B中没有对他进行定值，那么变量在入口处也是活跃的。
- 第二个方程表示：
  - 在B的某个后继中会使用该后继的入口处的值，那么他其实也可能使用B的出口处的值。

# 活跃变量数据流方程求解

- 设置初值:  $L\_IN[B_i] = \text{空}$ ;
- 重复执行一下步骤知道 $L\_IN[B_i]$ 不再改变:  
for( $i=1$ ;  $i < n$ ;  $i++$ )  
{  
     $L\_OUT[B_i] = U \ L\_IN[s]$ ;  $s$ 是 $B_i$ 的后继;  
     $L\_IN[B_i] = L\_USE[B_i] \cup (L\_OUT[B_i] - L\_DEF[B_i])$ ;  
}

# 活跃变量数据流方程例子

d1: -	m	1	j
d2: =	n		j
d3: =	u2		a

d4: +	i	1	i
d5: -	j	1	j

d6: =	u2	a
-------	----	---

d7: =	u3	i
-------	----	---

- $L\_DEF[B1] = \{i, j, a\}$
- $L\_USE[B1] = \{m, n, u1\}$
- $L\_DEF[B2] = \text{空}$
- $L\_USE[B2] = \{i, j\}$
- $L\_DEF[B3] = \{a\}$
- $L\_USE[B3] = \{u2\}$
- $L\_DEF[B4] = \{i\}$
- $L\_USE[B4] = \{u3\}$

# 迭代过程

- 第一次循环：
  - $L\_OUT[B1]=\text{空}$        $L\_IN[B1]=\{m,n,u1\}$
  - $L\_OUT[B2]=\text{空}$        $L\_IN[B2]=\{i,j\}$
  - $L\_OUT[B3]=\{i,j\}$        $L\_IN[B3]=\{i,j,u2\}$
  - $L\_OUT[B4]=\{i,j,u2\}$     $L\_IN[B4]=\{j,u2,u3\}$
- 第二次循环：
  - $L\_OUT[B1]=\{i,j,u2,u3\}$        $L\_IN[B1]=\{m,n,u1,u2,u3\}$
  - $L\_OUT[B2]=\{i,j,u2,u3\}$        $L\_IN[B2]=\{i,j,u2,u3\}$
  - $L\_OUT[B3]=\{i,j,u2,u3\}$        $L\_IN[B3]=\{i,j,u2,u3\}$
  - $L\_OUT[B4]=\{i,j,u2,u3\}$        $L\_IN[B4]=\{j,u2,u3\}$
- 第三次循环各个值不再改变，完成求解。

# 可用表达式数据流方程

- 如果一个流图的首节点到达点 $p$ 的每个路径上面都有对 $x \text{ op } y$ 的计算，并且在最后一个这样的计算到点 $p$ 之间某有对 $x, y$ 进行定值，那么表达式 $x \text{ op } y$ 在点 $p$ 是可用的。
- 表达式可用的直观意义：在点 $p$ 上， $x \text{ op } y$ 已经在之前被计算过，不需要重新计算。
- 注意：如果对于有间接赋值四元式的情况，需要保证最后的计算 $x \text{ op } y$ 的点之间不会间接改变 $x$ , 或者 $y$ 的值：比如指针不会指向 $x$ 或者 $y$ 的存储区域，特别是当 $x$ 为某个数组的时候。
- 书上的讲解是针对没有间接赋值四元式的情况处理的。



# 概念

- 对表达式的注销：如果某个基本块B对x或者y定值，且以后没有重新计算 $x \text{ op } y$ ，那么称B注销表达式 $x \text{ op } y$ 。
- 表达式的产生：如果某个基本块B对 $x \text{ op } y$ 进行计算，而以后并没有在对x或者y重新定值，那么称B产生表达式 $x \text{ op } y$ 。
- 表达式的全集：在计算某个流图中的可用表达式的时候，表达式的讨论范围被限定在该出现在流图中的四元式对应的表达式。

# 记号

- $E\_OUT[B]$ : 在基本块出口处的可用表达式集合。
- $E\_IN[B]$ : 在基本块入口处的可用表达式集合。
- $E\_GEN[B]$ : 基本块B所产生的可用表达式的集合。
- $E\_KILL[B]$ : 基本块B所注销掉的可用表达式的集合。
- $E\_GEN[B]$ 和 $E\_KILL[B]$ 的值可以直接从流图计算出来。因此在数据流方程中，可以将 $E\_GEN[B]$ 和 $E\_KILL[B]$ 当作已知量看待。

# E\_GEN[B]的计算

- 对于一个基本块B，E\_GEN[B]的计算过程如下：
- 初始设置：E\_GEN[B]=空；
- 顺序扫描每个四元式：
  - 对于四元式op x y z，把x op y加入E\_GEN[B]，
  - 从E\_GEN[B]中删除和z相关的表达式。
- 最后的E\_GEN[B]就是相应的集合。

# E\_KILL[B]的计算

- 设流图的表达式全集为E;
- 初始设置:  $EK = \text{空}$ ;
- 顺序扫描基本块的每个四元式:
  - 对于四元式  $op\ x\ y\ z$ , 把表达式  $x\ op\ y$  从EK中消除;
  - 把E中所有和z相关的四元式加入到EK中。
- 扫描完所有的四元式之后, EK就是所求的  $E\_KILL[B]$ 。

# 数据流方程

1.  $E\_OUT[B] = (E\_IN[B] - E\_KILL[B]) \cup E\_GEN[B]$
2.  $E\_IN[B] = \bigcap E\_OUT[p] \quad B \neq B1, p \text{ 是 } B \text{ 的前驱。}$
3.  $E\_IN[B1] = \text{空集}$

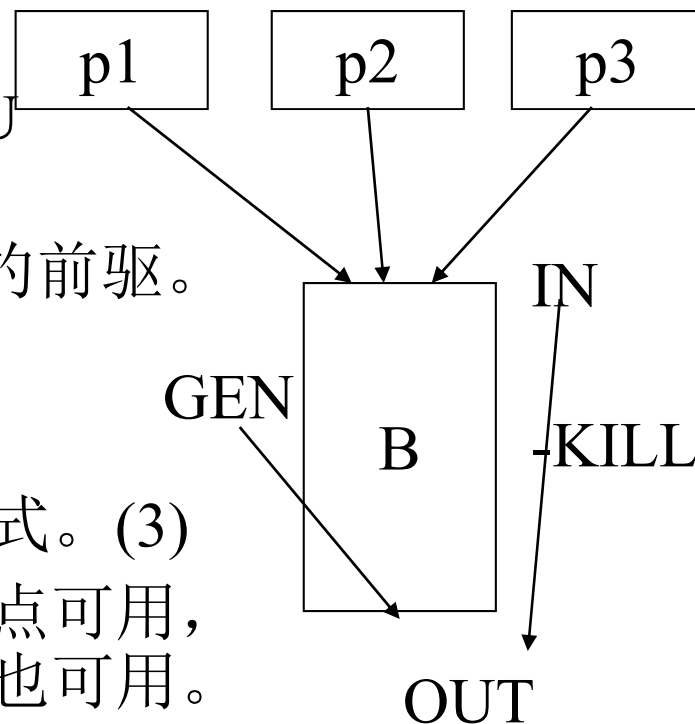
• 说明:

- 在程序开始的时候, 无可用表达式。(3)
- 一个表达式在某个基本块的入口点可用, 必须要求它在所有前驱的出口点也可用。

(2)

- 一个表达式在某个基本块的出口点可用, 或者该表达式是由它产生的; 或者该表达式在入口点可用, 且没有被注销掉。

(1)



# 方程求解算法

- 迭代算法
- 初始化:  $E\_IN[B1]=\text{空};$   
 $E\_OUT[B1]=E\_GEN[B1]; E\_OUT[Bi]=U-$   
 $E\_KILL[Bi](i \geq 2)$
- 重复执行下列算法直到 $E\_OUT$ 稳定:

FOR ( $i=2; i \leq n; i++$ )

{

$E\_IN[Bi] = \cap E\_OUT[p]$ ,  $p$ 是 $Bi$ 的前驱;

$E\_OUT[Bi] = E\_GEN[Bi] \cup (E\_IN[Bi] - E\_KILL[Bi]);$

}

# 算法说明

- 初始化值和前面的两个算法不同。  
E\_OUT[Bi]的初值大于实际的值。
- 在迭代的过程种，E\_OUT[Bi]的值逐渐缩小直到稳定。
- U表示四元式的全集，就是四元式序列中所有表达式x op y的集合。

