

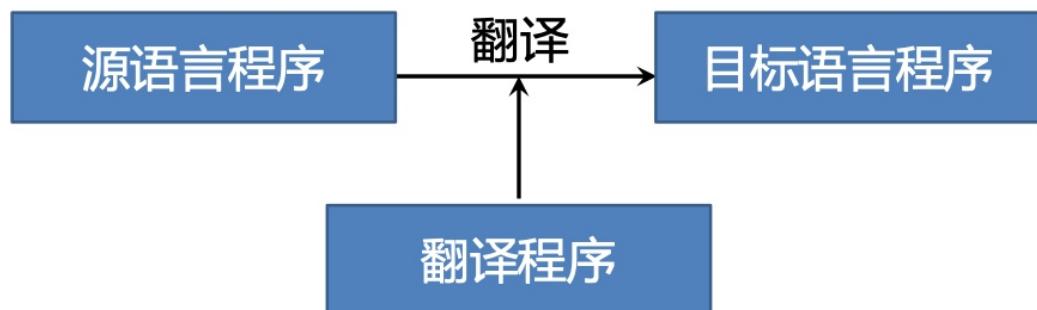
# 引论

## 一、编译与解释

### 1.1 编译程序

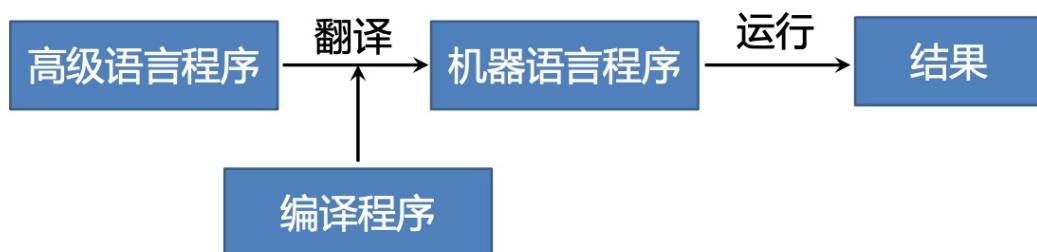
翻译程序 (Translator)

- 把某一种语言程序（源语言程序）等价地转换成另一种语言程序（目标语言程序）的程序



编译程序 (Compiler)

- 把某一种高级语言程序等价地转换成另一种低级语言程序（如汇编、机器语言）的程序



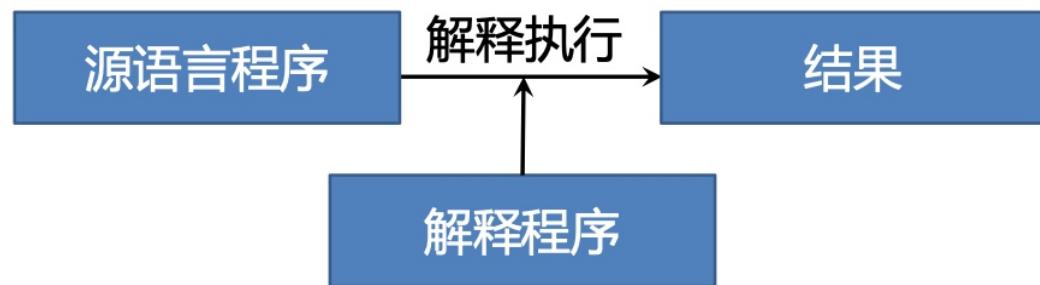
#### • 分类

- 诊断编译程序 (Diagnostic Compiler)
  - 发现代码中的错误
- 优化编译程序 (Optimizing Compiler)
  - 提高编译效率
- 交叉编译程序 (Cross Compiler)
  - 宿主机与目标机不同
  - 概念解释
    - 宿主机 – 运行编译程序计算机
    - 目标机 – 运行目标语言程序计算机
- 可变目标编译程序 (Retargetable Compiler)
  - 只要改变和目标机器有关的部分，就可以针对不同目标平台生成不同的目标机器上的代码

## 1.2 解释程序

解释程序 (Interpreter)

- 把源语言写的源程序作为输入，但不产生目标程序，而是边解释边执行源程序



## 二、计算思维

### 2.1 概述

「定义」计算思维是运用计算机科学的基础概念去求解问题、设计系统和理解人类的行为，它包括了一系列广泛的计算机科学的思维方法。

「分类」

- 抽象
- 自动化
- 问题分解
- 递归
- 权衡
- 保护、冗余、容错、纠错和恢复
- 利用启发式推理来寻求解答
- 在不确定情况下的规划、学习和调度
- ...

### 2.2 编译中的计算思维

#### 2.2.1 抽象

- 方法
  - 忽略一个主题中与当前问题（或目标）无关的那些方面
  - 从众多的事物中抽取出共同的、本质性的特征，舍弃其非本质的特征
  - 是一种从个体把握一般、从现象把握本质的认知过程和思维方法
- 编译中的体现
  - 有限自动机
  - 形式文法

- ...

## 2.2.2 自动化

- 方法
  - 将抽象思维的结果在计算机上实现，是一个将计算思维成果物化的过程，也是将理论成果应用于技术的实践
  - 自动化的思维方法不仅体现在编译程序本身的工作机制上，更体现在了编译程序的生成工具的研究和设计上
- 编译中的体现
  - 有限自动机
  - 预测分析程序
  - 算符优先分析
  - LR分析
  - ...

## 2.2.3 分解

- 方法
  - 将大规模的复杂问题分解成若干个较小规模的、更简单的问题加以解决
- 编译中的体现
  - 中间语言的引入
  - 编译多阶段
  - 多遍分析过程
  - ...

## 2.2.4 递归

- 方法
  - 问题的解决依赖于类似问题的解决，只不过后者的复杂程度或规模较原来的问题更小
- 编译中的体现
  - 递归下降分析
  - 基于树遍历的属性计算
  - 语法制导翻译
  - ...

## 2.2.5 权衡

- 方法
  - 理论可实现 vs. 实际可实现
- 编译中的体现
  - 尽管上下文无关文法不是万能的，但我们依然用它来处理高级程序设计语言
  - 优化措施的选择

## 三、编译过程

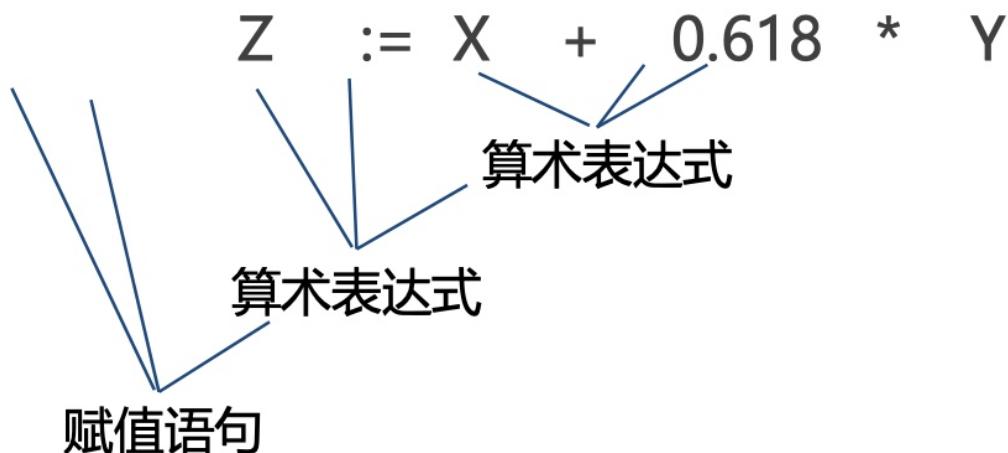
### 3.1 词法分析

- 「任务」
  - 输入源程序，对构成源程序的字符串进行扫描和分解，识别出单词符号
- 「依循的原则」
  - 构词规则
- 「描述工具」
  - 有限自动机

for      i           :=      1      to      100      do  
基本字 标识符 赋值号 整常数 基本字 整常数 基本字

### 3.2 语法分析

- 「任务」
  - 在词法分析的基础上，根据语法规则把单词符号串分解成各类语法单位(语法范畴)
- 「依循原则」
  - 语法规则
- 「描述工具」
  - 上下文无关文法



### 3.3 中间代码生成

- 「任务」
  - 对各类语法单位按语言的语义进行初步翻译
- 「依循原则」
  - 语义规则
- 「描述工具」

- 属性文法
- 「中间代码」
  - 三元式、四元式、树、...

## 3.4 优化

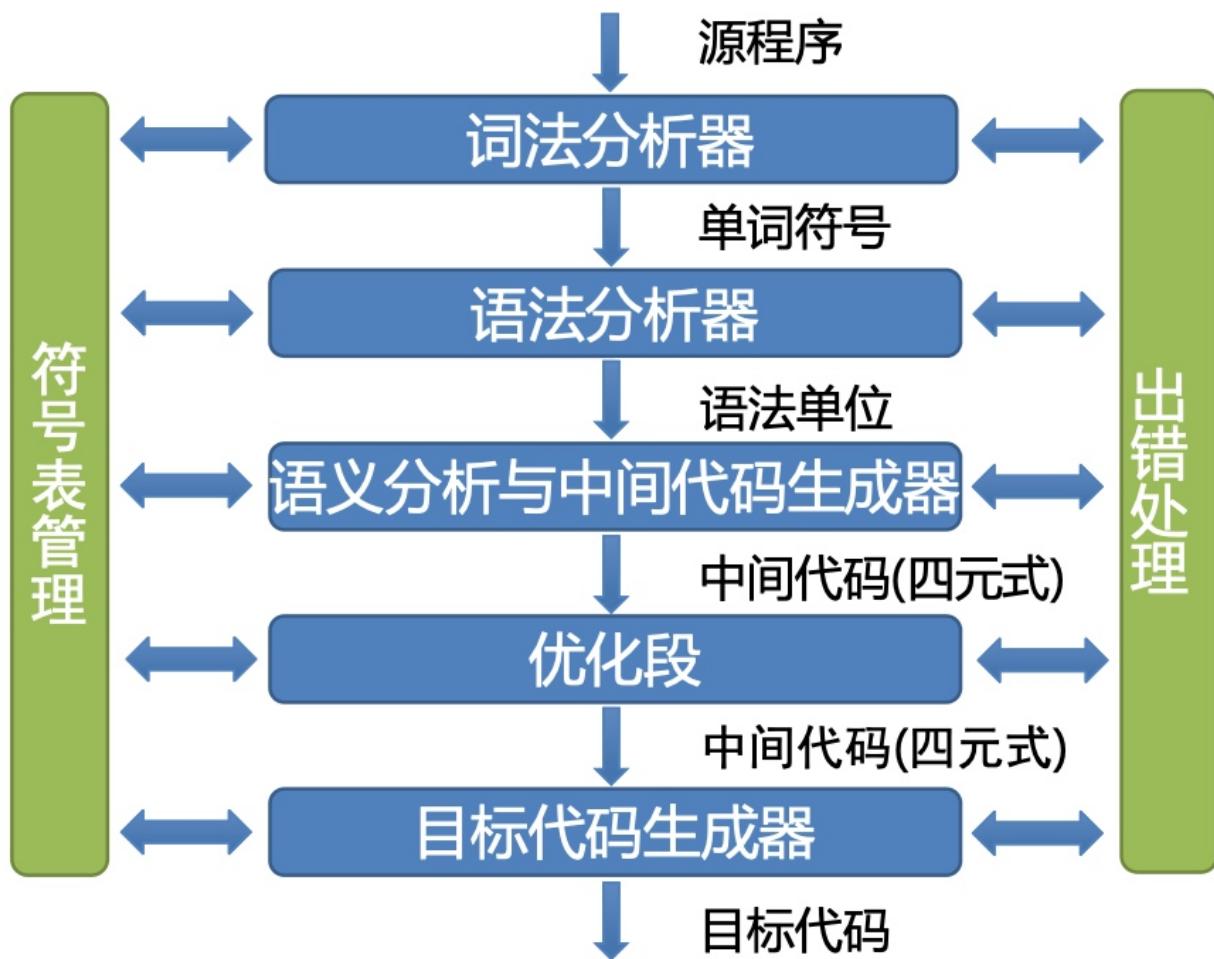
- 「任务」
  - 对前阶段产生的中间代码进行加工变换，以期在最后阶段产生更高效的目标代码
- 「依循原则」
  - 程序的等价变换规则

## 3.5 目标代码产生

- 「任务」
  - 把中间代码转换成特定机器上的目标代码
- 「依赖」
  - 依赖于硬件系统结构和机器指令的含义
- 「三种形式的目标代码」
  - 汇编指令代码：需要进行汇编
  - 绝对指令代码：可直接运行
  - 可重新定位指令代码：需要链接

# 三、编译程序结构

## 3.1 总框图



### 3.1.1 出错处理

- 出错处理程序
  - 发现源程序中的错误，把有关错误信息报告给用户
- 语法错误
  - 源程序中不符合语法（或词法）规则的错误
  - 非法字符、括号不匹配、缺少;、...
- 语义错误
  - 源程序中不符合语义规则的错误
  - 说明错误、作用域错误、类型不一致、...

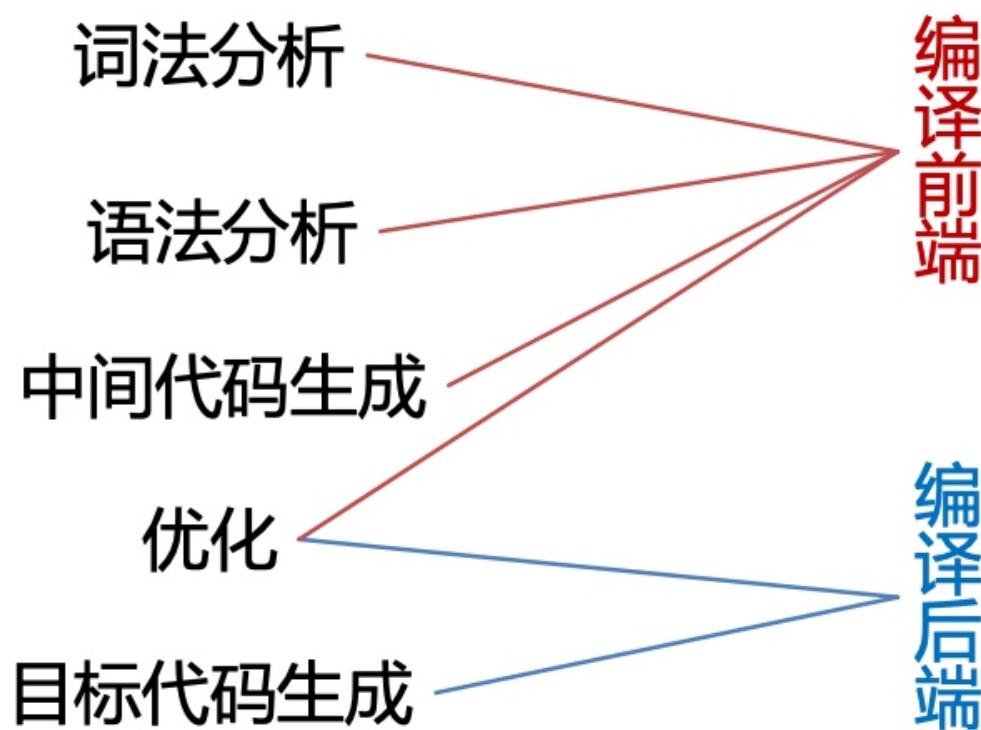
### 3.1.2 遍 (pass)

- “遍”：对源程序或源程序的中间表示从头到尾扫描一次
- 阶段 v.s. 遍
  - 一遍可以由若干段组成
  - 一个阶段也可以分若干遍来完成

## 3.2 前端后端



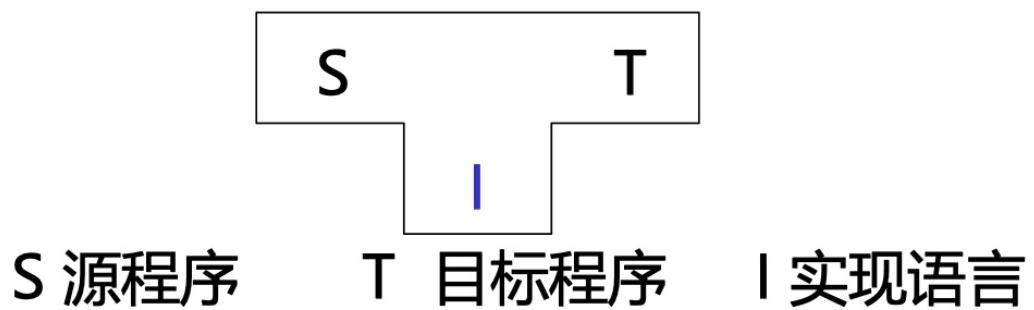
- 编译前端
  - 与源语言有关，如词法分析、语法分析、语义分析与中间代码产生，与机器无关的优化
- 编译后端
  - 与目标机有关，与目标机有关的优化、目标代码生成
- 分离的好处
  - 程序逻辑结构清晰
  - 优化更充分，有利于移植



## 四、编译程序生成

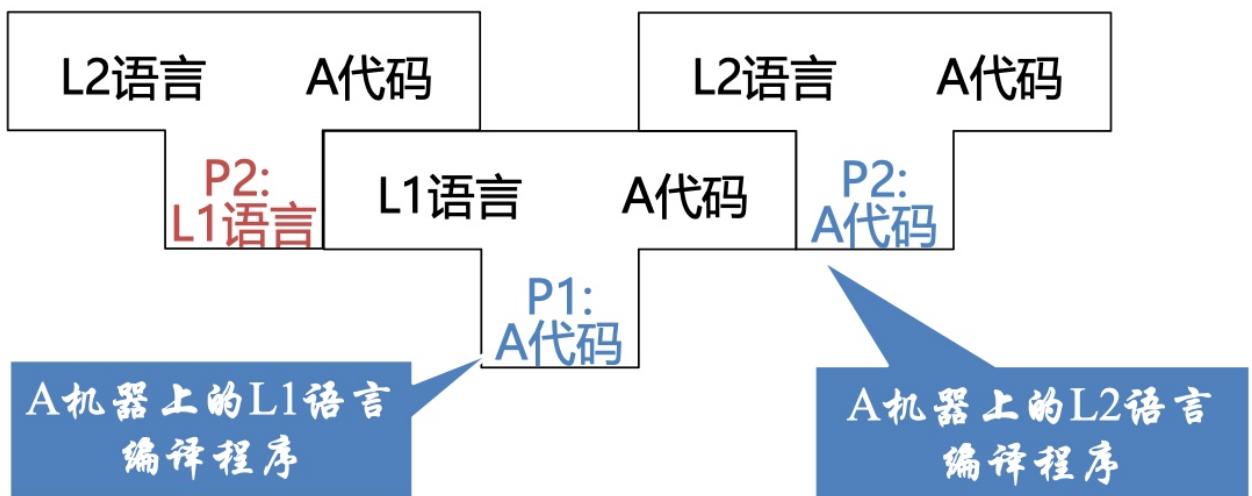
### 4.1 概述

- 以汇编语言和机器语言为工具
  - 优点：针对具体机器，充分发挥计算机功能，程序效率高
  - 缺点：程序难读、难写、易出错、难维护、生产的效率低
- 高级语言
  - 程序易读、易理解、容易维护、生产效率高
  -

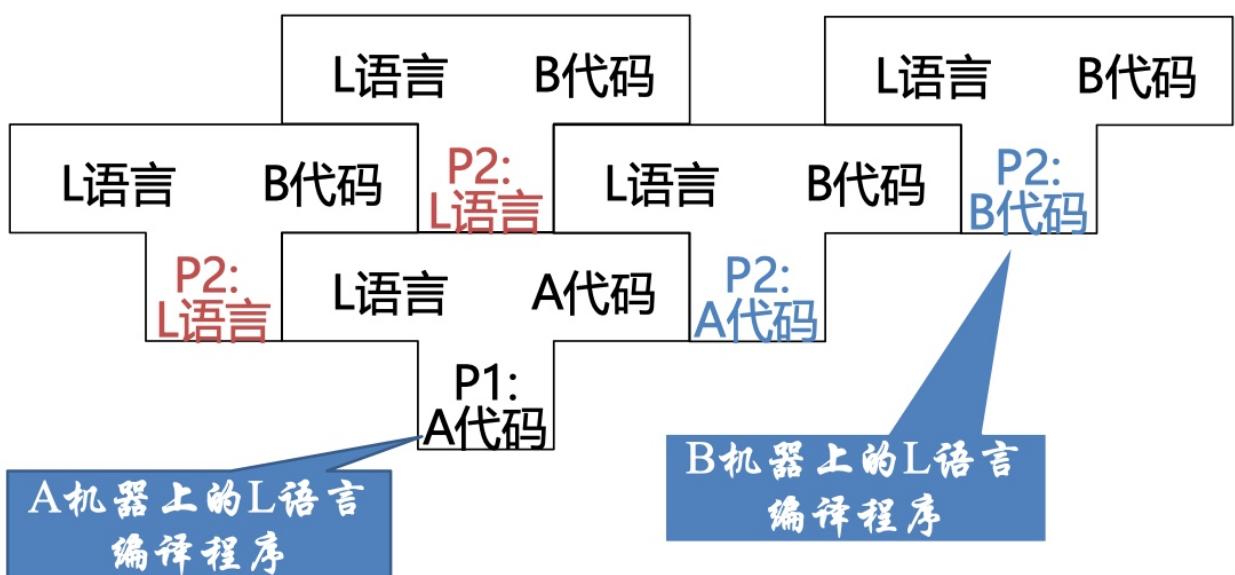


## 4.2 编译程序生成与移植

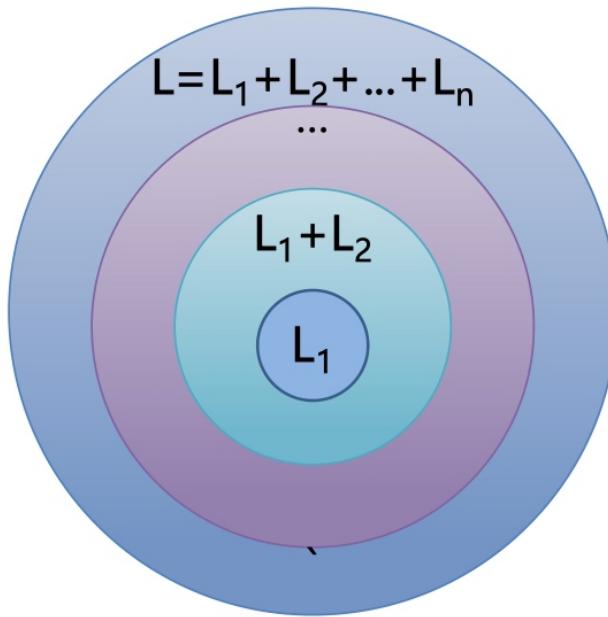
- 「生成」利用已有的某种语言的编译程序实现另一种语言的编译程序



- 「移植」把一种机器上的编译程序移植到另一种机器上



- 「自编译方式」不断扩大涵盖语言的范围



- 「自动生成」



## 高级程序设计语言

### 一、语言概述

#### 1.1 语法 v.s. 语义

- 程序本质上是一定字符集上的字符串
- 语法：一组规则，用它可以形成和产生一个合式（well-formed）的程序
  - 定义了程序的形式结构
  - 定义语法单位的意义属于语义问题
  -

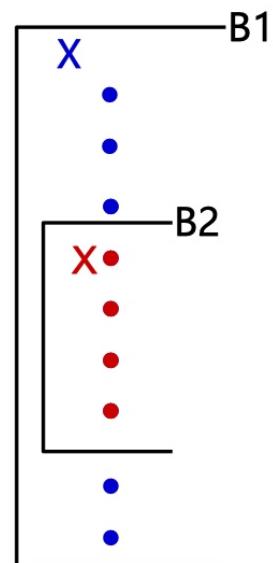
- ▶ **词法规则**: 单词符号的形成规则
  - ▶ 单词符号是语言中具有独立意义的最基本结构
  - ▶ 一般包括: 常数、标识符、基本字、算符、界符等
  - ▶ 描述工具: 有限自动机
- ▶ **语法规则**: 语法单位的形成规则
  - ▶ 语法单位通常包括: 表达式、语句、分程序、过程、函数、程序等;
  - ▶ 描述工具: 上下文无关文法

- 语义: 一组规则, 定义一个程序的意义
  - 例如“关于函数调用时参数传递方法的描述”属于语义定义

## 1.2 作用域

- 同一个标识符在不同过程中代表不同的名字
- 作用域: 一个名字能被使用的区域范围
- 规则: “最近嵌套原则”

- ▶ 一个在子程序B1中说明的名字X只在B1中有效 (局部于B1)
- ▶ 如果B2是B1的一个内层子程序且B2中对标识符X没有新的说明, 则原来的名字X在B2中仍然有效
- ▶ 如果B2对X重新作了说明, 那么, B2对X的任何引用都是指重新说明过的这个X



## 1.3 标识符 v.s. 名字

标识符是语法概念, 名字是语义概念。

「标识符」

- 以字母开头的, 由字母数字组成的字符串

## 「名字」

- 含义：标识程序中的对象
- 意义和属性：
  - 值：单元中的内容
  - 属性：类型和作用域
- 说明方式
  - 说明语句明确规定
    - int score
  - 隐含说明
    - FORTRAN 以 I,J,K,...,N 为首的名字代表整型，否则为实型
  - 动态确定
    - 走到哪里，是什么，算什么
- 名字的绑定
  - 名字的绑定是指将标识符与所代表的程序数据或代码进行关联
  - 静态绑定：发生在编译过程中，如变量声明、类型定义、函数定义
  - 动态绑定：发生在运行过程中，如多态、虚函数

## 「二者区别」

- 标识符是语法概念
- 名字有确切的意义和属性

## 1.4 左值与右值

赋值语句：A := B

- 名字的左值：该名字代表的存储单元的地址
- 名字的右值：该名字代表的存储单元的内容

## 「简单判断」

- 出现在赋值号左边的值必须具有左值，出现在赋值号右边的值则必须具有右值。

## 二、语法描述

### 2.1 基本概念

- **字母表：**一个有穷字符集，记为  $\Sigma$
- **字符：**字母表中每个元素
- **字 / 字符串：**  $\Sigma$  上的字（也叫字符串）是指由  $\Sigma$  中的字符所构成的一个有穷序列
- **空字：**不包含任何字符的序列，记做  $\epsilon$ 
  - 空字是字符串，不是字符
- **字的全体：**  $\Sigma^*$  表示  $\Sigma$  上的所有字的全体，包含空字

例如: 设  $\Sigma = \{a, b\}$ , 则

$$\Sigma^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$$

- 子集连接 (积) :

- $\Sigma^*$  的子集  $U$  和  $V$  的连接 (积) 定义为

$$UV = \{\alpha\beta | \alpha \in U \& \beta \in V\}$$

$$U = \{a, aa\}$$

$$V = \{b, bb\}$$

$$UV = \{ab, abb, aab, aabb\}$$

- n次积

- $V$  自身的n次积记为  $V^n$

- $V^0 = \{\epsilon\}$

- $V^*$  是  $V$  的闭包:  $V^* = V^0 \cup V^1 \cup V^2 \cup \dots$

- $V^+$  是  $V$  的正规闭包:  $V^+ = VV^*$

- 「区别」

- $V^*$  中始终有空字, 但如果  $V$  中原来没有空字, 则  $V^+$  中不会有空字

## 2.2 上下文无关文法

「上下文无关文法  $G$  的定义 – 四元组」

$$G = (V_T, V_N, S, P)$$

- $V_T$ : 终结符 (Terminal) 集合, 非空
- $V_N$ : 非终结符 (Noterminal) 集合, 非空, 且  $V_T \cap V_N = \emptyset$
- $S$ : 文法的开始符号,  $S \in V_N$ 
  - $S$  是特殊非终结符, 表示所定义的语言最终感兴趣的语法单位, 如英语描述中的“句子”, 程序语言描述的“程序”
- $P$ : 产生式集合 (有限), 每个产生式形式如下
  - $P \rightarrow \alpha, P \in V_N, \alpha \in (V_T \cup V_N)^*$
  - $P \rightarrow \alpha$  表示 “ $P$  定义为  $\alpha$ ”
- 开始符  $S$  至少必须在某个产生式的左部出现一次

- ▶ 约定
- $$\begin{array}{l} P \rightarrow \alpha_1 \\ P \rightarrow \alpha_2 \\ \dots \\ P \rightarrow \alpha_n \end{array}$$
- 可缩写为  $\xrightarrow{\quad} P \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$
- ▶ 其中，“|”读成“或”，称 $\alpha_i$ 为P的一个候选式  
 ▶ 表示一个文法时，通常只给出开始符号和产生式

## 2.3 推导

### 2.3.1 基本概念

- $\Rightarrow$ : 直接推出，只能对一个非终结符推导一次
- $\rightarrow$ : 被定义为

▶ 定义：称 $\alpha A \beta$ 直接推出 $\alpha \gamma \beta$ ，即

$$\alpha A \beta \xrightarrow{} \alpha \gamma \beta$$

仅当 $A \rightarrow \gamma$ 是一个产生式，且 $\alpha, \beta \in (V_T \cup V_N)^*$ 。

▶ 如果 $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n$ ，则我们称这个序列是从 $\alpha_1$ 到 $\alpha_n$ 的一个推导。若存在一个从 $\alpha_1$ 到 $\alpha_n$ 的推导，则称 $\alpha_1$ 可以推出 $\alpha_n$ 。

「\*推出 & +推出」

$\alpha_1 \xrightarrow{*} \alpha_n$  从 $\alpha_1$ 出发，经过0步或若干步推出 $\alpha_n$   
 $\alpha_1 \xrightarrow{+} \alpha_n$  从 $\alpha_1$ 出发，经过1步或若干步推出 $\alpha_n$   
 $\alpha \xrightarrow{*} \beta$  即  $\alpha = \beta$  或  $\alpha \xrightarrow{+} \beta$

「概念辨析 – 句型 | 句子 | 语言」

- ▶ 如果 $S \xrightarrow{*} \alpha$ ，则称 $\alpha$ 是一个句型。
- ▶ 仅含终结符号的句型是一个句子。
- ▶ 文法G所产生的句子的全体是一个语言，记为  $L(G)$ :

$$L(G) = \left\{ \alpha \mid S \xrightarrow{+} \alpha, \alpha \in V_T^* \right\}$$

## 「句型、句子推导练习」

- 文法 => 句子

► 请证明  $(i^*i+i)$  是文法

$G(E) : E \rightarrow i \mid E+E \mid E^*E \mid (E)$   
的一个句子。

证明：

$$\begin{aligned} E &\Rightarrow (E) \\ &\Rightarrow (E+E) \\ &\Rightarrow (E^*E+E) && (i^*i+i) \text{ 是文法 } G \text{ 的句子} \\ &\Rightarrow (i^*E+E) && E, (E), (E^*E+E), \dots, (i^*i+i) \text{ 是句型。} \\ &\Rightarrow (i^*i+E) \\ &\Rightarrow (i^*i+i) \end{aligned}$$

- 句子 => 文法

◦ 此类题目稍微难一些，需要用递归思想来解决，优先考虑最简结构

► 请给出产生语言为  $\{a^m b^n \mid 1 \leq n \leq m \leq 2n\}$  的文法

$G_4(S) :$

$$\begin{aligned} S &\rightarrow ab \mid aab \\ S &\rightarrow aSb \mid aaSb \end{aligned}$$

此类题目要使用递归的思想来解决，即先找到  $S$  能推出的最简单的句子，然后再进行扩展，即可发现规律，找到合适的文法

## 2.3.2 语法树

### 「最左/右推导」

► 从一个句型到另一个句型的推导往往不唯一

$$E+E \Rightarrow i+E \Rightarrow i+i$$

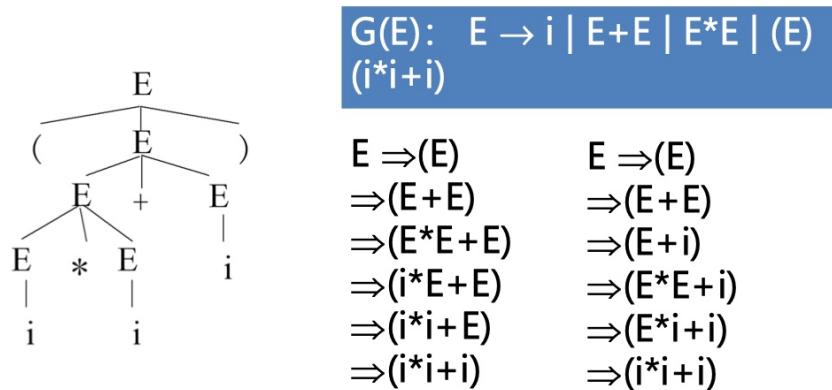
$$E+E \Rightarrow E+i \Rightarrow i+i$$

► **最左推导**：任何一步  $\alpha \Rightarrow \beta$  都是对  $\alpha$  中的最左非终结符进行替换

► **最右推导**：任何一步  $\alpha \Rightarrow \beta$  都是对  $\alpha$  中的最右非终结符进行替换

## 「语法树」

- ▶ 用一张图表示一个句型的推导,称为语法树
- ▶ 一棵语法树是不同推导过程的共性抽象

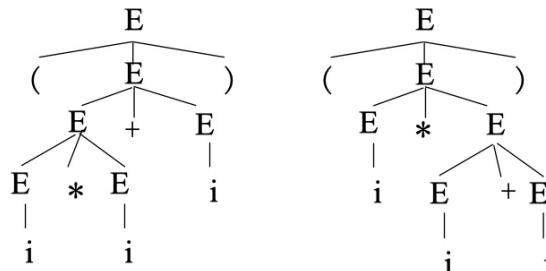


### 2.3.3 二义性

「二义性举例」

- ▶ 一个句型是否只对应唯一一棵语法树?
- ▶  $(i^*i+i)$

$G(E): E \rightarrow i \mid E+E \mid E^*E \mid (E)$   
是二义的(ambiguous)



「文法 / 语言二义性」

- 文法二义性: 文法存在某个句子对应两棵不同语法树
  - 文法二义性问题是不可判定问题, 不存在一个算法, 能在有限步骤中, 确切地判定一个文法是否二义
  - 但仍然存在很多充分条件可以判定一个文法是非二义的
    - 例如一个文法如果属于 LR 文法, 则一定不是二义文法
- 语言二义性: 存在一个能推导出该语言的无二义文法

## 2.4 形式语言

### 2.4.1 概述

- 乔姆斯基在1956年建立形式语言体系, 将文法分为四种类型: 0、1、2、3型

- 四种类型唯一区别在于产生式

►  $G = (V_T, V_N, S, P)$

- $V_T$ : 终结符(Terminal)集合(非空)
- $V_N$ : 非终结符(Noterminal)集合(非空), 且  $V_T \cap V_N = \emptyset$
- $S$ : 文法的开始符号,  $S \in V_N$
- $P$ : 产生式集合(有限)

- 0型 (短语文法, 图灵机)

► 产生式形如:  $\alpha \rightarrow \beta$

► 其中:  $\alpha \in (V_T \cup V_N)^*$  且至少含有一个非终结符;  $\beta \in (V_T \cup V_N)^*$

- 1型 (上下文有关文法, 线性界限自动机)

► 产生式形如:  $\alpha \rightarrow \beta$

► 其中:  $|\alpha| \leq |\beta|$ , 仅  $S \rightarrow \epsilon$  例外

- 2型 (上下文无关文法, 非确定下推自动机)

► 产生式形如:  $A \rightarrow \beta$

► 其中:  $A \in V_N$ ;  $\beta \in (V_T \cup V_N)^*$

- 3型 (正规文法, 有限自动机)

◦

► 产生式形如:  $A \rightarrow \alpha B$  或  $A \rightarrow \alpha$

► 其中:  $\alpha \in V_T^*$ ;  $A, B \in V_N$

**右线性文法**

► 产生式形如:  $A \rightarrow B\alpha$  或  $A \rightarrow \alpha$

► 其中:  $\alpha \in V_T^*$ ;  $A, B \in V_N$

**左线性文法**

## ◦ 正规式、正规集

设 $\Sigma$ 是有穷字母表，并定义辅助字母表 $\Sigma'=\{\Phi, \epsilon, |, ., ^*, (, )\}$

1.  $\epsilon, \Phi$ 都是 $\Sigma$ 上的正规式，它们所表示的正规集为 $\{\epsilon\}, \Phi$ ；
2. 任何 $a$ 是一个正规式，若 $a \in \Sigma$ ，它所表示的正规集为 $\{a\}$ ；
3. 如果 $R_1$ 和 $R_2$ 是正规式，它们表示的正规集分别为 $L_1$ 和 $L_2$ ，则 $R_1|R_2, R_1 \cdot R_2, R_1^*, (R_1)$ 也是正规式，并且它们所表示的正规集分别为 $L_1 \cup L_2; L_1 \cdot L_2; L_1^*; L_1$ ；
4. 仅有有限次使用上述三步骤而定义的表达式才是 $\Sigma$ 上的正规式，仅有这些正规式表示的字集才是 $\Sigma$ 上的正规集。

注意：不要混淆 $\Phi$ 和 $\epsilon$ ，正规表达式 $\epsilon$ 描述的语言只含一个空字符串 $\epsilon$ ，而 $\Phi$ 表示的语言不含有任何字符串。

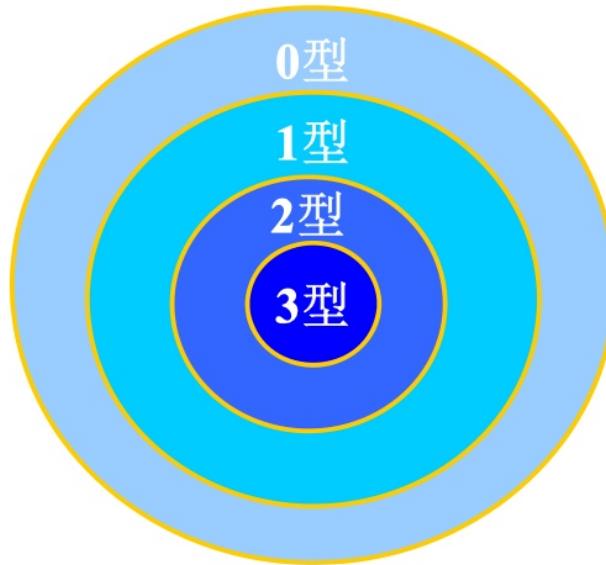
程序设计语言的单词都能用正规式来定义。若两个正规式 $e_1, e_2$ 表示的正规集相同，则称它们等价。记作： $e_1 = e_2$

例：令 $\Sigma = \{a, b\}$ ，则 $\Sigma$ 上的正规式和相应正规集为

正规式	正规集
a	{a}
a b	{a,b}
ab	{ab}
(a b)(a b)	{aa, ab, ba, bb}
$a^*$	{ $\epsilon, a, aa, \dots$ 任意个a的串}
$(a b)^*$	{ $\epsilon, a, aa, \dots$ 任意个a的串} { $\epsilon, a, b, aa, ab, bb, \dots$ 所有由 a 和 b 组成的串}
$(a b)^* (aa bb)(a b)^*$	$\Sigma$ 上所有含有两个相继的a或两个相继的b组成的串}

