

# 编译原理与技术



*LI Wensheng, SC, BUPT*



各位好!

我是李文生，很高兴认识大家!

我的联系方式:

北京邮电大学 计算机学院

[wenshli@bupt.edu.cn](mailto:wenshli@bupt.edu.cn)

010-62282929

- **使用教材：《编译原理与技术》**

清华大学出版社

- **课程安排：**

总学时：64（讲课48学时 + 实验16学时）

小学期：1.5周（设计一个小型的编译器）

- **主要参考书：**

《编译原理》 Alfred V. Aho等 著，赵建华等 译

机械工业出版社 2009. 1

《现代编译原理——C语言描述》 Andrew W. Appel 著

人民邮电出版社 2005. 9

## ■ 先修课程:

形式语言与自动机、算法与数据结构

程序设计语言: Pascal、C

## ■ 相关课程:

操作系统原理、计算机组成与体系结构

汇编语言、软件工程

## ■ 学习方法:

预习、听课、复习、答疑、实验

## ■ 答疑时间、地点:

时间: 每周日下午3: 00-5: 00      地点: 教3-623

## ■ 助教:

■ 刘天骄: 13426081665    rockets1131@gmail.com

■ 张 闯: 18511599276    290025723@qq.com

# 本课程内容及时安排

§ 1.	编译概述	2 学时
§ 2.	形式语言与自动机	(自己复习)
§ 3.	词法分析	4 学时
§ 4.	语法分析	12 学时
§ 5.	语法制导翻译技术	10 学时
§ 6.	类型检查	4 学时
§ 7.	运行时刻环境	6 学时
§ 8.	中间代码生成	4 学时
§ 9.	代码生成	4 学时
§ 10.	代码优化	2 学时

# 课程重点及难点

## ■ 重点:

- 语法分析
- 语法制导翻译
- 类型检查
- 运行时刻环境
- 中间代码生成

## ■ 难点:

- 语法分析
- 语法制导翻译

# 成绩评定

- 平时成绩：20 % （包括：作业+实验）
- 期中测试：20 %
- 期末测试：60 %

# 第1章 编译概述



*LI Wensheng, SCST, BUPT*

知识点： 翻译和解释的概念  
编译的阶段、任务、及典型结构  
编译程序的伙伴工具

# 编译概述

## 简介

- 1. 1 翻译和解释
- 1. 2 编译的阶段和任务
- 1. 3 编译有关的其他概念
- 1. 4 编译程序的伙伴工具
- 1. 5 编译原理的应用

## 小结



# 简介

- 什么是编译？
  - ◆ 把源程序转换成等价的目标程序的过程即是编译。
- 编译程序的设计涉及到的知识：
  - ◆ 程序设计语言
  - ◆ 形式语言与自动机理论
  - ◆ 计算机体系结构
  - ◆ 数据结构
  - ◆ 算法分析与设计
  - ◆ 操作系统
  - ◆ 软件工程等

# 1.1 翻译和解释

一、程序设计语言

二、翻译程序

# 一、程序设计语言

## ■ 低级语言

- ◆ 机器语言
- ◆ 符号语言    汇编语言

## ■ 高级语言

- ◆ 过程性语言—面向用户的语言 如：C、Pascal
- ◆ 专用语言—面向问题的语言 如：SQL
- ◆ 面向对象的语言 如：Java、C++

# 高级语言的优点

- 高级语言独立于机器。所编程序移植性比较好。
- 不必考虑存储单元的分配问题，数据的外部形式转换成机器的内部形式等细节。
  - ◆ 用变量描述存储单元
- 具有丰富的数据结构和控制结构。
  - ◆ 数据结构：数组、记录等
  - ◆ 控制结构：循环、分支、过程调用等。
- 更接近于自然语言。
  - ◆ 可读性好，便于维护。
- 编程效率高。

## 二、翻译程序

- 翻译程序扫描所输入的源程序，并将其转换为目标程序，或将源程序直接翻译成结果。



- 翻译程序分为两大类：
  - ◆ 编译程序(即编译器)：把源程序翻译成目标程序的翻译程序。
  - ◆ 解释程序(即解释器)：直接执行源程序的翻译程序。

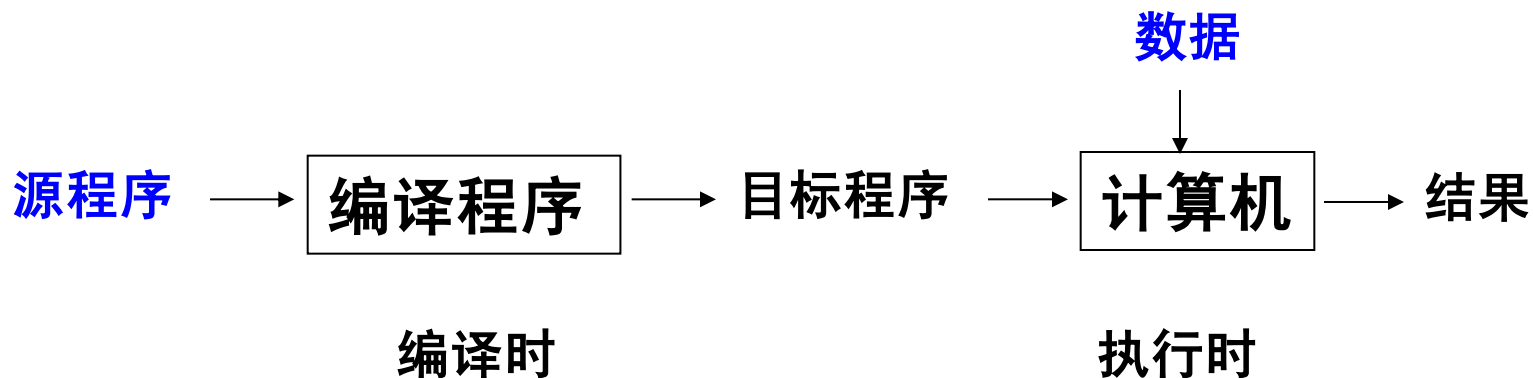
# 编译程序

- 源程序是用高级语言或汇编语言编写的，目标程序是用目标语言表示的。
- 两类编译程序：
  - ◆ 汇编程序
  - ◆ 编译程序

汇编语言程序 → **汇编程序** → 机器语言程序

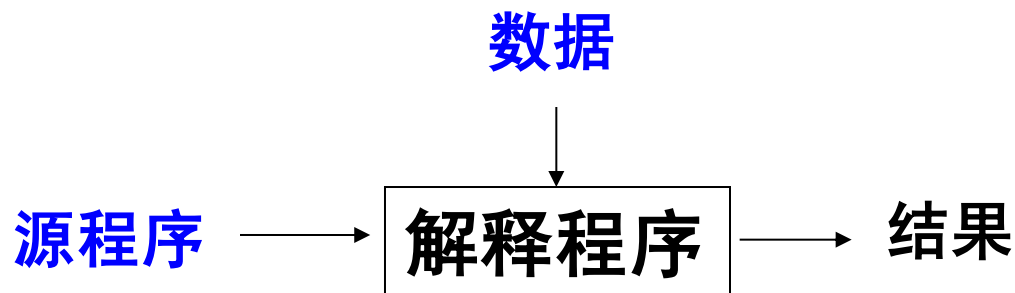
高级语言程序 → **编译程序** → 低级语言程序

# 编译和执行阶段



- **编译时间：**实现源程序到目标程序的转换所占用的时间。
- **源程序和数据**是在不同时间（即分别在编译阶段和运行阶段）进行处理的。

# 解释程序

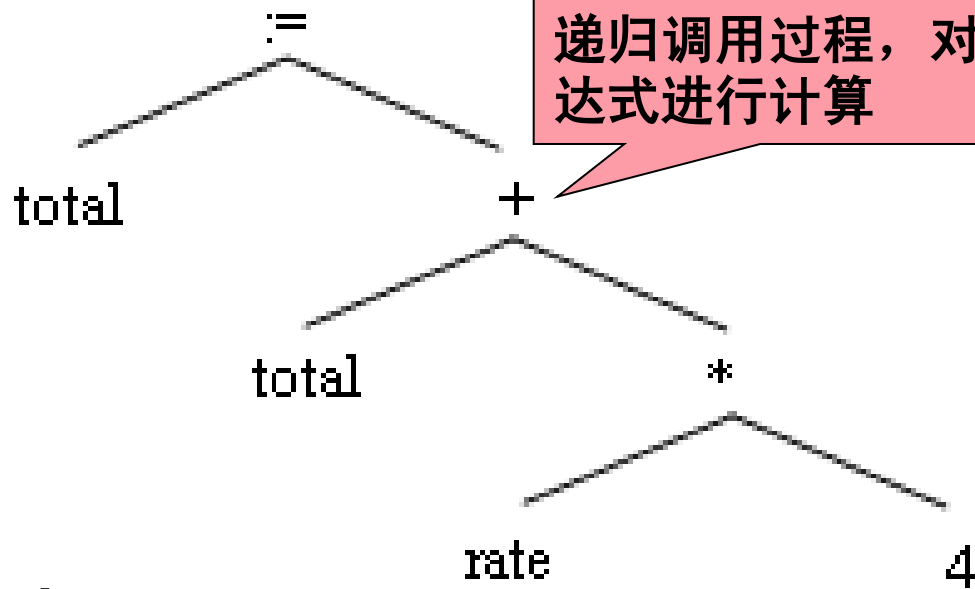


- 解释程序解释执行源程序，不生成目标程序
- 同时处理源程序和数据
- 一种有效的方法：先将源程序转换为某种中间形式，然后对中间形式的程序解释执行。



# total:=total+rate\*4 的解释过程

- 解释程序先将源程序转换成一棵树



调用一个过程，执行右边的表达式，计算结果送入**total**的存储单元

递归调用过程，对表达式进行计算

- 遍历该树，执行结点上所规定的动作。

# Java语言处理器

## ■ 结合了编译和解释过程

- ◆ 编译：一个Java源程序首先被编译成一个称为字节码（bytecode）的中间表示形式。
- ◆ 解释：Java虚拟机对字节码解释执行。

## ■ 即时（Just-in-time）编译

- ◆ Java即时编译器在运行中间程序处理输入的前一刻首先把字节码翻译成为机器语言，然后再执行程序。

# 1.2 编译的阶段和任务

## 一、分析阶段

根据源语言的定义，分析源程序的结构

1. 词法分析
2. 语法分析
3. 语义分析

## 二、综合阶段

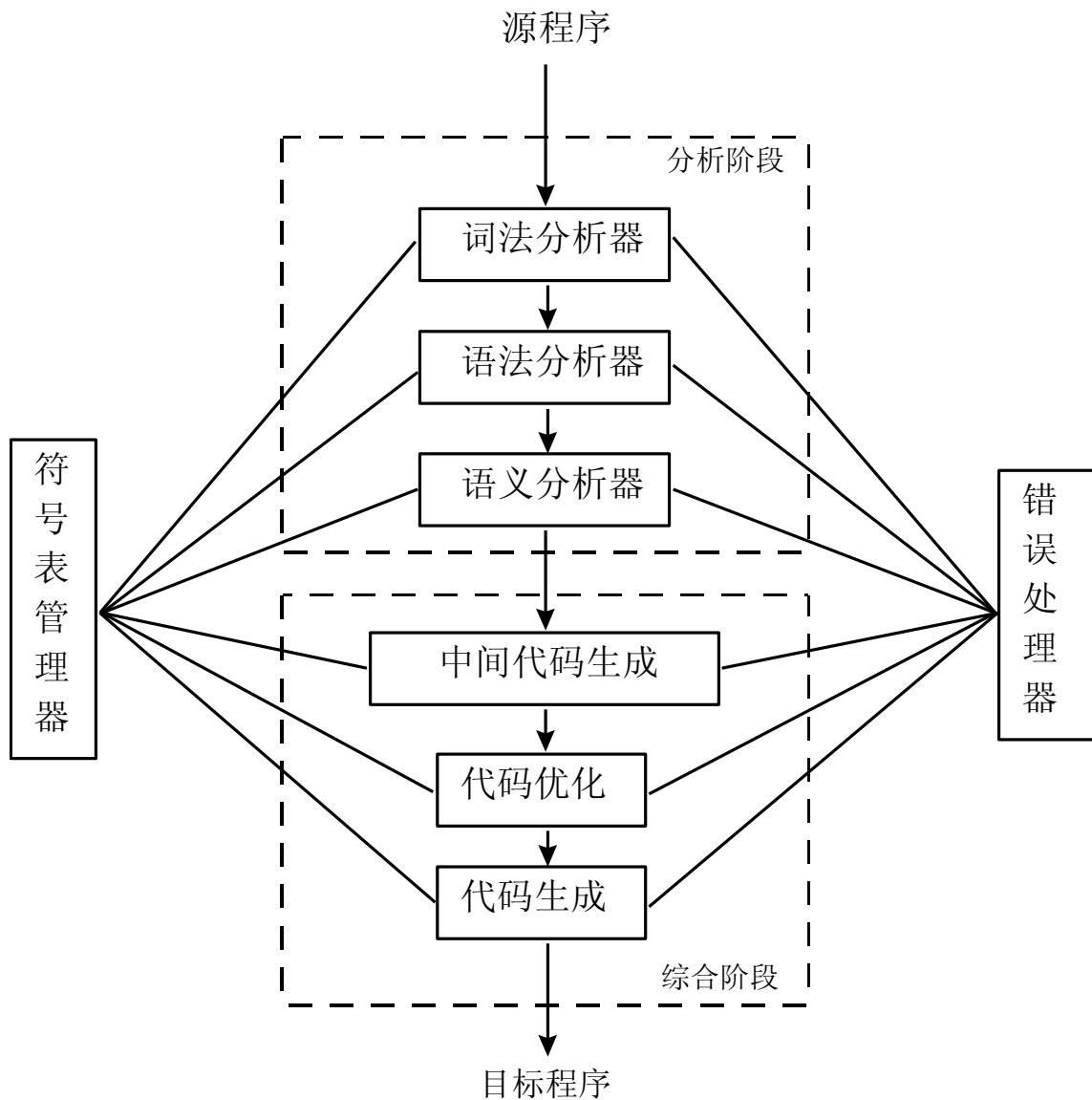
根据分析结果构造出所要求的目标程序

4. 中间代码生成
5. 代码优化
6. 目标代码生成

## 三、符号表的管理

## 四、错误诊断和处理

# 编译程序的典型结构



# 一、分析阶段

- **任务：**根据源语言的定义，对源程序进行结构分析和语义分析，从而把源程序正文转换为某种内部表示。
- **分析阶段**是对源程序结构的静态分析。
- **任务划分：**
  1. 词法分析
  2. 语法分析
  3. 语义分析

# 1. 词法分析

- 扫描，线性分析
- 词法分析器：
  - ◆ 依次读入源程序中的每个字符，对构成源程序的字符串进行分解，识别出每个具有独立意义的字符串（即单词），将其转换成记号（**token**），并组织成记号流。
  - ◆ 把需要存放的单词放到符号表中，如变量名，标号，常量等。
- 形成记号的字符串叫做该记号的单词(**lexeme**)。
- 工作依据：源语言的构词规则(即词法)，也称为模式(**pattern**)。
  - ◆ C语言的标识符的模式是：以字母或下划线开头，由字母、数字或下划线组成的符号串。

# 对 $\text{total} := \text{total} + \text{rate} * 4$ 的词法分析

(1) 标识符 **total**

(2) 赋值号 **:=**

(3) 标识符 **total**

(4) 加号 **+**

(5) 标识符 **rate**

(6) 乘号 **\***

(7) 整常数 **4**

# 空格、注释的处理及其他

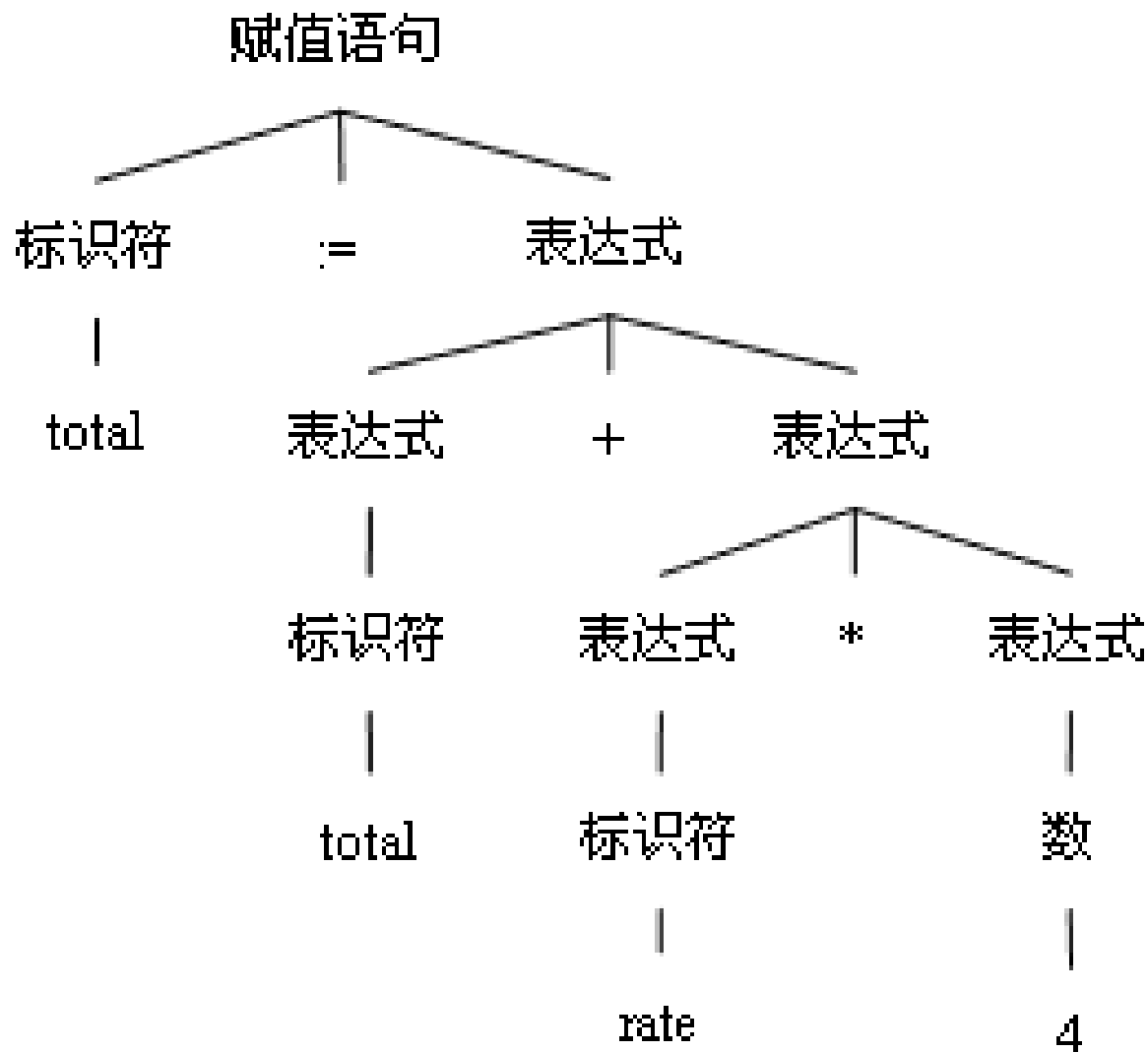
- 分隔记号的空格：被删去
- 源程序中的注释：被跳过
- 识别出来的标识符要放入符号表。
- 某些记号还要具有“属性值”
  - ◆ 如发现标识符**total**时，词法分析器不仅产生一个单词符号的类别标记如**id**，还把它的单词**total**填入符号表（如果**total**在表中不存在的话），则**total**的记号就包括两部分：单词符号的类别标记**id**、属性值（即指向符号表中**R**条目的指针）。



## 2. 语法分析

- 层次结构的分析
- 把记号流按语言的语法结构层次地分组，以形成语法短语。
- 源程序的语法短语常用分析树表示
- 工作依据：源语言的语法规则
- 程序的层次结构通常由递归的规则表示，如表达式的定义如下：
  - (1) 任何一个标识符是一个表达式
  - (2) 任何一个数是一个表达式
  - (3) 如果 $\text{expr}_1$ 和 $\text{expr}_2$ 是表达式， $\text{expr}_1 + \text{expr}_2$ 、 $\text{expr}_1 * \text{expr}_2$ 、 $(\text{expr}_1)$ 也都是表达式。

# total := total + rate \* 4 的分析树



# 语句的递归定义

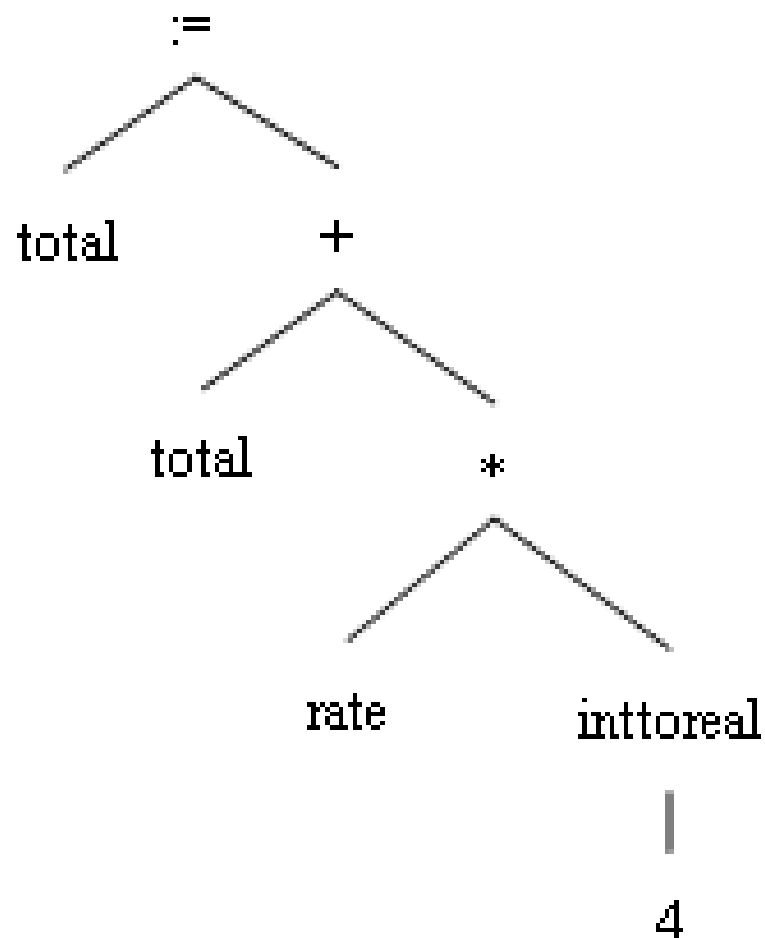
- 如果**id**是一个标识符，**expr**是一个表达式，则 **id:=expr** 是一个语句。
- 如果**expr**是表达式，**stmt**是语句，则 **while (expr) do stmt** 和 **if (expr) then stmt** 都是语句。

### 3. 语义分析

- 对语句的意义进行检查分析
- 收集类型等必要信息
- 用语法分析确定的层次结构表示各语法成份
- 工作依据：源语言的语义规则
- 一个重要任务：类型检查
  - ◆ 根据规则检查每个运算符及其运算对象是否符合要求；
  - ◆ 数组的下标是否合法；
  - ◆ 过程调用时，形参与实参个数、类型是否匹配等

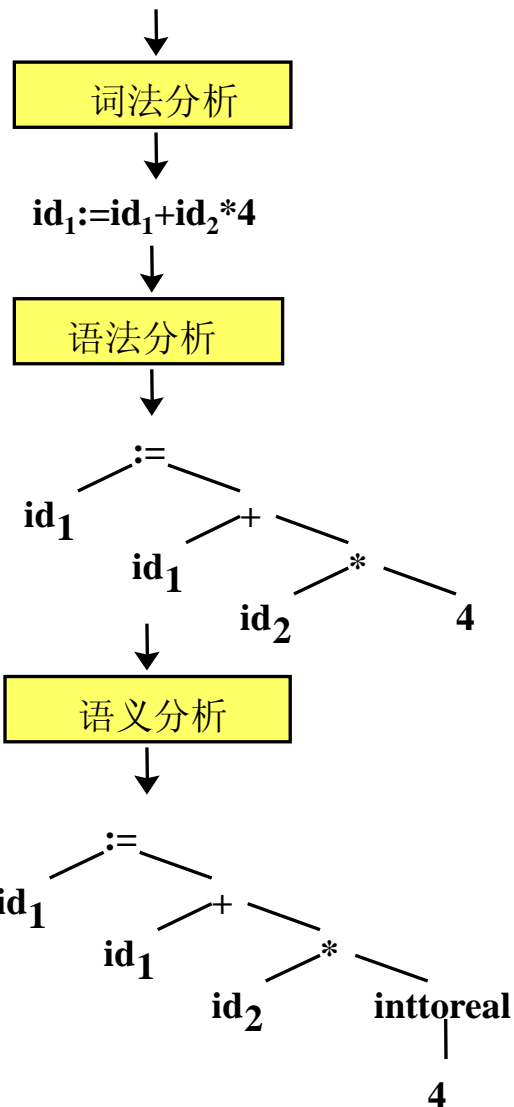
# 赋值语句 $total := total + rate * 4$

## 插入转换符的语法树



# total:=total+rate\*4 的各分析步骤及其中间结果

total:=total+rate\*4



## 二、综合阶段

- **任务：** 根据所制定的源语言到目标语言的对应关系，对分析阶段所产生的中间形式进行综合加工，从而得到与源程序等价的目標程序。
- **任务划分：**
  - (4) 中间代码生成
  - (5) 代码优化
  - (6) 目标代码生成

## 4. 中间代码生成

- 中间代码：一种抽象的机器程序
- 中间代码应具有两个重要的特点：
  - ◆ 易于产生
  - ◆ 易于翻译成目标代码
- 中间代码有多种形式
- **三地址代码**具有的特点：
  - ◆ 每条指令除了赋值号之外，最多还有一个运算符。
  - ◆ 编译程序必须生成临时变量名，以便保留每条指令的计算结果。
  - ◆ 有些“三地址”指令少于三个操作数



## $total := total + rate * 4$ 的三地址代码

$temp_1 := \text{intto real}(4)$

$temp_2 := id_2 * temp_1$

$temp_3 := id_1 + temp_2$

$id_1 := temp_3$

## 5. 代码优化

### 什么是代码优化？

对代码进行改进，使之占用的空间少、运行速度快。

代码优化首先是在中间代码上进行的。

```
temp1 := inttoreal(4)
```

```
temp2 := id2 * temp1
```

```
temp3 := id1 + temp2
```

```
id1 := temp3
```

```
temp1 := id2 * 4.0  
temp2 := id1 + temp1  
id1 := temp2
```

“优化编译程序”：

能够完成大多数优化的编译程序

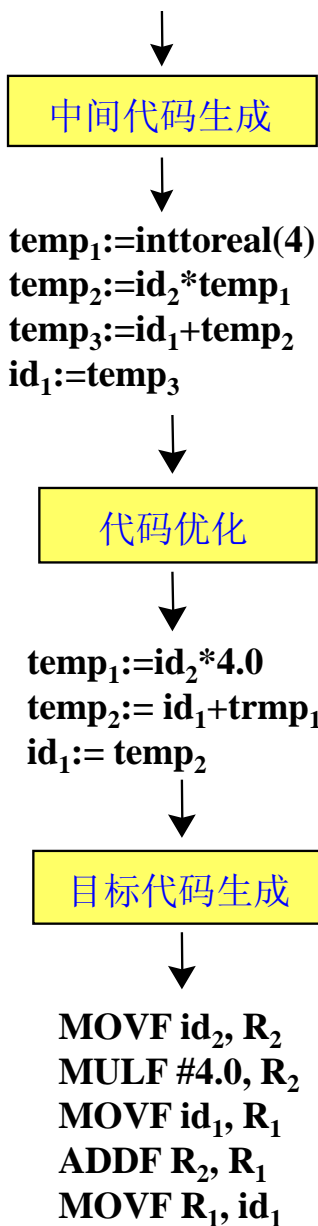
## 6. 目标代码生成

- 生成的目标代码一般是可以重定位的机器代码或汇编语言代码。
- 涉及到的两个重要问题：
  - ◆ 对程序中使用的每个变量要指定存储单元
  - ◆ 对变量进行寄存器分配

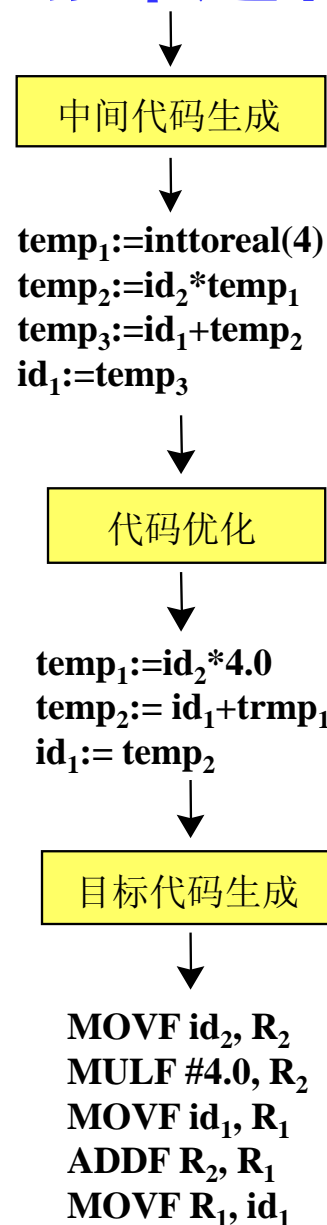
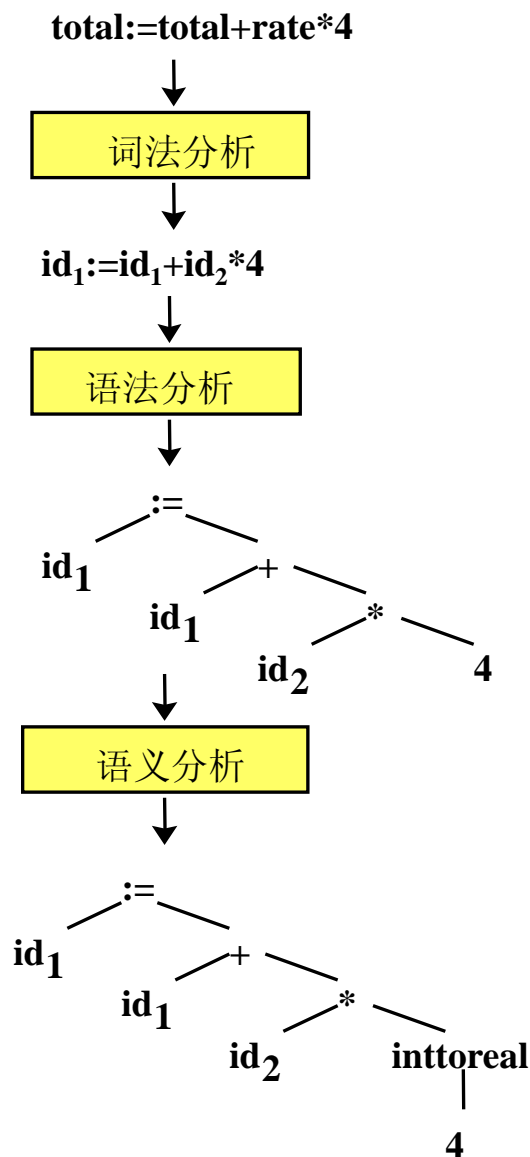
## total := total + rate \* 4 的目标代码

```
MOVF id2, R2  
MULF #4.0, R2  
MOVF id1, R1  
ADDF R2, R1  
MOVF R1, id1
```

# 赋值语句 $total := total + rate * 4$ 的各综合步骤及结果



# total:=total+rate\*4 的翻译过程



# 三、符号表管理

- 编译程序的一项重要工作：
  - ◆ 记录源程序中使用的标识符
  - ◆ 收集每个标识符的各种属性信息
- 符号表是由若干记录组成的数据结构
  - ◆ 每个标识符在表中有一条记录
  - ◆ 记录的域是标识符的属性
- 对符号表结构的要求：
  - ◆ 应允许快速地找到标识符的记录
  - ◆ 可在其中存取数据
- 标识符的各种属性是在编译的各个不同阶段填入符号表的。

# 声明语句: `float total, rate;`

## ■ 词法分析器:

- ◆ `float`是关键字
- ◆ `total`、`rate`是标识符
- ◆ 在符号表中创建这两个标识符的记录

## ■ 语义分析器:

- ◆ `total`、`rate`都表示变量
  - ◆ `float`表示这两个变量的类型
  - ◆ 可以把这两种属性及存储分配信息填入符号表。
- 在语义分析和生成中间代码时，还要在符号表中填入对这个`float`型变量进行存储分配的信息。



## 四、错误处理

- 在编译的每个阶段都可能检测到源程序中存在的错误
  - ◆ 词法分析程序可以检测出非法字符错误。
  - ◆ 语法分析程序能够发现记号流不符合语法规则的错误。
  - ◆ 语义分析程序试图检测出具有正确的语法结构，但对所涉及的操作无意义的结构。
  - ◆ 代码生成程序可能发现目标程序区超出了允许范围的错误。
  - ◆ 由于计算机容量的限制，编译程序的处理能力受到限制而引起的错误。
- 处理与恢复
  - ◆ 判断位置和性质
  - ◆ 适当的恢复

# 1.3 编译有关的其他概念

一、前端和后端

二、“遍”

# 一、前端和后端

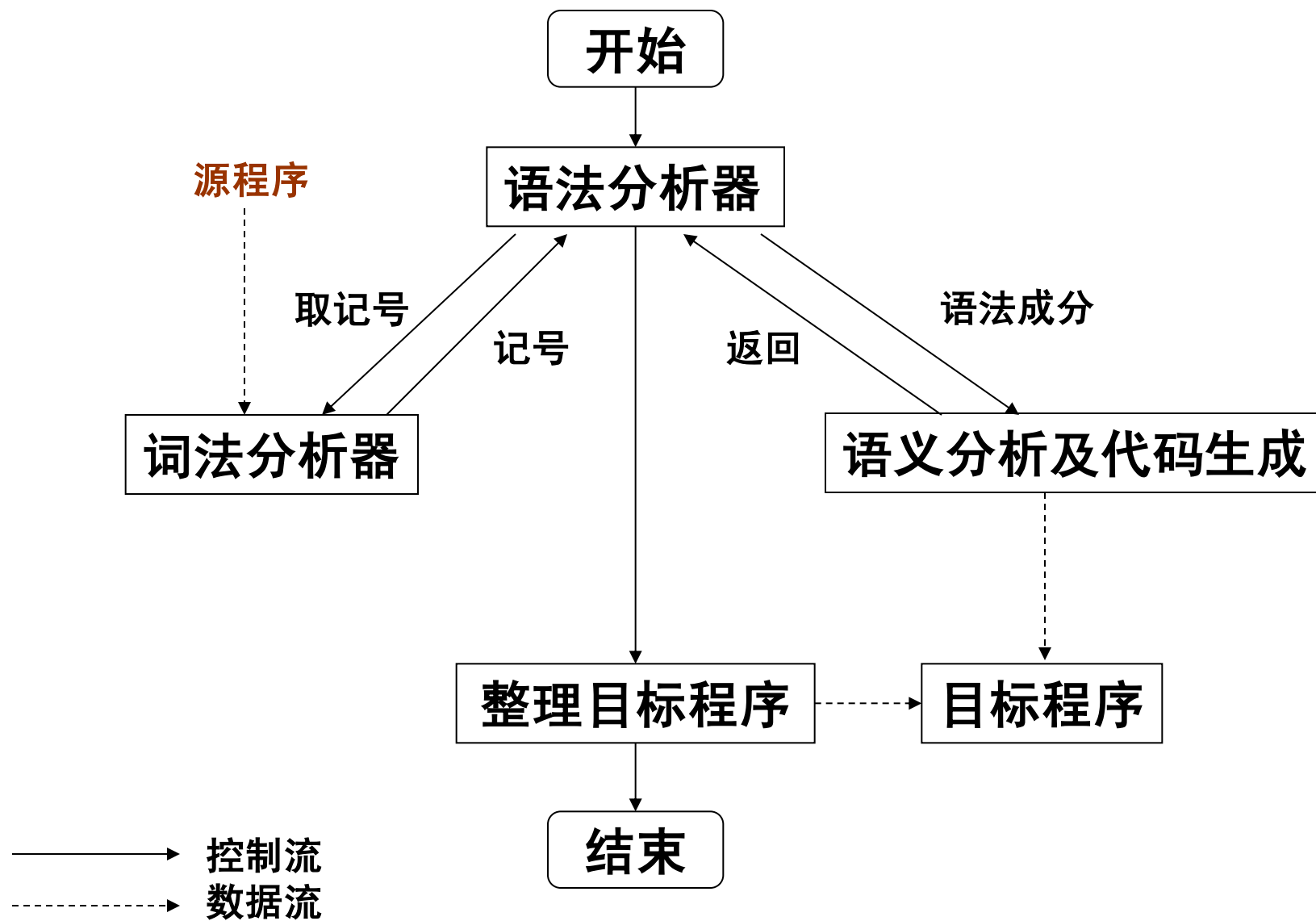
- 前端主要由与源语言有关而与目标机器无关的那些部分组成
  - ◆ 词法分析、语法分析、符号表的建立、语义分析和中间代码生成
  - ◆ 与机器无关的代码优化工作
  - ◆ 相应的错误处理工作和符号表操作
- 后端由编译程序中与目标机器有关的部分组成
  - ◆ 与机器有关的代码优化、目标代码的生成
  - ◆ 相应的错误处理和符号表操作
- 把编译程序划分成前端和后端的优点：
  - ◆ 便于移植、便于编译程序的构造。

## 二、遍

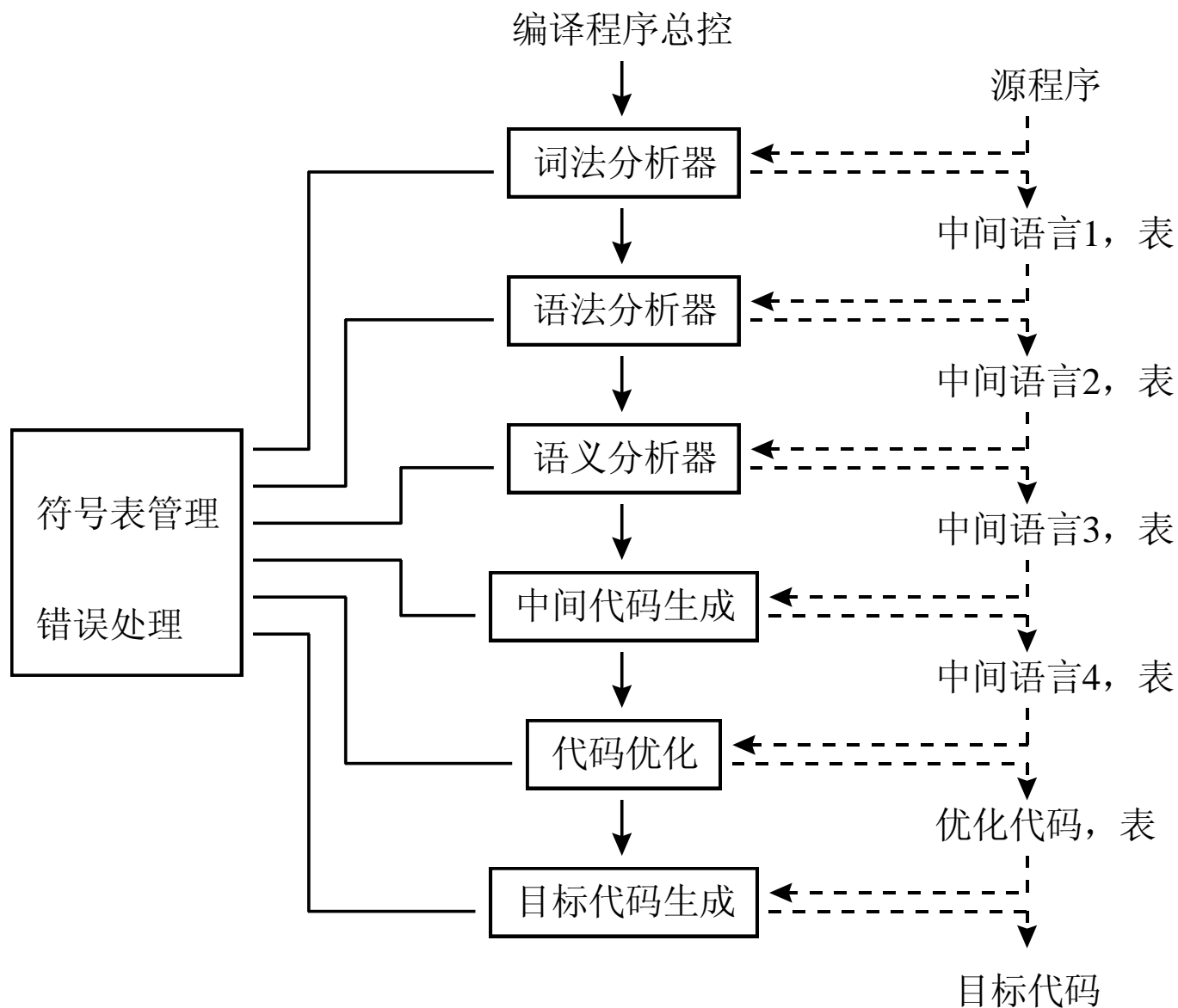
- 一“遍”是指对源程序或其中间形式从头到尾扫描一遍，并作相关的加工处理，生成新的中间形式或目标程序。
- 编译程序的结构受“遍”的影响
  - ◆ 遍数
  - ◆ 分遍方式

一遍扫描的编译程序  
多遍编译程序

# 一遍扫描的编译程序



# 多遍编译程序



# 编译程序分遍的优缺点

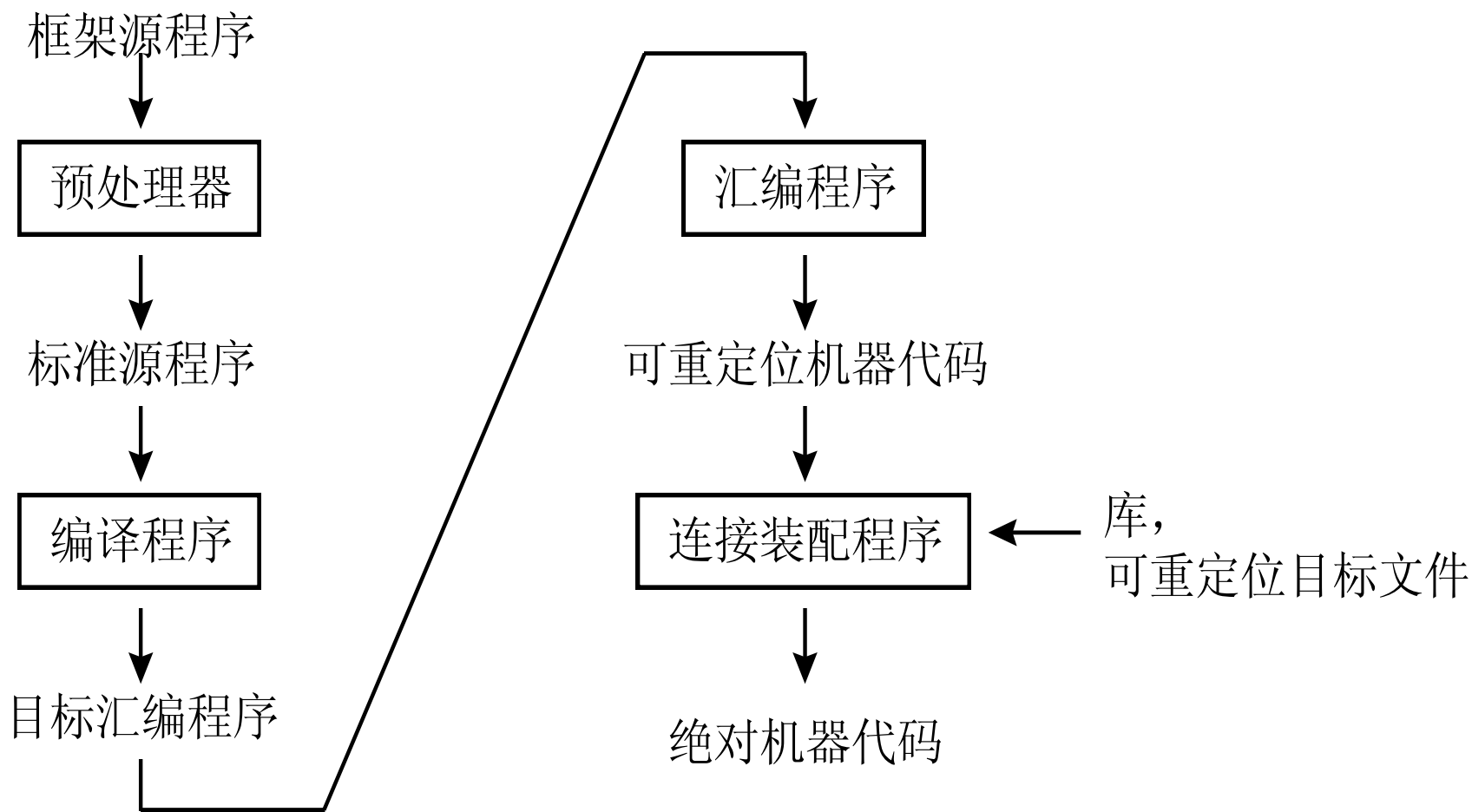
## ■ 分遍的主要好处:

- ◆ 可以减少对主存容量的要求
- ◆ 可使各遍编译程序功能独立、单纯，相互联系简单，编译程序结构清晰。
- ◆ 能够实现更充分的优化工作，获得高质量目标程序。
- ◆ 通过分遍将编译程序的前端和后端分开，可以为编译程序的移植创造条件。

## ■ 分遍的缺点:

- ◆ 增加了不少重复性的工作。

# 1.4 编译程序的伙伴工具





# 编译程序的前后处理器

- 一、预处理器
- 二、汇编程序
- 三、连接装配程序

# 一、预处理器

- 预处理器的主要功能:

1. 宏处理
2. 文件包含
3. 语言扩充

# 1. 宏处理器

- 预处理器允许用户在源程序中定义宏。

- C语言源程序中的一个宏定义：

```
#define prompt(s) fprintf(stderr, s)
```

- 宏处理器处理两类语句，即宏定义和宏调用。

- ◆ **宏定义**通常用统一的字符或关键字表示，如define或macro，宏定义由宏名字及宏体组成，通常宏处理器允许在宏定义中使用形参。
- ◆ **宏调用**由调用宏的命令名（宏名）和所提供的实参组成。宏处理器用实参代替宏体中的形参，再用变换后的宏体替换宏调用本身。

## 2. 文件包含

- 预处理器把文件的包含声明扩展为程序正文。
- C语言程序中的“头文件”包含声明行：  
`#include <stdio.h>`
- 预处理器处理到该语句时，就用文件stdio.h的内容替换此语句。

### 3. 语言扩充

- 有些预处理器用更先进的控制流和数据结构来增强原来的语言。
- 例如：
  - ◆ 预处理器可以将类似于while或if-then-else语句结构的内部宏提供给用户使用，而这些结构在原来的程序设计语言中是没有的。
  - ◆ 当程序中使用了这样的结构时，由预处理器通过宏调用实现语言功能的扩充。

## 二、汇编程序

- 汇编语言用助记符表示操作码，用标识符表示存储地址。
- 赋值语句 $b:=a+2$ 相应的汇编语言程序为：  
    MOV a, R<sub>1</sub>  
    ADD #2, R<sub>1</sub>  
    MOV R<sub>1</sub>, b
- 最简单的汇编程序对输入作两遍扫描。

# 第一遍

- 找出标志存贮单元的所有标识符，并将它们存贮到**汇编符号表**中。
  - ◆ 汇编符号表独立于编译程序的符号表
- 在符号表中指定该标识符所对应的存储单元地址，此地址是在首次遇到该标识符时确定的。
- 假定一个字包括4个字节，每个变量占一个字，完成第一遍扫描后，得到汇编符号表：

标识符	地址
a	0
b	4

## 第二遍

- 把每个用助记符表示的操作码翻译为二进制表示的机器代码。
- 把用标识符表示的存储地址翻译为汇编符号表中该标识符所对应的地址。
- 输出通常是可重定位的机器代码。
  - ◆ 起始地址为0，各条指令及其所访问的地址都是相对于0的逻辑地址。
  - ◆ 当装入内存时，可以指定任意的地址L作为开始单元。
- 输出中要对那些需要重定位的指令做出标记
  - ◆ 标记供装入程序识别，以便计算相应的物理地址。



# 可重定位机器代码

**b:=a+2的汇编代码**  
**MOV a, R<sub>1</sub>**  
**ADD #2, R<sub>1</sub>**  
**MOV R<sub>1</sub>, b**

## ■ 假定:

0001 代表 MOV S, R

0011 代表 ADD

0010 代表 MOV R, D

## ■ 假定机器指令的格式为:

操作符    寄存器    寻址模式    地址

## ■ 第二遍输出的可重定位机器代码:

0001    01    00    00000000\*

0011    01    10    00000010

0010    01    00    00000100\*

# 绝对机器代码

可重定位机器代码:

0001 01 00 00000000\*

0011 01 10 00000010

0010 01 00 00000100\*

- 假如装入内存的起始地址为L=00001111
- 则a和b的地址分别是15和19
- 则装入后的机器代码为:

0001 01 00 00001111

0011 01 10 00000010

0010 01 00 00010011

# 三、连接装配程序

## ■ 装入

- ◆ 读入可重定位的机器代码
- ◆ 修改重定位地址
- ◆ 把修改后的指令和数据放在内存的适当地方或形成可执行文件

## ■ 连接

- ◆ 把几个可重定位的机器代码文件连接成一个可执行的程序

# 1.5 编译原理的应用

- 语法制导的结构化编辑器
- 程序格式化工具
- 软件测试工具
- 程序理解工具
- 高级语言的翻译工具

# 语法制导的结构化编辑器

- 具有通常的正文编辑和修改功能
- 能象编译程序那样对源程序进行分析，把恰当的层次结构加在程序上。
- 可以保证源程序
  - ◆ 无语法错误
  - ◆ 有统一的可读性好的程序格式
- 结构化编辑器能够执行一些对编制源程序有用的附加任务，如：
  - ◆ 检查用户的输入是否正确
  - ◆ 自动提供关键字
  - ◆ 从BEGIN或左括号跳到与其相匹配的END或右括号