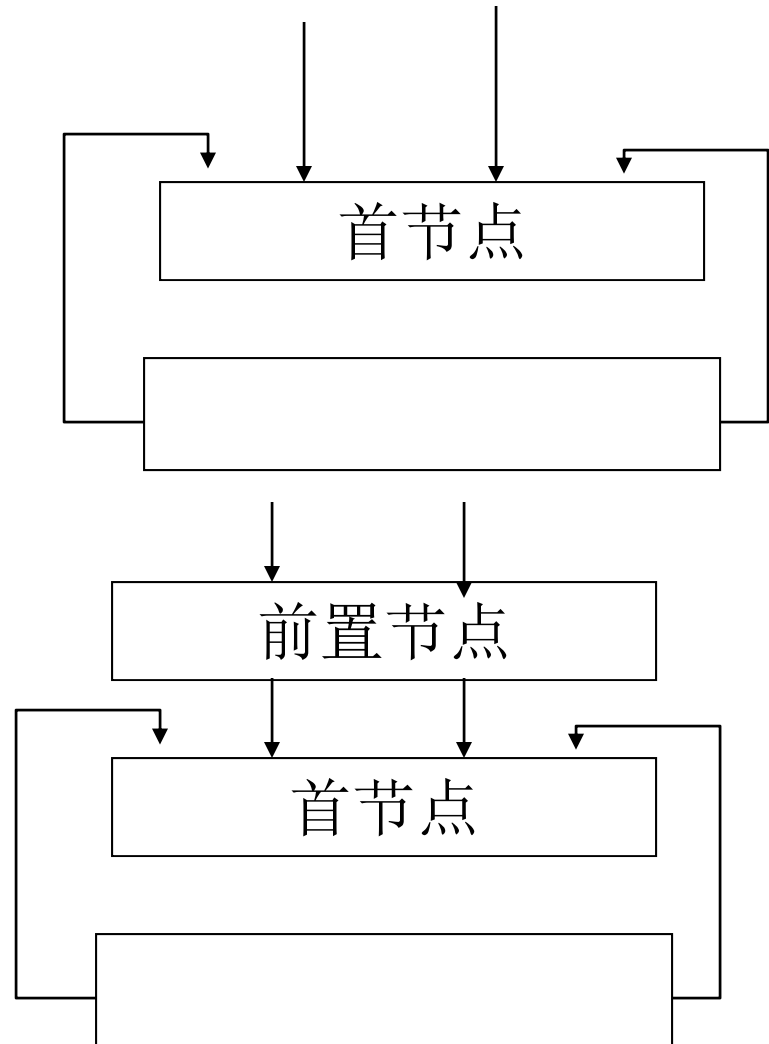
# 寻找循环不变表达式算法

- 输入：循环L，已经建立的UD链
- 输出：不变四元式
- 步骤1：如果四元式的运算分量或者是常数，或者其所有定值点在循环外部。
- 步骤2：重复执行下面的步骤：
  - 如果一个四元式没有被加过标记，且运算分量要么是常数，要么其UD链中的定值点在循环外，或者该定值点已经被加过标记。
- 说明：一个四元式的是不变的，当且仅当其分量在循环中是不变的。主要有三种情况：常量，定值点在循环外部，或者定值点的四元式也是不变的。

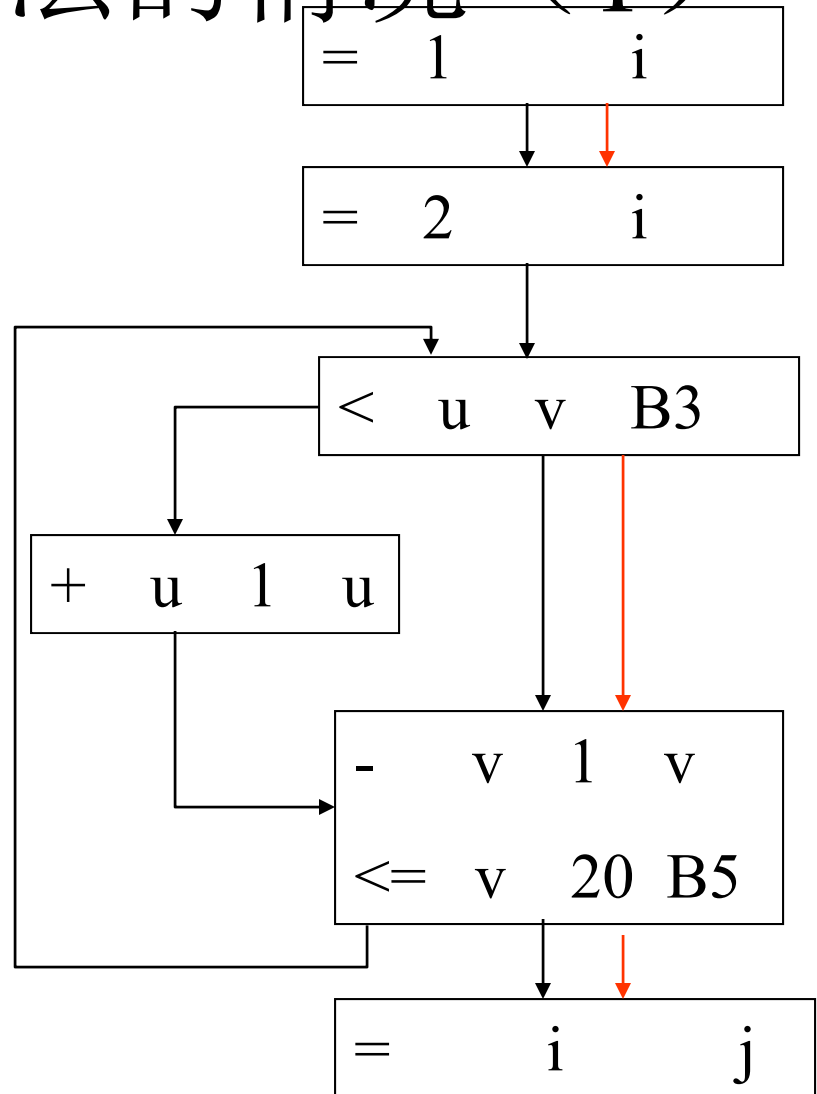
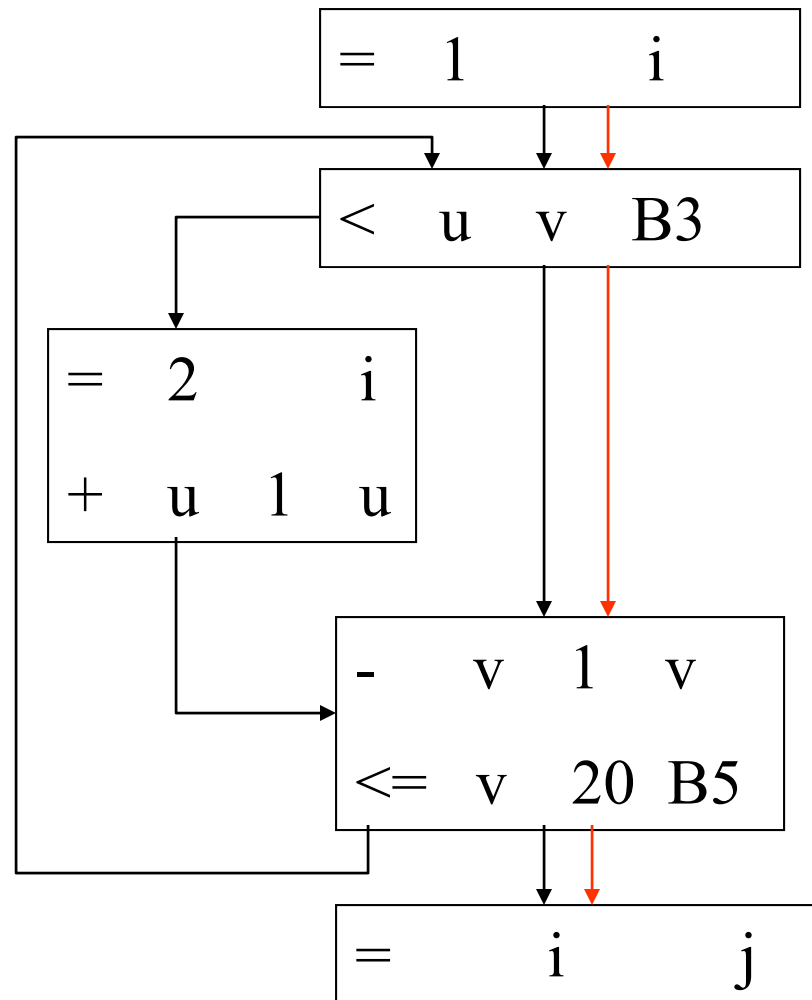


# 不变四元式外提方法

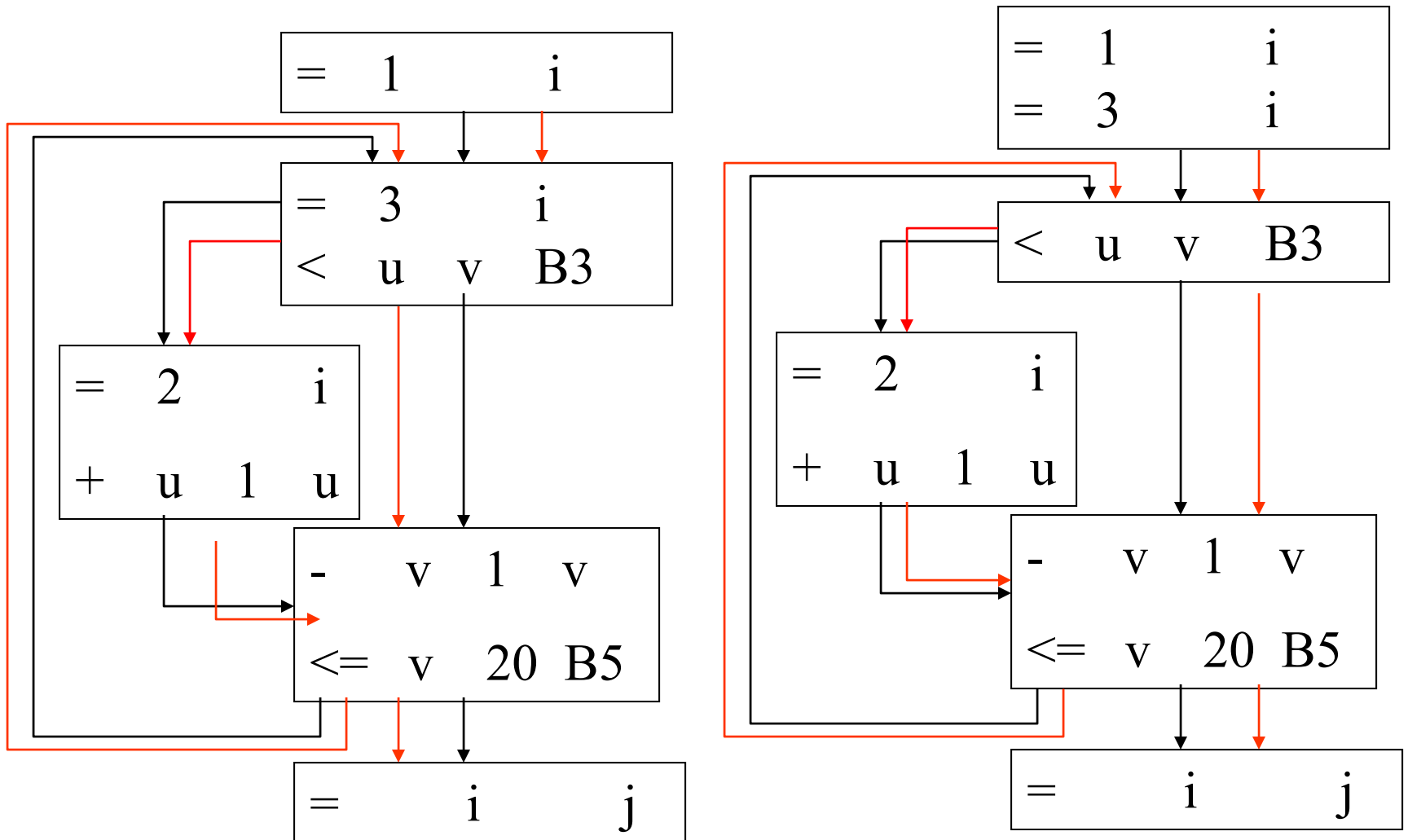
- 对于不变四元式，可以在进入循环之前首先计算该四元式的值，然后在循环内部使用该值。
- 可以将四元式的值外体到紧靠循环之前。