## 2.4.2 文法对比

- 四种类型文法描述能力比较



- 上下文无关文法 v.s. 正规文法

►  $L_5 = \{a^n b^n | n \geq 1\}$  不能由正规文法产生，但可由上下文无关文法产生

$G_5(S)$ :

$$S \rightarrow aSb \mid ab$$

- 上下文无关文法 v.s. 上下文有关文法

►  $L_6 = \{a^n b^n c^n | n \geq 1\}$  不能由上下文无关文法产生，但可由上下文有关文法产生

$$G_6(S): \begin{aligned} S &\rightarrow aSBC \mid aBC \\ CB &\rightarrow BC \\ aB &\rightarrow ab \\ bB &\rightarrow bb \\ bC &\rightarrow bc \\ cC &\rightarrow cc \end{aligned}$$

- 上下文无关文法的局限 – 权衡思想

►  $L_7 = \{\alpha\alpha | \alpha \in \{a,b\}^*\}$  不能由上下文无关文法产生，甚至连上下文有关文法也不能产生，只能由0型文法产生

使用语义分析解决

- 标识符引用
- 过程调用过程中，“形-实参数的对应性”（如个数，顺序和类型一致性）
- 对于现今程序设计语言，在编译程序中，仍然采用上下文无关文法来描述其语言结构

## 词法分析

### 一、状态转换图

#### 1.1 词法分析器概述

##### 1.1.1 功能

- 功能
  - 输入源程序、输出单词符号
- 单词符号种类
  - 基本字：如begin、repeat、for、...
  - 标识符：用来表示各种名字，如变量名、数组名和过程名
  - 常数：各种类型的常数

- 运算符：+、-、\*、/、...
- 界符：逗号、分号、括号和空白

### 1.1.2 输出

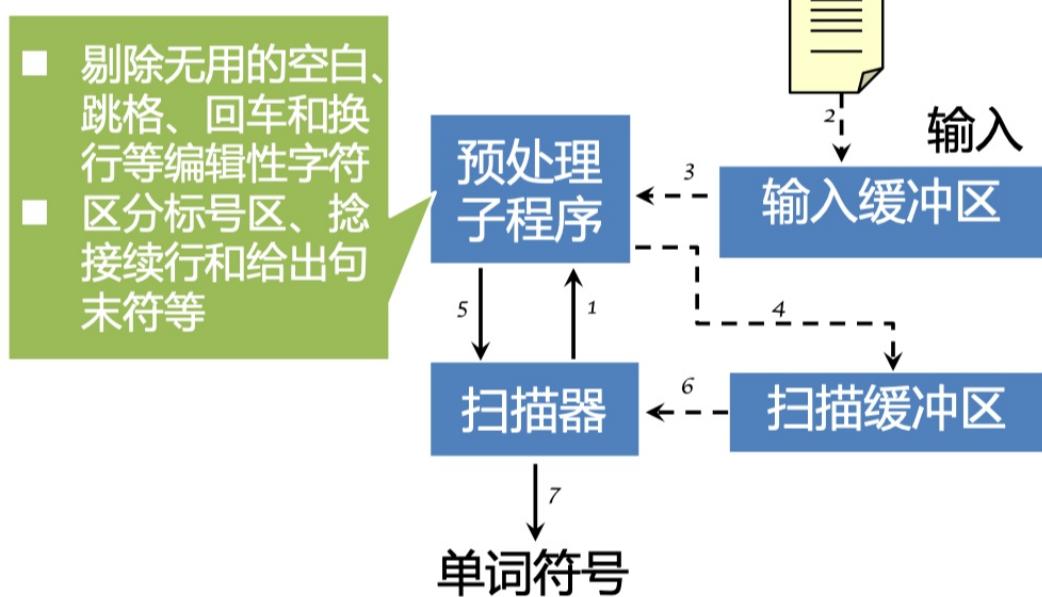
- 输出的单词符号的表示形式
    - (单词种类编号， 单词自身值)
  - 单词种别通常用整数编码表示
    - 基本字、运算符和界符都是一符一种
    - 如果一个种别有多个单词符号，则对于每个单词符号，给出种别编码和自身的值
    -
- ▶ **标识符**单列一种；标识符自身的值表示成按机器字节划分的内部码
- ▶ **常数**按类型分种；常数的值则表示成标准的二进制形式

「举例 – `if (5 = m) goto 100`」

```
if      (34, -)
(
整常数  (20, '5' 的二进制)
=
标识符  (26, 'm')
)
goto   (30, -)
标号   (19, '100' 的二进制)
```

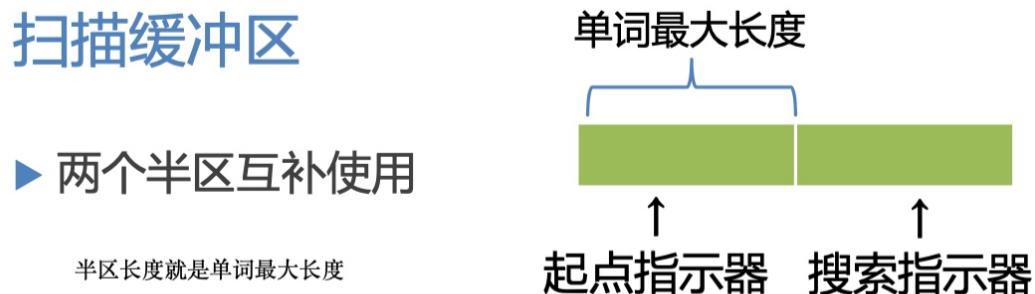
### 1.1.3 词法分析器结构

- 词法分析器的处理流程

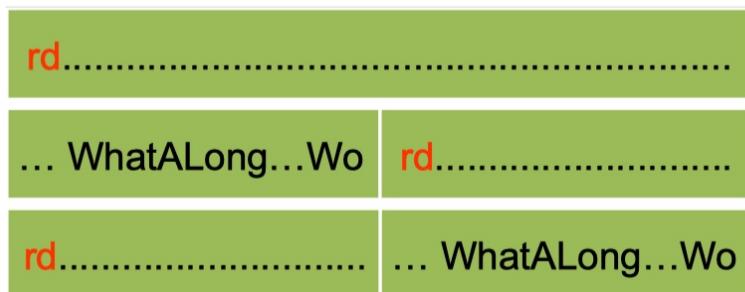


- 扫描缓冲区结构

- 两个半区互补使用，且单词最大长度不能超过半区长度



WhatALong...Word



- 超前搜索：识别单词符号

- 需要的情况

► 例如

DO99K=1, 10

DO 99 K = 1, 10

DO99K=1.10

IF (5.EQ.M) GOTO 55

IF (5.EQ.M) GOTO55

IF (5)=55

► 需要超前搜索才能确定哪些是基本字

- 不需要的情况

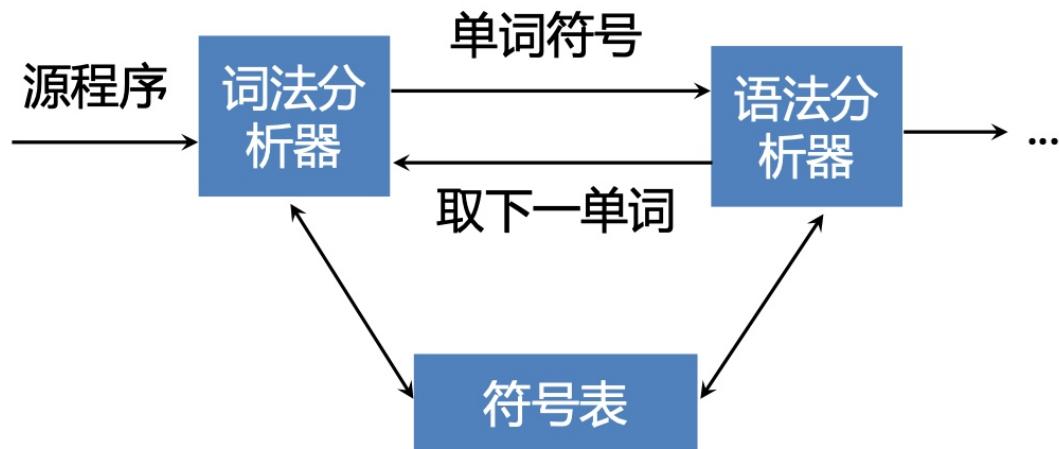
- 所有基本字都是保留字;用户不能用它们作自己的标识符
- 基本字作为特殊的标识符来处理，使用保留字表
- 如果基本字、标识符和常数(或标号)之间没有确定的运算符或界符作间隔，则必须使用一个空白符作间隔

DO99K=1, 10

要写成 DO 99 K=1, 10

#### 1.1.4 编译器中地位

- 词法分析器在编译器中地位



#### 1.2 状态转换图

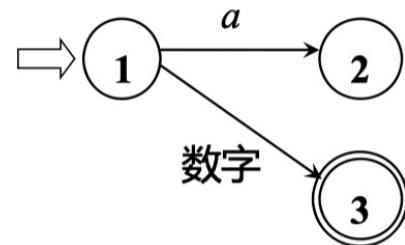
### 1.2.1 基础定义

- 结点、箭弧、状态

◦

▶ 状态转换图是一张有限方向图

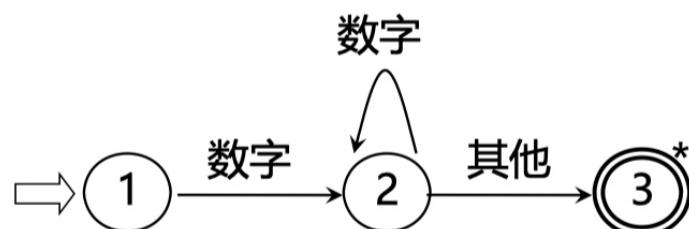
- ▶ 结点代表状态，用圆圈表示
- ▶ 状态之间用箭弧连结，箭弧上的标记(字符)代表射出结状态下可能出现的输入字符或字符类
- ▶ 一张转换图只包含有限个状态，其中有一个为初态，至少要有一个终态



- 如果终结状态上有\*，则表示识别之后需要退还一个识别的字符
- $\alpha$  被识别

◦

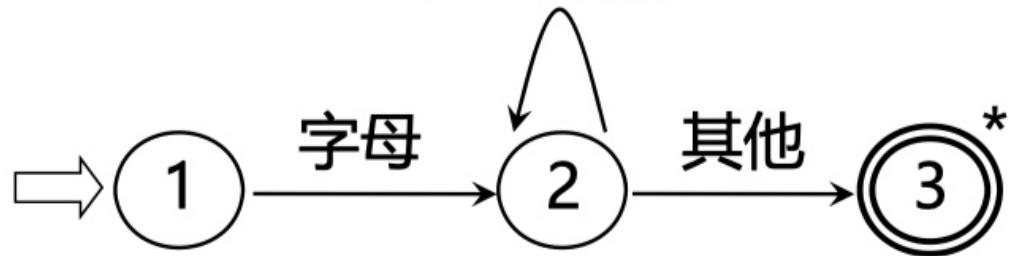
- ▶ 状态转换图可用于识别(或接受)一定的字符串
- ▶ 若存在一条从初态到某一终态的道路，且这条路上所有弧上的标记符连接成的字等于 $\alpha$ ，则称 $\alpha$ 被该状态转换图所识别(接受)



识别整常数的状态转换图

◦

# 字母或数字

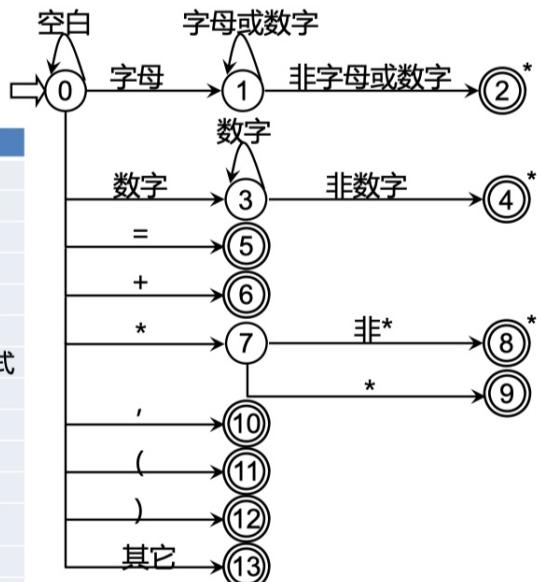


## 1.2.2 设计示例

「示例」

### ▶ 设计状态转换图

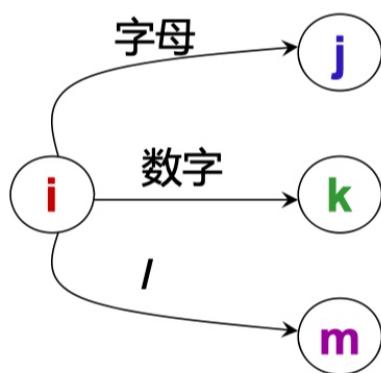
单词符号	种别编码	助忆符	内码值
DIM	1	\$DIM	-
IF	2	\$IF	-
DO	3	\$DO	-
STOP	4	\$STOP	-
END	5	\$END	-
标识符 常数(数)	6	\$ID	内部字符串
	7	\$INT	标准二进制形式
=	8	\$ASSIGN	-
+	9	\$PLUS	-
*	10	\$STAR	-
**	11	\$POWER	-
,	12	\$COMMA	-
(	13	\$LPAR	-
)	14	\$RPAR	-



「转换图 => 代码」

### ▶ 不含回路的分叉结点

▶ 可用一个CASE语句或一组IF-THEN-ELSE语句实现



```

GetChar( );
if (IsLetter( ))
{...状态j的对应程序段...;}
else if (IsDigit( ))
{...状态k的对应程序段...;}
else if (ch=='/')
{...状态m的对应程序段...;}
else
{...错误处理...;}
  
```

## ▶ 含回路的状态结点

- ▶ 对应一段由**WHILE**结构和**IF**语句构成的程序

字母或数字



```
GetChar();
while (IsLetter() or IsDigit())
    GetChar();
...状态j的对应程序段...
```

## ▶ 终态结点

- ▶ 表示识别出某种单词符号，对应**返回**语句



```
RETURN (C, VAL)
```

其中，C为单词种别，VAL为单词自身值

「代码实现」

```
int code, value;
strToken := ""; /* 置 strToken 为空串 */
GetChar(); GetBC();
if (IsLetter())
begin
    while (IsLetter() or IsDigit())
begin
    Concat(); /* 把ch中的字符连接到 strToken */
    GetChar();
end
Retract(); /* 子程序，把搜索指针回调一个字符位置 */
code := Reserve();
```

```

if (code = 0)
begin
    value := InsertId(strToken);
    return ($ID, value);
end
else
    return (code, -);
end
else if (IsDigit())
begin
    while(IsDigit())
begin
    Concat(); GetChar();
end
Retract();
value := InsertConst(strToken);
return ($INT, value);
end
else if (ch = '=') return ($ASSIGN, -);
else if (ch = '+') return ($PLUS, -);
else if (ch = '*')
begin
    GetChar();
    if (ch = '*') return ($POWER, -);
    Retract(); return ($STAR, -);
end
else if (ch = ',') return ($COMMA, -);
else if (ch = '(') return ($LPAR, -);
else if (ch = ')') return ($RPAR, -);
else ProcError(); /* 错误处理 */

```

### 1.2.3 状态图一般化代码

- 变量 curState 保存现有的状态
- 用二维数组表示状态图, stateTrans[state][ch]

```

curState = 初态
GetChar();
while(stateTrans[curState][ch] 有定义) {
    // 存在后继状态, 读入、拼接
    Concat();
    // 转换入下一状态, 读入下一字符
    curState = stateTrans[curState][ch];
    if curState 是终态 then 返回 strToken 中的单词
    GetChar();
}

```

## 二、词法规则的形式化

### 2.1 正规式与正规集

#### 2.1.1 定义

- 二者关系
    - 正规集可以用正规式表示
    - 正规式是表示正规集的一种方法
    - 一个字集合是正规集当且仅当它能用正规式表示
  - 递归定义
    - $\varepsilon$  和  $\emptyset$  都是  $\Sigma$  上的正规式，它们所表示的正规集为  $\{\varepsilon\}$  和  $\emptyset$ 
      - 其中  $\{\varepsilon\}$  表示集合里有一个元素，即空字。长度为 0，不包含任何字符。
      - $\varepsilon$  是字、正规式
      - $\emptyset$  是集合、正规式
    - 任何  $a \in \Sigma$ ,  $a$  是  $\Sigma$  上的正规式，它所表示的正规集为  $\{a\}$ 
      - $a$  是字符、字、正规式
    -
- 3) 假定  $e_1$  和  $e_2$  都是  $\Sigma$  上的正规式，它们所表示的正规集为  $L(e_1)$  和  $L(e_2)$ ，则
- i)  $(e_1 | e_2)$  为正规式，它所表示的正规集为  $L(e_1) \cup L(e_2)$
  - ii)  $(e_1 \cdot e_2)$  为正规式，它所表示的正规集为  $L(e_1)L(e_2)$
  - iii)  $(e_1)^*$  为正规式，它所表示的正规集为  $(L(e_1))^*$
- 仅由 **有限次** 使用上述三步骤而定义的表达式才是  $\Sigma$  上的正规式，仅由这些正规式表示的字集才是  $\Sigma$  上的正规集。

#### 2.1.2 正规式的等价性

► 若两个正规式所表示的正规集相同，则称这两个正规式等价。如

$$b(ab)^* = (ba)^*b$$

$L(b(ab)^*)$	$L((ba)^*b)$
$= L(b)L((ab)^*)$	$= L((ba)^*)L(b)$
$= L(b)(L(ab))^*$	$= (L(ba))^*L(b)$
$= L(b)(L(a)L(b))^*$	$= (L(b)L(a))^*L(b)$
$= \{b\} \{ab\}^*$	$= \{ba\}^* \{b\}$
$= \{b\} \{\epsilon, ab, abab, ababab, \dots\}$	$= \{\epsilon, ba, baba, bababa, \dots\} \{b\}$
$= \{b, bab, babab, bababab, \dots\}$	$= \{b, bab, babab, bababab, \dots\}$

$$\therefore L(b(ab)^*) = L((ba)^*b)$$

$$\therefore b(ab)^* = (ba)^*b$$

### 2.1.3 正规式的性质

► 对正规式，下列等价成立

- $e_1|e_2 = e_2|e_1$  交换律
- $e_1 | (e_2|e_3) = (e_1|e_2)|e_3$  结合律
- $e_1(e_2e_3) = (e_1e_2)e_3$  结合律
- $e_1(e_2|e_3) = e_1e_2|e_1e_3$  分配律
- $(e_2|e_3)e_1 = e_2e_1|e_3e_1$  分配律
- $e\epsilon = \epsilon e = e$        $e_1e_2 < e_2e_1$

$L(e_1 e_2)$
$= L(e_1) \cup L(e_2)$
$= L(e_2) \cup L(e_1)$
$= L(e_2 e_1)$

## 2.2 DFA

### 2.2.1 定义

确定有限自动机 M (DFA – Deterministic Finite Automata)

- M 是一个五元式， $M = (S, \Sigma, f, S_0, F)$

1.  $S$ : 有穷状态集
2.  $\Sigma$ : 输入字母表(有穷)
3.  $f$ : 状态转换函数, 为  $S \times \Sigma \rightarrow S$  的单值部分映射,  $f(s, a) = s'$  表示: 当现行状态为  $s$ , 输入字符为  $a$  时, 将状态转换到下一状态  $s'$ ,  $s'$  称为  $s$  的一个后继状态
4.  $S_0 \in S$  是唯一的一个初态
5.  $F \subseteq S$  : 终态集(可空)

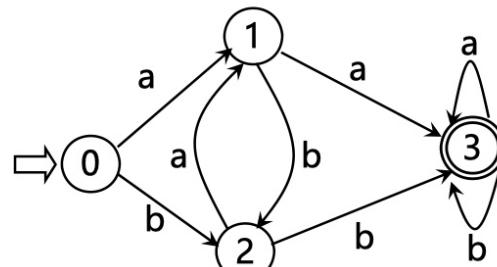
## 2.2.2 举例

- DFA 构建

► DFA  $M = (\{0, 1, 2, 3\}, \{a, b\}, f, 0, \{3\})$ , 其中  $f$  定义如下:

$$\begin{array}{ll} f(0, a) = 1 & f(0, b) = 2 \\ f(1, a) = 3 & f(1, b) = 2 \\ f(2, a) = 1 & f(2, b) = 3 \\ f(3, a) = 3 & f(3, b) = 3 \end{array}$$

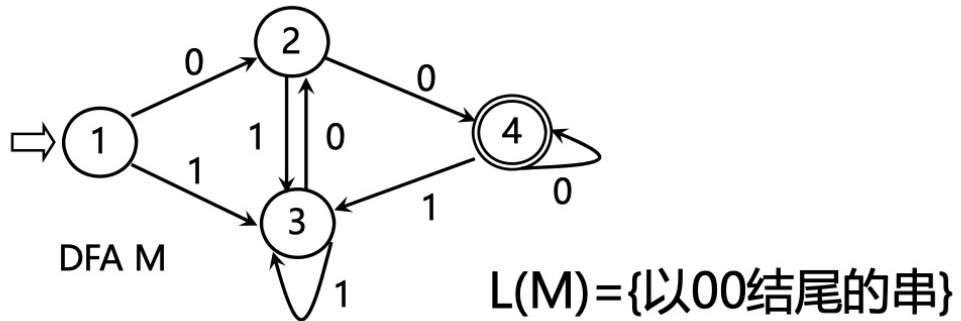
	a	b
0	1	2
1	3	2
2	1	3
3	3	3



状态转换图

- 字被  $M$  所识别, 即存在一条从初态到终态的道路
  - DFA  $M$  所识别的字的全体记为  $L(M)$

- ▶ 对于 $\Sigma^*$ 中的任何字 $\alpha$ , 若存在一条从初态到某一终态的道路, 且这条路上所有弧上的标记符连接成的字等于 $\alpha$ , 则称 $\alpha$ 为DFA M所识别(接收)
- ▶ DFA M所识别的字的全体记为 $L(M)$



### 2.2.3 DFA 程序实现

DFA 可以使用状态转换图的一般代码来进行程序实现。

```

curState = 初态
GetChar();
while(stateTrans[curState][ch] 有定义) {
    // 存在后继状态, 读入、拼接
    Concat();
    // 转换入下一状态, 读入下一字符
    curState = stateTrans[curState][ch];
    if curState 是终态 then 返回 strToken 中的单词
    GetChar();
}

```

## 2.3 NFA

### 2.3.1 定义

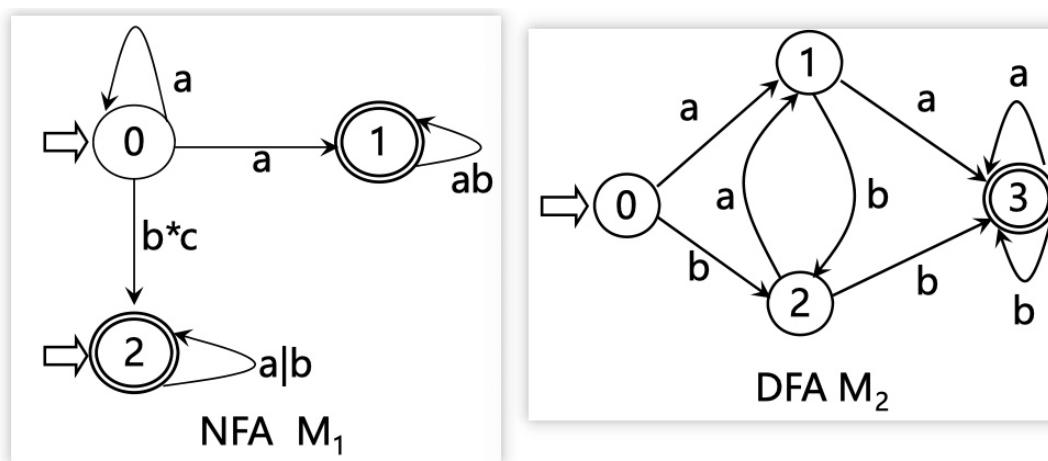
非确定有限自动机 M (NFA – Nondeterministic Finite Automata)

- M 是一个五元式,  $M = (S, \Sigma, f, S_0, F)$ 
  - 与 DFA 的最大区别在于三点
    - 初态可以有多个
    - 转移边上可以是正规式, 而 DFA 上是字符
    - 对于一个节点来说, 一个正规式可以映射多个节点
    -

	NFA	DFA
初始状态	不唯一	唯一
弧上的标记	字(单字符字、 $\epsilon$ )	字符
转换关系	非确定	确定

1.  $S$ : 有穷状态集
2.  $\Sigma$  : 输入字母表(有穷)
3.  $f$ : 状态转换函数, 为  $S \times \Sigma^* \rightarrow 2^S$  的部分映射
4.  $S_0 \subseteq S$  是非空的初态集
5.  $F \subseteq S$  : 终态集(可空)

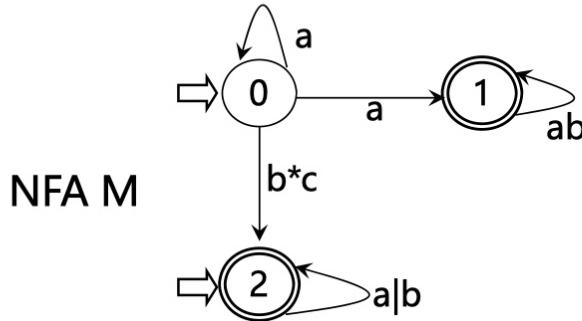
- DFA 与 NFA 举例



- $a$  被 NFA  $M$  所识别

◦

- ▶ 对于 $\Sigma^*$ 中的任何字 $\alpha$ , 若存在一条从初态到某一终态的道路, 且这条路上所有弧上的标记字连接成的字等于 $\alpha$ (忽略那些标记为 $\varepsilon$ 的弧), 则称 $\alpha$ 为NFA M所识别(接收)
- ▶ NFA M所识别的字的全体记为 $L(M)$



### 2.3.2 DFA 与 NFA

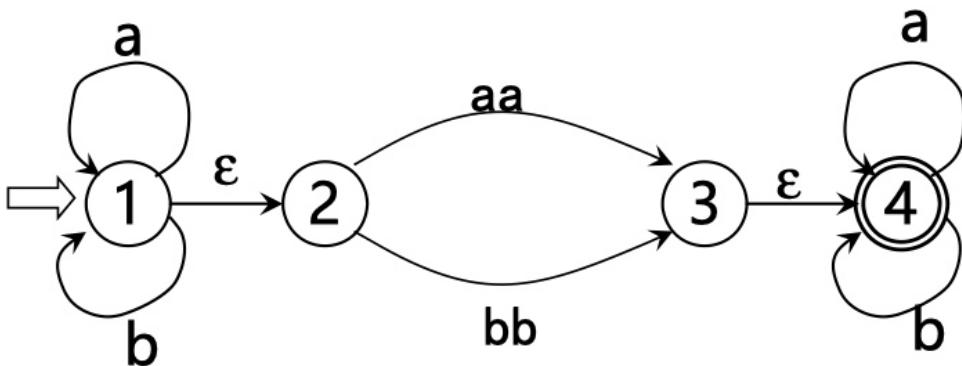
- DFA、NFA 等价理论
  - DFA 与 NFA 识别能力相同
    - ▶ 定义: 对于任何两个有限自动机M和M', 如果  $L(M)=L(M')$ , 则称M与M'等价
    - ▶ 自动机理论中一个重要的结论: 判定两个自动机等价性的算法是存在的
    - ▶ 对于每个NFA M存在一个DFA M', 使得  $L(M)=L(M')$
    - ▶ **DFA与NFA识别能力相同!**
- DFA 与 NFA 优点
  - DFA: 易于程序实现
  - NFA: 设计更容易, 易于人工设计

## 三、有限自动机的等价性

证明 DFA 与 NFA 等价, 其证明过程其实就是算法构造过程。通过构造将 NFA 转化为 DFA, 并且其能识别的字的全体  $L(M)$  一致, 因此二者等价。

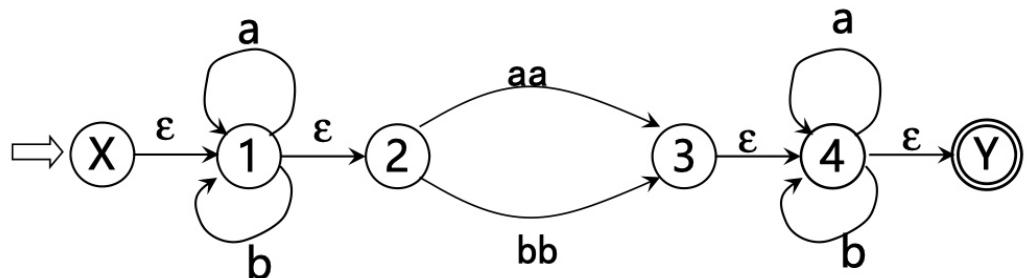
### 3.1 NFA 改造

- 初始 NFA

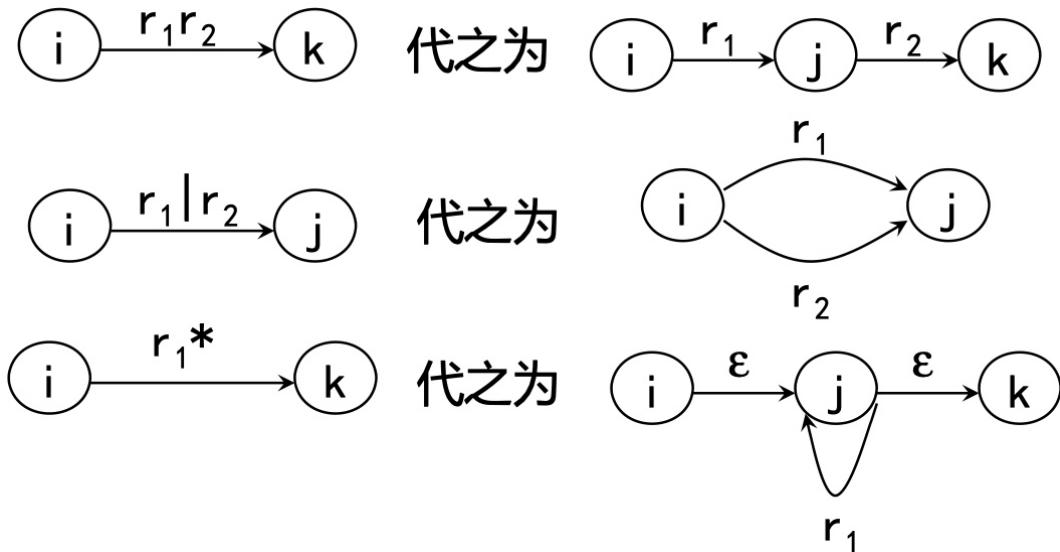


- 引入初态结点 X 和终态结点 Y

- 解决初始状态唯一性



- 简化弧上的标记



## 3.2 NFA 转换

### 3.2.1 $\epsilon$ -闭包

- 对于 I 来说，其  $\epsilon$  闭包就是往 I 中加入经过任意条  $\epsilon$  弧可以到达的状态

- ▶ NFA确定化--子集法 (**解决 $\epsilon$ 弧和转换关系**)
- ▶ 设 $I$ 是的状态集的一个子集，定义 $I$ 的 $\epsilon$ -闭包 $\epsilon$ -closure( $I$ )为：

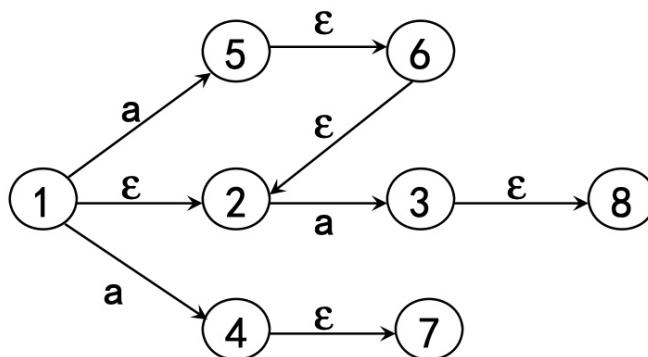
- ▶ 若 $s \in I$ , 则 $s \in \epsilon\text{-closure}(I)$ ;
- ▶ 若 $s \in I$ , 则从 $s$ 出发经过任意条 $\epsilon$ 弧而能到达的任何状态 $s'$ 都属于 $\epsilon\text{-closure}(I)$

即,

$$\epsilon\text{-closure}(I) = I \cup \{s' \mid \text{从某个 } s \in I \text{ 出发经过任意条 } \epsilon \text{ 弧能到达 } s'\}$$

### 3.2.2 $I_a$

- $I_a = \epsilon\text{-closure}(J)$
- $J$  为  $I$  中节点出发经过一条  $a$  弧而到达的状态集合



$I_a = \epsilon\text{-closure}(J)$   
其中,  $J$  为  $I$  中的某  
个状态出发经过  
一条  $a$  弧而到达的  
状态集合。

$$\epsilon\text{-closure}(\{1\}) = \{1, 2\} = I_a ?$$

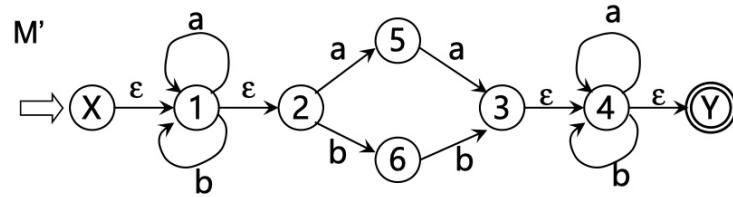
$$J = \{5, 4, 3\}$$

$$I_a = \epsilon\text{-closure}(J) = \epsilon\text{-closure}(\{5, 4, 3\})$$

$$= \{5, 4, 3, 6, 2, 7, 8\}$$

### 3.2.3 子集构造

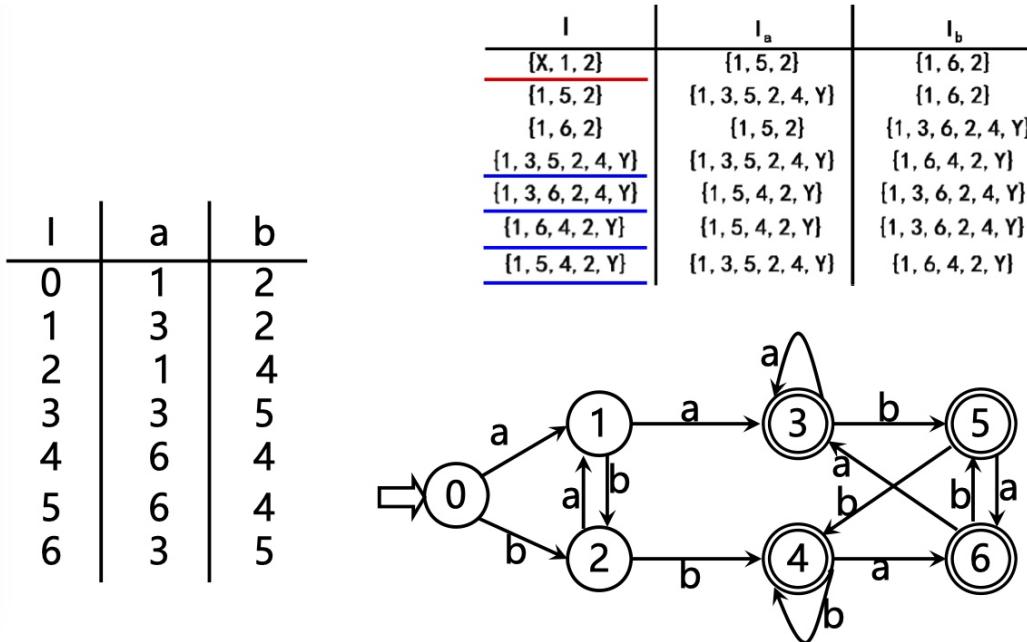
- 初始状态为初始节点 $X$ 的 $\epsilon$ 闭包



$I$	$I_a$	$I_b$
$\epsilon\text{-closure}(\{X\}) = \{X, 1, 2\}$	$\{1, 5, 2\}$	$\{1, 6, 2\}$
$\{1, 5, 2\}$	$\{1, 3, 5, 2, 4, Y\}$	$\{1, 6, 2\}$
$\{1, 6, 2\}$	$\{1, 5, 2\}$	$\{1, 3, 6, 2, 4, Y\}$
$\{1, 3, 5, 2, 4, Y\}$	$\{1, 3, 5, 2, 4, Y\}$	$\{1, 6, 4, 2, Y\}$
$\{1, 3, 6, 2, 4, Y\}$	$\{1, 5, 4, 2, Y\}$	$\{1, 3, 6, 2, 4, Y\}$
$\{1, 6, 4, 2, Y\}$	$\{1, 5, 4, 2, Y\}$	$\{1, 3, 6, 2, 4, Y\}$
$\{1, 5, 4, 2, Y\}$	$\{1, 3, 5, 2, 4, Y\}$	$\{1, 6, 4, 2, Y\}$

- 生成新的 DFA

- 包含初态的状态集合为 DFA初态
- 包含终态的状态集合为 DFA终态



### 3.3 DFA 化简

#### 3.3.1 状态等价性

- 状态等价性

- ▶ DFA的化简(最小化)
  - ▶ 对于给定的DFA  $M$ , 寻找一个状态数比 $M$ 少的DFA  $M'$ , 使得 $L(M)=L(M')$
  - ▶ 状态的等价性
    - ▶ 假设 $s$ 和 $t$ 为 $M$ 的两个状态, 称 $s$ 和 $t$ 等价: 如果从状态 $s$ 出发能读出某个字 $\alpha$ 而停止于终态, 那么同样, 从 $t$ 出发也能读出 $\alpha$ 而停止于终态; 反之亦然
    - ▶ 两个状态不等价, 则称它们是可区别的

- 化简思想

- ▶ 基本思想
  - ▶ 把 $M$ 的状态集划分为一些不相交的子集, 使得任何两个不同子集的状态是可区别的, 而同一子集的任何两个状态是等价的。
  - ▶ 最后, 让每个子集选出一个代表, 同时消去其他状态。

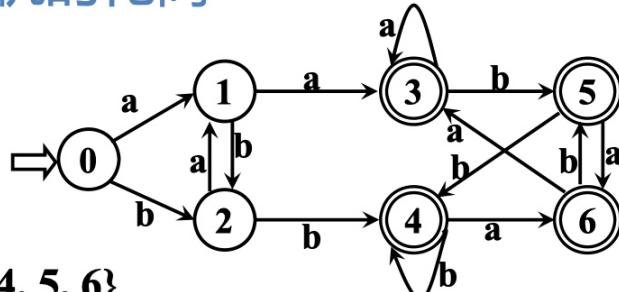
### 3.3.2 化简算法

- 首先, 将  $S$  划分为终态与非终态两个子集
  - 原因是终态可以识别空字, 而非终态不可以
- 其次, 对于每个子集检查能不能进一步划分
  - 
  - ▶ 假定到某个时候,  $\Pi$ 已含 $m$ 个子集, 记为 $\Pi=\{I^{(1)}, I^{(2)}, \dots, I^{(m)}\}$ , 检查 $\Pi$ 中的每个子集看是否能进一步划分:
    - ▶ 对某个 $I^{(i)}$ , 令 $I^{(i)}=\{s_1, s_2, \dots, s_k\}$ , 若存在一个输入字符 $a$ 使得 $I_a^{(i)}$ 不会包含在现行 $\Pi$ 的某个子集 $I^{(j)}$ 中, 则至少应把 $I^{(i)}$ 分为两个部分。
- 重复第二步, 直至子集数不再增长
- 最后, 若  $I$  含有原来的初态, 则为新的初态; 若含有原来的终态, 则为新的终态。