# 程序格式化工具

- 读入并分析源程序
- 使程序结构变得清晰可读，如：
  - ◆ 用缩排方式表示语句的嵌套层次结构
  - ◆ 用一种专门的字型表示注释等

# 软件测试工具

## 静态测试 —— 静态测试器

读入源程序

在不运行该程序的情况下对其进行分析，以发现程序中潜在的错误或异常。

不可能执行到的死代码

未定义就引用的变量

试图使用一个实型变量作为指针等。

## 动态测试 —— 动态测试器

利用测试用例实际执行程序

记录程序的实际执行路线

将实际运行结果与期望结果进行比较，以发现程序中的错误或异常。

# 程序理解工具

- 对源程序进行分析
- 确定各模块间的调用关系
- 记录程序数据的静态属性和结构属性
- 画出控制流程图



# 高级语言的翻译工具

- 将用某种高级语言开发的程序翻译为另一种高级语言表示的程序

# 编译技术的其他应用

## ■ 高级程序设计语言的实现

- ◆ 编译技术和语言发展互动的一个早期的例子：C语言中的关键字 `register`

## ■ 针对计算机体系结构的优化

### ◆ 并行性

- 现代微处理器采用的指令级并行性，对程序员透明，硬件自动检测顺序指令流之间的依赖关系，一个硬件调度器可以改变指令的执行顺序。

### ➤ 处理器并行

### ◆ 内存层次结构

- 高效利用寄存器、高速缓存

## ■ 新计算机体系结构的设计

### ◆ RISC

### ◆ 专用体系结构

# 小 结

- 什么是编译
- 翻译程序：
  - ◆ 编译程序（编译程序、汇编程序）、解释程序
  - ◆ 二者的异同
- 编译系统（编译环境）
- 编译程序的伙伴工具、功能及工作原理
  - ◆ 预处理器
  - ◆ 汇编程序
  - ◆ 连接装配程序

# 小结 (续)

- 编译程序的各组成部分及其功能
  - ◆ 词法分析
  - ◆ 语法分析
  - ◆ 语义分析
  - ◆ 中间代码生成
  - ◆ 代码优化
  - ◆ 目标代码生成
- 前端和后端
- 遍
- 编译程序的设计涉及到的知识

# 作业

- 1. 1
- 1. 3
- 1. 4
- 1. 5

## 第3章 词法分析



*LI Wensheng, SCST, BUPT*

**基础知识：PASCAL、C语言、正规表达式**

**正规文法、有限自动机**

**知识点：词法分析器的作用、地位**

**记号、模式**

**词法分析器的状态转换图**

# 词法分析

## 简介

- 3.1 词法分析程序与语法分析程序的关系
- 3.2 词法分析程序的输入与输出
- 3.3 记号的描述和识别
- 3.4 词法分析程序的设计与实现
- 3.5 软件工具LEX

## 小结

# 简介

- 词法分析任务由词法分析程序完成
- 本章内容安排

讨论用手工方式设计并实现词法分析程序的方法和步骤

- ◆ 词法分析程序的作用
- ◆ 词法分析程序的地位
- ◆ 源程序的输入与词法分析程序的输出
- ◆ 单词符号的描述及识别
- ◆ 词法分析程序的设计与实现

词法分析程序自动生成工具LEX简介



# 词法分析程序的作用

## ■ 词法分析程序的作用：

扫描源程序字符流

按照源语言的词法规则识别出各类单词符号

产生用于语法分析的记号序列

词法检查

创建符号表

把识别出来的标识符插入符号表中

与用户接口的一些任务：

- 跳过源程序中的注释和空白
- 把错误信息和源程序联系起来

# 3.1 词法分析程序与语法分析程序的关系

- 词法分析程序与语法分析程序之间的三种关系
  - ◆ 词法分析程序作为独立的一遍
  - ◆ 词法分析程序作为语法分析程序的子程序
  - ◆ 词法分析程序与语法分析程序作为协同程序

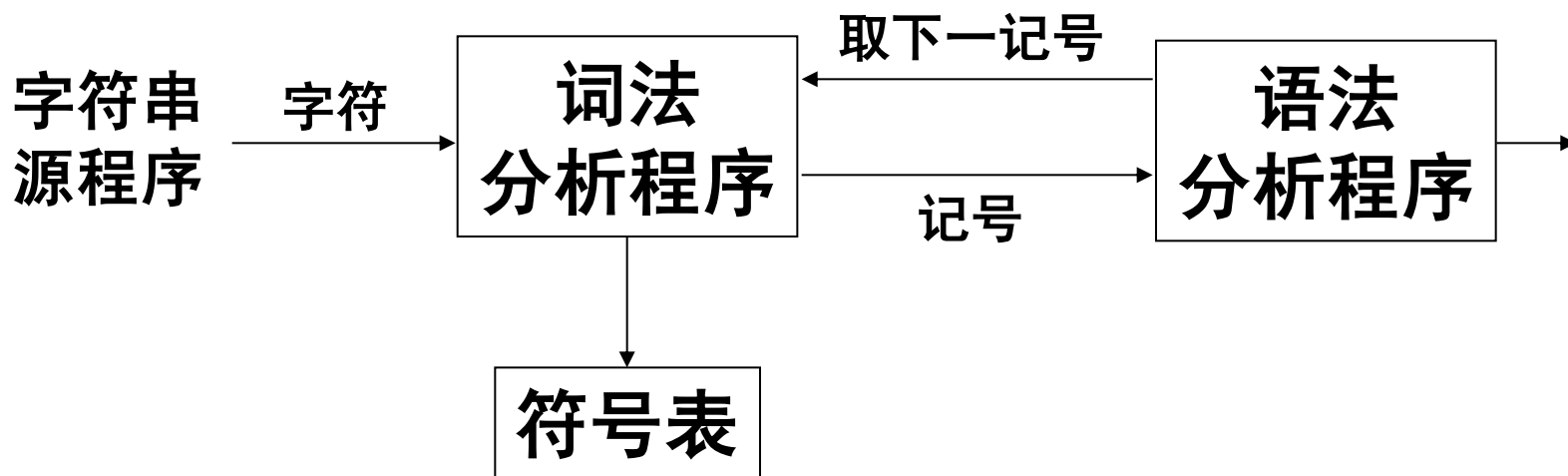
# 词法分析程序作为独立的一遍



## ■ 输出放入一个中间文件

磁盘文件  
内存文件

# 词法分析程序作为语法分析程序的子程序



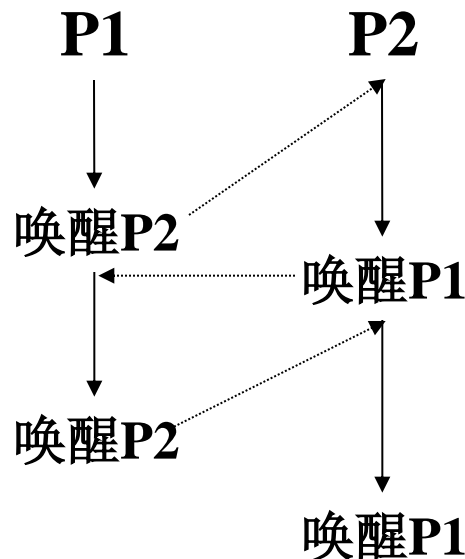
- 避免了中间文件
- 省去了取送符号的工作
- 有利于提高编译程序的效率

# 词法分析程序与语法分析程序作为协同程序

## ■ 协同程序:

如果两个或两个以上的程序，它们之间交叉地执行，这些程序称为协同程序。

词法分析程序与语法分析程序在同一遍中，以生产者和消费者的关系同步运行。



# 分离词法分析程序的好处

## ■ 可以简化设计

- ◆ 词法程序很容易识别并去除空格、注释，使语法分析程序致力于语法分析，结构清晰，易于实现。

## ■ 可以改进编译程序的效率

- ◆ 利用专门的读字符和处理记号的技术构造更有效的词法分析程序。

## ■ 可以加强编译程序的可移植性

- ◆ 在词法分析程序中处理特殊的或非标准的符号。

## 3.2 词法分析程序的输入与输出

- 一、词法分析程序的实现方法
- 二、设置缓冲区的必要性
- 三、配对缓冲区
- 四、词法分析程序的输出

# 一、词法分析程序的实现方法

- 利用词法分析程序自动生成器
  - ◆ 从基于正规表达式的规范说明自动生成词法分析程序。
  - ◆ 生成器提供用于源程序字符流读入和缓冲的若干子程序
- 利用传统的系统程序设计语言来编写
  - ◆ 利用该语言所具有的输入/输出能力来处理读入操作
- 利用汇编语言来编写
  - ◆ 直接管理源程序字符流的读入

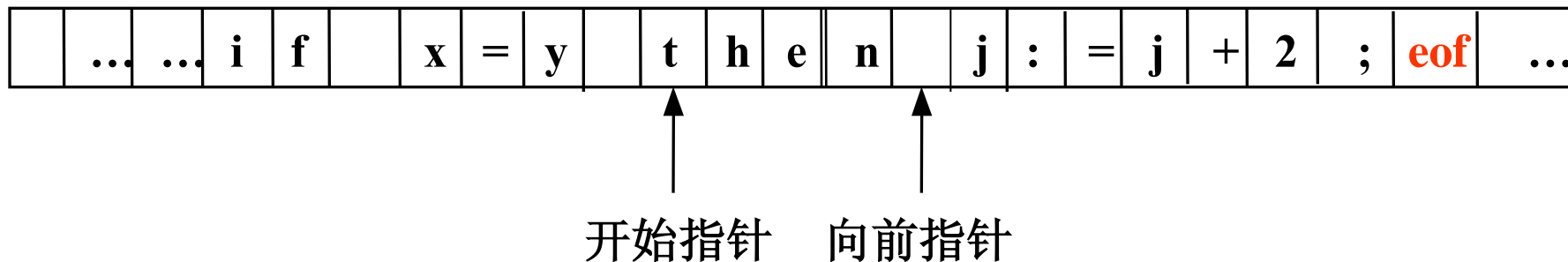


Wensheng Li BUPT

- 12

### 三、配对缓冲区

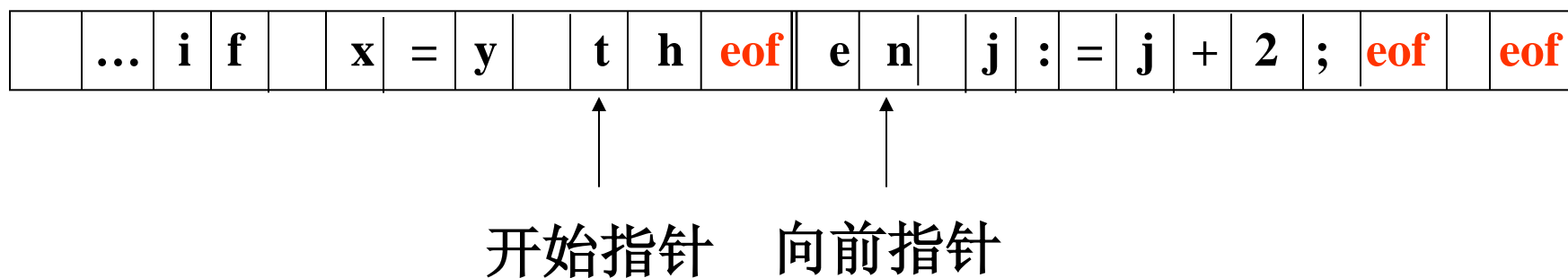
- 把一个缓冲器分为相同的两半，每半各含N个字符，一般N=1KB或4KB。



# 测试指针的过程(1)

```
IF (向前指针在左半区的终点) {  
    读入字符串, 填充右半区;  
    向前指针前移一个位置;  
}  
ELSE IF (向前指针在右半区的终点) {  
    读入字符串, 填充左半区;  
    向前指针移到缓冲区的开始位置;  
}  
ELSE 向前指针前移一个位置;
```

# 每半区带有结束标记的缓冲器



## 测试指针的过程 (2)

向前指针前移一个位置;

**IF** (向前指针指向 **eof**) {

**IF** (向前指针在左半区的终点) {

        读入字符串, 填充右半区;

        向前指针前移一个位置;

    };

**ELSE IF** (向前指针在右半区的终点) {

        读入字符串, 填充左半区;

        向前指针指向缓冲区的开始位置;

    };

**ELSE** 终止词法分析;

}

## 四、词法分析程序的输出——记号

### ■ 记号、模式和单词

- ◆ 记号：是指某一类单词符号的种别编码，如标识符的记号为**id**，数的记号为**num**等。
- ◆ 模式：是指某一类单词符号的构词规则，如标识符的模式是“由字母开头的字母数字串”。
- ◆ 单词：是指某一类单词符号的一个特例，如**position**是标识符。

### ■ 记号的属性

# 记号的属性

- 词法分析程序在识别出一个记号后，要把与之有关的信息作为它的属性保留下来。
- 记号影响语法分析的决策，属性影响记号的翻译。
- 在词法分析阶段，对记号只能确定一种属性
  - ◆ 标识符：单词在符号表中入口的指针
  - ◆ 常数：它所表示的值
  - ◆ 关键字：（一符一种、或一类一种）
  - ◆ 运算符：（一符一种、或一类一种）
  - ◆ 分界符：（一符一种、或一类一种）

# **total:=total+rate\*4** 的词法分析结果

**<id, 指向标识符total在符号表中的入口的指针>**

**<assign\_op, - >**

**<id, 指向标识符total在符号表中的入口的指针>**

**<plus\_op, - >**

**<id, 指向标识符rate在符号表中的入口的指针>**

**<mul\_op, - >**

**<num, 整数值4>**



# 3.3 记号的描述和识别

- 识别单词是按照记号的模式进行的，一种记号的模式匹配一类单词的集合。
    - ◆ 为设计词法程序，对模式要给出规范、系统的描述。
  - 正规表达式和正规文法是描述模式的重要工具。
    - ◆ 二者具有同等表达能力
    - ◆ 正规表达式：清晰、简洁
    - ◆ 正规文法：便于识别
- 
- 一、词法与正规文法
  - 二、记号的文法
  - 三、状态转换图与记号的识别

# 一、词法与正规文法

文法?

- 把源语言的文法 $G$ 分解为若干子文法:

$G_0$ 、  
 $G_1$ 、 $G_2$ 、...、 $G_n$

语法

词法

- 词法: 描述语言的标识符、常数、运算符和标点符号等记号的文法  
—— 正规文法
- 语法: 借助于记号来描述语言的结构的文章  
—— 上下文无关文法

## 二、记号的文法

- 标识符
- 常数
  - ◆ 整数
  - ◆ 无符号数
- 运算符
- 分界符
- 关键字

# 标识符

- 标识符定义为“由字母打头的、由字母或数字组成的符号串”

正则表达式?

- 描述标识符集合的正规表达式:

**letter(letter|digit)\***

- 表示标识符集合的正规定义式:

letter  $\rightarrow$  **A|B|...|Z|a|b|...|z**

digit  $\rightarrow$  **0|1|...|9**

id  $\rightarrow$  letter(letter|digit)\*

# 把正规定义式转换为相应的正规文法

$$\begin{aligned} & (\text{letter} \mid \text{digit})^* \\ &= \varepsilon \mid (\text{letter} \mid \text{digit})^+ \\ &= \varepsilon \mid (\text{letter} \mid \text{digit})(\text{letter} \mid \text{digit})^* \\ &= \varepsilon \mid \text{letter}(\text{letter} \mid \text{digit})^* \mid \text{digit}(\text{letter} \mid \text{digit})^* \\ &= \varepsilon \mid (\text{A} \mid \dots \mid \text{Z} \mid \text{a} \mid \dots \mid \text{z})(\text{letter} \mid \text{digit})^* \\ &\quad \mid (0 \mid \dots \mid 9)(\text{letter} \mid \text{digit})^* \\ &= \varepsilon \mid \text{A}(\text{letter} \mid \text{digit})^* \mid \dots \mid \text{Z}(\text{letter} \mid \text{digit})^* \\ &\quad \mid \text{a}(\text{letter} \mid \text{digit})^* \mid \dots \mid \text{z}(\text{letter} \mid \text{digit})^* \\ &\quad \mid 0(\text{letter} \mid \text{digit})^* \mid \dots \mid 9(\text{letter} \mid \text{digit})^* \end{aligned}$$

# 标识符的正规文法

$$\begin{aligned} id &\rightarrow A \textit{ rid} | \dots | Z \textit{ rid} | a \textit{ rid} | \dots | z \textit{ rid} \\ rid &\rightarrow \varepsilon | A \textit{ rid} | B \textit{ rid} | \dots | Z \textit{ rid} \\ &\quad | a \textit{ rid} | b \textit{ rid} | \dots | z \textit{ rid} \\ &\quad | 0 \textit{ rid} | 1 \textit{ rid} | \dots | 9 \textit{ rid} \end{aligned}$$

- 一般写作:

$$\begin{aligned} id &\rightarrow \textit{ letter rid} \\ rid &\rightarrow \varepsilon | \textit{ letter rid} | \textit{ digit rid} \end{aligned}$$

# 常数——整数

- 描述整数结构的正规表达式为：

$$(\text{digit})^+$$

- 对此正规表达式进行等价变换：

$$(\text{digit})^+ = \text{digit}(\text{digit})^*$$

$$(\text{digit})^* = \varepsilon \mid \text{digit}(\text{digit})^*$$

- 整数的正规文法：

$$\text{digits} \rightarrow \text{digit remainder}$$

$$\text{remainder} \rightarrow \varepsilon \mid \text{digit remainder}$$

# 常数——无符号数

- 无符号数的正规表达式为：

**$(\text{digit})^+ (.( \text{digit})^+)? (\text{E}(+|-)?(\text{digit})^+)?$**

- 正规定义式为

$\text{digit} \rightarrow \mathbf{0|1|...|9}$

$\text{digits} \rightarrow \text{digit}^+$

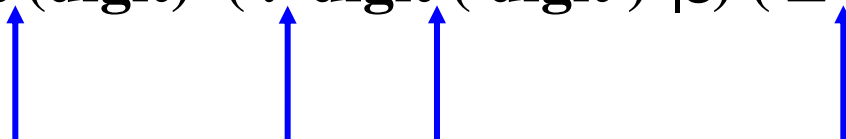
$\text{optional\_fraction} \rightarrow (.\text{digits})?$

$\text{optional\_exponent} \rightarrow (\mathbf{\text{E}(+|-)?}\text{digits})?$

$\text{num} \rightarrow \text{digits optional\_fraction optional\_exponent}$



# 把正规定义式转换为正规文法

$$\begin{aligned} & (\text{digit})^+ (.( \text{digit})^+ )? (E(+|-)?( \text{digit})^+ )? \\ & =(\text{digit})^+ (.( \text{digit})^+ | \varepsilon) (E(+|-|\varepsilon)( \text{digit})^+ | \varepsilon) \\ & =\text{digit}(\text{digit})^* (.\text{digit}(\text{digit})^* | \varepsilon) (E(+|-|\varepsilon)\text{digit}(\text{digit})^* | \varepsilon) \end{aligned}$$


num1 表示无符号数的第一个数字之后的部分

num2 表示小数点以后的部分

num3 表示小数点后第一个数字以后的部分

num4 表示E之后的部分

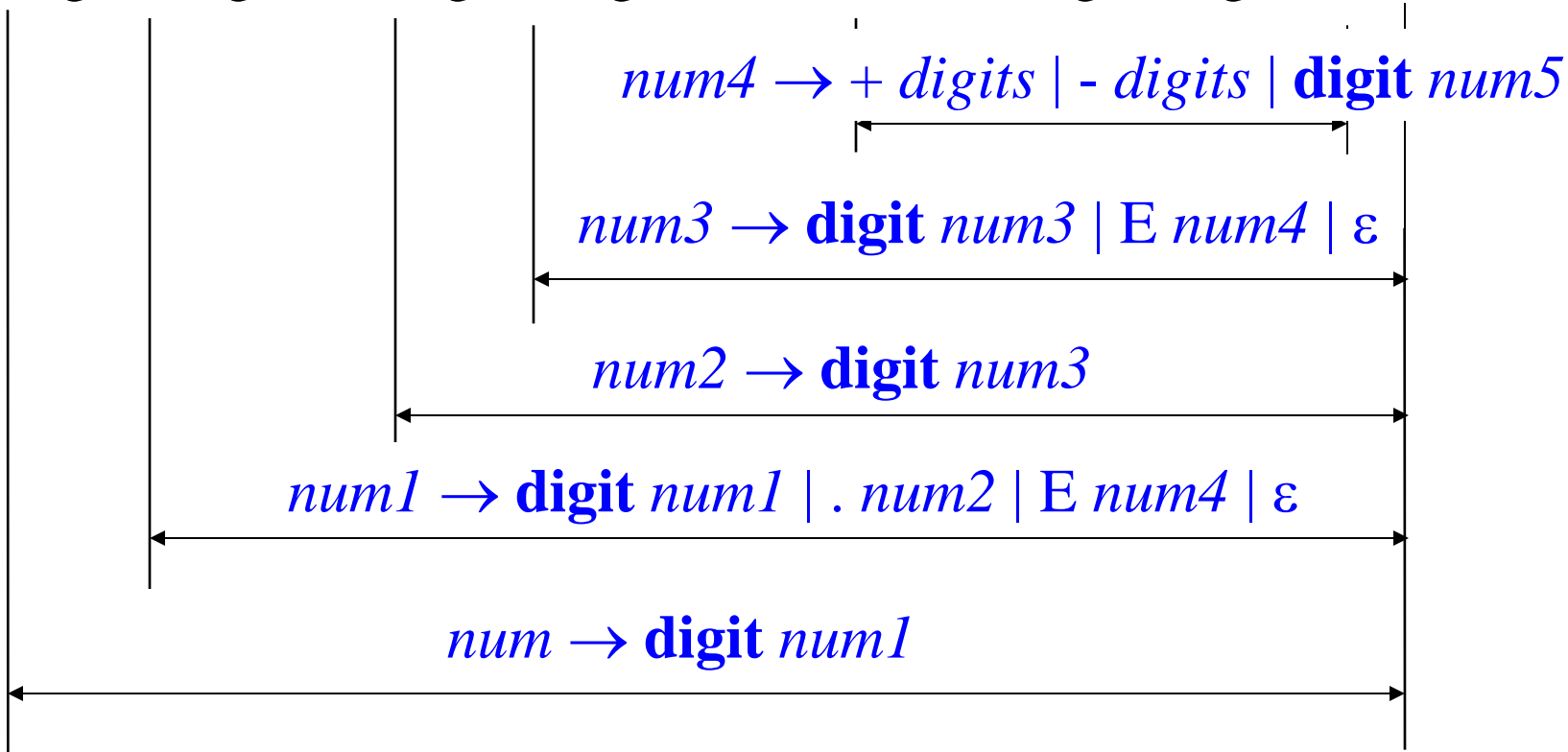
num5 表示 $(\text{digit})^*$

digits 表示 $(\text{digit})^+$

# 无符号数分析图

$digits \rightarrow \mathbf{digit} \ num5$   
 $num5 \rightarrow \mathbf{digit} \ num5 \mid \varepsilon$

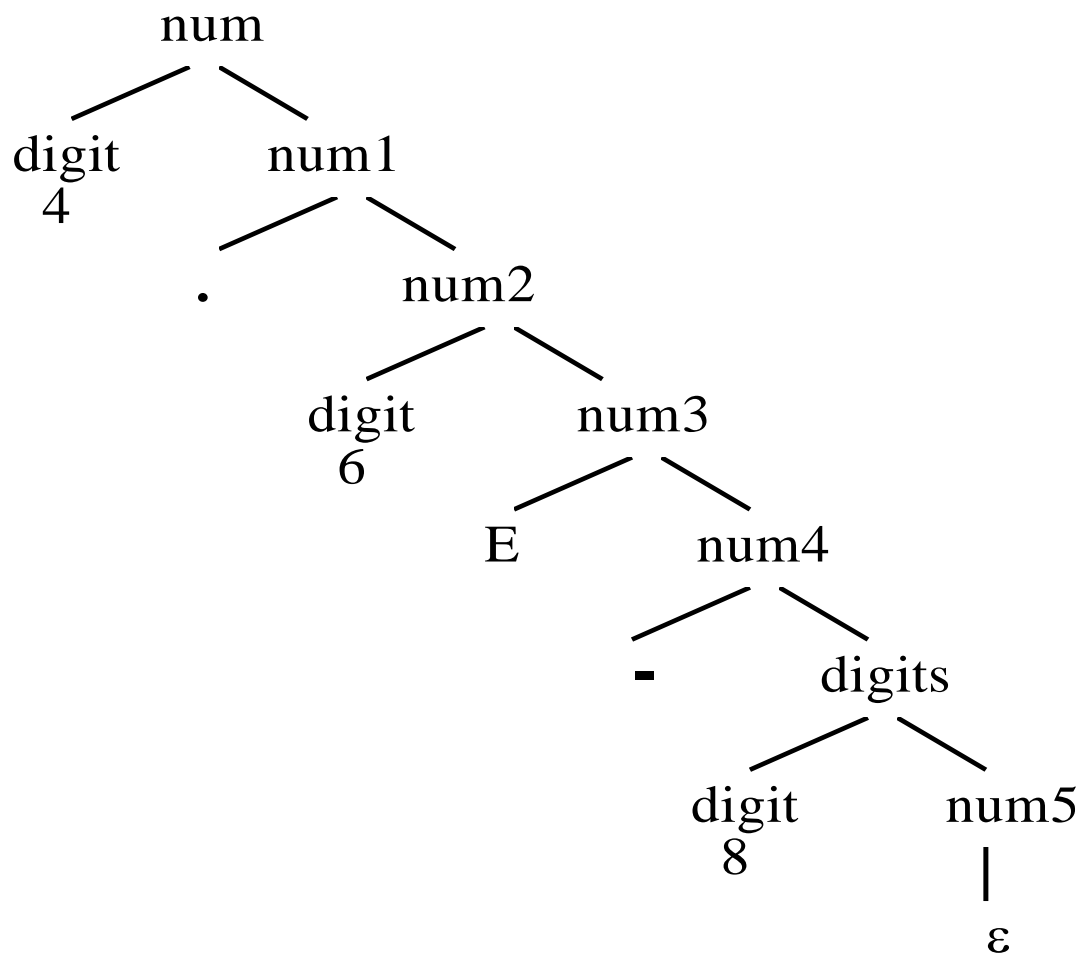
$digit \ (digit)^* \ (. \ digit \ (digit)^* | \varepsilon) \ (E \ (+|-| \varepsilon) \ digit \ (digit)^* | \varepsilon)$



# 无符号数的正规文法

**$num \rightarrow digit\ num1$**   
 **$num1 \rightarrow digit\ num1 \mid .\ num2 \mid E\ num4 \mid \varepsilon$**   
 **$num2 \rightarrow digit\ num3$**   
 **$num3 \rightarrow digit\ num3 \mid E\ num4 \mid \varepsilon$**   
 **$num4 \rightarrow +\ digits \mid -\ digits \mid digit\ num5$**   
 **$digits \rightarrow digit\ num5$**   
 **$num5 \rightarrow digit\ num5 \mid \varepsilon$**

# 无符号数 4.6E-8 的分析树



# 运算符

- 关系运算符的正规表达式为：

$\lt \mid \leq \mid = \mid \lt \gt \mid \gt = \mid \gt$

- 正规定义式：

$\text{relop} \rightarrow \lt \mid \leq \mid = \mid \lt \gt \mid \gt = \mid \gt$

- 关系运算符的正规文法：

$\text{relop} \rightarrow \lt \mid \lt \text{equal} \mid = \mid \lt \text{greater} \mid \gt \mid \gt \text{equal}$

$\text{greater} \rightarrow \gt$

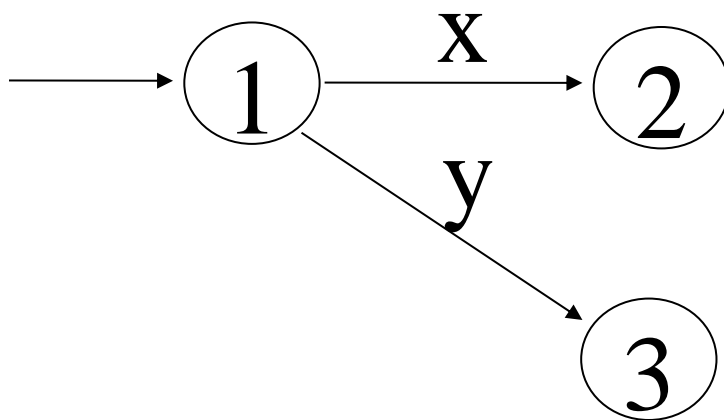
$\text{equal} \rightarrow =$

# 三、状态转换图与记号的识别

- 状态转换图
- 利用状态转换图识别记号
- 为线性文法构造相应的状态转换图
  - ◆ 状态集合的构成
  - ◆ 状态之间边的形成

# 状态转换图

- 状态转换图是一张有限的方向图
  - ◆ 图中结点代表状态，用圆圈表示。
  - ◆ 状态之间用有向边连接。
  - ◆ 边上的标记代表在射出结状态下，可能出现的输入符号或字符类。



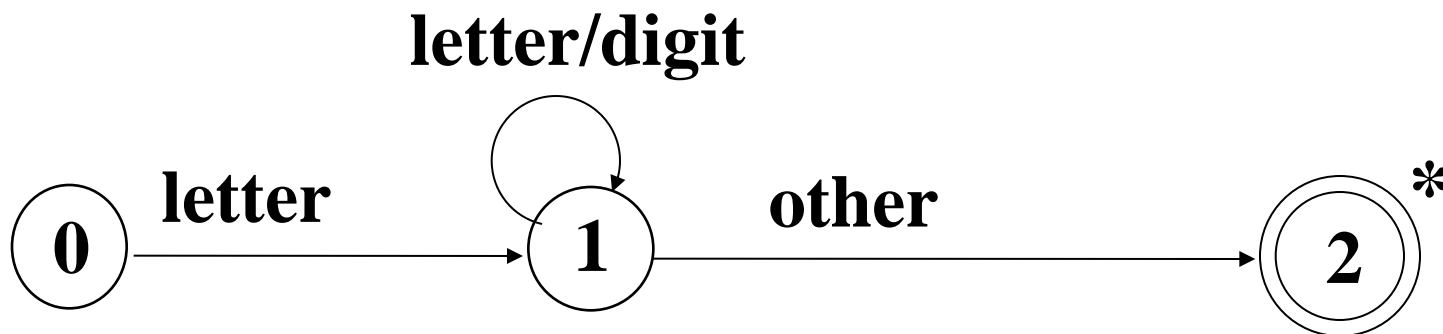
# 标识符的状态转换图

- 标识符的文法产生式:

***id*  $\rightarrow$  letter *rid***

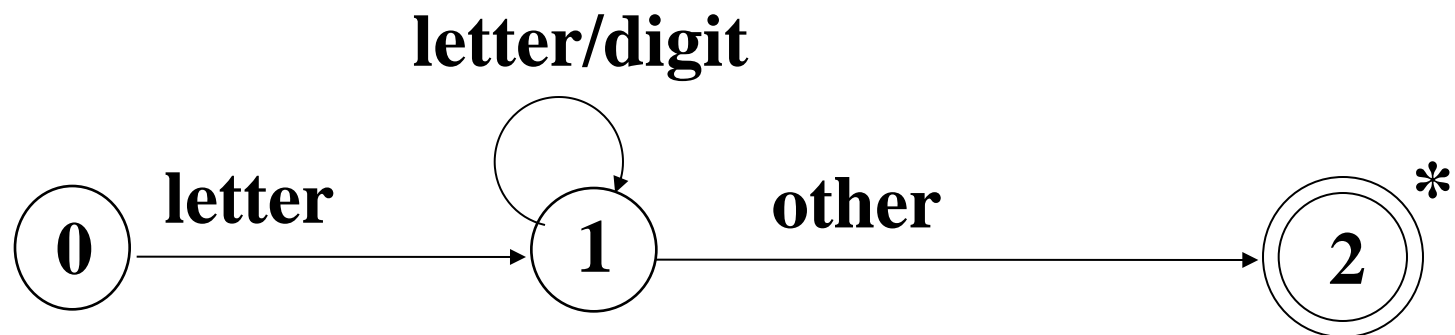
***rid*  $\rightarrow \varepsilon \mid$  letter *rid*  $\mid$  digit *rid***

- 标识符的状态转换图





# 利用状态转换图识别记号



- 语句 `D099K=1.10` 中标识符 `D099K` 的识别过程

# 为线性文法构造相应的状态转换图

## ■ 状态集合的构成

- ◆ 对文法 $G$ 的每一个非终结符号设置一个对应的状态
- ◆ 文法的开始符号对应的状态称为初态
- ◆ 增加一个新的状态，称为终态。

## ■ 状态之间边的形成

- ◆ 对产生式 $A \rightarrow aB$ ，从 $A$ 状态到 $B$ 状态画一条标记为 $a$ 的边
- ◆ 对产生式 $A \rightarrow a$ ，从 $A$ 状态到终态画一条标记为 $a$ 的边
- ◆ 对产生式 $A \rightarrow \varepsilon$ ，从 $A$ 状态到终态画一条标记为 $\varepsilon$ 的边

# 无符号数的右线性文法的状态转换图

$num \rightarrow digit\ num1$

$num1 \rightarrow digit\ num1 \mid \cdot\ num2 \mid E\ num4 \mid \varepsilon$

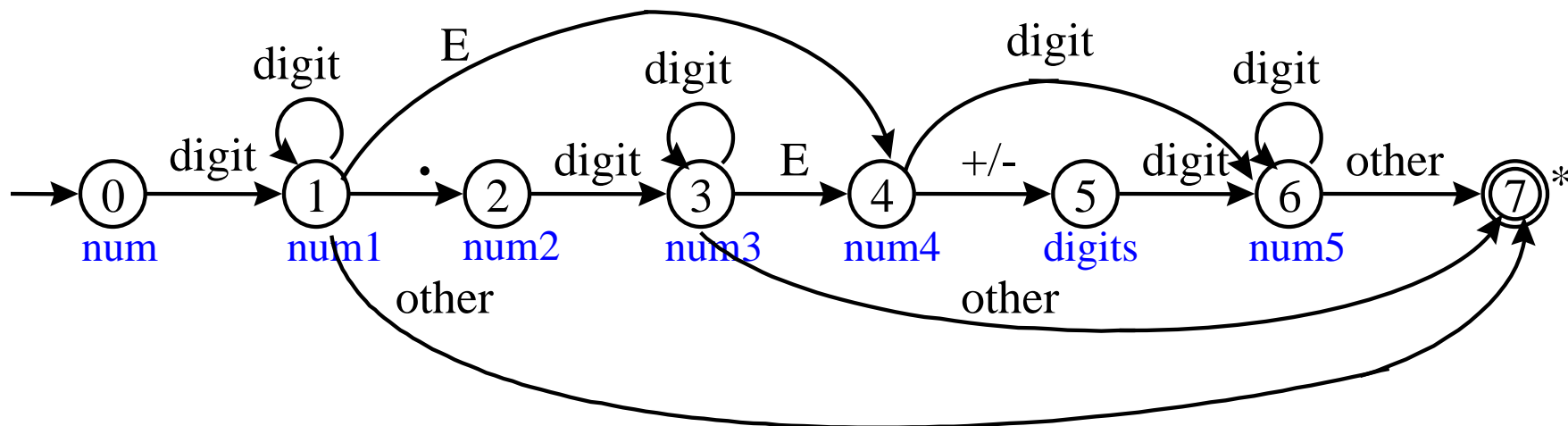
$num2 \rightarrow digit\ num3$

$num3 \rightarrow digit\ num3 \mid E\ num4 \mid \varepsilon$

$num4 \rightarrow +\ digits \mid -\ digits \mid digit\ num5$

$digits \rightarrow digit\ num5$

$num5 \rightarrow digit\ num5 \mid \varepsilon$



# 3.4 词法分析程序的设计与实现

## 一、文法及状态转换图

1. 语言说明
2. 记号的正规文法
3. 状态转换图

## 二、词法分析程序的构造

## 三、词法分析程序的实现

1. 输出形式
2. 设计全局变量和过程
3. 编制词法分析程序

# 一、文法及状态转换图

## ■ 语言说明

**标识符**：以字母开头的、后跟字母或数字组成的符号串。

**保留字**：标识符的子集。

**无符号数**：同**PASCAL**语言中的无符号数。

**关系运算符**：<、<=、=、<>、>=、>。

**标点符号**：+、-、\*、/、(、)、:、'、；等。

**赋值号**：:=

**注释标记**：以‘/\*’开始，以‘\*/’结束。

**单词符号间的分隔符**：空格

# 记号的正规文法

- 标识符的文法

**$id \rightarrow \text{letter } rid$**

**$rid \rightarrow \varepsilon \mid \text{letter } rid \mid \text{digit } rid$**

- 无符号整数的文法

**$digits \rightarrow \text{digit } remainder$**

**$remainder \rightarrow \varepsilon \mid \text{digit } remainder$**

# 记号的正规文法 (续)

## ■ 无符号数的文法

**$num \rightarrow digit\ num1$**

**$num1 \rightarrow digit\ num1 \mid .\ num2 \mid E\ num4 \mid \varepsilon$**

**$num2 \rightarrow digit\ num3$**

**$num3 \rightarrow digit\ num3 \mid E\ num4 \mid \varepsilon$**

**$num4 \rightarrow +\ digits \mid -\ digits \mid digit\ num5$**

**$digits \rightarrow digit\ num5$**

**$num5 \rightarrow digit\ num5 \mid \varepsilon$**

## ■ 关系运算符的文法

**$relop \rightarrow < \mid <equal \mid = \mid <greater \mid > \mid >equal$**

**$greater \rightarrow >$**

**$equal \rightarrow =$**

# 记号的正规文法(续)

- 赋值号的文法

*assign\_op* → *:equal*

*equal* → =

- 标点符号的文法

*single* → + | - | \* | / | ( | ) | : | ' | ;

- 注释头符号的文法

*note* → / *star*

*star* → \*

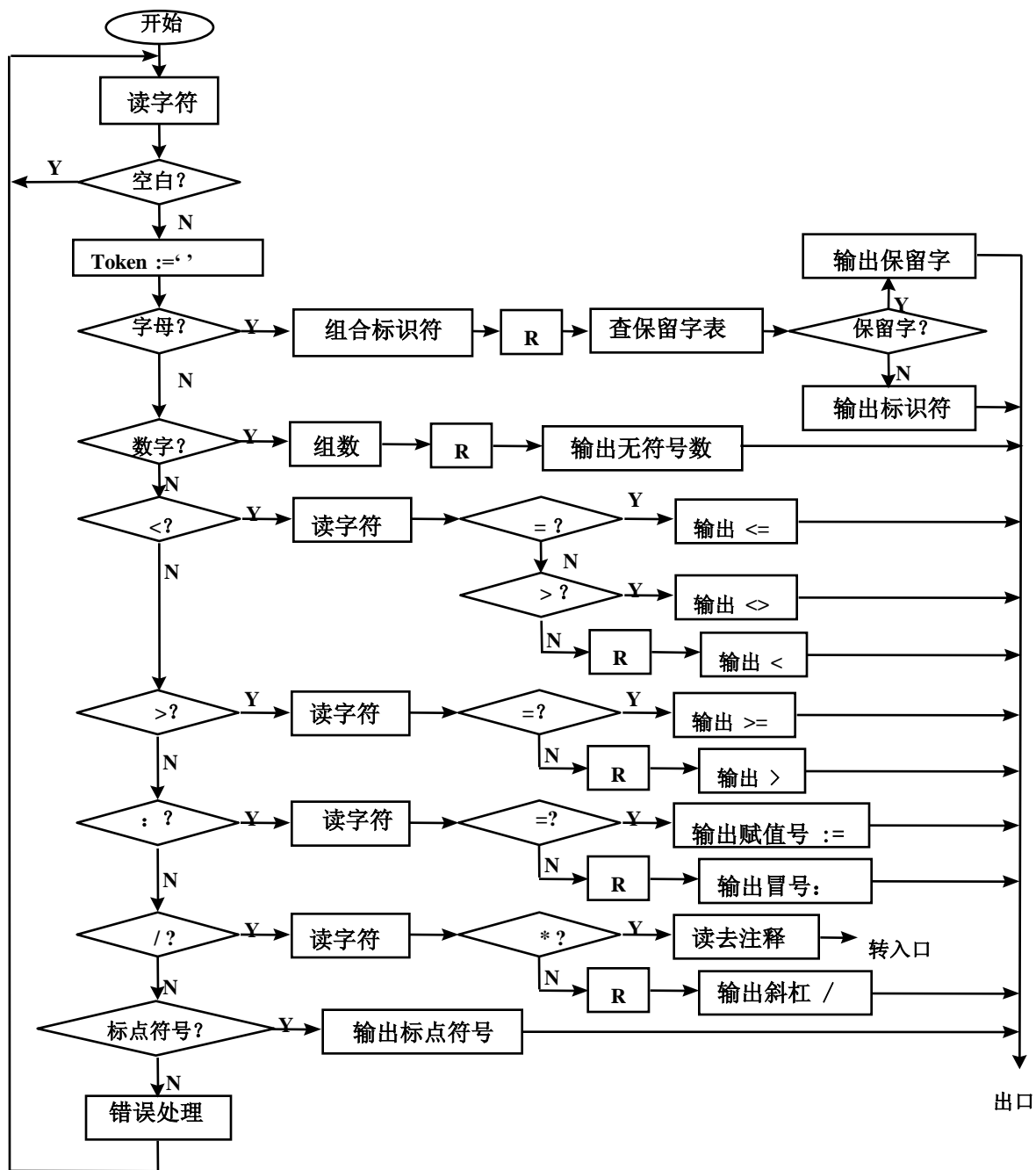


Wensheng Li BUPT



## 二、词法分析程序的构造

根据上述状态转换图，再加上相应的语义动作，就可以构造出的词法分析程序的算法框图。



说明：R 为一过程，其功能是向前指针回退一个字符；Token 为字符数组，用于存放单词符号

# 三、词法分析程序的实现

- 输出形式
- 设计全局变量和过程
- 编制词法分析程序

# 输出形式

- 利用翻译表，将识别出的单词的记号以二元式的形式加以输出
- 二元式的形式：  
    〈记号，属性〉

正规表达式	记号	属性
if	if	—
then	then	—
else	else	—
id	id	符号表入口指针
num	num	常数表入口指针 / val
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE
:=	assign-op	—
+	+	—
—	—	—
*	*	—
/	/	—
(	(	—
)	)	—
,	,	—
;	;	—
:	:	—

# 设计全局变量和过程

- 转换图中的每一状态，分别用一段程序实现。
- 如果某一状态有若干条射出边，则程序：
  - ◆ 读一个字符
  - ◆ 根据读到的字符，选择标记与之匹配的边到达下一个状态，即程序控制转去执行下一个状态对应的语句序列。

## 设计全局变量和过程（续）

- (1) **C**: 字符变量，存放当前读进的字符。
- (2) **iskey**: 整型变量
- (3) **token**: 字符数组，存放单词的字符串。
- (4) **lexemebegin**: 字符指针，指向输入缓冲区中当前单词的开始位置。
- (5) **forward**: 字符指针，向前扫描指针。
- (6) **buffer**: 字符数组，输入缓冲区。
- (7) **get\_char**: 读字符过程，每调用一次，读进一个字符，并把它放入char中，且向前指针forward指向下一个字符。
- (8) **get\_nbc**: 过程，每次调用时，检查char中是否为空字符，若是，则反复调用该过程，直到char进入一个非空字符为止。



## 设计全局变量和过程（续）

- (9) **cat**: 过程，每次调用把当前char中的字符与token中的字符串连接起来。
- (10) **letter**和**digit**: 布尔函数，分别判断char中的字符是否为字母或数字，若是则返回true，否则返回false。
- (11) **retract**: 过程，向前扫描指针forward后退一个字符。
- (12) **reserve**: 函数，查保留字表，若此函数的返回值为0，则表示token中的字符串是标识符，否则为保留字。
- (13) **DTB**: 十/二进制的转换过程，它将token中的数字串转换成二进制的数值表示。
- (14) **table\_insert**: 过程，将标识符插入符号表。
- (15) **error**: 错误处理函数。
- (16) **return**: 返回过程，收集并携带必要的信息返回调用程序。

# 编制词法分析程序（类C语言描述）

```
token="";
get_char;
get_nbc;
SWITCH (C)
{
CASE 'a'..'z': WHILE (letter || digit) {
                cat; get_char; }
                retract;
                iskey=reserve;
                IF iskey==0 {
                    table_insert;
                    return(ID, 指向该标识符在符号表的入口指针);
                };
ELSE return (关键字的记号, -);
BREAK;
```

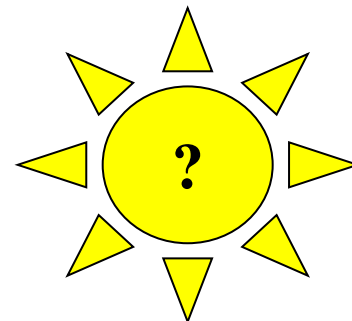
# 编制词法分析器（续）

```
CASE '0'..'9': ...      //识别出无符号数，并存入token  
    return(num, DTB(token));  
    BREAK;  
CASE '<': get_char;  
    IF (C=='=') return(relop, LE);  
    ELSE IF (C=='>') return(relop, NE);  
        ELSE { retract; return(relop, LT);}  
    BREAK;  
CASE '=' : return(relop, EQ); BREAK;  
CASE '>': get_char;  
    IF (C=='=') return(relop, GE);  
    ELSE { retract; return(relop, GT);}  
    BREAK;
```

# 编制词法分析器（续）

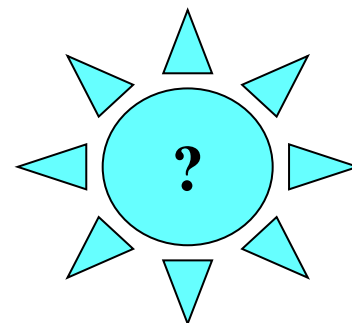
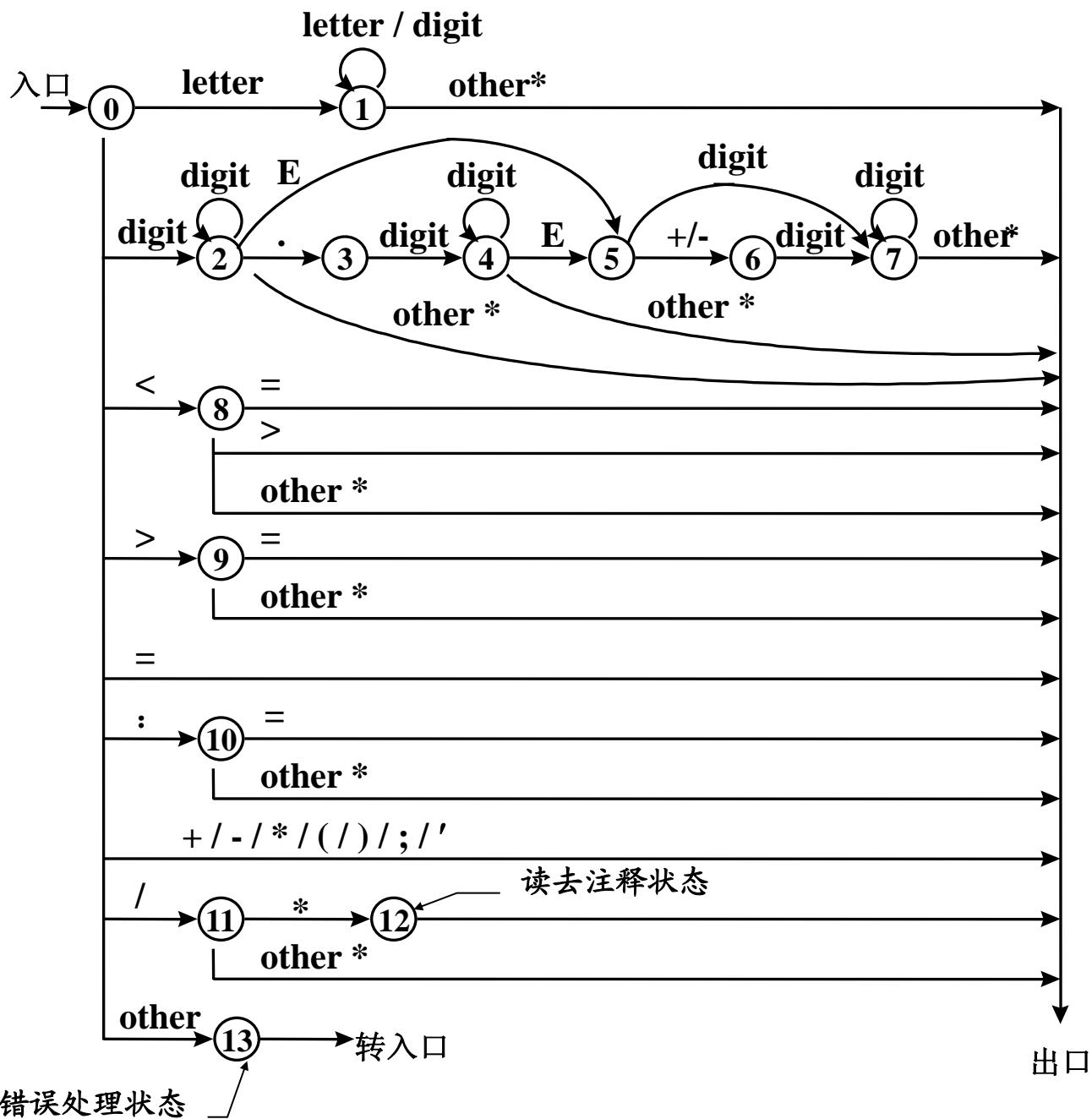
```
CASE ':' : get_char;  
    IF (C=='=') return(assign-op, -);  
    retract; return(':', -);  
    BREAK;  
CASE '+' : return('+', -); BREAK;  
CASE '-' : return('-', -); BREAK;  
CASE '(' : return('(', -); BREAK;  
CASE ')' : return(')', -); BREAK;  
CASE ';' : return(';', -); BREAK;
```

# 编制词法分析器（续）



```
CASE '/' : get_char;  
    IF (C=='*') {  
        loop_1:  get_char;  
            WHILE (!'*') get_char;  
            get_char;  
            IF (C=='/') BREAK;  
            GOTO loop_1;  
    };  
    ELSE { retract; return('/',-);};  
    BREAK;  // end of case '/'  
DEFAULT: error();  
}  //end of switch
```

# 状态转换图

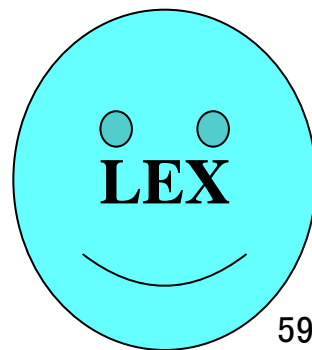


# 小 结

- 词法分析器的作用
- 与语法分析器的关系
  - ◆ 独立、子程序、协同程序
- 配对缓冲区
  - ◆ 必要性、算法
- 记号
  - ◆ 记号、模式、单词
  - ◆ 属性
  - ◆ 二元式形式            <记号, 属性>
  - ◆ 描述: 正规表达式、正规文法
  - ◆ 识别: 状态转换图

# 小结 (续)

- 词法分析器的设计与实现
  - ◆ 各类单词符号的正规表达式
  - ◆ 各类单词符号的正规文法
  - ◆ 构造与文法相应的状态转换图
  - ◆ 合并为词法分析器的状态转换图
  - ◆ 增加语义动作，构造词法分析器的程序框图
  - ◆ 确定输出形式、设计翻译表
  - ◆ 定义变量和过程
  - ◆ 编码实现





# 作业

- 3.2 (1) (4)

- 3.4

- 3.8

- 3.9

- 程序设计1

- 方法1 (必做)

- 方法2 (选作)

## 第3章 词法分析——LEX



*LI Wensheng, SCST, BUPT*

使用LEX的流程

LEX 源文件格式

LEX 工作原理

# 一、使用LEX的流程

LEX源程序  
lex.l → LEX编译器 → Lex.yy.c

lex.yy.c → C编译器 → Lex.yy.o 或 a.out

字符流源程序 → a.out → 记号序列

## 二、LEX源程序

一个LEX源程序由三部分组成：

1. 说明部分
2. 翻译规则
3. 辅助过程

**说明部分**

**%%**

**翻译规则**

**%%**

**辅助过程**

# 1. 说明部分

- 包括：
  - 变量说明
  - 标识符常量说明
  - 正规定义
- 正规定义中的名字可在翻译规则中用作正规表达式的成分
- C语言的说明必须用分界符 “ % { ” 和 “ % } ” 括起来。

## 2. 翻译规则

- 形式:

$P_1 \quad \{ \text{动作}_1 \}$

$P_2 \quad \{ \text{动作}_2 \}$

...