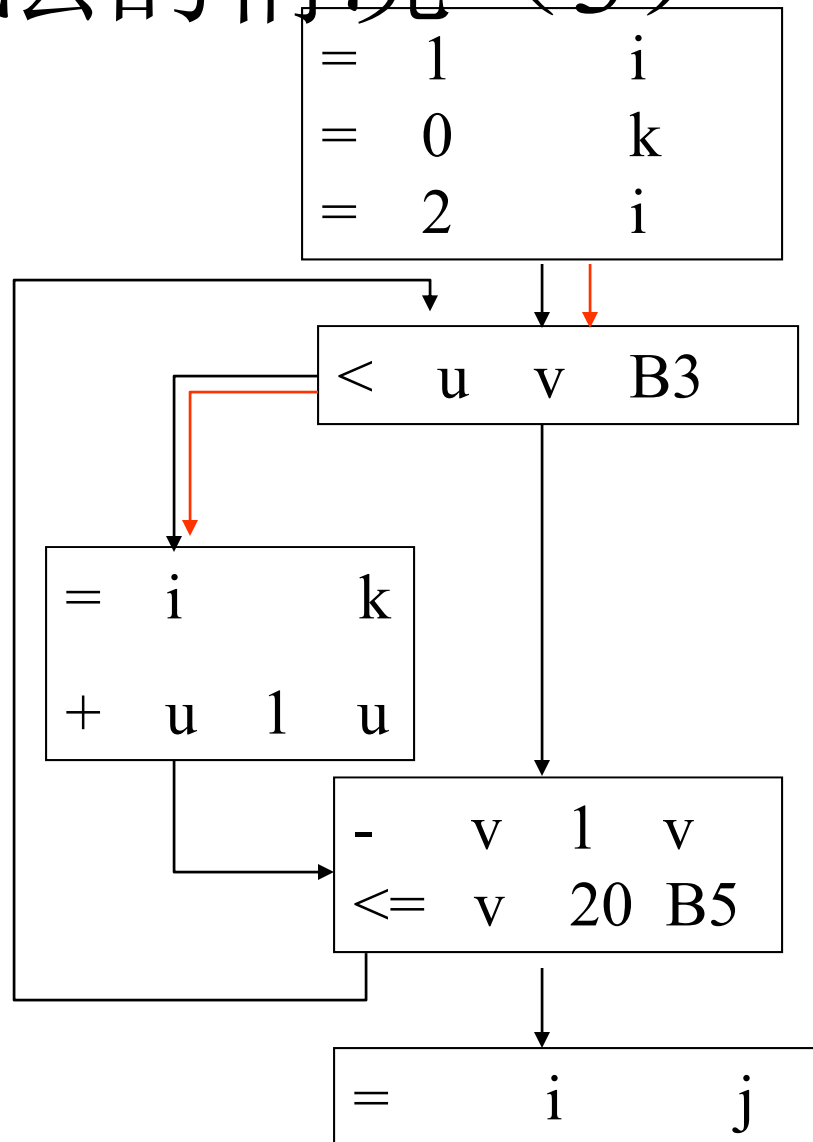
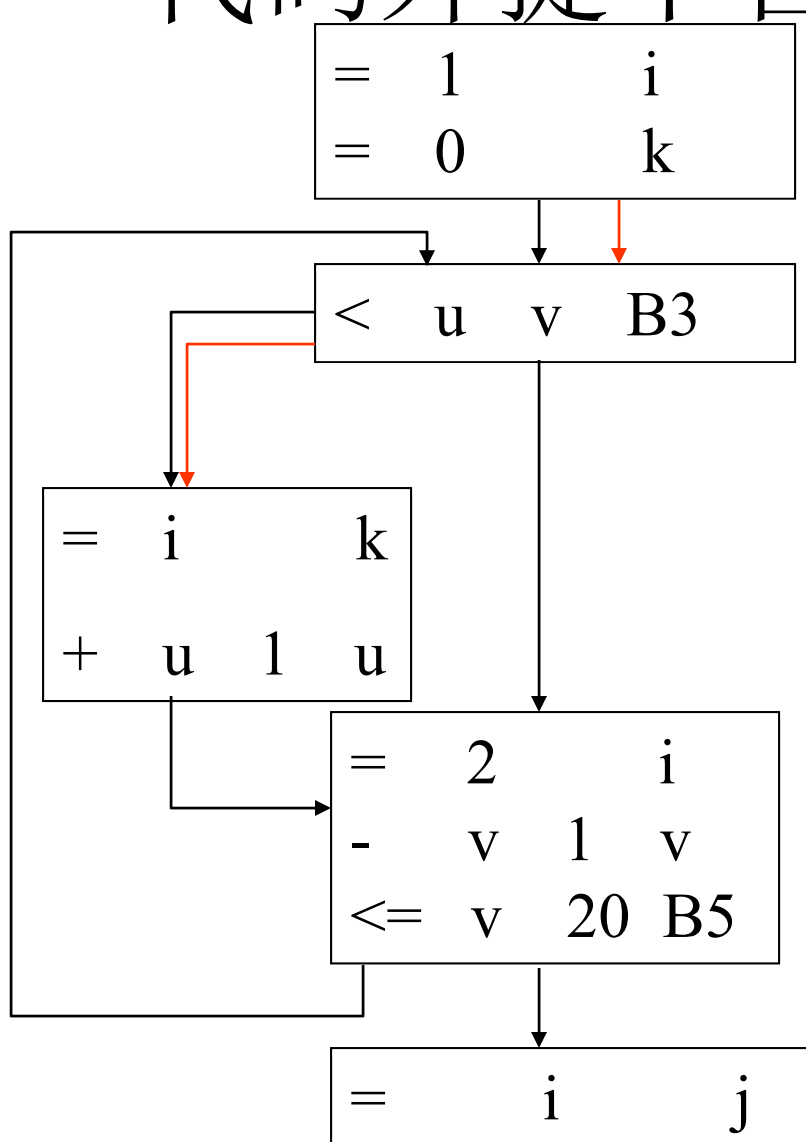
# 代码外提不合法的情况 (1)