### 3.3.3 示例

- 化简算法

## 确定有限自动机的化简



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

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

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

$$I^{(11)} = \{0, 2\}$$

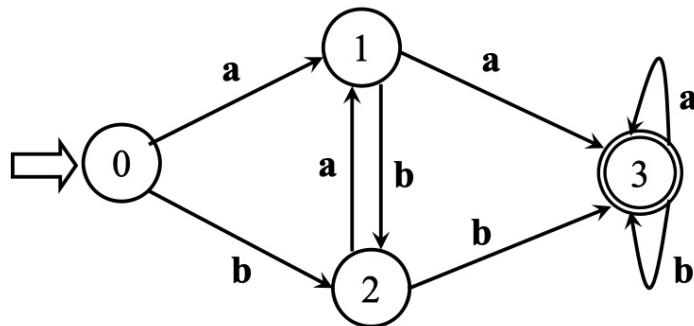
$$I_a^{(11)} = \{1\} \quad I_b^{(11)} = \{2, 4\}$$

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

$$I_a^{(2)} = \{3, 6\} \quad I_b^{(2)} = \{4, 5\}$$

- 生成化简后的 DFA

$\{0\} \{1\} \{2\} \{3, 4, 5, 6\}$



## 四、正规式与有限自动机的等价性

一个正规式与一个有限自动机  $M$  等价，即  $L(r) = L(M)$ 。

### ► FA -> 正规式

► 对任何FA M，都存在一个正规式r，使得  $L(r) = L(M)$ 。

### ► 正规式 -> FA

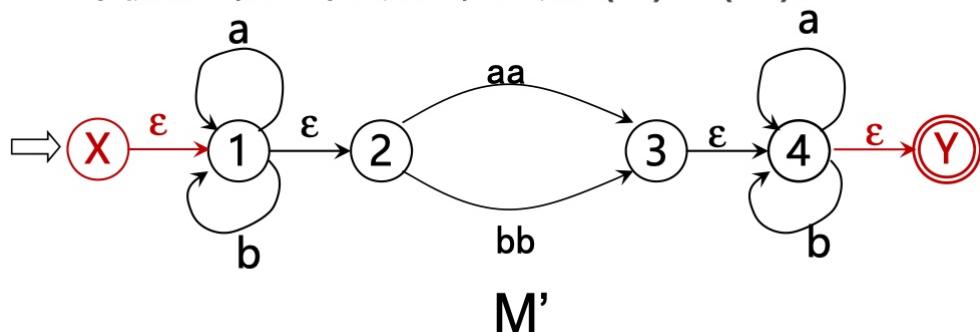
► 对任何正规式r，都存在一个FA M，使得  $L(M) = L(r)$ 。

## 4.1 NFA 构造正规式

证明：对  $\Sigma$  上任一个NFA M，都存在一个  $\Sigma$  上的正规式 r，使得  $L(r) = L(M)$ 。

- 添加初态与终态

► 在M的转换图上加进两个状态X和Y，从X用 $\epsilon$ 弧连接到M的所有初态结点，从M的所有终态结点用 $\epsilon$ 弧连接到Y，从而形成一个新的NFA，记为M'，它只有一个初态X和一个终态Y，显然 $L(M)=L(M')$ 。

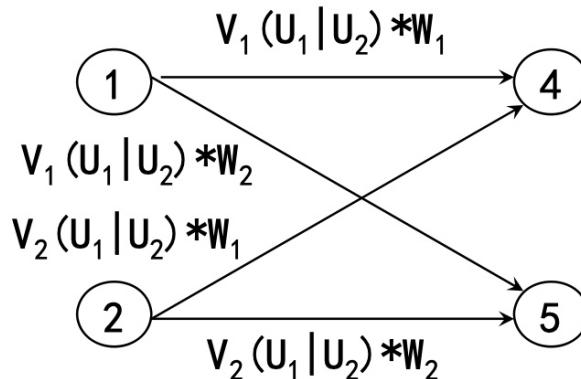
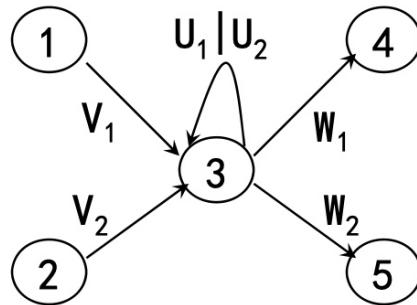
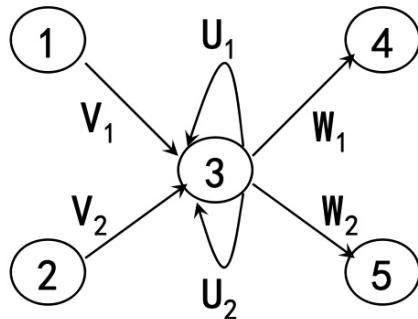


- 反复应用下述三条规则，消去结点



- 举例

## 为NFA构造正规式



## 4.2 为正规式构造NFA

- 定理

- 对于任何正规式  $r$ , 都存在一个 FA  $M$ , 使得  $L(M) = L(r)$ 。

- ▶ 定理: 对任何**正规式** $r$ , 都存在一个**FA M**, 使得 $L(M)=L(r)$ 。
- ▶ 定理: 对于 $\Sigma$ 上的**正规式** $r$ , 都存在一个**NFA M**, 使 **$L(M)=L(r)$** , 并且**M只有一个初态和一个终态, 而且没有从终态出发的箭弧**。

### 4.2.1 归纳法证明

- 归纳法证明

- ▶ 对给定正规式  $r$  中的运算符数目进行归纳
  - ▶ 验证  $r$  中的运算符数目为 0 时, 结论成立。
  - ▶ 假设结论对于运算符数目少于  $k$  ( $k \geq 1$ ) 的正规式成立
  - ▶ 基于该假设, 证明结论对于运算符数目为  $k$  的正规式成立。

• 归纳法第一步

- ▶ 若 $r$ 具有零个运算符，则 $r=\epsilon$ 或 $r=\phi$ 或 $r=a$ ，其中 $a \in \Sigma$ 。
- ▶ 针对上述3类正规式 $r$ ，分别按照下图构造NFA  $M$ ， $M$ 只有一个初态和一个终态，而且没有从终态出发的箭弧，而且使 $L(M)$ 和对应的 $L(r)$ 相等。



• 归纳法第二步

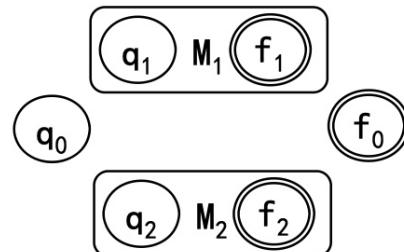
## 为正规式构造NFA

- $r=r_1|r_2$
- $r=r_1.r_2$
- $r=r_1^*$

▶ 假设对于运算符数目少于 $k$ ( $k \geq 1$ )的正规式成立。

▶ 当 $r$ 中含有 $k$ 个运算符时， $r$ 有三种情形：

▶ 情形1： $r=r_1|r_2$ ， $r_1$ 和 $r_2$ 中运算符数目少于 $k$ 。从而，由归纳假设，对 $r_i$ 存在 $M_i = \langle S_i, \Sigma_i, \delta_i, q_{i0}, \{f_i\} \rangle$ ，使得 $L(M_i) = L(r_i)$ ，并且 $M_i$ 没有从终态出发的箭弧 ( $i=1,2$ )。不妨设 $S_1 \cap S_2 = \emptyset$ ，在 $S_1 \cup S_2$ 中加入两个新状态 $q_0, f_0$ 。



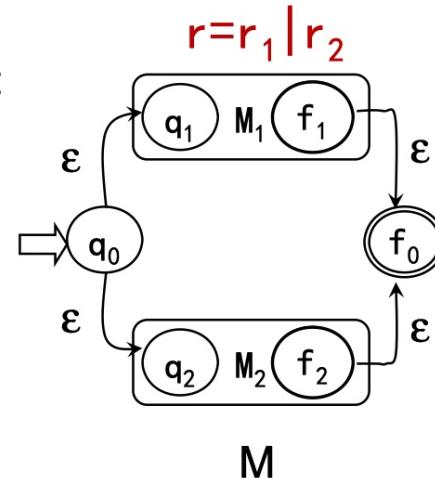
- $r=r_1|r_2$
- $r=r_1.r_2$
- $r=r_1^*$

## 为正规式构造NFA

▶ 令  $M = \langle S_1 \cup S_2 \cup \{q_0, f_0\}, \Sigma_1 \cup \Sigma_2, \delta, q_0, \{f_0\} \rangle$ , 其中  $\delta$  定义如下( $M$  的状态转换如右图):

- $\delta(q_0, \epsilon) = \{q_1, q_2\}$
- $\delta(q, a) = \delta_1(q, a)$ , 当  $q \in S_1 - \{f_1\}$ ,  $a \in \Sigma_1 \cup \{\epsilon\}$
- $\delta(q, a) = \delta_2(q, a)$ , 当  $q \in S_2 - \{f_2\}$ ,  $a \in \Sigma_2 \cup \{\epsilon\}$
- $\delta(f_1, \epsilon) = \delta(f_2, \epsilon) = \{f_0\}$ .

▶  $L(M) = L(M_1) \cup L(M_2)$   
 $= L(r_1) \cup L(r_2) = L(r_1 | r_2) = L(r)$



## 为正规式构造NFA

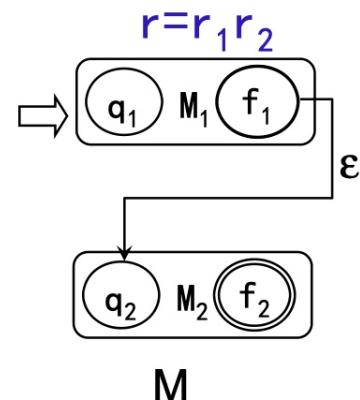
- $r=r_1|r_2$
- $r=r_1.r_2$
- $r=r_1^*$

▶ 情形1:  $r=r_1|r_2$  结论成立

▶ 情形2:  $r=r_1.r_2$ , 设  $M_i$  同情形1 ( $i=1, 2$ )

▶ 令  $M = \langle S_1 \cup S_2, \Sigma_1 \cup \Sigma_2, \delta, q_1, \{f_2\} \rangle$ , 其中  $\delta$  定义如下( $M$  的状态转换如右图):

- $\delta(q, a) = \delta_1(q, a)$ , 当  $q \in S_1 - \{f_1\}$ ,  $a \in \Sigma_1 \cup \{\epsilon\}$
  - $\delta(q, a) = \delta_2(q, a)$ , 当  $q \in S_2$ ,  $a \in \Sigma_2 \cup \{\epsilon\}$
  - $\delta(f_1, \epsilon) = \{q_2\}$
- ▶  $L(M) = L(M_1)L(M_2)$   
 $= L(r_1)L(r_2) = L(r_1.r_2) = L(r)$



## 为正规式

- 1. 对任何FA M, 都存在一个正规式r, 使得 $L(r)=L(M)$ 。
- 2. 对任何正规式r, 都存在一个FA M, 使得 $L(M)=L(r)$ 。

- $r=r_1|r_2$
- $r=r_1.r_2$
- $r=r_1^*$

- ▶ 情形1:  $r=r_1|r_2$  结论成立
- ▶ 情形2:  $r=r_1.r_2$  结论成立
- ▶ 情形3:  $r=r_1^*$ 。设 $M_1$ 同情形1

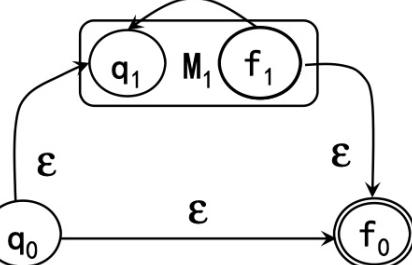
▶ 令 $M = \langle S_1 \cup \{q_0, f_0\}, \Sigma_1, \delta, q_0, \{f_0\} \rangle$ , 其中 $q_0, f_0 \notin S_1$ ,  $\delta$ 定义如下( $M$ 的状态转换如右图)：

$$(a) \delta(q_0, \epsilon) = \delta(f_1, \epsilon) = \{q_1, f_0\}$$

$$(b) \delta(q, a) = \delta_1(q, a), \text{ 当 } q \in S_1 - \{f_1\}, a \in \Sigma_1 \cup \{\epsilon\}$$

$$\begin{aligned} \triangleright L(M) &= L(M_1)^* = L(r_1)^* = L(r_1^*) \\ &= L(r) \end{aligned}$$

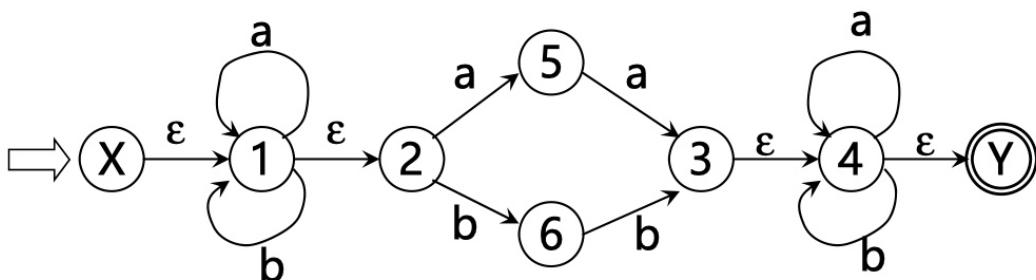
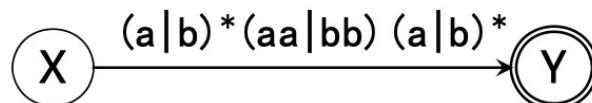
$$r=r_1^*$$



$M$

### 4.2.2 示例

- ▶  $(a|b)^*(aa|bb)(a|b)^*$



## 五、词法分析完整流程

- 根据单词符号确定正规集
- 根据正规集确定正规式
- 根据正规式确定 NFA
- 将 NFA 转换为 DFA
- 化简 DFA
- 基于 DFA 进行程序实现

单词符号	种别编码	助忆符	内码值
DIM	1	\$DIM	-
IF	2	SIF	-

## 正规式、正则表达式

DIM, IF, DO, STOP, END  
number, name, age  
125, 2169  
...

DIM  
IF  
DO  
STOP  
END  
**letter(letter|digit)\***  
**digit(digit)\***

```
curState = 初态
GetChar();
while( stateTrans[curState][ch] 有定义) {
    //存在后继状态，读入、拼接
    Concat();
    //转换入下一状态，读入下一字符
    curState= stateTrans[curState][ch];
    if curState是终态 then 返回strToken中的单词
    GetChar();
}
```

正规集

正规式

易于人工设计

FA  
DFA

NFA

DFA

## 语法分析

- 语法分析方法
  - 自下而上：算符优先分析法、LR分析法
  - 自上而下：递归下降分析法、预测分析程序

### ▶ 自下而上(Bottom-up)

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

### ▶ 自上而下(Top-down)

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

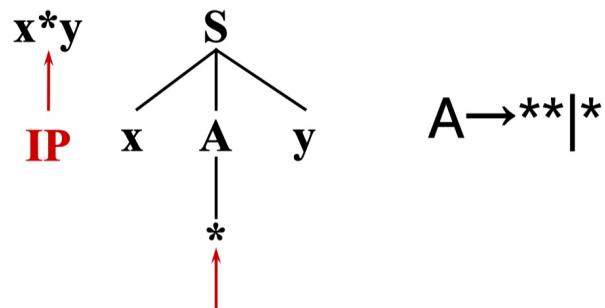
## 一、自上而下分析

# 1.1 左递归 & 回溯

## 1.1.1 面临的问题

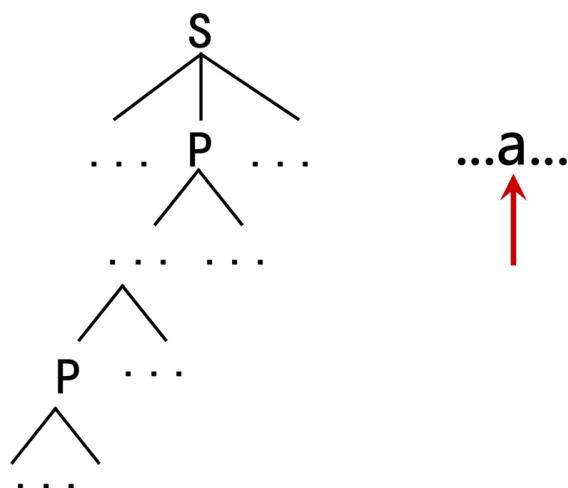
- 回溯问题

- 分析过程中，当一个非终结符用某一个候选匹配成功时，这种匹配可能是暂时的
- 因此如果在后续匹配中出错的话，不得不“回溯”
- 



- 文法左递归问题

- $P \rightarrow P\alpha$ , 匹配过程会无限循环下去
- 



## 1.1.2 消除左递归

「消除直接左递归」

- 核心思想：左递归变右递归

-

## ► 假定P关于的全部产生式是

$$P \rightarrow P\alpha_1 | P\alpha_2 | \dots | P\alpha_m | \beta_1 | \beta_2 | \dots | \beta_n$$

(每个 $\alpha$ 都不等于 $\epsilon$ , 每个 $\beta$ 都不以P开头)

## ► 左递归变右递归

$$P \rightarrow \beta_1 P' | \beta_2 P' | \dots | \beta_n P'$$

$$P' \rightarrow \alpha_1 P' | \alpha_2 P' | \dots | \alpha_m P' | \epsilon$$

- 例题

  - 

### 习题

#### ► 给定文法G(E):

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T^* F \mid F$$

$$F \rightarrow (E) \mid i$$

请消除其直接左递归。

G(E):

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \epsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \epsilon$$

$$F \rightarrow (E) \mid i$$

「消除间接左递归」

- 要求

  - 

## ► 一个文法消除左递归的条件

### ► 不含以 $\epsilon$ 为右部的产生式

### ► 不含回路

$$P \xrightarrow{+} P$$

- 核心思想

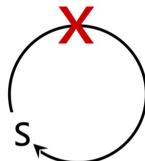
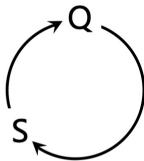
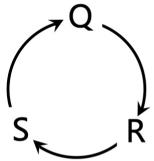
  - 消除回路中的节点

◦

► 给定文法  $G(S)$ :

$$\begin{array}{l} S \rightarrow Qc | c \\ Q \rightarrow Rb | b \\ R \rightarrow Sa | a \end{array}$$

$$S \Rightarrow Qc \Rightarrow Rbc \Rightarrow Sabc$$



$$\begin{array}{l} S \rightarrow Qc | c \\ Q \rightarrow Rb | b \\ R \rightarrow Sa | a \end{array}$$

$$\begin{array}{l} S \rightarrow Qc | c \\ Q \rightarrow Sab | ab | b \\ R \rightarrow Sa | a \end{array}$$

$$\begin{array}{l} S \rightarrow Sabc | abc | bc | c \\ Q \rightarrow Sab | ab | b \\ R \rightarrow Sa | a \end{array}$$

- 算法

◦

1. 把文法  $G$  的所有非终结符按任一种顺序排列  $P_1, P_2, \dots, P_n$ ; 按此顺序执行:

2. FOR  $i:=1$  TO  $n$  DO

BEGIN                  把  $P_i$  的规则改造成  $P_i \rightarrow a \dots | P_{i+1} \dots | P_{i+2} \dots | \dots | P_{i+k} \dots$

FOR  $j:=1$  TO  $i-1$  DO  $P_i \rightarrow a \dots | P_{j+1} \dots | P_{j+2} \dots | \dots | P_{j+k} \dots$

把形如  $P_i \rightarrow P_j \gamma$  的规则改写成

$P_i \rightarrow \delta_1 \gamma | \delta_2 \gamma | \dots | \delta_k \gamma$ ; (其中  $P_i \rightarrow \delta_1 | \delta_2 | \dots | \delta_k$

是关于  $P_j$  的所有规则)       $P_i \rightarrow a \dots | P_j \dots | P_{j+1} \dots | \dots | P_{j+k} \dots$

消除关于  $P_i$  规则的直接左递归性

END                   $P_i \rightarrow a \dots | P_{i+1} \dots | P_{i+2} \dots | \dots | P_{i+k} \dots$

3. 化简由2所得的文法, 去除从开始符号出发永远无法到达的非终结符的产生规则。

- 例题

$$\begin{array}{l} S \rightarrow Qc | c \\ Q \rightarrow Rb | b \\ R \rightarrow Sa | a \end{array}$$

顺序 1: R、Q、S

答案:

$$S \rightarrow abcS' | bcS' | cS'$$

$$S' \rightarrow abcS' | \text{空字}$$

顺序 2: S、Q、R

答案:

$$S \rightarrow Qc | c$$

$Q \rightarrow Rb \mid b$   
 $R \rightarrow bcaR' \mid caR' \mid aR'$   
 $R' \rightarrow bcaR' \mid \text{空字}$

### 1.1.3 消除回溯

- 消除回溯的核心在于能确定非终结符  $A$  所有候选可能推出的第一个符号，因此引入 FIRST 集
  -

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

$$FIRST(\alpha) = \{a \mid \alpha \xrightarrow{*} a \dots, a \in V_T\}$$

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

- 提取左公共因子 —— 令非终结符的所有候选首符集两两不相交
  -

▶ 假定关于  $A$  的规则是

$$A \rightarrow \delta\beta_1 \mid \delta\beta_2 \mid \dots \mid \delta\beta_n \mid \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_m$$

(其中，每个  $\gamma$  不以  $\delta$  开头)

那么，可以把这些规则改写成

$$\begin{aligned} A &\rightarrow \delta A' \mid \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_m \\ A' &\rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n \end{aligned}$$

▶ 经过反复提取左因子，就能够把每个非终结符 (包括新引进者) 的所有候选首符集变成两两不相交

- 由于有一些非终结符会推出空字，因此我们需要定义 FOLLOW 集
  -

▶ 假定  $S$  是文法  $G$  的开始符号，对于  $G$  的任何非终结符  $A$ ，我们定义  $A$  的 **FOLLOW** 集合

$$FOLLOW(A) = \{a \mid S \xrightarrow{*} \dots Aa \dots, a \in V_T\}$$

特别是，若  $S \xrightarrow{*} \dots A$ ，则规定  $\# \in FOLLOW(A)$

## 1.2 LL(1) 文法

### 1.2.1 LL(1) 定义

- 文法定义

- 第一个 L 表示“从左到右扫描串”
- 第二个 L 表示“最左推导”
- 第三个 1 表示“每一步只需向前查看一个符号”

▶ 构造不带回溯的自上而下分析的文法条件

1. 文法不含左递归
2. 对于文法中每一个非终结符A的各个产生式的候选首符集两两不相交。即，若

$$A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$$

则  $\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \emptyset \quad (i \neq j)$

3. 对文法中的每个非终结符A，若它存在某个候选首符集包含 $\varepsilon$ ，则

$$\text{FIRST}(\alpha_i) \cap \text{FOLLOW}(A) = \emptyset, \quad i = 1, 2, \dots, n$$

如果一个文法G满足以上条件，则称该文法G为LL(1)文法。

- LL(1) 分析法

- LL(1) 文法可以对输入串进行有效的无回溯的自上而下分析

▶ 假设要用非终结符A进行匹配，面临的输入符号为a，A的所有产生式为

$$A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$$

1. 若  $a \in \text{FIRST}(\alpha_i)$ ，则指派  $\alpha_i$  执行匹配任务；
2. 若 a 不属于任何一个候选首符集，则：
  - (1) 若  $\varepsilon$  属于某个  $\text{FIRST}(\alpha_i)$  且  $a \in \text{FOLLOW}(A)$ ，则让 A 与  $\varepsilon$  自动匹配。
  - (2) 否则，a 的出现是一种语法错误。

### 1.2.2 FIRST

- 定义

$$\text{FIRST}(\alpha) = \left\{ a \mid \alpha \xrightarrow{*} a \dots, a \in V_T \right\}$$

- 求解算法

- 核心思想：用“对有限产生式的反复扫描”代替“穷尽所有推导”

- 构造每个文法符号的 FIRST 集

- 对每一  $X \in V_T \cup V_N$ , 连续使用下面的规则, 直至每个集合 FIRST 不再增大为止:
1. 若  $X \in V_T$ , 则  $\text{FIRST}(X) = \{X\}$ 。
  2. 若  $X \in V_N$ , 且有产生式  $X \rightarrow a \dots$ , 则把  $a$  加入到  $\text{FIRST}(X)$  中; 若  $X \rightarrow \epsilon$  也是一条产生式, 则把  $\epsilon$  也加到  $\text{FIRST}(X)$  中。
  3.
    - 若  $X \rightarrow Y \dots$  是一个产生式且  $Y \in V_N$ , 则把  $\text{FIRST}(Y)$  中的所有非  $\epsilon$ -元素都加到  $\text{FIRST}(X)$  中;
    - 若  $X \rightarrow Y_1 Y_2 \dots Y_{i-1} Y_i \dots Y_k$  是一个产生式,  $Y_1, \dots, Y_{i-1}$  都是非终结符,
      - 对于任何  $j$ ,  $1 \leq j \leq i-1$ ,  $\text{FIRST}(Y_j)$  都含有  $\epsilon$  (即  $Y_1 \dots Y_{i-1} \xrightarrow{*} \epsilon$ ), 则把  $\text{FIRST}(Y_j)$  中的所有非  $\epsilon$ -元素都加到  $\text{FIRST}(X)$  中
      - 若所有的  $\text{FIRST}(Y_j)$  均含有  $\epsilon$ ,  $j = 1, 2, \dots, k$ , 则把  $\epsilon$  加到  $\text{FIRST}(X)$  中。

- 构造任何符号串的 FIRST 集

- 对文法 G 的任何符号串  $\alpha = X_1 X_2 \dots X_n$  构造集合  $\text{FIRST}(\alpha)$
1. 置  $\text{FIRST}(\alpha) = \text{FIRST}(X_1) \setminus \{\epsilon\}$ ;
  2. 若对任何  $1 \leq j \leq i-1$ ,  $\epsilon \in \text{FIRST}(X_j)$ , 则把  $\text{FIRST}(X_j) \setminus \{\epsilon\}$  加至  $\text{FIRST}(\alpha)$  中; 特别是, 若所有的  $\text{FIRST}(X_j)$  均含有  $\epsilon$ ,  $1 \leq j \leq n$ , 则把  $\epsilon$  也加至  $\text{FIRST}(\alpha)$  中。显然, 若  $\alpha = \epsilon$  则  $\text{FIRST}(\alpha) = \{\epsilon\}$ 。

### 1.2.3 FOLLOW

- 定义

$$\text{FOLLOW}(A) = \left\{ a \mid S \xrightarrow{*} \dots Aa \dots, a \in V_T \right\}$$

- 构造每个非终结符的 FOLLOW 集

- 对于文法G的每个非终结符A构造FOLLOW(A)的办法是，连续使用下面的规则，直至每个FOLLOW不再增大为止：
1. 对于文法的开始符号S，置#于FOLLOW(S)中；
  2. 若 $A \rightarrow \alpha B \beta$ 是一个产生式，则把FIRST( $\beta \setminus \{\epsilon\}$ )加至FOLLOW(B)中；
  3. 若 $A \rightarrow \alpha B$ 是一个产生式，或 $A \rightarrow \alpha B \beta$ 是一个产生式而 $\beta \stackrel{*}{\Rightarrow} \epsilon$ （即 $\epsilon \in \text{FIRST}(\beta)$ ），则把FOLLOW(A)加至FOLLOW(B)中

#### 1.2.4 习题练习

- 求解步骤
  - 消除左递归
  - 提取左公共因子
  - 求出每个非终结符的 FIRST 集、 FOLLOW 集
  - 判定是否符合 LL(1) 文法
- 练习题

► 对于文法G(E)：

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

构造每个非终结符的FIRST和FOLLOW集合

$$\begin{array}{ll} \text{FIRST}(E) = \{ (, i \} & \text{FOLLOW}(E) = \{ \#, ) \} \\ \text{FIRST}(E') = \{ +, \epsilon \} & \text{FOLLOW}(E') = \{ \#, ) \} \\ \text{FIRST}(T) = \{ (, i \} & \text{FOLLOW}(T) = \{ +, \#, ) \} \\ \text{FIRST}(T') = \{ *, \epsilon \} & \text{FOLLOW}(T') = \{ +, \#, ) \} \\ \text{FIRST}(F) = \{ (, i \} & \text{FOLLOW}(F) = \{ *, +, \#, ) \} \end{array}$$

根据LL(1)文法要求，可以判定该文法符合LL(1)。

#### 1.3 递归下降分析器

### 1.3.1 概述

- 总体思想
  - 分析程序由一组子程序组成，对每一语法单位（非终结符）构造一个相应的子程序，识别对应的语法单位
- 定义全局过程、变量
  - ADVANCE：把输入串指示器 IP 指向下一个输入符号，即读入一个单词符号
  - SYM：IP 当前所指的输入符号
  - ERROR：出错处理子程序
- 代码实现概述
  - 依据 FIRST、FOLLOW 集进行匹配

**A → TE' | BC | ε** 对应的递归下降子程序为  
**PROCEDURE A;**  
**BEGIN**  
    **IF SYM ∈ FIRST(TE') THEN**  
        **BEGIN T; E' END**  
    **ELSE IF SYM ∈ FIRST(BC) THEN**  
        **BEGIN B; C END**  
    **ELSE IF SYM ∉ FOLLOW(A) THEN**  
        **ERROR**  
**END;**

### 1.3.2 程序示例

- 文法

► 对于文法G(E):

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

构造每个非终结符的FIRST和FOLLOW集合

$FIRST(E) = \{ (, i \}$	$FOLLOW(E) = \{ \#, ) \}$
$FIRST(E') = \{ +, \epsilon \}$	$FOLLOW(E') = \{ \#, ) \}$
$FIRST(T) = \{ (, i \}$	$FOLLOW(T) = \{ +, \#, ) \}$
$FIRST(T') = \{ *, \epsilon \}$	$FOLLOW(T') = \{ +, \#, ) \}$
$FIRST(F) = \{ (, i \}$	$FOLLOW(F) = \{ *, +, \#, ) \}$

- Procedure F

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

- PROCEDURE E

```
PROCEDURE E;
BEGIN
  T; E';
END:
```

- PROCEDURE E'

- 此处可以检查 FOLLOW 集，也可以不检查，对程序正确性没有影响

```
PROCEDURE E';
IF SYM = '+' THEN
BEGIN
```

```
ADVANCE;  
T; E';  
END
```

- PROCEDURE T

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

- PROCEDURE T'

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

## 1.4 扩充的巴克斯范式

### 1.4.1 定义

- ▶ 在元符号“ $\rightarrow$ ”或“ $::=$ ”和“|”的基础上，扩充几个元语言符号：
  - ▶ 用花括号 $\{\alpha\}$ 表示闭包运算 $\alpha^*$ 。
  - ▶ 用 $\{\alpha\}_0^n$ 表示 $\alpha$ 任意重复0次至n次。
  - ▶ 用方括号 $[\alpha]$ 表示 $\{\alpha\}_0^1$ ，即表示 $\alpha$ 的出现可有可无(等价于 $\alpha|\varepsilon$ )。

- 举例

► 例如，通常的“实数”可定义为：  
 $\text{Decimal} \rightarrow [\text{Sign}]\text{Integer}.\{\text{digit}\}[\text{Exponent}]$   
 $\text{Exponent} \rightarrow E[\text{Sign}]\text{Integer}$   
 $\text{Integer} \rightarrow \text{digit}\{\text{digit}\}$   
 $\text{Sign} \rightarrow + \mid -$

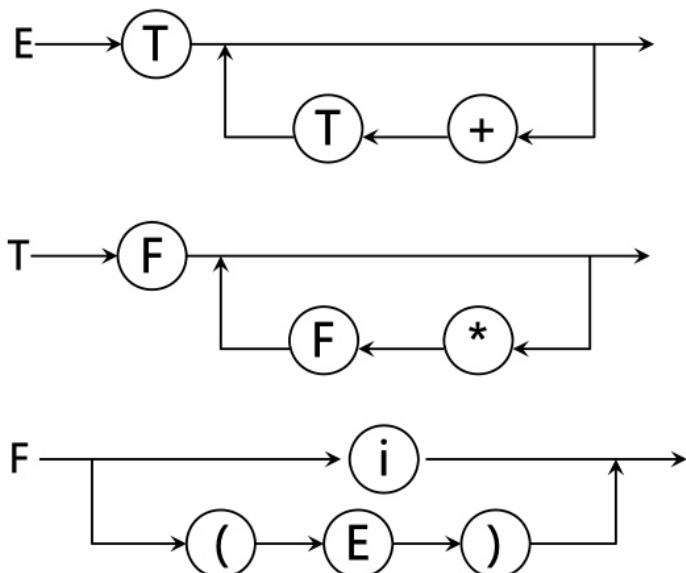
用扩充的巴克斯范式来描述语法，直观易懂，便于表示左递归消除和左公共因子提取。

#### 1.4.2 示例

► 文法  $G(E)$ ：  
 $E \rightarrow T \mid E + T$   
 $T \rightarrow F \mid T^*F$   
 $F \rightarrow i \mid (E)$

可表示成

$E \rightarrow T\{+T\}$   
 $T \rightarrow F\{^*F\}$   
 $F \rightarrow i \mid (E)$



## 设计递归下降分析

- ▶  $E \rightarrow T\{+T\}$
- ▶  $T \rightarrow F\{*F\}$
- ▶  $F \rightarrow i \mid (E)$
- ▶ 可构造一组递归下降分析程序

```
PROCEDURE T;  
BEGIN  
    F;  
    WHILE SYM= '*' DO  
        BEGIN  
            ADVANCE;  
            F  
        END  
    END;
```

```
PROCEDURE E;  
BEGIN  
    T;  
    WHILE SYM= '+' DO  
        BEGIN  
            ADVANCE;  
            T  
        END  
    END;
```

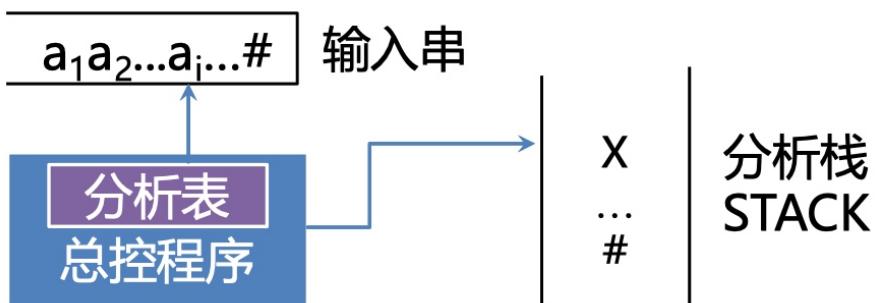
```
PROCEDURE F;  
IF SYM= 'i' THEN ADVANCE  
ELSE  
    IF SYM= '(' THEN  
        BEGIN  
            ADVANCE;  
            E;  
            IF SYM= ')' THEN  
                ADVANCE  
            ELSE ERROR  
        END  
    ELSE ERROR;
```

## 1.5 预测分析程序

### 1.5.1 预测分析程序组成

- 总控程序
- 分析表
- 分析栈

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



### 1.5.2 构造预测分析表

- 分析产生式语句的FIRST集
- 如果非终结符可以推空，则加入 FOLLOW 集

► 构造G的分析表M[A, a], 确定每个产生式  
A→α在表中的位置

1. 对文法G的每个产生式A→α执行第2步和第3步；
2. 对每个终结符a ∈ FIRST(α), 把A→α加至M[A, a]中；
3. 若ε ∈ FIRST(α), 则对任何b ∈ FOLLOW(A)把A→α加至M[A, b]中。
4. 把所有无定义的M[A, a]标上“出错标志”。

### • 练习例子

## 分析表的构造

► 对于文法G(E):

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

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

构造该文法的预测分析表。

1. 对文法G的每个产生式A→α执行第2步和第3步；
2. 对每个终结符a ∈ FIRST(α), 把A→α加至M[A, a]中；
3. 若ε ∈ FIRST(α), 则对任何b ∈ FOLLOW(A)把A→α加至M[A, b]中。
4. 把所有无定义的M[A, a]标上“出错标志”。

### 1.5.3 预测分析示例

步骤	符号栈	输入串	所用产生式
----	-----	-----	-------

0	#E	i <sub>1</sub> *i <sub>2</sub> +i <sub>3</sub> #	
1	#E'T	i <sub>1</sub> *i <sub>2</sub> +i <sub>3</sub> #	E→TE'
2	#E'T'F	i <sub>1</sub> *i <sub>2</sub> +i <sub>3</sub> #	T→FT'
3	#E'T'i	i <sub>1</sub> *i <sub>2</sub> +i <sub>3</sub> #	F→i
4	#E'T'	*i <sub>2</sub> +i <sub>3</sub> #	
5	#E'T'F*	*i <sub>2</sub> +i <sub>3</sub> #	T'→*FT'
6	#E'T'F	i <sub>2</sub> +i <sub>3</sub> #	
7	#E'T'i	i <sub>2</sub> +i <sub>3</sub> #	F→i
8	#E'T'	+i <sub>3</sub> #	
9	#E'	+i <sub>3</sub> #	T'→ε
10	#E'T+	+i <sub>3</sub> #	E'→+TE'
11	#E'T	i <sub>3</sub> #	
12	#E'T'F	i <sub>3</sub> #	T→FT'
13	#E'T'i	i <sub>3</sub> #	F→i
14	#E'T'	#	
15	#E'	#	T'→ε
16	#	#	E'→ε

## 预测分析示例

► 对于文法G(E):

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

输入串为i<sub>1</sub>\*i<sub>2</sub>+i<sub>3</sub>, 利用分析表进行预测分析

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

### 1.5.4 二义性文法

并不是所有文法在消除左递归、提取左公共因子后都符合 LL(1) 文法要求，因此我们可以根据具体情况来修改预测分析表。

- 文法举例

► G(S):

$$\begin{aligned} S &\rightarrow iCtS \mid iCtSeS \mid a \\ C &\rightarrow b \end{aligned}$$

提取左因子之后，改写成：

► G(S):

$$\begin{aligned} S &\rightarrow iCtSS' \mid a \\ S' &\rightarrow eS \mid \epsilon \\ C &\rightarrow b \end{aligned}$$

- 二义文法对应的预测分析表

	a	b	e	i	t	#
S	$S \rightarrow a$			$S \rightarrow iCtSS'$		
S'			$S' \rightarrow eS$			$S' \rightarrow \epsilon$
C		$C \rightarrow b$				

此处出现冲突的格子其实代表的是 if...then if...then...else... 语句中，else 匹配的是第一个 if，还是第二个 if 的问题，因此我们可以根据程序的实际要求进行选择，并删除另一个，使该文法满足 LL(1) 的特性。

## 二、自下而上分析

自下而上分析采用“移进-归约”思想。

- 基本思想
  - 用一个寄存符号的先进后出栈，把输入符号一个一个地移进栈里，当栈顶形成某个产生式的候选式时，即把栈顶的这一部分替换成（归约为）该产生式的左部符号。
- 核心方法
  - 算符优先分析法
    - 按照算符的优先关系和结合性质进行语法分析
    - 适合分析表达式
  - LR 分析法
    - 规范规约：句柄作为可归约串