$P_n \quad \{ \text{动作}_n \}$

- $P_i$  是一个正规表达式，描述一种记号的模式。

- 动作 $i$  是C语言的程序语句，表示当一个串匹配模式 $P_i$ 时，词法分析器应执行的动作。

### 3. 辅助过程

- 对翻译规则的补充
- 翻译规则部分中某些动作需要调用的过程，如果不是C语言的库函数，则要在此给出具体的定义。
- 这些过程也可以存入另外的程序文件中，单独编译，然后和词法分析器连接装配在一起。

# LEX源程序举例

- 正规定义式:

**if → if**

**then → then**

**else → else**

**relop → < | <= | = | <> | > | >=**

**id → letter(letter|digit)\***

**num → digit<sup>+</sup>(.digit<sup>+</sup>)?(E(+|-)?digit<sup>+</sup>)?**



# 相应的 LEX 源程序 框架

```
/* 说明部分 */
%{
#include <stdio.h>
/* C语言描述的标识符常量的定义，如 LT、LE、EQ、NE、GT、
   GE、IF、THEN、ELSE、ID、NUMBER、RELOP */
extern yylval;
}%
/* 正规定义式 */
delim    [ \t\n]
ws       {delim}+
letter   [A-Za-z]
digit    [0-9]
id       {letter}({letter}|{digit})*
num      {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?
%%
```

# 相应的 LEX 源程序 框架

```
/* 规则部分 */
{ws}      { /* 没有动作, 也不返回 */ }
if        { return(IF); }
then      { return(THEN); }
else      { return(ELSE); }
{id}      { yylval=install_id(); return(ID); }
{num}     { yylval=install_num();
           return(NUMBER); }

"<"      { yylval=LT; return(RELOP); }
"<="    { yylval=LE; return(RELOP); }
"="      { yylval=EQ; return(RELOP); }
"<>"    { yylval=NE; return(RELOP); }
">"      { yylval=GT; return(RELOP); }
">="    { yylval=GE; return(RELOP); }

%%
```

如果没有return语句, 则, 处理完整个输入之后才会返回!!

# 相应的 LEX 源程序 框架

```
/* 辅助过程 */
```

```
int install_id() {
```

```
    /* 把单词插入符号表并返回该单词在符号表中的位置
```

```
    yytext指向该单词的第一个字符
```

```
    yyleng给出它的长度 */
```

```
}
```

```
int install_num() {
```

```
    /* 类似于上面的过程，但单词是常数 */
```

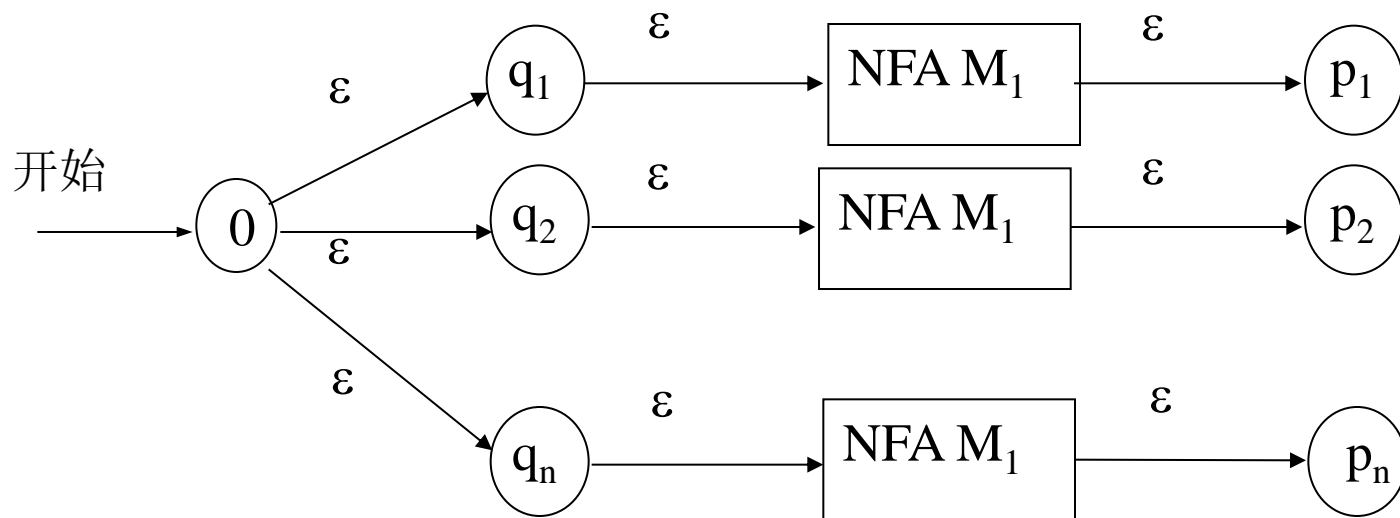
```
}
```

## 二、LEX的工作原理

1. LEX的工作过程
2. 处理二义性问题的两条规则
3. LEX工作过程举例
4. 控制执行程序

# 1. 工作过程

- 扫描每一条翻译规则 $P_i$ ，为之构造一个非确定的有限自动机NFA  $M_i$
- 将各条翻译规则对应的NFA  $M_i$ 合并为一个新的NFA  $M$



- 将NFA  $M$ 确定化为DFA  $D$ ，并生成该DFA  $D$ 的状态转换矩阵和控制执行程序。

## 2. 二义性处理

### ■ 最长匹配原则

- 在识别单词符号过程中，当有几个规则看来都适用时，则实施最长匹配的那个规则

### ■ 优先匹配原则

- 如有几条规则可以同时匹配一字符串，并且匹配的长度相同，则实施最上面的规则。

### 3. 工作过程举例

%%

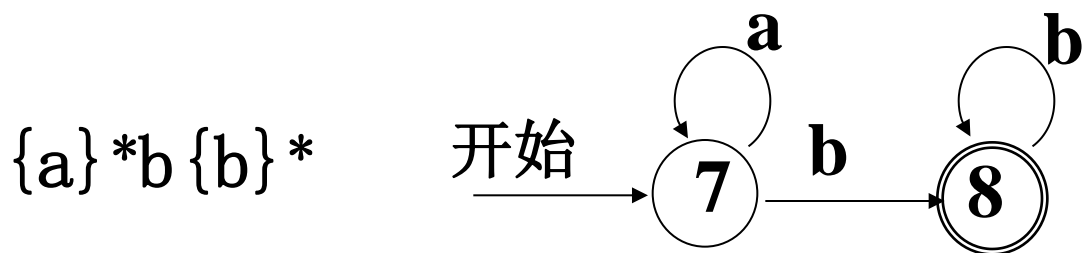
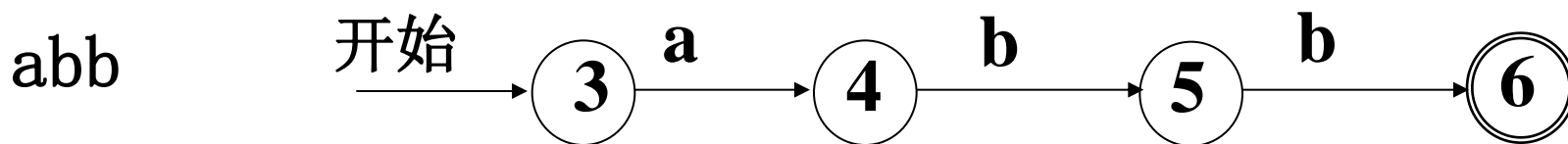
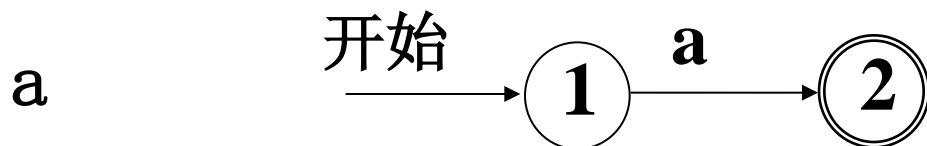
a { }

abb { }

{a}\*b{b}\* { }

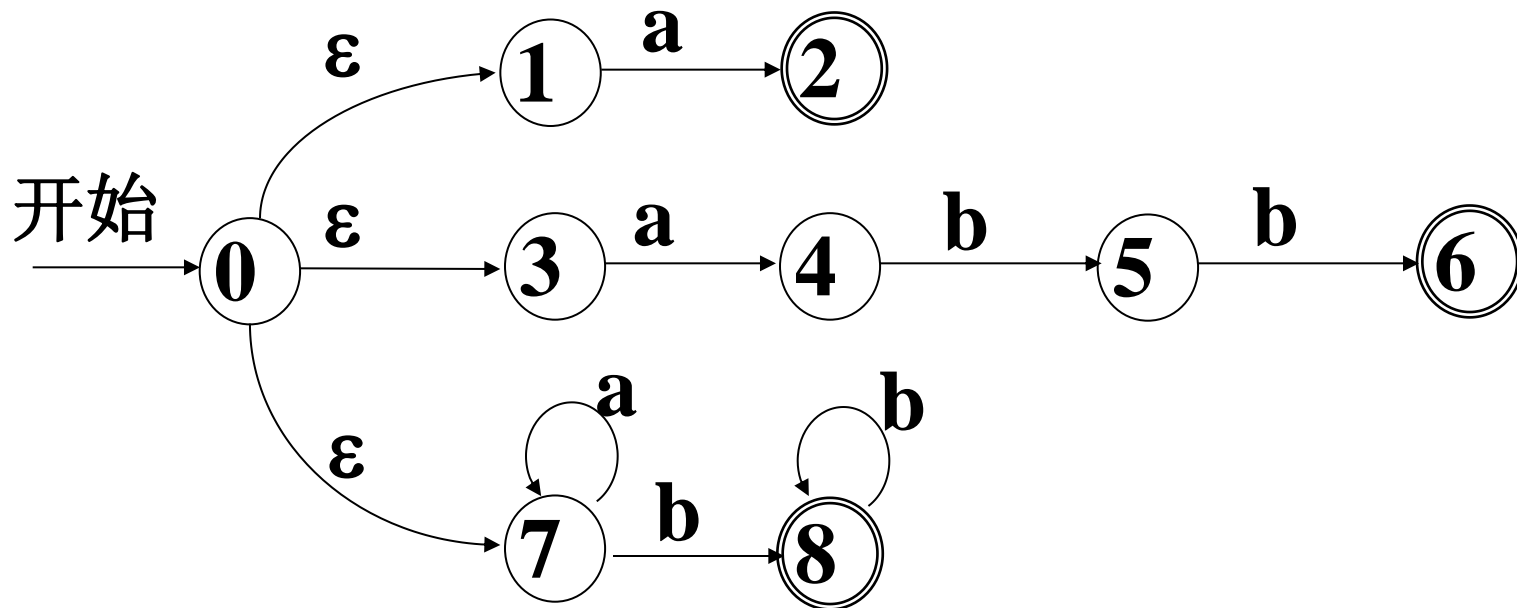
%%

# 读LEX源程序， 分别生成非确定的有限自动机





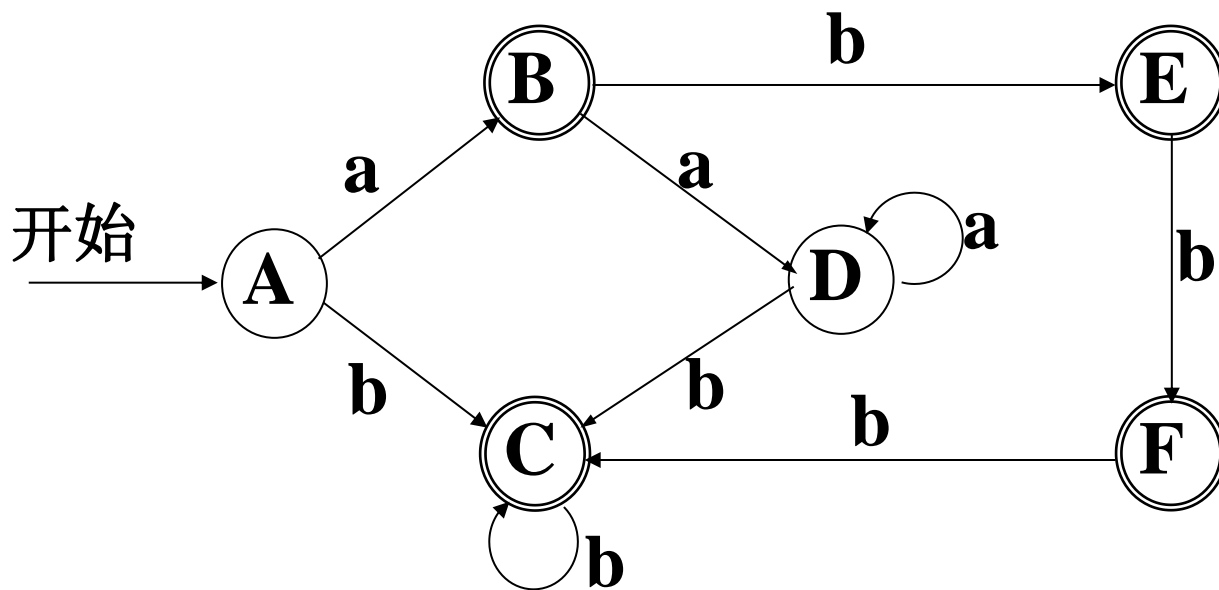
# 合并为一个NFA M



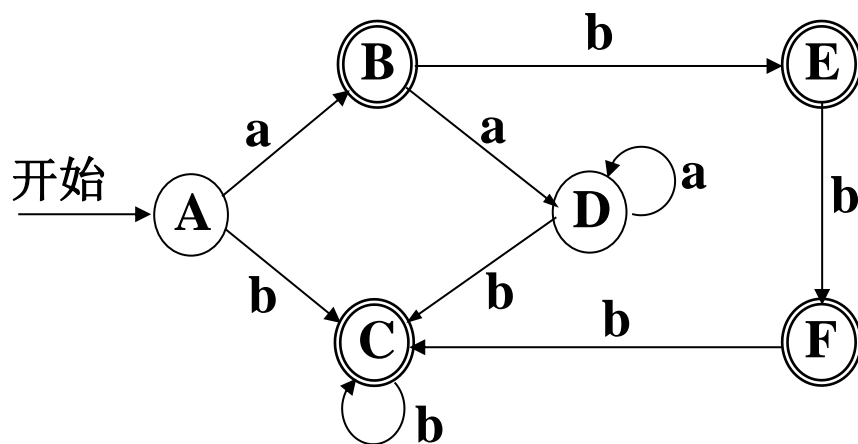
## 将该NFA M确定化为DFA D

DFA  $D = (\{a, b\}, \{A, B, C, D, E, F\}, A, \{B, C, E, F\}, \varphi)$

其中:  $A = \{0, 1, 3, 7\}$        $B = \{2, 4, 7\}$        $C = \{8\}$   
 $D = \{7\}$                        $E = \{5, 8\}$        $F = \{6, 8\}$



## 4. 控制执行程序



$A = \{0, 1, 3, 7\}$

$B = \{2, 4, 7\}$

$C = \{8\}$

$D = \{7\}$

$E = \{5, 8\}$

$F = \{6, 8\}$

输入字符串为 aba...

读入

状态

—

A

a

B

b

E

a

—

# 第4章 语法分析



*LI Wensheng, SCST, BUPT*

知识点：预测分析方法、**LL(1)**分析程序  
移进-归约分析方法、**LR**分析程序  
**SLR(1)**、**LR(1)**、**LALR(1)**分析表

# § 4 语法分析

## 简介

**4.1 语法分析程序**

**4.2 自顶向下分析方法**

**4.3 自底向上分析方法**

**4.4 LR分析方法**

**4.5 软件工具YACC**

## 小结

# 语法分析简介

- 语法分析是编译过程的核心部分
- 语法分析任务由语法分析程序完成
- 语法分析的工作依据是：语言的语法规则
- 本章内容安排：

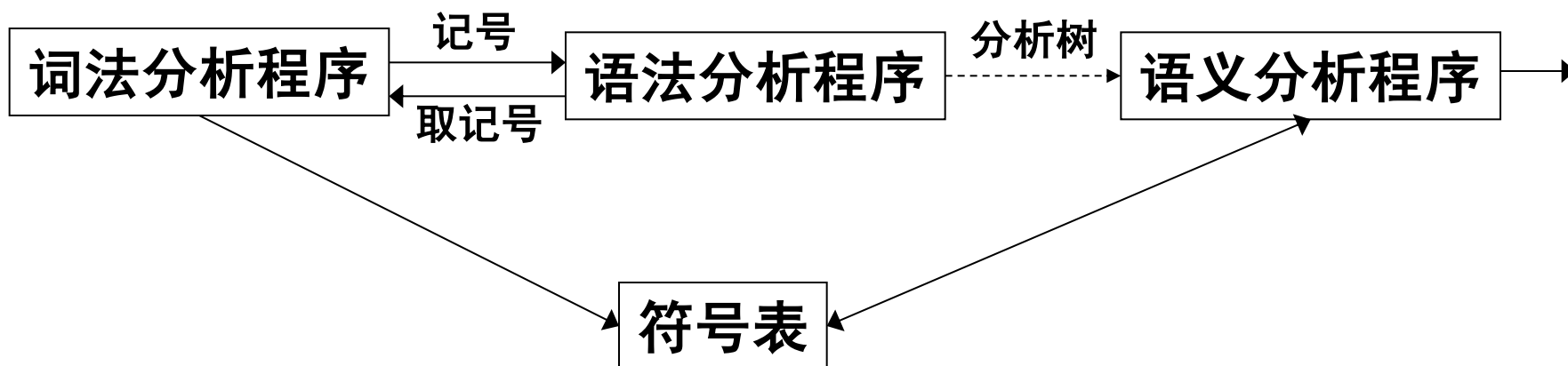
首先讨论常用的语法分析方法和技术

自顶向下的方法

自底向上的方法

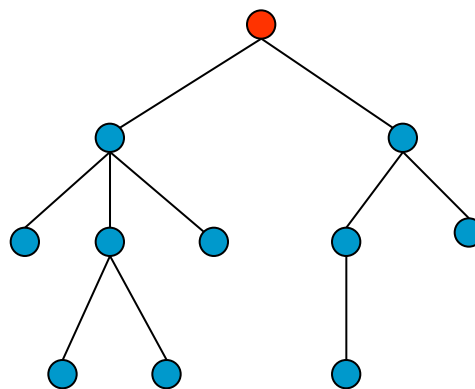
# 4.1 语法分析程序

- 语法分析程序的任务
  - ◆ 从源程序记号序列中识别出各类语法成分
  - ◆ 进行语法检查
- 语法分析程序的地位:



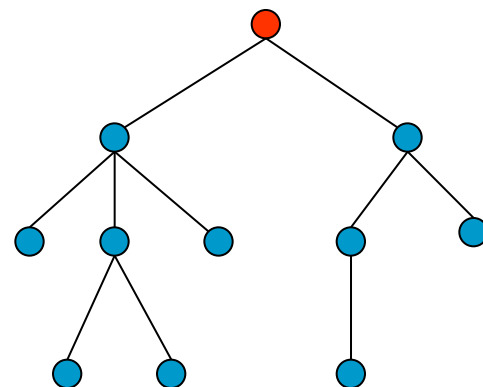
## ■ 语法分析程序的作用

- 输入：记号流/记号序列
- 工作依据：语法规则
- 功能：将记号组合成语法成分、语法检查
- 输出：分析树
- 错误处理



## ■ 常用的分析方法

- ◆ 自顶向下的方法：  
从树根到叶子来建立分析树
- ◆ 自底向上的方法：  
从树叶到树根来建立分析树



## ■ 对输入符号串的扫描顺序：自左向右



## 4.2 自顶向下分析方法

- 一、递归下降分析
- 二、递归调用预测分析
- 三、非递归预测分析

# 一、递归下降分析

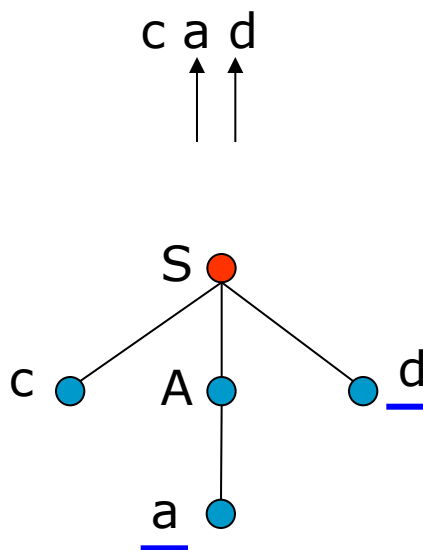
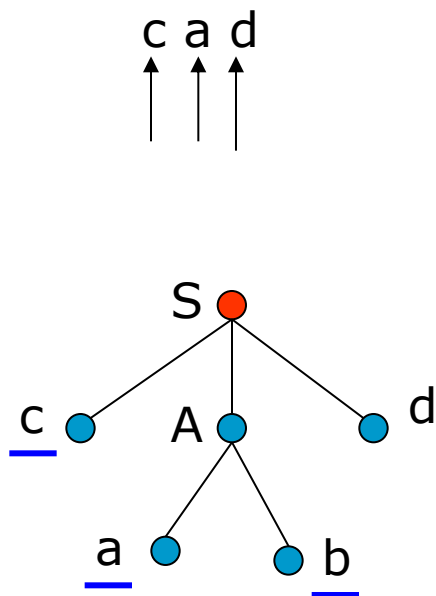
- 从文法的开始符号出发，进行推导，试图推出要分析的输入串的过程。
- 对给定的输入串，从对应于文法开始符号的根结点出发，自顶向下地为输入串建立一棵分析树。
- 试探过程，是反复使用不同产生式谋求匹配输入串的过程。

例：试分析输入串 $\omega = \text{cad}$ 是否为如下文法的一个句子。

$S \rightarrow \text{cAd}$

$A \rightarrow \text{ab} \mid \text{a}$

(文法4.1)



$S \Rightarrow \text{cAd} \Rightarrow \text{cad}$

推导?

试图为输入符号串建立一个最左推导序列的过程。

## ■ 为什么采用最左推导？

- ◆ 因为对输入串的扫描是自左至右进行的，只有使用最左推导，才能保证按扫描的顺序匹配输入串。

## ■ 递归下降分析方法的实现

- ◆ 文法的**每一个非终结符号对应一个递归过程**，即可实现这种带回溯的递归下降分析方法。
- ◆ 每个过程作为一个布尔过程，一旦发现它的某个产生式与输入串匹配，则用该产生式展开分析树，并返回**true**，否则分析树不变，返回**false**。

## ■ 实践中存在的困难和缺点

- ◆ 左递归的文法，可能导致分析过程陷入死循环。
- ◆ 回溯
- ◆ 工作的重复
- ◆ 效率低、代价高：穷尽一切可能的试探法。

## 二、递归调用预测分析

- 一种确定的、不带回溯的递归下降分析方法
- 如何克服回溯？
- 对文法的要求
- 预测分析程序的构造

# 如何克服回溯？

- 能够根据所面临的输入符号准确地指派一个候选式去执行任务。
- 该选择的工作结果是确信无疑的。
- 例：

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

当前输入符号： **a**

指派  $\alpha_i$  去匹配输入符号串

# 对文法的要求

左递归?

1. 不含左递归

$$A \Rightarrow \dots \Rightarrow A\alpha$$

2.  $\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \phi \quad (i \neq j)$

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

$$\text{FIRST}(\alpha_i) = \{ a \mid \alpha_i \xRightarrow{*} a\beta, a \in V_T, \alpha_i, \beta \in (V_T \cup V_N)^* \}$$

如果  $\alpha_i \xRightarrow{*} \epsilon$ , 则规定  $\epsilon \in \text{FIRST}(\alpha_i)$ 。

非终结符号A的所有候选式的开头终结符号集两两互不相交

例：有如下产生**PASCAL**类型子集的文法：

***type*** → ***simple*** | ↑**id** | **array**[***simple***] **of** ***type***

***simple*** → **integer** | **char** | **num** **dotdot** **num** （文法4.2）

分析输入串：**array** [**num** **dotdot** **num**] **of** **char**

- ***type***的三个候选式，有：

**FIRST**(***simple***) = { **integer**, **char**, **num** }

**FIRST**(↑**id**) = { ↑ }

**FIRST**(**array**[***simple***] **of** ***type***) = { **array** }

- ***simple***的三个候选式，有：

**FIRST** (**integer**) = { **integer** }

**FIRST**(**char**) = { **char** }

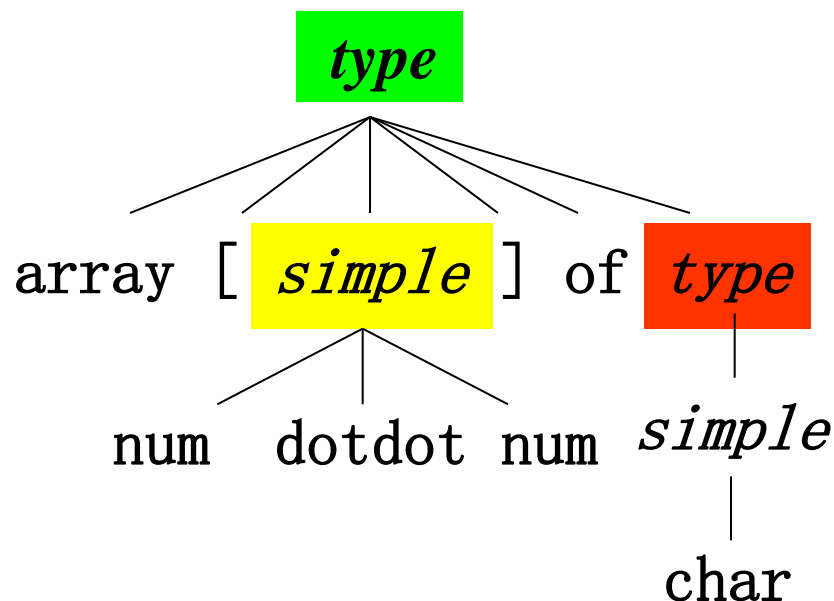
**FIRST**(**num** **dotdot** **num**) = { **num** }



输入: array [num dotdot num] of char

↑                    ↑                    ↑

树:



产生式:  $A \rightarrow \varepsilon$

缺席匹配

当递归下降分析程序没有适当候选式时, 可以用一个  $\varepsilon$ -候选式, 表示其它候选式可以缺席。

# 预测分析程序的构造

- 预测分析程序的转换图
- 转换图的工作过程
- 转换图的化简
- 预测分析程序的实现

# 预测分析程序的转换图

- 为预测分析程序建立转换图作为其实现蓝图
- 每一个非终结符号有一张图
- 有向边的标记可以是终结符号，也可以是非终结符号。
- 在一个非终结符号A上的转移意味着对相应A的过程的调用。
- 在一个终结符号a上的转移，意味着下一个输入符号若为a，则应做此转移。

# 从文法构造转换图

## ■ 改写文法

- ◆ 重写文法

- ◆ 消除左递归

- ◆ 提取左公因子



改写文法?

## ■ 对每一个非终结符号A，做如下工作：

- ◆ 创建一个初始状态和一个终结状态。

- ◆ 对每一个产生式 $A \rightarrow X_1 X_2 \dots X_n$ 创建一条从初态到终态的路径，有向边的标记依次为 $X_1, X_2, \dots, X_n$ 。

例：为如下文法构造预测分析程序转换图

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id$$

(文法4.3)

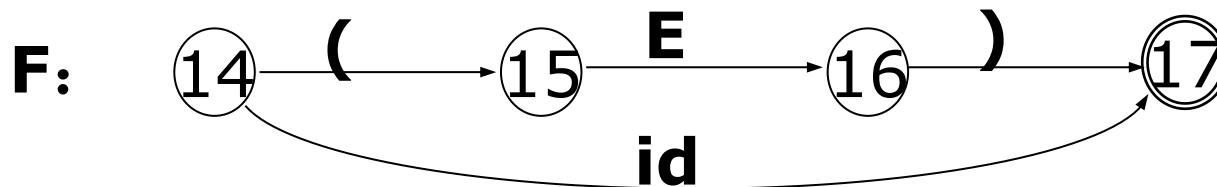
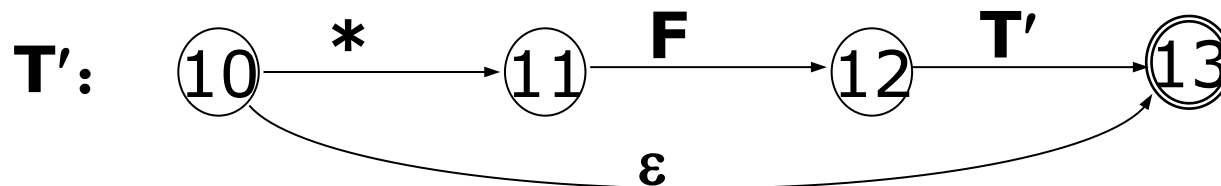
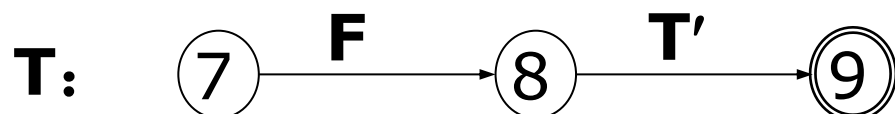
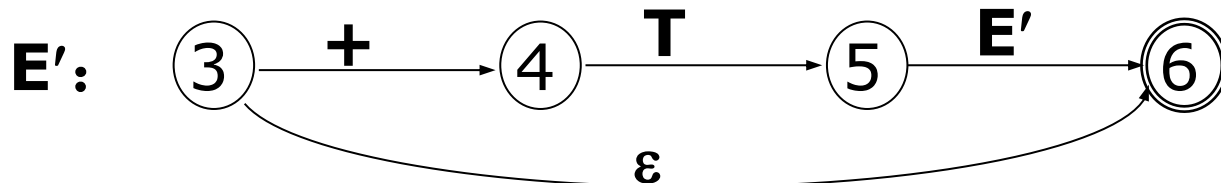
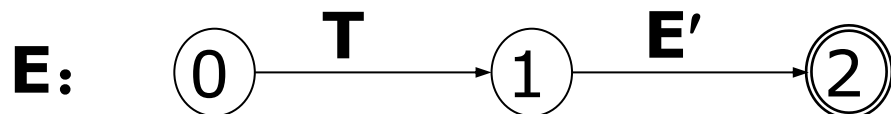
- 消除文法中存在的左递归，得到

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

(文法4.4)

■ 为每个非终结符号构造转换图：

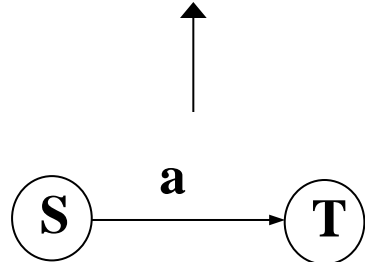
$E \rightarrow TE'$   
 $E' \rightarrow +TE' | \varepsilon$   
 $T \rightarrow FT'$   
 $T' \rightarrow *FT' | \varepsilon$   
 $F \rightarrow (E) | id$



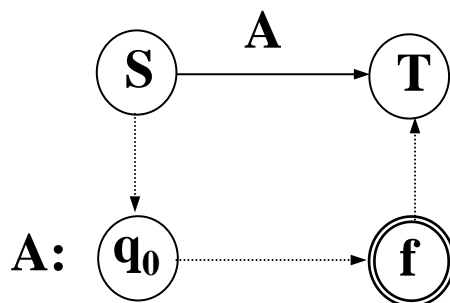
# 转换图的工作过程

- 从文法开始符号所对应的转换图的开始状态开始分析
- 经过若干动作之后，处于状态**S**

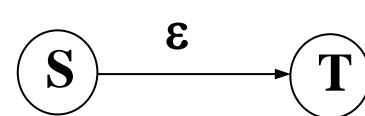
输入串: ... **a** ...



输入串: ... **a** ..... **b** ...

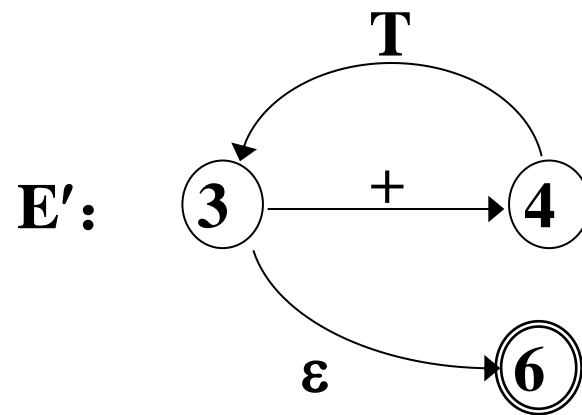
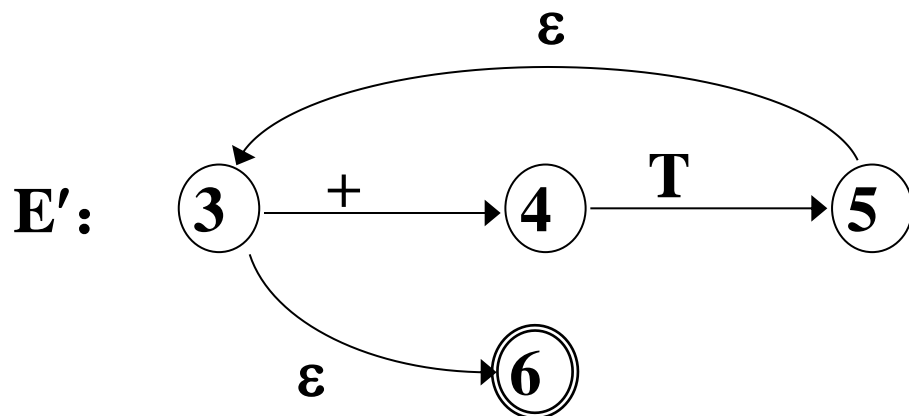
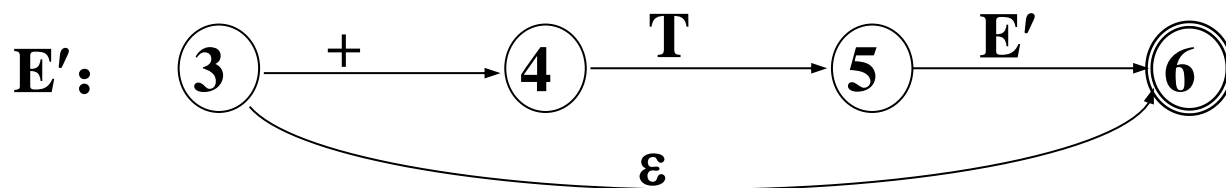


输入串: ... **a** ...



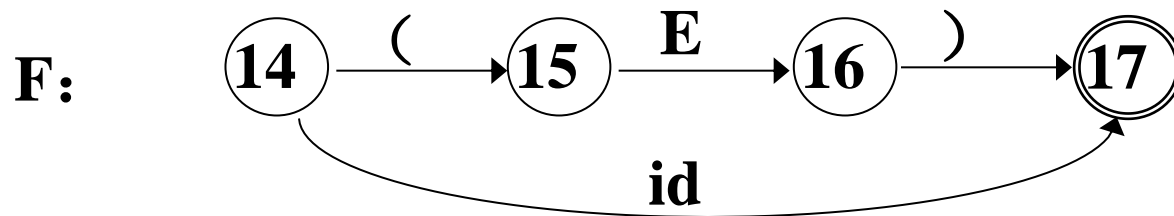
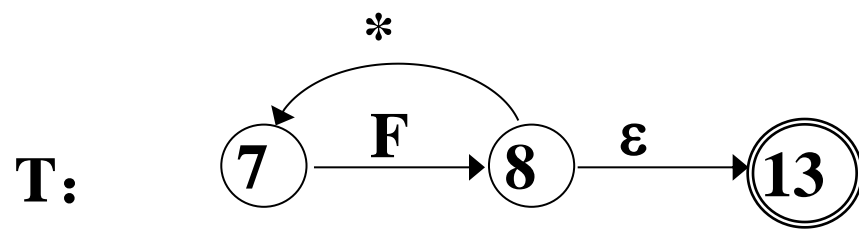
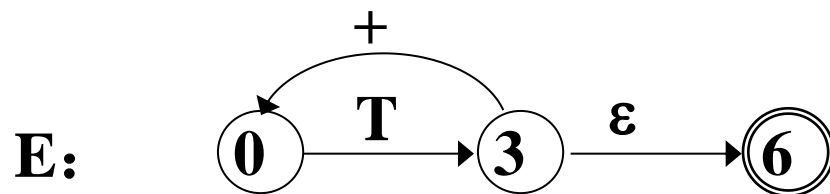
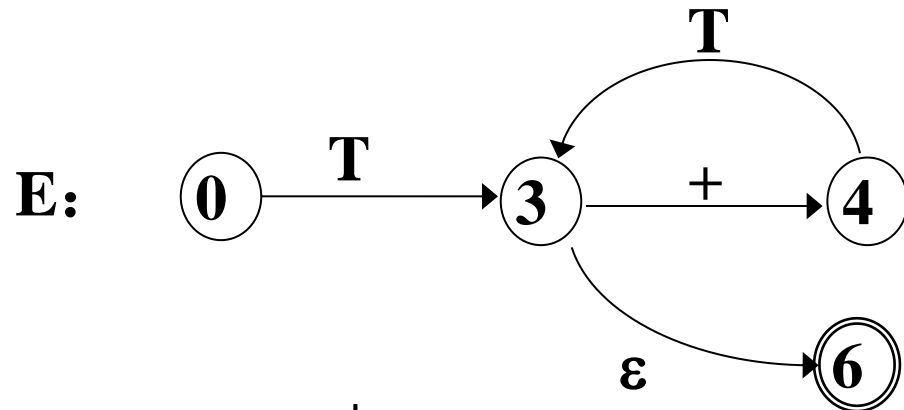
# 转换图的化简

## ■ 用代入的方法进行化简





## ■ 把E'的转换图代入E的转换图:

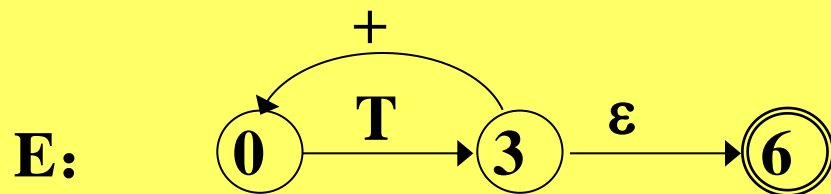


利用状态转换图识别  
符号串 `id+id*id`

# 预测分析程序的实现

- 要求用来描述预测分析程序的语言允许递归调用
- E的过程:

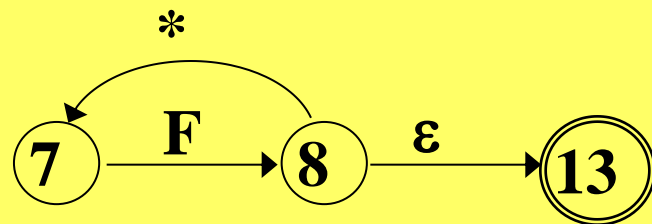
```
void procE(void)
{
    procT();
    if (char == '+') {
        forward pointer;
        procE();
    }
}
```

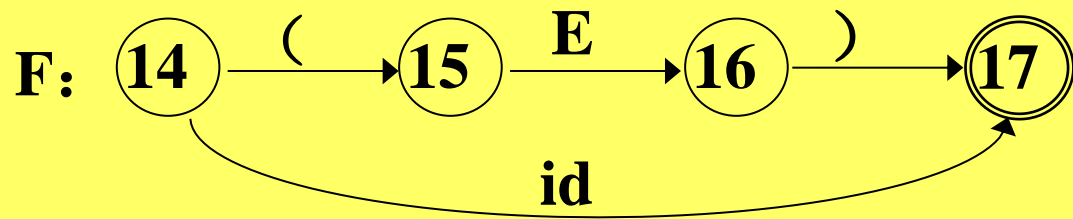


■ T的过程:

```
void procT(void)
{
    procF();
    if (char=='*') {
        forward pointer;
        procT();
    }
}
```

T:





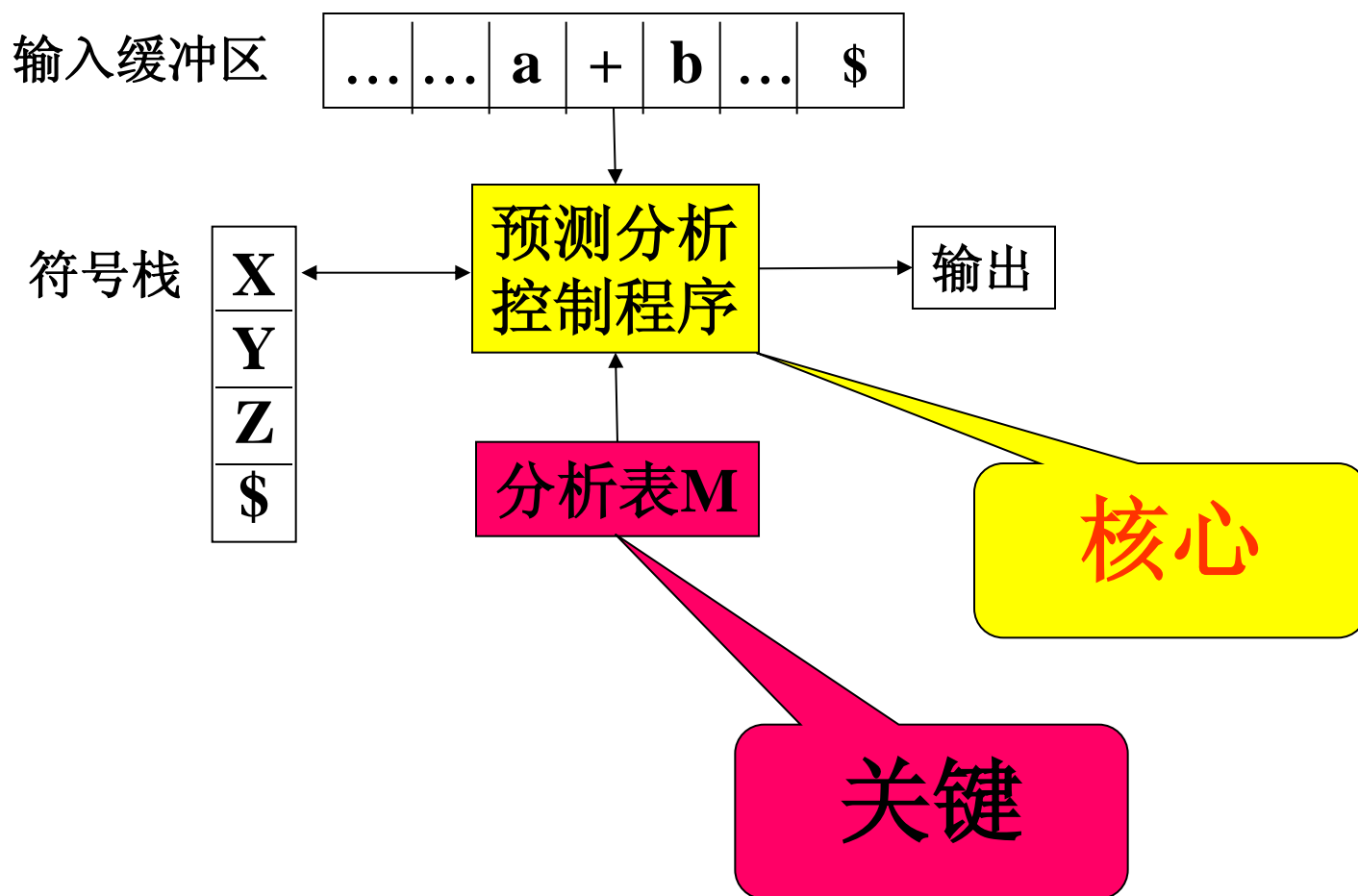
■ F的过程:

```
void procF(void)
{
    if (char=='(') {
        forward pointer;
        procE();
        if (char==')') {
            forward pointer;
        };
        else error();
    };
    else if (char=='id') {
        forward pointer;
        else error();
    }
}
```

# 三、非递归预测分析

- 使用一张**分析表**和一个**栈**联合控制，实现对输入符号串的自顶向下分析。
- 预测分析程序的模型及工作过程
- 预测分析表的构造
- **LL(1)**文法
- 预测分析方法中的错误处理示例

# 预测分析程序的模型及工作过程



# 每部分的作用

## ■ 输入缓冲区:

存放被分析的输入符号串，串后随右尾标志符\$。

... .. \$

## ■ 符号栈:

存放一系列文法符号，\$存于栈底。分析开始时，先将\$入栈，以标识栈底，然后再将文法的开始符号入栈。

\$S

## ■ 分析表:

二维数组 $M[A, a]$ ， $A \in V_N$ ， $a \in V_T \cup \{\$ \}$ 。

根据给定的 $A$ 和 $a$ ，在分析表 $M$ 中找到将被调用的产生式。

## ■ 输出流:

分析过程中不断产生的产生式序列。

$$A \left[ \begin{array}{c} \dots a \dots \\ A \rightarrow a \dots \end{array} \right]$$

## ■ 预测分析控制程序:

根据栈顶符号 $X$ 和当前输入符号 $a$ ，决定分析动作有4种可能:

- $X=a=\$$ ，宣告分析成功，停止分析
- $X=a\neq \$$ ，从栈顶弹出 $X$ ，输入指针前移一个位置
- $X\in V_T$ ，但 $X\neq a$ ，报告发现错误，调用错误处理程序，以报告错误及进行错误恢复。
- 若 $X\in V_N$ ，访问分析表 $M[X,a]$



■  $M[X, a] = X \rightarrow Y_1 Y_2 \dots Y_n$

先将 $X$ 从栈顶弹出，然后把产生式的右部符号串按反序（即按 $Y_n$ 、 $\dots$ 、 $Y_{n-1}$ 、 $Y_2$ 、 $Y_1$ 的顺序）一一推入栈中；

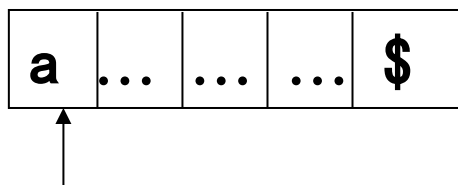
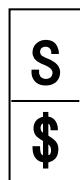
■  $M[X, a] = X \rightarrow \varepsilon$

从栈顶弹出 $X$ ；

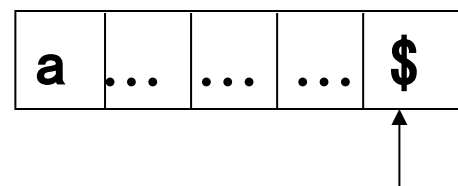
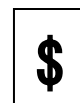
■  $M[X, a] = \text{error}$

调用出错处理程序。

开始状态:



成功状态:



## 算法4.1 非递归预测分析方法

输入：输入符号串 $\omega$ ，文法 $G$ 的一张预测分析表 $M$ 。

输出：若 $\omega$ 在 $L(G)$ 中，则输出 $\omega$ 的最左推导，否则报告错误。

方法：分析开始时， $\$$ 在栈底，文法开始符号 $S$ 在栈顶， $\omega\$$ 在输入缓冲区中置 $ip$ 指向  $\omega\$$  的第一个符号；

```
do {  
    令 $X$ 是栈顶符号， $a$ 是 $ip$ 所指向的符号；  
    if ( $X$ 是终结符号或 $\$$ ) {  
        if ( $X==a$ ) {  
            从栈顶弹出 $X$ ； $ip$ 前移一个位置；  
        };  
        else error();  
    else /*  $X$ 是非终结符号 */  
        if ( $M[X,a]=X\rightarrow Y_1Y_2\cdots Y_k$ ) {  
            从栈顶弹出 $X$ ；  
            把 $Y_k$ 、 $Y_{k-1}$ 、 $\cdots$ 、 $Y_2$ 、 $Y_1$ 压入栈， $Y_1$ 在栈顶；  
            输出产生式 $X\rightarrow Y_1Y_2\cdots Y_k$ ；  
        };  
        else error();  
    }while( $X!=\$$ ) /* 栈不空，继续 */
```

例： 文法4. 4的预测分析表M， 试分析输入串id+id\*id。

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

栈	输入	输出	左句型
\$E	id+id*id\$		E
\$E'T	id+id*id\$	$E \rightarrow TE'$	TE'
\$E'T'F	id+id*id\$	$T \rightarrow FT'$	FT'E'
\$E'T'id	id+id*id\$	$F \rightarrow id$	idT'E'
\$E'T'	+id*id\$		idT'E'
\$E'	+id*id\$	$T' \rightarrow \varepsilon$	idE'

	id	+	*	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

$\$E'T+$	$+id*id\$$	$E' \rightarrow +TE'$	$id+TE'$
$\$E'T$	$id*id\$$		$id+TE'$
$\$E'T'F$	$id*id\$$	$T \rightarrow FT'$	$id+FT'E'$
$\$E'T'id$	$id*id\$$	$F \rightarrow id$	$id+idT'E'$
$\$E'T'$	$*id\$$		$id+idT'E'$
$\$E'T'F*$	$*id\$$	$T' \rightarrow *FT'$	$id+id*FT'E'$
$\$E'T'F$	$id\$$		$id+id*FT'E'$
$\$E'T'id$	$id\$$	$F \rightarrow id$	$id+id*idT'E'$
$\$E'T'$	$\$$		$id+id*idT'E'$
$\$E'$	$\$$	$T' \rightarrow \epsilon$	$id+id*idE'$
$\$$	$\$$	$E' \rightarrow \epsilon$	$id+id*id$

# 预测分析表的构造

## ■ FIRST集合及其构造

定义：对任何文法符号串 $\alpha \in (V_T \cup V_N)^*$ ,

**FIRST**( $\alpha$ )是 $\alpha$ 可以推导出的开头终结符号集合  
描述为：

$$\text{FIRST}(\alpha) = \{ a \mid \alpha \xRightarrow{*} a \cdots, a \in V_T \}$$

若 $\alpha \xRightarrow{*} \varepsilon$ , 则 $\varepsilon \in \text{FIRST}(\alpha)$ 。