# 代码外提不合法的情况 (2)



# 代码外提不合法的情况 (3)



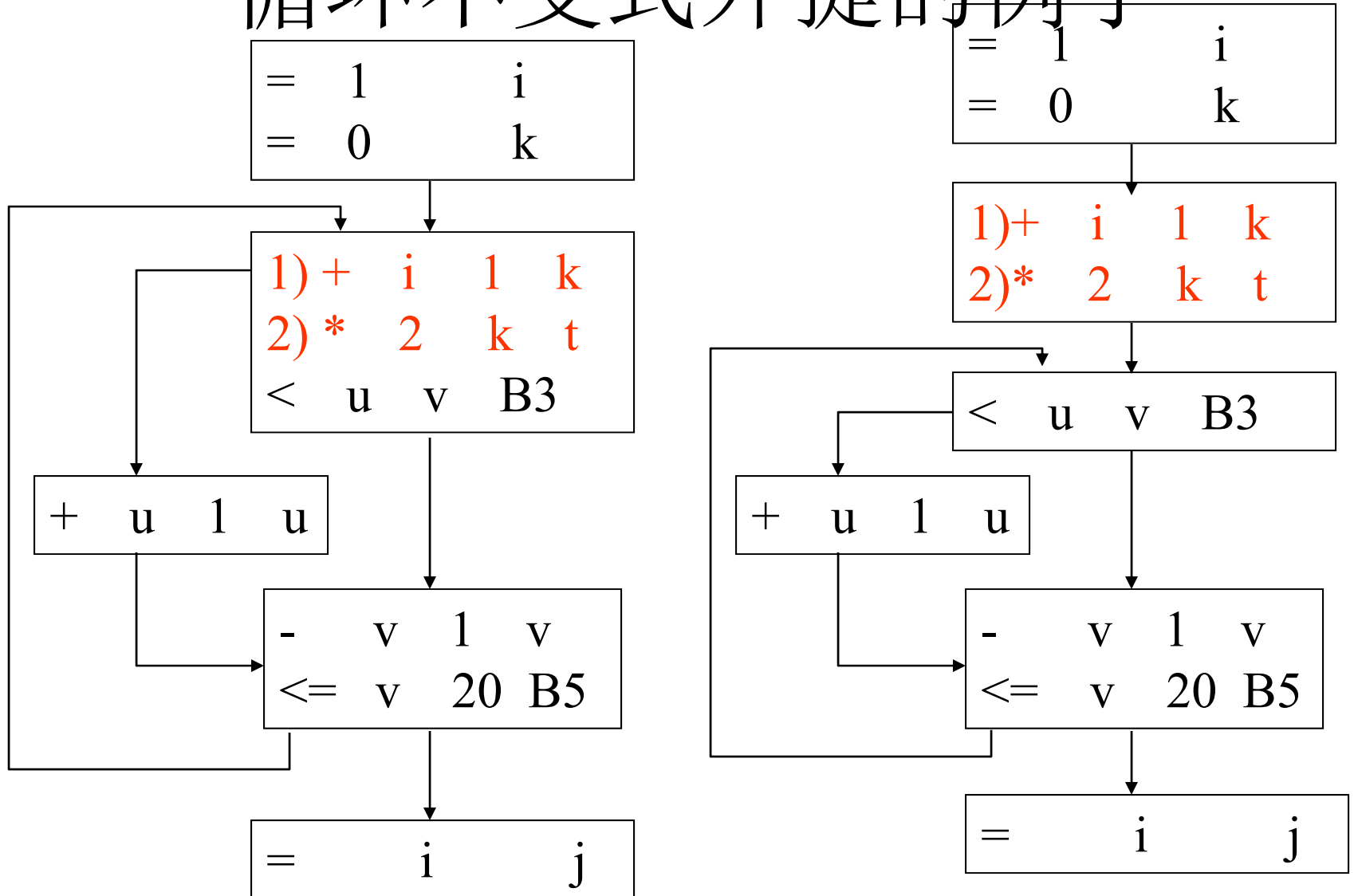
# 不变四元式外提的条件

- 不变四元式  $op \ x \ y \ z$  可以外提的条件：
  - Q 所在的基本块节点是循环所有出口节点的必经节点。出口节点是指有后继节点在循环外的节点。（反例：情况1）
  - 循环中  $z$  没有其他的定值点。（反例：情况2）
  - 循环中  $z$  的引用点仅由 Q 到达。（反例：情况3）

# 外提算法

- 步骤1：寻找一切不变四元式。
- 步骤2：对于找到的每个循环，检查是否满足上面说的三个条件。
- 步骤3：按照不变四元式找出的次序，把所找到的满足上述条件的四元式外提到前置节点中。
  - 如果四元式有分量在循环中定值，只有将定值点外提后，该四元式才可以外提。

# 循环不变式外提的例子



# 关于归纳变量的优化

- 基本归纳变量：如果循环中，对于 $i$ 的定值只有形状如 $i = i + c$ 的定值，那么 $i$ 称为循环的基本归纳变量。
- 归纳变量族：如果循环中对变量 $j$ 的定值都是形状如 $j = C1 * i + C2$ ，且 $i$ 为基本归纳变量，那么称 $j$ 为归纳变量，且属于 $i$ 族。 $i$ 属于 $i$ 族。
- 对于定值为 $C1 * i + C2$ 的 $i$ 族归纳变量 $j$ ，我们用 $(i, C1, C2)$ 来表示。
- 对于 $i$ 族的归纳变量 $(i, C1, C2)$ ，要求循环内部对此变量进行定值的时候，一定是赋给 $C1 * i + C2$ 。



## 例子（源程序）

```
for(i = 0; i<100; i++)
```

```
{
```

```
    j = 4*i;
```

```
    printf(“%i”, j);
```

```
}
```

- i是基本归纳变量。j是i族归纳变量，可以表示为(i, 4, 0)。

# 寻找循环的归纳变量算法

- 步骤1：扫描循环的四元式，找出所有基本归纳变量。对应于每个基本归纳变量的三元组如 $(i, 1, 0)$ 。
- 步骤2：寻找循环中只有一个赋值的变量 $k$ ，且对它的定值为如下形式， $k=j*b$ ,  $k=b*j$ ,  $k=j/b$ ,  $k=j+(-)b$ ,  $k=b+(-)j$ 。其中 $j$ 为基本归纳变量，或者已经找到的归纳变量。
  - 如果 $j$ 为基本归纳变量，那么 $k$ 属于 $j$ 族。 $k$ 对应的三元式可以确定。
  - 如果 $j$ 不是基本归纳变量，且属于 $i$ 族，那么我们还要求：
    - 循环中对 $j$ 的赋值以及和对 $k$ 的赋值之间没有对 $i$ 的赋值，且
    - 没有 $j$ 在循环外的定值可以到达 $k$ 的这个定值点。 $j$ 的三元式可以相应确定。

# 算法说明

- 关于j不是基本归纳变量的时候，算法中的两个要求实际上是保证：对k进行赋值的时候，j变量当时的值一定等于 $C1 * (i \text{ 当时的值}) + C2$ 。

# 归纳变量的计算强度削减算法

- 对于每个基本归纳变量 $i$ ，对其族中的每个归纳变量 $j(i, c, d)$ 执行下列步骤：
  - 建立新的临时变量 $t$ 。
  - 用 $t = c * i + d$ 四元式代替对 $j$ 的赋值。
  - 对于每个定值 $i = i + n$ 之后，添加 $t = t + c * n$ 。
  - 在前置节点的末尾，添加四元式 $t = c * i + d$ 和 $t = t + c * n$ 。使得在循环开始的时候,  $t = c * i + d$ 。
- 当两个归纳变量具有同样的三元式的时候，可以只建立一个临时变量。

# 算法的说明

- 在优化过程中，为 $j(i,c,d)$ 引入的变量 $t$ 保证了在任何时刻(不包括在对 $i$ 新定值后并且在 $t$ 重新定值前，但是由于两者的四元式时紧连的)都满足 $t=c*i+d$ 。
- 如果在某些情况下，程序运行时对 $j$ 的定值的次数远远少于对 $i$ 的定值，经过优化的程序需要对 $t$ 多次赋值，实际的效率可能降低。

# 归纳变量的删除

- 有些归纳变量的作用就是控制循环的次数。如果循环出口处的判断可以使用其它的变量代替，那么可以删除这些归纳变量。

# 归纳变量的删除算法

- 步骤1：对于每个基本归纳变量，取i族的某个归纳变量j（尽量使得c,d简单）。把每个对i的测试修改称为用j代替。
  - $\text{relop } i \ x \ B$  修改为  $\text{relop } i \ c*x+d \ B$ 。
  - $\text{relop } i_1 \ i_2 \ B$ ，如果能够找到三元组 $(i_1, c, d)$ 和 $(i_2, c, d)$ ，那么可以将其修改为  $\text{relop } j_1 \ j_2 \ B$  (假设  $c > 0$ )。
  - 如果归纳变量不再被引用，那么可以删除和它相关的四元式。
  - 如果基本归纳变量在循环出口处活跃，上面的算法不可以直接使用。
- 步骤2：考虑对j的赋值  $= t \ j$ 。如果每次对j引用和这个赋值四元式之间没有任何对t的赋值（此时，每次使用j的时候都有  $t=j$ ），可以在引用j的地方用t代替，同时删除四元式  $= t \ j$ （如果j在循环的出口处活跃，需要增加  $= t \ j$ ）。

# 全局公共子表达式消除

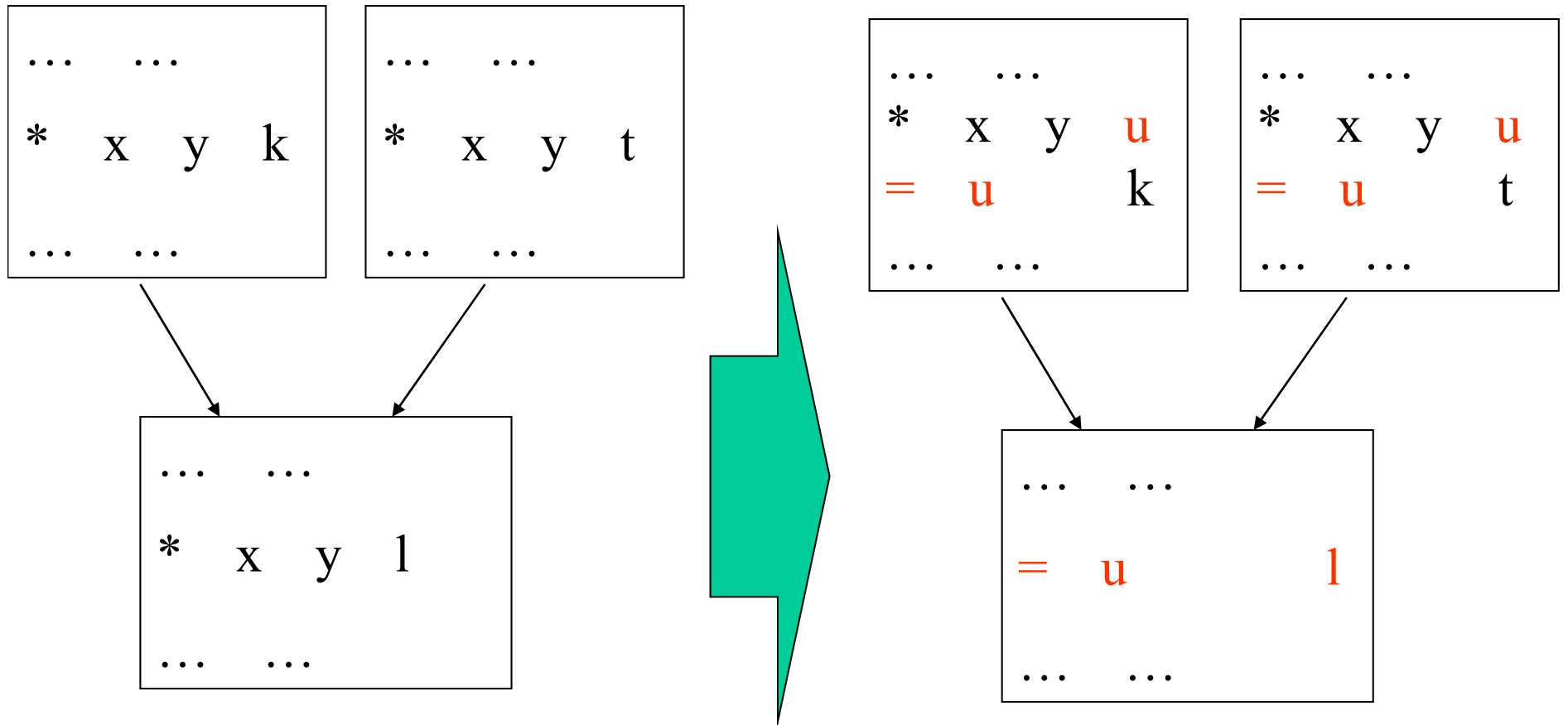
- 对于循环中某个四元式 $op\ x\ y\ z$ ，如果在所有到达这个四元式的路径上，已经计算了 $x\ op\ y$ ，且计算之后 $x, y$ 的值再没有修改过，那么显然我们可以使用以前计算得到的值。
- 可用表达式表达的就是这个概念。
- 如果有四元式 $op\ x\ y\ z$ ，且在四元式前，表达式 $x\ op\ y$ 可用，那么我们可以用下面的方法优化：在前面不同的地方计算 $x\ op\ y$ 时，把值记录到同一个临时变量 $u$ 里面。把这个四元式改变为 $=\ u\ \quad z$ 。