### 2.1 基础概念

#### 2.1.1 短语

- 定义（短语、直接短语）
  - ▶ 定义：令G是一个文法，S是文法的开始符号，假定 $\alpha\beta\delta$ 是文法G的一个句型，如果有
 
$$S \xrightarrow{*} \alpha A \delta \text{ 且 } A \xrightarrow{+} \beta$$
 则 $\beta$ 称是句型 $\alpha\beta\delta$ 相对于非终结符A的短语。
  - ▶ 如果有 $A \rightarrow \beta$ ，则称 $\beta$ 是句型 $\alpha\beta\delta$ 相对于规则 $A \rightarrow \beta$ 的直接短语。

- 语法树的角度
  - 两代以上子树的叶子结点构成的序列为短语
  - 子树只有两代，则为直接短语

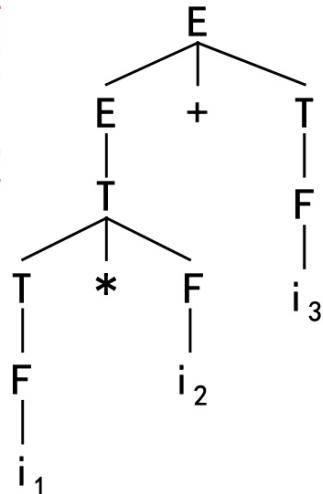
► 在一个句型对应的语法树中

► 以某非终结符为根的**两代以上的子树的所有末端结点从左到右排列**就是相对于该非终结符的一个**短语**

► 如果子树只有两代，则该短语就是**直接短语**

短语： $i_1, i_2, i_3, i_1*i_2, i_1*i_2+i_3$

直接短语： $i_1, i_2, i_3$



### 2.1.2 素短语

- 定义（素短语、最左素短语）

- 属于短语
- 至少有一个终结符
- 除自身外不再包含更小的素短语

► 一个文法G的句型的**素短语**是指这样一个短语，它至少含有一个终结符，并且，除它自身之外不再含任何更小的素短语

► **最左素短语**是指处于句型最左边的那个素短语

- 示例

## 示例：各类短语的识别

► 考虑文法G(E)：

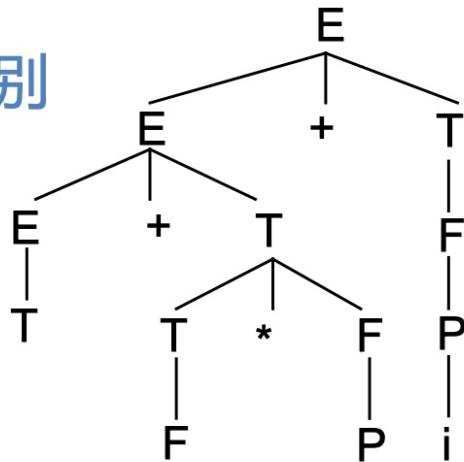
(1)  $E \rightarrow E + T \mid T$

(2)  $T \rightarrow T^* F \mid F$

(3)  $F \rightarrow P \uparrow F \mid P$

(4)  $P \rightarrow (E) \mid i$

对于句型：  $T + F^* P + i$



短语：  $T, F, P, i, F^*P, T+F^*P, T+F^*P+i$

直接短语：  $T, F, P, i$

素短语：  $i, F^*P$

最左素短语：  $F^*P$

## 2.2 算符优先文法

### 2.2.1 优先关系

► 任何两个可能相继出现的终结符a与b可能三种优先关系

►  $a < b$  a的优先级**低于**b

►  $a \equiv b$  a的优先级**等于**b

►  $a > b$  a的优先级**高于**b

► 算符优先关系与数学上的 $<>=$ 不同

►  $+ < +$

►  $a < b$ 并不意味着 $b > a$ , 如 ( $< +$  和  $+ < ($ )

### 2.2.2 算符文法

- 算符文法定义

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

► 约定：

- a、b代表任意终结符
- P、Q、R代表任意非终结符
- '...' 代表由终结符和非终结符组成的任意序列，包括空字

- 算符优先文法定义

- 引入终结符的优先级

- 假定G是一个不含 $\epsilon$ -产生式的算符文法。对于任何一对终结符a、b，我们说：
      1. **a=b**，当且仅当文法G中含有形如 $P \rightarrow \dots ab\dots$ 或 $P \rightarrow \dots aQb\dots$ 的产生式；
      2. **a<b**，当且仅当G中含有形如 $P \rightarrow \dots aR\dots$ 的产生式，而 $R \xrightarrow{+} b\dots$ 或 $R \xrightarrow{+} Qb\dots$ ；
      3. **a>b**，当且仅当G中含有形如 $P \rightarrow \dots Rb\dots$ 的产生式，而 $R \xrightarrow{+} \dots a$ 或 $R \xrightarrow{+} \dots aQ$ 。
    - 如果一个算符文法G中的任何终结符对(a, b)至多只满足**a=b**、**a<b**和**a>b**这三个关系之一，则称G是一个**算符优先文法**。

### 2.2.3 FIRSTVT、LASTVT

- FIRSTVT

► 反复使用下面两条规则构造集合FIRSTVT(P)

1. 若有产生式 $P \rightarrow a\dots$ 或 $P \rightarrow Qa\dots$ ，则 $a \in \text{FIRSTVT}(P)$
2. 若 $a \in \text{FIRSTVT}(Q)$ ，且有产生式 $P \rightarrow Q\dots$ ，则 $a \in \text{FIRSTVT}(P)$

$$\text{FIRSTVT}(P) = \{a \mid P \xrightarrow{+} a\dots \text{ 或 } P \xrightarrow{+} Q a\dots, \quad a \in V_T \text{ 且 } Q \in V_N\}$$

- LASTVT

► 反复使用下面两条规则构造集合LASTVT(P)

1. 若有产生式  $P \rightarrow \dots a$  或  $P \rightarrow \dots aQ$ , 则  
 $a \in \text{LASTVT}(P)$
2. 若  $a \in \text{LASTVT}(Q)$ , 且有产生式  $P \rightarrow \dots Q$ , 则  
 $a \in \text{LASTVT}(P)$

$$\text{LASTVT}(P) = \{ a \mid P \xrightarrow{+} \dots a \text{ 或 } P \xrightarrow{+} \dots aQ, a \in V_T \text{ 且 } Q \in V_N \}$$

• 具体实现方式

- 利用栈不断遍历所有产生式
- 若栈 STACK 不空, 就将栈顶项弹出, 记此项为  $(Q, a)$ 。对于每个形如  $P \rightarrow Q \dots$  的产生式, 若  $F[P, a]$  为假, 则变其值为真且将  $(P, a)$  推进 STACK 栈。
- 上述过程一直重复, 直至栈 STACK 为空为止。

► 算法的一种实现

- 布尔数组  $F[P, a]$ , 使得  $F[P, a]$  为真的条件是, 当且仅当  $a \in \text{FIRSTVT}(P)$ 。开始时, 按上述的规则1对每个数组元素  $F[P, a]$  赋初值。
- 栈 STACK, 把所有初值为真的数组元素  $F[P, a]$  的符号对  $(P, a)$  全都放在 STACK 之中。

- |  |
|--|
| <p>1. 若有产生式 <math>P \rightarrow a \dots</math> 或 <math>P \rightarrow Q a \dots</math>, 则 <math>a \in \text{FIRSTVT}(P)</math></p> <p>2. 若 <math>a \in \text{FIRSTVT}(Q)</math>, 且有产生式 <math>P \rightarrow Q \dots</math>, 则<br/><math>a \in \text{FIRSTVT}(P)</math></p> |
|--|

• FIRSTVT、LASTVT 具体作用

- 根据 FIRSTVT 和 LASTVT 集合, 检查每个产生式的候选式, 确定满足关系  $\prec$  和  $\succ$  的所有终结符对
  - 假定有个产生式的一个候选形为  $\dots aP \dots$ , 那么, 对任何  $b \in \text{FIRSTVT}(P)$ , 有  $a \prec b$
  - 假定有个产生式的一个候选形为  $\dots Pb \dots$ , 那么, 对任何  $a \in \text{LASTVT}(P)$ , 有  $a \succ b$

- 示例

- 求解时还需要考虑  $\#E\#$  式子

## 示例：构造优先关系表

$G(E)$ :

- $E \rightarrow E + T | T$
- $T \rightarrow T * F | F$
- $F \rightarrow P \uparrow F | P$
- $P \rightarrow (E) | i$

- 计算文法G的FIRSTVT和LASTVT
- 构造构造优先关系表
- G是算符优先文法

还要考虑  $\#E\#$

$\text{FIRSTVT}(E)=\{+, *, \uparrow, (, i\}$   
 $\text{FIRSTVT}(T)=\{*, \uparrow, (, i\}$   
 $\text{FIRSTVT}(F)=\{\uparrow, (, i\}$   
 $\text{FIRSTVT}(P)=\{(, i\}$   
  
 $\text{LASTVT}(E)=\{+, *, \uparrow, ), i\}$   
 $\text{LASTVT}(T)=\{*, \uparrow, ), i\}$   
 $\text{LASTVT}(F)=\{\uparrow, ), i\}$   
 $\text{LASTVT}(P)=\{), i\}$

	+	*	$\uparrow$	i	(	)	#
+	>	<	<	<	<	<	>
*	>	>	<	<	<	<	>
$\uparrow$	>	>	<	<	<	<	>
i	>	>	>			>	>
(	<	<	<	<	<	<b>=</b>	
)	>	>	>			>	>
#	<	<	<	<	<	<b>=</b>	

### 2.2.4 最左素短语定理

- 具体定理

- 出现了一个山峰式的句型

- 算符优先文法句型(括在两个#之间)的一般形式:

$\#N_1a_1N_2a_2\dots N_na_nN_{n+1}\#$

其中,  $a_i$ 都是终结符,  $N_i$ 是可有可无的非终结符。

- 定理: 一个算符优先文法G的任何句型的最左素短语是满足如下条件的最左子串  $N_ja_j\dots N_ia_iN_{i+1}$ ,

$a_{j-1} < a_j$

$a_j = a_{j+1}, \dots, a_{i-1} = a_i$

$a_i > a_{i+1}$

$\#N_1a_1N_2a_2\dots a_{j-1} N_j a_j\dots N_i a_i N_{i+1} a_{i+1} \dots N_n a_n N_{n+1}\#$

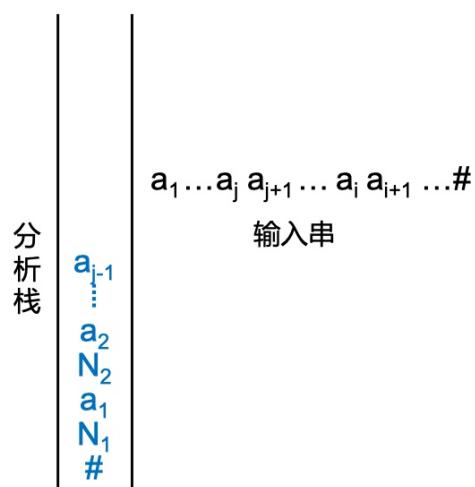
### 2.2.5 算符优先分析算法

- 具体算法过程

```

#N1a1N2a2... aj-1 Nj aj...Ni ai Ni+1 ai+1 ...NnanNn+1#
k:=1;
S[k]:='#';
REPEAT
    把下一个输入符号读进a中;
    IF S[k]∈VT THEN j:=k ELSE j:=k-1;
    WHILE S[j]>a DO
        BEGIN
            REPEAT
                Q:=S[j];
                IF S[j-1]∈VT THEN j:=j-1
                ELSE j:=j-2
            UNTIL S[j]≤Q;
            把S[j+1]...S[k]归约为某个N;
            k:=j+1;
            S[k]:=N
        END OF WHILE;
        IF S[j]≤a OR S[j]=a THEN
            BEGIN k:=k+1;S[k]:=a END
        ELSE ERROR /*调用出错诊察程序*/
    UNTIL a='#'

```



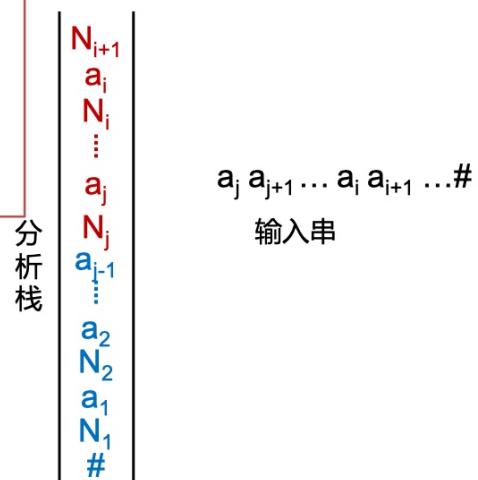
- 具体归约过程

- 从左至右，终结符对终结符，非终结符对非终结符，且只要求终结符对应匹配

```

#N1a1N2a2... aj-1 Nj aj...Ni ai Ni+1 ai+1 ...NnanNn+1#
k:=1;
S[k]:='#';
REPEAT
    把自左至右，终结符对终结符，非终结符对非终结符，而且对应的终结符相同。
    IF S[j+1]...S[k]归约为某个N;
    BEGIN
        N → X1 X2 ... Xk-j
        ↓   ↓   ...   ↓
        S[j+1] S[j+2] ... S[k]
    ELSE j:=j-2
    UNTIL S[j]≤Q;
    把S[j+1]...S[k]归约为某个N;
    k:=j+1;
    S[k]:=N
    END OF WHILE;
    IF S[j]≤a OR S[j]=a THEN
        BEGIN k:=k+1;S[k]:=a END
    ELSE ERROR /*调用出错诊察程序*/
    UNTIL a='#'

```



- 算符分析树 v.s. 语法树

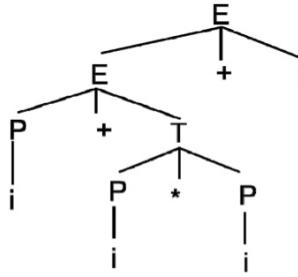
- 二者不同，分析树可能会更简便

## ► 算符优先分析结果不一定是语法树

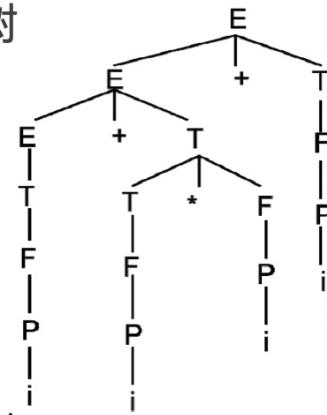
考虑文法  $G(E)$ :

- (1)  $E \rightarrow E + T \mid T$
- (2)  $T \rightarrow T * F \mid F$
- (3)  $F \rightarrow P \uparrow F \mid P$
- (4)  $P \rightarrow (E) \mid i$

的句子  $i+i*i+i$



算符优先分析得到的分析树



语法树

## 2.3 LR 分析法

### 2.3.1 LR 文法

- LR 文法定义
  - 对于一个文法，如果能够构造一张分析表，使得它的每个入口均是唯一确定的，则这个文法就称为 LR 文法。
- LR( $k$ ) 文法定义
  - 一个文法，如果能用一个每步顶多向前检查  $k$  个输入符号的 LR 分析器进行分析，则这个文法就称为 LR( $k$ ) 文法。
- LR 文法与二义文法
  - LR 文法不是二义的，二义文法肯定不会是 LR 的
  - LR 文法是二义文法的真子集，因此是二义文法的充分条件
- 实际应用
  - 虽然 LR 文法不能涵盖所有二义文法，但程序设计语言仍然使用 LR 分析法，如果分析表中有冲突，则人工手动去除

### 2.3.2 概念说明

- LR 分析法
  - 关键在于生成分析表，再根据分析表进行语法分析



- 句柄
  - 一个句型的最左直接短语就是该句型的句柄

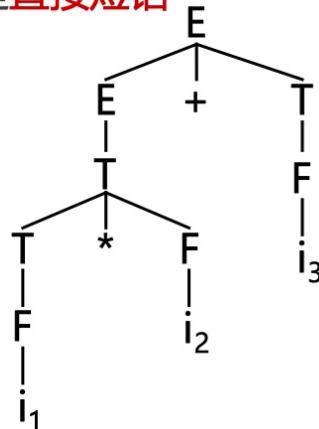
► 在一个句型对应的语法树中

- 以某非终结符为根的两代以上的子树的所有末端结点从左到右排列就是相对于该非终结符的一个短语
- 如果子树只有两代，则该短语就是直接短语
- 最左两代子树末端就是句柄

短语:  $i_1 i_2 i_3 i_1 * i_2 i_1 * i_2 + i_3$

直接短语:  $i_1 i_2 i_3$

句柄:  $i_1$



- 规范归约
  - 规范归约即每次对句柄进行归约
  - 规范归约 = 最左归约 = 最右推导的逆过程
  - 最右推导 = 规范推导
  - 由规范推导推出的句型称为规范句型

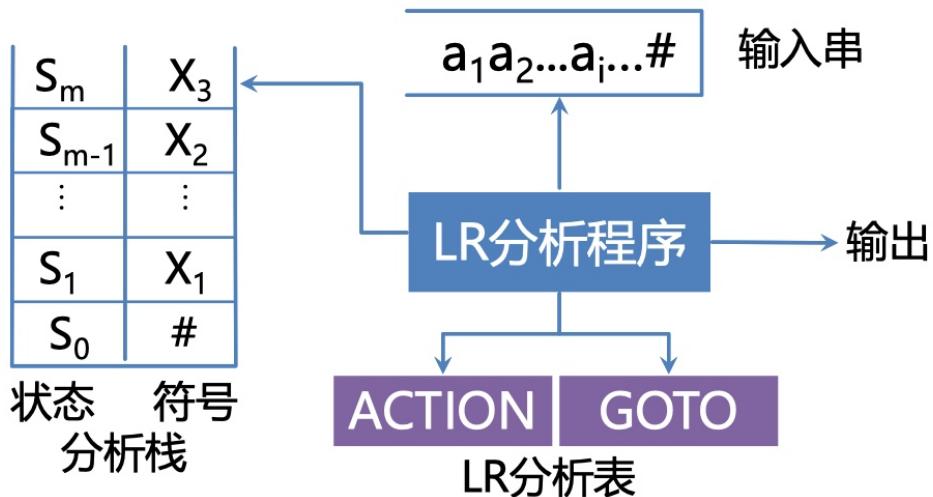
▶ 定义：假定 $\alpha$ 是文法G的一个句子，我们称序列 $\alpha_n, \alpha_{n-1}, \dots, \alpha_0$ 是 $\alpha$ 的一个规范归约，如果此序列满足：

1.  $\alpha_n = \alpha$
2.  $\alpha_0$ 为文法的开始符号，即 $\alpha_0 = S$
3. 对任何 $i$ ,  $0 \leq i \leq n$ ,  $\alpha_{i-1}$ 是从 $\alpha_i$ 经把句柄替换成为相应产生式左部符号而得到的

### 2.3.3 LR 分析过程

- 规范规约的关键在于寻找句柄
  - 历史：已移入符号栈的内容
  - 展望：根据产生式推测未来可能遇到的输入符号
  - 现实：当前的输入符号

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



### • LR 分析表

- ACTION[s, a]: 当状态 s 面临输入符号 a 时，应采取什么动作
- GOTO[s, X]: 状态 s 面对文法符号 X 时，下一状态是什么
- 分析表中 s 表示移进 (shift)，r 表示归约 (reduce)

# LR分析过程

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

► 分析器根据  $ACTION(s_m, a_i)$  确定下一步动作

1. 若  $ACTION(s_m, a_i)$  为移进，且  $s$  为下一状态，则格局变为：

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

2. 若  $ACTION(s_m, a_i)$  为按  $A \rightarrow \beta$  归约，格局变为：

$(s_0 s_1 \dots s_{m-r} s, \# X_1 \dots X_{m-r} A, a_i a_{i+1} \dots a_n \#)$

此处， $s = GOTO(s_{m-r}, A)$ ,  $r$  为  $\beta$  的长度,  $\beta = X_{m-r+1} \dots X_m$

3. 若  $ACTION(s_m, a_i)$  为“接受”，则格局变化过程终止，宣布分析成功。

4. 若  $ACTION(s_m, a_i)$  为“报错”，则格局变化过程终止，报告错误。

其中归约的具体步骤如下所示：

$(s_0 s_1 \dots s_{m-r} s_{m-r+1} \dots s_m, \# X_1 \dots X_{m-r} X_{m-r+1} \dots X_m, a_i a_{i+1} \dots a_n \#)$
$(s_0 s_1 \dots s_{m-r}, \# X_1 \dots X_{m-r}, a_i a_{i+1} \dots a_n \#)$
$(s_0 s_1 \dots s_{m-r}, \# X_1 \dots X_{m-r} A, a_i a_{i+1} \dots a_n \#)$
$(s_0 s_1 \dots s_{m-r} s, \# X_1 \dots X_{m-r} A, a_i a_{i+1} \dots a_n \#)$

- LR分析过程性质

- 在任何时候，分析栈里面的内容和扫描串剩下的内容拼接起来，总是一个规范句型。
- 一旦栈顶出现了当时栈内内容和栈外输入串拼接出来的那个句型的句柄，马上就会归约。

## 2.3.4 LR 分析示例

- 图 1

步骤	状态	符号	输入串	文法G(E):
(1)	0	#	i*i+i#	(1) E→E + T
(2)	05	#i	*i+i#	(2) E→T
(3)	03	#F	*i+i#	(3) T→T*F
(4)	02	#T	*i+i#	(4) T→F
(5)	027	#T*	i+i#	(5) F→(E)
				(6) F→i

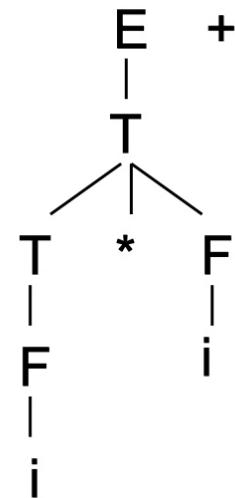
状态	ACTION						GOTO		
	i	+	*	(	)	#	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4			9	3	
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

T      \*
  
 |  
 F  
 |  
 i

• 图 2

步骤	状态	符号	输入串	文法G(E):
(5)	027	#T*	i+i#	(1) E→E + T
(6)	0275	#T*i	+i#	(2) E→T
(7)	027 <u>10</u>	#T*T	+i#	(3) T→T*F
(8)	02	#T	+i#	(4) T→F
(9)	01	#E	+i#	(5) F→(E)
(10)	016	#E+	i#	(6) F→i

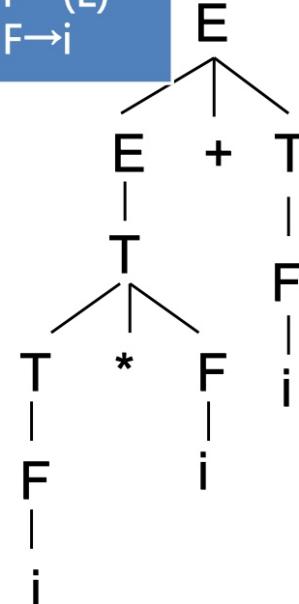
步骤	状态	符号	输入串	文法G(E):
(5)	027	#T*	i+i#	(1) E→E + T
(6)	0275	#T*i	+i#	(2) E→T
(7)	027 <u>10</u>	#T*T	+i#	(3) T→T*F
(8)	02	#T	+i#	(4) T→F
(9)	01	#E	+i#	(5) F→(E)
(10)	016	#E+	i#	(6) F→i



• 图 3

步骤	状态	符号	输入串
(10)	016	#E+	i#
(11)	0165	#E+i	#
(12)	0163	#E+F	#
(13)	0169	#E+T	#
(14)	01	#E	#
(15)	接受		

文法G(E):  
(1) E→E + T  
(2) E→T  
(3) T→T\*F  
(4) T→F  
(5) F→(E)  
(6) F→i

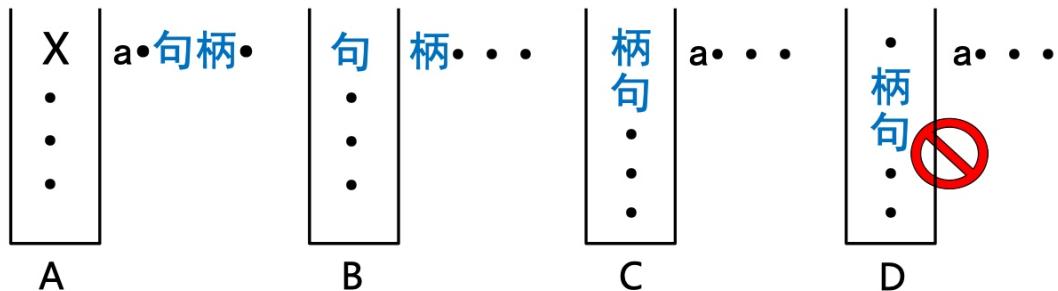


状态	ACTION						GOTO		
	i	+	*	(	)	#	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

## 2.4 LR(0) 文法

### 2.4.1 活前缀

- 概念引入原因
  - 为了定义在规范规约时，不会出现 D 这种情况，引入活前缀概念



- 活前缀定义
  - 规范句型的一个前缀，且该前缀不含句柄之后的任何符号
  - 活前缀后续符号均为入栈，因此为终结符

- ▶ 活前缀：是指规范句型的一个前缀，这种前缀不含句柄之后的任何符号。即，对于规范句型  $\alpha\beta\delta$ ,  $\beta$  为句柄，如果  $\alpha\beta=u_1u_2\dots u_r$ , 则符号串  $u_1u_2\dots u_i (1 \leq i \leq r)$  是  $\alpha\beta\delta$  的活前缀。( $\delta$  必为终结符串)
- ▶ 规范归约过程中，保证分析栈中总是活前缀，就说明分析采取的移进/归约动作是正确的

## 2.4.2 文法拓广

- 文法拓广目的
  - 与构造 DFA 时引入初态、终态节点目的一致
  - 使得文法的 DFA 中只有一个初态、一个终态

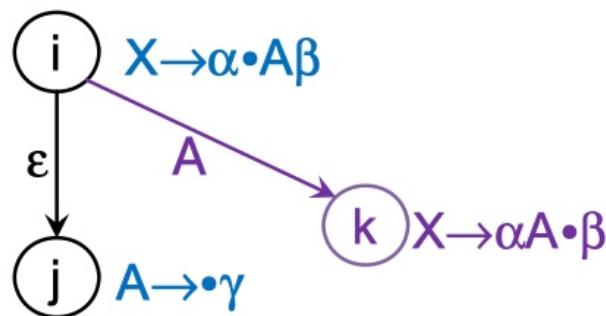
- ▶ 将文法  $G(S)$  拓广为  $G'(S')$ 
  - ▶ 构造文法  $G'$ ，它包含了整个  $G$ ，并引进不出现在  $G$  中的非终结符  $S'$ 、以及产生式  $S' \rightarrow S$ ,  $S'$  是  $G'$  的开始符号
  - ▶ 称  $G'$  是  $G$  的拓广文法

## 2.4.3 LR(0) 项目

- ▶ LR(0) 项目
  - ▶ 在每个产生式的右部添加一个圆点
  - ▶ 表示我们在分析过程中看到了产生式多大部分
- ▶  $A \rightarrow XYZ$  有四个项目
  - ▶  $A \rightarrow \cdot XYZ$   $A \rightarrow X \cdot YZ$   $A \rightarrow XY \cdot Z$   $A \rightarrow XYZ \cdot$
  - ▶  $A \rightarrow \alpha \cdot$  称为“归约项目”
  - ▶ 归约项目  $S' \rightarrow \alpha \cdot$  称为“接受项目”
  - ▶  $A \rightarrow \alpha \cdot a\beta$  ( $a \in V_T$ ) 称为“移进项目”
  - ▶  $A \rightarrow \alpha \cdot B\beta$  ( $B \in V_N$ ) 称为“待约项目”

## 2.4.4 识别活前缀的 DFA 构造法1

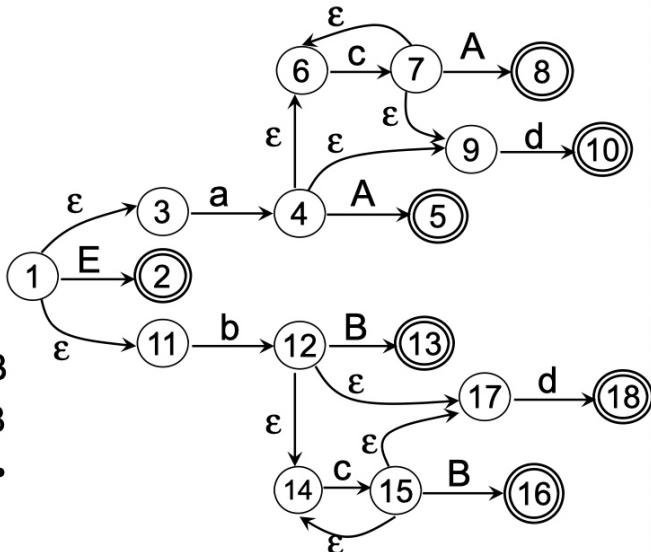
- 总体思想
  - 先构造 NFA，再将 NFA 通过构造的方式转换成 DFA
- NFA 构造方法



- DFA 构造举例

### 识别活前缀的NFA

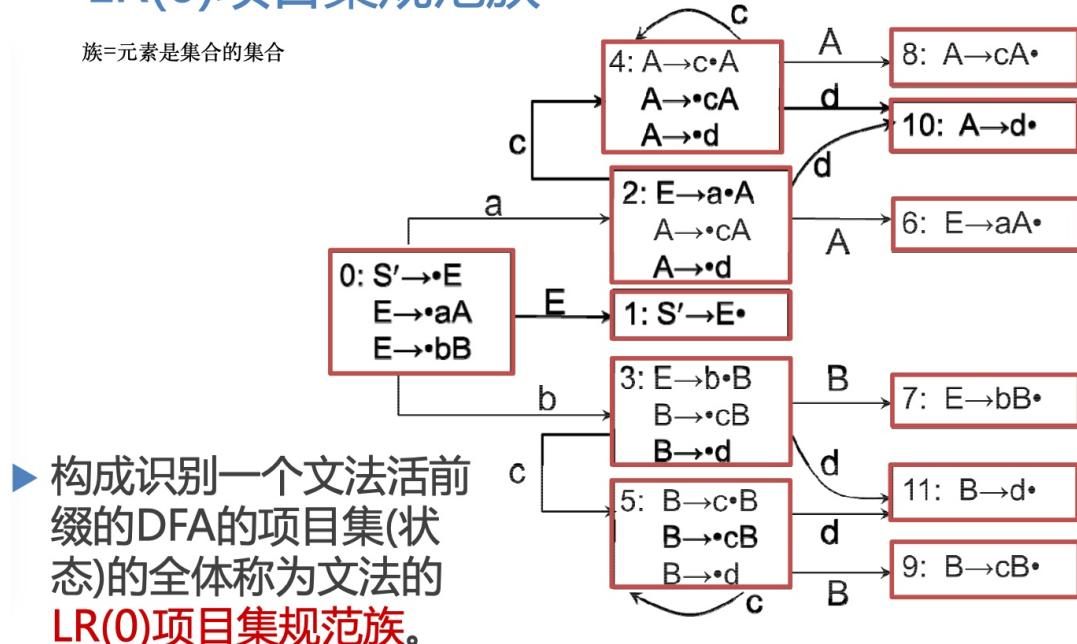
1.  $S' \rightarrow \bullet E$
2.  $S' \rightarrow E \bullet$
3.  $E \rightarrow \bullet aA$
4.  $E \rightarrow a \bullet A$
5.  $E \rightarrow aA \bullet$
6.  $A \rightarrow \bullet cA$
7.  $A \rightarrow c \bullet A$
8.  $A \rightarrow cA \bullet$
9.  $A \rightarrow \bullet d$
10.  $A \rightarrow d \bullet$
11.  $E \rightarrow \bullet bB$
12.  $E \rightarrow b \bullet B$
13.  $E \rightarrow bB \bullet$
14.  $B \rightarrow \bullet cB$
15.  $B \rightarrow c \bullet B$
16.  $B \rightarrow cB \bullet$
17.  $B \rightarrow \bullet d$
18.  $B \rightarrow d \bullet$



- NFA => DFA
  - DFA 的项目集为文法的LR(0)项目集规范族
  - 族 = 元素是集合的集合

# LR(0)项目集规范族

族=元素是集合的集合



## 2.4.5 识别活前缀的 DFA 构造法2

- 总体思想
  - 通过引入有效项目的概念，直接构造出 DFA
- 有效项目概念

### 有效项目的性质

项目  $A \rightarrow \beta_1 \cdot \beta_2$  对活前缀  $\alpha \beta_1$  是有效的，其条件是存在规范推导：

$$S' \xrightarrow{*} \alpha A \omega \Rightarrow_R \alpha \beta_1 \beta_2 \omega$$

- ▶ 若项目  $A \rightarrow \alpha \cdot B \beta$  对活前缀  $\eta = \delta \alpha$  是有效的且  $B \rightarrow \gamma$  是一个产生式，则项目  $B \rightarrow \cdot \gamma$  对  $\eta = \delta \alpha$  也是有效的。**

证明：

若项目  $A \rightarrow \alpha \cdot B \beta$  对活前缀  $\eta = \delta \alpha$  是有效的，则有

$$S' \xrightarrow{*} \delta A \omega \Rightarrow_R \delta \alpha B \beta \omega$$

设  $\beta \omega \xrightarrow{*} \varphi \omega$ , 那么

$$S' \xrightarrow{*} \delta A \omega \Rightarrow_R \delta \alpha B \beta \omega \xrightarrow{*} \delta \alpha B \varphi \omega \Rightarrow_R \delta \alpha \gamma \varphi \omega$$

所以， $B \rightarrow \cdot \gamma$  对  $\eta = \delta \alpha$  也是有效的

- LR(0)项目集规范族构造

- 拓广文法

## ► 将文法G(S)拓广为G'(S')

- ▶ 构造文法 $G'$ ，它包含了整个 $G$ ，并引进不出现在 $G$ 中的非终结符 $S'$ 、以及产生式 $S' \rightarrow S$ ， $S'$ 是 $G'$ 的开始符号
- ▶  $G'$ 唯一的“接受”态：仅含项目 $S' \rightarrow S^\bullet$ 的状态

- 构造闭包

## 项目集的闭包CLOSURE

- ▶ 假定 $I$ 是文法 $G'$ 的任一项目集，定义和构造 $I$ 的闭包 $CLOSURE(I)$ 如下：
  1.  $I$ 的任何项目都属于 $CLOSURE(I)$ ；
  2. 若 $A \rightarrow \alpha \cdot B\beta$ 属于 $CLOSURE(I)$ ，那么，对任何关于 $A$ 的产生式 $B \rightarrow \gamma$ ，项目 $B \rightarrow \cdot \gamma$ 也属于 $CLOSURE(I)$ ；
  3. 重复执行上述两步骤直至 $CLOSURE(I)$ 不再增大为止。

- 状态转移

## 状态转换函数

- ▶ 为了识别活前缀，我们定义一个状态转换函数 $GO$ 是一个状态转换函数。 $I$ 是一个项目集， $X$ 是一个文法符号。函数值 $GO(I, X)$ 定义为：

$$GO(I, X) = CLOSURE(J)$$

其中

$$J = \{ \text{任何形如 } A \rightarrow \alpha X \cdot \beta \text{ 的项目} \mid A \rightarrow \alpha \cdot X\beta \text{ 属于 } I \}.$$

- ▶ 直观上说，若 $I$ 是对某个活前缀 $\gamma$ 有效的项目集，那么， $GO(I, X)$ 便是对 $\gamma X$ 有效的项目集。

## 2.4.6 LR(0)分析表构造

- 通过 DFA 构造分析表

### LR(0)分析表的ACTION和GOTO子表构造

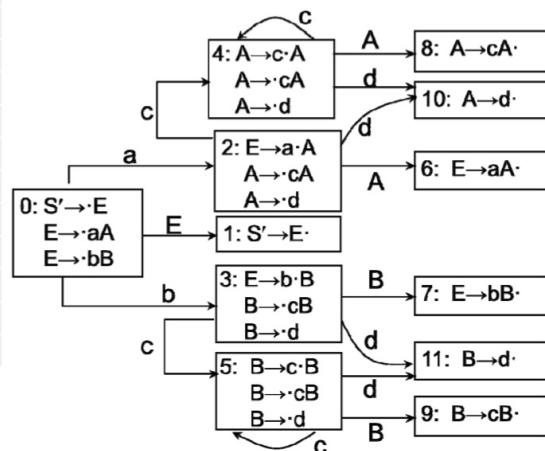
- 若项目  $A \rightarrow \alpha \cdot a\beta$  属于  $I_k$  且  $GO(I_k, a) = I_j$ ,  $a$  为终结符, 则置  $ACTION[k, a]$  为 “ $s_j$ ”。
- 若项目  $A \rightarrow \alpha \cdot$  属于  $I_k$ , 那么, 对任何终结符  $a$ (或结束符#), 置  $ACTION[k, a]$  为 “ $r_j$ ”(假定产生式  $A \rightarrow \alpha$  是文法  $G'$  的第  $j$  个产生式)。
- 若项目  $S' \rightarrow S \cdot$  属于  $I_k$ , 则置  $ACTION[k, #]$  为 “acc”。
- 若  $GO(I_k, A) = I_j$ ,  $A$  为非终结符, 则置  $GOTO[k, A] = j$ 。
- 分析表中凡不能用规则1至4填入信息的空白格均置上“报错标志”。

- 举例

### 示例：LR(0)分析表的构造

状态	ACTION					GOTO		
	a	b	c	d	#	E	A	B
0	s2	s3				1		
1					acc			
2			s4 s10			6		
3			s5 s11				7	
4			s4 s10			8		
5			s5 s11				9	
6	r1 r1 r1 r1 r1							
7	r2 r2 r2 r2 r2							
8	r3 r3 r3 r3 r3							
9	r5 r5 r5 r5 r5							
10	r4 r4 r4 r4 r4							
11	r6 r6 r6 r6 r6							

0:  $S' \rightarrow E$   
 1:  $E \rightarrow aA$   
 2:  $E \rightarrow bB$   
 3:  $A \rightarrow cA$   
 4:  $A \rightarrow d$   
 5:  $B \rightarrow cB$   
 6:  $B \rightarrow d$



根据分析表进行句子识别的部分与“2.3.4 LR分析示例”的内容一致，此处不再赘述。

## 2.5 SLR(1) 文法

### 2.5.1 引入原因

- LR(0) 出错原因
  - LR(0) 方法不够强，容易出现“移进-归约”冲突与“归约-归约”冲突
  - 即 LR(0) 分析表构造中，出现冲突

► I<sub>1</sub>、I<sub>2</sub>和I<sub>9</sub>都含有“移进 - 归约”冲突

$$I_1: S' \rightarrow E^\bullet \\ E \rightarrow E^\bullet + T$$

$$I_2: E \rightarrow T^\bullet \\ T \rightarrow T^\bullet * F$$

$$I_9: E \rightarrow E + T^\bullet \\ T \rightarrow T^\bullet * F$$

## 2.5.2 SLR(1) 解决冲突

- 引入 SLR(1) 文法进行冲突解决
    - 简单说，就是加入对 FOLLOW 集的考虑，减少冲突
- 假定LR(0)规范族的一个项目集I={A<sub>1</sub>→α•a<sub>1</sub>β<sub>1</sub>, A<sub>2</sub>→α•a<sub>2</sub>β<sub>2</sub>, ..., A<sub>m</sub>→α•a<sub>m</sub>β<sub>m</sub>, B<sub>1</sub>→α•, B<sub>2</sub>→α•, ..., B<sub>n</sub>→α•} 如果集合{a<sub>1</sub>, ..., a<sub>m</sub>}, FOLLOW(B<sub>1</sub>), ..., FOLLOW(B<sub>n</sub>)两两不相交(包括不得有两个FOLLOW集合有#)，则当状态I面临任何输入符号a时：
1. 若a是某个a<sub>i</sub>, i=1,2,...,m, 则移进;
  2. 若a∈FOLLOW(B<sub>i</sub>), i=1,2,...,n, 则用产生式B<sub>i</sub>→α进行归约;
  3. 此外，报错。
- SLR(1)解决办法：S-Simple, 1-最多向前看一个单词