## ■ 构造每个文法符号 $X \in V_T \cup V_N$ 的FIRST(X)

- 若 $X \in V_T$ , 则 $\text{FIRST}(X) = \{X\}$ ;
- 若 $X \in V_N$ , 且有产生式 $X \rightarrow a...$ , 其中  $a \in V_T$ , 则把 $a$ 加入到 $\text{FIRST}(X)$ 中;
- 若 $X \rightarrow \varepsilon$ 也是产生式, 则  $\varepsilon$  也加入到 $\text{FIRST}(X)$ 中。
- 若 $X \rightarrow Y...$ 是产生式, 且 $Y \in V_N$ , 则把 $\text{FIRST}(Y)$ 中的所有非 $\varepsilon$ 元素加入到 $\text{FIRST}(X)$ 中;

若 $X \rightarrow Y_1 Y_2 \dots Y_k$ 是产生式, 如果对某个 $i$ ,  $\text{FIRST}(Y_1)$ 、 $\text{FIRST}(Y_2)$ 、...、 $\text{FIRST}(Y_{i-1})$ 都含有 $\varepsilon$ ,

即 $Y_1 Y_2 \dots Y_{i-1} \xRightarrow{*} \varepsilon$ , 则把 $\text{FIRST}(Y_i)$ 中的所有非 $\varepsilon$ 元素加入到 $\text{FIRST}(X)$ 中;

若所有 $\text{FIRST}(Y_i)$ 均含有 $\varepsilon$ , 其中 $i=1, 2, \dots, k$ , 则把 $\varepsilon$  加入到 $\text{FIRST}(X)$ 中。

## ■ FOLLOW集合及其构造

定义：假定**S**是文法**G**的开始符号，对于**G**的任何非终结符号**A**，函数**FOLLOW(A)**是所有句型中，紧跟**A**之后出现的终结符号或\$组成的集合。

描述为：

$$\text{FOLLOW}(A) = \{ a \mid S \xRightarrow{*} \cdots Aa \cdots, a \in V_T \}$$

特别地，若  $S \xRightarrow{*} \cdots A$ ，则规定  $\$ \in \text{FOLLOW}(A)$ 。

## ■ 构造每个非终结符号A的函数FOLLOW(A)

- 对文法开始符号S，置\$于FOLLOW(S)中，\$为输入符号串的右尾标志。
- 若 $A \rightarrow \alpha B \beta$ 是产生式，则把FIRST( $\beta$ )中的所有非 $\epsilon$ 元素加入到FOLLOW(B)中。
- 若 $A \rightarrow \alpha B$ 是产生式，或 $A \rightarrow \alpha B \beta$ 是产生式并且 $\beta \xrightarrow{*} \epsilon$ ，则把FOLLOW(A)中的所有元素加入到FOLLOW(B)中。



例：构造文法4.4中每个非终结符号的 **FIRST** 和 **FOLLOW**函数。

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

	<b>FIRST</b>	<b>FOLLOW</b>
<b>E</b>	(, id	\$, )
<b>E'</b>	+, $\varepsilon$	\$, )
<b>T</b>	(, id	\$, ), +
<b>T'</b>	*, $\varepsilon$	\$, ), +
<b>F</b>	(, id	\$, ), +, *

## 算法4.2 预测分析表的构造方法

输入：文法**G**

输出：文法**G**的预测分析表**M**

方法：

```
for (文法G的每个产生式 $A \rightarrow \alpha$ ) {  
    for (每个终结符号 $a \in \text{FIRST}(\alpha)$ )  
        把 $A \rightarrow \alpha$ 放入 $M[A, a]$ 中;  
    if ( $\epsilon \in \text{FIRST}(\alpha)$ )  
        for (任何 $b \in \text{FOLLOW}(A)$ )  
            把 $A \rightarrow \alpha$ 放入 $M[A, b]$ 中;  
};  
for (所有无定义的 $M[A, a]$ ) 标上错误标志。
```

# 例：为文法4.4构造预测分析表

**$E \rightarrow TE'$**

**$E' \rightarrow +TE' \mid \varepsilon$**

**$T \rightarrow FT'$**

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

**$F \rightarrow (E) \mid id$**

	FIRST	FOLLOW
E	(, id	\$, )
E'	+, $\varepsilon$	\$, )
T	(, id	\$, ), +
T'	*, $\varepsilon$	\$, ), +
F	(, id	\$, ), +, *

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

# LL(1)文法

例：考虑如下映射程序设计语言中if语句的文法

$$S \rightarrow iEtSS' \mid a$$

$$S' \rightarrow eS \mid \varepsilon$$

$$E \rightarrow b \quad (\text{文法4.5})$$

构造各非终结符号的**FIRST**和**FOLLOW**函数：

	S	S'	E
FIRST	i, a	e, $\varepsilon$	b
FOLLOW	\$, e	\$, e	t

应用算法4.2，构造该文法的分析表：

	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow eS$ $S' \rightarrow \varepsilon$			$S' \rightarrow \varepsilon$
E		$E \rightarrow b$				

# LL(1)文法（续）

- 如果一个文法的预测分析表M不含多重定义的表项，则称该文法为LL(1)文法。
- LL(1)的含义：
  - ◆ 第一个L表示从左至右扫描输入符号串
  - ◆ 第二个L表示生成输入串的一个最左推导
  - ◆ 1表示在决定分析程序的每步动作时，向前看一个符号

# LL(1)文法的判断

- 一个文法是LL(1)文法，当且仅当它的每一个产生式 $A \rightarrow \alpha \mid \beta$ ，满足：
  - ◆  $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \phi$  并且
  - ◆ 若 $\beta$ 推导出 $\varepsilon$ ，则 $\text{FIRST}(\alpha) \cap \text{FOLLOW}(A) = \phi$
- 构造出的分析表中不含多重定义的表项
- 文法4.5不是LL(1)文法
  - ◆ 因为 $\text{FIRST}(es) \cap \text{FOLLOW}(S') = \{e\}$
  - ◆  $M[S', e]$ 中有两个产生式

例：判断下面的文法是否为LL(1)文法？

若不是，可否改写为LL(1)文法？

$S \rightarrow (L) | a$

$L \rightarrow L, S | S$

- 文法含有左递归，故不是LL(1)文法
- 改写文法：消除左递归

$S \rightarrow (L) | a$

$L \rightarrow SL'$

$L' \rightarrow ,SL' | \epsilon$

$\text{FIRST}((L)) \cap \text{FIRST}(a) = \phi$

$\text{FIRST}(,SL') \cap \text{FOLLOW}(L') = \phi$

	FIRST	FOLLOW
S	( a	\$ , )
L	( a	)
L'	, ε	)

	a	(	)	,	\$
S	$S \rightarrow a$	$S \rightarrow (L)$			
L	$L \rightarrow SL'$	$L \rightarrow SL'$			
L'			$L' \rightarrow \epsilon$	$L' \rightarrow ,SL'$	

# 预测分析方法中的错误处理示例

- 分析过程中，有两种情况可以发现源程序中的语法错误：
  - ◆  $X \in V_T$ ，但  $X \neq a$ ；
  - ◆  $X \in V_N$ ，但  $M[X, a]$  为空。
- 错误处理方法：
  - ◆ 第一种情况，弹出栈顶的终结符号；
  - ◆ 第二种情况，跳过剩余输入符号串中的若干个符号，直到可以继续进行分析为止。



## ■ 带有同步化信息的分析表的构造

- ◆ 对于  $A \in V_N$ ,  $b \in FOLLOW(A)$ , 若  $M[A, b]$  为空, 则加入 “**synch**”

## ■ 带有同步化信息的分析表的使用

- 若  $M[A, b]$  为空, 则跳过  $a$ ;
- 若  $M[A, b]$  为 **synch**, 则弹出  $A$ 。

# ■ 例：构造文法4.4的带有同步化信息的分析表

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

	FIRST	FOLLOW
E	(, id	\$, )
E'	+, $\varepsilon$	\$, )
T	(, id	\$, ), +
T'	*, $\varepsilon$	\$, ), +
F	(, id	\$, ), +, *

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

作业:  
4.1  
4.3  
4.5

栈	输入	输出
\$E	*id*+id\$	出错, $M[E,*]=$ 空白, 跳过*
\$E	id*+id\$	$E \rightarrow TE'$
\$E'T	id*+id\$	$T \rightarrow FT'$
\$E'T'F	id*+id\$	$F \rightarrow id$
\$E'T'id	id*+id\$	
\$E'T'	*+id\$	$T' \rightarrow *FT'$
\$E'T'F*	*+id\$	
\$E'T'F	+id\$	出错, $M[F,+]=synch$ , 弹出F
\$E'T'	+id\$	$T' \rightarrow \epsilon$
\$E'	+id\$	$E' \rightarrow +TE'$
\$E'T+	+id\$	
\$E'T	id\$	$T \rightarrow FT'$
\$E'T'F	id\$	$F \rightarrow id$
\$E'T'id	id\$	
\$E'T'	\$	$T' \rightarrow \epsilon$
\$E'	\$	$E' \rightarrow \epsilon$
\$	\$	

	id	+	*	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	synch	synch
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	synch		$T \rightarrow FT'$	synch	synch
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$	synch	synch	$F \rightarrow (E)$	synch	synch

## ■ 带有同步化信息的分析表的使用

- 若 $M[A,b]$ 为空, 则跳过a;
- 若 $M[A,b]$ 为synch, 则弹出A。
- $X \in V_T$ , 但 $X \neq a$  弹出栈顶的终结符号;

## 4.3 自底向上分析方法

- 对输入串的扫描：自左向右
- 分析树的构造：自底向上
- 分析过程：
  - ◆ 从输入符号串开始分析
  - ◆ 查找当前句型的“可归约串”
  - ◆ 使用规则，把它归约成相应的非终结符号
  - ◆ 重复
- 关键：找出“可归约串”
- 常用方法：
  - ◆ 算符优先分析方法      ——最左素短语
  - ◆ LR分析方法              ——句柄

# “移进-归约”分析方法

- 符号栈：存放文法符号

- 分析过程：

- (1) 把输入符号一个个地移进栈中。
- (2) 当栈顶的符号串形成某个产生式的一个候选式时，在一定条件下，把该符号串替换（即归约）为该产生式的左部符号。
- (3) 重复(2)，直到栈顶符号串不再是“可归约串”为止。
- (4) 重复(1)–(3)，直到最终归约出文法开始符号S。

# 规范归约

例：分析符号串**abbcd**e是否为如下文法的句子。

1)  $S \rightarrow aAcBe$

2)  $A \rightarrow b$

3)  $A \rightarrow Ab$

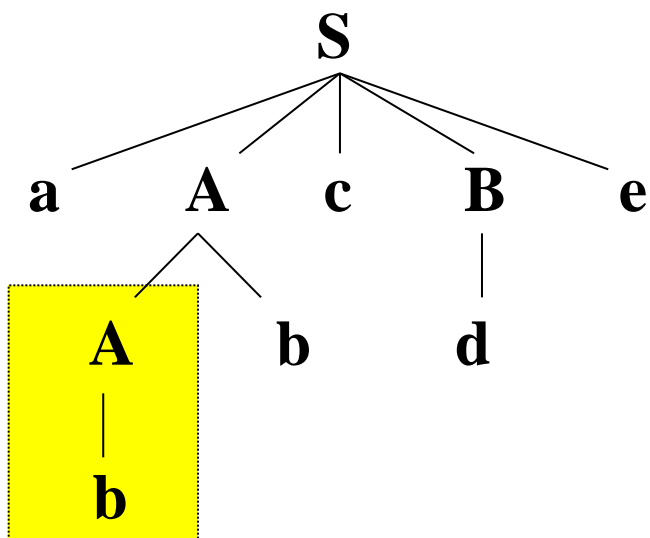
4)  $B \rightarrow d$  (文法4.6)

## ■ 最右推导

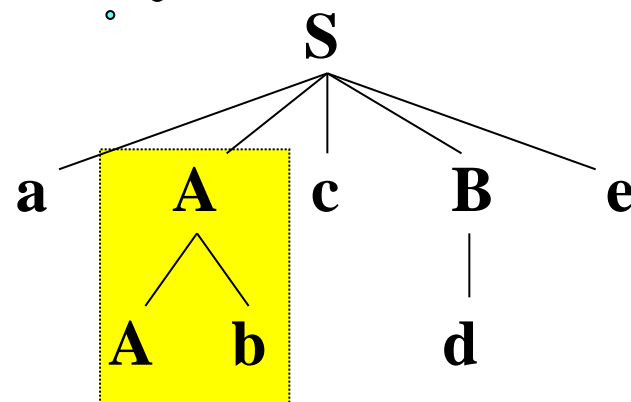
$$\begin{array}{ccccccc} & 1) & & 4) & & 3) & & 2) \\ S & \Rightarrow_{rm} & aAcBe & \Rightarrow_{rm} & aAcde & \Rightarrow_{rm} & aAbcde & \Rightarrow_{rm} & abbcde \end{array}$$

# 最右推导的逆过程

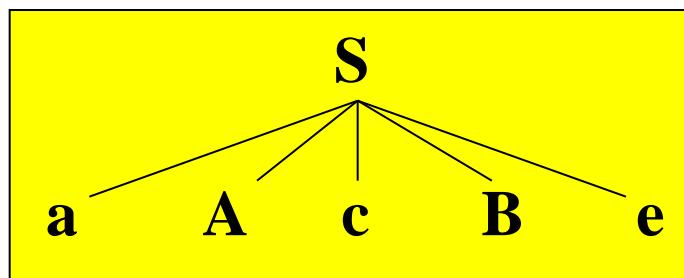
句柄? 子树?



$\Rightarrow$   
(2)



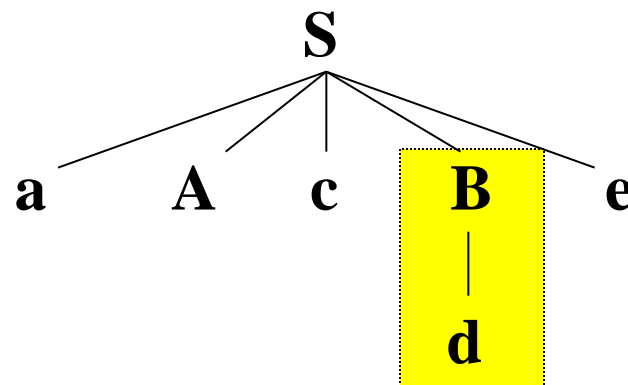
$\Downarrow$  (3)



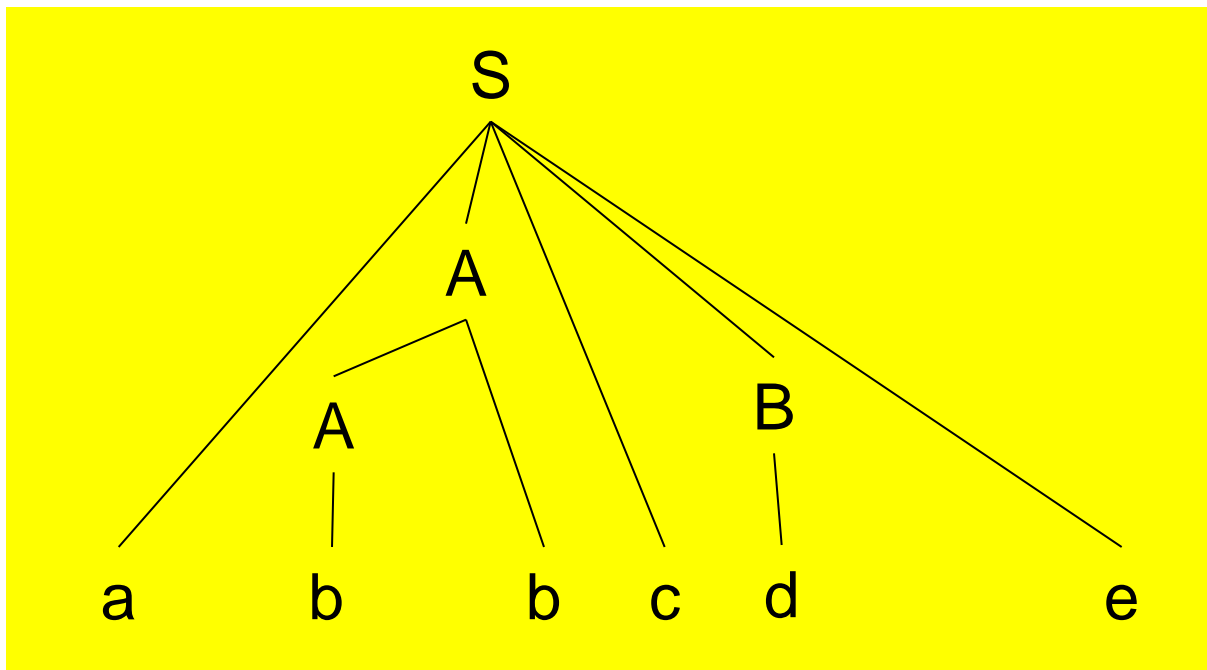
$\Downarrow$  (1)

$S$

$\Leftarrow$   
(4)



# 最右推导的逆过程



**a b b c d e**  
**a A b c d e**  
**a A c d e**  
**a A c B e**  
**S**

**$A \rightarrow b$**

**$A \rightarrow Ab$**

**$B \rightarrow d$**

**$S \rightarrow aAcBe$**

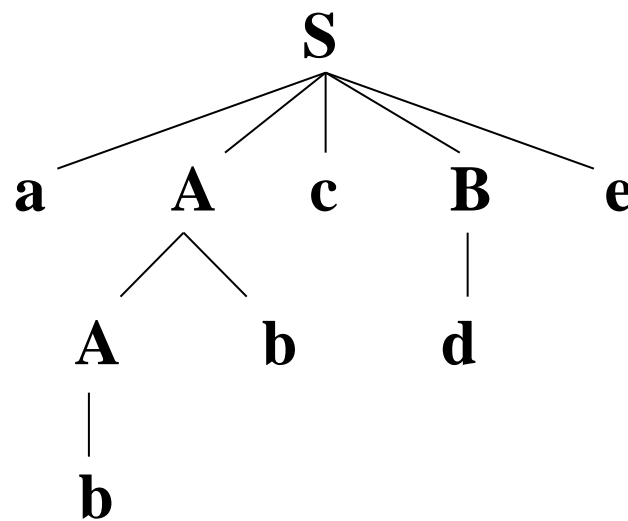
**最右推导:**

**$S \Rightarrow aAcBe$**

**$\Rightarrow aAcde$**

**$\Rightarrow aAbcde$**

**$\Rightarrow abbcde$**





# 规范归约的定义

定义：假定 $\alpha$ 是文法 $G$ 的一个句子，我们称右句型序列  $\alpha_n, \alpha_{n-1}, \dots, \alpha_1, \alpha_0$  是 $\alpha$ 的一个规范归约，如果序列满足：

(1)  $\alpha_n = \alpha, \alpha_0 = S$

(2) 对任何 $i(0 < i \leq n)$ ， $\alpha_{i-1}$ 是经过把 $\alpha_i$ 的句柄替换为相应产生式的左部符号而得到的。

- 规范归约是关于 $\alpha$ 的一个最右推导的逆过程，因此规范归约也称为最左归约。
- **abbcde**的一个规范归约是如下的右句型序列：  
**abbcde, aAbcde, aAcde, aAcBe, S。**

# 句柄的最左性

- 规范句型：最右推导得到的句型
- 规范句型的特点：句柄之后没有非终结符号
- 利用句柄的最左性：与符号栈的栈顶相关
- 不同的最右推导，其逆过程也是不同

例：考虑文法  $E \rightarrow E+E \mid E * E \mid (E) \mid id$  的句子  $id+id*id$

$E \Rightarrow E+E \Rightarrow E+E * E \Rightarrow \underline{E+E * id} \Rightarrow E+id * id \Rightarrow id+id * id$

$E \Rightarrow E * E \Rightarrow E * id \Rightarrow \underline{E+E * id} \Rightarrow E+id * id \Rightarrow id+id * id$

句柄：  $E+E$   
产生式：  $E \rightarrow E+E$

句柄：  $id$   
产生式：  $E \rightarrow id$

# “移进-归约”方法的实现

- 使用一个寄存文法符号的**栈**和一个存放输入符号串的**缓冲区**。
  - ◆ 分析开始时，先将符号\$入栈，以示栈底；
  - ◆ 将\$置入输入符号串之后，以示符号串的结束。



例：对文法4. 6的句子abbcede的规范归约过程

栈	输入	分析动作
1) \$	abbcede\$	shift
2) \$a	bbcede\$	shift
3) \$a <u>b</u>	bcde\$	reduce by $A \rightarrow b$
4) \$aA	bcde\$	shift
5) \$a <u>Ab</u>	cde\$	reduce by $A \rightarrow Ab$
6) \$aA	cde\$	shift
7) \$aAc	de\$	shift
8) \$aAc <u>d</u>	e\$	reduce by $B \rightarrow d$
9) \$aAcB	e\$	shift
10) \$aAc <u>Be</u>	\$	reduce by $S \rightarrow aAcBe$
11) \$S	\$	accept

# 句子abbcbde的规范归约过程

a b b c d e \$

a b b c d e

a A b c d e

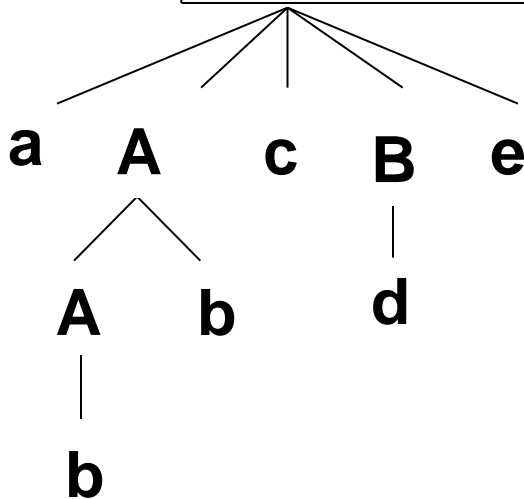
a A c d e

a A c B e

S

栈

\$ S



# “移进-归约”分析方法的分析动作

**移进**：把下一个输入符号移进到栈顶。

**归约**：用适当的归约符号去替换这个串。

**接受**：宣布分析成功，停止分析。

**错误处理**：调用错误处理程序进行诊断和恢复。

## 分析过程中的动作冲突：

“移进-归约”冲突

“归约-归约”冲突

## 4.4 LR分析方法

- 一、**LR**分析技术简介
- 二、**LR**分析程序的模型及工作过程
- 三、**SLR(1)**分析表的构造
- 四、**LR(1)**分析表的构造
- 五、**LALR(1)**分析表的构造
- 六、**LR**分析方法对二义文法的应用
- 七、**LR**分析的错误处理与恢复

# 一、LR分析技术简介

## ■ LR(k)的含义:

- ◆ L 表示自左至右扫描输入符号串
- ◆ R 表示为输入符号串构造一个最右推导的逆过程
- ◆ k 表示为作出分析决定而向前看的输入符号的个数。

## ■ LR分析方法的基本思想

- ◆ “历史信息”：记住已经移进和归约出的整个符号串；
- ◆ “预测信息”：根据所用的产生式推测未来可能遇到的输入符号；
- ◆ 根据“历史信息”和“预测信息”，以及“现实”的输入符号，确定栈顶的符号串是否构成相对于某一产生式的句柄。



# LR分析技术简介(续)

## ■ LR分析技术是一种比较完备的技术

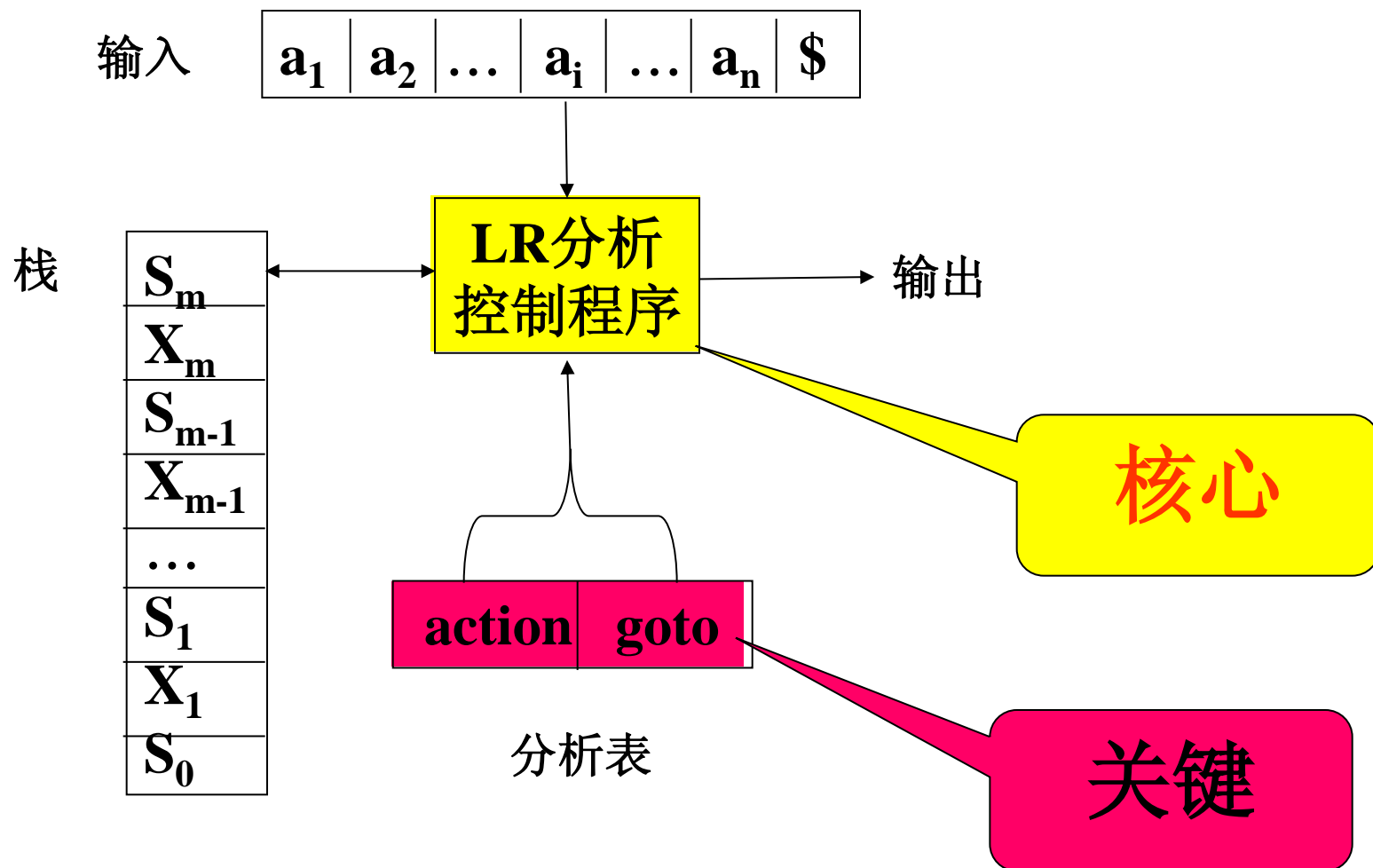
- ◆ 可以分析所有能用上下文无关文法书写的程序设计语言的结构;
- ◆ 最一般的无回溯的“移进-归约”方法;
- ◆ 能分析的文法类是预测分析方法能分析的文法类的真超集;
- ◆ 分析过程中, 能及时发现错误, 快到自左至右扫描输入的最大可能。

## ■ LR分析方法的不足之处

- ◆ 手工编写LR分析程序的工作量太大
- ◆ 需要专门的工具, 即LR分析程序生成器 (如YACC)

## 二、LR分析程序的模型及工作过程

### ■ LR分析程序的模型



# 分析表

- LR分析程序工作的依据
- **goto**[ $S_m$ ,  $X$ ]: 状态 $S_m$ 经 $X$ 转移的一个状态
- **action**[ $S_m$ ,  $a_i$ ]: 状态 $S_m$ 面临输入符号 $a_i$ 时应采取的分析动作

**移进**: 把当前输入符号 $a_i$ 及由 $S_m$ 和 $a_i$ 所决定的下一个状态 $S = \text{goto}[S_m, a_i]$ 推进栈, 向前扫描指针前移。

**归约**: 用某产生式 $A \rightarrow \beta$ 进行归约, 若 $\beta$ 的长度为 $r$ , 归约的动作从栈顶起向下弹出 $2r$ 项, 使 $S_{m-r}$ 成为栈顶状态, 然后把文法符号 $A$ 及状态 $S = \text{goto}[S_{m-r}, A]$ 推进栈。

**接受**: 宣布分析成功, 停止分析。

**出错**: 调用出错处理程序, 进行错误恢复。

# LR分析控制程序

- 核心部分，对所有的LR分析程序都是一样的。
- LR分析控制程序的工作过程
  - 分析开始时，初始的二元式为： $(S_0, a_1a_2\cdots a_n\$)$
  - 分析过程中每步的结果，均可表示为如下的二元式： $(S_0X_1S_1X_2\cdots X_mS_m, a_ia_{i+1}\cdots a_n\$)$
  - 若 $\text{action}[S_m, a_i] = \text{shift } S$ ，并且 $S = \text{goto}[S_m, a_i]$ ，则二元式变为： $(S_0X_1S_1X_2\cdots X_mS_ma_iS, a_{i+1}\cdots a_n\$)$
  - 若 $\text{action}[S_m, a_i] = \text{reduce by } A \rightarrow \beta$ ，则二元式变为： $(S_0X_1S_1X_2\cdots X_{m-r}S_{m-r}AS, a_ia_{i+1}\cdots a_n\$)$
  - 若 $\text{action}[S_m, a_i] = \text{accept}$ （接受），表示分析成功，二元式变化过程终止。
  - 若 $\text{action}[S_m, a_i] = \text{error}$ （出错），表示发现错误，调用错误处理程序。

# “活前缀”的概念

前缀？

定义：一个规范句型的一个前缀，如果不含句柄之后的任何符号，则称它为该句型的一个活前缀。

例：右句型**aAbcde**的句柄是**Ab**

该句型的活前缀有 $\varepsilon$ 、**a**、**aA**、**aAb**；

句型**aAcde**的句柄是**d**

它的活前缀有： $\varepsilon$ 、**a**、**aA**、**aAc**、**aAcd**。

之所以称它们为活前缀，是因为在其右边增加某些终结符号之后，就可以使之成为一个规范句型。

分析过程中  $(S_0X_1S_1X_2\cdots X_mS_m, a_ia_{i+1}\cdots a_n\$)$

$X_1X_2\cdots X_ma_ia_{i+1}\cdots a_n$ 是一个右句型，

$X_1X_2\cdots X_m$ 是它的一个活前缀。

## 算法4.3 LR分析控制程序

输入：文法**G**的一张分析表和一个输入符号串 $\omega$

输出：若 $\omega \in L(G)$ ，得到 $\omega$ 的自底向上的分析，否则报错

方法：开始时，初始状态 $S_0$ 在栈顶， $\omega\$$ 在输入缓冲区中。

置 $ip$ 指向 $\omega\$$ 的第一个符号；

**do {**

    令**S**是栈顶状态，**a**是 $ip$ 所指向的符号

**if (action[S, a] == shift S') {**

        把**a**和**S'**依次入栈；

        推进 $ip$ ，使它指向下一个输入符号；

**};**

**else if (action[S, a] == reduce by  $A \rightarrow \beta$ ) {**

        从栈顶弹出 $2 * |\beta|$ 个符号；

        令**S'**是现在的栈顶状态，把**A**和**goto[S', A]**入栈；

        输出产生式 $A \rightarrow \beta$ ；

**};**

**else if (action[S, a] == accept) return;**

**else error();**

**} while(1).**

# 例：对输入符号串 $id+id*id$ 的分析过程

栈	输入	分析动作	
0	$id+id*id\$$	shift 5	
0id5	$+id*id\$$	reduce by $F \rightarrow id$	goto[0,F]=3
0F3	$+id*id\$$	reduce by $T \rightarrow F$	goto[0,T]=2
0T2	$+id*id\$$	reduce by $E \rightarrow T$	goto[0,E]=1
0E1	$+id*id\$$	shift 6	
0E1+6	$id*id\$$	shift 5	
0E1+6id5	$*id\$$	reduce by $F \rightarrow id$	goto[6,F]=3
0E1+6F3	$*id\$$	reduce by $T \rightarrow F$	goto[6,T]=9
0E1+6T9	$*id\$$	shift 7	
0E1+6T9*7	$id\$$	shift 5	
0E1+6T9*7id5	$\$$	reduce by $F \rightarrow id$	goto[7,F]=10
0E1+6T9*7F10	$\$$	reduce by $T \rightarrow T * F$	goto[6,T]=9
0E1+6T9	$\$$	reduce by $E \rightarrow E + T$	goto[0,E]=1
0E1	$\$$	acc	

# 三、SLR(1)分析表的构造

- 中心思想：
  - ◆ 为给定的文法构造一个识别它所有活前缀的**DFA**
  - ◆ 根据该**DFA**构造文法的分析表
- 构造识别给定文法的所有活前缀的**DFA**
- **SLR(1)**分析表的构造



# 构造识别给定文法所有活前缀的DFA

- 活前缀与句柄之间的关系
  - ◆ 活前缀不含有句柄的任何符号
  - ◆ 活前缀只含有句柄的部分符号
  - ◆ 活前缀已经含有句柄的全部符号
- 分析过程中，分析栈中出现的活前缀
  - ◆ 第一种情况，期望从剩余输入串中能够看到由某产生式 $A \rightarrow \alpha$ 的右部 $\alpha$ 所推导出的终结符号串；
  - ◆ 第二种情况，某产生式 $A \rightarrow \alpha_1 \alpha_2$ 的右部子串 $\alpha_1$ 已经出现在栈顶，期待从剩余的输入串中能够看到 $\alpha_2$ 推导出的符号串；
  - ◆ 第三种情况，某一产生式 $A \rightarrow \alpha$ 的右部符号串 $\alpha$ 已经出现在栈顶，用该产生式进行归约。

# LR(0)项目

- 右部某个位置上标有圆点的产生式称为文法G的一个LR(0)项目
- 产生式 $A \rightarrow XYZ$ 对应应有4个LR(0)项目

$A \rightarrow \cdot XYZ$

$A \rightarrow X \cdot YZ$

$A \rightarrow XY \cdot Z$

$A \rightarrow XYZ \cdot$

} 移进-待约项目

归约项目

- 归约项目：圆点在产生式最右端的LR(0)项目
- 接受项目：对文法开始符号的归约项目
- 待约项目：圆点后第一个符号为非终结符号的LR(0)项目
- 移进项目：圆点后第一个符号为终结符号的LR(0)项目
- 产生式 $A \rightarrow \varepsilon$ ，只有一个LR(0)归约项目  $A \rightarrow \cdot$ 。

# 拓广文法

- 任何文法  $G = (V_T, V_N, S, \varphi)$ ，都有等价的文法  $G' = (V_T, V_N \cup \{S'\}, S', \varphi \cup \{S' \rightarrow S\})$ ，称  $G'$  为  $G$  的拓广文法。
- 拓广文法  $G'$  的接受项目是唯一的(即  $S' \rightarrow S \cdot$ )

## 定义4.10: LR(0)有效项目

项目 $A \rightarrow \beta_1 \beta_2$ 对活前缀 $\gamma = \alpha \beta_1$ 是有效的, 如果存在一个规范推倒导:

$$S \xRightarrow{*} \alpha A \omega \Rightarrow \alpha \beta_1 \beta_2 \omega.$$

- 推广: 若项目 $A \rightarrow \alpha B \beta$ 对活前缀 $\gamma = \delta \alpha$ 是有效的, 并且 $B \rightarrow \eta$ 是一个产生式, 则项目 $B \rightarrow \eta$ 对活前缀 $\gamma = \delta \alpha$ 也是有效的。
- LR(0)项目 $S' \rightarrow S$ 是活前缀 $\epsilon$ 的有效项目  
(这里取 $\gamma = \epsilon$ ,  $\beta_1 = \epsilon$ ,  $\beta_2 = S$ ,  $A = S'$ )

## 定义4. 11: LR(0)有效项目集 和LR(0)项目集规范族

文法**G**的某个活前缀 $\gamma$ 的所有**LR(0)**有效项目组成的集合称为 $\gamma$ 的**LR(0)**有效项目集。

文法**G**的所有**LR(0)**有效项目集组成的集合称为**G**的**LR(0)**项目集规范族。

## 定义4.12: 闭包 (closure)

设 $I$ 是文法 $G$ 的一个 $LR(0)$ 项目集合,  $\text{closure}(I)$ 是从 $I$  出发, 用下面的方法构造的项目集:

- (1)  $I$ 中的每一个项目都属于 $\text{closure}(I)$ ;
- (2) 若项目 $A \rightarrow \alpha \cdot B \beta$ 属于 $\text{closure}(I)$ ,  
且 $B \rightarrow \eta$ 是 $G$ 的一个产生式,  
若 $B \rightarrow \cdot \eta$ 不属于 $\text{closure}(I)$ ,  
则将 $B \rightarrow \cdot \eta$ 加入 $\text{closure}(I)$ ;
- (3) 重复规则(2), 直到 $\text{closure}(I)$ 不再增大为止。

## 定义4.13: 转移函数go

若 $I$ 是文法 $G$ 的一个 $LR(0)$ 项目集， $X$ 是一个文法符号，定义

$$go(I, X) = closure(J)$$

其中： $J = \{ A \rightarrow \alpha X \cdot \beta \mid \text{当 } A \rightarrow \alpha \cdot X \beta \text{ 属于 } I \text{ 时} \}$

$go(I, X)$ 称为转移函数

项目 $A \rightarrow \alpha X \cdot \beta$ 称为 $A \rightarrow \alpha \cdot X \beta$ 的后继

**直观含义：**若 $I$ 中的项目 $A \rightarrow \alpha \cdot X \beta$ 是某个活前缀 $\gamma = \delta \alpha$ 的有效项目， $J$ 中的项目 $A \rightarrow \alpha X \cdot \beta$ 是活前缀 $\delta \alpha X$ （即 $\gamma X$ ）的有效项目。

## 算法4.4 构造文法G的LR(0)项目集规范族

输入：文法G

输出：G的LR(0)项目集规范族C

方法：

**C = {closure({ $S' \rightarrow \cdot S$ })};**

**do**

**for** (对C中的每一个项目集I和每一个文法符号X)

**if** ( $\text{go}(I, X)$ 不为空, 且不在C中)

    把 $\text{go}(I, X)$ 加入C中;

**while** (没有新项目集加入C中);

这里 $\text{closure}(\{S' \rightarrow \cdot S\})$ 是活前缀  $\varepsilon$  的有效项目集



例：构造如下文法G的LR(0)项目集规范族：

$S \rightarrow aA | bB$     $A \rightarrow cA | d$     $B \rightarrow cB | d$    （文法4.7）

■ 拓广文法G'：

$S' \rightarrow S$     $S \rightarrow aA | bB$     $A \rightarrow cA | d$     $B \rightarrow cB | d$

■ 活前缀 $\varepsilon$ 的有效项目集

$I_0 = \text{closure}(\{S' \rightarrow \cdot S\}) = \{ S' \rightarrow \cdot S, S \rightarrow \cdot aA, S \rightarrow \cdot bB \}$

■ 从 $I_0$ 出发的转移有

活前缀

$I_1 = \text{go}(I_0, S) = \text{closure}(\{S' \rightarrow S \cdot\}) = \{S' \rightarrow S \cdot\}$    ——S

$I_2 = \text{go}(I_0, a) = \text{closure}(\{S \rightarrow a \cdot A\})$   
 $= \{S \rightarrow a \cdot A, A \rightarrow \cdot cA, A \rightarrow \cdot d\}$    ——a

$I_3 = \text{go}(I_0, b) = \text{closure}(\{S \rightarrow b \cdot B\})$   
 $= \{S \rightarrow b \cdot B, B \rightarrow \cdot cB, B \rightarrow \cdot d\}$    ——b

## ■ 从 $I_2$ 出发的转移有

活前缀

$$I_4 = \text{go}(I_2, A) = \text{closure}(\{S \rightarrow aA \cdot\}) = \{S \rightarrow aA \cdot\} \quad - \quad aA$$

$$\begin{aligned} I_5 &= \text{go}(I_2, c) = \text{closure}(\{A \rightarrow c \cdot A\}) \\ &= \{A \rightarrow c \cdot A, A \rightarrow \cdot cA, A \rightarrow \cdot d\} \quad - \quad ac \end{aligned}$$

$$I_6 = \text{go}(I_2, d) = \text{closure}(\{A \rightarrow d \cdot\}) = \{A \rightarrow d \cdot\} \quad - \quad ad$$

## ■ 从 $I_3$ 出发的转移有

活前缀

$$I_7 = \text{go}(I_3, B) = \text{closure}(\{S \rightarrow bB \cdot\}) = \{S \rightarrow bB \cdot\} \quad - \quad bB$$

$$\begin{aligned} I_8 &= \text{go}(I_3, c) = \text{closure}(\{B \rightarrow c \cdot B\}) \\ &= \{B \rightarrow c \cdot B, B \rightarrow \cdot cB, B \rightarrow \cdot d\} \quad - \quad bc \end{aligned}$$

$$I_9 = \text{go}(I_3, d) = \text{closure}(\{B \rightarrow d \cdot\}) = \{B \rightarrow d \cdot\} \quad - \quad bd$$

- 从 $I_5$ 出发的转移有
 

$I_{10} = \text{go}(I_5, A) = \text{closure}(\{A \rightarrow cA \cdot\}) = \{A \rightarrow cA \cdot\}$	活前缀
$\text{go}(I_5, c) = \text{closure}(\{A \rightarrow c \cdot A\}) = I_5$	—acA
$\text{go}(I_5, d) = \text{closure}(\{A \rightarrow d \cdot\}) = I_6$	—acc
	—acd

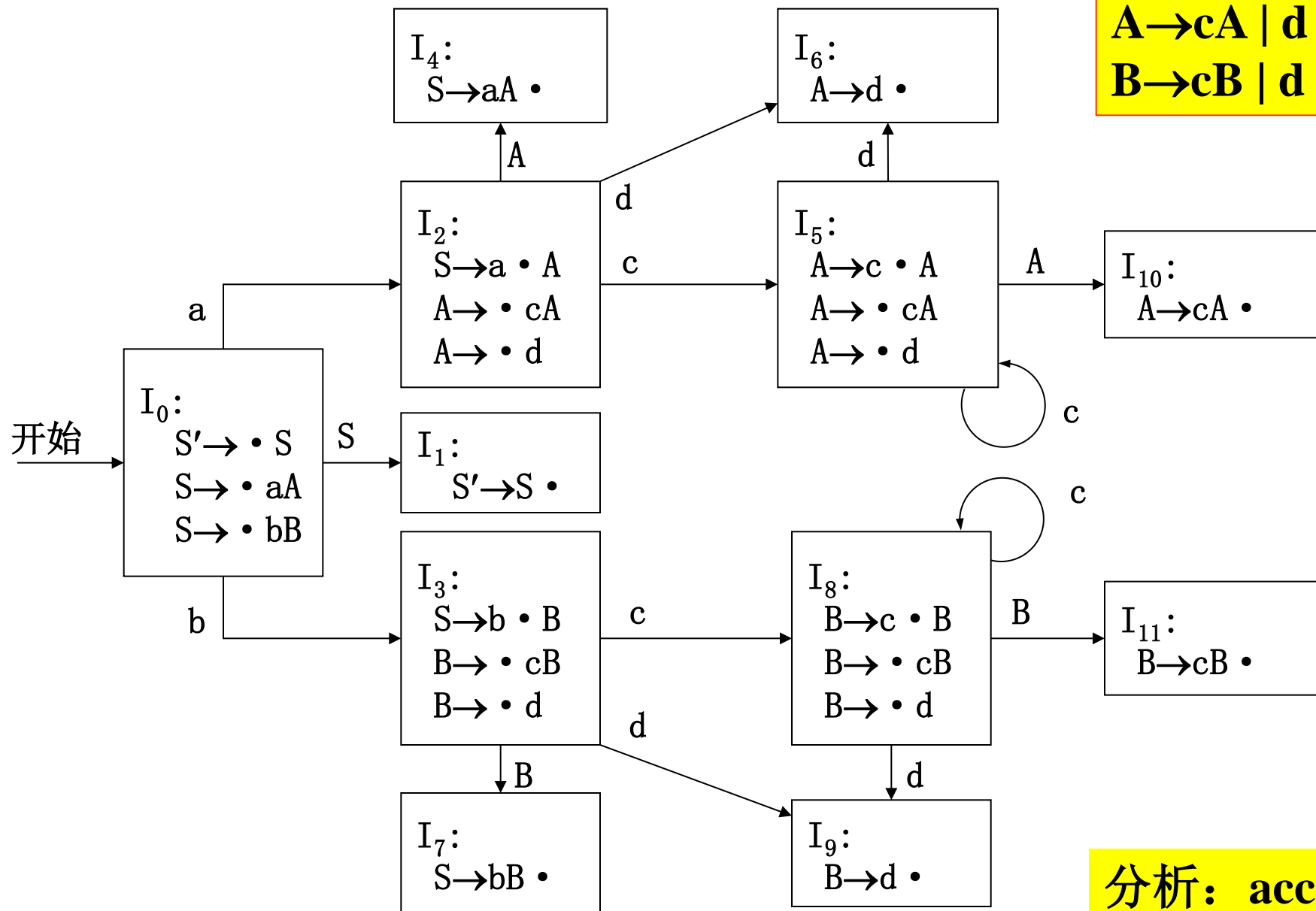
- 从 $I_8$ 出发的转移有
 

$I_{11} = \text{go}(I_8, B) = \text{closure}(\{B \rightarrow cB \cdot\}) = \{B \rightarrow cB \cdot\}$	活前缀
$\text{go}(I_8, c) = \text{closure}(\{B \rightarrow c \cdot B\}) = I_8$	—bcB
$\text{go}(I_8, d) = \text{closure}(\{B \rightarrow d \cdot\}) = I_9$	—bcc
	—bcd

- 文法 $G'$ 的LR(0)项目集规范族
 
$$C = \{I_0, I_1, \dots, I_{11}\}$$

# 识别文法G'的所有活前缀的DFA

$S' \rightarrow S$   
 $S \rightarrow aA \mid bB$   
 $A \rightarrow cA \mid d$   
 $B \rightarrow cB \mid d$



分析: **accd**

# SLR(1) 分析表的构造

- SLR分析方法的一个特征
  - ◆ 如果文法的有效项目集中有冲突动作，多数冲突可通过考察有关非终结符号的FOLLOW集合而得到解决
- 如项目集： $I = \{X \rightarrow \alpha \cdot b \beta, A \rightarrow \alpha \cdot, B \rightarrow \alpha \cdot\}$ 
  - ◆ 存在移进-归约冲突
  - ◆ 存在归约-归约冲突
- 冲突的解决方法，查看FOLLOW(A)和FOLLOW(B)
  - ◆  $FOLLOW(A) \cap FOLLOW(B) = \Phi$
  - ◆  $b \notin FOLLOW(A)$  并且  $b \notin FOLLOW(B)$
  - ◆ 决策：
    - 当 $a=b$ 时，把 $b$ 移进栈里；
    - 当 $a \in FOLLOW(A)$ 时，用产生式 $A \rightarrow \alpha$ 进行归约；
    - 当 $a \in FOLLOW(B)$ 时，用产生式 $B \rightarrow \alpha$ 进行归约。

## 算法4.5 构造SLR(1)分析表

输入：拓广文法 $G'$  输出： $G'$ 的SLR分析表 方法如下：

- 1.构造 $G'$ 的LR(0)项目集规范族 $C=\{I_0, I_1, \dots, I_n\}$ 。
- 2.对于状态 $i$ （对应于项目集 $I_i$ 的状态）的分析动作如下
  - a) 若 $A \rightarrow \alpha \cdot a \beta \in I_i$ ，且 $go(I_i, a) = I_j$ ，则置  
 $action[i, a] = sj$
  - b) 若 $A \rightarrow \alpha \cdot \in I_i$ ，则对所有 $a \in FOLLOW(A)$ ，置  
 $action[i, a] = r \ A \rightarrow \alpha$
  - c) 若 $S' \rightarrow S \cdot \in I_i$ ，则置 $action[i, \$] = acc$ ，表示分析成功
- 3.若 $go(I_i, A) = I_j$ ， $A$ 为非终结符号，则置 $goto[i, A] = j$
- 4.分析表中凡不能用规则(2)、(3)填入信息的空白表项，均置为出错标志error。
- 5.分析程序的初态是包含项目 $S' \rightarrow S$ 的有效项目集所对应的状态。

# 例：构造文法4.7的SLR(1)分析表

- 构造出该文法的LR(0)项目集规范族、及识别文法活前缀的DFA
- 考察  $I_0 = \{ S' \rightarrow S, S \rightarrow aA, S \rightarrow bB \}$ 
  - ◆ 对项目  $S' \rightarrow S$ , 有  $go(I_0, S) = I_1$ , 所以  $goto[0, S] = 1$
  - ◆ 对项目  $S \rightarrow aA$ , 有  $go(I_0, a) = I_2$ , 所以  $action[0, a] = s2$
  - ◆ 对项目  $S \rightarrow bB$ , 有  $go(I_0, b) = I_3$ , 所以  $action[0, b] = s3$
- 考察  $I_1 = \{ S' \rightarrow S \}$ 
  - ◆ 项目  $S' \rightarrow S \cdot$  是接受项目, 所以  $action[1, \$] = acc$
- 考察  $I_2 = \{ S \rightarrow aA, A \rightarrow cA, A \rightarrow d \}$ 
  - ◆ 对项目  $S \rightarrow aA$ , 有  $go(I_2, A) = I_4$ , 所以  $goto[2, A] = 4$
  - ◆ 对项目  $A \rightarrow cA$ , 有  $go(I_2, c) = I_5$ , 所以  $action[2, c] = s5$
  - ◆ 对项目  $A \rightarrow d$ , 有  $go(I_2, d) = I_6$ , 所以  $action[2, d] = s6$

■ 考察  $I_3 = \{S \rightarrow b B, B \rightarrow cB, B \rightarrow d\}$

- ◆ 对项目  $S \rightarrow b B$ , 有  $go(I_3, B) = I_7$ , 所以  $goto[3, B] = 7$
- ◆ 对项目  $B \rightarrow cB$ , 有  $go(I_3, c) = I_8$ , 所以  $action[3, c] = s8$
- ◆ 对项目  $B \rightarrow d$ , 有  $go(I_3, d) = I_9$ , 所以  $action[3, d] = s9$

■ 考察  $I_4 = \{ S \rightarrow aA \}$

- ◆ 项目  $S \rightarrow aA$  是归约项目, 因为  $FOLLOW(S) = \{\$ \}$ , 所以  $action[4, \$] = r1$

■ 考察  $I_5 = \{ A \rightarrow c A, A \rightarrow cA, A \rightarrow d \}$

- ◆ 对项目  $A \rightarrow c A$ , 有  $go(I_5, A) = I_{10}$ , 所以  $goto[5, A] = 10$
- ◆ 对项目  $A \rightarrow cA$ , 有  $go(I_5, c) = I_5$ , 所以  $action[5, c] = s5$
- ◆ 对项目  $A \rightarrow d$ , 有  $go(I_5, d) = I_6$ , 所以  $action[5, d] = s6$