# 全局公共子表达式消除

- 对于四元式 $Q(\text{op } x \text{ y } z)$ ，如果 $x \text{ op } y$ 再 $Q$ 之前可用，那么执行如下步骤：
  - 从 $Q$ 开始逆向寻找，找到所有离 $Q$ 最近的计算了 $x \text{ op } y$ 四元式。
  - 建立新的临时变量 $u$ 。
  - 把步骤1中找到的四元式 $\text{op } x \text{ y } t$ 用下列四元式代替： $\text{op } x \text{ y } u$ 和 $= u \quad t$ 。
  - 用 $= u \quad t$ 替代 $Q$ 。

# 全局公共子表达式消除(图示)



# 复写四元式的消除

- 中间代码的生成可能产生四元式。
- 消除公共子表达式和删除归纳变量的时候，也会产生复写四元式。
- 通过复写四元式传播的变量大多数是临时变量。而对临时变量可以使用复写传播来消除复写四元式。
- 原理： $= x \quad y$ 的效果是 $x$ 和 $y$ 的值相等。如果某个引用 $y$ 的四元式运行的时候， $x$ 和 $y$ 的值一定相等。那么我们可以使用对 $y$ 的引用来替代对 $x$ 的引用。这样做有时可以消除复写四元式 $= x \quad y$ 。

# 消除复写四元式的条件

- 对于复写四元式  $d := x \quad y$ ，如果对  $y$  对应于  $d$  点的一切引用点  $u$  满足下列条件，那么我们可以删除  $d$ ，并且在这些引用点上用对  $x$  的引用代替对  $y$  的引用。
  - 此定值点是  $d$  到达  $u$  的唯一的  $y$  的定值。
  - 从  $d$  到达  $u$  的路径（可以包含圈和多个  $u$ ，但是不包含多个  $d$ ）上，没有对  $x$  的定值。
- 这两个条件实际上是说，当程序运行到达  $u$  点的时候， $x$  和  $y$  的值总是相等。

# 记号

- $C\_GEN[B]$ : 在基本块B中的所有复写四元式 $= x \quad y$ , 并且从该四元式开始到B的出口,  $x$ 和 $y$ 都没有再定值。
- $C\_KILL[B]$ : 包含了所有在流图中出现并且满足下列条件的复写四元式 $= x \quad y$ : B中存在对 $x$ 或者 $y$ 的定值, 并且之后在没有复写四元式 $= x \quad y$ 出现。
- $C\_IN[B]$ : 表示在B的入口点, 有效的复写四元式 $= x \quad y$ 的集合, 也就是到达入口的路径上都要经过 $= x \quad y$ 并且之后没有对 $x$ 或者 $y$ 定值。
- $C\_OUT[B]$ : 表示在B的出口点有效的复写四元式的集合。

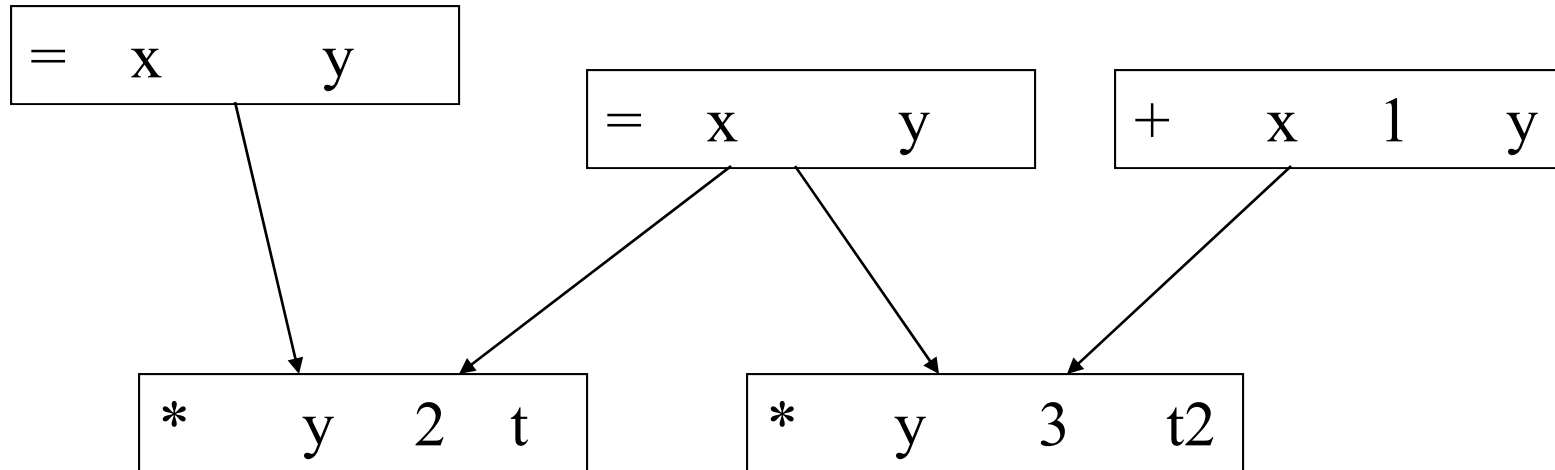
# 数据流方程

- $C\_OUT[B] = C\_GEN[B] \cup (C\_IN[B] - C\_KILL[B])$
- $C\_IN[B] = \cap E\_OUT[p]$ ; 其中B不等于B1, p是B的前驱。
- $C\_IN[B1] = \text{空集}$ 。
- 其中 $C\_GEN[B]$ 和 $C\_IN[B]$ 可以直接从流图得到, 所以在方程中作为已知量处理。
- 求解方法和可用表达式的求解相同。