## 2.5.3 SLR(1) 分析表

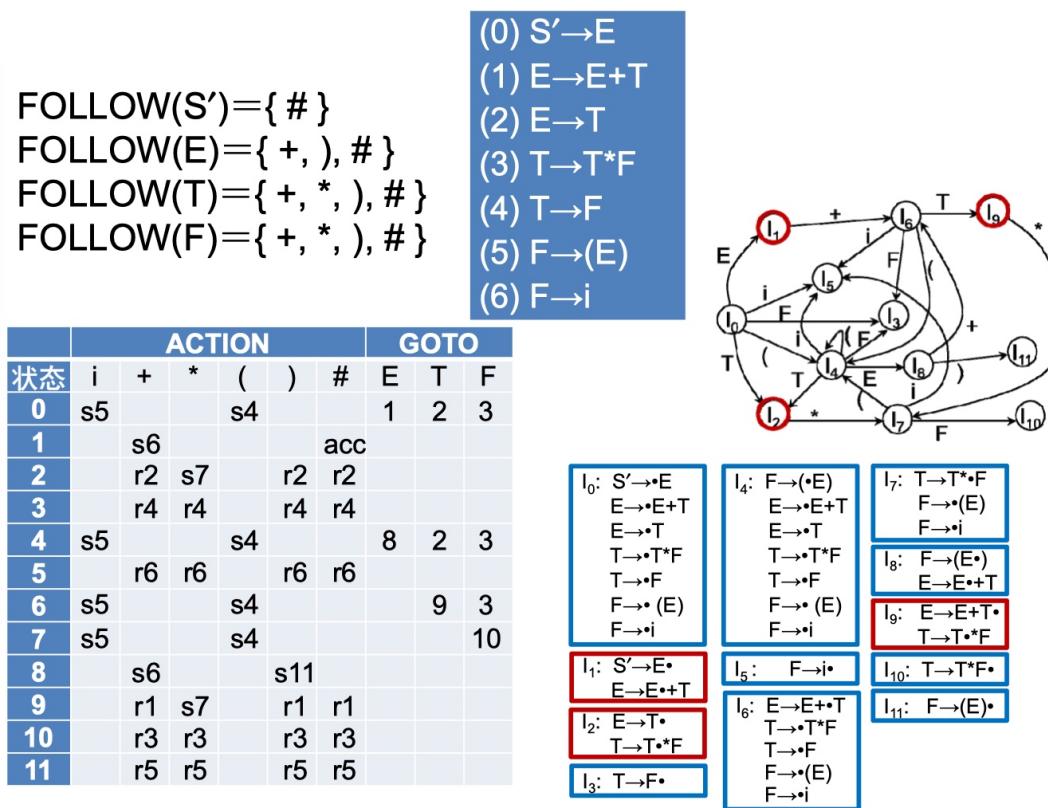
- 构造算法
  - SLR(1) 在分析表构造时只需将 FOLLOW 集考虑进去

# SLR(1)分析表的ACTION和GOTO子表构造

1. 若项目  $A \rightarrow \alpha \cdot a\beta$  属于  $I_k$  且  $GO(I_k, a) = I_j$ ,  $a$  为终结符, 则置  $ACTION[k, a]$  “ $s_j$ ”;
2. 若项目  $A \rightarrow \alpha \cdot$  属于  $I_k$ , 那么, 对任何终结符  $a \in FOLLOW(A)$ , 置  $ACTION[k, a]$  为 “ $r_j$ ”; 其中, 假定  $A \rightarrow \alpha$  为文法  $G'$  的第  $j$  个产生式;
3. 若项目  $S' \rightarrow S \cdot$  属于  $I_k$ , 则置  $ACTION[k, #]$  为 “acc”;
4. 若  $GO(I_k, A) = I_j$ ,  $A$  为非终结符, 则置  $GOTO[k, A] = j$ ;
5. 分析表中凡不能用规则 1 至 4 填入信息的空白格均置上 “报错标志”。

按上述方法构造出的 ACTION 和 GOTO 表如果不含多重入口, 则称该文法为 SLR(1) 文法。

- 构造示例



## 2.6 LR(1) 文法

### 2.6.1 引入原因

- 根本原因

- FOLLOW 集提供的信息太泛，其中有些字符其实并不会出现
- 因此我们需要更加精确的文法来构造分析表

- ▶ SLR在方法中，如果项目集 $I_i$ 含项目 $A \rightarrow \alpha^*$ 而且下一输入符号 $a \in FOLLOW(A)$ ，则状态 $i$ 面临 $a$ 时，可选用“用 $A \rightarrow \alpha$ 归约”动作
- ▶ 但在有些情况下，当状态 $i$ 显现于栈顶时，当前单词是 $a$ ，栈里的活前缀 $\beta\alpha$ 未必允许把 $\alpha$ 归约为 $A$ ，因为可能根本就不存在一个形如“ $\beta A a$ ”的规范句型
- ▶ 在这种情况下，用“ $A \rightarrow \alpha$ ”归约不一定合适
- ▶ FOLLOW集合提供的信息太泛

$$FOLLOW(A) = \{a \mid S \xrightarrow{*} \dots Aa \dots, a \in V_T\}$$

## 2.6.2 LR(k) 项目

- 扩展 LR(0) 项目，引入展望串

- ▶ LR(k)项目：扩展LR(0)项目，附带有k个终结符  
 $[A \rightarrow \alpha^* \beta, a_1 a_2 \dots a_k]$   
 $a_1 a_2 \dots a_k$  称为向前搜索字符串(或展望串)。
- ▶ 归约项目 $[A \rightarrow \alpha^*, a_1 a_2 \dots a_k]$ 的意义：当它所属的状态呈现在栈顶且后续的k个输入符号为 $a_1 a_2 \dots a_k$ 时，才可以把栈顶上的 $\alpha$ 归约为 $A$
- ▶ 对于任何移进或待约项目 $[A \rightarrow \alpha^* \beta, a_1 a_2 \dots a_k]$ ,  $\beta \neq \epsilon$ ，搜索字符串 $a_1 a_2 \dots a_k$ 没有直接作用

## 2.6.3 DFA 构造

- 项目集的闭包

► 假定I是文法G'的任一项目集，定义和构造I的闭包CLOSURE(I)如下：

1. I的任何项目都属于CLOSURE(I)。
2. 若项目 $[A \rightarrow \alpha \cdot B\beta, a]$ 属于CLOSURE(I)， $B \rightarrow \xi$ 是一个产生式，那么，对于FIRST( $\beta a$ ) 中的每个终结符b，如果 $[B \rightarrow \cdot \xi, b]$ 原来不在CLOSURE(I)中，则把它加进去。
3. 重复执行步骤2，直至CLOSURE(I)不再增大为止。

- 项目集的转换函数

► 令I是一个项目集，X是一个文法符号，函数GO(I, X)定义为：

$$GO(I, X) = CLOSURE(J)$$

其中

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

#### 2.6.4 LR(1) 分析表

- 构造算法

# LR(1)分析表的ACTION和GOTO子表构造

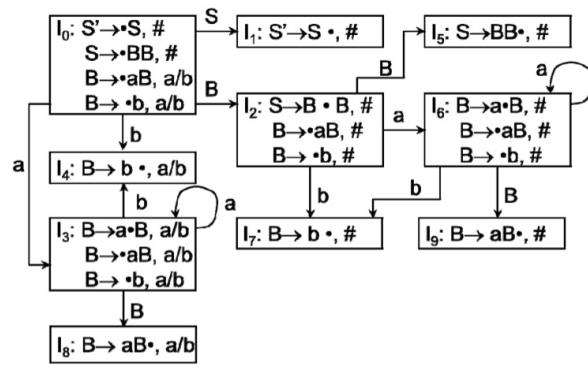
1. 若项目  $[A \rightarrow \alpha \cdot a\beta, b]$  属于  $I_k$  且  $GO(I_k, a) = I_j$ ,  $a$  为终结符, 则置  $ACTION[k, a]$  为 “ $s_j$ ”。
2. 若项目  $[A \rightarrow \alpha \cdot, a]$  属于  $I_k$ , 则置  $ACTION[k, a]$  为 “ $r_j$ ”; 其中假定  $A \rightarrow \alpha$  为文法  $G'$  的第  $j$  个产生式。
3. 若项目  $[S' \rightarrow S \cdot, \#]$  属于  $I_k$ , 则置  $ACTION[k, \#]$  为 “acc”。
4. 若  $GO(I_k, A) = I_j$ , 则置  $GOTO[k, A] = j$ 。
5. 分析表中凡不能用规则1至4填入信息的空白栏均填上 “出错标志”。

- 构造示例

## 示例：LR(1)分析表的构造

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

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



## 2.7 LALR(1) 文法

- 文法性质

- 项目集个数 = SLR(1)
- 通过合并 LR(1) 中的同心集得到, 如果 LR(1) 中原本就没有“移进–归约”冲突, 则 LALR 中也不会有, 但可能会出现“归约–归约”冲突
- 文法目的: 识别能力比 SLR(1) 更强, 但项目集个数一致

- 同心集合并

- 同心集定义

- 两个集合忽略展望串时，完全相同

- 同心集合并举例

- I5 与 I9 同心，合并后为 I59

```
I5: A -> d·, a
```

```
    B -> d·, c
```

```
I9: A -> d·, c
```

```
    B -> d·, a
```

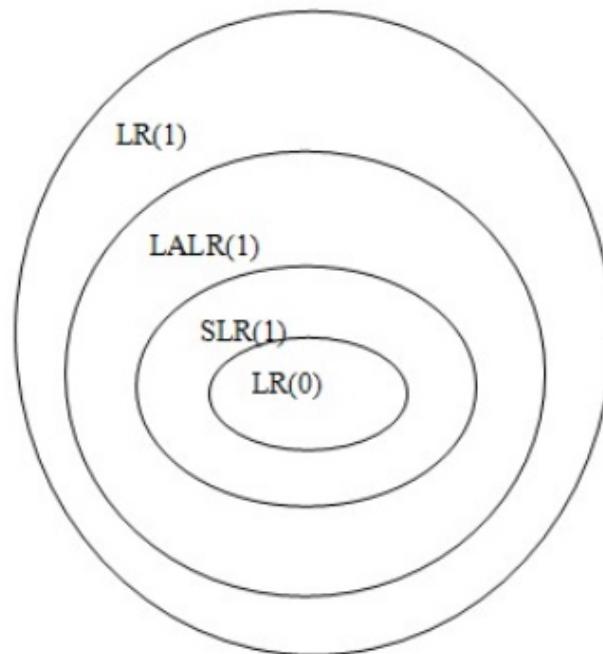
```
I59: A -> d·, a/c
```

```
    B -> d·, a/c
```

## 2.8 LR 文法总结

- 文法能力

- LR(0)、SLR(1)、LALR(1)、LR(1) 文法，前者为后者的真子集



- LR(1) 文法

- 具有规范的 LR(1) 分析表的文法称为 LR(1) 文法
  - 使用 LR(1) 分析表的分析器叫做一个规范的 LR 分析器

# 一、概述

属性文法，也称为属性翻译文法，以“上下文无关文法”为基础，扩充了以下两部分内容：

- 每个文法符号（终结符或非终结符）有“值”（属性）
- 每个产生式有一组属性的语义规则，对属性进行计算和传递

产生式	语义规则
$L \rightarrow E_n$	$\text{print}(E.\text{val})$
$E \rightarrow E_1 + T$	$E.\text{val} := E_1.\text{val} + T.\text{val}$
$E \rightarrow T$	$E.\text{val} := T.\text{val}$
$T \rightarrow T_1 * F$	$T.\text{val} := T_1.\text{val} * F.\text{val}$
$T \rightarrow F$	$T.\text{val} := F.\text{val}$
$F \rightarrow (E)$	$F.\text{val} := E.\text{val}$
$F \rightarrow \text{digit}$	$F.\text{val} := \text{digit}.lexval$

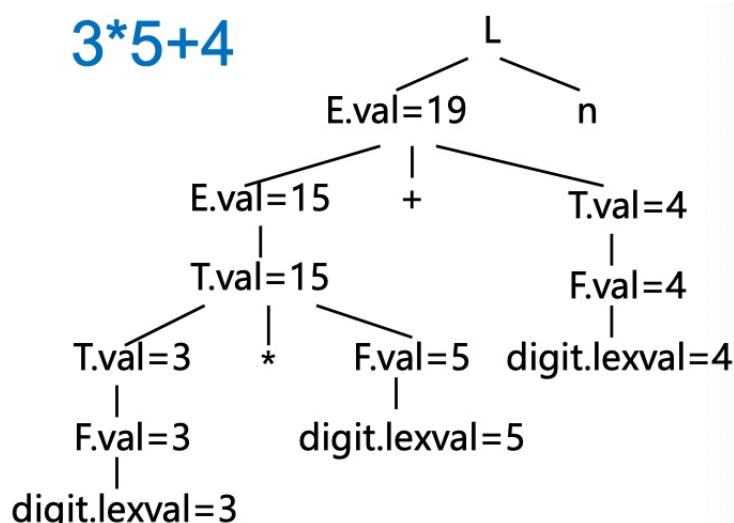
属性中有两类属性，一种是综合属性，另一种是继承属性。

## 1.1 综合属性

- 自下而上传递信息
- 语法规则：产生式右部确定左部

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

- 语法树：子节点确定父节点

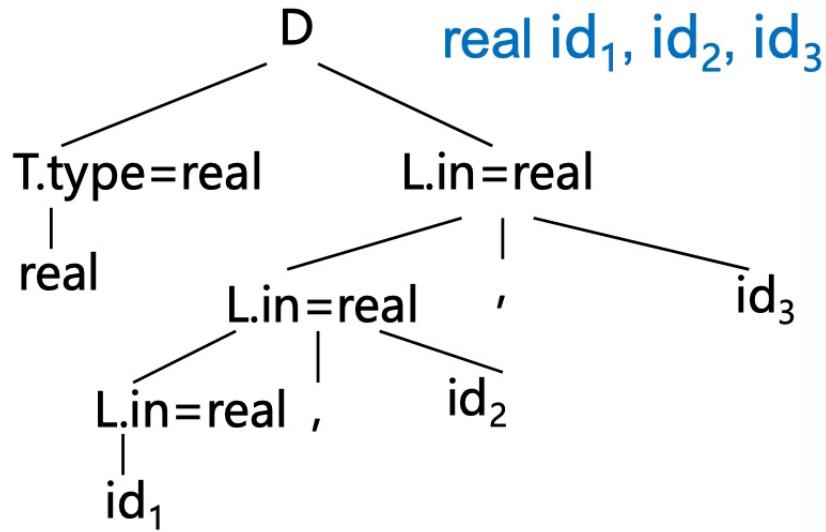


## 1.2 继承属性

- 自上而下传递信息
- 语法规则：产生式左部确定右部

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

- 语法树：（父节点 + 兄弟节点）确定（子节点）



## 1.3 语义规则

产生式  $A \rightarrow \alpha$  对应的语义规则如下：

$$b := f(c_1, c_2, \dots, c_k)$$

其中共有两种情况：

- b 是 A 的综合属性， $c_i$  是产生式右边文法符号的属性
- b 是产生式右边文法符号的继承属性， $c_i$  是 A 或产生式右边文法符号的属性

由此可以如下结论：

- 终结符只有综合属性，且由词法分析器提供
- 非终结符可以有综合属性、继承属性
- 文法开始符号的所有继承属性作为属性计算前的初始值

语义规则功能：

- 属性计算、静态语义检查
- 符号表操作、代码生成

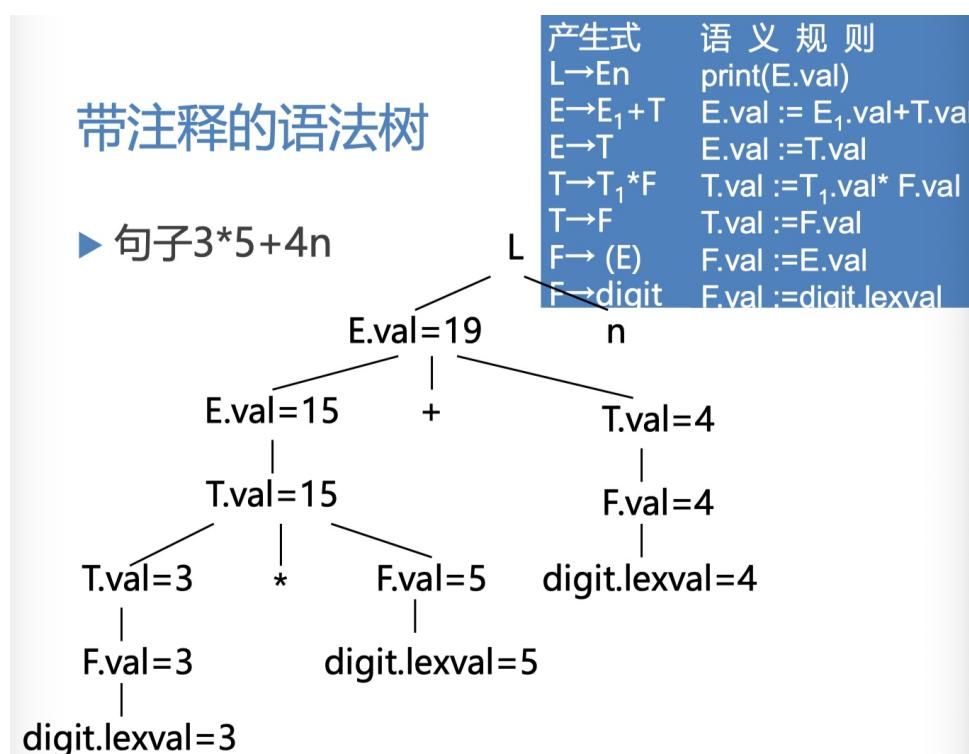
► 考虑非终结符A, B和C, 其中, A有一个继承属性a和一个综合属性b, B有综合属性c, C有继承属性d。产生式A→BC不可能有规则

- A. C.d:=B.c+1
- B. A.b:=B.c+C.d
- C. B.c := A.a

## 二、带注释的语法树

### 2.1 S-属性文法

- 语法树中，一个结点的综合属性的值由其子结点和它本身的属性值确定
- 使用自底向上的办法在每一个结点处使用语义规则计算综合属性的值
- 仅使用综合属性的属性文法为 **S-属性文法**



- 分析过程

### ► 句子 $3 * 5 + 4n$

产生式	代码段
$L \rightarrow En$	<code>print(val[top])</code>
$E \rightarrow E_1 + T$	$val[ntop] := val[top-2] + val[top]$
$E \rightarrow T$	
$T \rightarrow T_1 * F$	$val[ntop] := val[top-2] * val[top]$
$T \rightarrow F$	
$F \rightarrow (E)$	$val[ntop] := val[top-1]$
$F \rightarrow digit$	

步骤	状态栈	符号栈	属性(val)	输入串
0	0	#	-	$3 * 5 + 4n$
1	5	#3	-3	$* 5 + 4n$
2	3	#F	-3	$* 5 + 4n$
3	2	#T	-3	$* 5 + 4n$
4	27	#T*	-3-	$5 + 4n$
5	275	#T*5	-3-5	$+ 4n$
6	2710	#T*F	-3-5	$+ 4n$
7	2	#T	-15	$+ 4n$
8	1	#E	-15	$+ 4n$
9	16	#E+	-15-	$4n$
10	165	#E+4	-15-4	n
11	163	#E+F	-15-4	n
11	169	#E+T	-15-4	n
12	1	#E	-19	n
13		#En	-19-	
14		#L	-19	

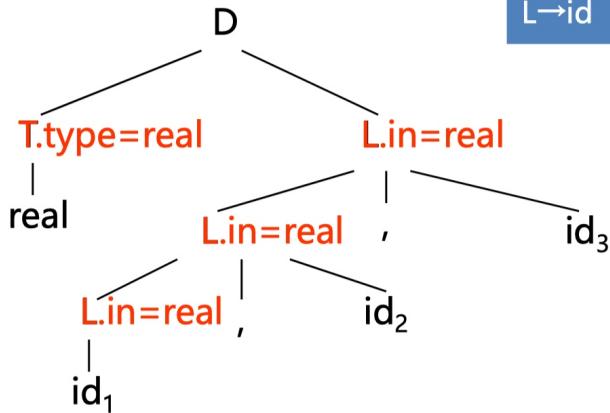
## 2.2 L-属性文法

语法树中，结点的继承属性由其父结点、其兄弟结点和其本身的某些属性确定。

- 继承属性，常用于表示上下文依赖关系

### 带注释的语法树

#### ► 句子 $real \ id_1, \ id_2, \ id_3$



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

name	type
...	...
$id_3$	real
$id_2$	real
$id_1$	real

符号表

- 文法定义

- ▶ 一个属性文法称为**L-属性文法**, 如果对于每个产生式 $A \rightarrow X_1 X_2 \dots X_n$ , 其每个语义规则中的每个属性或者是**综合属性**, 或者是 $X_i (1 \leq i \leq n)$ 的一个**继承属性**且这个继承属性仅依赖于:
  - (1) 产生式中 $X_i$ 左边符号 $X_1, X_2, \dots, X_{i-1}$ 的属性
  - (2)  $A$ 的继承属性
- ▶ **S-属性文法**一定是**L-属性文法**

### 三、属性计算

#### 3.1 概述

语义规则的计算功能如下:

- 产生代码
- 在符号表中存放信息
- 给出错误信息
- 执行任何其它动作

「总结」对输入串的翻译就是根据语义规则进行计算。

#### 3.2 语法制导翻译法

语法制导翻译法: 由源程序的**语法结构所驱动**的处理办法

**输入串 → 语法树 → 按照语义规则计算属性**

基于属性文法的处理方法有:

- 依赖图
- 树遍历
- 一遍扫描

#### 3.3 依赖图

「功能」描述一棵语法树中的结点的继承属性和综合属性之间的相互依赖关系

##### 3.3.1 构建算法

```

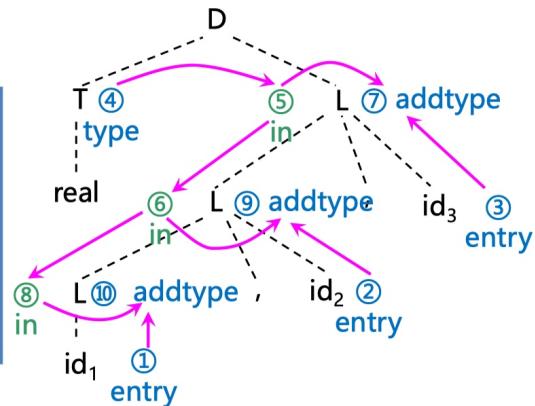
for 语法树中每一结点n do
    for 结点n的文法符号的每一个属性a do
        为a在依赖图中建立一个结点;
for 语法树中每一个结点n do
    for 结点n所用产生式对应的每一个语义规则
        b:=f(c1,c2,...,ck) do
            for i:=1 to k do
                从ci结点到b结点构造一条有向边;

```

### 3.3.2 依赖图举例

► 句子real id<sub>1</sub>, id<sub>2</sub>, id<sub>3</sub>的依赖图

产生式	语义规则
D→TL	L.in := T.type
T→int	T.type := integer
T→real	T.type := real
L→L <sub>1</sub> , id	L <sub>1</sub> .in := L.in addtype(id.entry, L.in)
L→id	addtype(id.entry, L.in)



- 如果一属性文法不存在属性之间的循环依赖关系，则该文法为**良定义的**
- 一个依赖图的任何**拓扑排序**都给出一个语法树中结点的语义规则计算的有效顺序

### 3.3.3 属性的计算次序

- 基础文法用于建立输入符号串的语法分析树
- 根据语义规则建立依赖图
- 根据依赖图的拓扑排序，得到计算语义规则的顺序

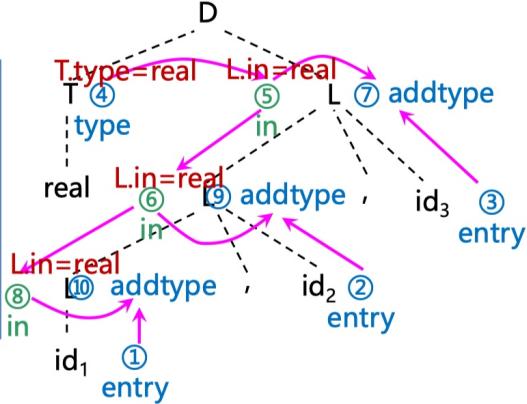
输入串 → 语法树 → 依赖图 → 语义规则计算次序

## 依赖图示例

► 句子real id<sub>1</sub>, id<sub>2</sub>, id<sub>3</sub>的依赖图

name	type
...	...
id <sub>3</sub>	real
id <sub>2</sub>	real
id <sub>1</sub>	real

产生式	语义规则
D→TL	L.in := T.type
T→int	T.type := integer
T→real	T.type := real
L→L <sub>1</sub> , id	L <sub>1</sub> .in := L.in addtype(id.entry, L.in)
L→id	addtype(id.entry, L.in)



## 3.4 树遍历

「具体方法」通过树遍历的方法计算属性的值

- 输入时，树中已有开始符号的继承属性和终结符的综合属性
- 深度优化，从左到右的遍历

输入串 → 语法树 → 遍历语法树  
计算属性

### 3.4.1 树遍历算法

## 树遍历算法

```
While 还有未被计算的属性 do  
  VisitNode(S) /*S是开始符号*/
```

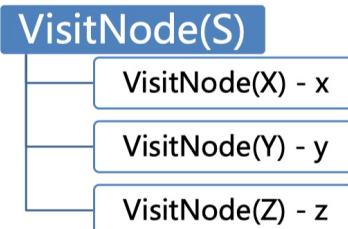
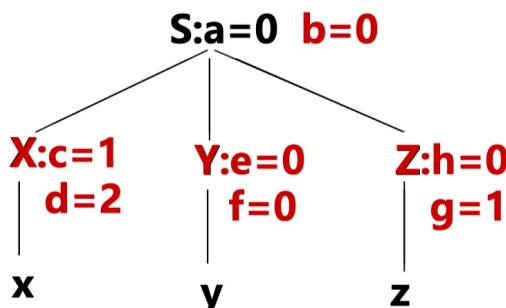
- 计算思维的典型方法--递归
  - 问题的解决又依赖于类似问题的解决，只不过后者的复杂程度或规模较原来的问题更小
  - 一旦将问题的复杂程度和规模化简到足够小时，问题的解法其实非常简单

```
procedure VisitNode (N:Node);  
begin  
  if N是一个非终结符 then /*假设其产生式为N→X1...Xm */  
    for i :=1 to m do  
      if Xi∈VN then /*即Xi是非终结符*/  
        begin  
          计算Xi的所有能够计算的继承属性;  
          VisitNode (Xi)  
        end;  
  计算N的所有能够计算的综合属性  
end
```

### 3.4.2 树遍历举例

#### 树遍历算法示例

- ▶ 输入串为xyz
- ▶ 假设S.a的初始值为0



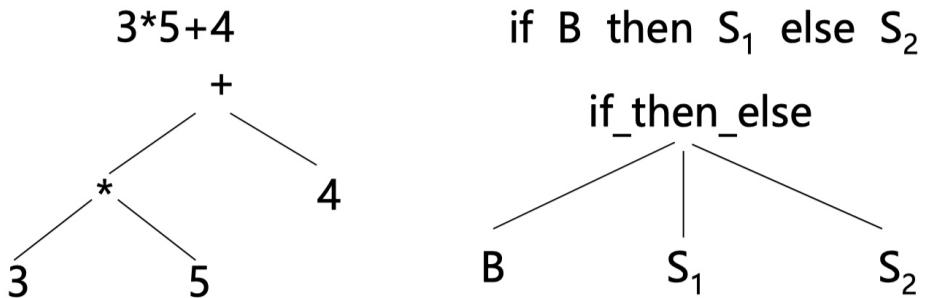
产生式	语义规则
S→XYZ	Z.h := S.a
X→x	X.c := Z.g
Y→y	S.b := X.d - 2
Z→z	Y.e := S.b
	X.d := 2*X.c
	Y.f := Y.e*3
	Z.g := Z.h+1

### 3.5 一遍扫描

- 在语法分析的同时计算属性值
- 适用于 S-属性文法 / L-属性文法

#### 3.5.1 抽象语法树

- ▶ 抽象语法树(Abstract Syntax Tree, AST), 在语法树中去掉那些对翻译不必要的信息, 从而获得更有效的源程序中间表示



### 3.5.2 建立抽象语法树

- ▶ **mknod(op, left, right)** 建立一个运算符号结点, 标号是op, 两个域left和right分别指向左子树和右子树
- ▶ **mkleaf(id, entry)** 建立一个标识符结点, 标号为id, 一个域entry指向标识符在符号表中的入口
- ▶ **mkleaf(num, val)** 建立一个数结点, 标号为num, 一个域val用于存放数的值

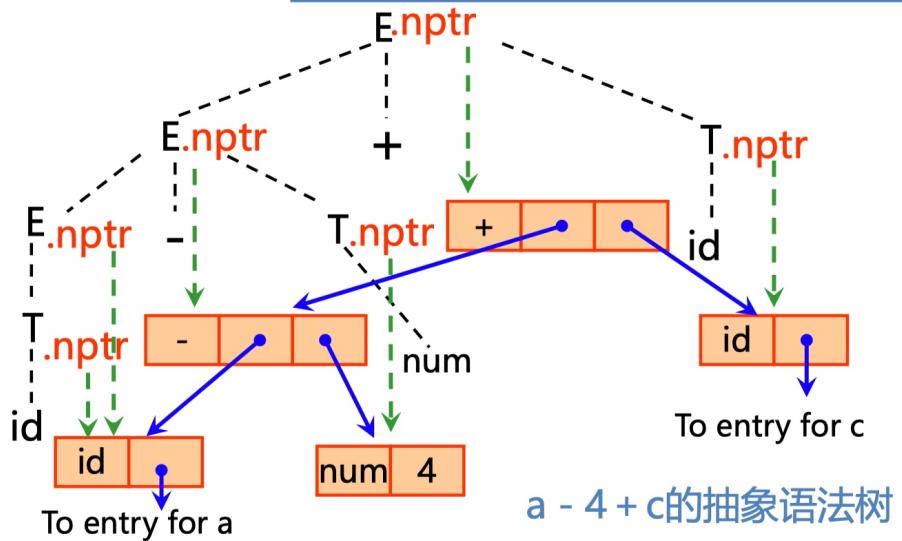
### 3.5.3 一遍扫描举例

## 建立抽象语法树的语义规则

### 产生式      语义规则

$E \rightarrow E_1 + T$	$E.\text{nptr} := \text{mknod}('+' , E_1.\text{nptr}, T.\text{nptr})$
$E \rightarrow E_1 - T$	$E.\text{nptr} := \text{mknod}('-', E_1.\text{nptr}, T.\text{nptr})$
$E \rightarrow T$	$E.\text{nptr} := T.\text{nptr}$
$T \rightarrow (E)$	$T.\text{nptr} := E.\text{nptr}$
$T \rightarrow \text{id}$	$T.\text{nptr} := \text{mkleaf}(\text{id}, \text{id}.entry)$
$T \rightarrow \text{num}$	$T.\text{nptr} := \text{mkleaf}(\text{num}, \text{num}.val)$

$E \rightarrow E_1 + T$	$E.nptr := \text{mknode}( '+', E_1.nptr, T.nptr )$
$E \rightarrow E_1 - T$	$E.nptr := \text{mknode}( ' - ', E_1.nptr, T.nptr )$
$E \rightarrow T$	$E.nptr := T.nptr$
$T \rightarrow id$	$T.nptr := \text{mkleaf} ( id, id.entry )$
$T \rightarrow num$	$T.nptr := \text{mkleaf} ( num, num.val )$



## 语义分析和中间代码生成

### 一、中间语言

#### 1.1 概述

「特点」

- 独立于机器
- 复杂性界于源语言和目标语言之间

「优点」

- 使编译程序的结构在逻辑上更为简单明确
- 便于进行与机器无关的代码优化工作
- 易于移植



「常用的中间语言」

- 后缀式：逆波兰表示
- 图表示：抽象语法树（AST）、有向无环图（DAG）

- 三地址代码：四元式、三元式、间接三元式

## 1.2 后缀式

### 1.2.1 定义

- ▶ 如果E是一个变量或常量，则E的后缀式是E自身。
- ▶ 如果E是 $E_1 \text{ op } E_2$ 形式的表达式，其中op是任何二元操作符，则E的后缀式为 $E'_1 E'_2 \text{ op}$ ，其中 $E'_1$ 和 $E'_2$ 分别为 $E_1$ 和 $E_2$ 的后缀式。
- ▶ 如果E是 $(E_1)$ 形式的表达式，则 $E_1$ 的后缀式就是E的后缀式。

### 1.2.2 计算方式

- ▶ 后缀式表示法不用括号
  - ▶ 只要知道每个算符的目数，对于后缀式，不论从哪一端进行扫描，都能对它进行无歧义地分解。
- ▶ 后缀式的计算
  - ▶ 用一个栈实现
  - ▶ 自左至右扫描后缀式，每碰到运算量就把它推进栈。每碰到k目运算符就把它作用于栈顶的k个项，并用运算结果代替这k个项。

### 1.2.3 表达式 => 后缀式

- 表达式 => 后缀式的属性文法

产生式	语义规则
$E \rightarrow E_1 \text{ op } E_2$	$E.\text{code} := E_1.\text{code} \parallel E_2.\text{code} \parallel \text{op}$
$E \rightarrow (E_1)$	$E.\text{code} := E_1.\text{code}$
$E \rightarrow \text{id}$	$E.\text{code} := \text{id}$

- ▶  $E.\text{code}$ 表示E后缀形式
- ▶ op表示任意二元操作符
- ▶ “||” 表示后缀形式的连接

- 中缀表达式 => 后缀式

▶ 数组POST存放后缀式：k为下标，初值为1

产生式	语义规则
$E \rightarrow E_1 op E_2$	$E.code := E_1.code    E_2.code    op$
$E \rightarrow (E_1)$	$E.code := E_1.code$
产生式	程序段
$E \rightarrow E_1 op E_2$	$\{ POST[k]:=op; k:=k+1 \}$
$E \rightarrow (E_1)$	$\{ \}$

a+b+c的分析和翻译： POST 

1	2	3	4	5	...
a	b	+	c	+	...