■ 考察  $I_6 = \{ A \rightarrow d \}$

- ◆ 项目  $A \rightarrow d$  是归约项目, 因为  $FOLLOW(A) = \{\$ \}$ , 所以  $action[6, \$] = r4$



# 文法4. 7的SLR(1) 分析表

状态	action					goto		
	a	b	c	d	\$	S	A	B
0	s2	S3				1		
1					acc			
2			s5	s6			4	
3			s8	s9				7
4					r1			
5			s5	s6			10	
6					r4			
7					r2			
8			s8	s9				11
9					r6			
10					r3			
11					r5			

# SLR(1)文法

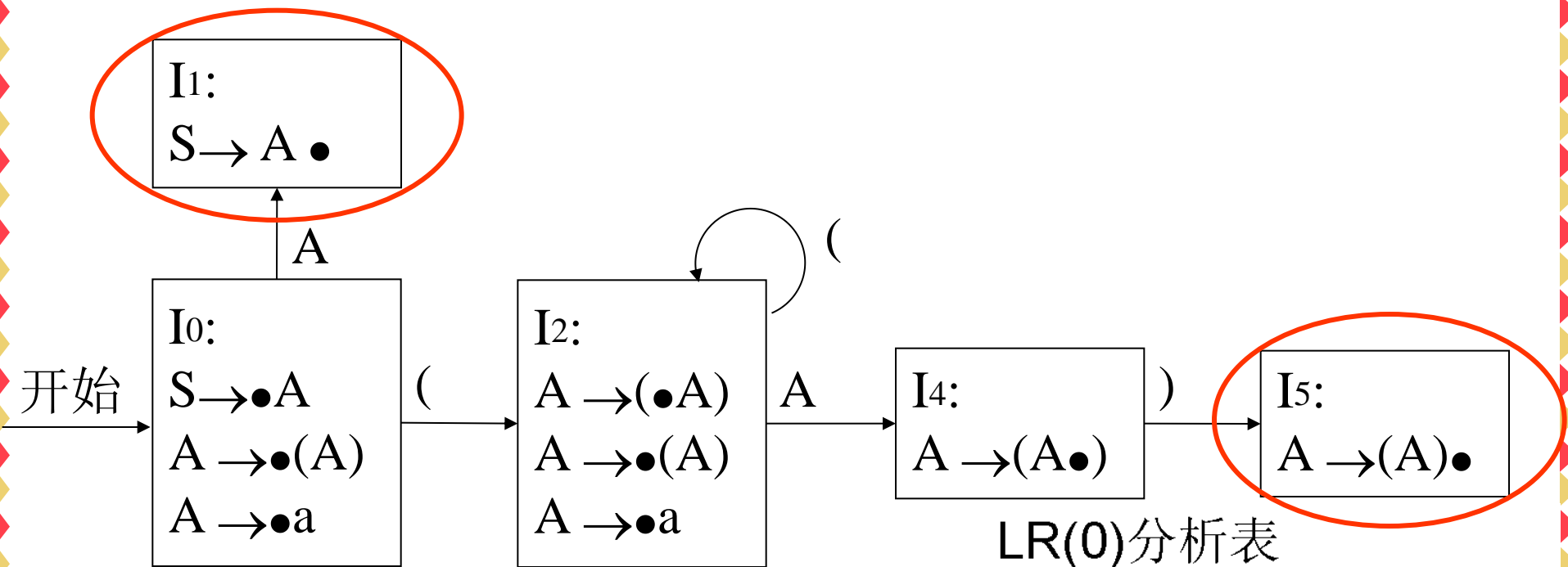
- 若用算法4.5构造出来的分析表不含有冲突，则该分析表称为该文法的SLR(1)分析表
- 具有SLR(1)分析表的文法称为SLR(1)文法

# LR(0)分析表和LR(0)文法

- 如果在执行上述算法的过程中，始终没有向前看任何输入符号，则构造的SLR分析表称为LR(0)分析表
- 具有LR(0)分析表的文法称为LR(0)文法。
- 一个文法是LR(0)文法，当且仅当该文法的每个活前缀的有效项目集中：
  - ◆ 要么所有元素都是移进-待约项目
  - ◆ 要么只含有唯一的归约项目
- 具有如下产生式的文法是一个LR(0)文法。

$$A \rightarrow (A) | a$$

拓广文法: 0)  $S \rightarrow A$  1)  $A \rightarrow (A)$  2)  $A \rightarrow a$



LR(0)分析表

状态	action				goto
	a	(	)	\$	
0	s3	s2			1
1				acc	
2	s3	s2			4
3	r2	r2	r2	r2	
4			s5		
5	r1	r1	r1	r1	

每一个**SLR(1)**文法都是无二义的文法  
但并非无二义的文法都是**SLR(1)**文法。

- 例：有文法**G**具有如下产生式：

$S \rightarrow L = R$

$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow id$

$R \rightarrow L$  (文法4.8)

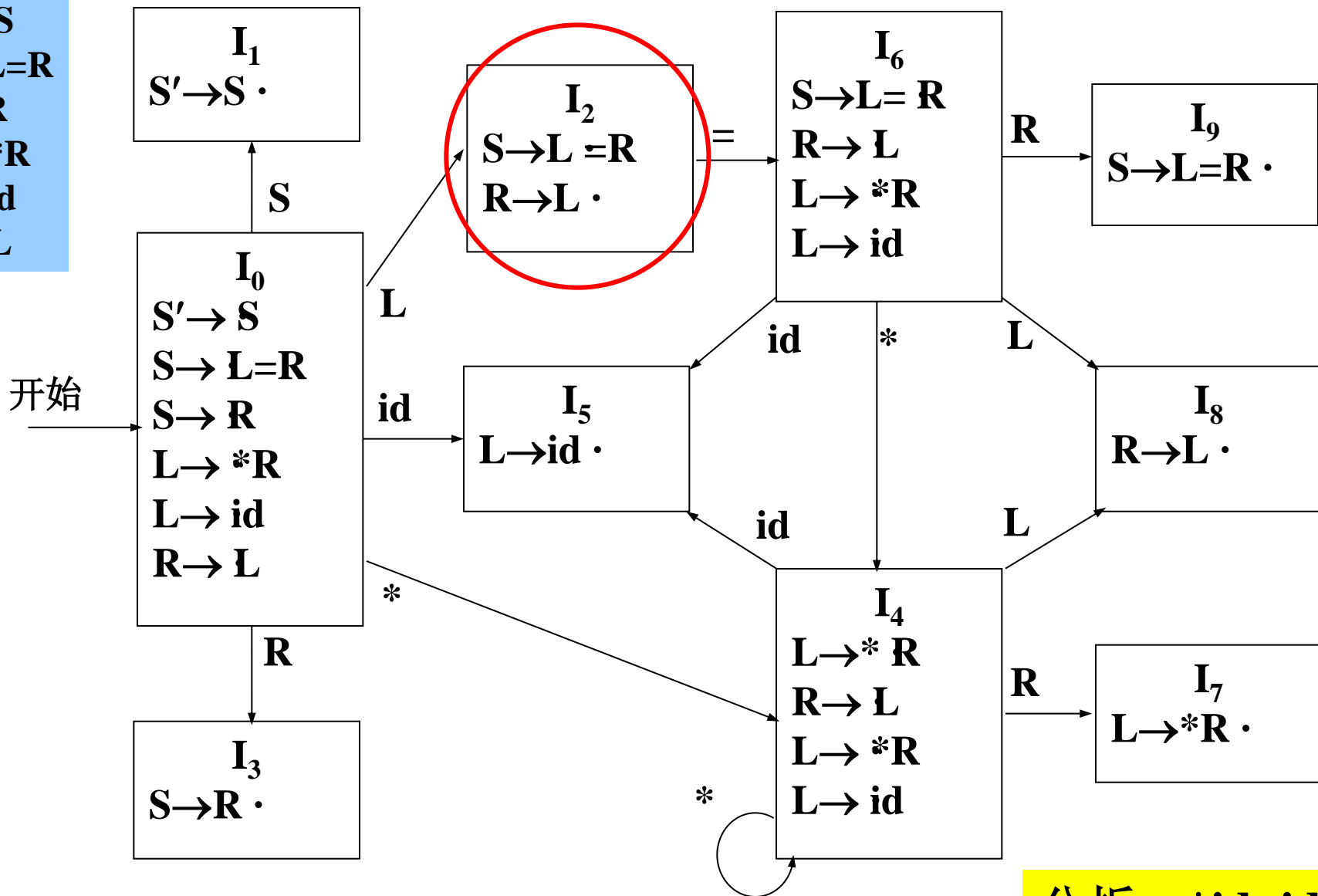
- 该文法无二义性，但不是**SLR(1)**文法。
- 拓广文法**G'**的产生式如下：

(0)  $S' \rightarrow S$       (1)  $S \rightarrow L = R$       (2)  $S \rightarrow R$

(3)  $L \rightarrow *R$       (4)  $L \rightarrow id$       (5)  $R \rightarrow L$

# 构造文法G'的LR(0)项目集规范族及识别活前缀的DFA

$S' \rightarrow S$   
 $S \rightarrow L=R$   
 $S \rightarrow R$   
 $L \rightarrow *R$   
 $L \rightarrow id$   
 $R \rightarrow L$



分析:  $*id=id$

# 应用算法4.5，为之构造SLR分析表

## ■ 考察 $I_2$ :

- ◆ 项目 $S \rightarrow L \cdot = R$ 与 $R \rightarrow L \cdot$ 存在移进-归约冲突，
- ◆ 根据第一个项目，有： $action[2, =] = s6$
- ◆ 根据第二个项目，由于 $OLLOW(R) = \{=, \$\}$ 所以有： $action[2, \$] = action[2, =] = r5$

## ■ 存在“移进-归约”冲突。

## ■ 原因是在SLR分析表中未包含足够多的预测信息。

## 四、LR(1)分析表的构造

- LR(k)项目:  $[A \rightarrow \alpha \cdot \beta, a_1 a_2 \dots a_k]$ 
  - ◆  $A \rightarrow \alpha \cdot \beta$  是一个LR(0)项目
  - ◆  $a_i (i=1, 2, \dots, k)$  是终结符号
  - ◆  $a_1 a_2 \dots a_k$  称为该项目的向前看符号串
- 向前看符号串仅对归约项目  $[A \rightarrow \alpha \cdot, a_1 a_2 \dots a_k]$  起作用, 对任何移进或待约项目  $[A \rightarrow \alpha \cdot \beta, a_1 a_2 \dots a_k]$ , ( $\beta \neq \epsilon$ ) 是没有意义的。
- 归约项目  $[A \rightarrow \alpha \cdot, a_1 a_2 \dots a_k]$  意味着, 当它所属项目集对应的状态在栈顶, 且后续的k个输入符号为  $a_1 a_2 \dots a_k$  时, 才允许把栈顶的文法符号串  $\alpha$  归约为  $A$ 。



## 定义4.14: LR(1)有效项目

称一个LR(1)项目  $[A \rightarrow \alpha \beta, a]$  对活前缀  $\gamma = \delta\alpha$  是有效的, 如果存在一个规范推导:

$$S \xRightarrow{*} \delta A \omega \Rightarrow \delta \alpha \beta \omega.$$

其中  $\omega$  的第一个符号为  $a$ , 或者  $\omega = \varepsilon$ , 且  $a = \$$ 。

- **推广**: 若项目  $[A \rightarrow \alpha B \beta, a]$  对活前缀  $\gamma = \delta\alpha$  是有效的, 并且  $B \rightarrow \eta$  是一个产生式, 则对任何  $b \in \text{FIRST}(\beta a)$ , 项目  $[B \rightarrow \eta, b]$  对活前缀  $\gamma = \delta\alpha$  也是有效的。
- $b$  或是从  $\beta$  推出的开头终结符号, 或者  $\beta \Rightarrow \varepsilon$ , 而  $b = a$ 。
- LR(1)项目  $[S' \rightarrow S, \$]$  对活前缀  $\varepsilon$  是有效的。

## 定义4.15: LR(1)有效项目集 和 LR(1)项目集规范族

- 文法 $G$ 的某个活前缀 $\gamma$ 的所有LR(1)有效项目组成的集合称为 $\gamma$ 的LR(1)有效项目集。
- 文法 $G$ 的所有LR(1)有效项目集组成的集合称为 $G$ 的LR(1)项目集规范族。

## 定义4.16: 闭包 (closure)

设 $I$ 是文法 $G$ 的一个 $LR(1)$ 项目集， $\text{closure}(I)$ 是从 $I$ 出发，用下面的方法构造的项目集

- (1)  $I$ 中的每一个项目都属于 $\text{closure}(I)$ ;
- (2) 若项目 $[A \rightarrow \alpha \cdot B \beta, a]$ 属于 $\text{closure}(I)$ ，且 $B \rightarrow \eta$ 是 $G$ 的一个产生式，则对任何终结符号 $b \in \text{FIRST}(\beta a)$ ，若项目 $[B \rightarrow \cdot \eta, b]$ 不属于集合 $\text{closure}(I)$ ，则将它加入 $\text{closure}(I)$ ;
- (3) 重复规则(2)，直到 $\text{closure}(I)$ 不再增大为止。

## 定义4.17：转移函数go

若 $I$ 是文法 $G$ 的一个 $LR(1)$ 项目集， $X$ 是一个文法符号，定义：

$$go(I, X) = closure(J)$$

其中： $J = \{ [A \rightarrow \alpha X \cdot \beta, a] \mid \text{当} [A \rightarrow \alpha \cdot X \beta, a] \text{属于} I \text{时} \}$

项目 $[A \rightarrow \alpha X \cdot \beta, a]$ 称为 $[A \rightarrow \alpha \cdot X \beta, a]$ 的后继。

直观含义：

若 $I$ 是某个活前缀  $\gamma$  的有效项目集，

则  $go(I, X)$  便是对活前缀  $\gamma X$  的有效项目集。

## 算法4.6 构造文法G的LR(1)项目集规范族

输入：拓广文法G'

输出：G'的LR(1)项目集规范族

方法：

**$C = \{\text{closure}(\{[S' \rightarrow \cdot S, \$]\})\};$**

**do**

**for (C中的每一个项目集I和每一个文法符号X)**

**if (go(I,X)不为空, 且不在C中)**

**把go(I,X)加入C中;**

**while (没有新项目集加入C中).**

# 例：构造如下文法的LR(1)项目集规范族：

(1)  $S \rightarrow CC$       (2)  $C \rightarrow cC$       (3)  $C \rightarrow d$       (文法4.9)

## ■ 拓广文法：

(0)  $S' \rightarrow S$       (1)  $S \rightarrow CC$       (2)  $C \rightarrow cC$       (3)  $C \rightarrow d$

## ■ 根据算法4.6构造其LR(1)项目集规范族。

$$I_0 = \text{closure}(\{[S' \rightarrow S, \$]\})$$

$$= \{[S' \rightarrow S, \$] \quad [S \rightarrow CC, \$] \quad [C \rightarrow cC, c/d] \quad [C \rightarrow d, c/d]\}$$

$$I_1 = \text{go}(I_0, S) = \text{closure}(\{[S' \rightarrow S ; \$]\}) = \{[S' \rightarrow S ; \$]\}$$

$$I_2 = \text{go}(I_0, C) = \text{closure}(\{[S \rightarrow C C, \$]\})$$

$$= \{[S \rightarrow C C, \$] \quad [C \rightarrow cC, \$] \quad [C \rightarrow d, \$]\}$$

$$I_3 = \text{go}(I_0, c) = \text{closure}(\{[C \rightarrow c C, c/d]\})$$

$$= \{[C \rightarrow c C, c/d] \quad [C \rightarrow cC, c/d] \quad [C \rightarrow d, c/d]\}$$

$$I_4 = \text{go}(I_0, d) = \text{closure}(\{[C \rightarrow d ; c/d]\}) = \{[C \rightarrow d ; c/d]\}$$

$$I_5 = \text{go}(I_2, C) = \text{closure}(\{[S \rightarrow CC; \$]\}) = \{[S \rightarrow CC; \$]\}$$

$$I_6 = \text{go}(I_2, c) = \text{closure}(\{[C \rightarrow c \text{ €}, \$]\}) \\ = \{[C \rightarrow c \text{ €}, \$] \quad [C \rightarrow cC, \$] \quad [C \rightarrow d, \$]\}$$

$$I_7 = \text{go}(I_2, d) = \text{closure}(\{[C \rightarrow d; \$]\}) = \{[C \rightarrow d; \$]\}$$

$$I_8 = \text{go}(I_3, C) = \text{closure}(\{[C \rightarrow cC; c/d]\}) = \{[C \rightarrow cC; c/d]\}$$

$$\text{go}(I_3, c) = \text{closure}(\{[C \rightarrow c \text{ €}, c/d]\}) = I_3$$

$$\text{go}(I_3, d) = \text{closure}(\{[C \rightarrow d; c/d]\}) = I_4$$

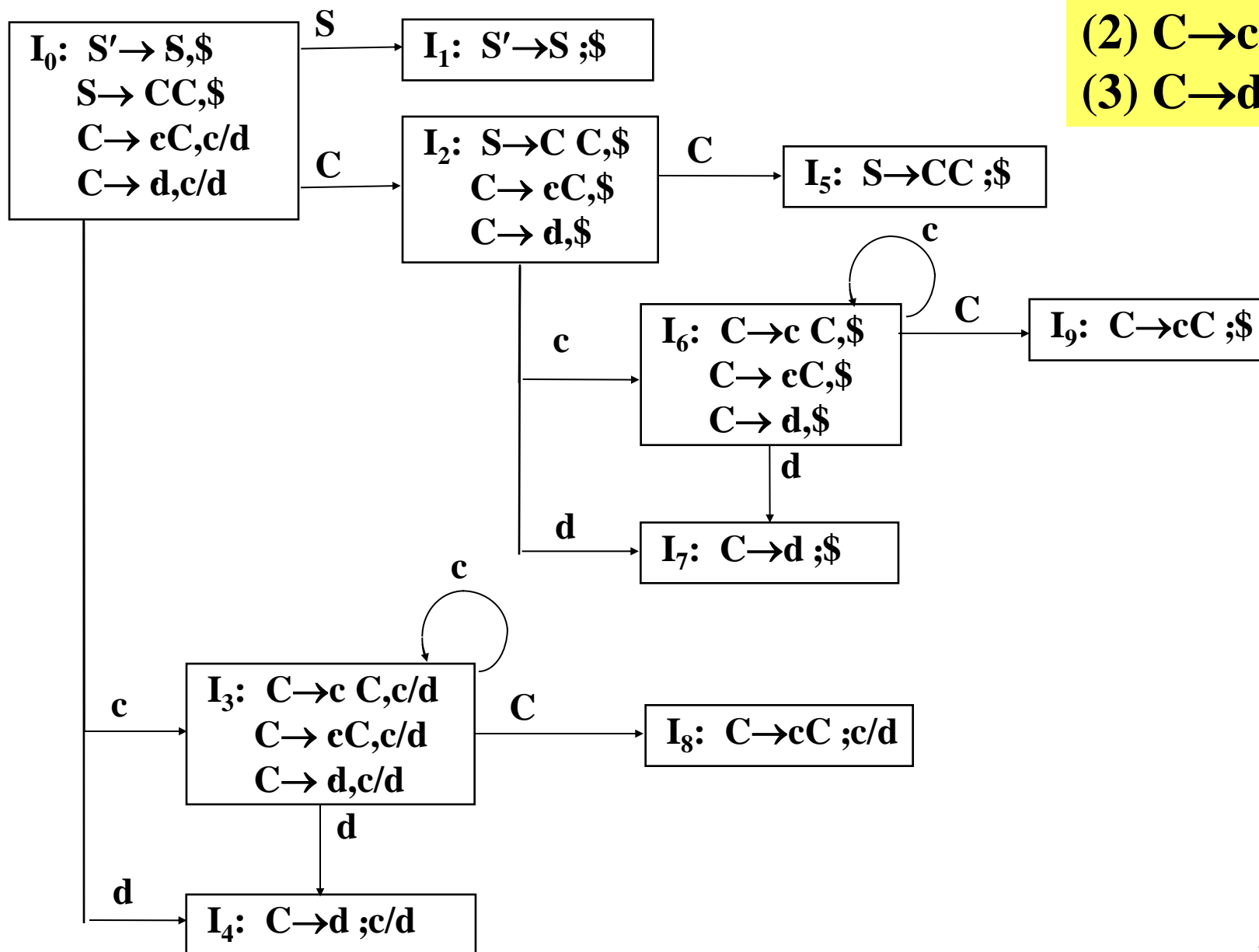
$$I_9 = \text{go}(I_6, C) = \text{closure}(\{[C \rightarrow cC; \$]\}) = \{[C \rightarrow cC; \$]\}$$

$$\text{go}(I_6, c) = \text{closure}(\{[C \rightarrow c \text{ €}, \$]\}) = I_6$$

$$\text{go}(I_6, d) = \text{closure}(\{[C \rightarrow d; \$]\}) = I_7$$

# 识别文法4.9所有活前缀的DFA

- (0)  $S' \rightarrow S$
- (1)  $S \rightarrow CC$
- (2)  $C \rightarrow cC$
- (3)  $C \rightarrow d$





# 算法4.7 构造LR(1)分析表

输入：拓广文法 $G'$  输出：文法 $G'$ 的分析表

方法如下：

- 1.构造文法 $G'$ 的LR(1)项目集规范族 $C=\{I_0, I_1, \dots, I_n\}$
- 2.对于状态 $i$ （代表项目集 $I_i$ ），分析动作如下：
  - a) 若 $[A \rightarrow \alpha \cdot a \beta, b] \in I_i$ ，且 $go(I_i, a) = I_j$ ，则置 $action[i, a] = sj$
  - b) 若 $[A \rightarrow \alpha \cdot, a] \in I_i$ ，且 $A \neq S'$ ，则置 $action[i, a] = rj$
  - c) 若 $[S' \rightarrow S \cdot, \$] \in I_i$ ，则置 $action[i, \$] = acc$
- 3.若对非终结符号 $A$ ，有 $go(I_i, A) = I_j$ ，则置 $goto[i, A] = j$
- 4.凡是不能用上述规则填入信息的空白表项，均置上出错标志 $error$ 。
- 5.分析程序的初态是包括 $[S' \rightarrow \cdot s, \$]$ 的有效项目集所对应的状态。

# 例：构造文法4.9的LR(1)分析表

## ■ 考察 $I_0$ :

- ◆  $[S' \rightarrow S, \$]$  且  $go(I_0, S) = I_1$ , 故:  $goto[0, S] = 1$
- ◆  $[S \rightarrow \epsilon C, \$]$   $go(I_0, C) = I_2$   $goto[0, C] = 2$
- ◆  $[C \rightarrow \epsilon C, c/d]$   $go(I_0, c) = I_3$   $action[0, c] = s3$
- ◆  $[C \rightarrow d, c/d]$   $go(I_0, d) = I_4$   $action[0, d] = s4$

## ■ 考察 $I_1$ : 由于 $[S' \rightarrow S; \$]$ 故 $action[1, \$] = acc$

## ■ 考察 $I_2$ :

- ◆  $[S \rightarrow C \epsilon, \$]$  且  $go(I_2, C) = I_5$  故:  $goto[2, C] = 5$
- ◆  $[C \rightarrow \epsilon C, \$]$   $go(I_2, c) = I_6$   $action[2, c] = s6$
- ◆  $[C \rightarrow d, \$]$   $go(I_2, d) = I_7$   $action[2, d] = s7$

## ■ 考察 $I_4$ :

- ◆  $[C \rightarrow d; c/d]$  故  $action[4, c] = action[4, d] = r3$

# 文法4.9的LR(1)分析表

状态	action			goto	
	c	d	\$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

**例：试构造文法4.8的LR(1)分析表**

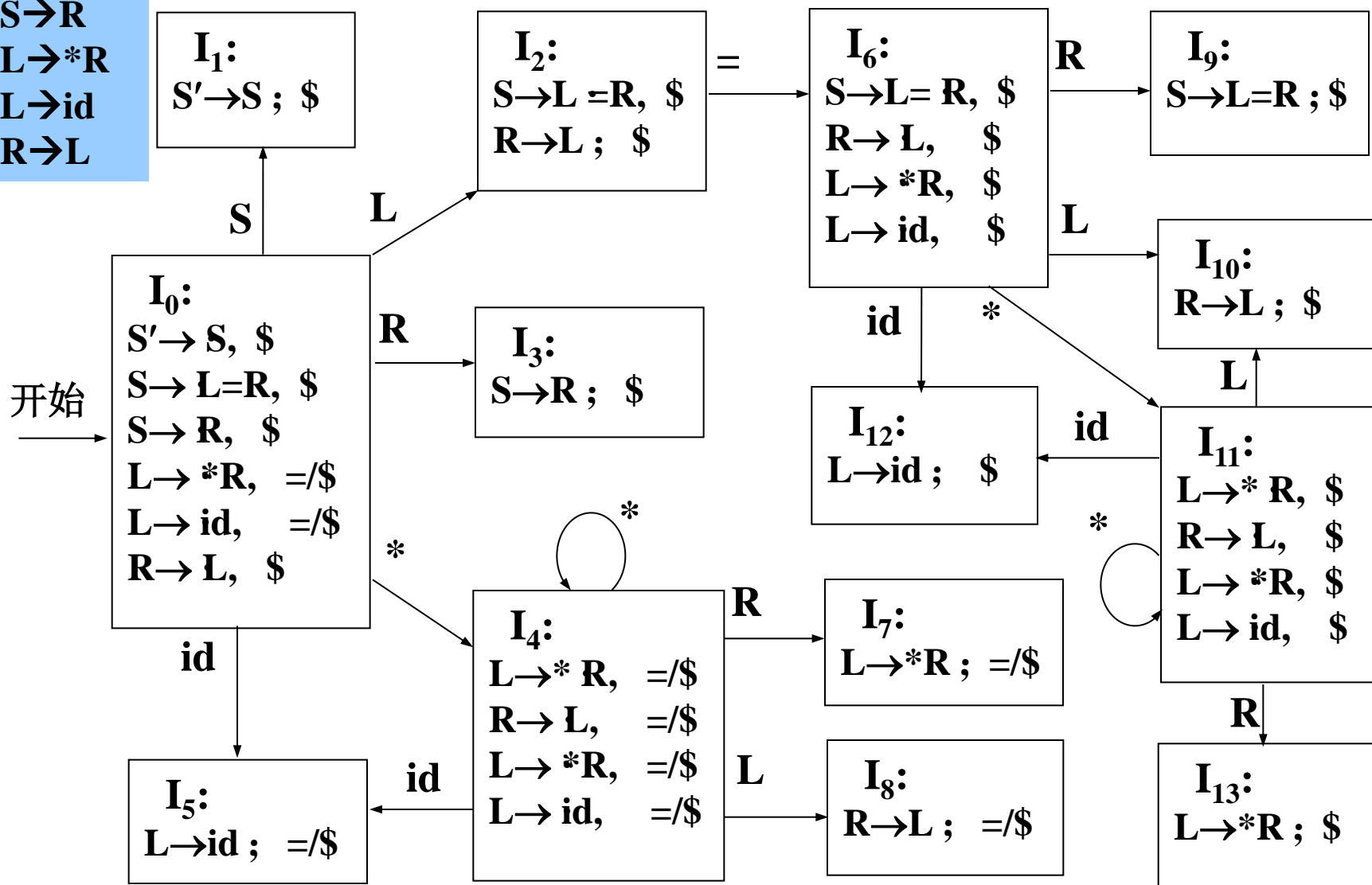
**(1)  $S \rightarrow L = R$     (2)  $S \rightarrow R$     (3)  $L \rightarrow *R$   
(4)  $L \rightarrow id$         (5)  $R \rightarrow L$**

■ **拓广文法 $G'$**

**(0)  $S' \rightarrow S$     (1)  $S \rightarrow L = R$     (2)  $S \rightarrow R$   
(3)  $L \rightarrow *R$     (4)  $L \rightarrow id$         (5)  $R \rightarrow L$**

■ **构造文法 $G'$ 的LR(1)项目集规范族及识别所有活前缀的DFA**

- 0)  $S' \rightarrow S$
- 1)  $S \rightarrow L=R$
- 2)  $S \rightarrow R$
- 3)  $L \rightarrow *R$
- 4)  $L \rightarrow id$
- 5)  $R \rightarrow L$



分析  $*id=id$

# 文法4. 8的LR(1) 分析表

状态	action				goto		
	=	*	id	\$	S	L	R
0		s4	s5		1	2	3
1				acc			
2	s6			r5			
3				r2			
4		s4	s5			8	
5	r4			r4			
6		s11	s12			10	9
7	r3			r3			
8	r5			r5			
9				r1			
10				r5			
11		s11	s12			10	13
12				r4			
13				r3			

# 五、LALR(1)分析表的构造

## ■描述LR(1)项目集特征的两个定义

**定义4.18** 如果两个LR(1)项目集去掉搜索符号之后是相同的，则称这两个项目集**具有相同的心**（**core**），即这两个项目集是**同心集**。

**定义4.19** 除去初态项目集外，一个**项目集的核**（**kernel**）是由该项目集中那些圆点不在最左边的项目组成。LR(1)初态项目集的核中有且只有项目  $[S' \rightarrow \cdot S, \$]$ 。

# 构造LALR(1)分析表的基本思想

- 合并LR(1)项目集规范族中的同心集，以减少分析表的状态数。
- 用核代替项目集，以减少项目集所需的存储空间
- $go(I, X)$ 仅仅依赖于I的心，因此LR(1)项目集合并后的转移函数可以通过 $go(I, X)$ 自身的合并得到。
- 同心集的合并，可能导致归约-归约的冲突，但不会产生新的移进-归约冲突



# 同心集的合并不会引进新的移进-归约冲突

如果合并后的项目集中存在移进-归约冲突，则意味着：

项目  $[A \rightarrow \alpha \cdot, a]$  和  $[B \rightarrow \beta \cdot a \gamma, b]$  处于合并后的同一项目集中

合并前必存在某个  $c$ ，使得  $[A \rightarrow \alpha \cdot, a]$  和  $[B \rightarrow \beta \cdot a \gamma, c]$  同处于某个项目集中。

说明，原来的  $LR(1)$  项目集中已经存在移进-归约冲突。

# 同心集的合并可能导致归约-归约冲突

例：有文法**G**：

**$S \rightarrow aAd \mid bBd \mid aBe \mid bAe$**

**$A \rightarrow c$**

**$B \rightarrow c$**             (文法4.10)

## ■ 拓广文法**G'**

(0)  **$S' \rightarrow S$**

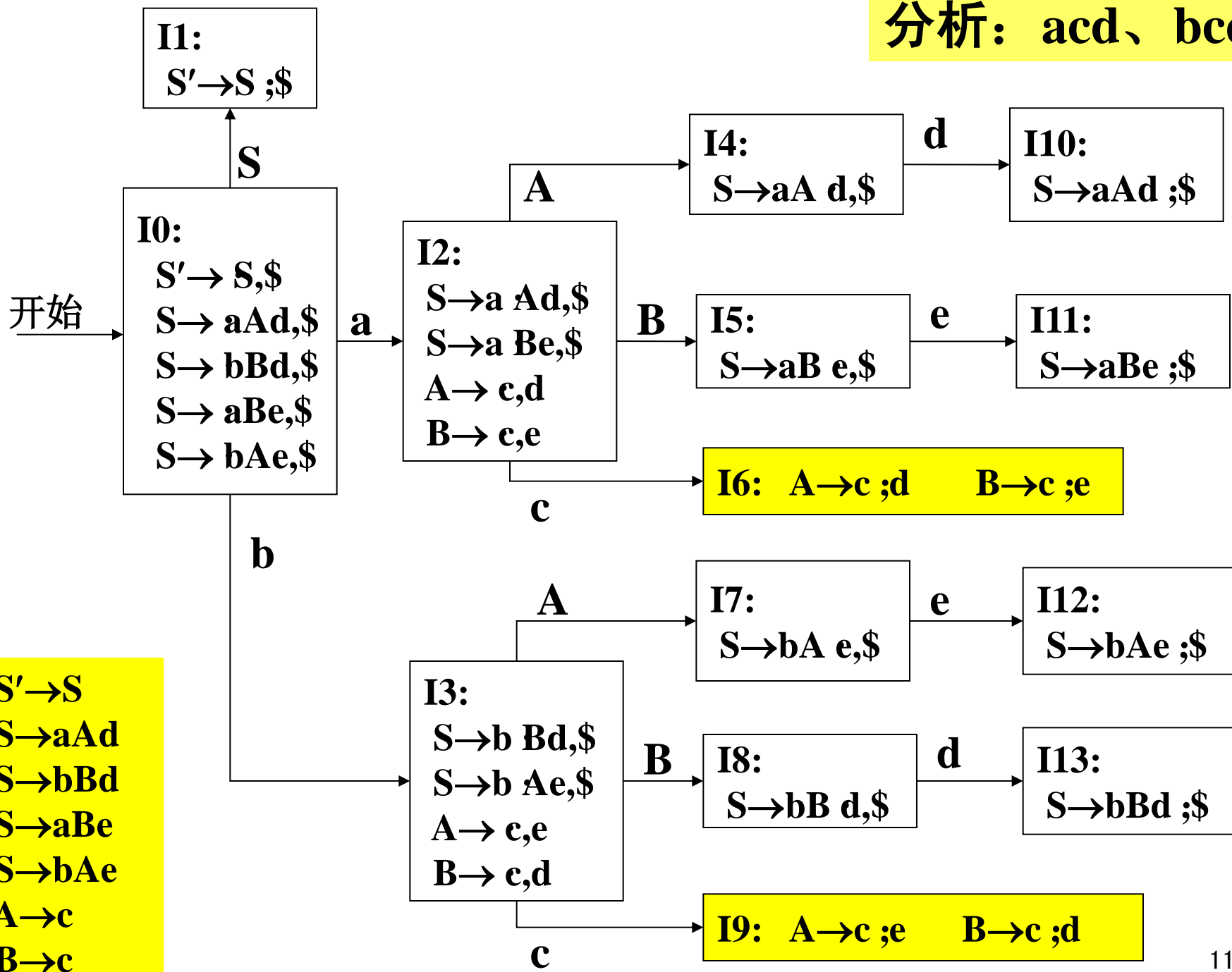
(1)  **$S \rightarrow aAd$**    (2)  **$S \rightarrow bBd$**    (3)  **$S \rightarrow aBe$**    (4)  **$S \rightarrow bAe$**

(5)  **$A \rightarrow c$**         (6)  **$B \rightarrow c$**

## ■ 构造识别文法**G'**的所有活前缀的DFA

$I_0 = \{ [S' \rightarrow S, \$]$

$[S \rightarrow aAd, \$] \quad [S \rightarrow bBd, \$] \quad [S \rightarrow aBe, \$] \quad [S \rightarrow bAe, \$] \}$



# 文法4.10的LR(1)分析表

状态	action						goto		
	a	b	c	d	e	\$	S	A	B
0	s2	s3					1		
1						acc			
2			s6					4	5
3			s9					7	8
4				s10					
5					s11				
6				r5	r6				
7					s12				
8				s13					
9				r6	r5				
10						r1			
11						r3			
12						r4			
13						r2			

# 考察同心集

## ■ LR(1)项目集规范族中

- ◆ 活前缀ac的有效项目集是 $I_6$
- ◆ 活前缀bc的有效项目集是 $I_9$
- ◆ 这两个项目集都不含冲突项目， 且是同心集。

## ■ 它们合并后得到的集合为：

$\{[A \rightarrow c\cdot, d/e] [B \rightarrow c\cdot, d/e]\}$

## ■ 含有归约-归约冲突。

# LALR(1)分析表的构造

## ■ 基本思想是：

- ◆ 首先构造LR(1)项目集规范族；
- ◆ 如果它不存在冲突，就把同心集合并在一起；  
如果LR(1)项目集规范族中含有冲突，  
则该文法不是LR(1)文法，  
不能为它构造LALR分析程序。
- ◆ 若合并后的项目集规范族不存在归约-归约冲突，  
就按这个项目集规范族构造分析表。  
如果合并后得到的项目集规范族中含有冲突，  
则该文法是LR(1)文法，但不是LALR文法，  
因而，也不能为它构造LALR分析程序。

## 算法4.8 构造LALR(1)分析表

输入：一个拓广文法 $G'$

输出：文法 $G'$ 的LALR(1)分析表

方法：

1.构造文法 $G'$ 的LR(1)项目集规范族

$$C = \{I_0, I_1, \dots, I_n\}.$$

2.合并 $C$ 中的同心集，得到一个新的项目集规范族  
 $C' = \{J_0, J_1, \dots, J_m\}$ ，其中含有项目 $[S' \rightarrow \cdot S, \$]$ 的 $J_k$ 为分析表的初态。

3.从 $C'$ 出发，构造action子表

a) 若 $[A \rightarrow \alpha \cdot a \beta, b] \in J_i$ ，且 $\text{go}(J_i, a) = J_j$ ，则  
置 $\text{action}[i, a] = sj$

b) 若 $[A \rightarrow \alpha \cdot, a] \in J_i$ ，则置 $\text{action}[i, a] = r \ A \rightarrow \alpha$

c) 若 $[S' \rightarrow S \cdot, \$] \in J_i$ ，则置 $\text{action}[i, \$] = \text{acc}$

## 4.构造goto子表

设 $J_k = \{I_{i1}, I_{i2}, \dots, I_{it}\}$

由于这些 $I_i$ 是同心集，因此

$go(I_{i1}, X)$ 、 $go(I_{i2}, X)$ 、...、 $go(I_{it}, X)$ 也是同心集

把所有这些项目集合并后得到的集合记作 $J_i$ ，则有：

$$go(J_k, X) = J_i$$

于是，若 $go(J_k, A) = J_i$ ，则置 $goto[k, A] = I$

5.分析表中凡不能用上述规则填入信息的空表项，均置上出错标志。



# 例：构造文法4.9的LALR(1)分析表。

- 有三对同心集可以合并，即

- ◆  $I_3$ 和 $I_6$ 合并，得到项目集：

$$I_{36} = \{[C \rightarrow c \cdot, c/d/\$] \quad [C \rightarrow cC, c/d/\$] \quad [C \rightarrow d, c/d/\$]\}$$

- ◆  $I_4$ 和 $I_7$ 合并，得到项目集：

$$I_{47} = \{[C \rightarrow d \cdot, c/d/\$]\}$$

- ◆  $I_8$ 和 $I_9$ 合并，得到项目集：

$$I_{89} = \{[C \rightarrow cC \cdot, c/d/\$]\}$$

- 同心集合并后得到的新的项目集规范族为：

$$C' = \{I_0, I_1, I_2, I_{36}, I_{47}, I_5, I_{89}\}$$

- 利用算法4.8，可以为该文法构造LALR(1)分析表。

# 文法4. 9的LALR(1) 分析表

状态	action			goto	
	c	d	\$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

- 此文法是LALR(1)文法
- 转移函数go的计算只依赖于项目集的心

# LALR(1) 分析程序和LR(1) 分析程序的比较

## ■ LALR(1) 分析程序对符号串cdcd的分析过程

步骤	栈	输入	分析动作
0	0	cdcd\$	shift
1	0c <sup>36</sup>	dcd\$	shift
2	0c <sup>36</sup> d <sup>47</sup>	cd\$	reduce by $C \rightarrow d$
3	0c <sup>36</sup> C <sup>89</sup>	cd\$	reduce by $C \rightarrow cC$
4	0C <sup>2</sup>	cd\$	shift
5	0C <sup>2</sup> c <sup>36</sup>	d\$	shift
6	0C <sup>2</sup> c <sup>36</sup> d <sup>47</sup>	\$	reduce by $C \rightarrow d$
7	0C <sup>2</sup> c <sup>36</sup> C <sup>89</sup>	\$	reduce by $C \rightarrow cC$
8	0C <sup>2</sup> C <sup>5</sup>	\$	reduce by $S \rightarrow CC$
9	0S <sup>1</sup>	\$	accept

## ■ LR(1) 分析程序对符号串cdcd的分析过程

步骤	栈	输入	分析动作
0	0	cdcd\$	shift
1	0c <sup>3</sup>	dcd\$	shift
2	0c <sup>3</sup> d <sup>4</sup>	cd\$	reduce by $C \rightarrow d$
3	0c <sup>3</sup> C <sup>8</sup>	cd\$	reduce by $C \rightarrow cC$
4	0C <sup>2</sup>	cd\$	shift
5	0C <sup>2</sup> c <sup>6</sup>	d\$	shift
6	0C <sup>2</sup> c <sup>6</sup> d <sup>7</sup>	\$	reduce by $C \rightarrow d$
7	0C <sup>2</sup> c <sup>6</sup> C <sup>9</sup>	\$	reduce by $C \rightarrow cC$
8	0C <sup>2</sup> C <sup>5</sup>	\$	reduce by $S \rightarrow CC$
9	0S <sup>1</sup>	\$	accept

## ■ LR(1) 分析程序对符号串ccd的分析过程

步骤	栈	输入	分析动作
0	0	ccd\$	shift
1	0c3	cd\$	shift
2	0c3c3	d\$	shift
3	0c3c3d4	\$	error

## ■ LALR(1) 分析程序对符号串ccd的分析过程

步骤	栈	输入	分析动作
0	0	ccd\$	shift
1	0c36	cd\$	shift
2	0c36c36	d\$	shift
3	0c36c36d47	\$	reduce by $C \rightarrow d$
4	0c36c36C89	\$	reduce by $C \rightarrow cC$
5	0c36C89	\$	reduce by $C \rightarrow cC$
6	0C2	\$	error

# 文法分类

非二义性文法

二义性文法

作业:  
4.9  
4.14  
4.16

**LL(k)**

**LR(k)**

**LL(1)**

**LR(1)**

**LALR(1)**

**SLR**

**LL(0)**

**LR(0)**

## 六、LR分析方法对二义文法的应用

- **定理：任何二义性文法决不是LR文法，因而也不是SLR或LALR文法。**
- **程序设计语言的某些结构用二义性文法描述比较直观，使用方便。例如关于算术表达式的文法和if语句的文法。**
- **在所有情况下，都说明了消除二义性的一些规则（即这类结构的使用限制）。**

# 利用优先级和结合规则解决表达式冲突

- 描述算术表达式集合的二义性文法:

$E \rightarrow E + E \mid E * E \mid (E) \mid id$  (文法4.11)

- 无二义性的文法:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

- 前者具有两个明显的优点:

- ◆ 改变运算符的优先级或结合规则时, 文法本身无需改变, 只需改变限制条件。
- ◆ LR分析表所包含的状态数比后者少得多



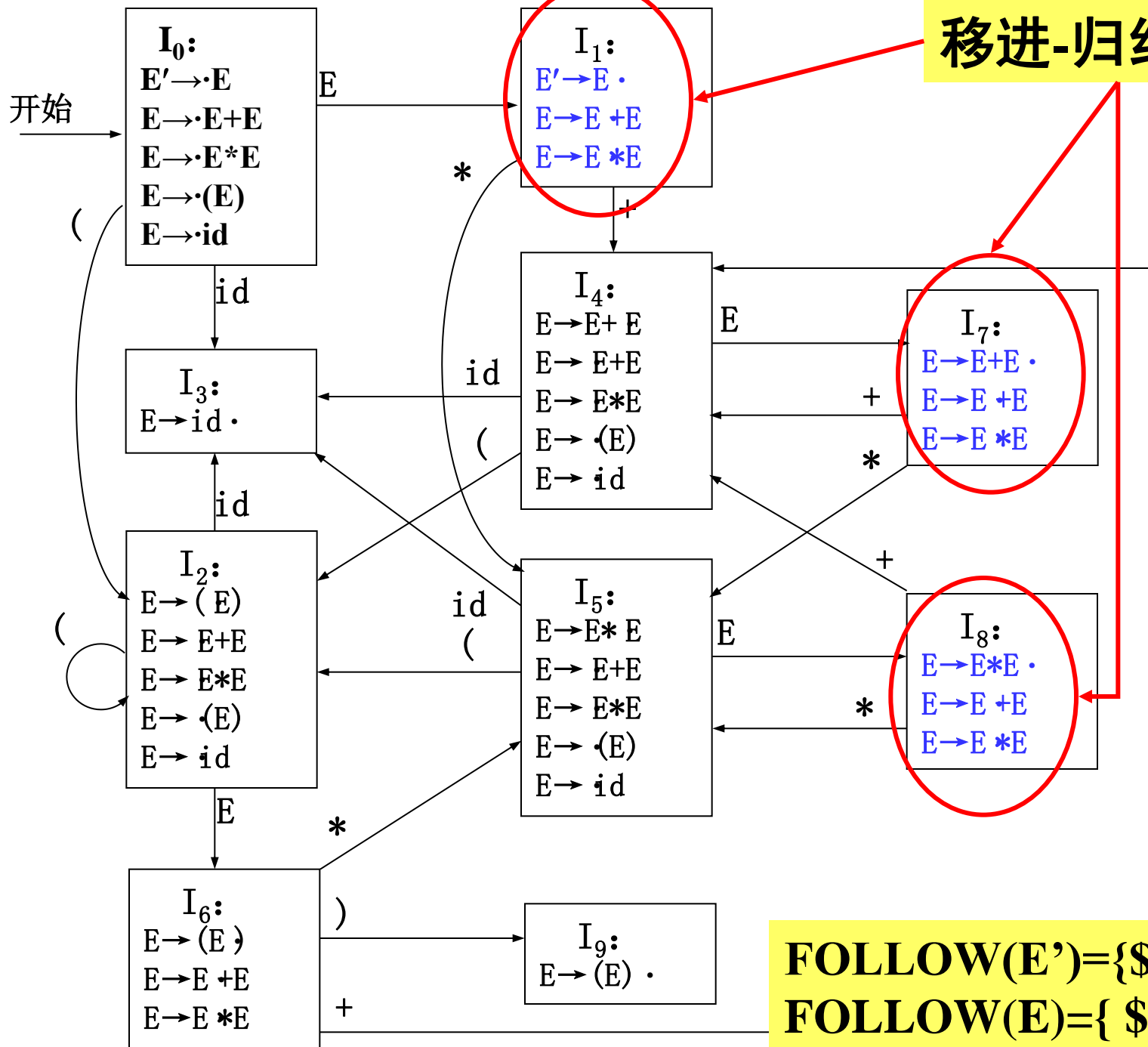
# 例：构造文法4.11的LR分析表。

- 其拓广文法**G'**具有产生式：

(0)  $E' \rightarrow E$       (1)  $E \rightarrow E + E$       (2)  $E \rightarrow E * E$   
(3)  $E \rightarrow (E)$       (4)  $E \rightarrow id$

- 构造文法**G'**的**LR(0)**项目集规范族及识别所有活前缀的**DFA**

# 移进-归约冲突



# ‘+’和 ‘\*’ 的优先级及结合规则共有4种情况

- (1) \* 优先于 + ， 遵从左结合规则
- (2) \* 优先于 + ， 遵从右结合规则
- (3) + 优先于 \* ， 遵从左结合规则
- (4) + 优先于 \* ， 遵从右结合规则

$I_7:$ $E \rightarrow E + E \cdot$ $E \rightarrow E + E$ $E \rightarrow E * E$	$I_8:$ $E \rightarrow E * E \cdot$ $E \rightarrow E + E$ $E \rightarrow E * E$
---	---

条件	状态	action					goto E
		id	+	*	(	)	\$
(1)	7		r1	s5		r1	r1
	8		r2	r2		r2	r2
(2)	7		s4	s5		r1	r1
	8		r2	s5		r2	r2
(3)	7		r1	r1		r1	r1
	8		s4	r2		r2	r2
(4)	7		s4	r1		r1	r1
	8		s4	s5		r2	r2

# (1) 文法4.11的LR分析表

状态	action						goto
	id	+	*	(	)	\$	E
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

# 利用最近最后匹配原则解决if语句冲突

## ■ 映射程序设计语言中if-then-else结构的文法:

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

|  $\text{if } E \text{ then } S$

|  $\text{others}$

(文法4.12)

## ■ 对该文法进行抽象

◆ 用i表示 “if E then”

◆ 用e表示 “else”

◆ 用a表示 “others”

## ■ 得到文法:

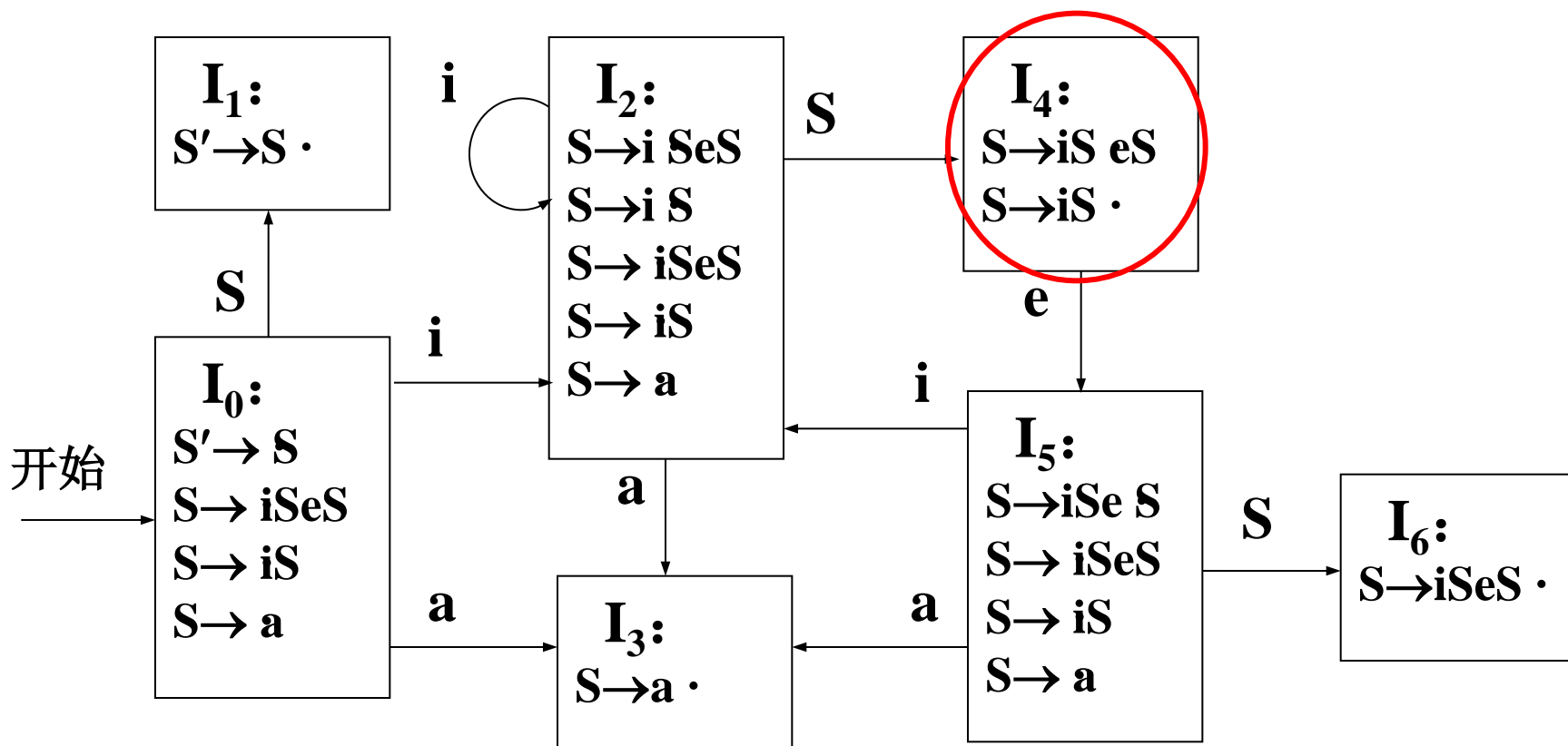
$S \rightarrow iS \mid iSeS \mid a$

(文法4.13)