# 复写传播算法

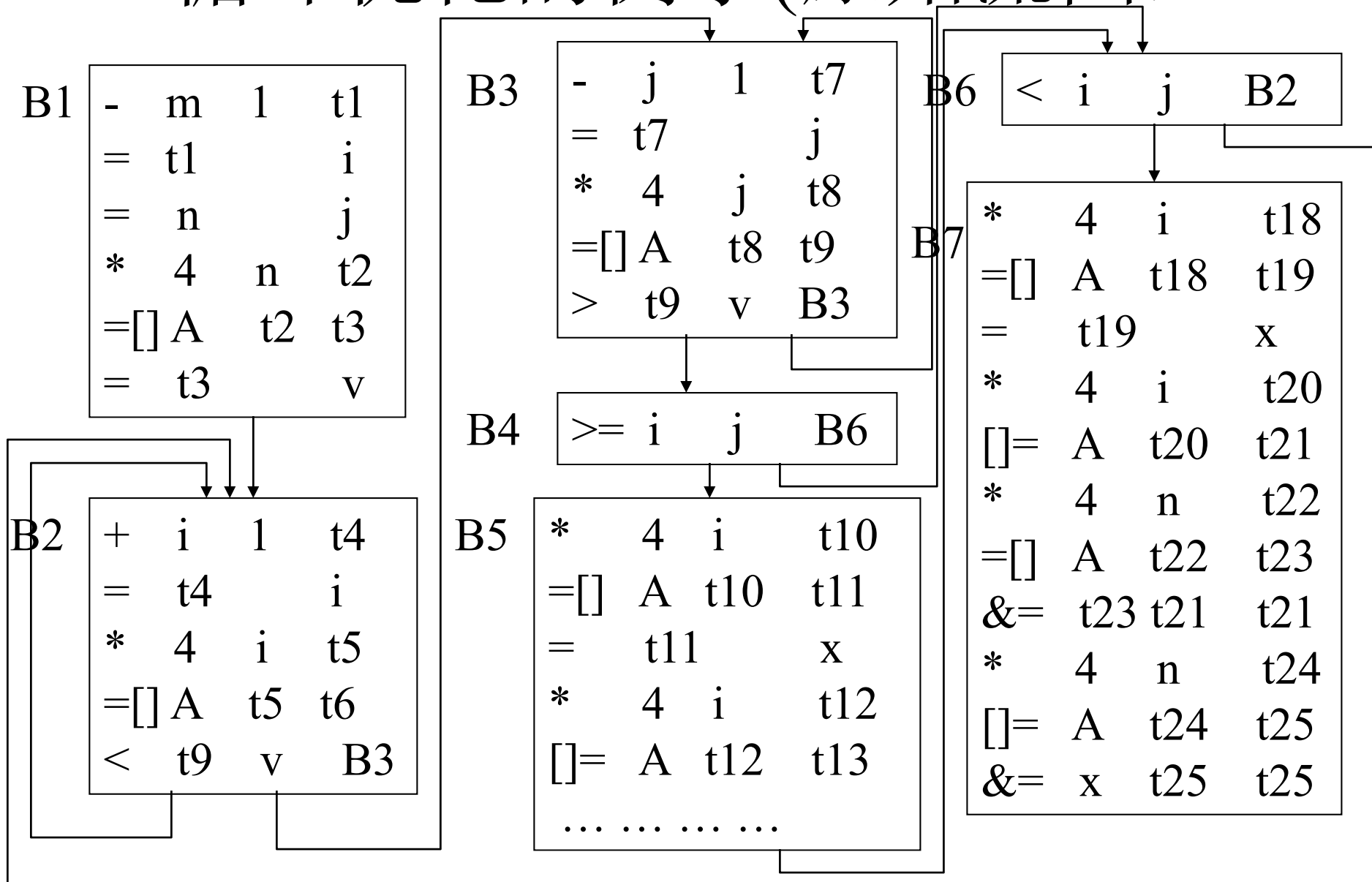
- 对于每个复写四元式  $d := x \quad y$  执行下列步骤。
  - 找出该  $y$  定值所能够到达的那些对  $y$  的引用。
  - 确定是否对于每个这样的  $y$ ， $d$  都在  $C\_IN[B]$  中， $B$  是包含这个引用的基本块，而且  $B$  中该引用的前面没有  $x$  或者  $y$  的定值。
  - 如果  $d$  满足条件(2)，删除  $d$ ，且把步骤1中找到的的对  $y$  的引用用  $x$  代替。

# 复写传播算法的例子





# 循环优化的例子(原始流图)



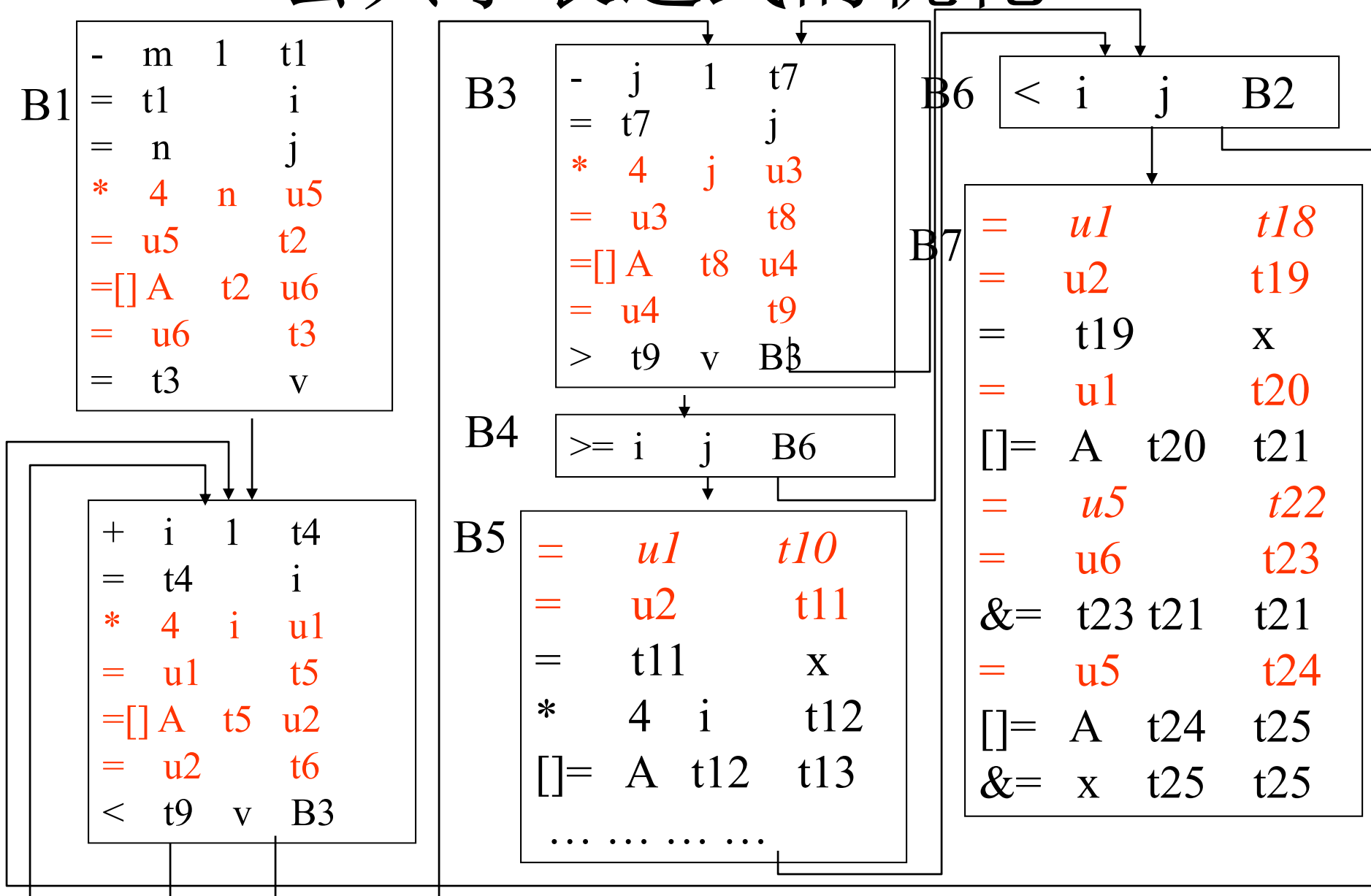
# 寻找回边以及自然循环

- 回边有：  $B2 \rightarrow B2$ ,  $B3 \rightarrow B3$ ,  $B6 \rightarrow B2$ 。
- $B2 \rightarrow B2$ 对应的循环是  $\{B2\}$
- $B3 \rightarrow B3$ 对应的循环是  $\{B3\}$
- $B6 \rightarrow B2$ 对应的循环是  $\{B2, B3, B4, B5, B6\}$
- 删除回边之后，得到的图是无回路的，所以这个流图是可归纳流图。

# 寻找公共子表达式

- $4*i$ : 在B5, B7入口处可用(U1)。
- $A[4*i]$ : 在B5, B7的入口处可用(U2)。
- $4*j$ : 在B5的入口处可用(U3)。
- $A[4*j]$ : 在B5的入口处可用(U4)。
- $4*n$ : 在B7的入口处可用(U5)。
- $A[4*n]$ : 在B7的入口处可用(U6)。

# 公共子表达式的优化



# 寻找归纳变量

- 基本归纳变量有 $i, j$ 。
- 归纳变量有：
  - $u1$ :  $i$ 族,  $(i, 4, 0)$
  - $u2$ :  $j$ 族,  $(j, 4, 0)$
- 如果分析得更加深入，会发现 $[] =$  四元式得到的地址也是归纳变量。但是，由于我们不考虑对间接赋值的优化，同时地址的处理也不在范围之内，所以我们没有列出。

# 归纳变量的优化

B1

```
- m 1 t1
= t1 i
= n j
* 4 n u5
= u5 t2
=[] A t2 u6
= u6 t3
= t3 v
* 4 i u1
* 4 j u3
```

B3

```
- j 1 t7
= t7 j
- u3 4 u3
=[] A u3 u4
= u4 t9
> t9 v B3
```

B6

```
< u1 u3 B2
```

B4

```
>= u1 u3 B6
```

B5

```
= u2 t11
= t11 x
= u1 t12
[]= A t12 t13
... ..
```

B7

```
= u2 t19
= t19 x
= u1 t20
[]= A t20 t21
= u5 t22
= u6 t23
&= t23 t21 t21
= u5 t24
[]= A t24 t25
&= x t25 t25
```

```
+ i 1 t4
= t4 i
+ u1 4 u1
=[] A u1 u2
= u2 t6
< t9 v B3
```

# 优化结果

B1

```
- m 1 t1
= t1 i
= n j
* 4 n u5
= u5 t2
=[] A t2 u6
= u6 t3
= t3 v
* 4 i u1
* 4 j u3
```

B3

```
- u3 4 u3
=[] A u3 u4
= u4 t9
> t9 v B3
```

B4

```
>= u1 u3 B6
```

B5

```
= u2 t11
= t11 x
= u1 t12
[]= A t12 t13
... ..
```

B6

```
< u1 u3 B2
```

B7

```
= u2 t19
= t19 x
= u1 t20
[]= A t20 t21
= u5 t22
= u6 t23
&= t23 t21 t21
= u5 t24
[]= A t24 t25
&= x t25 t25
```

# 窥孔优化

- 是指对代码局部进行改进的简单有效的技术。只考虑目标代码中的指令序列，把可能的指令序列替换成为更短更快的指令。
  - 冗余指令删除。
  - 控制流优化
  - 代数化简
  - 特殊指令使用



# 冗余指令删除

- 多余存取指令的删除：
  - MOV    b     R0
  - ADD    c     R0
  - MOV    R0    a
  - MOV    a     R0
  - SUB     d     R0
  - MOV    R0    c
- 第四条指令没有任何效果，可以删除。
- 如果有标号转向第四条指令，则不可以删除。

# 冗余指令删除

- 死代码删除：
  - 将不会被执行的代码删除。
  - 如果目标代码中有无条件转移指令，且其后的指令标号，那么后面的指令可以删除。
- 产生死代码的原因可以是调试信息，或者优化的结果。
- 对于条件转移指令，如果其条件永远成立或者永远不成立，可以转换成为等价的无条件转移指令。

# 控制流优化

- 在代码中出现转移指令到转移指令的情况时，某些条件下可以使用一个转移指令来代替。
- 转移到转移
  - GOTO L1;
  - ... ..
  - L1: GOTO L2;
- 显然第一个转移指令可以替代为GOTO L2.
- 如果在没有其他的指令转移到L1，那么第二个指令可以被删除。

# 控制流优化（续）

- 转移到条件转移：
  - L0: GOTO L1                      ...      ...
  - L1: IF b THEN GOTO L2;
  - L3: ... ..
- 如果只有一个转移指令到达L1，且L1前面是无条件转移指令，那么可以修改成为：
  - IF b THEN GOTO L2;
  - GOTO L3
  - L3: ... ..