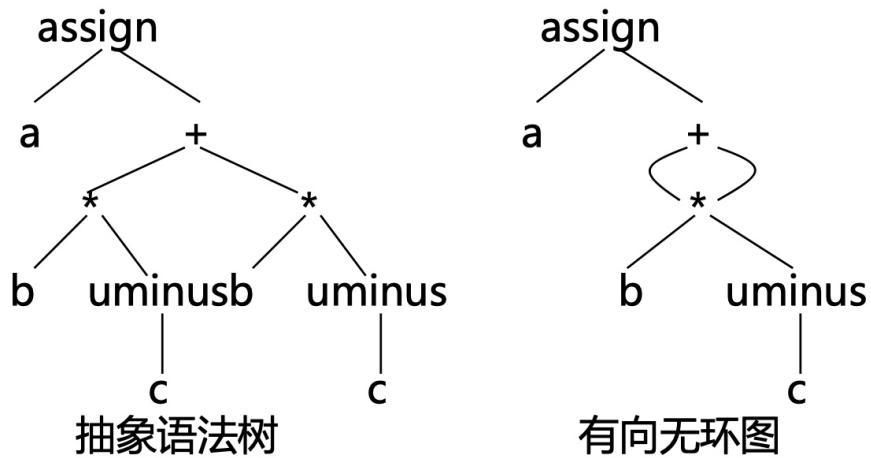
## 1.3 图表示法

### 1.3.1 有向无环图

- 对表达式中的每个子表达式，DAG中都有一个结点
- 一个内部结点代表一个操作符，它的孩子代表操作数
- 在一个 DAG 中代表公共子表达式的结点具有多个父节点

### 1.3.2 举例

▶  $a := b * (-c) + b * (-c)$  的图表示法



## 1.4 三地址代码

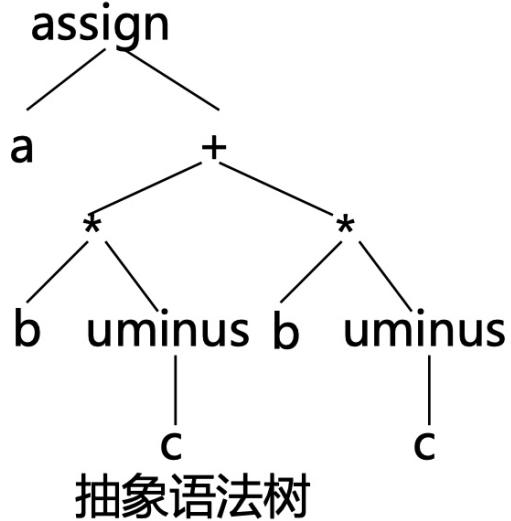
#### 1.4.1 概述

- 「形式」  $x := y \ op \ z$
- 「理解」 三地址代码可以看成是抽象语法树或有向无环图的一种线性表示
- 「抽象语法树 vs. 三地址代码」

►  $a := b^*(-c) + b^*(-c)$  的图表示法

抽象语法树对应的三地址代码：

```
T1 := -c
T2 := b * T1
T3 := - c
T4 := b * T3
T5 := T2 + T4
a := T5
```



- 「有向无环图 vs. 三地址代码」

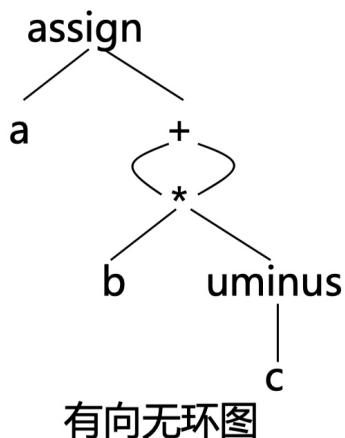
►  $a := b^*(-c) + b^*(-c)$  的图表示法

抽象语法树对应的三地址代码：

```
T1 := -c
T2 := b * T1
T3 := - c
T4 := b * T3
T5 := T2 + T4
a := T5
```

有向无环图对应的三地址代码：

```
T1 := - c
T2 := b * T1
T5 := T2 + T2
a := T5
```



- 「三地址语句的种类」

- ▶  $x := y \text{ op } z$
- ▶  $x := \text{op } y$
- ▶  $x := y$
- ▶ **goto L**
- ▶ if  $x \text{ relop } y \text{ goto } L$  或 if  $a \text{ goto } L$
- ▶ 传参、转子： param  $x$ 、 call  $p,n$
- ▶ 返回语句： return  $y$
- ▶ 索引赋值：  $x := y[i]$ 、  $x[i] := y$
- ▶ 地址和指针赋值：  $x := \&y$ 、  $x := *y$ 、  $*x := y$

#### 1.4.2 四元式

- ▶ 一个带有四个域的记录结构，这四个域分别称为  $op$ ,  $arg1$ ,  $arg2$  及  $result$
- ▶  $a := b * (-c) + b * (-c)$  的四元式形式

序号	OP	arg1	arg2	result
(0)	<b>uminus</b>	<b>c</b>	-	<b>T<sub>1</sub></b>
(1)	*	<b>b</b>	<b>T<sub>1</sub></b>	<b>T<sub>2</sub></b>
(2)	<b>uminus</b>	<b>c</b>	-	<b>T<sub>3</sub></b>
(3)	*	<b>b</b>	<b>T<sub>3</sub></b>	<b>T<sub>4</sub></b>
(4)	+	<b>T<sub>2</sub></b>	<b>T<sub>4</sub></b>	<b>T<sub>5</sub></b>
(5)	<b>:=</b>	<b>T<sub>5</sub></b>	-	<b>a</b>

#### 1.4.3 三元式

在四元式基础上优化，用编号代表结果。

- ▶ 用三个域表示：op、arg1和arg2
- ▶ 计算结果引用：引用计算该值的语句的位置
- ▶  $a := b * (-c) + b * (-c)$  的三元式形式

序号	OP	arg1	arg2
(0)	uminus	c	-
(1)	*	b	(0)
(2)	uminus	c	-
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

▶  $x[i] := y$

▶  $x := y[i]$

序号	OP	arg1	arg2
(0)	[]=	x	i
(1)	:=	(0)	y

序号	OP	arg1	arg2
(0)	=[]	y	i
(1)	:=	x	(0)

#### 1.4.4 间接三元式

在三元式的基础上再度优化，使得式子发生变化时，可变性更强。

- 「定义」 间接三元式 = 三元式表 + 间接码表
  - 间接码表：一张指示器表，按运算的先后次序列出有关三元式在三元式表中的位置
- 「优点」 方便优化，节省空间

▶ 语句

$X := (A + B) * C;$

$Y := D \uparrow (A + B)$

的间接三元式

间接码表

(0)  
(1)  
(2)  
(0)  
(3)  
(4)

序号	OP	arg1	arg2
(0)	+	A	B
(1)	*	(0)	C
(2)	:=	X	(1)
(3)	$\uparrow$	D	(0)
(4)	:=	Y	(3)

## 二、赋值语句的翻译

### 2.1 赋值语句的属性文法

#### 2.1.1 概述

- 「赋值语句形式」  $id := E$
- 「赋值语句意义」
  - 对表达式  $E$  求值并置于变量  $T$  中
  - $id.place := T$
- 「赋值语句 => 三地址代码的 S-属性文法」
  - ▶ 非终结符号  $S$  有综合属性  $S.code$ , 它代表赋值语句  $S$  的三地址代码
  - ▶ 非终结符号  $E$  有两个属性
    - ▶  $E.place$  表示存放  $E$  值的单元的名字(地址)
    - ▶  $E.code$  表示对  $E$  求值的三地址语句序列
  - ▶ 函数  $newtemp$  的功能是, 每次调用它时, 将返回一个不同的临时变量名字, 如  $T_1, T_2, \dots$

#### 2.1.2 语义规则

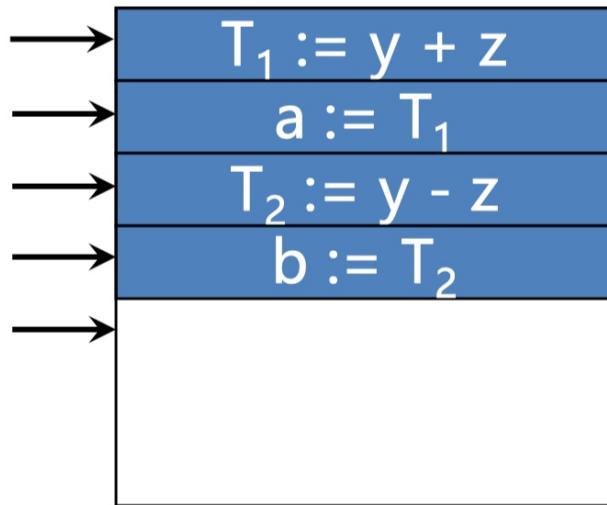
- 赋值语句生成三地址代码的 S-属性文法
- $gen()$  函数用于生成语句的三地址代码

产生式	语义规则
$S \rightarrow id := E$	$S.code := E.code \parallel gen(id.place := E.place)$
$E \rightarrow E_1 + E_2$	$E.place := newtemp;$ $E.code := E_1.code \parallel E_2.code \parallel gen(E.place := E_1.place + E_2.place)$
$E \rightarrow E_1 * E_2$	$E.place := newtemp;$ $E.code := E_1.code \parallel E_2.code \parallel gen(E.place := E_1.place * E_2.place)$
$E \rightarrow -E_1$	$E.place := newtemp;$ $E.code := E_1.code \parallel gen(E.place := 'uminus' E_1.place)$
$E \rightarrow (E_1)$	$E.place := E_1.place;$ $E.code := E_1.code$
$E \rightarrow id$	$E.place := id.place;$ $E.code = ''$

### 2.2 赋值语句的翻译模式

- 产生赋值语句三地址代码的翻译模式

## ► 过程emit将三地址代码送到输出文件中



$S \rightarrow id := E$	{ p:=lookup(id.name); if p $\neq$ nil then emit(p ':=' E.place) else error }
$E \rightarrow E_1 + E_2$	{ E.place:=newtemp; emit(E.place ':=' E <sub>1</sub> .place '+' E <sub>2</sub> .place)}
$E \rightarrow E_1 * E_2$	{ E.place:=newtemp; emit(E.place ':=' E <sub>1</sub> .place '*' E <sub>2</sub> .place)}

产生式	语义规则
$S \rightarrow id := E$	S.code:=E.code    gen(id.place ':=' E.place)
$E \rightarrow E_1 + E_2$	E.place:=newtemp; E.code:=E <sub>1</sub> .code    E <sub>2</sub> .code    gen(E.place ':=' E <sub>1</sub> .place '+' E <sub>2</sub> .place)
$E \rightarrow E_1 * E_2$	E.place:=newtemp; E.code:=E <sub>1</sub> .code    E <sub>2</sub> .code    gen(E.place ':=' E <sub>1</sub> .place '*' E <sub>2</sub> .place)

$E \rightarrow -E_1$	{ E.place:=newtemp; emit(E.place ':=' 'uminus' E <sub>1</sub> .place) }
$E \rightarrow (E_1)$	{ E.place:=E <sub>1</sub> .place }
$E \rightarrow id$	{ p:=lookup(id.name); if p $\neq$ nil then E.place:=p else error }

产生式	语义规则
$E \rightarrow -E_1$	E.place:=newtemp; E.code:=E <sub>1</sub> .code    gen(E.place ':=' 'uminus' E <sub>1</sub> .place)
$E \rightarrow (E_1)$	E.place:=E <sub>1</sub> .place; E.code:=E <sub>1</sub> .code
$E \rightarrow id$	E.place:=id.place; E.code=''

### 三、数组元素引用的翻译

#### 3.1 数组元素地址计算

##### 3.1.1 计算公式

- 设  $A$  为  $n$  维数组，按行存放，每个元素宽度为  $w$ 
  - $low_i$  为第  $i$  维的下界
  - $up_i$  为第  $i$  维的上界
  - $n_i$  为第  $i$  维可取值的个数 ( $n_i = up_i - low_i + 1$ )
  - $base$  为  $A$  的第一个元素相对地址
- 元素  $A[i_1, i_2, \dots, i_k]$  相对地址公式
  - 可变部分  $V = (((\dots((i_1 n_2 + i_2) n_3 + i_3) \dots) n_k + i_k) * w$
  - 不变部分  $C = base - (((\dots((low_1 n_2 + low_2) n_3 + low_3) \dots) n_k + low_k) * w$
  - 相对地址 =  $V + C$

$$(((\dots i_1 n_2 + i_2) n_3 + i_3) \dots) n_k + i_k) \times w + \\ base - (((\dots (low_1 n_2 + low_2) n_3 + low_3) \dots) n_k + low_k) \times w$$

##### 3.1.2 产生式 / 属性定义

###### • 产生式

► id 出现的地方也允许下面产生式中的 L 出现

$L \rightarrow id [ Elist ] | id$

$Elist \rightarrow Elist, E | E$

为了便于处理，文法改写为

$L \rightarrow Elist ] | id$

$Elist \rightarrow Elist, E | id [ E$

###### • 属性定义

- ▶ 引入下列语义变量或语义过程
  - ▶ Elist.ndim: 下标个数计数器 **记录下标个数**
  - ▶ Elist.place: 保存临时变量的名字, 这些临时变量存放已形成的Elist中的下标表达式计算出来的值
  - ▶ Elist.array: 保存数组名
  - ▶ limit(array, j) : 函数过程, 它给出数组array的第j维的长度

- ▶ 代表变量的非终结符L有两项语义值
  - ▶ L.place
    - ▶ 若L为简单变量i, 指变量i的符号表入口
    - ▶ 若L为下标变量, 指存放**不变部分**的临时变量的名字
  - ▶ L.offset
    - ▶ 若L为简单变量, null
    - ▶ 若L为下标变量, 指存放**可变部分**的临时变量的名字

## 3.2 带数组元素引用的赋值语句翻译模式

此翻译模式与自下而上的语法分析方法结合在一起, 可以一遍扫描就完成语法分析和翻译。

### 3.2.1 赋值语句类别

- (1)  $S \rightarrow L := E$
- (2)  $E \rightarrow E + E$
- (3)  $E \rightarrow (E)$
- (4)  $E \rightarrow L$
- (5)  $L \rightarrow Elist ]$
- (6)  $L \rightarrow id$
- (7)  $Elist \rightarrow Elist, E$
- (8)  $Elist \rightarrow id [ E$

### 3.2.2 赋值语句翻译

(1)  $S \rightarrow L := E$

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

(2)  $E \rightarrow E_1 + E_2$

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

(3)  $E \rightarrow (E_1)$  {E.place:=E<sub>1</sub>.place}

(4)  $E \rightarrow L$

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

(5)  $L \rightarrow Elist ]$

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

(6)  $L \rightarrow id$  { L.place:=id.place; L.offset:=null }

(7) Elist $\rightarrow$  Elist<sub>1</sub>, E

```

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

```

(8) Elist $\rightarrow$  id [ E

```

{ Elist.place:=E.place;
  Elist.ndim:=1;
  Elist.array:=id.place }

```

### 3.3 类型转换

「类型转换的三个功能」

- 检查操作数类型
- 确定结果的类型
- 如果有必要，将整型转变为实型

「类型转换的三地址代码举例」

- ▶  $x := y + i*j$ , 其中x、y为实型; i、j为整型
- ▶ 该赋值句产生的三地址代码为:

```

T1 := i int* j
T3 := inttoreal T1
T2 := y real+ T3
x := T2

```

「类型转换属性文法」

- ▶ 用E.type表示非终结符E的类型属性
- ▶ 产生式 $E \rightarrow E_1 \text{ op } E_2$ 的语义动作中关于E.type的语义规则可定义为：
 

```
{ if E1.type=integer and E2.type=integer
          E.type:=integer
        else E.type:=real }
```

「类型转换的语义动作举例」

```
{ E.place:=newtemp;
if E1.type=integer and E2.type=integer then begin
  emit (E.place ':=' E1.place 'int+' E2.place);
  E.type:=integer
end
else if E1.type=real and E2.type=real then begin
  emit (E.place ':=' E1.place 'real+' E2.place);
  E.type:=real
end

else if E1.type=integer and E2.type=real then begin
  u:=newtemp;
  emit (u ':=' 'inttoreal' E1.place);
  emit (E.place ':=' u 'real+' E2.place);
  E.type:=real
end
else if E1.type=real and E2.type=integer then begin
  u:=newtemp;
  emit (u ':=' 'inttoreal' E2.place);
  emit (E.place ':=' E1.place 'real+' u);
  E.type:=real
end
else E.type:=type_error}
```

## 四、布尔表达式的翻译

### 4.1 布尔表达式概述

## 4.1.1 文法 & 用途

### 「文法」

$$E \rightarrow E \text{ or } E \mid E \text{ and } E \mid \text{not } E \mid (E) \mid i \text{ rop } i \mid i$$

- $\text{rop}$  为关系运算符，包含  $[>, \geq, <, \leq, \neq, ==]$
- $i$  表示单个布尔变量
- 运算优先级：括号 > 条件表达式 > not > and > or

### 「用途」

- 用于逻辑演算，计算逻辑值
- 用于控制语句的条件式

## 4.1.2 布尔表达式两种计算方法

### 「数值表示法」

- 如同计算算术表达式一样，一步步算
- 举例

$$\begin{aligned} & 1 \text{ or } (\text{not } 0 \text{ and } 0) \text{ or } 0 \\ & = 1 \text{ or } (1 \text{ and } 0) \text{ or } 0 \\ & = 1 \text{ or } 0 \text{ or } 0 \\ & = 1 \text{ or } 0 \\ & = 1 \end{aligned}$$

► A or B and C>D 翻译成

(1) ( $>$ , C, D, T1)	(1) $T_1 := C > D$
(2) (and, B, T1, T2)	(2) $T_2 := B \text{ and } T_1$
(3) (or, A, T2, T3)	(3) $T_3 := A \text{ or } T_2$

### 「带优化的翻译法」

- 相比数值表示法，可以减少很多冗余计算，效率更高
- 适合于作为条件表达式的布尔表达式使用
- 举例

- 把A or B解释成 if A then true else B
- 把A and B解释成 if A then B else false
- 把not A解释成 if A then false else true

## 4.2 按数值表示法翻译

### 4.2.1 数值表示法

「a or b and not c」

- $T_1 := \text{not } c$
- $T_2 := b \text{ and } T_1$
- $T_3 := a \text{ or } T_2$

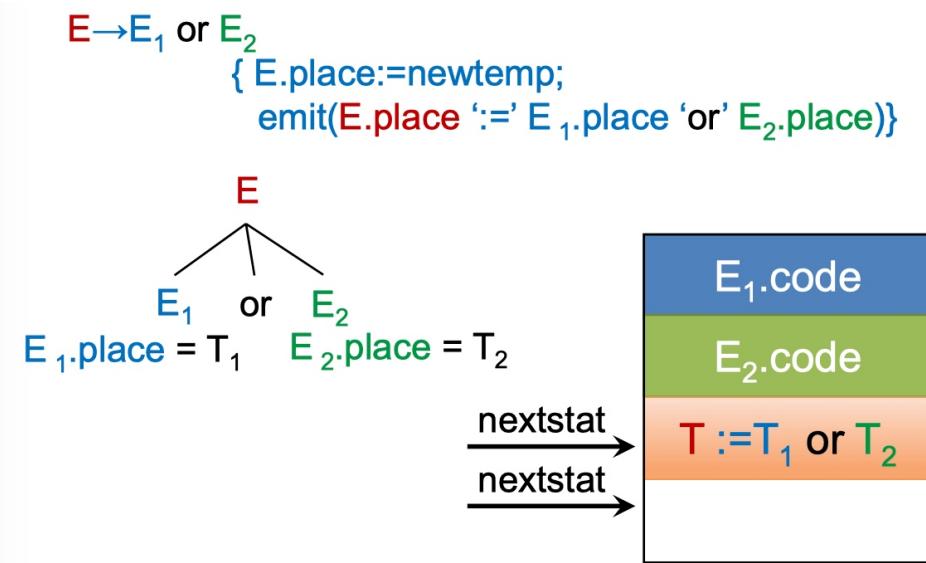
「a < b」

- 等价于「if a < b then 1 else 0」，进行如下翻译
- 100 : if a < b goto 103
- 101 :  $T := 0$
- 102 : goto 104
- 103 :  $T := 1$
- 104 :

#### 4.2.2 翻译模式

「基础说明」

- 过程 emit 将三地址代码送到输出文件中
- nextstat: 输出序列中下一条三地址语句的地址索引
- 过程 emit 每产生一条指令，nextstat 加 1
- 举例



「布尔表达式翻译模式」

$E \rightarrow E_1 \text{ or } E_2$   
{ E.place:=newtemp;  
emit(E.place ':=' E<sub>1</sub>.place 'or' E<sub>2</sub>.place)}

$E \rightarrow E_1 \text{ and } E_2$   
{ E.place:=newtemp;  
emit(E.place ':=' E<sub>1</sub>.place 'and' E<sub>2</sub>.place)}

$E \rightarrow \text{not } E_1$   
{ E.place:=newtemp;  
emit(E.place ':=' 'not' E<sub>1</sub>.place) }

$E \rightarrow (E_1)$  { E.place:=E<sub>1</sub>.place }

$E \rightarrow id_1 \text{ relop } id_2$   
{ E.place:=newtemp;  
emit('if' id<sub>1</sub>.place relop.op id<sub>2</sub>.place 'goto' nextstat+3);  
emit(E.place ':=' '0');  
emit('goto' nextstat+2);  
emit(E.place ':=' '1') }

$E \rightarrow id$   
{ E.place:=id.place }

a<b 翻译成

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

## 「布尔表达式 a<b or c<d and e<f 翻译举例」

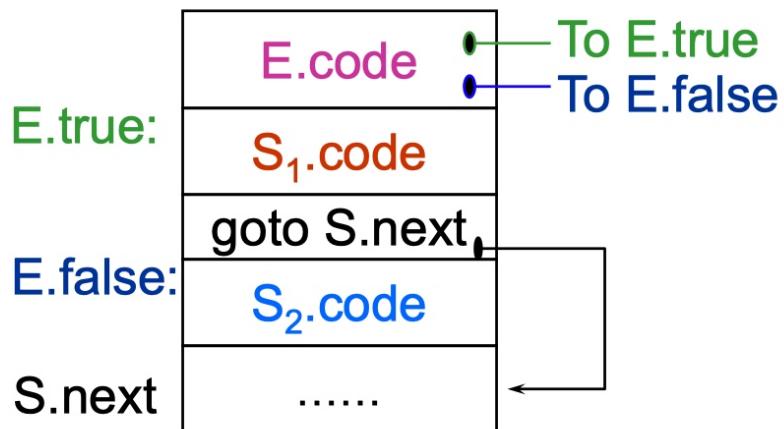
100:	if a<b goto 103
101:	T <sub>1</sub> :=0
102:	goto 104
103:	T <sub>1</sub> :=1
104:	if c<d goto 107
105:	T <sub>2</sub> :=0
106:	goto 108
107:	T <sub>2</sub> :=1
108:	if e<f goto 111
109:	T <sub>3</sub> :=0
110:	goto 112
111:	T <sub>3</sub> :=1
112:	T <sub>4</sub> :=T <sub>2</sub> and T <sub>3</sub>
113:	T <sub>5</sub> :=T <sub>1</sub> or T <sub>4</sub>

E→id <sub>1</sub> relop id <sub>2</sub>	{ E.place:=newtemp; emit('if' id <sub>1</sub> .place relop.op id <sub>2</sub> .place 'goto' nextstat+3); emit(E.place ':=' '0'); emit('goto' nextstat+2); emit(E.place ':=' '1') }
E→id	{ E.place:=id.place }
E→E <sub>1</sub> or E <sub>2</sub>	{ E.place:=newtemp; emit(E.place ':=' E <sub>1</sub> .place 'or' E <sub>2</sub> .place)}
E→E <sub>1</sub> and E <sub>2</sub>	{ E.place:=newtemp; emit(E.place ':=' E <sub>1</sub> .place 'and' E <sub>2</sub> .place) }

## 4.3 带优化的翻译

「核心思想」从左到右判断，当结果已经得出时，退出。

► 条件语句 if E then  $S_1$  else  $S_2$   
赋予 E 两种出口:一真一假



「举例」

► if  $a > c \text{ or } b < d$  then  $S_1$  else  $S_2$  翻译成三地址代码

```

if a>c goto L2  "真" 出口
goto L1
L1:if b<d goto L2  "真" 出口
      goto L3        "假" 出口
L2:(关于S1的三地址代码序列)
      goto Lnext
L3:(关于S2的三地址代码序列)
Lnext: 整个 if else 语句的 next 地址
  
```

## 4.4 布尔表达式的属性文法

### 4.4.1 属性

- 语义函数 newlabel, 返回一个新的符号标号
- 对于一个布尔表达式 E, 设置两个继承属性
  - E.true 是 E 为 ‘真’ 时控制流转向的标号
  - E.false 是 E 为 ‘假’ 时控制流转向的标号
- E.code 记录 E 生成的三地址代码序列

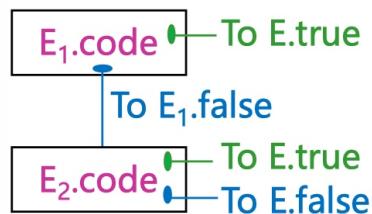
### 4.4.2 文法

- 产生布尔表达式三地址代码的属性文法

$E \rightarrow E_1 \text{ or } E_2$

### 产生式 语义规则

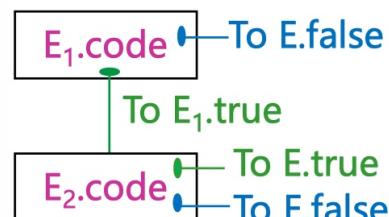
$E \rightarrow E_1 \text{ or } E_2$	$E_1.\text{true} := E.\text{true};$ $E_1.\text{false} := \text{newlabel};$ $E_2.\text{true} := E.\text{true};$ $E_2.\text{false} := E.\text{false};$ $E.\text{code} := E_1.\text{code} \parallel \text{gen}(E_1.\text{false} ':') \parallel E_2.\text{code}$
-------------------------------------	--



$E \rightarrow E_1 \text{ and } E_2$

### 产生式 语义规则

$E \rightarrow E_1 \text{ and } E_2$	$E_1.\text{true} := \text{newlabel};$ $E_1.\text{false} := E.\text{false};$ $E_2.\text{true} := E.\text{true};$ $E_2.\text{false} := E.\text{fasle};$ $E.\text{code} := E_1.\text{code} \parallel \text{gen}(E_1.\text{true} ':') \parallel E_2.\text{code}$
--------------------------------------	--



$E \rightarrow \text{not } E_1, E \rightarrow (E_1)$

### 产生式 语义规则

$E \rightarrow \text{not } E_1$	$E_1.\text{true} := E.\text{false};$ $E_1.\text{false} := E.\text{true};$ $E.\text{code} := E_1.\text{code}$
---------------------------------	--

$E \rightarrow (E_1)$	$E_1.\text{true} := E.\text{true};$ $E_1.\text{false} := E.\text{false};$ $E.\text{code} := E_1.\text{code}$
-----------------------	--

「 $E \rightarrow id_1\ relOp\ id_2$ 」

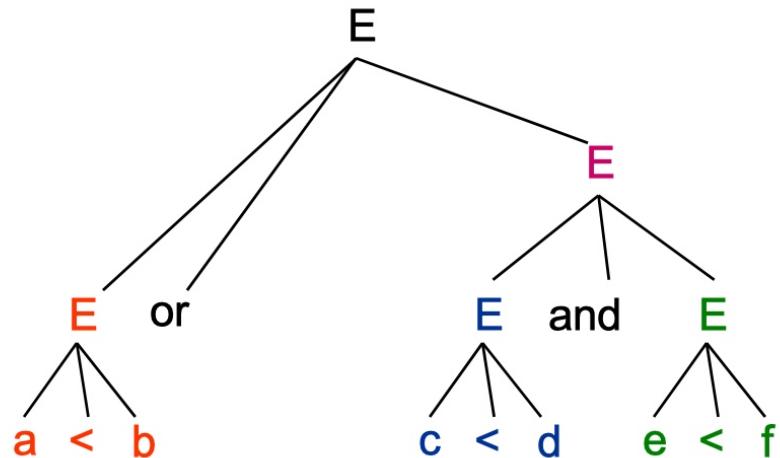
产生式	语义规则
$E \rightarrow id_1 \text{ relop } id_2$	$E.\text{code} := \text{gen('if' } id_1.\text{place relop}.op id_2.\text{place 'goto' } E.\text{true})$ $\quad\quad\quad    \text{ gen('goto' } E.\text{false})$
$E \rightarrow \text{true}$	$E.\text{code} := \text{gen('goto' } E.\text{true})$
$E \rightarrow \text{false}$	$E.\text{code} := \text{gen('goto' } E.\text{false})$

- 举例

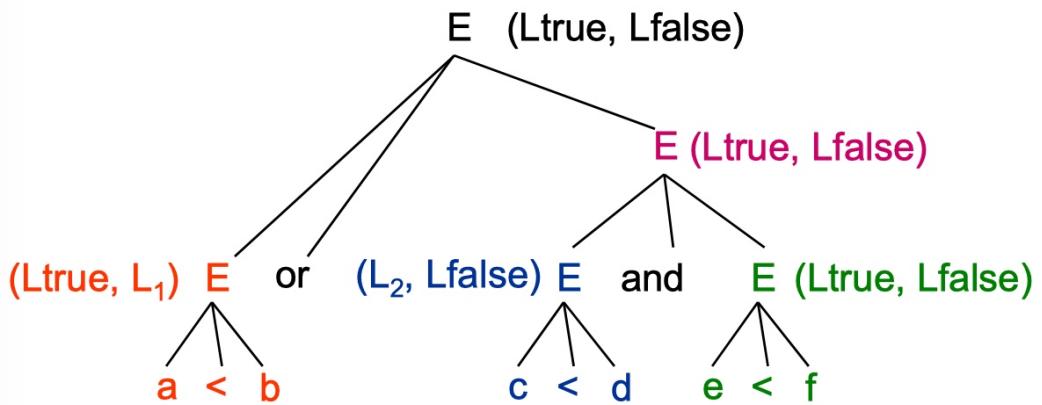
「布尔表达式：  $a < b \text{ or } c < d \text{ and } e < f$ 」

## 「翻译流程：第一遍扫描 — 构建语法树」

- 整个表达式的真假出口分别置为 Ltrue、Lfalse



### 「翻译流程：第二遍扫描 — 自上而下求出继承属性」



「翻译流程：第三遍扫描 — 自上而下求出综合属性 E.code」

```

E.code: (red)
    if a < b goto Ltrue
    goto L1
E.code: (blue)
    if c < d goto L2
    goto Lfalse
E.code: (green)
    if e < f goto Ltrue
    goto Lfalse

E.code: (purple)
    if c < d goto L2
    goto Lfalse
L2: if e < f goto Ltrue
    goto Lfalse

E.code: (black)
    if a < b goto Ltrue
    goto L1
L1: if c < d goto L2
    goto Lfalse
L2: if e < f goto Ltrue
    goto Lfalse

```

## 4.5 布尔表达式的一遍扫描翻译模式

### 4.5.1 定义引入

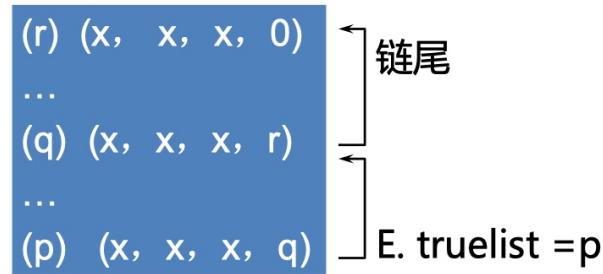
#### 「翻译过程以四元式为中间语言」

- 四元式存入一个数组中，数组下标代表四元式的标号
- 约定
  - 四元式 (jnz, a, -, p) 表示 if a goto p
  - 四元式 (jrop, x, y, p) 表示 if x rop y goto p
  - 四元式 (j, -, -, p) 表示 goto p
- 过程 emit 将四元式代码送到输出数组中

#### 「E 综合属性 E.truelist 和 E.falselist」

- E.truelist 和 E.falselist 分别记录布尔表达式 E 所对应的四元式中需回填“真”、“假”出口的四元式的标号所构成的链表
- 举例

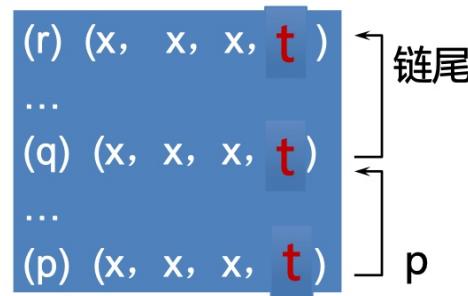
▶ 例如，假定E的四元式中需要回填“真”出口的p, q, r三个四元式，则E.truelist为下列链：



## 「语义变量和过程」

- 变量 *nextquad*
  - 指向下一条将要产生但尚未形成的四元式的地址（标号）
  - *nextquad* 初值为 1，每当执行一次 emit 之后，*nextquad* 自动增一
- 函数 *makelist(i)*
  - 创建一个仅含 i 的新链表，其中 i 式四元式数组的一个下标（标号）
  - 函数返回指向这个链的指针
- 函数 *merge(p<sub>1</sub>, p<sub>2</sub>)*
  - 把以 *p<sub>1</sub>* 和 *p<sub>2</sub>* 为链首的两条链合并为一
  - 函数值返回合并后的链首
- 过程 *backpatch(p, t)*
  - 完成“回填”
  - 把 *p* 所链接的每个四元式的第四区段都填为 *t*

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



## 「布尔表达式的文法」

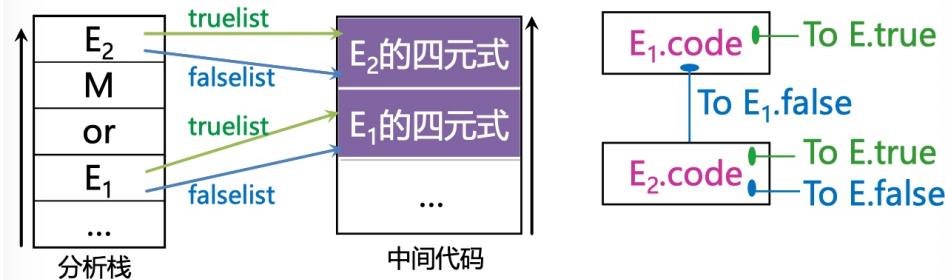
- $E \rightarrow E_1 \text{ or } M \ E_2$
- $E \rightarrow E_1 \text{ and } M \ E_2$
- $E \rightarrow \text{not } E_1$
- $E \rightarrow (E_1)$
- $E \rightarrow id_1 \ relop \ id_2$
- $E \rightarrow id$

- $M \rightarrow \varepsilon$ 
  - $\{ M.\text{quad} := \text{nextquad} \}$

#### 4.5.2 布尔表达式翻译模式

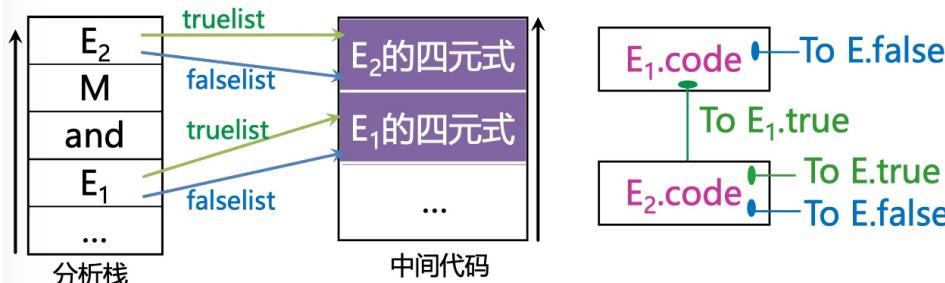
(1)  $E \rightarrow E_1 \text{ or } M E_2$

{ backpatch( $E_1.\text{falselist}$ ,  $M.\text{quad}$ );  
 $E.\text{truelist} := \text{merge}(E_1.\text{truelist}, E_2.\text{truelist})$ ;  
 $E.\text{falselist} := E_2.\text{falselist}$  }



(2)  $E \rightarrow E_1 \text{ and } M E_2$

{ backpatch( $E_1.\text{truelist}$ ,  $M.\text{quad}$ );  
 $E.\text{truelist} := E_2.\text{truelist}$ ;  
 $E.\text{falselist} := \text{merge}(E_1.\text{falselist}, E_2.\text{falselist})$  }



(3)  $E \rightarrow \text{not } E_1$

{  $E.\text{truelist} := E_1.\text{falselist}$ ;  
 $E.\text{falselist} := E_1.\text{truelist}$  }

(4)  $E \rightarrow (E_1)$

{  $E.\text{truelist} := E_1.\text{truelist}$ ;  
 $E.\text{falselist} := E_1.\text{falselist}$  }

(5)  $E \rightarrow id_1 \text{ relop } id_2$   
{ E.truelist:=makelist(nextquad);  
E.falselist:=makelist(nextquad+1);  
emit('j' relop.op ',' id<sub>1</sub>.place ',' id<sub>2</sub>.place ',' '0');  
emit('j, - , - , 0') }

(6)  $E \rightarrow id$   
{ E.truelist:=makelist(nextquad);  
E.falselist:=makelist(nextquad+1);  
emit('jnz ',' id.place ',' ' - ',' '0');  
emit(' j, - , - , 0') }

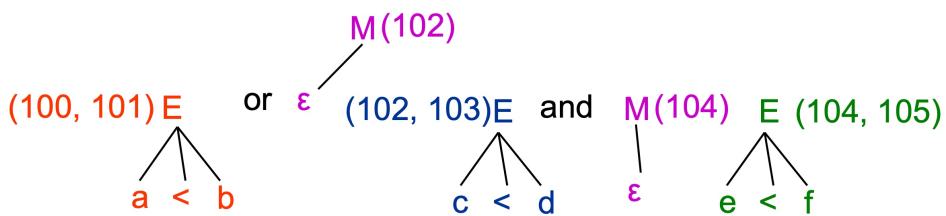
「举例：一遍扫描」

a < b or c < d and e < f

(5)  $E \rightarrow id_1 \text{ relop } id_2$   
{ E.truelist:=makelist(nextquad);  
E.falselist:=makelist(nextquad+1);  
emit('j' relop.op ',' id<sub>1</sub>.place ',' id<sub>2</sub>.place ',' '0');  
emit('j, - , - , 0') }

(7)  $M \rightarrow \epsilon \quad \{ M.\text{quad} := \text{nextquad} \}$

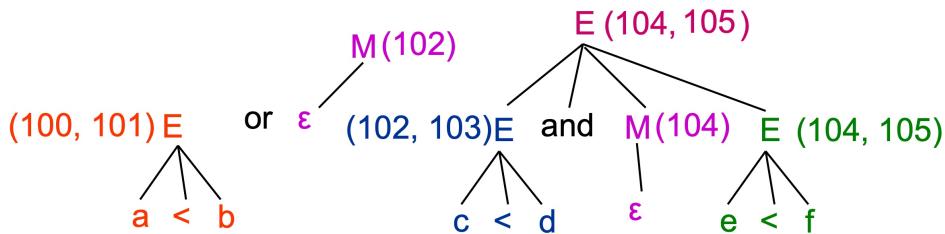
100	( j<, a, b, 0 )
101	( j, -, -, 0 )
102	( j<, c, d, 0 )
103	( j, -, -, 0 )
104	( j<, e, f, 0 )
105	( j, -, -, 0 )



a < b or c < d and e < f

(2)  $E \rightarrow E_1 \text{ and } M E_2$   
 { backpatch( $E_1$ .truelist,  $M$ .quad);  
 $E$ .truelist:= $E_2$ .truelist;  
 $E$ .falselist:=merge( $E_1$ .falselist, $E_2$ .falselist) }

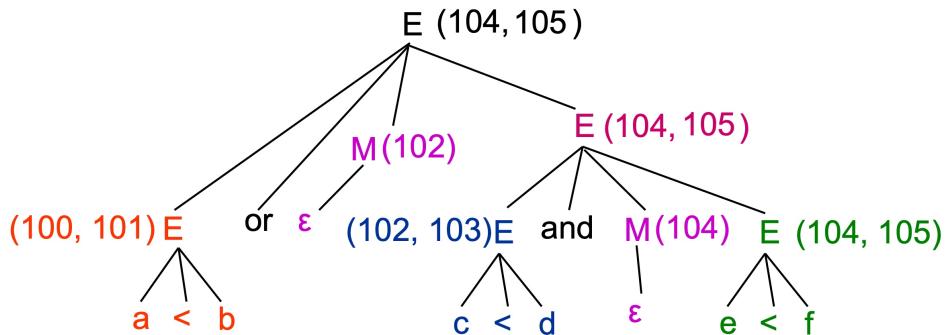
100	( j<, a, b, 0 )
101	( j, -, -, 0 )
102	( j<, c, d, 104 )
103	( j, -, -, 0 )
104	( j<, e, f, 0 )
105	( j, -, -, 103 )



a < b or c < d and e < f

(1)  $E \rightarrow E_1 \text{ or } M E_2$   
 { backpatch( $E_1$ .falselist,  $M$ .quad);  
 $E$ .truelist:=merge( $E_1$ .truelist,  $E_2$ .truelist);  
 $E$ .falselist:= $E_2$ .falselist }

100	( j<, a, b, 0 )
101	( j, -, -, 102 )
102	( j<, c, d, 104 )
103	( j, -, -, 0 )
104	( j<, e, f, 100 )
105	( j, -, -, 103 )



## 五、控制语句的翻译

「常用的控制语句」

- 单分支:  $S \rightarrow \text{if } E \text{ then } S_1$
- 双分支:  $S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$
- 循环:  $S \rightarrow \text{while } E \text{ do } S_1$

### 5.1 控制语句的属性文法

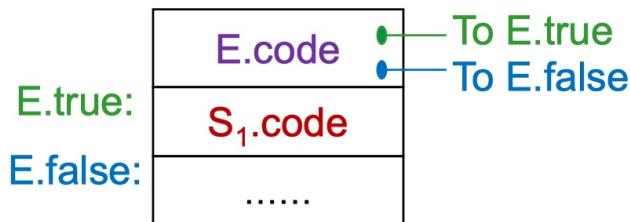
## 5.1.1 基础定义

### 「属性定义」

- E.true、E.false 表示 E 为真 / 假时的跳转语句标号
- S.begin、S.next 表示语句 S 开始执行、执行结束后的地址

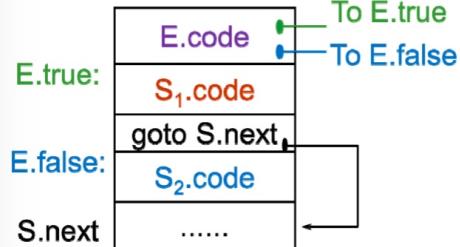
### 「if-then 语句的属性文法」

产生式	语义规则
$S \rightarrow \text{if } E \text{ then } S_1$	$E.\text{true} := \text{newlabel};$ $E.\text{false} := S.\text{next};$ $S_1.\text{next} := S.\text{next}$ $S.\text{code} := E.\text{code} \parallel \text{gen}(E.\text{true} ':') \parallel S_1.\text{code}$



### 「if-then-else 语句的属性文法」

产生式	语义规则
$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$	$E.\text{true} := \text{newlabel};$ $E.\text{false} := \text{newlabel};$ $S_1.\text{next} := S.\text{next}$ $S_2.\text{next} := S.\text{next};$ $S.\text{code} := E.\text{code} \parallel$ $\quad \text{gen}(E.\text{true} ':') \parallel S_1.\text{code} \parallel$ $\quad \text{gen('goto' } S.\text{next}) \parallel$ $\quad \text{gen}(E.\text{false} ':') \parallel S_2.\text{code}$