其拓广文法 $G'$ 为:

(0)  $S' \rightarrow S$     (1)  $S \rightarrow iSeS$     (2)  $S \rightarrow iS$     (3)  $S \rightarrow a$

文法4.14的LR(0)项目集规范族及识别所有活前缀的DFA:



**FOLLOW(S) = { \$, e }**

## 最近最后匹配原则:

else与离它最近的一个未匹配的then相匹配

### 文法4.14的LR分析表

状态	action				goto
	i	e	a	\$	S
0	s2		s3		1
1				acc	
2	s2		s3		4
3		r3		r3	
4		s5		r2	
5	s2		s3		6
6		r1		r1	

# 例：分析输入符号串iaea

步骤	栈	输入	分析动作
1	0	iaea\$	shift
2	0i2	iaea\$	shift
3	0i2i2	aea\$	shift
4	0i2i2a3	ea\$	reduce by $S \rightarrow a$
5	0i2i2S4	ea\$	shift
6	0i2i2S4e5	a\$	shift
7	0i2i2S4e5a3	\$	reduce by $S \rightarrow a$
8	0i2i2S4e5S6	\$	reduce by $S \rightarrow iSeS$
9	0i2S4	\$	reduce by $S \rightarrow iS$
10	0S1	\$	accept



# 七、LR分析的错误处理与恢复

## ■ LR分析程序可采取以下恢复策略：

- ◆ 首先，从栈顶开始退栈，可能弹出0个或若干个状态，直到出现状态 $S$ 为止，根据 $S$ 在 $goto$ 子表中可以找到一个非终结符号 $A$ ，即它有关于 $A$ 的转移；
  - ◆ 然后，抛弃0个或若干个输入符号，直到找到符号 $a$ 为止， $a \in FOLLOW(A)$ ，即 $a$ 可以合法地跟在 $A$ 的后面；
  - ◆ 然后，分析程序把状态 $goto[S, A]$ 压入栈，继续进行语法分析。
- 在弹栈过程中出现的 $A$ 可能不止一个，通常选择表示主要结构成分的非终结符号。
- 实际上是跳过包含错误的一部分终结符号串。

# 例：运算符

期待输入符号为运算符或右括号，而遇到的却是运算对象（id或左括号）。

诊断信息：“缺少运算符”

恢复策略：把运算符 ‘+’ 压入栈，转移到状态4。

即遇到

状态							goto
	id	(	)	\$	E		
0	s3	e1	e1	s2	e2	e1	1
1	e3	s4	s5	e3	e2	acc	
2	s3	e1	e1	s2	e2	e1	6
3	r4	r4	r4	r4	r4	r4	
4	s3	e1	e1	s2	e2	e1	7
5	s3	e1	e1	s2	e2	e1	8
6	e3	s4	s5	e3	s9	e4	
7	r1	r1	s5	r1	r1		
8							
9							

期待输入符号为运算符或右括号，而遇到的却是输入串结束标志 ‘\$’。

诊断信息：“缺少右括号”

恢复策略：把右括号压入栈，转移到状态9。

**e1:** 在状态**0**、**2**、**4**、**5**，期待输入符号为运算对象的首字符，即**id**或**(**，而输入中出现的却是运算符号‘**+**’或‘**\***’，或是输入串结束标志‘**\$**’。

- ◆ 策略：把一个假想的**id**压入栈，并将状态**3**推入栈顶。

- ◆ 诊断信息：“缺少运算对象”

**e2:** 在状态**0**、**1**、**2**、**4**、**5**，期待输入符号为运算对象的首字符或运算符号，但却遇到右括号。

- ◆ 策略：删掉输入的右括号

- ◆ 诊断信息：“括号不匹配”

**e3:** 在状态**1**、**6**，期待输入符号为运算符号或右括号，而遇到的却是运算对象（**id**或左括号）。

- ◆ 策略：把运算符号‘**+**’压入栈，转移到状态**4**。

- ◆ 诊断信息：“缺少运算符号”

**e4:** 在状态**6**，期待输入符号为运算符号或右括号，而遇到的却是输入串结束标志‘**\$**’。

- ◆ 策略：把右括号压入栈，转移到状态**9**。

- ◆ 诊断信息：“缺少右括号”

## 例：分析符号串id+)

栈	输入	分析动作
0	id+)\$	shift
0id3	+) \$	reduce by $E \rightarrow id$
0E1	+) \$	shift
0E1+4	) \$	CALL e2 “括号不匹配”，删掉 ‘)’
0E1+4	\$	CALL e1 “缺少运算对象”，id压入栈
0E1+4id3	\$	reduce by $E \rightarrow id$
0E1+4E7	\$	reduce by $E \rightarrow E+E$
0E1	\$	accept

# 小结

## 一、自顶向下的分析方法

### ■ 递归下降分析方法

- ◆ 试探性、回溯
- ◆ 要求：文法不含左递归

### ■ 递归调用预测分析方法

- ◆ 不带回溯的递归分析方法
- ◆ 要求：文法不含左递归

对任何产生式： $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$

$$\mathbf{FIRST}(\alpha_i) \cap \mathbf{FIRST}(\alpha_j) = \emptyset$$

- ◆ 构造步骤：描述结构的上下文无关文法

根据文法构造预测分析程序的状态转换图

状态转换图化简

根据状态转换图构造递归过程

## ■ 非递归预测分析方法

- ◆ 不带回溯、不含递归
- ◆ 模型：

输入缓冲区：存放输入符号串 $a_1a_2\cdots a_n\$$

符号栈：分析过程中存放文法符号

分析表：二维表，每个 $A$ 有一行，每个 $a$ 包括 $\$$ 有一列

表项内容是产生式（关键）

控制程序：根据栈顶 $X$ 和当前输入 $a$ 决定分析动作（永恒的核心）

$X=a=\$$  分析成功

$X=a\neq \$$  弹出 $X$ ，扫描指针前移

$X$ 是非终结符号，查分析表： $M[X,a]$

$M[X,a]=X\rightarrow Y_1Y_2\cdots Y_K$ ，弹出 $X$ ， $Y_K$ 、 $\cdots$ 、 $Y_2$ 、 $Y_1$ 入栈

$M[X,a]=X\rightarrow\epsilon$ ，弹出 $X$

$M[X,a]=$ 空白，出错处理

输出：对输入符号串进行最左推导所用的产生式序列

## ■ 预测分析表的构造

构造每个文法符号的**FIRST**集合

构造每个非终结符号的**FOLLOW**集合

检查每个产生式 $A \rightarrow \alpha$

对任何 $a \in \text{FIRST}(\alpha)$ ,  $M[A, a] = A \rightarrow \alpha$

若 $\alpha \Rightarrow \varepsilon$ , 对所有 $b \in \text{FOLLOW}(A)$ ,  $M[A, b] = A \rightarrow \alpha$

## ■ LL(1)文法

◆ LL(1)的含义

◆ 判断一个文法是否为LL(1) 文法

构造分析表, 或者

检查每个产生式:  $A \rightarrow \alpha \mid \beta$

$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \phi$

若 $\beta \Rightarrow \varepsilon$ , 则 $\text{FIRST}(\alpha) \cap \text{FOLLOW}(A) = \phi$

## 二、自底向上分析方法

### ■ 移进-归约分析方法

- ◆ 分析栈、输入缓冲区
- ◆ 可归约串
- ◆ 规范归约：最右推导的逆过程

### ■ LR分析方法

- ◆ 模型：

输入缓冲器：

输出：分析动作序列

分析栈： $S_0X_1S_1...X_nS_n$

分析表：包括action和goto两部分（关键）

控制程序：根据栈顶状态 $S_n$ 和当前输入符号 $a$ 查分析表

$action[S_n, a]$ ，决定分析动作（永恒的核心）

$action[S_n, a]=s$   $i$ ， $a$ 入栈， $i$ 入栈  $i=goto(S_n, a)$

$action[S_n, a]=r$   $A \rightarrow \beta$ ，弹出 $2*|\beta|$ 个符号，  
 $A$ 入栈， $goto(S_{n-r}, A)$ 入栈

$action[S_n, a]=acc$ ，分析成功

$action[S_n, a]=$ 空白，出错处理



## SLR(1)分析表的构造

- LR(0)项目集规范族
- 识别文法所有活前缀的DFA
- 构造分析表：检查每个状态集

若 $A \rightarrow \alpha \cdot a \beta \in I_i$ ，且 $go(I_i, a) = I_j$ ，则置 $action[i, a] = sj$ ，

若 $A \rightarrow \alpha \cdot \in I_i$ ，则对所有 $a \in FOLLOW(A)$ ，置 $action[i, a] = r A \rightarrow \alpha$

若 $S' \rightarrow S \cdot \in I_i$ ，则置 $action[i, \$] = acc$ ，表示分析成功。

若 $go(I_i, A) = I_j$ ， $A$ 为非终结符号，则置 $goto[i, A] = j$

## LR(1)分析表的构造

- ◆ LR(1)项目集规范族
- ◆ 构造分析表：检查每个状态集

若 $[A \rightarrow \alpha \cdot a \beta, b] \in I_i$ ，且 $go(I_i, a) = I_j$ ，则置 $action[i, a] = sj$ ，

若 $[A \rightarrow \alpha \cdot, a] \in I_i$ ，则置 $action[i, a] = r A \rightarrow \alpha$ ，

若 $[S' \rightarrow S \cdot, \$] \in I_i$ ，则置 $action[i, \$] = acc$ ，表示分析成功。

若 $go(I_i, A) = I_j$ ， $A$ 为非终结符号，则置 $goto[i, A] = j$

# LALR(1)分析表的构造

- ◆ LR(1)项目集规范族，若没有冲突，继续
- ◆ 合并同心集，若没有冲突，则为LALR(1)项目集规范族
- ◆ 构造分析表，方法同LR(1)分析表的构造方法

## 利用LR技术处理二义文法

- ◆ 利用运算符的优先级和结合性质，处理算术表达式文法
- ◆ 利用最近最后匹配原则，处理IF语句文法。

## YACC的使用

- ◆ YACC说明文件
- ◆ 冲突的缺省处理原则
- ◆ 二义性处理机制

# 算法清单

算法4.1 非递归的预测分析方法

算法4.2 预测分析表的构造方法

算法4.3 LR分析算法

算法4.4 构造文法G的LR(0)项目集规范族

算法4.5 构造SLR分析表

算法4.6 构造文法G的LR(1)项目集规范族

算法4.7 构造LR(1)分析表

算法4.8 构造LALR(1)分析表

给定文法，构造 候选式/非终结符号 的 **FIRST** 集合  
构造 非终结符号 的 **FOLLOW** 集合

# 作业

- 4. 1

- 4. 3

- 4. 5

- 4. 9

- 4. 14

- 4. 16

- 程序设计2

- ◆ 方法2     (必做)

- ◆ 方法3     (必做)

# 课堂练习1

已知文法 $G[A]$ 为：

$$A \rightarrow aABe \mid a$$

$$B \rightarrow Bb \mid d$$

- (1) 试给出与 $G[A]$ 等价的LL(1)文法 $G'[A]$
- (2) 构造 $G'[A]$ 的预测分析表
- (3) 给出输入串aade的分析过程。

# 课堂练习2

有如下文法G[A]:

$$A \rightarrow BA \mid a$$
$$B \rightarrow aB \mid b$$

- (1) 判断该文法是以下哪些类型的文法，要求给出判断过程。  
LL(1)、LR(0)、SLR(1)
- (2) 构造该文法的LR(1)项目集规范族及识别其所有活前缀的DFA。
- (3) 构造该文法的LR(1)分析表
- (4) 给出对输入符号串abb的分析过程。

# 第4章 语法分析——YACC



*LI Wensheng, SCST, BUPT*

**YACC 使用方法**

**YACC 源程序**

# 内容安排

- 一、 **YACC**说明文件
- 二、 用**YACC**处理二义文法
- 三、 用**LEX**建立**YACC**的词法分析程序
- 四、 **YACC**内部名称
- 五、 **YACC**使用方式



# 一、YACC说明文件

- 由说明部分、翻译规则部分和辅助过程三部分组成，各部分之间用双百分号分隔。

## 说明部分：有任选的两节

- 第一节是处于`%{`和`%}`之间的部分，在这里是一些普通的C语言的声明。
- 第二节是文法记号的声明，一般
  - 以 `%start S` 的形式说明文法的开始符号
  - 用 `%token IF、DO、...、ID、...` 的形式说明记号。
  - 记号被YACC赋予了不会与任何字符值冲突的数字值

# 翻译规则部分

每条规则由一个产生式和有关的语义动作组成

- 产生式  $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$ ，在YACC说明文件中写成  
     $A : \alpha_1 \{ \text{语义动作1} \}$   
         $| \alpha_2 \{ \text{语义动作2} \}$   
         $\dots$   
         $| \alpha_n \{ \text{语义动作n} \}$   
    ;  
■ 用单引号括起来的单个字符，如 ‘c’，是由终结符号c组成的记号  
■ 没有用引号括起来、也没有被说明成token类型的字母数字串是非终结符号  
■ 语义动作是用C语言描述的语句序列
  - ‘\$\$’ 表示和产生式左部非终结符号相关的属性值，‘\$i’ 表示和产生式右部第i个文法符号相关的属性值。

# 辅助过程部分

用C语言书写一些语义动作中用到的辅助程序

- 名字为**yylex()**的词法分析程序必须提供

- 函数**main**

```
main()
{
    return yyparse();
}
```

- YACC生成的**yyparse**过程调用一个扫描过程**yylex**
- **yyparse** 每次调用 **yylex()** 时，得到一个二元式的记号：  
<记号，属性值>。
  - 返回的**记号**必须事先在YACC说明文件的第一部分中用**%token**说明
  - **属性值**必须通过YACC定义的变量**yyval**传给分析程序

## 二、YACC对二义文法的处理

### ■ 处理冲突的两条缺省规则

- “**归约-归约**”冲突，选择排在前面的产生式进行归约
- “**移进-归约**”冲突，选择执行移进动作

### ■ 处理移进-归约冲突的机制

- 利用 **%left** ‘+’ ‘-’ 说明 ‘+’ 和 ‘-’ 具有同样的优先级，并且遵从左结合规则。
- 利用 **%right** ‘↑’ 声名算符 ‘↑’ 遵从右结合规则。
- 利用 **%nonassoc** ‘<’ 说明某些二元运算符不具有结合性。
- 先出现的记号的优先级低
- 同一声明中的记号具有相同的优先级

## -产生式的优先级

- 和它最右边的终结符号的优先级一致。
- 最右终结符号不能给产生式以适当的优先级
  - 通过给产生式附加标记 **%prec <terminal>** 来限制它的优先级
  - 它的优先级和结合性质同这个指定的终结符号的一样
  - 这个终结符号可以是一个占位符，不是由词法分析程序返回的记号，仅用来决定一个产生式的优先级。
- **YACC**不报告用这种优先级和结合性质能够解决的移进-归约冲突。

### 三、用LEX建立YACC的词法分析程序

- LEX编译器将提供词法分析程序**yylex()**
- 如果用LEX产生词法分析程序，则YACC说明文件中第三部分的函数**yylex()**应由语句  
**#include "lex.yy.c"** 代替。
- 使用这条语句，程序**yylex()**可以访问YACC中记号的名字，因为LEX的输出是YACC输出文件的一部分，所以每个LEX动作都返回YACC知道的终结符号。

## 四、YACC内部名称

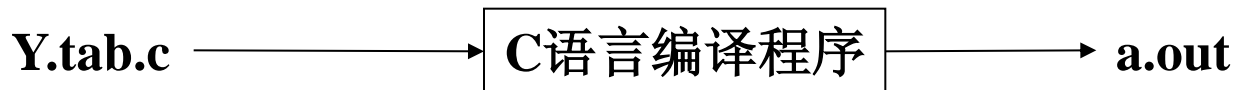
YACC内部名称	说明
y.tab.c	YACC输出文件名
y.tab.h	YACC生成的头文件，包含有记号定义
yyparse	YACC分析程序
yylval	栈中当前记号的值
yyerror	由YACC使用的用户定义的错误信息打印程序
error	YACC错误伪记号
yyerrok	在错误处理之后，使分析程序回到正常操作方式的程序
yychar	变量，记录导致错误的先行记号
YYSTYPE	定义分析栈值类型的预处理器符号
yydebug	变量，当由用户设置为1时，生成有关分析动作的运行信息

# 五、YACC 使用方式

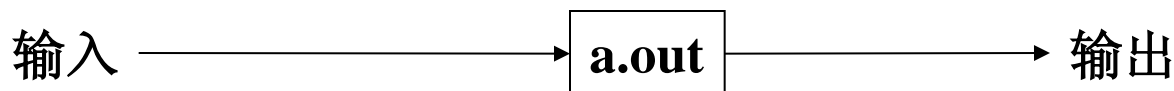
- YACC (Yet Another Compiler-Compiler)
- YACC使用方式:



`%yacc translate.y`



`%cc y.tab.c -ly -o a.out`





# 第5章 语法制导翻译技术



*LI Wensheng, SCST, BUPT*

知识点：语法制导定义、翻译方案

**S-属性定义、L-属性定义**

**S-属性定义的翻译**

**L-属性定义的翻译**

# 语法制导翻译技术

- 语义分析涉及到语言的语义
- 形式语义学的研究开始于20世纪60年代初
- 形式语义学可以分为三类
  - ◆ 操作语义学：通过说明程序在一个机器中是如何执行的来定义程序的语义，着重模拟数据加工过程中计算机系统的操作
  - ◆ 指称语义学：使用数学函数来描述程序和程序的构成，函数通过把语义值联系到正确的语法结构来描述程序的语义，主要描述数据加工的结果
  - ◆ 公理语义学：把数理逻辑应用于语言的语义，语言结构与谓词转换器联系在一起，语言结构的行为以命题刻画，通过描述程序执行对程序断言的影响来定义程序、语句或语言结构的语义，主要用于程序正确性证明
- 语法制导翻译技术
  - ◆ 多数编译程序普遍采用的一种技术
  - ◆ 比较接近形式化

# 语法制导翻译技术

- ◆ 根据翻译目标的要求确定每个产生式所包含的语义;
- ◆ 根据产生式包含的语义, 分析文法中每个符号的语义;
- ◆ 把这些语义以**属性**的形式附加到相应的文法符号上;
- ◆ 根据产生式的语义, 给出符号属性的求值规则(即**语义规则**), 从而形成**语法制导定义**。
- ◆ 在语法分析过程中, 当使用该产生式时, 根据语义规则对相应的属性进行求值, 从而完成翻译。

例如: 考虑算术表达式文法

- ◆ **总目标: 计算表达式的值**
- ◆ 产生式  $E \rightarrow E_1 + T$  的语义: 表达式的值由两个子表达式的值相加得到
- ◆ 分析每个符号的语义, 并以属性的形式记录:  $E.val$ 、 $E_1.val$ 、 $T.val$
- ◆ 求值规则:  $E.val = E_1.val + T.val$
- ◆ 语法制导定义: 产生式    语义规则

$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow \text{digit}$	$F.val = \text{digit}.val$

# 语法制导翻译技术（续）

## ■ 例如：考虑算术表达式文法

### ◆ 总目标：检查表达式的类型

### ◆ 产生式 $E \rightarrow E_1 + T$ 的语义：表达式的类型由两个子表达式的类型综合得到

### ◆ 分析每个符号的语义，并以属性的形式记录： **$E.type$** 、 **$E_1.type$** 、 **$T.type$**

### ◆ 求值规则：

```
if ( $E_1.type == integer$ ) && ( $T.type == integer$ )  
     $E.type = integer$ ;  
else ...
```

# 翻译结果—取决于翻译目标

## ■ 生成代码

- ◆ 可以为源程序产生中间代码
- ◆ 可以直接生成目标机指令

## ■ 对输入符号串进行解释执行

## ■ 向符号表中存放信息

## ■ 给出错误信息

## ■ 翻译的结果依赖于语义规则

- ◆ 使用语义规则进行计算所得到的结果就是对输入符号串进行翻译的结果。
- ◆ 如： $E \rightarrow E + T$  的翻译结果可以是：计算表达式的值、检查表达式的类型是否合法、为表达式创建语法树、生成代码等等。

# 语法制导翻译技术（续）

## ■ 进一步：

- ◆ 用一个或多个子程序（称为**语义动作**）所要完成的功能描述产生式的语义；
- ◆ 把语义动作**插入到产生式中**相应位置，从而形成**翻译方案**。
- ◆ 在语法分析过程中使用该产生式时，在**适当的时机**调用这些动作，完成所需要的翻译。

## ■ 语法制导定义是对翻译的高层次的说明，它隐蔽了一些实现细节，无须指明翻译时语义规则的计算次序。

## ■ 翻译方案指明了语义规则的计算次序，规定了语义动作的执行时机。

# 语法制导翻译的一般步骤

输入符号串

→ 分析树

→ 依赖图

→ 语义规则的计算顺序

→ 计算结果

# 语法制导翻译技术

5. 1 语法制导定义及翻译方案

5. 2 **S**-属性定义的自底向上翻译

5. 3 **L**-属性定义的自顶向下翻译

5. 4 **L**-属性定义的自底向上翻译

小 结



# 5.1 语法制导定义及翻译方案

- 对上下文无关文法的推广
- 每个文法符号都可以有一个**属性集**，其中可以包括两类属性：**综合属性**和**继承属性**。
  - ◆ 左部符号的综合属性是从该产生式右部文法符号的属性值计算出来的；在分析树中，一个内部结点的**综合属性**是从其子结点的属性值计算出来的。
  - ◆ 出现在产生式右部的某文法符号的**继承属性**是从其所在产生式的左部非终结符号和/或右部文法符号的属性值计算出来的；在分析树中，一个结点的**继承属性**是从其兄弟结点和/或父结点的属性值计算出来的。
  - ◆ 分析树中某个结点的**属性值**是由与在这个结点上所用产生式相应的**语义规则**决定的。
- 和产生式相联系的**语义规则**建立了属性之间的关系，这些关系可用**有向图**（即：**依赖图**）来表示。

# 本节内容安排

- 一、语法制导定义
- 二、依赖图
- 三、计算次序
- 四、S属性定义和L属性定义
- 五、翻译方案

# 一、语法制导定义

在一个语法制导定义中，对应于每一个文法产生式  $A \rightarrow \alpha$ ，都有与之相联系的一组语义规则，其形式为：  
 $b = f(c_1, c_2, \dots, c_k)$

这里， $f$ 是一个函数，而且或者

- (1)  $b$ 是 $A$ 的一个综合属性，且 $c_1, c_2, \dots, c_k$ 是产生式右部文法符号的属性，或者
- (2)  $b$ 是产生式右部某个文法符号的一个继承属性，且 $c_1, c_2, \dots, c_k$ 是 $A$ 或产生式右部任何文法符号的属性。

- 属性 $b$ 依赖于属性 $c_1, c_2, \dots, c_k$ 。
- 语义规则函数都不具有副作用的语法制导定义称为属性文法

# 语义规则

## ■ 一般情况：

- ◆ 语义规则函数可写成表达式的形式。

## ■ 某些情况下：

- ◆ 一个语义规则的唯一目的就是产生某个副作用，如打印一个值、向符号表中插入一条记录等；
- ◆ 这样的语义规则通常写成过程调用或程序段。
- ◆ 看成是相应产生式左部非终结符号的虚拟综合属性。
- ◆ 虚属性和符号 ‘=’ 都没有表示出来。

# 简单算术表达式求值的语法制导定义

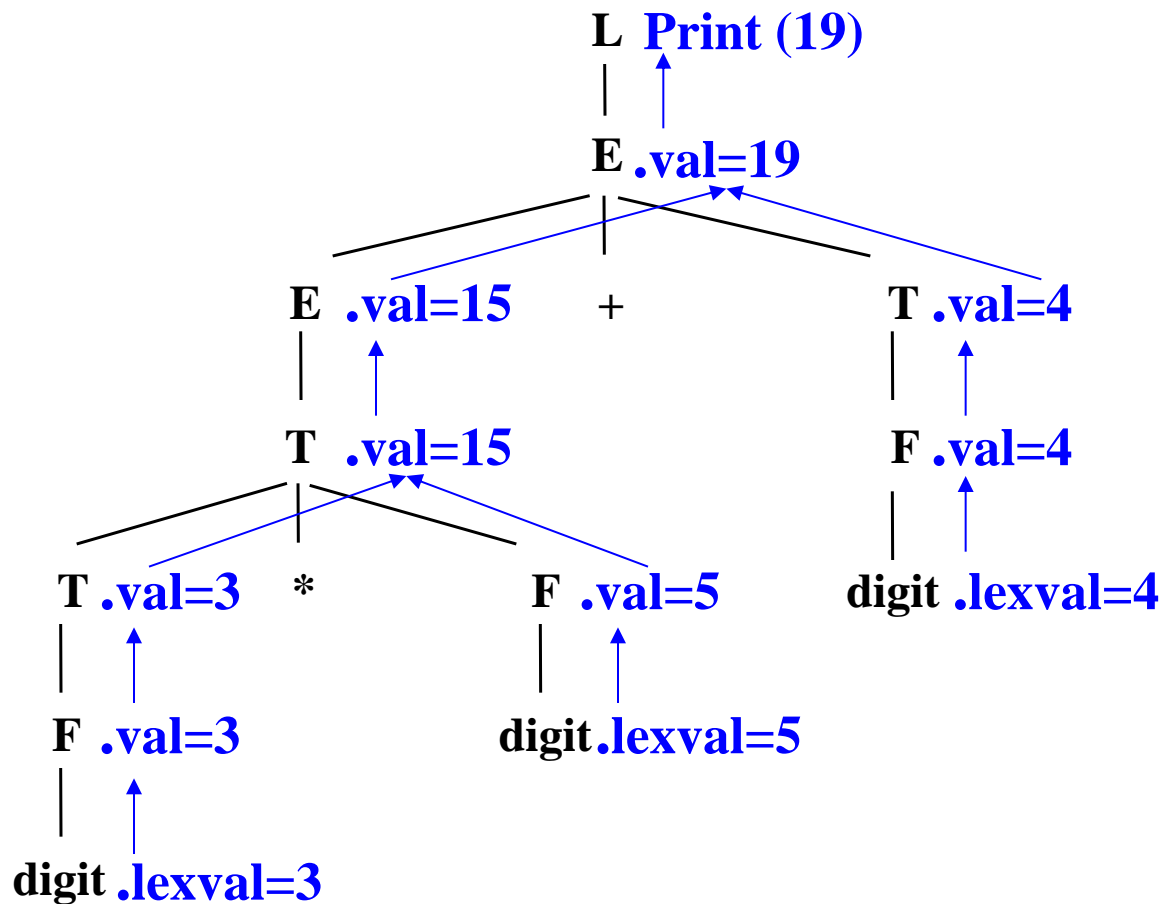
产生式	语义规则
$L \rightarrow E$	<b>print(E.val)</b>
$E \rightarrow E_1 + T$	<b>E.val = E<sub>1</sub>.val + T.val</b>
$E \rightarrow T$	<b>E.val = T.val</b>
$T \rightarrow T_1 * F$	<b>T.val = T<sub>1</sub>.val * F.val</b>
$T \rightarrow F$	<b>T.val = F.val</b>
$F \rightarrow (E)$	<b>F.val = E.val</b>
$F \rightarrow \text{digit}$	<b>F.val = digit.lexval</b>

- **综合属性val**与每一个非终结符号E、T、F相联系
- 表示相应非终结符号所代表的子表达式的整数值
- $L \rightarrow E$ 的语义规则是一个过程，打印出由E产生的算术表达式的值，可以认为是非终结符号**L**的一个**虚拟综合属性**。

# 综合属性

- 分析树中，如果一个结点的某一属性由其子结点的属性确定，则这种属性为该结点的**综合属性**。
- 如果一个语法制导定义仅仅使用综合属性，则称这种语法制导定义为**S-属性定义**。
- 对于**S-属性定义**，通常采用**自底向上**的方法对其分析树加注释，即从树叶到树根，按照语义规则计算每个结点的属性值。
- 简单台式计算机的语法制导定义是**S-属性定义**

# 3\*5+4的分析树加注释的过程



- 属性值的计算可以在语法分析过程中进行。

# 继承属性

- 分析树中，一个结点的**继承属性**值由该结点的父结点和/或它的兄弟结点的属性值决定。
- 可用继承属性表示程序设计语言结构中**上下文之间的依赖关系**
  - ◆ 可以跟踪一个标识符的类型
  - ◆ 可以跟踪一个标识符，了解它是出现在赋值号的右边还是左边，以确定是需要该标识符的值还是地址。

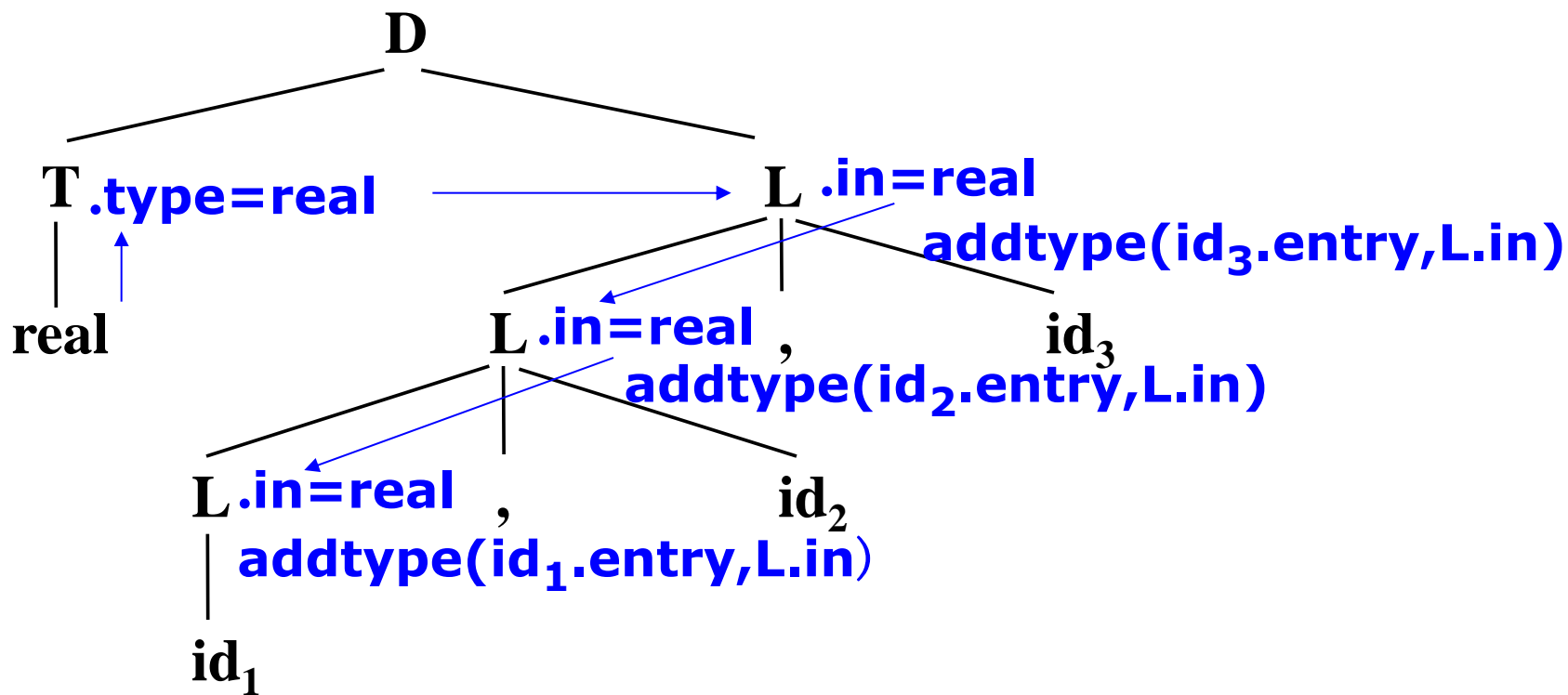


# 用继承属性L.in传递类型信息的语法制导定义

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

- **D**产生的声明语句包含了类型关键字**int**或**real**，后跟一个标识符表。
- **T**有综合属性**type**，其值由声明中的关键字确定。
- **L**的继承属性**L.in**，
  - ◆ 产生式 $D \rightarrow TL$ ，**L.in** 表示从其兄弟结点**T**继承下来的类型信息。
  - ◆ 产生式 $L \rightarrow L_1, id$ ， **L<sub>1</sub>.in** 表示从其父结点**L**继承下来的类型信息

# 语句 $\text{real id}_1, \text{id}_2, \text{id}_3$ 的注释分析树



**L**产生式的语义规则使用继承属性**L.in**把类型信息在分析树中向下传递;

并通过调用过程**addtype**, 把类型信息填入标识符在符号表中相应的表项中。

## 二、依赖图

- 分析树中，结点的继承属性和综合属性之间的相互依赖关系可以由依赖图表示。
- 为每个包含过程调用的语义规则引入一个**虚拟综合属性b**，以便把语义规则统一为 $b=f(c_1, c_2, \dots, c_k)$ 的形式。
- 依赖图中：
  - ◆ 为每个属性设置一个结点
  - ◆ 如果属性b依赖于c，那么从属性c的结点有一条有向边连到属性b的结点。

## 算法5.1 构造依赖图

输入：一棵分析树

输出：一张依赖图

方法：

**for** (分析树中每一个结点 $n$ )

**for** (结点 $n$ 处的文法符号的每一个属性 $a$ )

        为 $a$ 在依赖图中建立一个结点;

**for** (分析树中每一个结点 $n$ )

**for** (结点 $n$ 处所用产生式对应的每一个  
        语义规则  $b=f(c_1, c_2, \dots, c_k)$ )

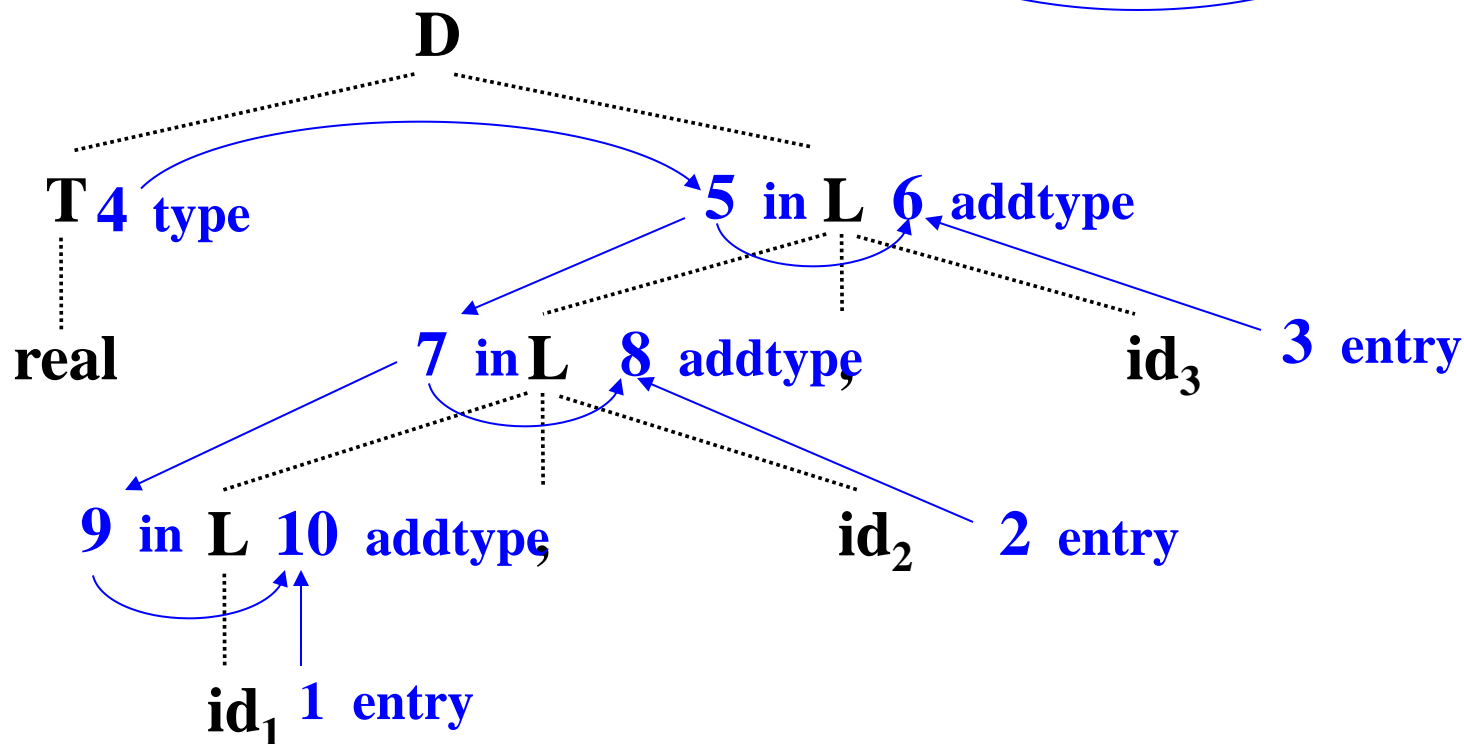
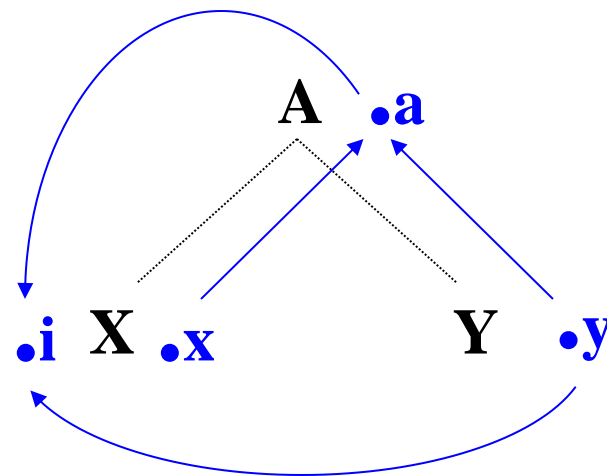
**for** ( $i=1; i \leq k; i++$ )

            从 $c_i$ 结点到 $b$ 结点构造一条有向边;

Wensheng Li BUPT

**A → XY**

## A. $a=f(X, x, Y, y)$

$$X. i = g(A. a, Y. y)$$


# 三、计算次序

## ■ 有向非循环图的拓扑排序

- ◆ 图中结点的一种排序 $m_1, m_2, \dots, m_k$
- ◆ 有向边只能从这个序列中前边的结点指向后面的结点
- ◆ 如果 $m_i \rightarrow m_j$ 是从 $m_i$ 指向 $m_j$ 的一条边，那么在序列中 $m_i$ 必须出现在 $m_j$ 之前。

## ■ 依赖图的任何拓扑排序

- ◆ 给出了分析树中结点的语义规则计算的有效顺序
- ◆ 在拓扑排序中，一个结点上语义规则 $b = f(c_1, c_2, \dots, c_k)$ 中的属性 $c_1, c_2, \dots, c_k$ 在计算 $b$ 时都是可用的。

## ■ 拓扑排序：

- 1、2、3、4、5、6、7、8、9、10
- 4、5、3、6、7、2、8、9、1、10

# 计算顺序

- 从拓扑排序中可以得到下面的程序， $a_n$ 代表依赖图中与序号 $n$ 的结点有关的属性：

**type=real;**

**in5=type;**

**addtype(id<sub>3</sub>.entry,in5);**

**in7=in5;**

**addtype(id<sub>2</sub>.entry,in7);**

**in9=in7;**

**addtype(id<sub>1</sub>.entry,in9);**

# 语法制导翻译过程

- 最基本的文法用于建立输入符号串的分析树；
- 为分析树构造依赖图；
- 对依赖图进行拓扑排序；
- 从这个序列得到语义规则的计算顺序；
- 照此计算顺序进行求值，得到对输入符号串的翻译。

输入符号串

————→ 分析树

————→ 依赖图

————→ 语义规则的计算顺序

————→ 计算结果



## 四、S属性定义和L属性定义

- **S属性定义**：仅涉及综合属性的语法制导定义
- **L属性定义**：一个语法制导定义是**L属性定义**，如果
  - ◆ 与每个产生式 $A \rightarrow X_1 X_2 \dots X_n$ 相应的每条语义规则计算的属性都是**A的综合属性**，或是 **$X_j$  ( $1 \leq j \leq n$ ) 的继承属性**，而该继承属性**仅依赖于**：
    - **A的继承属性**；
    - 产生式中 $X_j$ 左边的符号 $X_1$ 、 $X_2$ 、...、 $X_{j-1}$ 的属性；
- 每一个**S属性定义**都是**L属性定义**

例：非L属性定义的  
语法制导定义

产生式	语义规则
$A \rightarrow LM$	$L.i = 1(A.i)$ $M.i = m(L.s)$ $A.s = f(M.s)$
$A \rightarrow QR$	$R.i = r(A.i)$ $Q.i = q(R.s)$ $A.s = f(Q.s)$

例：是L属性定义的  
语法制导定义

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

# 属性计算顺序——深度优先遍历分析树

```
void deepfirst (n: node)
{
    for (n的每一个子结点m, 从左到右) {
        计算m的继承属性;
        deepfirst(m);
    };
    计算n的综合属性;
}.
```

- 以分析树的根结点作为**实参**
- L属性定义的属性都可以用**深度优先的顺序**计算。
  - ◆ **进入**结点前, 计算它的继承属性
  - ◆ 从结点**返回**时, 计算它的综合属性

# 五、翻译方案

- 上下文无关文法的一种便于翻译的书写形式
- 属性与文法符号相对应
- 语义动作括在花括号中，并插入到产生式右部某个合适的位置上
- 给出了使用语义规则进行属性计算的顺序
- 分析过程中翻译的注释

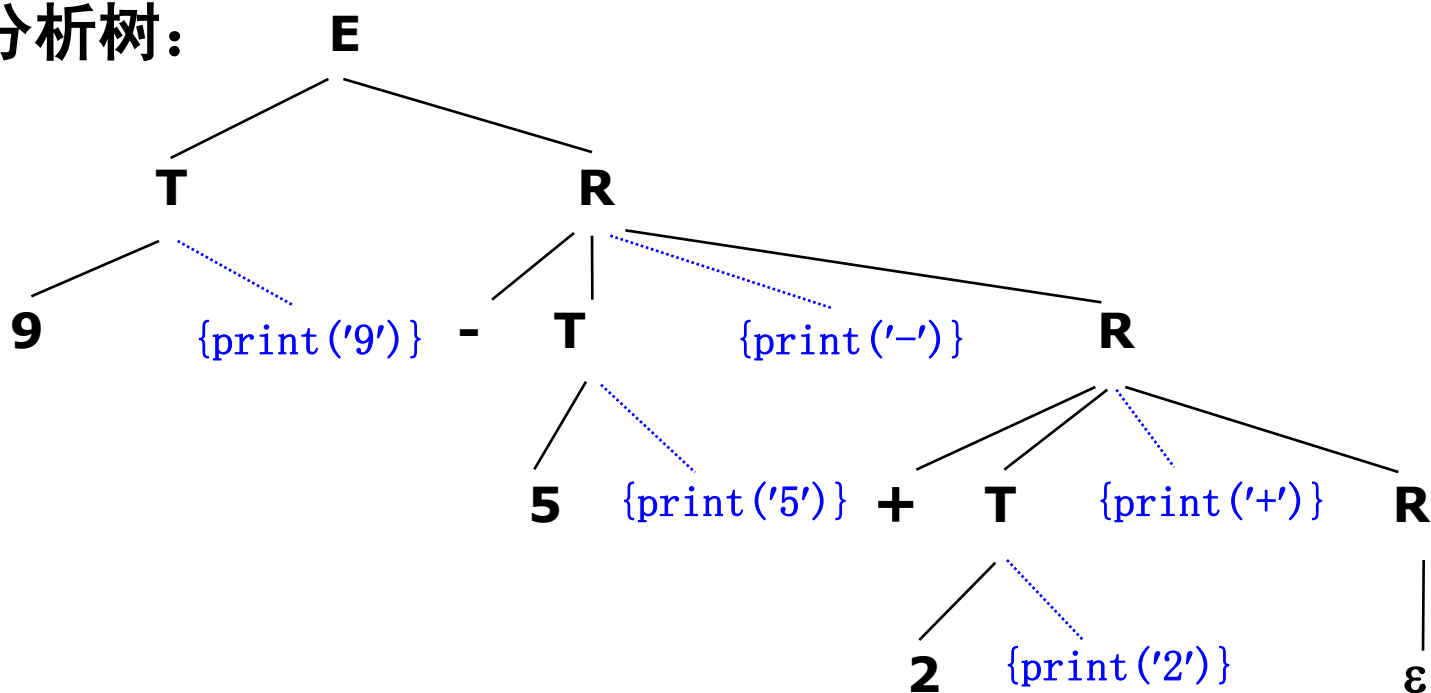
# 例：一个简单的翻译方案

$E \rightarrow TR$

$R \rightarrow +T \{ \text{print} ( '+' ) \} R_1$   
 $\quad \quad \quad | -T \{ \text{print} ( '-' ) \} R_1 \quad | \quad \varepsilon$

$T \rightarrow \text{num} \{ \text{print}(\text{num.val}) \}$

9-5+2的分析树:



语义动作作为相应产生式左部符号对应结点的子结点

深度优先遍历树中结点，执行其中的动作，打印出**95-2+**

# 翻译方案的设计

## ■ 对于S属性定义:

- ◆ 为每一个语义规则建立一个包含赋值的动作
- ◆ 把这个动作放在相应的产生式右边末尾

例：产生式                  语义规则

$T \rightarrow T_1 * F$        $T.val = T_1.val * F.val$

如下安排产生式和语义动作:

$T \rightarrow T_1 * F \{ T.val = T_1.val * F.val \}$

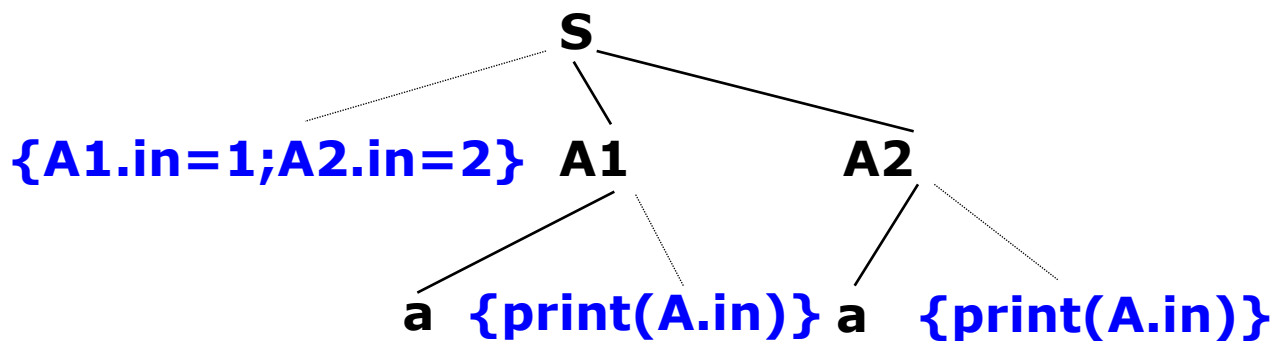
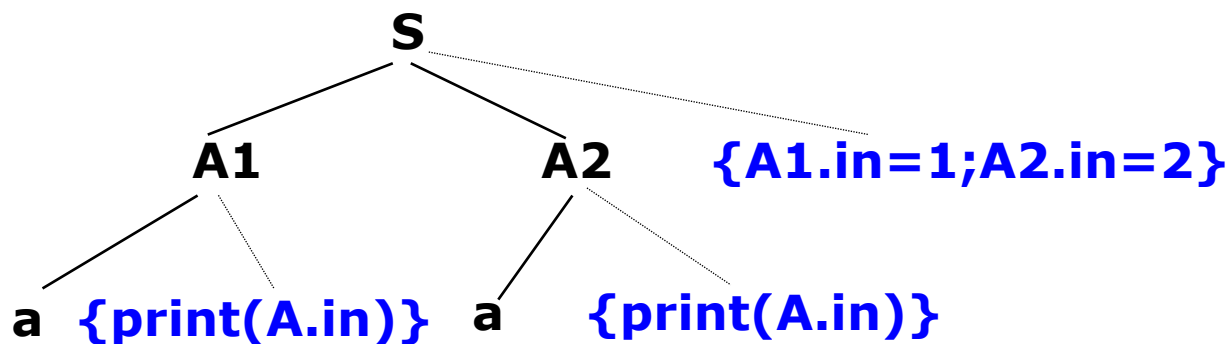
# 为L属性定义设计翻译方案的原则

- 产生式右部文法符号的**继承属性**必须在这个符号以前的动作中计算出来
  - ◆ 计算该继承属性的动作必须出现在相应文法符号之前
- 一个动作不能引用这个动作右边的文法符号的综合属性
- 产生式左边非终结符号的**综合属性**只有在它所引用的所有属性都计算出来之后才能计算
  - ◆ 这种属性的计算动作放在产生式右端末尾

例：考虑如下翻译方案

$S \rightarrow A_1 A_2 \{A_1.in=1; A_2.in=2\}$

$A \rightarrow a \{print(A.in)\}$





# L属性定义翻译方案设计举例

## ■ 语法制导定义

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

## ■ 翻译方案

$D \rightarrow T \{ \text{L.in} = \text{T.type} \} L$   
 $T \rightarrow int \{ T.type = integer \}$   
 $T \rightarrow real \{ T.type = real \}$   
 $L \rightarrow \{ \text{L1.in} = \text{L.in} \} L_1, id \{ addtype(id.entry, L.in) \}$   
 $L \rightarrow id \{ addtype(id.entry, L.in) \}$

## 例：为如下L属性定义设计翻译方案

产生式	语义规则
<b><math>S \rightarrow B</math></b>	<b><math>B.ps = 10</math> <math>S.ht = B.ht</math></b>
<b><math>B \rightarrow B_1 B_2</math></b>	<b><math>B_1.ps = B.ps</math> <math>B_2.ps = B.ps</math> <math>B.ht = \max(B_1.ht, B_2.ht)</math></b>
<b><math>B \rightarrow B_1 \text{sub} B_2</math></b>	<b><math>B_1.ps = B.ps</math> <math>B_2.ps = \text{shrink}(B.ps)</math> <math>B.ht = \text{disp}(B_1.ht, B_2.ht)</math></b>
<b><math>B \rightarrow \text{text}</math></b>	<b><math>B.ht = \text{text.h} \times B.ps</math></b>