# 控制流优化（续）

- 条件转移到转移
- IF b THEN GOTO L1
- ... ..
- GOTO L3
- 可以被修改为
- IF b THEN GOTO L3
- ... ..
- GOTO L3

# 代数化简

- 利用代数规则来优化代码。
- 例如：
  - 利用 $x=x+0$ 和 $x=x*1$ 来删除相应的指令。
  - 利用 $x^2=x*x$ 来削减计算强度。

# 特殊指令的使用

- 充分利用目标系统的某些高效的特殊指令来提高代码效率。
- 这些指令只可以在某些情况下使用。而识别这样的情况是优化的基础。
- 例如：INC指令可以用来替代加1的操作。

# 编译原理讲义

## (第九章 出错处理)

南京大学计算机系

赵建华



# 错误种类

- 词法错误：在词法分析阶段就可以发现的错误。
- 语法错误：程序的书写不符合语法规则。
- 语义错误：
  - 静态语义错误：编译程序可以发现。
  - 动态语义错误：源程序虽然能够被编译和执行，但是结果不对。一般是逻辑上的错误。
- 违反环境限制的错误：由于实现方面的问题，有些编译器不接受语言的全集。同时语言本身也有限制。比如：函数说明嵌套的深度，数组的最大层数。

# 错误复原

- 编写程序的过程中，出现错误是不可避免的。
- 编译程序在进行编译的时候，需要检查出程序的错误。并且给出提示。
- 当编译程序碰到源程序中的错误的时候，应该设法从错误中复原，并继续扫描程序以给出更多的提示信息。更多的信息使得程序员能够更加方便地修改程序。
- 可能复原是错误的。所以一般来讲，编译程序给出的第一个错误是可靠的，而其他的错误信息可能是不准确的。

# 错误复原的要点

- 株连信息的遏制
  - 株连信息是指由于源程序中的一个错误而导致编译程序向用户报告很多相关的出错信息。而这些信息是不真实的。
- 重复信息的遏制
  - 是指源程序中的一个错误而反映在源程序的多处。

# 株连信息的例子

- 已经说明过程`p(int a, int b)`；在使用的时候写成了`p(a.b)`。
- 编译程序扫描的时候，首先发现“.”是错误的，应为`a`不是结构类型的变量。然后，发现`b`是错误的。最后发现`p`的参数个数不同。
- 这些信息都是由于将‘,’误写做‘.’而引起的。

# 重复信息的例子

- 如果用户忘了申明变量i（或者声明的时候变量的拼写错误），而在程序中多出使用i。
- 编译程序在每次碰到对i的使用的时候都可能发现错误：变量i没有被定义。
- 这个错误信息将多次出现。而实际上这个错误是唯一的。

# 语法错误提示信息（自顶向下）

- 在自顶向下的扫描过程中，如果扫描程序碰到了非预期的符号输入，最简单的提示方法就是：unexpected symbol: ‘?’
- 自顶向下的分析过程中，扫描程序能够十分清楚地知道当前正在按照什么规则进行扫描，所以提示信息也可以给得更加详细：在扫描???的时候碰到非预期的符号??。

# 语法错误提示信息（自底向上）

- 当栈顶状态为 $S$ ，当前符号为 $a$ ，而分析表中 $A[S,a]$ 没有相应的值得时候，分析程序检查出源程序的错误。
- 简单的信息提示可以是：预期扫描 $\{???\}$ 但是扫描到了 $?$ 。其中预期扫描的符号是指所有使 $A[S,x]$ 有定义的符号 $x$ 。
- 通过分析 $S$ 对应的项集，我们可以知道，当前的分析程序试图规约到什么语法成分。那么提示信息何以更加详细。

# 错误的定位

- 在扫描程序检查到错误的时候，需要告诉用户，其错误发生在源程序的什么位置。
- 简单的方法是：在进行词法分析的时候，记录下每个词法单位（符号）的行和列。扫描程序碰到非预期的符号时，可以在提示信息中给出该符号的位置。



# 错误的复原

- 错误的复原是指编译程序遇到错误的时候，在给出错误提示后试图越过当前的错误继续扫描。
- 常用的方法是：对于某个语法成分，如果该语法成分有一个非常确定的符号作为结束符号。那么当扫描这个语法成分时发现错误，程序试图继续扫描到该结束符号。并把这个符号之前的东西规约成为相应的语法成分。

# 自顶向下时的错误复原

- 由于当前需要扫描的语法结构是很容易确定的，所以复原的工作相对容易。
- 在使用递归子程序法的时候，对应于每个语法符号的子程序可以添加两个参数：
  - 出现错误时，需要向后看那些符号。
  - 在扫描过程中，不能够越过哪些符号。
- 比如： `state(..., {';'}, {''})`

# 自底向上时的错误复原

- 首先要分析当前栈顶状态符号对应的项集。根据项集中的项，确定该项集的意义。确定该项集可能对应的规约符号，同时也确定如果碰到非预期的符号的时候，针对该项集应该如何扫描到哪个符号为止。
- 比如：项集  $\{E \rightarrow E.+E, E \rightarrow E.-E\}$ ，我们可以确定该项集试图规约得到一个表达式。如果我们没有碰到+或者-，那么应该扫描到一个‘;’为止。

# The tool ANTLR

赵建华

南京大学计算机科学与技术系

# 关于ANTLR

- 使用LL(K)分析技术，自动生成相应文法的语法分析程序（C, JAVA, ...）。
- 使用扩展的BNF作为文法的表示工具。
- 提供了相当强大的抽象语法树的构造工具。
- 网页地址：<http://wwwantlr.org>

# Antlr的输入文件

- 基本分为3个部分：
  - 全局的Option以及附加申明。
  - Parser的说明
  - Lexer的说明

# 全局的Option的例子

```
options {  
    language="Cpp";  
}
```

- 其他的Option可能包括:
  - namespace
  - mangleLiteralPrefix
  - ...
- 见文档中的options.html

# Parser部分的格式

*{ optional class preamble }*

*class YourParserClass extends Parser;*

*options*

*tokens...*

*parser rules...*

- Parser定义的主要部分在于parser rules的定义。



# Parser Rules

*rulename* : *alternative\_1*  
          | *alternative\_2*  
          ...  
          | *alternative\_n* ;

- 其中 **alternative\_1** 是一个 EBNF 表达式。比如：
  - statement : leftValue ASSIGN expression  
              | "if" expression "then" statement  
              ("else" statement)? "fi" ;