### 「while-do 语句的属性文法」

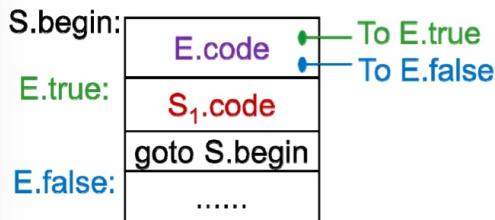
## 产生式

$S \rightarrow \text{while } E \text{ do } S_1$

## 语义规则

```

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



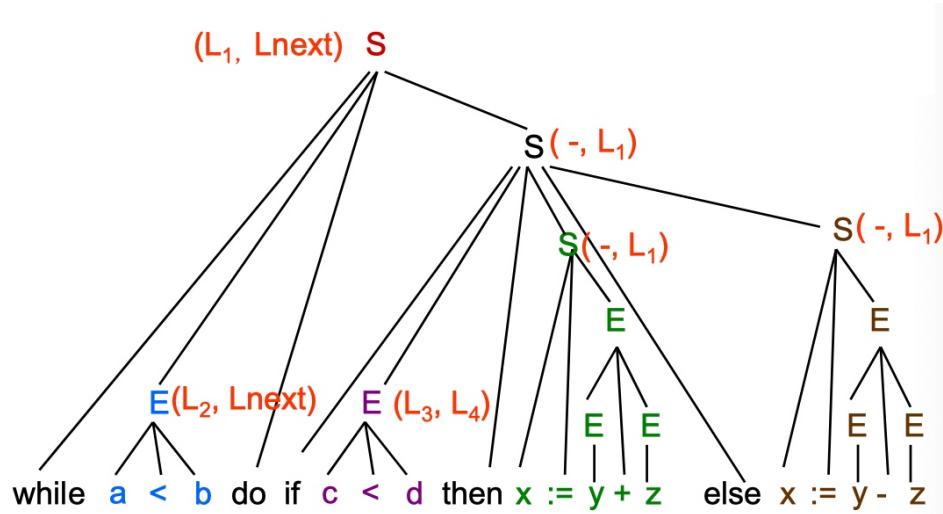
### 5.1.2 三遍扫描示例

「三遍扫描的属性计算示例」

```

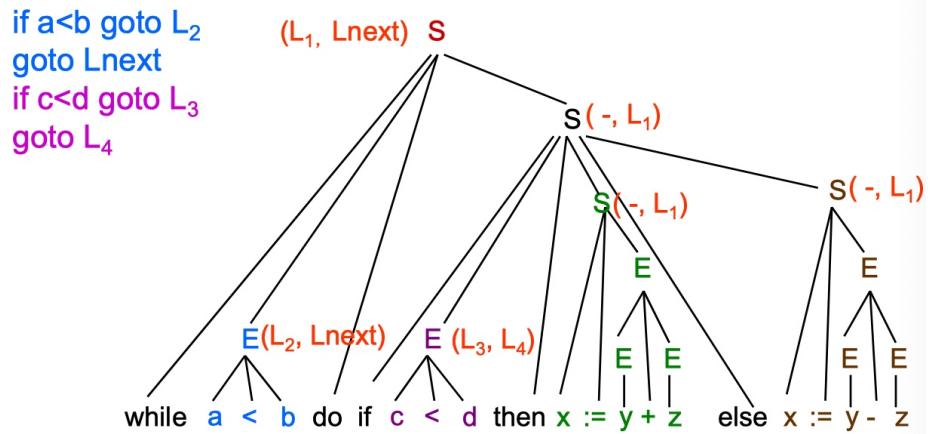
while a<b do
  if c<d then x:=y+z
  else x:=y-z
  
```

- 第一遍：构建语法树
- 第二遍：自上而下计算继承属性



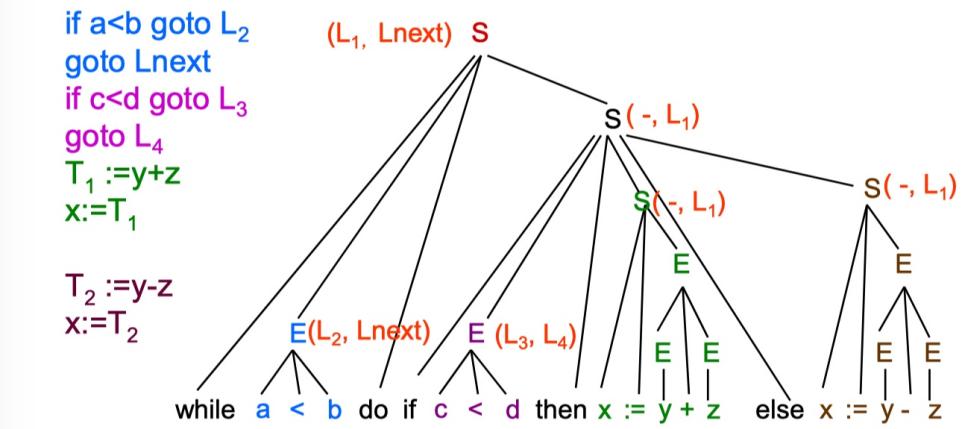
- 第三遍：自下而上计算综合属性 (code)

产生式 语义规则



## 产生式 语义规则

$S \rightarrow id := E$	$S.code := E.code \parallel gen(id.place := E.place)$
$E \rightarrow E_1 + E_2$	$E.place := newtemp;$ $E.code := E_1.code \parallel E_2.code \parallel gen(E.place := E_1.place + E_2.place)$



## 产生式

$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$

语义规则

E.true:=newlabel;

E.false:=newlab

$S_1.next := S.next$

S<sub>2</sub>.next=S.next  
S.code:=E.code

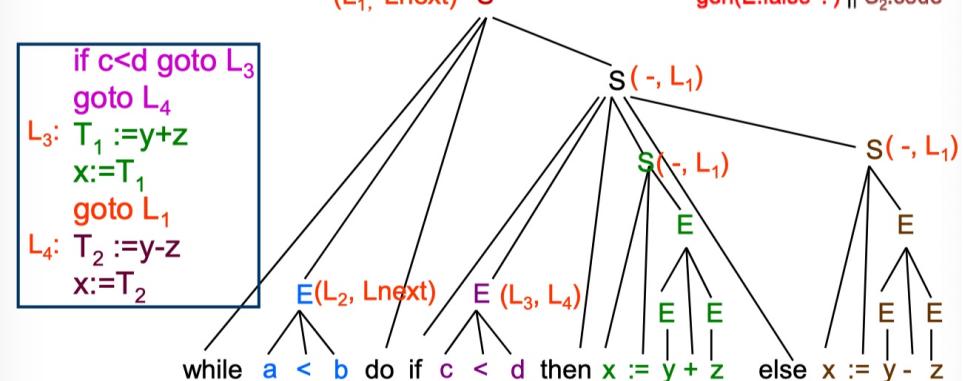
gen(E.tr)

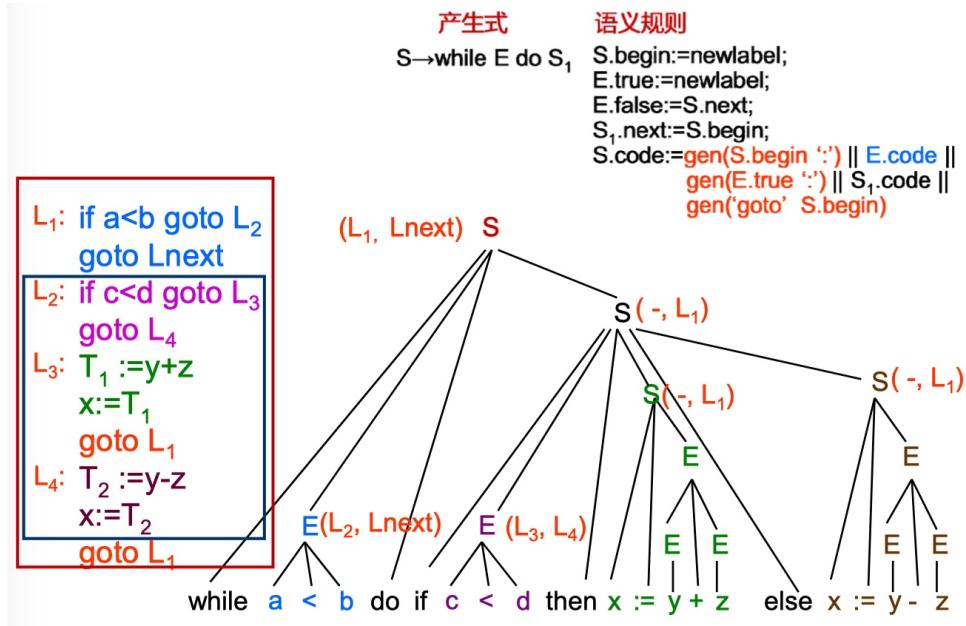
gen('goto' S.next) ||  
gen(E.false ':') || S<sub>2</sub>.code

```

if c<d goto L3
goto L4
L3: T1 := y+z
x := T1
goto L1
L4: T2 := y-z
x := T2

```





即上述 while 控制语句最终得到如下所示的翻译结果：

```

L1: if a<b goto L2
      goto Lnext
L2: if c<d goto L3
      goto L4
L3: T1 := y+z
      x := T1
      goto L1
L4: T2 := y-z
      x := T2
      goto L1
    
```

## 5.2 控制语句的属性文法

「控制语句的文法」

- $S \rightarrow \text{if } E \text{ then } S$
- $S \rightarrow \text{if } E \text{ then } S \text{ else } S$
- $S \rightarrow \text{while } E \text{ do } S$
- $S \rightarrow \text{begin } L \text{ end}$
- $S \rightarrow A$
- $L \rightarrow L; S$
- $L \rightarrow S$

其中 S 表示语句， L 表示语句表， A 表示赋值语句， E 表示布尔表达式。

### 5.2.1 if 语句的翻译模式

「改写后的产生式」

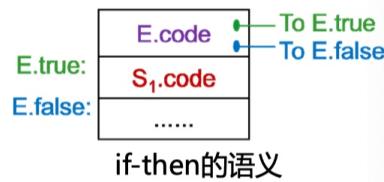
- $S \rightarrow \text{if } E \text{ then } M \ S_1$
- $S \rightarrow \text{if } E \text{ then } M_1 \ S_1 \ N \ \text{else } M_2 \ S_2$
- $M \rightarrow \epsilon$
- $N \rightarrow \epsilon$

「翻译模式」

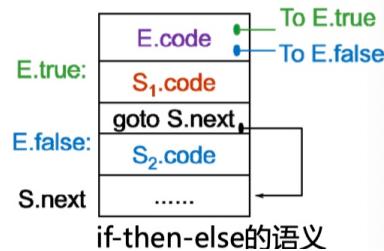
- $S.\text{nextlist}$  属性：保存  $S$  翻译生成的代码中需要回填的，要跳转到  $S$  的后继语句的那些四元式。

## if 语句的翻译模式

1.  $S \rightarrow \text{if } E \text{ then } M \ S_1$   
   { backpatch(E.truelist, M.quad);  
      $S.\text{nextlist} := \text{merge}(E.falselist, S_1.\text{nextlist})$  }



2.  $S \rightarrow \text{if } E \text{ then } M_1 \ S_1 \ N \ \text{else } M_2 \ S_2$   
   { backpatch(E.truelist, M<sub>1</sub>.quad);  
     backpatch(E.falselist, M<sub>2</sub>.quad);  
      $S.\text{nextlist} := \text{merge}(S_1.\text{nextlist}, N.\text{nextlist}, S_2.\text{nextlist})$  }



3.  $M \rightarrow \epsilon \ \{ M.\text{quad} := \text{nextquad} \}$

4.  $N \rightarrow \epsilon \ \{ N.\text{nextlist} := \text{makelist(nextquad)}; \text{emit('j, -, -, -')} \}$

## 5.2.2 while-do 语句的翻译模式

「改写后的产生式」

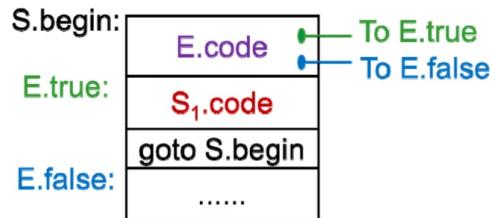
- $S \rightarrow \text{while } M_1 \ E \ \text{do } M_2 \ S_1$
- $M \rightarrow \epsilon$

「翻译模式」

## while-do语句的翻译模式

1.  $S \rightarrow \text{while } M_1 E \text{ do } M_2 S_1$   
{ backpatch(E.truelist, M<sub>2</sub>.quad);  
backpatch(S<sub>1</sub>.nextlist, M<sub>1</sub>.quad);  
S.nextlist := E.falselist;  
emit( 'j, - , - ; M<sub>1</sub>.quad) }

2.  $M \rightarrow \epsilon$   
{ M.quad := nextquad }



### 5.2.3 复合语句的翻译模式

「改写后的产生式」

- $S \rightarrow \text{begin } L \text{ end}$
- $L \rightarrow L_1; M S \mid S$
- $M \rightarrow \epsilon$

「翻译模式」

1.  $S \rightarrow \text{begin } L \text{ end}$   
{ S.nextlist := L.nextlist }
2.  $L \rightarrow L_1; M S$   
{ backpatch(L<sub>1</sub>.nextlist, M.quad);  
L.nextlist := S.nextlist }
3.  $M \rightarrow \epsilon$   
{ M.quad := nextquad }

「其它语句的翻译」

1.  $S \rightarrow A \quad \{ S.\text{nextlist} := \text{makelist( )} \}$
2.  $L \rightarrow S \quad \{ L.\text{nextlist} := S.\text{nextlist} \}$

## 5.2.4 一遍扫描示例

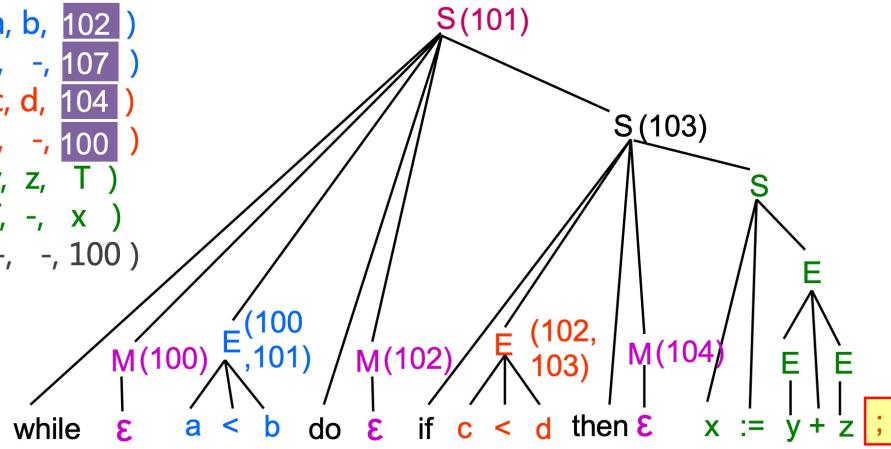
「控制语句示例」

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

```
S→while M1 E do M2 S1  
{ backpatch(E.truelist, M2.quad);  
  backpatch(S1.nextlist,  
            M1.quad);  
  S1.nextlist:=E.falselist  
  emit('j, -, -, ' M1.quad) }  
M→ε { M.quad:=nextquad }
```

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

100 (j<, a, b, 102 )  
101 (j, -, -, 107 )  
102 (j<, c, d, 104 )  
103 (j, -, -, 100 )  
104 (+, y, z, T )  
105 (:=, T, -, x )  
106 (j, -, -, 100 )  
107



## 优化代码

### 一、优化概述

#### 1.1 基本概念

「优化」

- 对程序进行各种等价变换，使得从变换后的程序出发，能生成更有效的目标代码。
  - 等价**: 不改变程序的运行结果
  - 有效**: 目标代码运行时间短，占用存储空间小

「遵循的原则」

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

「优化的级别」

- 局部优化、循环优化、全局优化

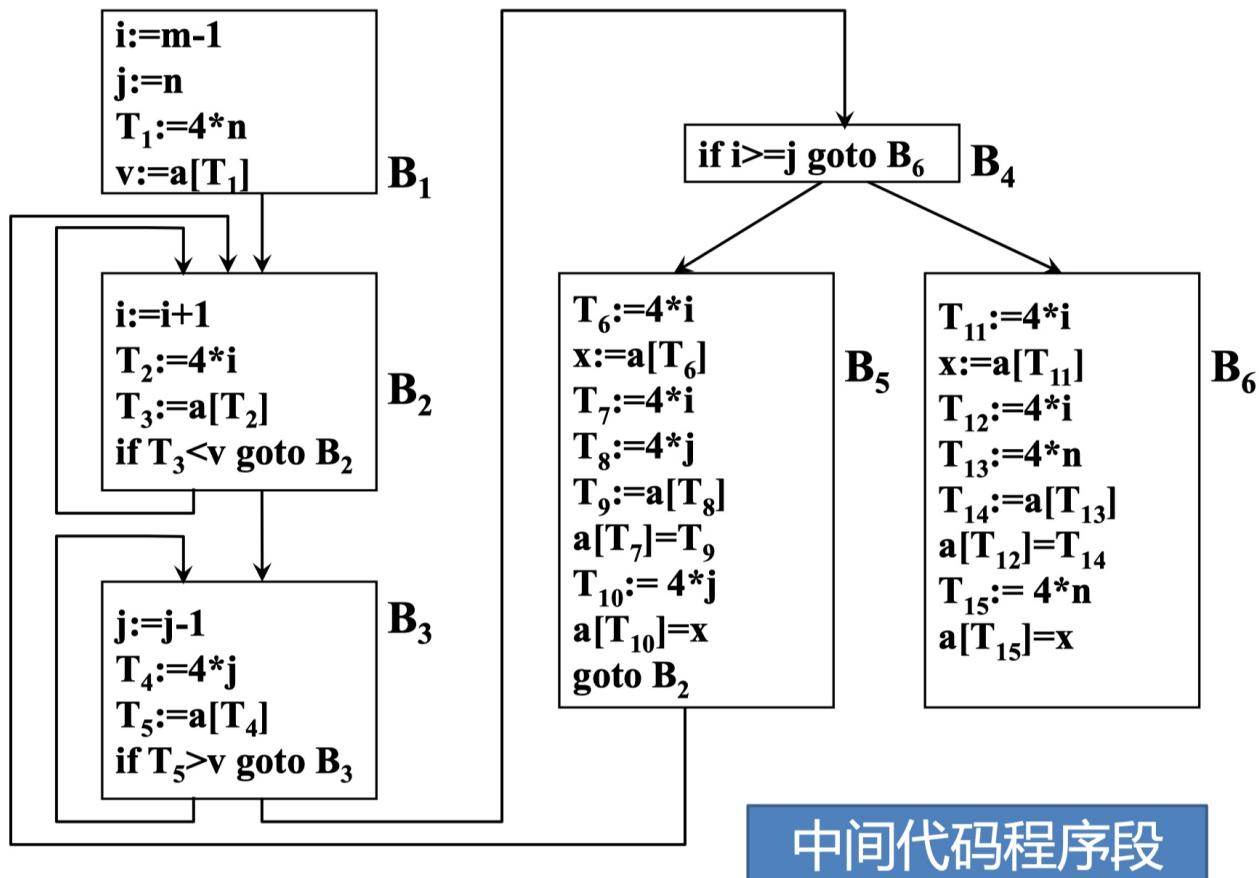
## 1.2 优化的种类

「总体列举」

- 删除多余运算（删除公用子表达式）
- 合并已知量
- 复写传播
- 删除无用赋值
- 代码外提
- 强度消弱
- 变换循环控制条件

### 原代码

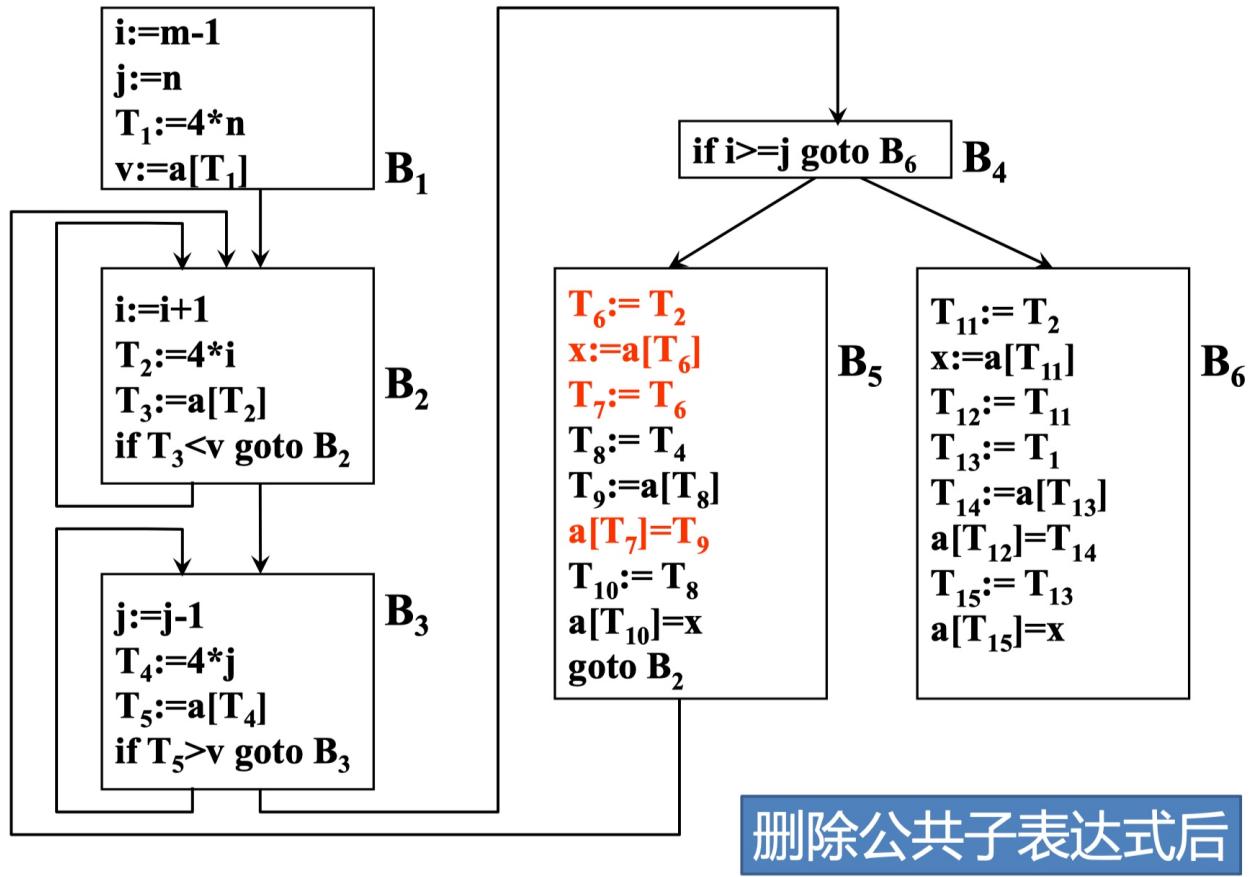
```
void quicksort(m, n);
int m, n;
{
    int i, j, v, x;
    if(n <= m) return;
    i = m-1, j = n, v = a[n];
    while(1) {
        do i = i+1; while(a[i] < v);
        do j = j-1; while(a[j] > v);
        if(i >= j) break;
        x = a[i], a[i] = a[j], a[j] = x;
    }
    x = a[i], a[i] = a[n], a[n] = x;
    quicksort(m, j); quicksort(i+1, n);
}
```



## 删除公共子表达式

```
T1 = 4*i
T2 = 4*i
```

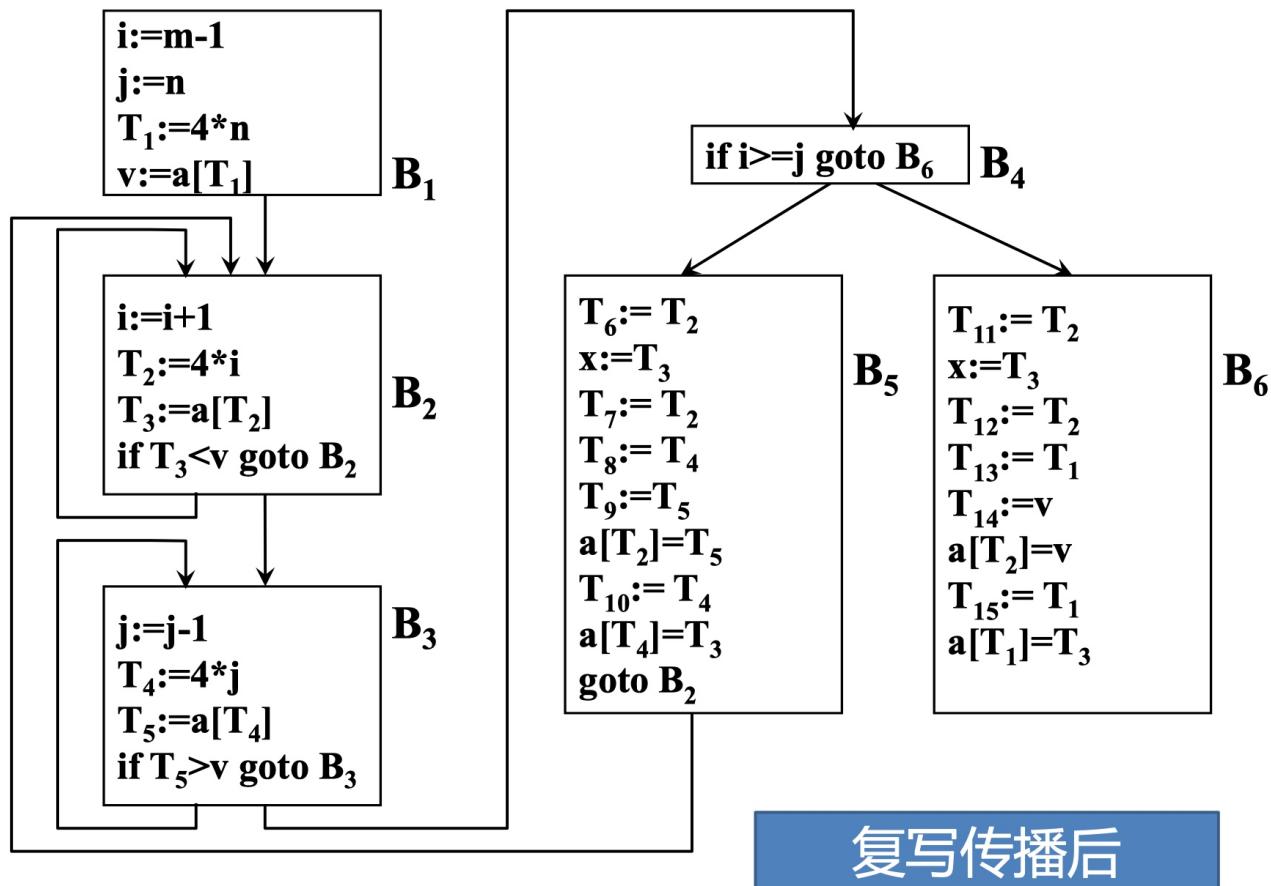
```
T1 = 4*i
T2 = T1
```



## 复写传播

```
T4 = T2
T3 = T4 + 1
```

```
T4 = T2
T3 = T2 + 1
```



复写传播后

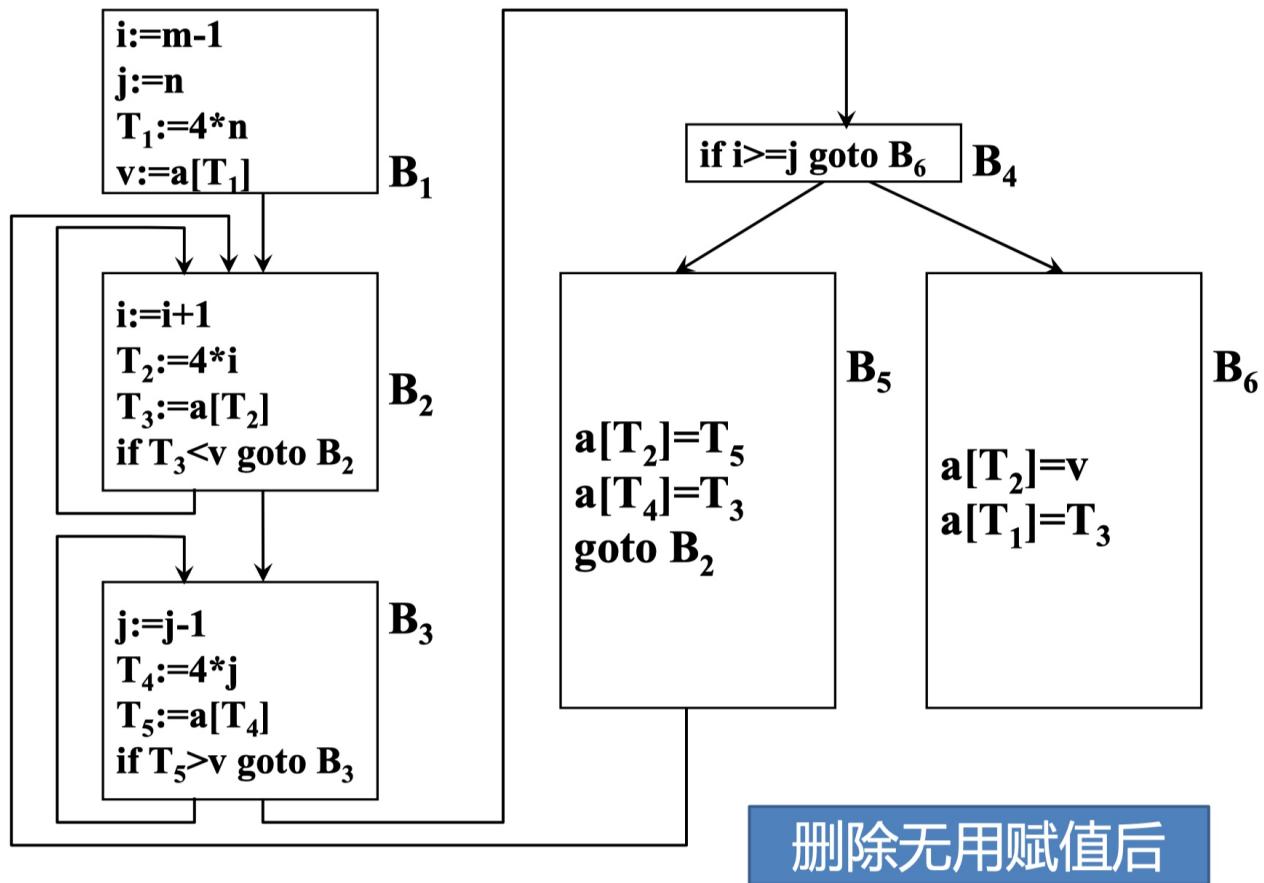
## 删除无用赋值

```

T4 = T2 // T4 无用
T3 = T2 + 1

T3 = T2 + 1

```



删除无用赋值后

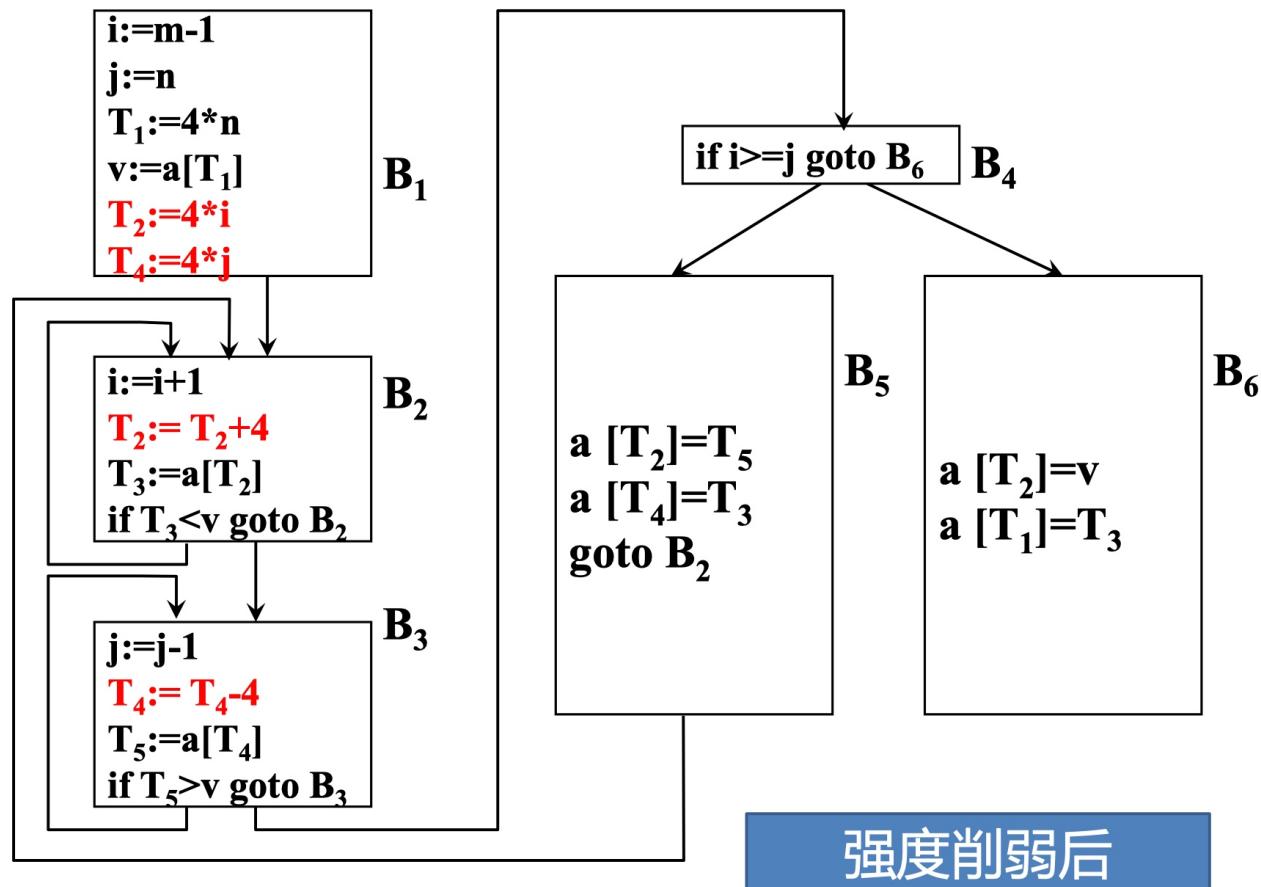
## 强度削弱

```

i = i + 1
T2 = 4 * i

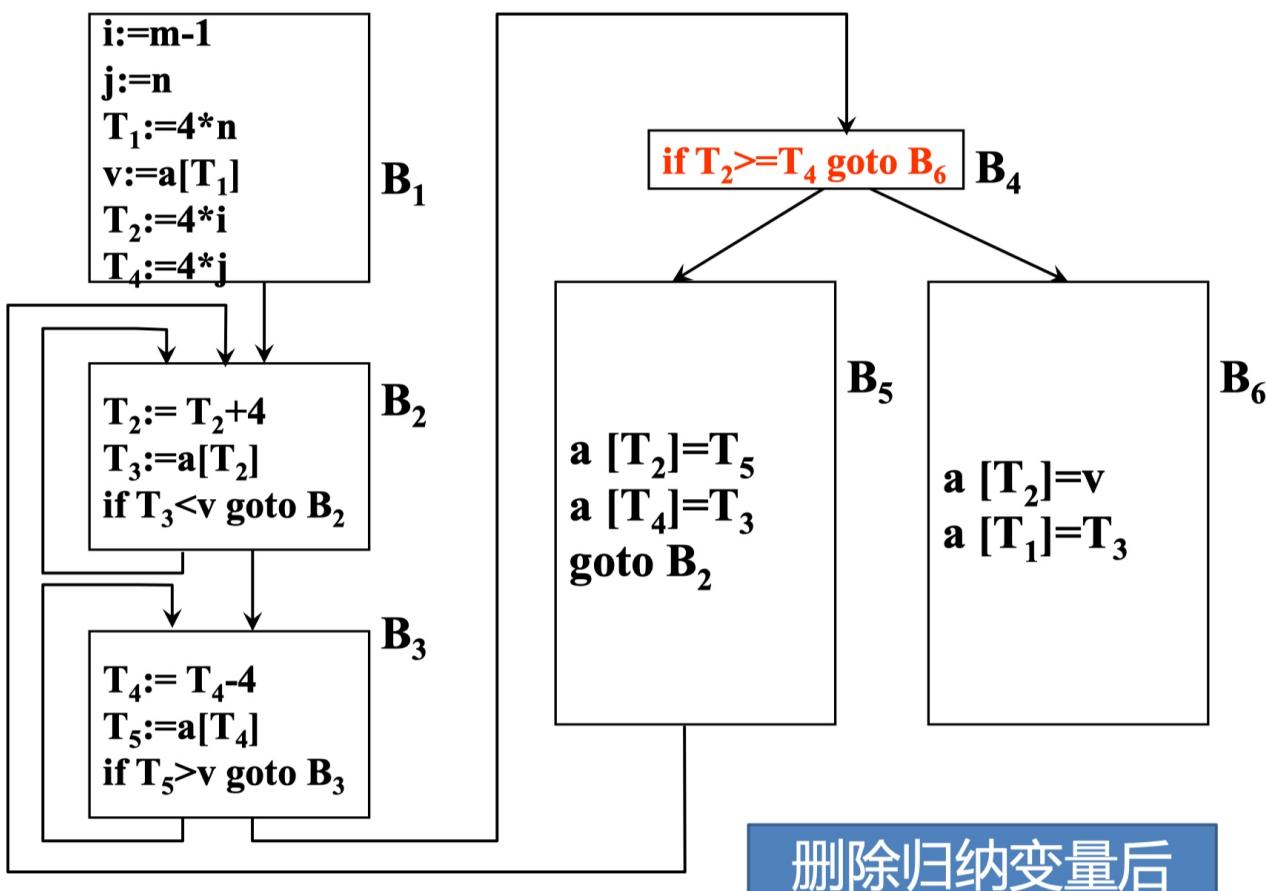
i = i + 1
T2 = T2 + 4

```



## 删除归纳变量

- 如下例所示， $i$ 、 $j$  可以被  $T_2$ 、 $T_4$  取代，因此删除  $i$ 、 $j$



## 1.3 数据流方程

入口活跃变量 = 出口活跃变量 - 被定义的变量 + 被引用的变量

## 二、局部优化

「局部优化」

- 局限于基本块范围内的优化

### 2.1 基本块

#### 2.1.1 概述

「基本块」

- 程序中一顺序执行语句序列，只有一个入口和一个出口。
- 入口就是第一个语句，出口就是最后一个语句。

「定值 & 引用」

- 对三地址语句为  $x := y + z$ , 称  $x$  定值并引用  $y$  和  $z$

「活跃的」

- 基本块中的一个名字在程序中的某个给定点是活跃的，指如果在程序中（本基本块 + 其它基本块）它的值在该点以后被引用

#### 2.1.2 划分基本块

一、找出中间语句程序中各个基本块的入口语句（三地址语句）

- 程序第一个语句
- 能由条件转移语句或无条件转移语句转移到的语句
- 紧跟在条件转移语句后面的语句

二、对以上求出的每个入口语句，确定其所属的基本块

- 入口语句 => 下一入口语句的前一天语句
- 入口语句 => 转移语句
- 入口语句 => 停语句

入口语句n

...

...

入口语句m

入口语句n

...

...

转移语句m

入口语句n

...

...

停语句m

三、凡未被纳入某一基本块中的语句，可以从程序中删除

### 2.1.3 基本块划分举例

- |                     |
|---------------------|
| (1) read X          |
| (2) read Y          |
| (3) R:=X mod Y      |
| (4) if R=0 goto (8) |
| (5) X:=Y            |
| (6) Y:=R            |
| (7) goto (3)        |
| (8) write Y         |
| (9) halt            |

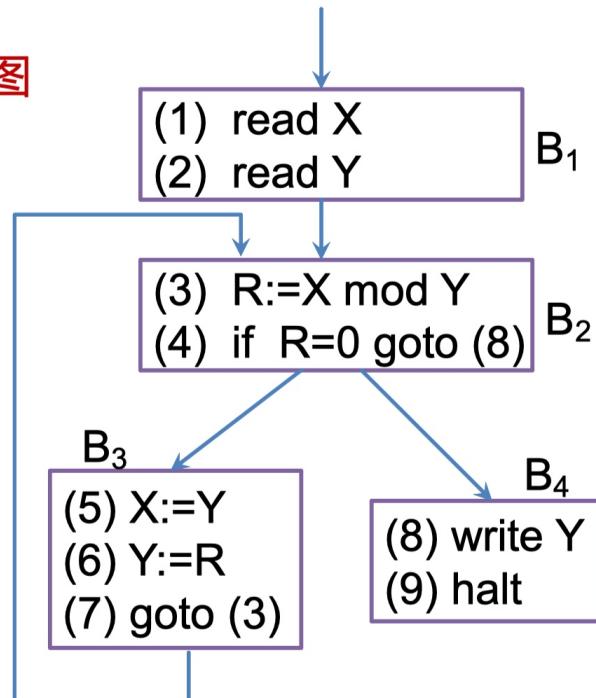
### 2.2 流图

## ▶ 以基本块为结点构成流图