- L属性定义
- 唯一的继承属性是非终结符号B的ps属性
  - ◆ 依赖于左部符号的继承属性，或者
  - ◆ 常数

# 把语义规则插入到产生式中适当的位置

$S \rightarrow \{ \mathbf{B.ps=10} \} \mathbf{B} \{ \mathbf{S.ht=B.ht} \}$

$S \rightarrow \{ \mathbf{B.ps=10} \}$

$\mathbf{B} \{ \mathbf{S.ht=B.ht} \}$

$\mathbf{B} \rightarrow \{ \mathbf{B_1.ps=B.ps} \}$

$\mathbf{B_1} \{ \mathbf{B_2.ps=B.ps} \}$

$\mathbf{B_2} \{ \mathbf{B.ht=max(B_1.ht,B_2.ht)} \}$

$\mathbf{B} \rightarrow \{ \mathbf{B_1.ps=B.ps} \}$

$\mathbf{B_1}$

$\mathbf{sub} \{ \mathbf{B_2.ps=shrink(B.ps)} \}$

$\mathbf{B_2} \{ \mathbf{B.ht=disp(B_1.ht,B_2.ht)} \}$

$\mathbf{B} \rightarrow \mathbf{text} \{ \mathbf{B.ht=text.h \times B.ps} \}$

作业:

5. 15

5. 16

## 5.2 S-属性定义的自底向上翻译

### ■ S属性定义:

- ◆ 只用综合属性的语法制导定义

一、构造表达式的语法树

二、构造表达式的语法树的语法制导定义

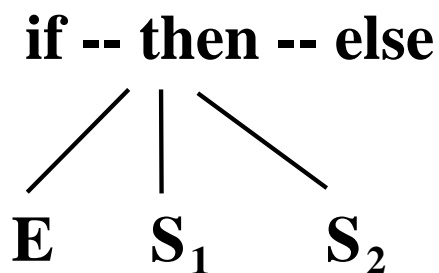
三、S属性定义的自底向上实现

# 一、构造表达式的语法树

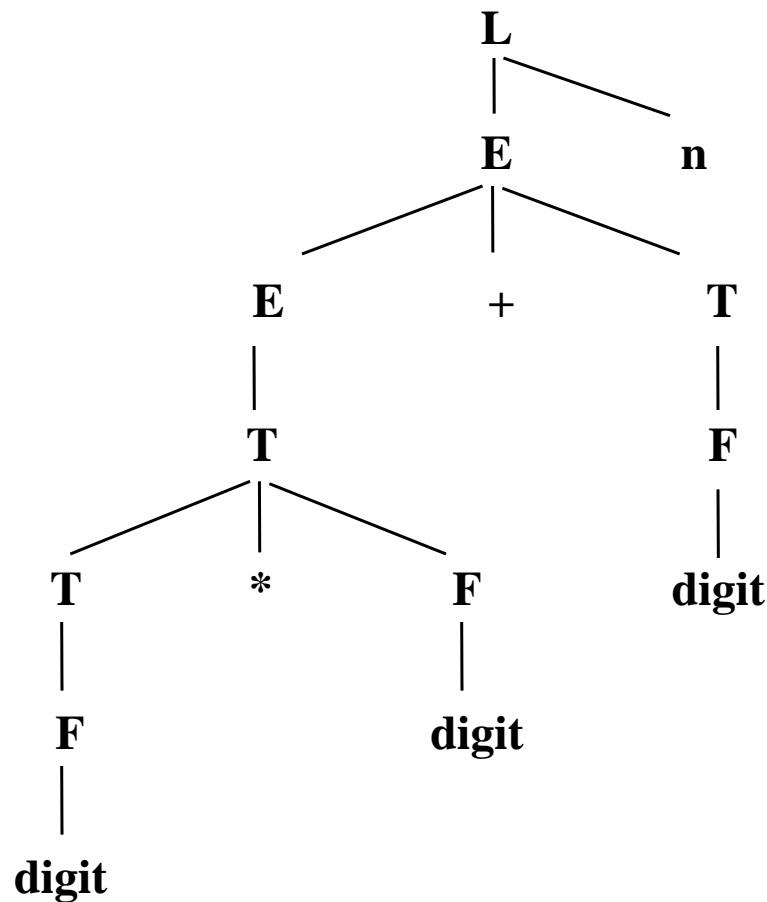
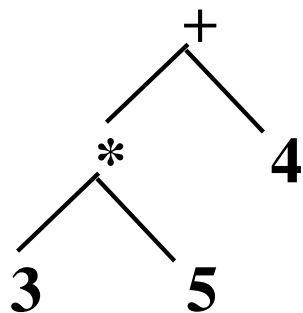
- 把语法规则中对语义无关紧要的具体规定去掉，剩下的本质性的东西称为**抽象语法**。
- 如：
  - ◆ 赋值语句： $x=y$ 、 $x:=y$ 、或 $y\rightarrow x$
  - ◆ 抽象形式：**assignment(variable,expression)**
- 语法树：
  - ◆ 分析树的抽象（或压缩）形式。
  - ◆ 也称为语法结构树或结构树。
  - ◆ **内部结点**表示运算符号，其**子结点**表示它的运算分量。

# 语法树示例

- $S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$  的语法树



- 表达式  $3*5+4$  的语法树



# 构造表达式的语法树

## ■ 表达式的语法树的形式

- ◆ 每一个运算符号或运算分量都对应树中的一个结点
- ◆ 运算符号结点的子结点分别是与该运算符的各个运算分量相应的子树的根。
- ◆ 每一个结点可包含若干个域：
  - 标识域、指针域、属性值域等

## ■ 在运算符结点中

- ◆ 一个域标识运算符号
- ◆ 其它各域包含指向与各运算分量相应的结点的指针
- ◆ 称运算符号为该结点的标号

# 构造函数

## ■ **makenode (op, left, right)**

- ◆ 建立一个运算符结点，标号是**op**；
- ◆ 域**left**和**right**是指向其左右运算分量结点的指针。

## ■ **makeleaf (id, entry)**

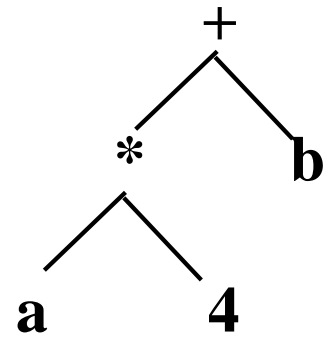
- ◆ 建立一个标识符结点，标号是**id**；
- ◆ 域**entry**是指向该标识符在符号表中的相应条目的指针。

## ■ **makeleaf (num, val)**

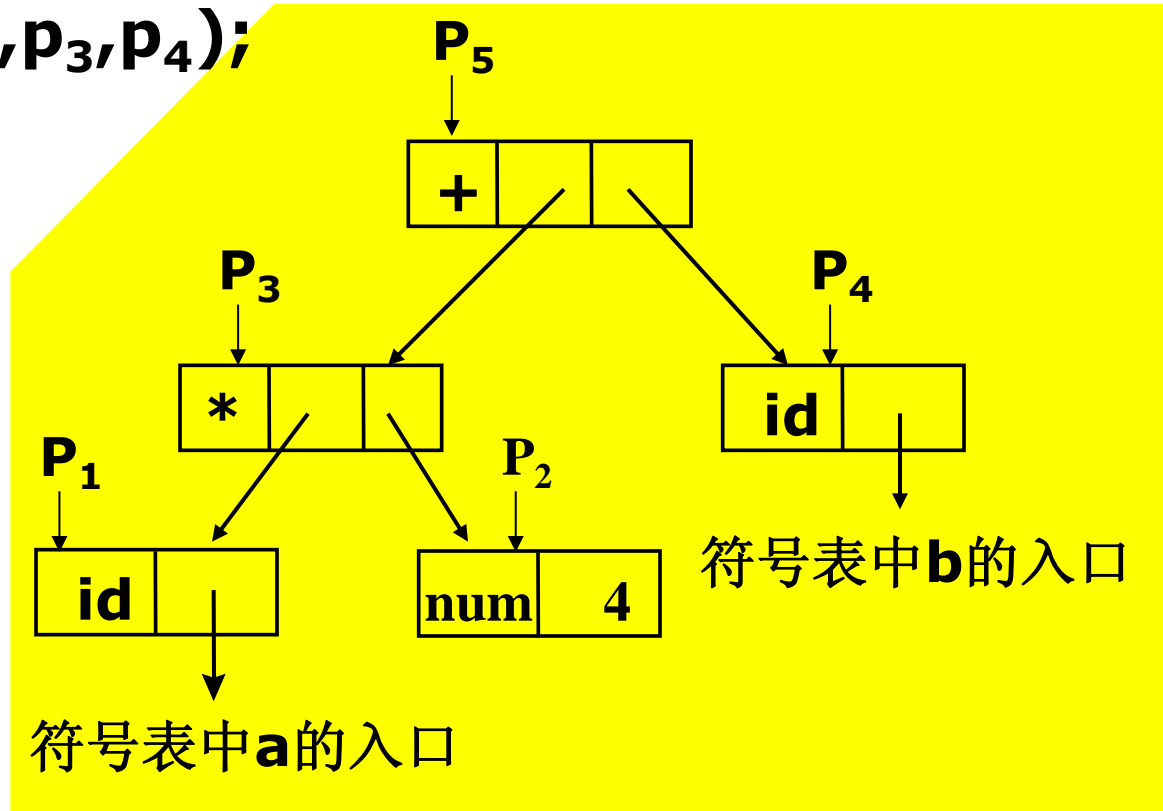
- ◆ 建立一个数结点，标号为**num**；
- ◆ 域**val**用于保存该数的值。



# 建立表达式 $a * 4 + b$ 的语法树



```
p1=makeleaf(id,entrya);  
p2=makeleaf(num,4);  
p3=makenode('*',p1,p2);  
p4=makeleaf(id,entryb);  
p5=makenode('+',p3,p4);
```



## 二、构造表达式语法树的语法制导定义

- 目标：为表达式创建语法树
- 产生式语义：创建与产生式左部符号代表的子表达式对应的子树，即创建子树的根结点。
- 文法符号的属性：记录所建结点， **E.nptr**、**T.nptr**、**F.nptr** 指向相应子树根结点的指针
- 产生式的语义动作举例：  
 $E \rightarrow E_1 + T$        $E.nptr = \text{makenode}('+', E_1.nptr, T.nptr)$   
 $T \rightarrow F$              $T.nptr = F.nptr$   
 $F \rightarrow id$             $F.nptr = \text{makeleaf}(id, id.entry)$   
 $F \rightarrow num$            $F.nptr = \text{makeleaf}(num, num.val)$

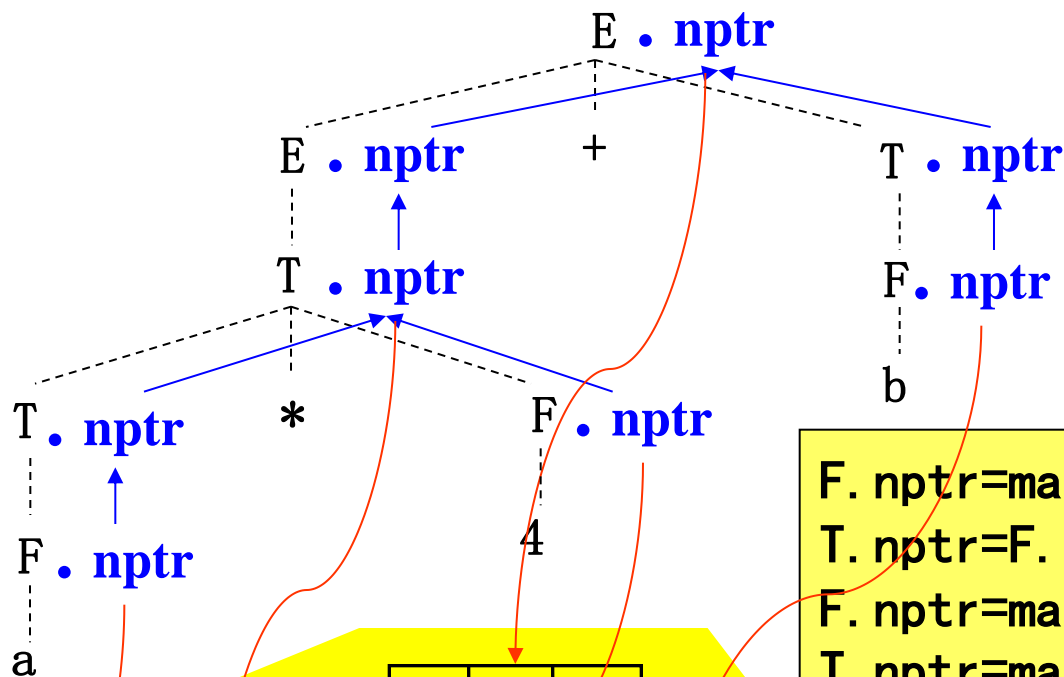
# 构造表达式语法树的语法制导定义

产生式	语义规则
$E \rightarrow E_1 + T$	$E.\text{nptr} = \text{makenode}(' + ', E_1.\text{nptr}, T.\text{nptr})$
$E \rightarrow T$	$E.\text{nptr} = T.\text{nptr}$
$T \rightarrow T_1 * F$	$T.\text{nptr} = \text{makenode}(' * ', T_1.\text{nptr}, F.\text{nptr})$
$T \rightarrow F$	$T.\text{nptr} = F.\text{nptr}$
$F \rightarrow (E)$	$F.\text{nptr} = E.\text{nptr}$
$F \rightarrow \text{id}$	$F.\text{nptr} = \text{makeleaf}(\text{id}, \text{id.entry})$
$F \rightarrow \text{num}$	$F.\text{nptr} = \text{makeleaf}(\text{num}, \text{num.val})$

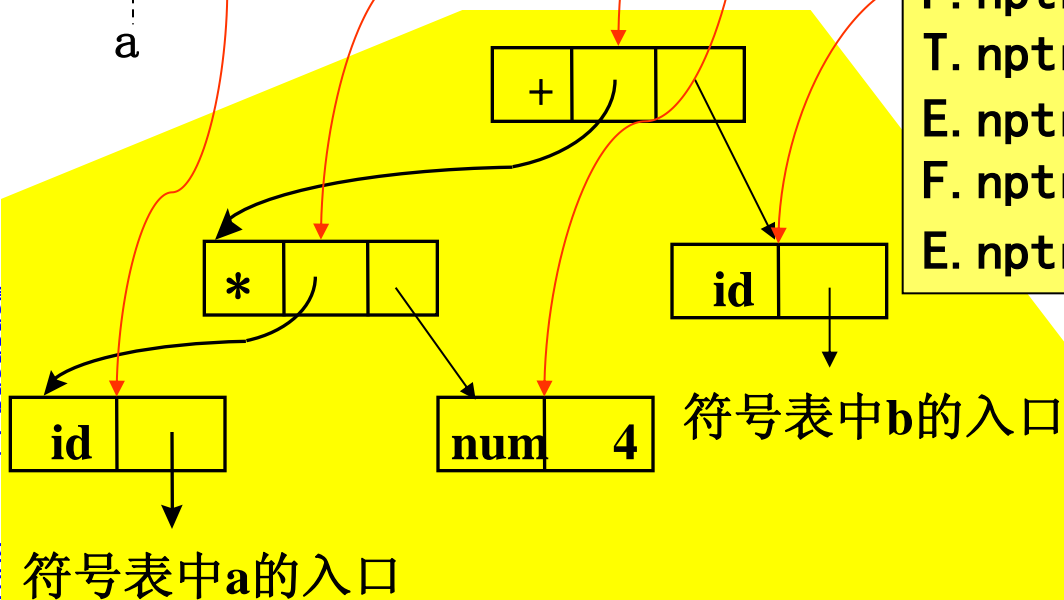
- 为了记录在构造过程中建立的子树，为每个非终结符号引入一个综合属性 **nptr**。
- **nptr** 是一个指针，指向语法树中相应非终结符号产生的表达式子树的根结点。

# 表达式a\*4+b的语法树的构造

$E \rightarrow E_1 + T$   
 $E \rightarrow T$   
 $T \rightarrow T_1 * F$   
 $T \rightarrow F$   
 $F \rightarrow (E)$   
 $F \rightarrow id$   
 $F \rightarrow num$

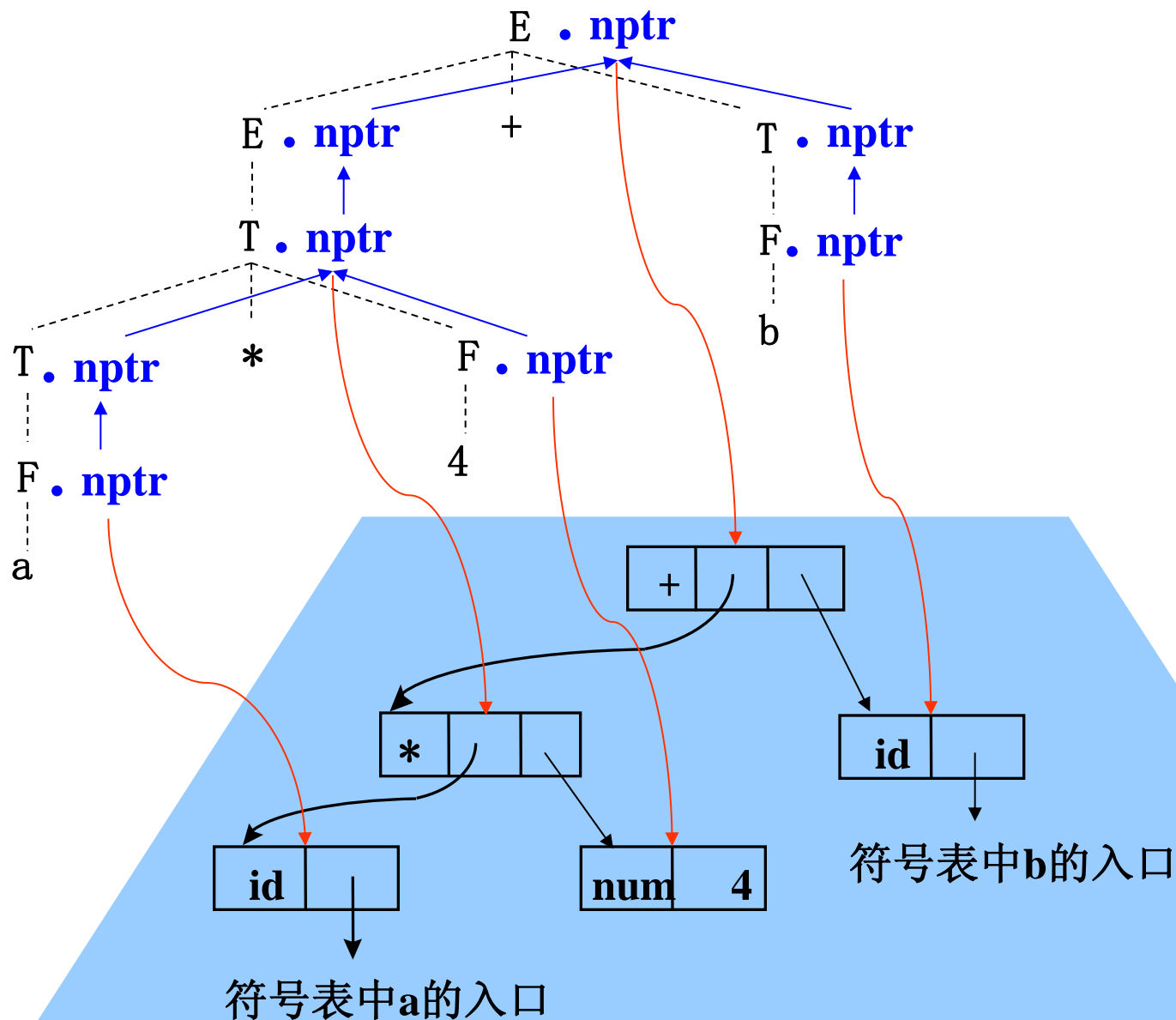


$F.nptr = \text{makeleaf}(id, \text{entry}_a)$   
 $T.nptr = F.nptr$   
 $F.nptr = \text{makeleaf}(\text{num}, 4)$   
 $T.nptr = \text{makenode}('*', T_1.nptr, F.nptr)$   
 $E.nptr = T.nptr$   
 $F.nptr = \text{makeleaf}(id, \text{entry}_b)$   
 $E.nptr = \text{makenode}('+', E_1.nptr, T.nptr)$



# 表达式 $a * 4 + b$ 的语法树的构造

$E \rightarrow E_1 + T$   
 $E \rightarrow T$   
 $T \rightarrow T_1 * F$   
 $T \rightarrow F$   
 $F \rightarrow (E)$   
 $F \rightarrow id$   
 $F \rightarrow num$



# 表达式的有向非循环图(dag)

## ■ dag与语法树相同的地方:

- ◆ 表达式的每一个子表达式都有一个结点
- ◆ 一个内部结点表示一个运算符号, 且它的子结点表示它的运算分量。

## ■ dag与语法树不同的地方:

- ◆ dag中, 对应一个公共子表达式的结点具有多个父结点
- ◆ 语法树中, 公共子表达式被表示为重复的子树

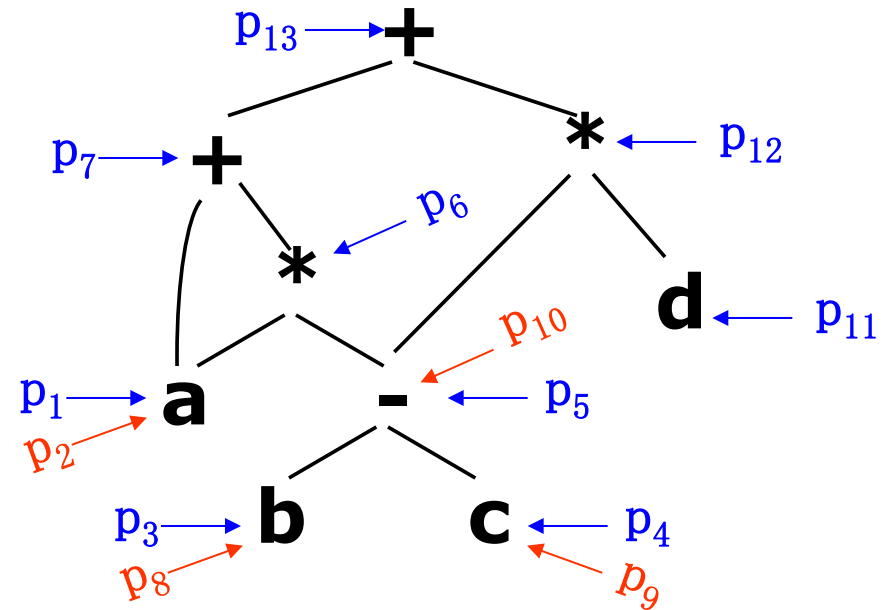
## ■ 为表达式创建dag的函数makenode和makeleaf

- ◆ 建立新结点之前先检查是否已经存在一个相同的结点
- ◆ 若已存在, 返回一个指向先前已构造好的结点的指针;
- ◆ 否则, 创建一个新结点, 返回指向新结点的指针。

# 为表达式 $a+a*(b-c)+(b-c)*d$ 构造dag

## 函数调用

- $p_1 = \text{makeleaf}(\text{id}, a);$
- $p_2 = \text{makeleaf}(\text{id}, a);$
- $p_3 = \text{makeleaf}(\text{id}, b);$
- $p_4 = \text{makeleaf}(\text{id}, c);$
- $p_5 = \text{makenode}('-', p_3, p_4);$
- $p_6 = \text{makenode}('*', p_2, p_5);$
- $p_7 = \text{makenode}('+', p_1, p_6);$
- $p_8 = \text{makeleaf}(\text{id}, b);$
- $p_9 = \text{makeleaf}(\text{id}, c);$
- $p_{10} = \text{makenode}('-', p_8, p_9);$
- $p_{11} = \text{makeleaf}(\text{id}, d);$
- $p_{12} = \text{makenode}('*', p_{10}, p_{11});$
- $p_{13} = \text{makenode}('+', p_7, p_{12});$



# 三、S-属性定义的自底向上实现

## ■ 已知

- ◆ 自底向上的分析方法中，分析程序使用一个**栈**来存放已经分析过的子树的信息。
- ◆ 分析树中某结点的**综合属性**由其子结点的属性值计算得到
- ◆ 自底向上的分析程序在分析输入符号串的**同时**可以计算综合属性

## ■ 考虑

- ◆ 如何**保存**文法符号的**综合属性值**？
- ◆ 保存属性值的**数据结构**怎样与**分析栈相联系**？
- ◆ **怎样保证**：每当进行归约时，由栈中正在归约的产生式右部符号的属性值计算其左部符号的综合属性值。



# 扩充分析栈

目的：使之能够保存综合属性

做法：在分析栈中**增加一个域**，来存放综合属性值

例：带有综合属性域的分析栈

top→	Z	Z.z
	Y	Y.y
	X	X.x
	---	---
	state	val

- 栈由一对数组state和val实现
- state元素是指向LR(1)分析表中状态的指针（或索引）
- 如果state[i]对应的符号为A，val[i]中就存放分析树中与结点A对应的属性值。

- **假设**综合属性刚好在每次归约前计算

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

# 修改分析程序

## ■ 对于终结符号

- ◆ 其综合属性值由词法分析程序产生
- ◆ 当分析程序执行移进操作时，其属性值随终结符号（或状态符号）一起入栈。

## ■ 为每个语义规则编写一段代码，以计算属性值

## ■ 对每一个产生式 $A \rightarrow XYZ$

- ◆ 把属性值的计算与归约动作联系起来
- ◆ 归约前，执行与产生式相关的代码段
- ◆ 归约：右部符号及其属性出栈  
左部符号及其属性入栈


## ■ LR分析程序中应增加计算属性值的代码段

top →

Z	Z.z
Y	Y.y
X	X.x
...	...

state val

# 例：用LR分析程序实现表达式求值

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

代码段

**$\text{print}(\text{val}[\text{top}])$**

**$\text{val}[\text{ntop}] = \text{val}[\text{top}-2] + \text{val}[\text{top}]$**

**$\text{val}[\text{ntop}] = \text{val}[\text{top}-2] * \text{val}[\text{top}]$**

**$\text{val}[\text{ntop}] = \text{val}[\text{top}-1]$**

栈指针变量**top**和**ntop**的控制：

- ◆ 当用  $A \rightarrow \beta$  归约时，若  $|\beta| = r$ ，在执行相应的代码段之前，  **$\text{ntop} = \text{top} - r + 1$** 。
- ◆ 在每一个代码段被执行之后，  **$\text{top} = \text{ntop}$**

# 对 $3*5+4$ 进行分析的动作序列 (见表4.8)

步骤	输入	分析栈	分析动作
(1)	$3*5+4\$$	state: 0 val: -	移进 5
(2)	$*5+4\$$	state: 0 5 val: - 3	归约, 用 $F \rightarrow \text{digit}$ goto[0,F]=3
(3)	$*5+4\$$	state: 0 3 val: - 3	归约, 用 $T \rightarrow F$ goto[0,T]=2
(4)	$*5+4\$$	state: 0 2 val: - 3	移进 7
(5)	$5+4\$$	state: 0 2 7 val: - 3 -	移进 5
(6)	$+4\$$	state: 0 2 7 5 val: - 3 - 5	归约, 用 $F \rightarrow \text{digit}$ goto[7,F]=10
(7)	$+4\$$	state: 0 2 7 10 val: - 3 - 5	归约, 用 $T \rightarrow T * F$ goto[0,T]=2

步骤	输入	分析栈	分析动作
(8)	+4\$	state: 0 2 val: - 15	归约, 用 $E \rightarrow T$ goto[0,E]=
(9)	+4\$	state: 0 1 val: - 15	移进 6
(10)	4\$	state: 0 1 6 val: - 15 -	移进 5
(11)	\$	state: 0 1 6 5 val: - 15 - 4	归约, 用 $F \rightarrow \text{digit}$ goto[6,F]=3
(12)	\$	state: 0 1 6 3 val: - 15 - 4	归约, 用 $T \rightarrow F$ goto[6,T]=9
(13)	\$	state: 0 1 6 9 val: - 15 - 4	归约, 用 $E \rightarrow E+T$ goto[0,E]=1
(14)	\$	state: 0 1 val: - 19	接受

作业:

5.1

5.2

5.3

5.4

## 5.3 L-属性定义的自顶向下翻译

- 在自顶向下的分析过程中实现L属性定义的翻译
- 预测分析方法对文法的要求
  - ◆ 不含左递归
  - ◆  $A \rightarrow \alpha \mid \beta$   
 $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \phi$

一、消除翻译方案中的左递归

二、预测翻译程序的设计

# 一、消除翻译方案中的左递归

## ■ 例：考虑对简单表达式求值的语法制导定义

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

翻译方案：

- (1)  $L \rightarrow E \{ \text{print}(E.\text{val}) \}$
- (2)  $E \rightarrow E_1 + T \{ E.\text{val} = E_1.\text{val} + T.\text{val} \}$
- (3)  $E \rightarrow T \{ E.\text{val} = T.\text{val} \}$
- (4)  $T \rightarrow T_1 * F \{ T.\text{val} = T_1.\text{val} * F.\text{val} \}$
- (5)  $T \rightarrow F \{ T.\text{val} = F.\text{val} \}$
- (6)  $F \rightarrow (E) \{ F.\text{val} = E.\text{val} \}$
- (7)  $F \rightarrow \text{digit} \{ F.\text{val} = \text{digit}.\text{val} \}$

消除左递归的方法：

$A \rightarrow A\alpha \mid \beta$

替换为：  $A \rightarrow \beta R$

$R \rightarrow \alpha R \mid \varepsilon$

由(2)和(3)有：

(2')  $E \rightarrow T \{ E.\text{val} = T.\text{val} \} M$

(3')  $M \rightarrow +T \{ E.\text{val} = E_1.\text{val} + T.\text{val} \} M_1$

(3'')  $M \rightarrow \varepsilon$

继承属性M. i：表示在M之前已经推导出的子表达式的值  
综合属性M. s：表示在M完全展开之后得到的表达式的值

为 (3'') 设置把 M.i 传递给 M.s 的语义动作, 得到:

$$(3'') \quad M \rightarrow_{\epsilon} \{M.s = M.i\}$$

对于 (2'),  $E \rightarrow T \{E.val = T.val\}$  M 通过 M 的属性 M.s 和 M.i 完成 E 和 T 的综合属性的传递  $E.val = T.val$ , 得到:

$$(2') \quad E \rightarrow T \{M.i = T.val\} \\ M \{E.val = M.s\}$$

对于 (3')  $M \rightarrow +T \{E.val = E_1.val + T.val\}$   $M_1$

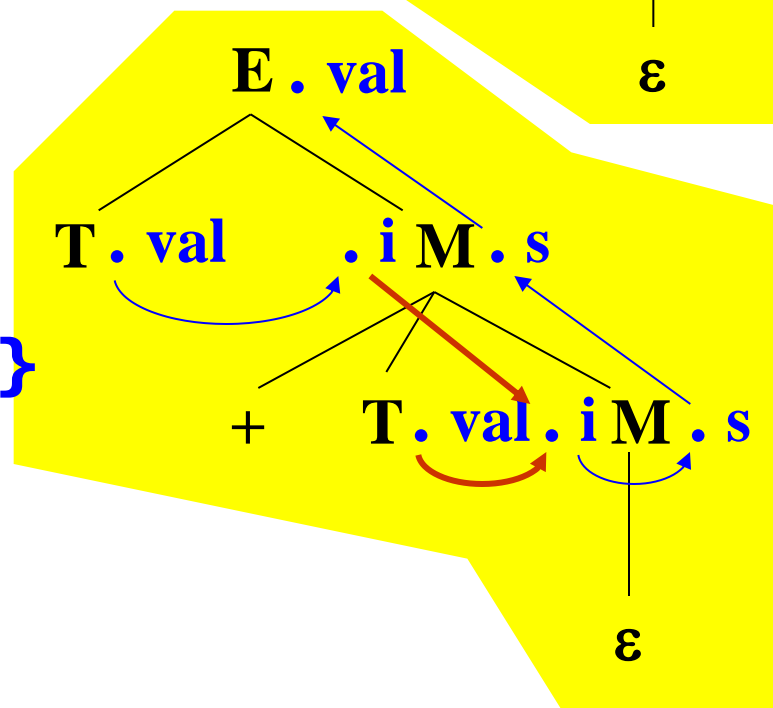
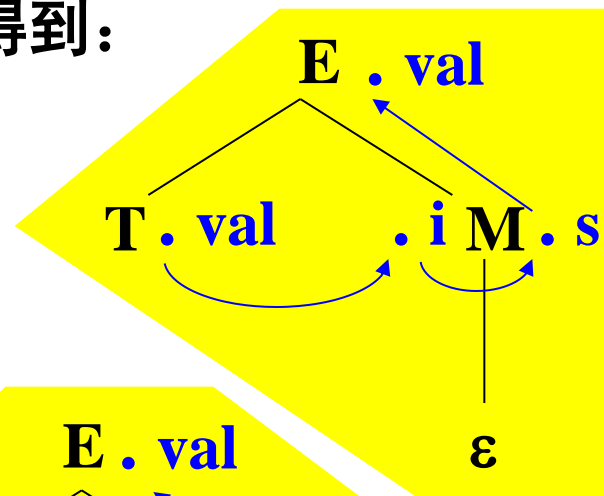
$M_1.i$  的语义规则为:  $M_1.i = M.i + T.val$

M.s 的语义规则为:  $M.s = M_1.s$

于是得到:

$$(3') \quad M \rightarrow +T \{M_1.i = M.i + T.val\} \\ M_1 \{M.s = M_1.s\}$$

同样, 通过引入非终结符号 N,  
可以得到 (4) 和 (5) 的变换结果





# 翻译方案

## 表达式 $3*5+4$ 的翻译过程

$L \rightarrow E \{ \text{print}(E.\text{val}) \}$

$E \rightarrow T \{ M.i = T.\text{val} \}$

$M \{ E.\text{val} = M.s \}$

$M \rightarrow +$

$T \{ M_1.i = M.i + T.\text{val} \}$

$M_1 \{ M.s = M_1.s \}$

$M \rightarrow \epsilon \{ M.s = M.i \}$

$T \rightarrow F \{ N.i = F.\text{val} \}$

$N \{ T.\text{val} = N.s \}$

$N \rightarrow *$

$F \{ N_1.i = N.i * F.\text{val} \}$

$N_1 \{ N.s = N_1.s \}$

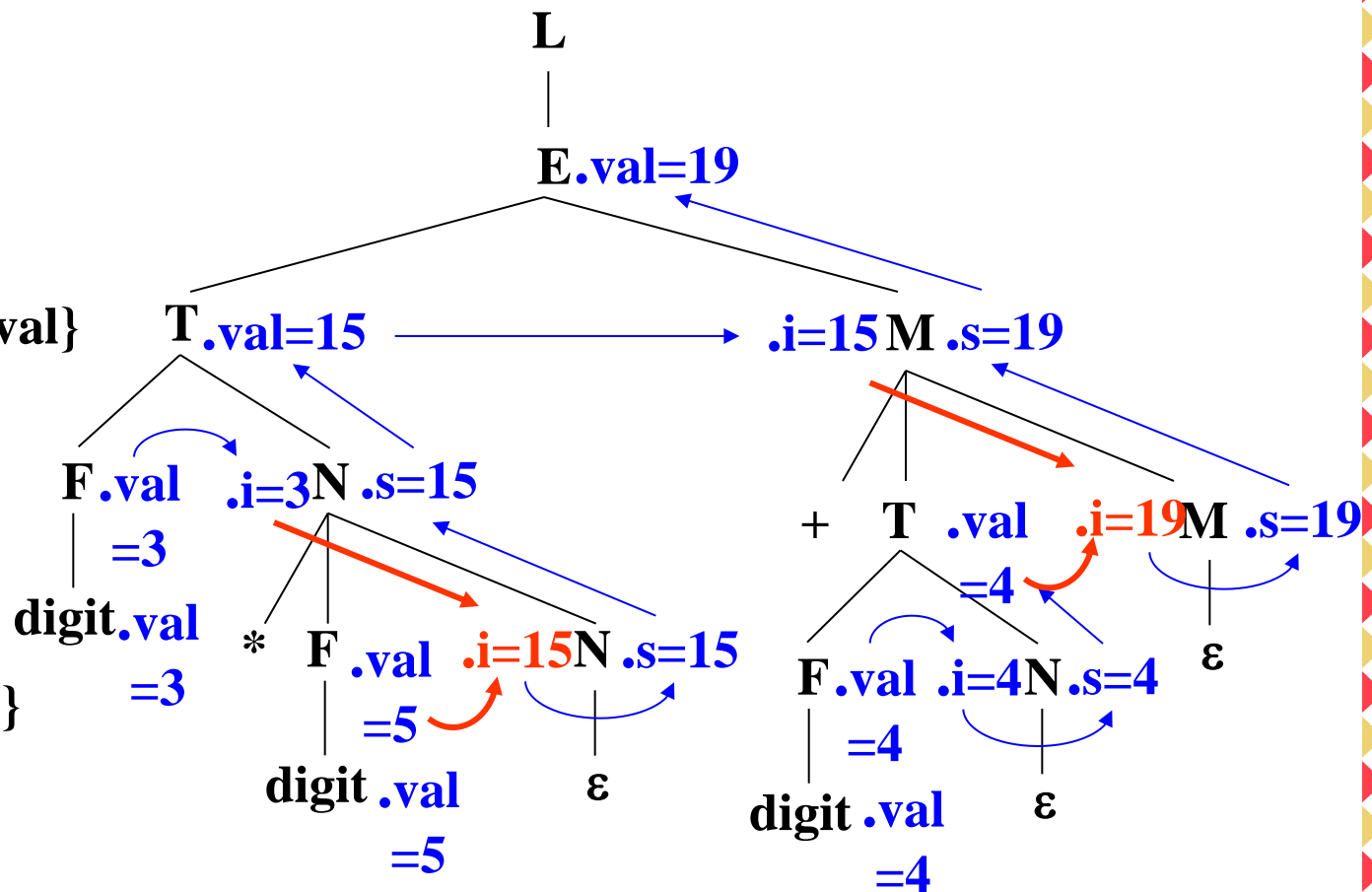
$N \rightarrow \epsilon \{ N.s = N.i \}$

$F \rightarrow ($

$E$

$) \{ F.\text{val} = E.\text{val} \}$

$F \rightarrow \text{digit} \{ F.\text{val} = \text{digit}.\text{lexval} \}$



Wensheng Li BUPT

## 翻译方案:

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

## 消除基本文法中的左递归:

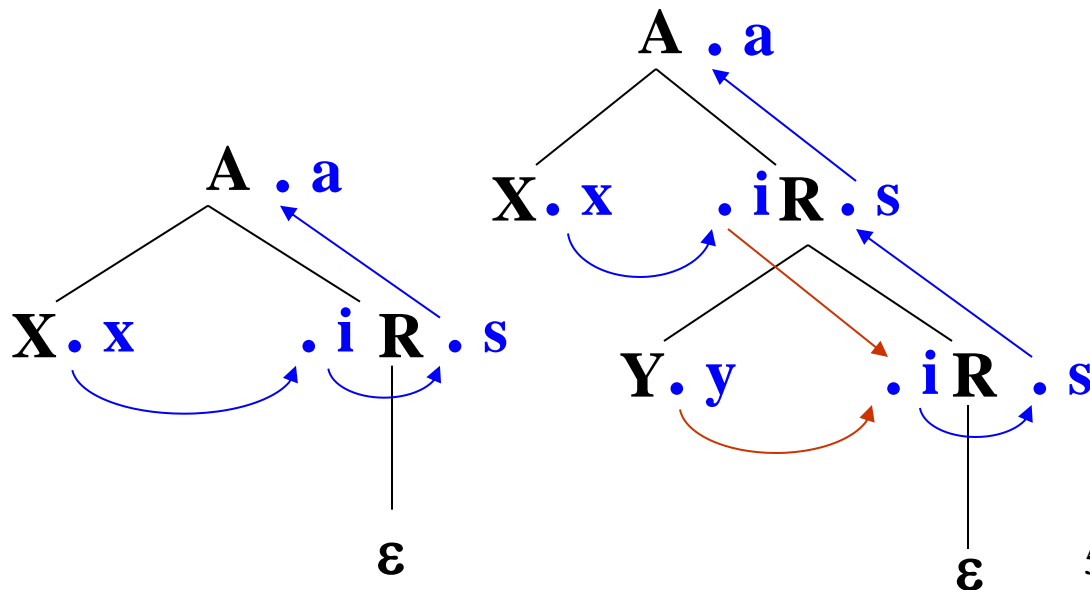
$$A \rightarrow XR$$
$$R \rightarrow YR \mid \varepsilon$$

为R设置继承属性R. i和综合属性R. s

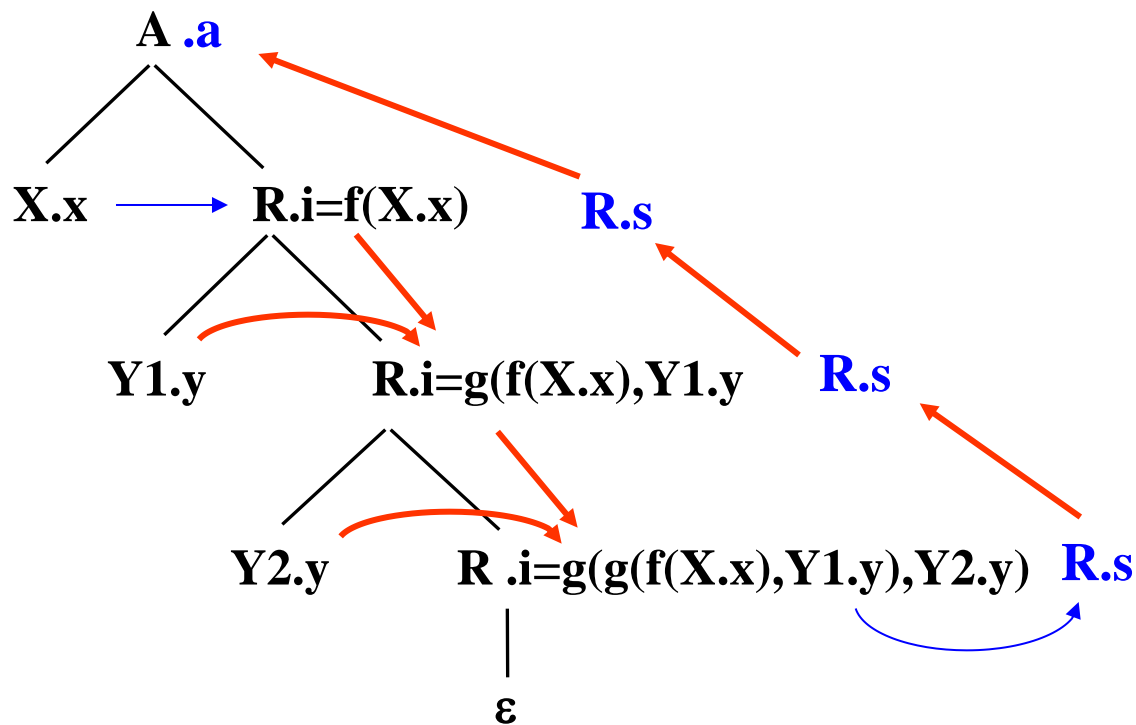
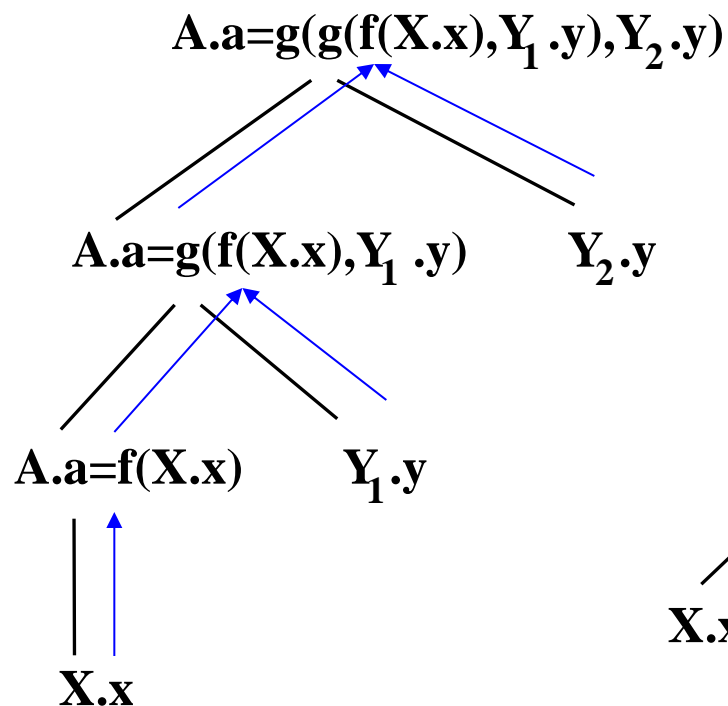
**R. i:** 表示在R之前已经扫描过的符号串的属性值

**R. s:** 表示在R完全展开为终结符号之后得到的符号串的属性值。

## 翻译方案转换为：

$$A \rightarrow X \quad \{R. i = f(X. x)\}$$
$$R \{A. a = R. s\}$$
$$R \rightarrow Y \quad \{R_i. i = g(R_i, Y, y)\}$$
$$R_1 \quad \{R. s = R_1. s\}$$
$$R \rightarrow_{\epsilon} \{R. s = R. i\}$$


# 计算属性的两种方法



## 二、预测翻译程序的设计

- 从翻译方案出发构造自顶向下的语法制导翻译程序

### 算法5.2：构造语法制导的预测翻译程序

输入：基础文法适合于预测分析的语法制导翻译方案

输出：语法制导翻译程序

方法：（修改预测分析程序的构造技术）

- (1) 为每个非终结符号A建立一个函数(可以是递归函数)
  - ◆ A的每一个继承属性对应函数的一个形参
  - ◆ A的综合属性作为函数的返回值
  - ◆ A产生式中的每个文法符号的每个属性都对应一个局部变量
- (2) A的函数的代码由多个分支组成

### (3) 与每个产生式相关的程序代码

- 按照从左到右的顺序考虑产生式右部的记号、非终结符号和语义动作
- 对带有综合属性 $x$ 的记号 $X$ 
  - 把属性 $x$ 的值保存于为 $X.x$ 声明的变量中
  - 产生一个匹配记号 $X$ 的调用
  - 推进扫描指针
- 对非终结符号 $B$ 
  - 产生一个函数调用语句 $c=B(b_1, b_2, \dots, b_k)$
  - $b_i (i=1, 2, \dots, k)$ 是对应于 $B$ 的继承属性的变量
  - $c$ 是对应于 $B$ 的综合属性的变量
- 对每一个语义动作
  - 把动作代码复制到分析程序中
  - 用代表属性的变量代替翻译方案中引用的属性

# 例：为简单表达式求值的翻译方案 构造翻译程序

- 为每个非终结符号构造一个函数

**void L(void)**

**int E(void)**

**int M(int in)**

**int T(void)**

**int N(int in)**

**int F(void)**

# 分别与 $E \rightarrow TM$ 、 $M \rightarrow +TM | \varepsilon$ 相应的分析过程

**// $E \rightarrow TM$**

```
void proc_E(void) {  
    proc_T();  
    proc_M();  
};
```

**// $M \rightarrow +TM | \varepsilon$**

```
void proc_M(void) {  
    if (lookahead == '+')  
    {  
        match( '+' );  
        proc_T();  
        proc_M();  
    }  
};
```

# 实现翻译方案的函数

$E \rightarrow T \{ M.i = T.val \}$   
 $M \{ E.val = M.s \}$

```
int E(void) {  
    int eval, tval, mi, ms;  
    tval=T();  
    mi=tval;  
    ms=M(mi);  
    eval=ms;  
    return eval;  
}
```



# 实现翻译方案的函数

```
int M(int in) {  
    int tval, i1, s1, s;  
    char addoplexeme;  
    if (lookahead == '+') { // 产生式  $M \rightarrow +TM$   
        addoplexeme = lexval;  
        match( '+' );  
        tval = T();  
        i1 = in + tval;  
        s1 = M(i1);  
        s = s1;  
    };  
    else s = in; // 产生式  $M \rightarrow \epsilon$   
    return s  
}
```

$M \rightarrow +$

$T \{ M_1.i = M.i + T.val \}$

$M_1 \{ M.s = M_1.s \}$

$M \rightarrow \epsilon \{ M.s = M.i \}$

## 5.4 L属性定义的自底向上翻译

- 在自底向上的分析过程中实现L属性定义的翻译
- 可以实现任何基于LL(1)文法的L属性定义
- 可以实现许多（不是全部）基于LR(1)文法的L属性定义

- 一、从翻译方案中去掉嵌入的动作
- 二、分析栈中的继承属性
- 三、模拟继承属性的计算
- 四、用综合属性代替继承属性

# 一、从翻译方案中去掉嵌入的动作

- 自底向上地处理继承属性
- 等价变换：  
使所有嵌入的动作都出现在产生式的右端末尾
- 方法：
  - ◆ 在基础文法中引入新的产生式，形如： $M \rightarrow \epsilon$
  - ◆  $M$ ：标记非终结符号，用来代替嵌入在产生式中的动作
  - ◆ 把被 $M$ 替代的动作放在产生式 $M \rightarrow \epsilon$ 的末尾

例：去掉如下翻译方案中嵌入的动作：

$$E \rightarrow TR$$
$$R \rightarrow +T \{\text{print}(' + ')\} R \mid -T \{\text{print}(' - ')\} R \mid \varepsilon$$
$$T \rightarrow \text{num} \{\text{print}(\text{num.val})\}$$