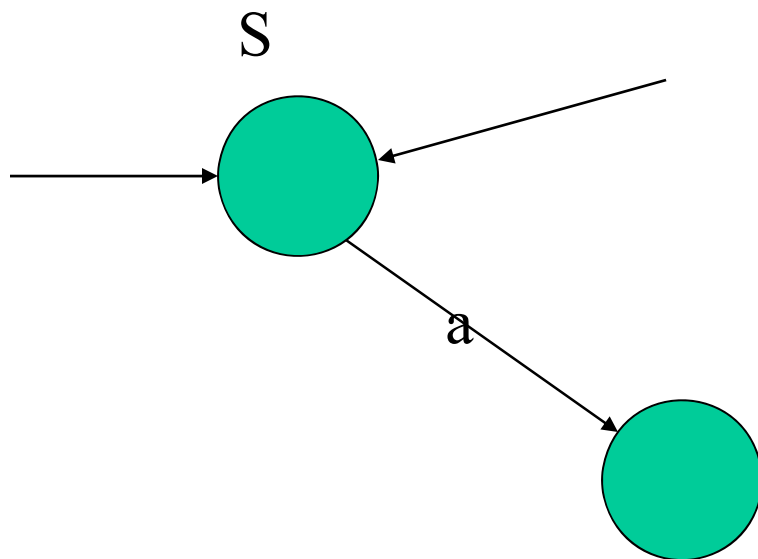
# Lexer部分的格式

# 有关习题

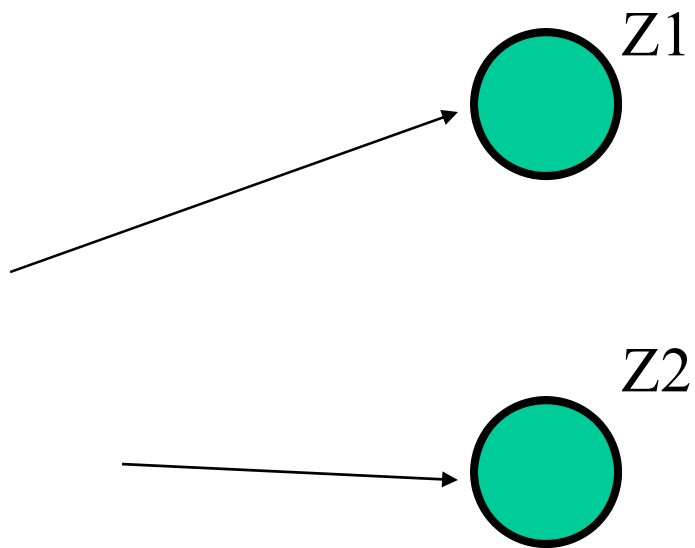
赵建华

# 正则文法与自动机

- 自动机到正则文法的转换：
  - 有边进入开始状态的情况。
  - 有多个接收状态的情况。



$$B = Sa \mid a$$

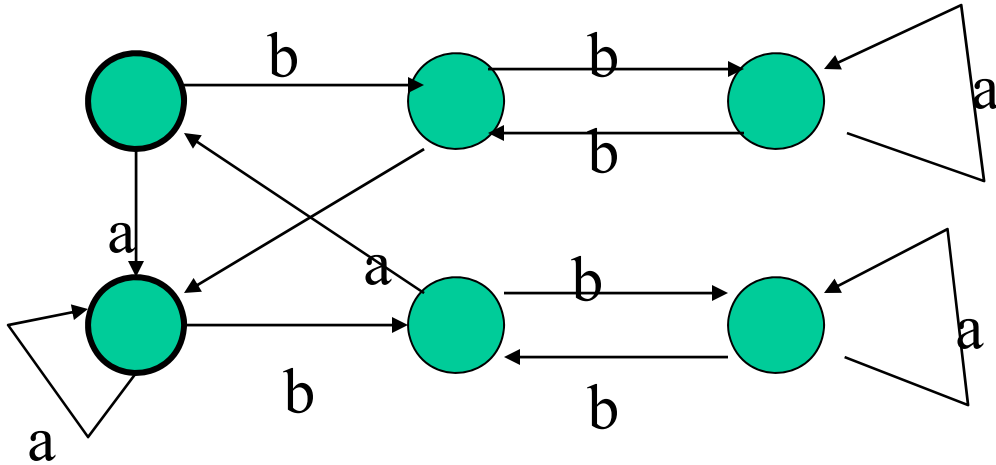


$$Z = Z_1 \mid Z_2$$

然后消除单规则

# 自动机

- 最小化时的问题
  - 首先应该消除死状态。
  - 在最小化的每一步，选择某个组。这个组被如下划分成为小组：
    - 同一小组中的状态，对于任意输入符号 $a$ ，要么他们都到达同一个组中的状态，要么都没有后继



- 初始分组  $A \{0,1\}$   $B \{2,3,4,5\}$
- 第一次:  $2-a->1$ ,  $3-a->0$ ,  $4-a->4$ ,  $5-a->5$   
 $2-b->3$ ,  $3-b->2$ ,  $4-b->5$ ,  $5-b->4$   
 可以分成:  $a \rightarrow A$ ,  $b \rightarrow B \{2,3\}$   
 $a \rightarrow B$ ,  $b \rightarrow B \{4,5\}$
- ....



# 文法与语言的证明的关系

- $L = L(G)$ 需要证明两个方面：
  - 所有 $L$ 中的符号串都能够由 $G$ 生成，
  - $G$ 所生成的符号串都在 $L$ 中间。
- 数学归纳法证明的是对于整数的命题 $M(n)$ 对于 $n$ 为整数成立。所以一般必须要把 $L=L(G)$ 改写成为：（也可能修改成为其他方式的命题）
  - 对于推导长度为 $n$ 得到的符号串必然在 $L$ 中间。
  - 对于 $L$ 中的长度为 $n$ 的符号串必然可以由 $G$ 推导得到。
- 证明中请不要使用“使用且只使用了所有规则”这样含糊的语句。

- $G: A ::= aAb \mid ab$
- $L(G) = \{a^n b^n \mid n \geq 1\}$
- 命题修改为
  - 推导长度为 $n$ 是推导出的符号串为  $a^n b^n$
  - $a^n b^n$ 必然可以由 $G$ 推导出来。

# 文法的描述

- 描述中必须指出识别符号。
- 规范分析是指最右推导（习题5, 2)

## 6.10

- 证明  $e = b \mid ae$  iff  $e = \{a\}^*b$
- 命题理解为: 如果  $e$  满足
  - $b$  属于  $|e|$
  - 如果某个符号串  $x$  在  $|e|$  中, 那么  $ax$  也在  $|e|$  中间。
  - 只有上面的符号串在  $|e|$  中。

# 递归下降法

- 在处理完毕之后，就是主程序调用识别符号之后，应该判断输入流是否已经读完。

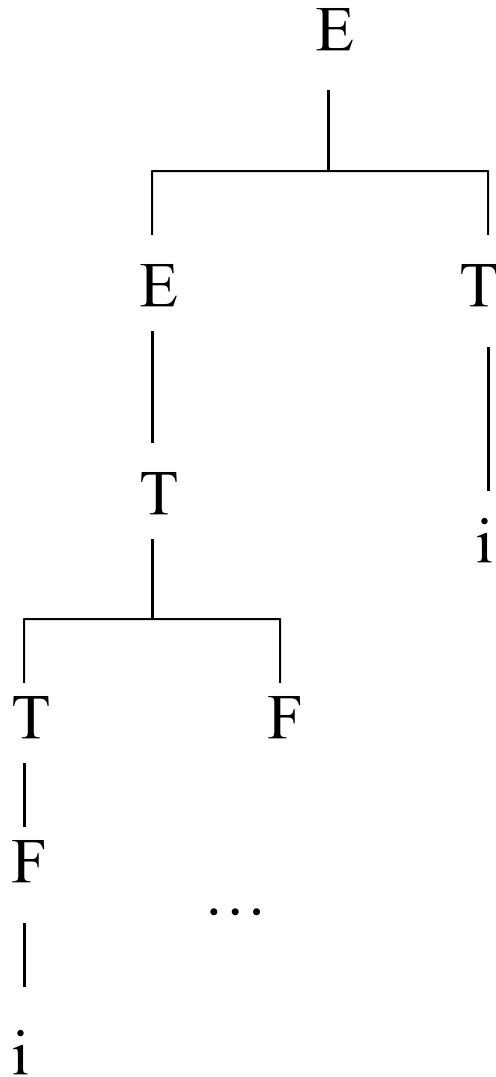
## 7.4

- $S ::= A \mid C$
- 应该首先使用LL(K)技术得到A和C的First。
- 在S对应的过程中，应该有语句首先判断应该调用A或者C

## 7.5

- $T' ::= T \mid \text{空}$
- 同样要使用LL(K)文法来得到如何使用选择规则的方法。

# 8.1



- $T > i, F > i \dots$
- 该题目的目的是熟悉优先关系定义的意义。实际的题目求解的过程中，不能够通过这样的方法得到。



# 11.11

- $E ::= E + E \mid EE \mid E^* \mid (E) \mid a \mid b \mid W$
- $W ::= aW \mid bW \mid \text{空}$
- 首先删除两个规则  $E \rightarrow a$  和  $E \rightarrow b$ 。将  $W \rightarrow \text{空}$  消除，得到  $W \rightarrow a$ ,  $W \rightarrow b$ ,  $E \rightarrow \text{空}$
- 根据题目的意思，在扫描的过程中，如果扫描到了  $a$  并且后面还是  $a$  的话，那么应该等到扫描到了  $b$  在将前面的  $a$  进行归约。
- 如果某个状态中有  $W \rightarrow a.W$  以及  $W \rightarrow x.$  时，如果后面的符号还是  $a$  的话，那么就移入。

# 11.11

- 有项集  $\{W \rightarrow a.W; W \rightarrow a.; W \rightarrow .a; W \rightarrow .b\}$
- 其中  $W \rightarrow a.W$  和  $W \rightarrow a.$  冲突且无法用 LR(k) 的方式解决。在输入符号为 a 的状态下冲突。
- 这是，我们可以要求在 a 时动作为移入，原因是：该项集表明在栈顶必然有一个 a，

## 12.2

- $B ::= B \text{ OR } L \mid L \dots$
- 应该添加一个新的识别符号Z,
- $Z \rightarrow B$ 或者 $Z \rightarrow B$ .
- $Z \rightarrow B \{ \text{printf}(\text{"\%i"}, B.\text{value}) \}$
- 不添加符号会重复打印
- 比如 `true OR true OR false`, 会使用规则  $B \rightarrow B \text{ OR } L$ 多次。

## 14.4

- Case 语句的翻译:
- 题目里面给出的不是文法，而是模式。
- 文法修改成为
- $\text{CaseSt} \rightarrow \text{"CASE" Expression "OF"}$   
 $\text{caseBranchList "END"}$
- $\text{caseBranchList} \rightarrow \text{caseBranchList ";" Branch};$
- $\text{Branch} \rightarrow \text{constValueList ":" Statement};$
- $\text{constValueList} \rightarrow \text{constValueList "," const}$

# 14.4

- 基本的模式上课的时候讲过。

# 实习一

# 功能

- 输入：
  - 一个正则表达式
  - 多个字符串
- 输出：
  - 对应于输入表达式的DFA.
  - 输出每个符号串是否对应的句子.
- 输入字母表为a-z, A-Z

# 描述

- 运行程序, 在命令行下输入正则表达式.以回车键结束.
- 程序判断正则表达式的语法是否正确(使用递归子程序法)
- 如果正确, 生成相应的DFA,并且在屏幕上打印出来.
- 提示用户输入字符串.
- 对于每个字符串,使用DFA来确定是否句子.屏幕上打印YES/NO.