```

(1) read X
(2) read Y
(3) R:=X mod Y
(4) if R=0 goto (8)
(5) X:=Y
(6) Y:=R
(7) goto (3)
(8) write Y
(9) halt

```



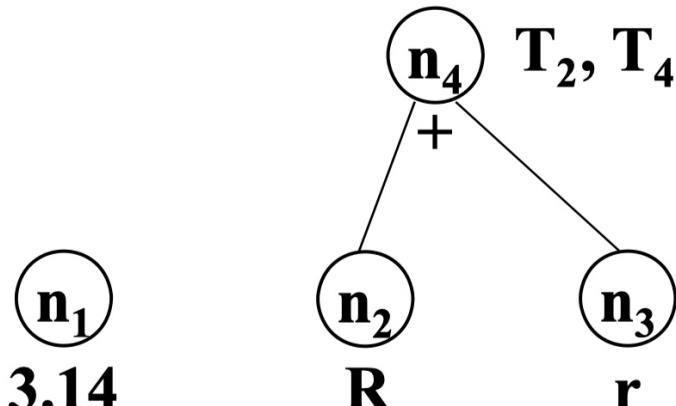
- 基本块为结点，程序第一条语句所在基本块是首结点

- 连边
  - B<sub>2</sub> 紧接着 B<sub>1</sub> 之后执行，且 B<sub>1</sub> 最后一条语句不是无条件转移语句，则 B<sub>1</sub> => B<sub>2</sub>
  - B<sub>1</sub> 最后一条语句是（无）条件转移语句，转移到 B<sub>2</sub> 的第一条语句，则 B<sub>1</sub> => B<sub>2</sub>

## 2.3 基本块的 DAG 表示

### 2.3.1 DAG 扩充

- 在 DAG 上增加标记和附加信息
  - 「叶结点」以标识符或常数作为标记，表示该结点代表该变量或常数的值
  - 「内部结点」以运算符作为标记，表示该结点代表应用该运算符对其后继结点所代表的值进行运算的结果
  - 「附加」各个结点可能附加一个或多个标识符表示这些变量具有该结点所代表的值



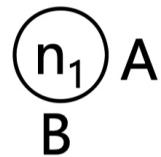
n<sub>1</sub>

3.14

## 2.3.2 四元式的 DAG 表示

「0型：  $A := B$ 」 ( $:=, B, -, A$ )

(0) 0型:  $A := B$   
( $:=, B, -, A$ )



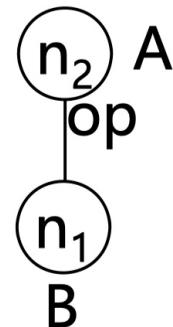
「0型： goto (s)」 ( $j, -, -, (s)$ )

0型: goto (s)  
( $j, -, -, (s)$ )



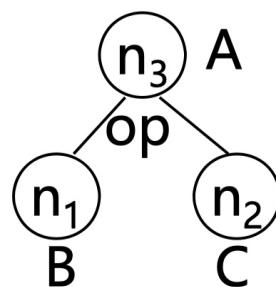
「1型：  $A := op B$ 」 ( $op, B, -, A$ )

1型:  $A := op B$   
( $op, B, -, A$ )



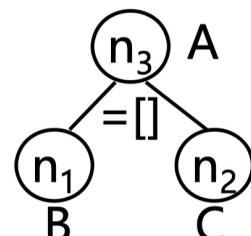
「2型：  $A := B op C$ 」 ( $op, B, C, A$ )

2型:  $A := B op C$   
( $op, B, C, A$ )



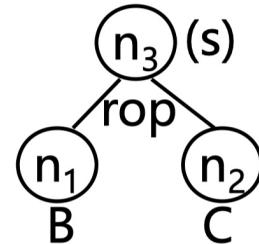
「2型：  $A := B[C]$ 」 ( $=[], B, C, A$ )

2型:  $A := B[C]$   
( $=[], B, C, A$ )



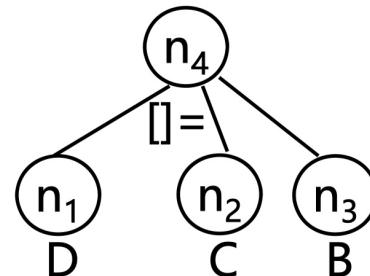
「2型: if B rop C goto (s)」 (jrop, B, C, (s))

2型: if B rop C goto (s)  
(jrop, B, C, (s))



「3型: D[C] := B」 ([]=, B, D, C)

3型: D[C]:=B  
([]=, B, D, C)



## 2.4 基本块的优化算法

### 2.4.1 优化算法概述

- 一个基本块，可用一个 DAG 来表示
- 对基本块中每一条四元式代码，依次构造对应的 DAG 图，最后基本块中所有四元式构造出来的 DAG 连成整个基本块的 DAG
- 具体流程
  - 准备操作数的结点
  - 合并已知量
  - 删减公共子表达式
  - 删减无用赋值

### 2.4.2 示例

「示例」

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

1. 准备操作数的结点
2. 合并已知量
3. 删除公共子表达式
4. 删除无用赋值

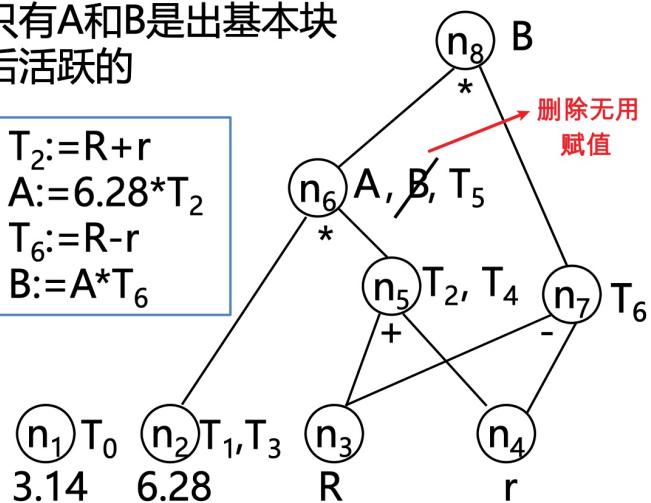
「优化结果」

### ► 优化后的四元式

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

若只有A和B是出基本块之后活跃的

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



### 2.4.3 构造基本块的 DAG 算法

参考上述示例，引入下述的具体过程。

- 引入函数 Node，保存和计算 DAG 中标识符与结点的对应关系

$$Node(A) = \begin{cases} n, & \text{如果存在一个结点 } n, \\ & A \text{ 是其上的标记或者附加标识符} \\ null, & \text{否则} \end{cases}$$

- 对基本块中每一四元式，依次执行以下步骤

1. 准备操作数的结点
2. 合并已知量
3. 删除公共子表达式
4. 删除无用赋值

## 「1. 准备操作数的结点」

- 如果  $\text{Node}(B)$  无定义，则构造一标记为  $B$  的叶结点并定义  $\text{Node}(B)$  为这个结点
  - 如果四元式为 0 型 ( $A := B$ )，则记  $\text{Node}(B) = n$ ，转 4
  - 如果四元式为 1 型 ( $A := \text{op } B$ )，转 2(1)
  - 如果四元式为 2 型 ( $A := B \text{ op } C$ )
    - 如果  $\text{Node}(C)$  无定义，则构造一标记为  $C$  的叶结点并定义  $\text{Node}(C)$  为这个结点
    - 转 2(2)

## 「2. 合并已知量」

- 如果  $\text{Node}(B)$  是标记为常数的叶结点，转 2(3)，否则转 3(1)
- 如果  $\text{Node}(B)$  和  $\text{Node}(C)$  都是标记为常数的叶结点，则转 2(4)，否则转 3(2)
- 执行  $\text{op } B$  (合并已知量)，令得到的新常数为  $P$ 
  - 如果  $\text{Node}(B)$  是处理当前四元式时新构造出来的结点，则删除它
  - 如果  $\text{Node}(P)$  无定义，则构造一用  $P$  作标识符的叶结点  $n$ ，置  $\text{Node}(P) = n$
  - 转 4
- 执行  $B \text{ op } C$  (合并已知量)，令得到的新常数为  $P$ 
  - 如果  $\text{Node}(B)$  或  $\text{Node}(C)$  是处理当前四元式时新构造出来的结点，则删除它
  - 如果  $\text{Node}(P)$  无定义，则构造一用  $P$  作标记的叶结点  $n$ ，置  $\text{Node}(P) = n$
  - 转 4

## 「3. 删除公共子表达式」

- 检查 DAG 中是否已有一结点，其唯一后继为  $\text{Node}(B)$ ，且标记为  $\text{op}$ ，即公共子表达式
  - 如果没有，则构造该结点  $n$
  - 否则，把已有的结点作为它的结点并设该结点为  $n$ ，转 4
- 检查 DAG 中是否已有一结点，其左后继为  $\text{Node}(B)$ ，右后继为  $\text{Node}(C)$ ，且标记为  $\text{op}$ ，即公共子表达式
  - 如果没有，则构造该结点  $n$
  - 否则，把已有的结点作为它的结点并设该结点为  $n$ ，转 4

## 「4. 删除无用赋值」

- 如果  $\text{Node}(A)$  无定义，则把  $A$  附加在结点  $n$  上并令  $\text{Node}(A) = n$
- 否则，先把  $A$  从  $\text{Node}(A)$  结点上的附加标识符集中删除
  - 如果  $\text{Node}(A)$  为叶结点，则其  $A$  标记不删除
  - 把  $A$  附加到新结点  $n$  上并置  $\text{Node}(A) = n$ ，转处理下一四元式

## 「信息总结」

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

## 二、循环优化

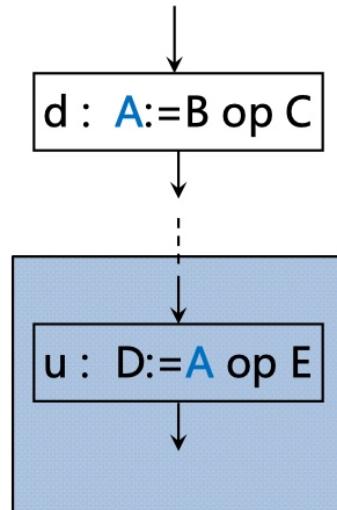
循环优化，主要有三大措施：

1. 代码外提
2. 强度消弱
3. 删除归纳变量（变换循环控制条件）

### 2.1 循环不变运算

- 具体定义

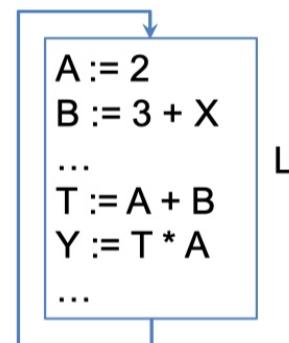
- ▶ 所谓变量A在某点d的**定值到达**另一点u（或称变量A的定值点d到达另一点u），是指流图中从d有一通路到达u且该通路上没有A的其它定值
- ▶ **循环不变运算**: 对四元式 $A := B \text{ op } C$ , 若B和C是常数，或者到达它们的B和C的定值点都在循环外



- 寻找算法

- 反复遍历

1. 依次查看L中各基本块的每个四元式，如果它的每个运算对象或为常数，或者定值点在L外，则将此四元式标记为“不变运算”；
2. 重复第3步直至没有新的四元式被标记为“不变运算”为止；
3. 依次查看尚未被标记为“不变运算”的四元式，如果它的每个运算对象或为常数，或定值点在L之外，或只有一个到达-定值点且该点上的四元式已被标记为“不变运算”，则把被查看的四元式标记为“不变运算”。



## 2.2 代码外提

- 找到不变运算
- 检查是否符合代码外提条件
- 保持次序前提下提出代码

1. 求出L的所有不变运算
2. 对每个不变运算  $s: A := B \text{ op } C$  或  $A := \text{op } B$  或  $A := B$  检查是否满足条件(1)或(2)

条件(1)

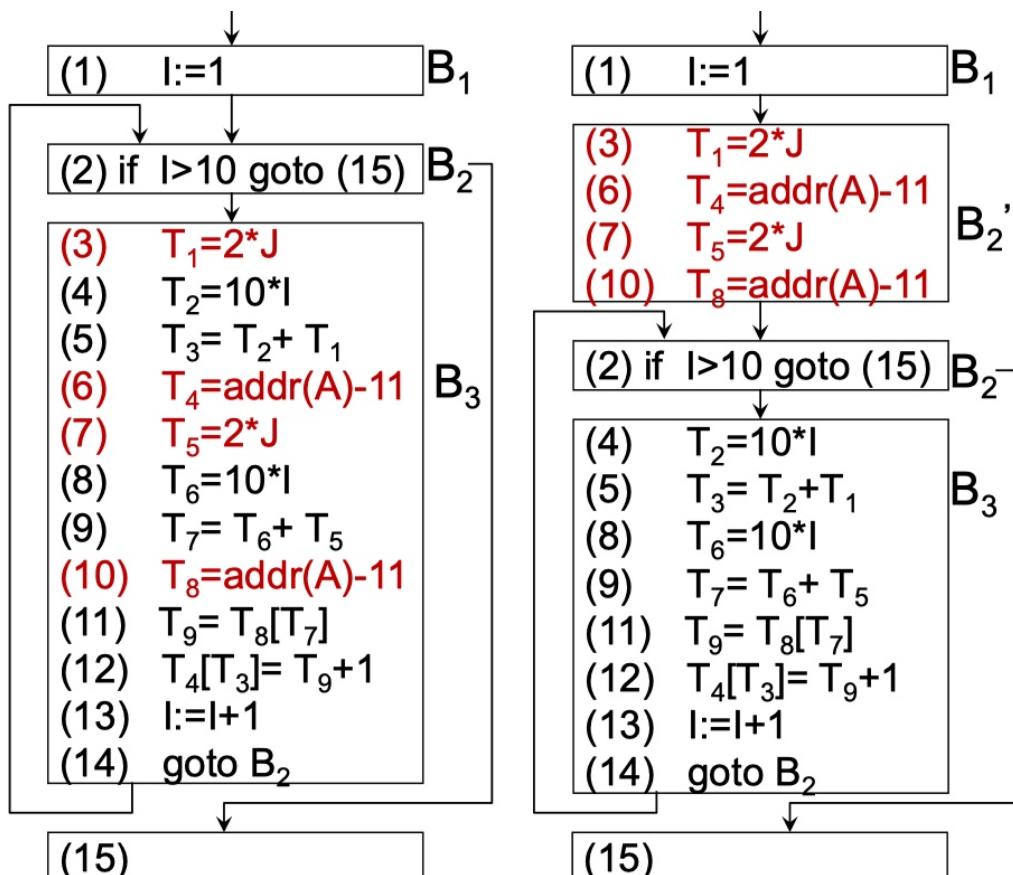
- (i) S所在的结点是L所有出口结点的必经结点;
- (ii) A在L中其他地方未再定值;
- (iii) L中所有A的引用点只有S中的A的定值才能到达。

条件(2)

- (i) A在离开L后不再是活跃的;
- (ii) A在L中其他地方未再定值;
- (iii) L中所有A的引用点只有S中的A的定值才能到达。

3.按步骤1所找出不变运算的次序，依次把符合步骤2的条件(1)或(2)的不变运算  $s$  外提到L的前置结点中。但是，如果  $s$  的运算对象(B或C)是在L中定值的，那么，只有当这些定值四元式都已外提到前置结点中时，才能把  $s$  也外提到前置结点中。

- 举例

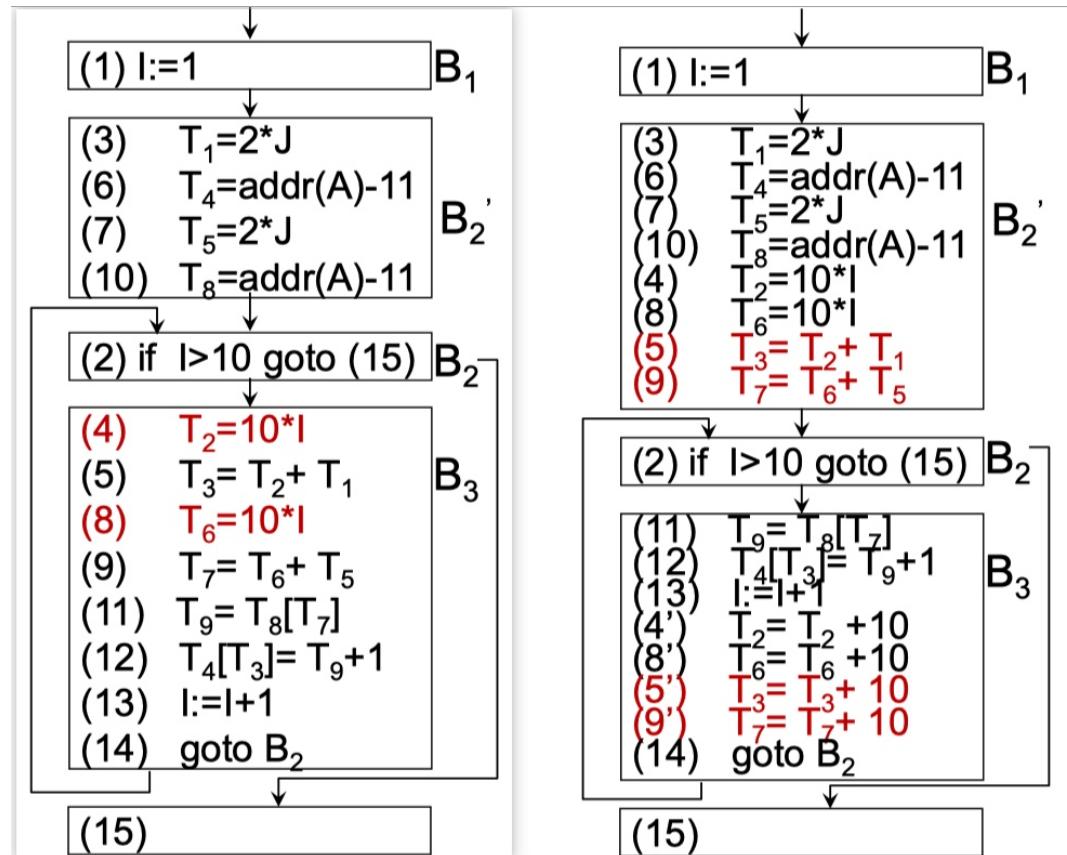


## 2.3 强度消弱

- 核心思想

- 将程序中执行时间较长的运算转换为执行时间较短的运算
- 例如将乘法换成加法

- 举例



- 适用范围

- 强度消弱通常是针对循环控制变量有先行关系的变量赋值进行
- 经过强度消弱后，循环中可能出现一些新的无用赋值
- 对于消弱下标变量地址计算的强度非常有效

## 2.4 删减归纳变量

### 2.4.1 归纳变量

- 基本归纳变量、归纳变量的定义

- ▶ 如果循环中对变量I只有唯一的形如  $I := I \pm C$  的赋值，且其中C为循环不变量，则称I为循环中的**基本归纳变量**
- ▶ 如果I是循环中一基本归纳变量，J在循环中的定值总是可化归为I的同一线性函数，也即  $J = C_1 * I \pm C_2$ ，其中  $C_1$  和  $C_2$  都是循环不变量，则称J是**归纳变量**，并称它与I同族。基本归纳变量也是归纳变量。

## 2.4.2 具体算法

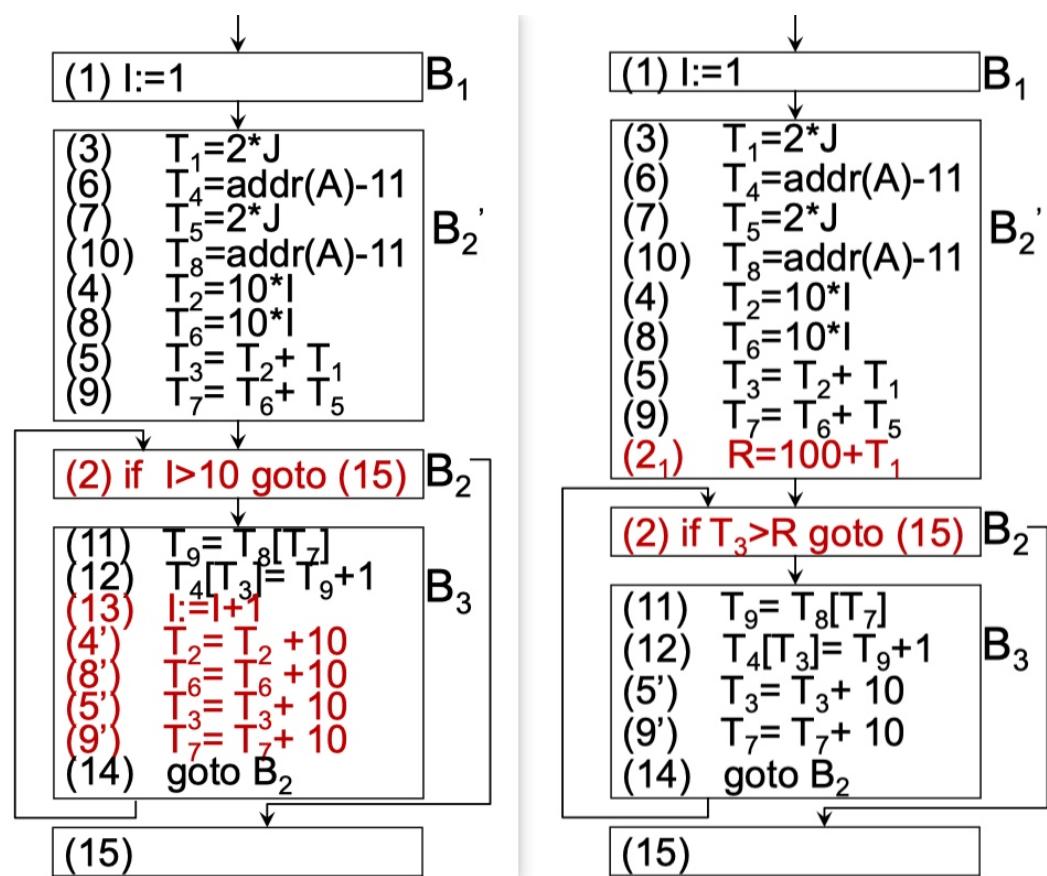
- 具体算法

### 删除归纳变量

- ▶ 删除归纳变量在强度削弱以后进行
- ▶ 强度削弱和删除归纳变量的统一算法框架

1. 利用循环不变运算信息，找出循环中所有基本归纳变量。
2. 找出所有其它归纳变量A，并找出A与已知基本归纳变量X的同族线性函数关系  $F_A(X)$ 。
3. 对2中找出的每一归纳变量A，进行强度削弱。
4. 删除对归纳变量的无用赋值。
5. 删除基本归纳变量。如果基本归纳变量B在循环出口之后不是活跃的，并且在循环中，除在其自身的递归赋值中被引用外，只在形如 `if B rop Y goto L` 的语句中被引用，则可选取一与B同族的归纳变量M来替换B进行条件控制。最后删除循环中对B的递归赋值的代码。

- 举例



## 目标代码生成

### 一、目标代码生成概述

#### 1.1 任务

将分析、翻译、优化后的中间代码转换成目标代码

#### 1.2 输入

- 源程序的中间表示，以及符号表中的信息
- 类型检查均已完成

#### 1.3 输出

- 绝对指令代码：能够立即执行的机器语言代码，所有地址已经定位
- 可重新定位指令代码：机器语言模块，需要与运行程序链接才能执行
- 汇编指令代码：需要汇编程序转换成可执行机器语言代码

### 二、抽象计算机模型

类型	指令形式	意义
直接地址型	$op\ R_i, M$	$(R_i) op (M) \Rightarrow R_i$
寄存器型	$op\ R_i, R_j$	$(R_i) op (R_j) \Rightarrow R_i$
变址型	$op\ R_i, c(R_j)$	$(R_i) op ((R_j)+c) \Rightarrow R_i$
间接型	$op\ R_i, *M$ $op\ R_i, *R_j$ $op\ R_i, *c(R_j)$	$(R_i) op ((M)) \Rightarrow R_i$ $(R_i) op ((R_j)) \Rightarrow R_i$ $(R_i) op (((R_j)+c)) \Rightarrow R_i$

- ▶  $op$  可以是常见的二目运算符
  - ▶ 如 ADD(加)、SUB(减)、MUL(乘)、DIV(除)
- ▶  $op$  也可以是一目运行符
  - ▶  $op\ R_i, M$  的意义为:  $op\ (M) \Rightarrow R_i$

其中  $(M)$  表示主存中  $M$  位置的数据。

## 其它指令

指 令	意 义
LD $R_i, B$	把B单元的内容取到寄存器R, 即 $(B) \Rightarrow R_i$
ST $R_i, B$	把寄存器 $R_i$ 的内容存到B单元, 即 $(R_i) \Rightarrow B$
J X	无条件转向X单元
CMP A, B	比较A单元和B单元的值, 根据比较情况设置内部二位特征寄存器CT的值: 根据 $A < B$ 或 $A = B$ 或 $A > B$ 分别置CT为0或1或2。
J < X	若 $CT = 0$ , 转X单元
J ≤ X	若 $CT = 0$ 或 $CT = 1$ , 转X单元
J = X	若 $CT = 1$ , 转X单元
J ≠ X	若 $CT \neq 1$ , 转X单元
J > X	若 $CT = 2$ , 转X单元
J ≥ X	若 $CT = 2$ 或 $CT = 1$ , 转X单元

## 三、代码生成

### 3.1 代码生成原则

- 以基本块为单位生成目标代码
  - 依次把四元式的中间代码转换成目标代码

- 在基本块的范围内考虑如何充分利用寄存器
- 进入基本块时，所有寄存器空闲
- 离开基本块时，把存在寄存器中的现行的值存回主存中，释放所有寄存器
- 不特别说明，所有说明变量在基本块出口之后均为非活跃变量
- 在一个基本块的范围内考虑充分利用寄存器
  -

### 要做到...

**尽可能留**: 在生成计算某变量值的目标代码时，尽可能让该变量保留在寄存器中

**尽可能用**: 后续的目标代码尽可能引用变量在寄存器中的值，而不访问内存

**及时腾空**: 在离开基本块时，把存在寄存器中的现行的值放到主存中

### 要知道...

**四元式指令**: 每条指令中各变量在将来会被使用的情况

**变量**: 每个变量现行值的存放位置

**寄存器**: 每个寄存器当前的使用状况

## 3.2 待用信息和活跃信息

### 3.2.1 概念解释

**活跃信息** – 将来会不会被引用

**待用信息** – 将来在什么地方会被引用

### 3.2.2 符号表示

- 二元组  $(x, x)$  表示变量的待用信息和活跃信息
  - 第 1 元:  $i$  表示将会在第  $i$  行被引用、 $^$  表示非待用
  - 第 2 元:  $y$  表示活跃， $^$  表示非活跃
- 信息变化
  - $(x, x) \rightarrow (x, x)$ , 用后者更新前者

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

- 计算方法

  - 

1. 把基本块中各变量的符号表中的待用信息栏填为“**非待用**”，并根据**该变量在基本块出口之后**是不是活跃的，把其中的活跃信息栏填为“**活跃**”或“**非活跃**”；
2. 从基本块出口到入口**由后向前**依次处理各个四元式*i:A:=B op C*：
  - 1) 把符号表中变量A的待用信息和活跃信息附加到四元式*i*上；
  - 2) 把符号表中A的待用信息和活跃信息分别置为“**非待用**”和“**非活跃**”；
  - 3) 把符号表中变量B和C的待用信息和活跃信息附加到四元式*i*上；
  - 4) 把符号表中B和C的待用信息均置为*i*，活跃信息均置为“**活跃**”；

变量名	初始状态→信息链	序号	四元式	左值	左操作数	右操作数
T	(^,y)	j	A:=B op T			
A	(^,^) →(n,y)	...	...	...	...	...
B	(^,^)	i	A:=B op C			
C	(^,^)	...	...	...	...	...

### 3.3 变量地址、寄存器描述

#### 3.3.1 目的

推出 AVALUE、RVALUE 目的：

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

### 3.3.2 概念

- AVALUE (变量地址描述数组)
  - 动态记录各变量现行值的存放位置
  - $AVALUE[A] = \{R1, R2, A\}$
- RVALUE (寄存器描述数组)
  - 动态记录各寄存器的使用信息
  - $RVALUE[R] = \{A, B\}$

注意，对于四元式  $A := B$

- 如果  $B$  的现行值在某寄存器  $R_i$  中，则无须生成目标代码
- 只须在  $RVALUE(R_i)$  中增加一个  $A$ ，并把  $AVALUE(A)$  改为  $R_i$

## 3.4 代码生成算法

### 3.4.1 算法描述

- 总体流程

▶ 对每个四元式:  $i: A := B \text{ op } C$ , 依次执行:

1. 以四元式:  $i: A := B \text{ op } C$  为参数, 调用函数过程  $GETREG(i: A := B \text{ op } C)$ , 返回一个寄存器  $R$ , 用作存放  $A$  的寄存器。
2. 利用  $AVALUE[B]$  和  $AVALUE[C]$ , 确定  $B$  和  $C$  现行值的存放位置  $B'$  和  $C'$ 。如果其现行值在寄存器中, 则把寄存器取作  $B'$  和  $C'$
3. 如果  $B' \neq R$ , 则生成目标代码:  
 $LD R, B'$   
 $op R, C'$   
否则生成目标代码  $op R, C'$   
如果  $B'$  或  $C'$  为  $R$ , 则删除  $AVALUE[B]$  或  $AVALUE[C]$  中的  $R$ 。
4. 令  $AVALUE[A] = \{R\}$ ,  $RVALUE[R] = \{A\}$ 。
5. 若  $B$  或  $C$  的现行值在基本块中不再被引用, 也不是基本块出口之后的活跃变量, 且其现行值在某寄存器  $R_k$  中, 则删除  $RVALUE[R_k]$  中的  $B$  或  $C$  以及  $AVALUE[B]$  或  $AVALUE[C]$  中的  $R_k$ , 使得该寄存器不再为  $B$  或  $C$  占用。

- 寄存器分配算法

## 寄存器分配算法

1. 尽可能用B独占的寄存器
2. 尽可能用空闲寄存器
3. 抢占非空闲寄存器

► 寄存器分配： $\text{GETREG}(i: A := B \text{ op } C)$  返回一个用来存放A的值的寄存器

1. 如果B的现行值在某个寄存器 $R_i$ 中， $\text{RVALUE}[R_i]$ 中只包含B，此外，或者B与A是同一个标识符，或者B的现行值在执行四元式 $A := B \text{ op } C$ 之后不会再引用，则选取 $R_i$ 为所需要的寄存器R，并转4；
2. 如果有尚未分配的寄存器，则从中选取一个 $R_i$ 为所需要的寄存器R，并转4；
3. 从已分配的寄存器中选取一个 $R_i$ 为所需要的寄存器R。最好使得 $R_i$ 满足以下条件：占用 $R_i$ 的变量的值也同时存放在该变量的贮存单元中，或者在基本块中要在最远的将来才会引用到或不会引用到。为 $R_i$ 中的变量生成必要的存数指令。
4. 给出R，返回。

## 生成存数指令

- 对 $\text{RVALUE}[R_i]$ 中每一变量M，如果M不是A，或者如果M是A又是C，但不是B并且B也不在 $\text{RVALUE}[R_i]$ 中，则
- (1) 如果 $\text{AVALUE}[M]$ 不包含M，则生成目标代码  $\text{ST } R_i, M$
  - (2) 如果M是B，或者M是C但同时B也在 $\text{RVALUE}[R_i]$ 中，则令 $\text{AVALUE}[M] = \{M, R_i\}$ ，否则令 $\text{AVALUE}[M] = \{M\}$
  - (3) 删除 $\text{RVALUE}[R_i]$ 中的M

### 3.4.2 算法举例

- 为基本块生成代码示例

## 为基本块生成

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

中间代码	目标代码	RVALUE	AVALUE
T:=A - B	LD R <sub>0</sub> , A SUB R <sub>0</sub> , B	R <sub>0</sub> 含有T	T在R <sub>0</sub> 中
U:=A - C	LD R <sub>1</sub> , A SUB R <sub>1</sub> , C	R <sub>0</sub> 含有T R <sub>1</sub> 含有U	T在R <sub>0</sub> 中 U在R <sub>1</sub> 中
V:=T + U	ADD R <sub>0</sub> , R <sub>1</sub>	R <sub>0</sub> 含有V R <sub>1</sub> 含有U	V在R <sub>0</sub> 中 U在R <sub>1</sub> 中
W:=V + U	ADD R <sub>0</sub> , R <sub>1</sub> ST R <sub>0</sub> , W	R <sub>0</sub> 含有W	W在R <sub>0</sub> 中

## 四、DAG的目标代码

作用：重排中间代码，减少目标代码

算法：

1. 构建 DAG，结点标记写在结点圆圈中，叶结点未编号，内部结点的编号写在各结点下面
2. 倒序填写中间代码编号，找到无父节点的结点，向左进行递归拓扑，遇到子结点或无法拓扑则停止

举例：

- 中间代码

```

T1 := A+B
T2 := A-B
F := T1*T2
T1 := A-B
T2 := A-C
T3 := B-C
T1 := T1*T2
G := T1*T3
    
```

- DAG图

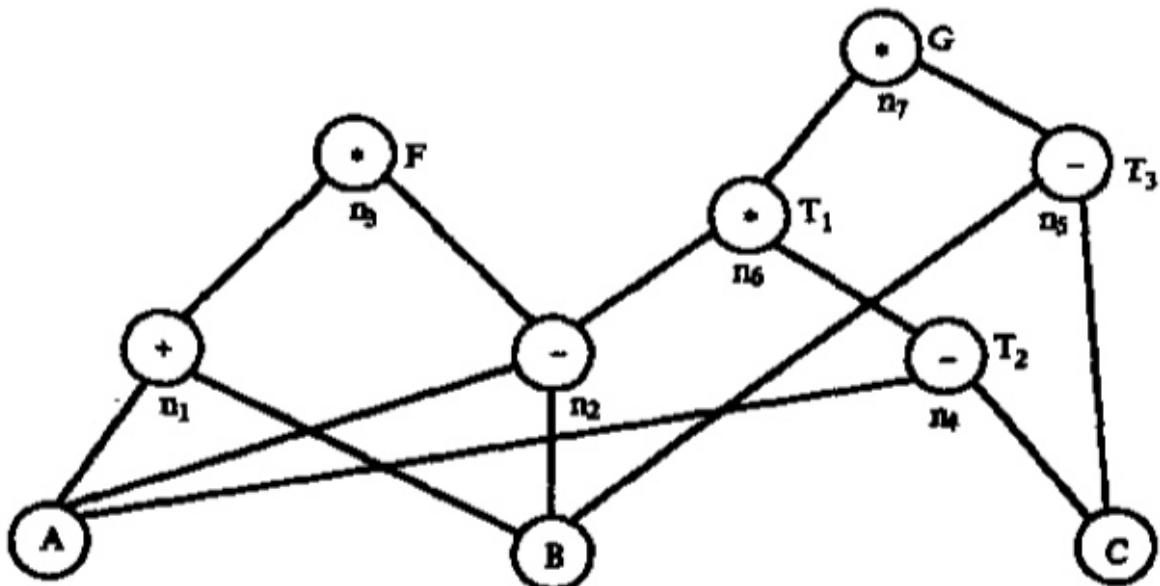


图 11.9 DAG

- 节点重排

图 11.9 的 DAG 含有 7 个内部结点, 我们应用前述算法把它们排序, 主要步骤如下。

第 1 步置初值:  $i = 7$ ;  $T$  的所有元素全为 null。内结  $n_3$  和  $n_7$  均满足第 3 步的要求, 假定选取  $T[7] = n_3$ 。结点  $n_3$  的最左子结(内结)  $n_1$  满足第 5 步的要求, 因此, 按第 6 步,  $T[6] = n_1$ 。但  $n_1$  的最左子结  $A$  为叶结, 不满足第 5 步的要求。现在只有  $n_7$  满足第 3 步的要求, 于是  $T[5] = n_7$ 。结点  $n_7$  的最左子结  $n_6$  满足第 5 步的条件, 因此,  $T[4] = n_6$ 。结点  $n_6$  的最左子结  $n_2$  同样满足第 5 步的要求, 因此,  $T[3] = n_2$ 。目前, 满足第 3 步要求的结点尚有  $n_4$  和  $n_5$ , 假定选取  $T[2] = n_4$ 。当最后把  $n_5$  列入  $T[1]$  后, 算法工作结束。至此, 我们所求出的图 11.9 的内结点顺序为

$n_5, n_4, n_2, n_6, n_7, n_1, n_3$

按上述结点次序可把图 11.9 的 DAG 重新表示为中间代码序列  $G'_1$ :

$$T_3 := B - C$$

$$T_2 := A - C$$

$$S_1 := A - B$$

$$T_1 := S_1 * T_2$$

$$G := T_1 * T_3$$

$$S_2 := A + B$$

$$F := S_2 * S_1$$

如果应用前述简单代码生成算法, 分别生成中间代码序列  $G_1$  和  $G'_1$  的目标代码, 将会进一步看到  $G'_1$  的目标代码优于  $G_1$  的目标代码, 这里从略。