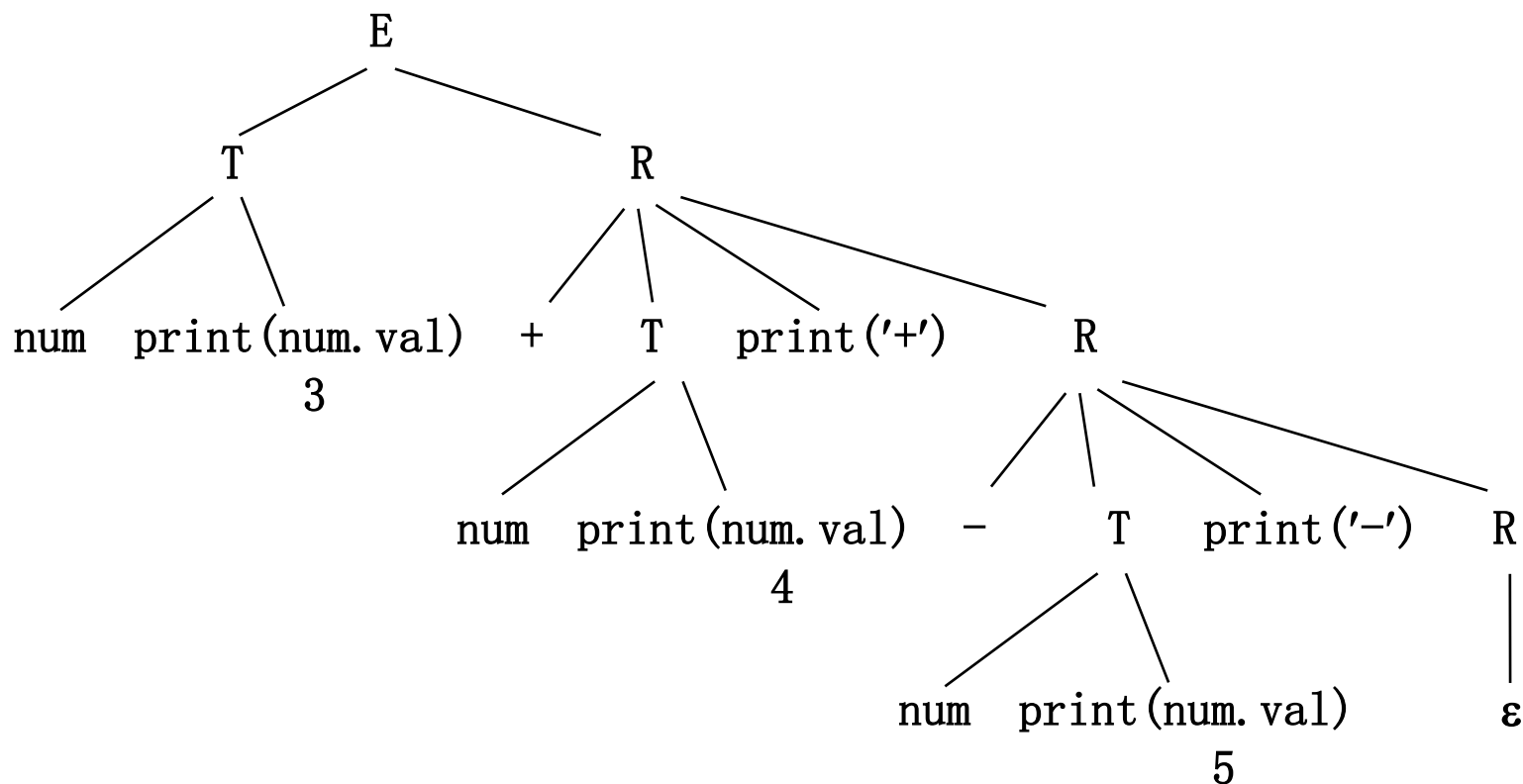
- 标记非终结符号 **M** 和 **N**，及产生式  $M \rightarrow \varepsilon$  和  $N \rightarrow \varepsilon$
- 用 **M** 和 **N** 替换出现在 **R** 产生式中的动作

### ■ 新的翻译方案

$$E \rightarrow TR$$
$$R \rightarrow +TMR \mid -TNR \mid \varepsilon$$
$$T \rightarrow \text{num} \{\text{print}(\text{num.val})\}$$
$$M \rightarrow \varepsilon \{\text{print}(' + ')\}$$
$$N \rightarrow \varepsilon \{\text{print}(' - ')\}$$

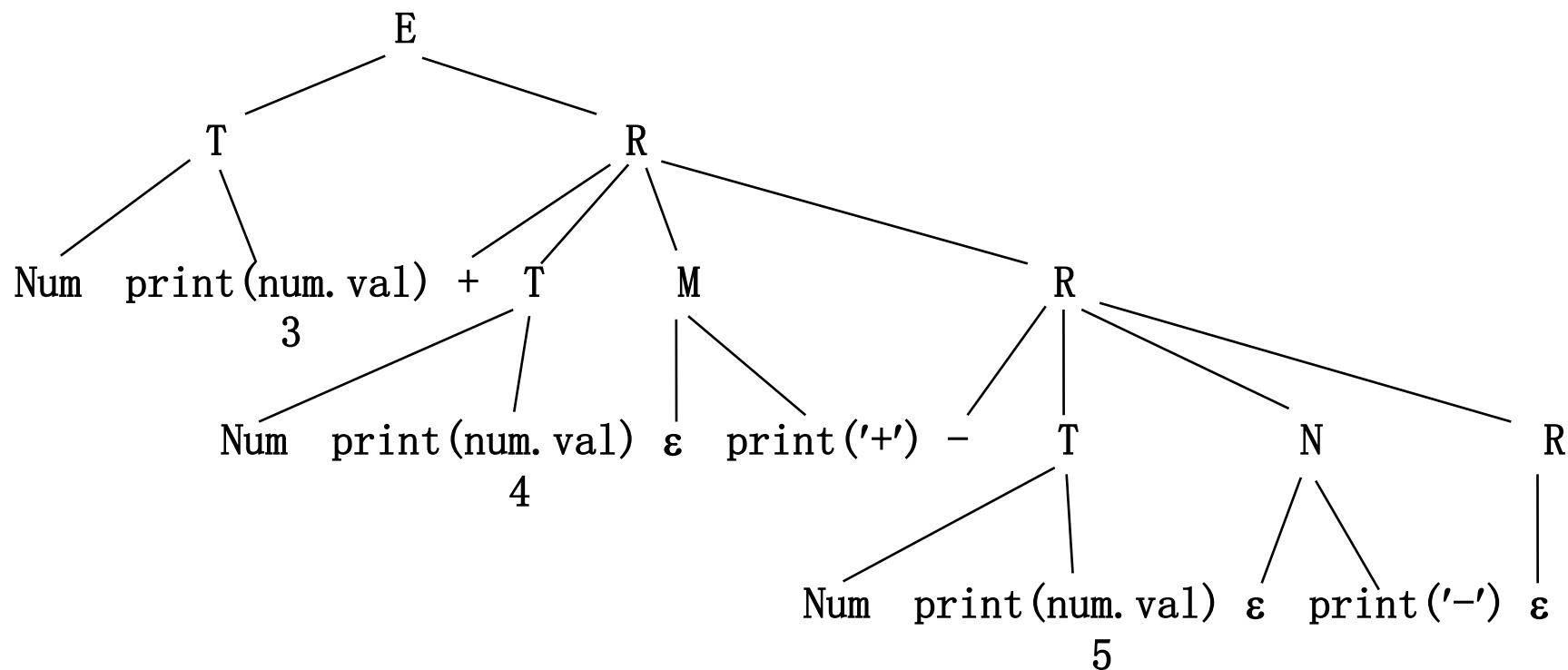
# 变换前、后的翻译方案是等价的

变换前，表达式 $3+4-5$ 的分析树：



- 深度优先的顺序进行遍历
- `print(num1.val) print(num2.val) print('+') print(num3.val) print('-')`
- 动作执行的结果是：34+5-

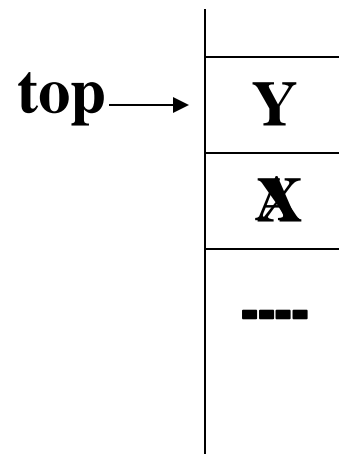
## 变换后，表达式3+4-5的分析树：



- 深度优先的顺序进行遍历
- `print(num1.val) print(num2.val) print('+') print(num3.val) print('-')`
- 动作执行的结果是：34+5-

## 二、分析栈中的继承属性

- LR分析程序对产生式 $A \rightarrow XY$ 的归约
- 考虑分析过程中属性的计算



$Y_k$	$Y_k \cdot y$
$X_n$	$X_n \cdot x$
$Y_2$	$Y_2 \cdot y$
$X_1$	$X_1 \cdot x$
...	
state	val

top →

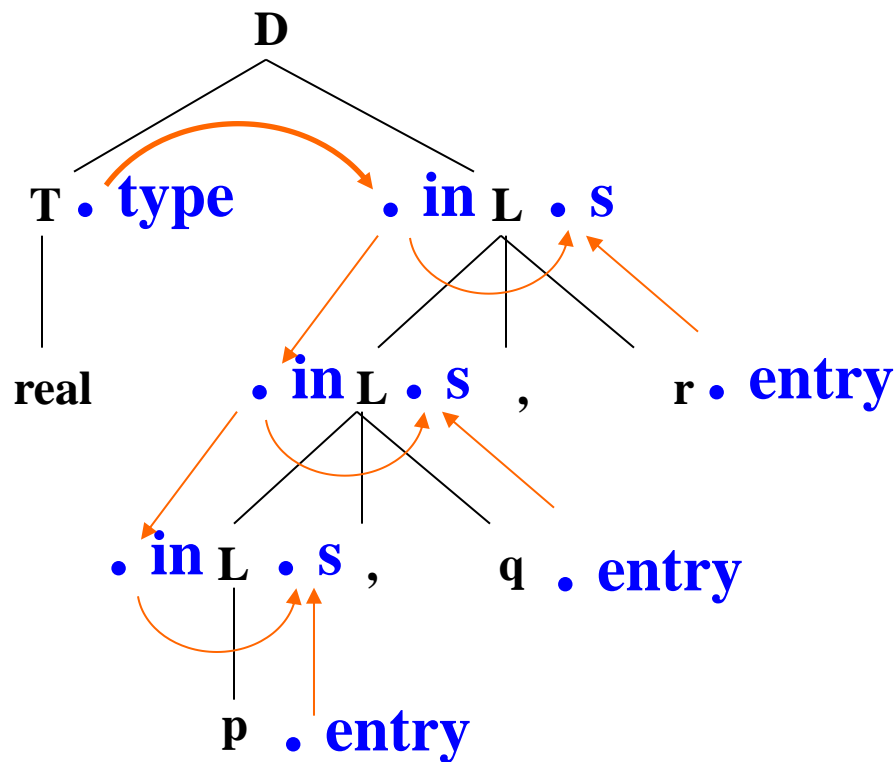
$$X \rightarrow X_1 X_2 \cdots X_n$$

$$Y \rightarrow Y_1 Y_2 \cdots Y_k$$

$$Y.i = X.s$$

# 复制规则的重要作用

输入符号串: **real p,q,r**



翻译方案:

$D \rightarrow T \{ \mathbf{L.in = T.type} \}$

**L**

$T \rightarrow \text{int} \{ T.type = \text{integer} \}$

$T \rightarrow \text{real} \{ T.type = \text{real} \}$

$L \rightarrow \{ \mathbf{L_1.in = L.in} \}$

$L_1, \text{id} \{ \text{addtype}(\text{id.entry}, L.in) \}$

$L \rightarrow \text{id} \{ \text{addtype}(\text{id.entry}, L.in) \}$



# 例：应用继承属性，用复制规则传递标识符的类型

输入	栈	分析动作
real p,q,r\$	state: val:	移进
p,q,r\$	state: real val: real	归约，用 $T \rightarrow \text{real}$
p,q,r\$	state: T val: real	移进
,q,r\$	state: T val: real	归约，用 $L \rightarrow \text{id}$
,q,r\$	state: T val: real	移进
q,r\$	state: T val: real	移进
,r\$	state: T val: real	归约，用 $L \rightarrow L, \text{id}$
,r\$	state: T val: real	移进
r\$	state: T val: real	移进
\$	state: T val: real	归约，用 $L \rightarrow L, \text{id}$
\$	state: T val: real	归约，用 $D \rightarrow TL$
\$	state: D val: -	接受

# 计算属性值的代码段

产生式	代码段
$D \rightarrow TL$	
$T \rightarrow \text{int}$	<code>val[ntop]=integer</code>
$T \rightarrow \text{real}$	<code>val[ntop]=real</code>
$L \rightarrow L, id$	<code>addtype(val[top], val[top-3])</code>
$L \rightarrow id$	<code>addtype(val[top], val[top-1])</code>

- **top**和**ntop**分别是归约前和归约后的栈顶指针
- 当用产生式 $L \rightarrow id$ 归约时，**L.in** 的位置？
- 当用产生式 $L \rightarrow L, id$ 进行归约时，**L.in** 的位置？
- 和**L.in**有关的动作 ？

### 三、模拟继承属性的计算

- 要想从栈中取得继承属性，**当且仅当**文法允许属性值在栈中存放的位置可以预测。

例：属性值在栈中的位置不可预测的语法制导定义

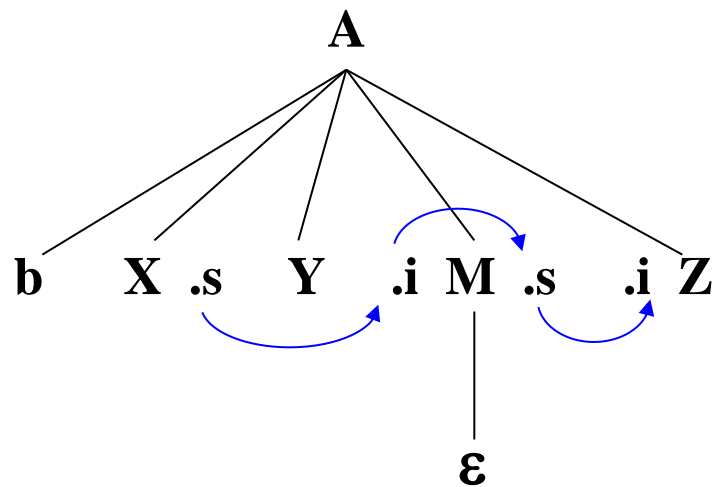
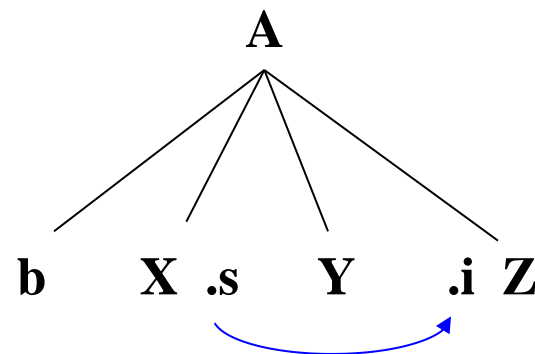
	产生式	语义规则
(1)	$A \rightarrow aXZ$	$Z.i = X.s$
(2)	$A \rightarrow bXYZ$	$Z.i = X.s$
(3)	$X \rightarrow x$	$X.s = 5$
(4)	$Y \rightarrow y$	$Y.s = 7$
(5)	$Z \rightarrow z$	$Z.s = g(Z.i)$

当用 $Z \rightarrow z$ 进行归约时，  
 $Z.i$ 可能在 $val[top-1]$ 处  
也可能在 $val[top-2]$ 处

# 模拟继承属性的计算

- 引入**标记非终结符号**，对原语法制导定义进行等价变换

	产生式	语义规则
(1)	$A \rightarrow aXZ$	$Z.i = X.s$
(2')	$A \rightarrow bXYMZ$	$M.i = X.s;$ $Z.i = M.s$
(3)	$X \rightarrow x$	$X.s = 5$
(4)	$Y \rightarrow y$	$Y.s = 7$
(5)	$Z \rightarrow z$	$Z.s = g(Z.i)$
(6)	$M \rightarrow \varepsilon$	$M.s = M.i$



# 用标记非终结符号模拟 非复制规则的语义规则

例：考虑如下的产生式及语义规则：

$A \rightarrow aXY$        $Y.i = f(X.s)$

$Y \rightarrow y$        $Y.s = g(Y.i)$

---	a	X	y
---		X.s	

↑  
top

## ■ 引入标记非终结符号N

$A \rightarrow aXNY$        $N.i = X.s; Y.i = N.s$

$N \rightarrow \epsilon$        $N.s = f(N.i)$

$Y \rightarrow y$        $Y.s = g(Y.i)$

---	a	X	N	y
---		X.s	N.s	

↑  
top

- 所有继承属性均由复制规则实现
- 继承属性在栈中的位置可以预测

## 算法5.3: L属性定义的自底向上分析和翻译

输入: 基础文法是LL(1)文法的L属性定义

输出: 在分析过程中计算所有属性值的分析程序

方法:

假设:

每个非终结符号A都有一个继承属性A.i

每一个文法符号X都有一个综合属性X.s

(1) 对每个产生式 $A \rightarrow X_1 X_2 \dots X_n$

引入n个新的标记非终结符号 $M_1, M_2, \dots, M_n$

用产生式 $A \rightarrow M_1 X_1 M_2 X_2 \dots M_n X_n$ 代替原来的产生式

$X_j$ 的继承属性与标记非终结符号 $M_j$ 相联系

属性 $X_j.i$ (也就是 $M_j.s$ )总是在 $M_j$ 处计算, 且发生在开始做归约到 $X_j$ 的动作之前。

(2) 在自底向上分析过程中，各个属性的值都可以被计算出来

### 第一种情况：用 $M_j \rightarrow \varepsilon$ 进行归约

#### ■ 已知：

- ◆ 每个标记非终结符号在文法中是唯一的
- ◆  $M_j$ 属于哪个形式为 $A \rightarrow M_1 X_1 M_2 X_2 \cdots M_n X_n$ 的产生式
- ◆ 计算属性 $X_j. i$ 需要哪些属性、以及它们的位置

state	...		$M_1$	$X_1$	$M_2$	$X_2$	...	$M_{j-1}$	$X_{j-1}$	$M_j$
val	...	$A. i$	$X_1. i$	$X_1. s$	$X_2. i$	$X_2. s$	...	$X_{j-1}. i$	$X_{j-1}. s$	$M_j. s$

Diagram illustrating the state and value stack for the reduction  $M_j \rightarrow \varepsilon$ . The stack contains states and values for non-terminals  $M_1, X_1, M_2, X_2, \dots, M_{j-1}, X_{j-1}, M_j$ . The value field contains the corresponding attributes  $i$  and  $s$ .

Arrows indicate the positions of the attributes used to compute  $X_j. i$  (the value of  $X_j$  at position  $i$ ):

- $top-2(j-1)$  points to  $A. i$
- $top-2(j-1)+1$  points to  $X_1. i$
- $top-2(j-1)+2$  points to  $X_1. s$
- $top-2(j-2)+1$  points to  $X_2. i$
- $top-2(j-2)+2$  points to  $X_2. s$
- $top-1$  points to  $X_{j-1}. i$
- $top$  points to  $X_{j-1}. s$
- $ntop$  points to  $M_j. s$

## 第二种情况：用 $A \rightarrow M_1 X_1 M_2 X_2 \cdots M_n X_n$ 进行归约

### ■ 已知：

- ◆ A. i的值、及其位置
- ◆ 计算A. s所需要的属性值均已在栈中已知的位置  
即各有关 $X_j$ 的位置上

state	...		$M_1$	$X_1$	$M_2$	$X_2$	...	$M_n$	$X_n$
val	...	A.i	$X_1.i$	$X_1.s$	$X_2.i$	$X_2.s$	...	$X_n.i$	$X_n.s$

$\uparrow$                        $\uparrow$                        $\uparrow$                        $\uparrow$

**top-2n**      **top-2n+2**      **top-2n+4**                      **top**

state	...		A
val	...	A.i	A.s

$\uparrow$                        $\uparrow$

**top-1**      **top**



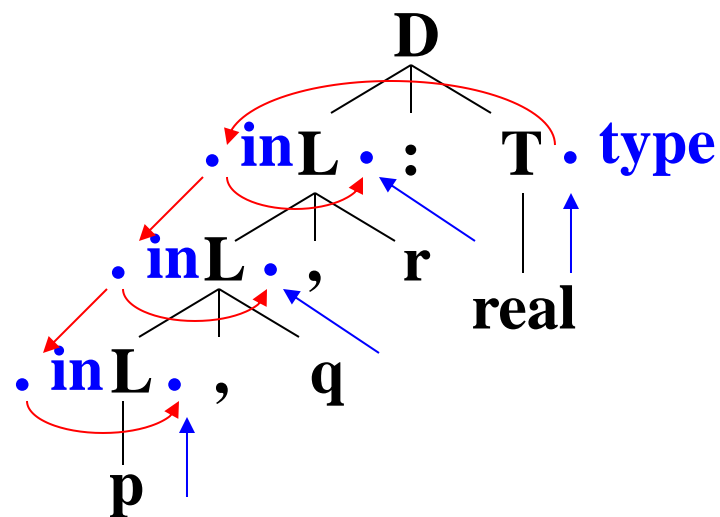
## 四、用综合属性代替继承属性

例：PASCAL的变量声明语句可由如下文法产生：

$D \rightarrow L:T$

$T \rightarrow \text{integer} | \text{real}$

$L \rightarrow L, \text{id} | \text{id}$



### ■ 问题：

标识符由 $L$ 产生，而类型不在 $L$ 的子树中

归约从左向右进行，类型信息从右向左传递

只用综合属性不能使类型和标识符联系在一起

# 解决方法

- 改写文法，使类型作为标识符表的最后一个元素

$D \rightarrow idL$

$L \rightarrow ,idL \mid :T$

$T \rightarrow integer \mid real$

- 改写后：

归约从右向左进行，类型信息从右向左传递

仅用综合属性即可把类型信息和标识符联系起来

$D \rightarrow idL$

$addtype(id.entry, L.type)$

$L \rightarrow ,idL_1$

$L.type = L_1.type;$

$addtype(id.entry, L_1.type)$

$L \rightarrow :T$

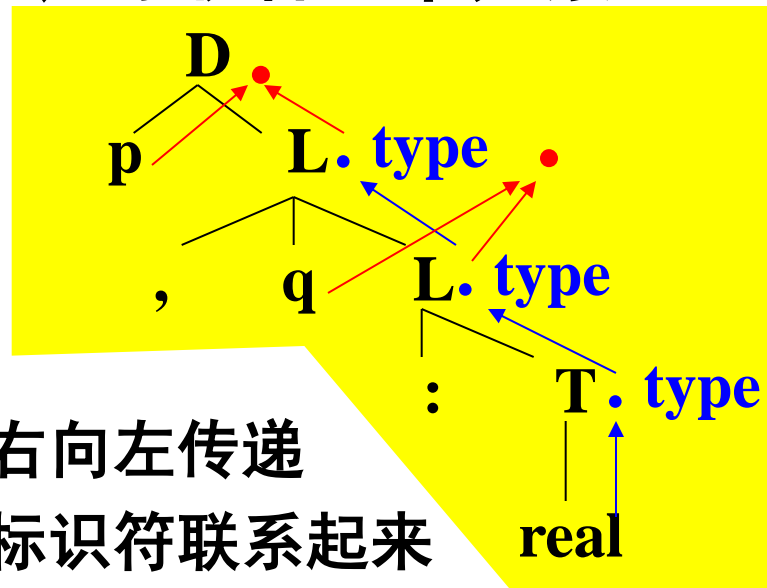
$L.type = T.type$

$T \rightarrow integer$

$T.type = integer$

$T \rightarrow real$

$T.type = real$



# 小 结

- 综合属性、继承属性
- 语法制导定义
  - ◆ S-属性定义
  - ◆ L-属性定义（继承属性应满足的**限制条件**）
- 翻译方案
  - ◆ 构造S-属性定义的翻译方案（语义动作放在产生式右尾）
  - ◆ 构造L-属性定义的翻译方案（语义动作插入**产生式之中**）
- S-属性定义的自底向上翻译（**改造LR分析程序**）
  - ◆ 分析栈的扩充
  - ◆ 分析控制程序的修改

# 小结 (续)

作业:

5.6

5.9

5.11

- L-属性定义的预测翻译
  - ◆ 为每个非终结符号构造递归函数
  - ◆ 形参、返回值、局部变量
  - ◆ 函数体、分支程序
  - ◆ 分支程序段的设计
- L-属性定义的自底向上翻译
  - ◆ 通过复制规则引用继承属性
  - ◆ 引入标记非终结符号及相应的产生式
  - ◆ 修改语义动作、使继承属性由复制规则实现

# 作业 (P. 172)

■ 5. 1

■ 5. 2

■ 5. 3

■ 5. 4

■ 5. 6

■ 5. 9

■ 5. 11

■ 5. 15

■ 5. 16

# 作业 5.15

有如下文法：

$$S \rightarrow (L) \mid a$$

$$L \rightarrow L, S \mid S$$

- (1) 设计一个语法制导定义，它输出配对的括号个数。
- (2) 构造一个翻译方案，它输出每个a的嵌套深度。

如对句子  $(a, (a, a))$  的输出结果是 1, 2, 2。

# 作业 5.16

令综合属性val给出在下面的文法中S产生的二进制数的值，如对于输入101.101：S.val=5.625

$$S \rightarrow L.L \mid L$$

$$L \rightarrow LB \mid B$$

$$B \rightarrow 0 \mid 1$$

请写出确定S.val值的语法制导定义。

# 第6章 语义分析



*LI Wensheng, SCST, BUPT*

知识点：符号表

类型体制

各语法成分的类型检查



# 语义分析

- 6. 1 语义分析的任务和地位
- 6. 2 符号表
- 6. 3 符号表的建立
- 6. 4 类型检查
- 6. 5 一个简单类型检查程序的说明
- 6. 6 类型检查有关的其他主题
- 小 结

## 6.1 语义分析的任务和地位

- 程序设计语言的结构由上下文无关文法来描述
- 程序结构正确与否与该结构的上下文有关，如：
  - 变量的作用域问题
  - 同一作用域内同名变量的重复声明问题
  - 表达式、赋值语句中的操作数的类型一致性问题
- 思考：
  - 设计上下文有关文法来描述语言中上下文有关的结构？
  - 理论上可行，构造有困难，构造相应的分析程序更困难
- 解决办法：
  - 设计专门的语义动作补充上下文无关文法的分析程序
  - 利用语法制导翻译技术实现语义分析

# 上下文有关信息的记录与使用

## ■ 符号表

- 记录编译过程中识别出的上下文有关的信息，如：
- 变量的类型
- 相对地址

## ■ 信息的引用

- 根据词法分析程序识别出的标识符的属性值（标识符在符号表中的入口），查找符号表中对应该标识符的记录，从而可以取得该标识符有关的信息。
- 如果编译的程序块处于该变量的作用域内，则这个变量将一直保留在符号表中

# 语义检查

- **动态检查**：目标程序运行时进行的检查
- **静态检查**：读入源程序、但不执行源程序的情况下进行的检查
  - **类型检查**
    - 对访问数据的操作和被访问数据的类型进行检查，检查操作的合法性和数据类型的相容性。
  - **控制流检查**
    - 检查控制语句是否使控制转移到一个合法的位置。
  - **唯一性检查**
    - 一个标识符在同一程序块中必须而且只能被说明一次
    - CASE语句中用于匹配选择表达式的常量必须各不相同
    - 枚举类型定义中的各元素不允许重复
  - **关联名字的检查**

# 类型检查

- 由类型检查程序完成
- 检验结构的类型是否和它的上下文所期望的一致，如：
  - 表达式中各运算对象的类型
  - 算术运算符 `mod` 的运算对象的类型
  - 用户定义函数的各参数类型、返回值类型

# 语义分析程序的作用和地位

## ■ 语义分析程序的作用

- 符号表的建立和管理
- 类型检查

## ■ 语义分析程序的地位



## ■ 生成目标代码时会用到语义分析的结果

- 重载运算符：一个运算符在不同的上下文中表示不同的运算
- 类型强制：编译程序把运算对象变换为上下文所期望的类型

## 6.2 符号表

- 符号表在翻译过程中起两方面的重要作用：
  - 检查语义（即上下文有关）的正确性
  - 辅助正确地生成代码
- 通过在符号表中插入和检索变量的属性来实现的
- 符号表是一张动态表
  - 在编译期间符号表的入口不断地增加
  - 在某些情况下又在不断地删除
- 编译程序需要频繁地与符号表进行交互，符号表的效率直接影响编译程序的效率。

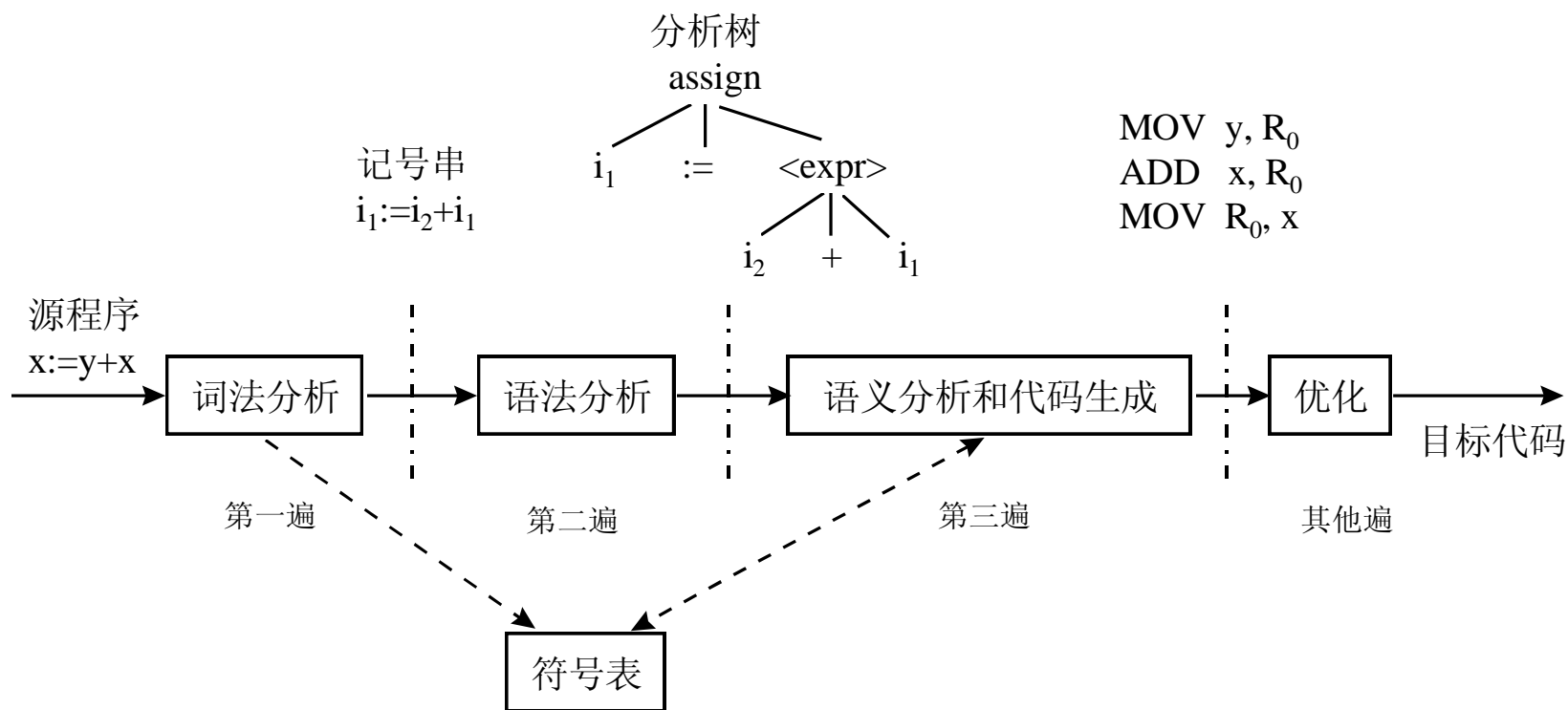
# 符号表

- 一、符号表的建立和访问时机
- 二、符号表内容
- 三、符号表操作
- 四、符号表组织



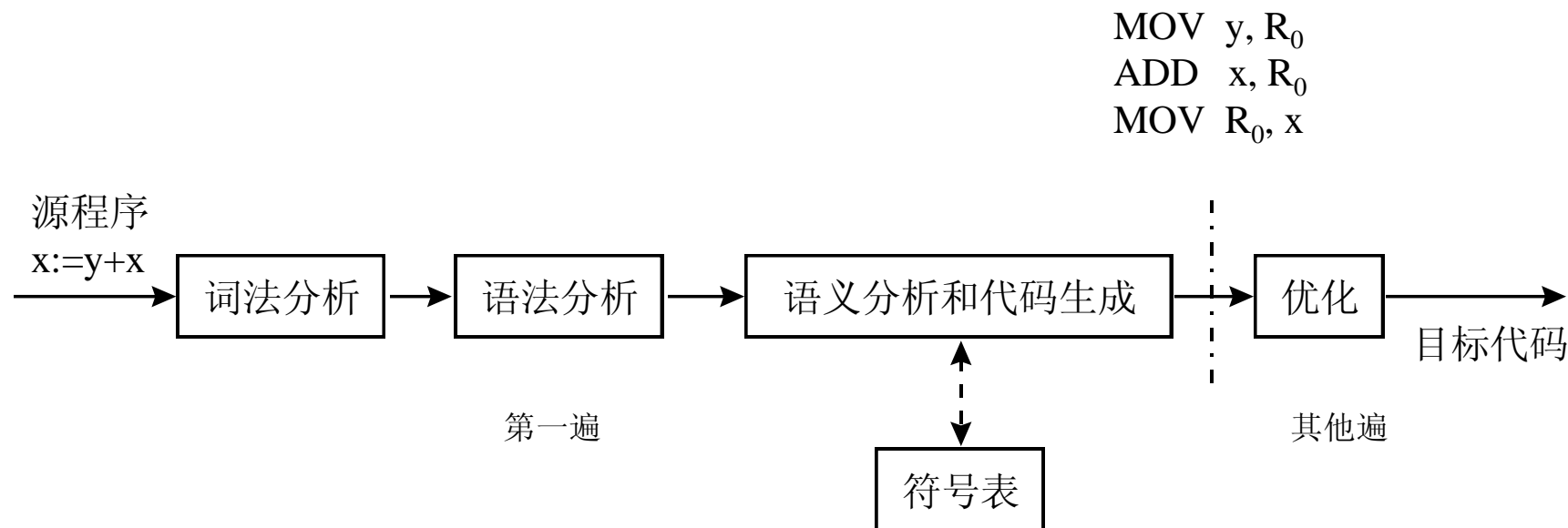
# 一、符号表的建立和访问时机

## 1. 多遍编译程序



变量在符号表中的位置作为词法分析程序产生的记号的属性

## 2. 合并遍的编译程序



### ■ 两方面的优点:

- 对语法分析程序来讲降低了文法的复杂性
- 允许用更系统的方法对上下文有关的错误进行检测和校正。

## 二、符号表内容

- 符号表中记录的是和标识符相关的属性
- 出现在符号表中的属性种类，在一定程度上取决于程序设计语言的性质。
- 符号表的典型形式：

	变量名	目标地址	类型	维数	声明行	引用行	指针
1	count	0	2	1	2	9, 14, 15	7
2	x_total	4	1	0	3	12, 14	0
3	form	8	3	2	4	36, 37, 38	6
4	b_loop	48	1	0	5	10, 11, 13	1
5	able_n	52	1	0	5	11, 23, 25	4
6	mlist	56	6	0	6	17, 21	2
7	flag	64	1	0	7	28, 29	3

# 变量名

- 变量名必须常驻内存

- 问题：

- 标识符长度是**可变的**

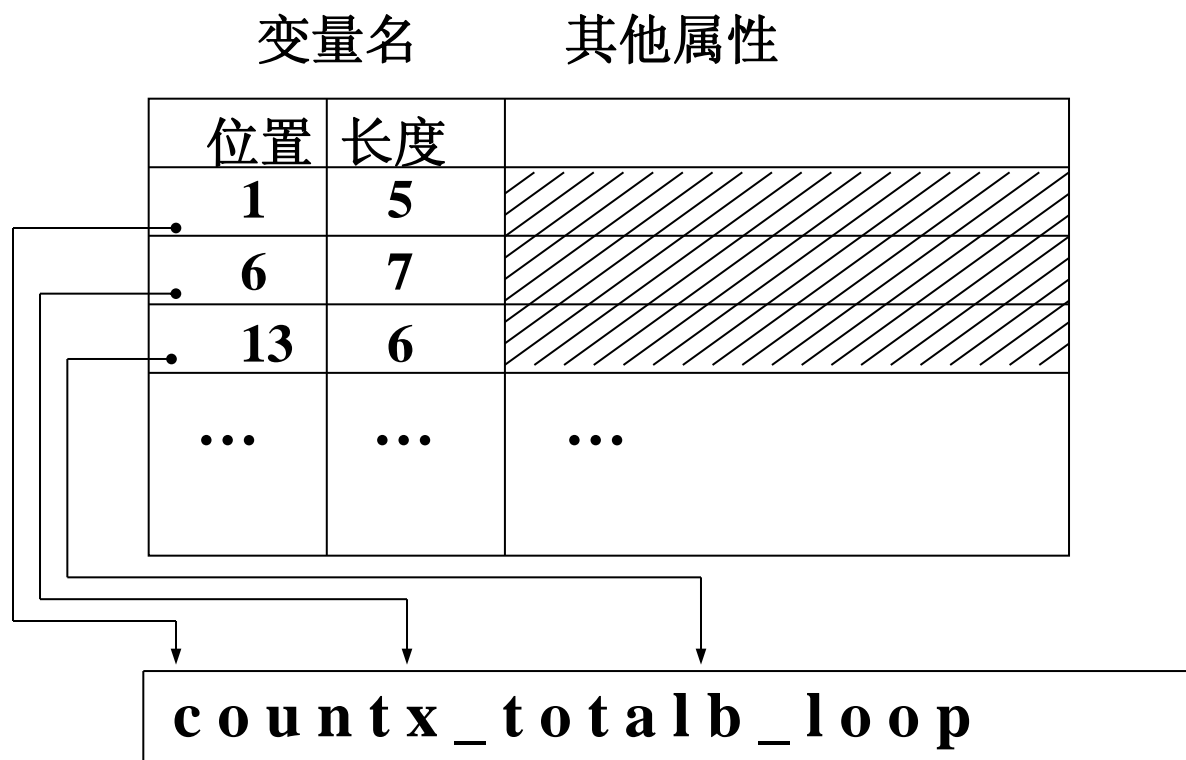
- 解决办法：

- 标识符长度有限制：设置一个长度固定的域，它的长度为该语言允许的标识符最大长度。
- 标识符长度没有限制：设置一个长度固定的域，域内存放一个串描述符，包含位置指针和长度两个子域，指针域指示该标识符在总的串区内的开始位置，长度域记录该标识符中的字符数。

存取速度较高，  
存储空间利用率较低

存取速度较慢，  
节省存储空间。

# 使用串描述符表示变量



# 目标地址

- 指示运行时变量值存放的相对位置
- 对于静态存储分配的语言（如Fortran），目标地址按连续的顺序分配，从0开始到m（m是分配给一个程序的数据区的最大值）。
- 对于块结构的语言（如Pascal），通常采用二元地址 $\langle BL, NO \rangle$ 
  - BL：块的嵌套深度，用于确定分配给声明变量的块的数据区的基址。
  - NO：变量的目标地址偏移量，指示该变量的存储单元在数据区中相对于基址的位置。

# 数据类型

- 当所编译的语言有数据类型（隐式或显式的）时，必须把类型属性存放到符号表中。
- 对于无类型的语言，可删除该域。
- 变量的类型属性用于：
  - 类型检查
  - 生成代码
  - 空间分配
- 变量的类型以一种编码形式存放在符号表中。

# 数组的维数/参数的个数

- 数组引用时，其维数应当与数组声明时定义的维数一致。
  - 类型检查阶段必须对这种一致性（维数、每维的长度）进行检查
  - 维数用于数组元素地址的计算。
- 过程调用时，实参必须与形参一致。
  - 实参的个数与形参的个数一致
  - 实参的类型与相应形参的类型一致
- 在符号表组织中：
  - 把参数的个数看作它的维数是很方便的，因此，可将这两个属性合并成一个。
  - 这种方法也是协调的，因为对这两种属性所做的类型检查是类似的。



# 交叉引用表

- 编译程序可以提供的—个十分重要的程序设计辅助工具：交叉引用表
- 编译程序—般设—个选项，用户可以选择是否生成交叉引用表

变量名	类型	维数	声明行	引用行
able_n	1	0	5	11, 23, 25
b_loop	1	0	5	10, 11, 13
count	2	1	2	9, 14, 15
flag	1	0	7	28, 29
form	3	2	4	36, 37, 38
mlist	6	0	6	17, 21
x_total	1	0	3	12, 14

# 链域

- 为了便于产生按字母顺序排列的交叉引用表
- 如果编译程序不产生交叉引用表，则链域以及语句的行号属性都可以从符号表中删除。

# 三、符号表操作

- 最常执行的操作：插入和检索
- 要求变量显式声明的强类型语言：
  - 编译程序在处理声明语句时调用两种操作
    - 检索：查重、确定新表目的位置
    - 插入：建立新的表目
  - 在程序中引用变量名时，调用检索操作
    - 查找信息，进行语义分析、代码生成
    - 可以发现未定义的名字
- 允许变量隐式声明的语言：
  - 标识符的每次出现都按首次出现处理
  - 检索：
    - 已经声明，进行类型检查，...
    - 首次出现，插入操作，从其作用推测出该变量的全部属性

# 定位和重定位操作

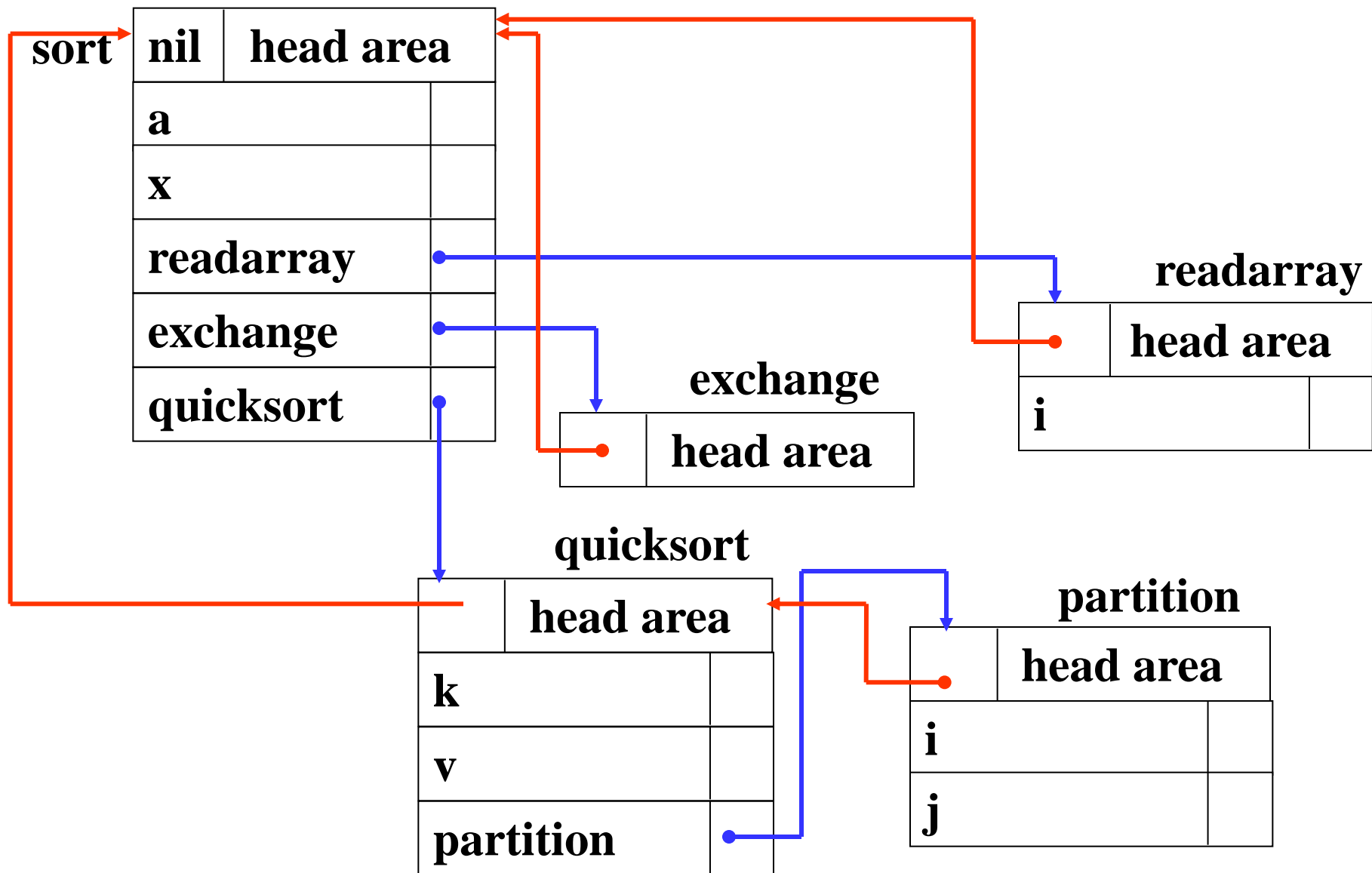
- 对于块结构的语言，在建立和删除符号表时还要使用两种附加的操作，即定位和重定位。
- 当编译程序识别出块的开始时，执行定位操作。
- 当编译程序遇到块的结束时，执行重定位操作。
- 定位操作：
  - 建立一个新的子表（包含于符号表中），在该块中声明的所有变量的属性都存放在此子表中。
- 重定位操作：
  - “删除”存放该块中局部变量的子表
  - 这些变量的作用域局部于该块，出了该块后这些变量不能再被引用。

# 读入数据，并进行排序的PASCAL程序

```
program sort (input,output);  
  var a : array[0..10] of integer;  
      x : integer;  
  procedure readarray;  
    var i : integer;  
  begin  
    for i:=1 to 9 do read(a[i])  
  end;  
  procedure exchange (i,j:integer)  
  begin  
    x:=a[i]; a[i]:=a[j]; a[j]:=x  
  end;
```

```
peocedure quicksort (m,n:integer);  
  var k,v : integer;  
  function partition (y,z :integer):integer;  
    var i,j : integer;  
  begin  
    ...a...;    ...v...;  
    exchange(i,j);    .....  
  end;  
  begin  
    .....  
    k=partition(m,n);  
    quicksort(m,k-1);  
    quicksort(k+1,n);    .....  
  end;{quicksort}  
begin readarray; quicksort(1,9)  
end. {sort }
```

# 符号表的逻辑结构



## 四、符号表组织

1. 非块结构语言的符号表组织
2. 块结构语言的符号表组织

# 1. 非块结构语言的符号表组织

## ■ 非块结构语言：

- 编写的每一个可独立编译的程序单元是一个不含子模块的单一模块
- 模块中声明的所有变量属于整个模块

## ■ 符号表组织

### – 无序线性表

- 属性记录按变量声明/出现的先后顺序填入表中
- 插入前都要进行检索，若发现同名变量
  - 对显式声明的语言：错误
  - 对隐式声明的语言：引用
- 适用于程序中出现的变量很少的情况



## – 有序线性表

- 按字母顺序对变量名排序的表
- 避免了对整个表的查找
- 线性查找：
  - 当遇到第一个比查找变量名值大的项目时，就可以判定该变量名不在表中了。
  - 当执行插入操作时，要增加额外的比较和移动操作。
  - 若使用单链结构表的话，可省去表记录的移动，但需要在每个表记录中增加一个链接字段。
- 折半查找：
  - 首先把变量名与中间项进行比较，结果或是找到该变量名，或是指出下一次要在哪半张表中进行。
  - 重复此过程，直到找到该变量名或确定该变量名不在表中为止。

## – 散列表

- 查找时间与表中记录数无关的一种符号表组织方式



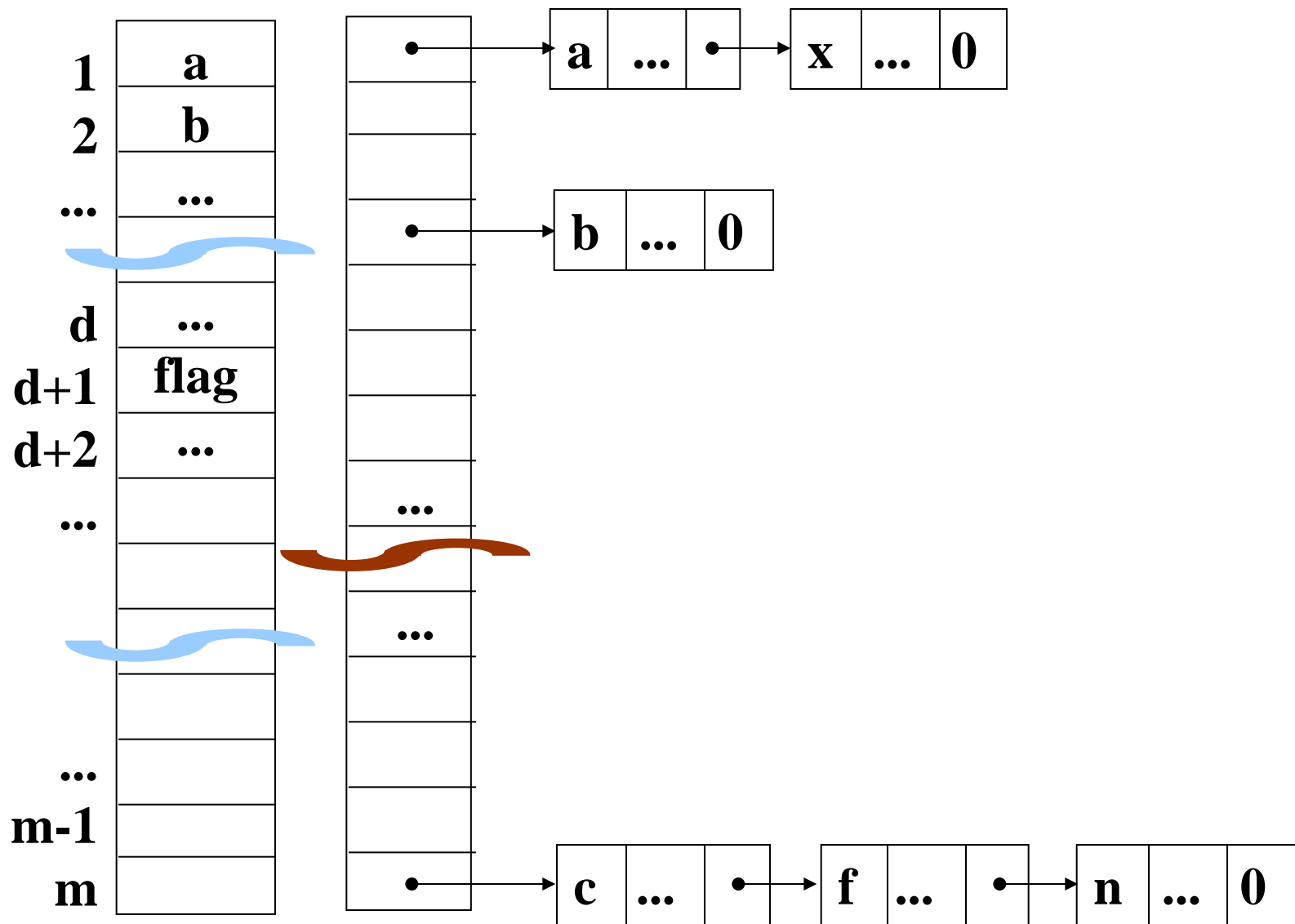
# 散列函数H

- **除法：**最常用的散列函数， $H(x) = (x \bmod m) + 1$ ，通常 $m$ 为一个大的素数，这样可使标识符尽可能均匀地分散在表中
- **中平方散列法：**先将标识符进行平方运算，然后将结果的两头几个值或数字去掉，直到剩下的位数或数字等于所期望的地址，必须对所有的标识符的平方值进行同样的处理。
- **折叠法：**标识符被分成若干段，可能除最后一段外，每一段与所需地址具有同样长度，然后各段加起来，忽略最后的进位，以构成一个地址。
- **长度相关法：**变量名的长度和名字的某个部分一起用来直接产生一个表地址，或更普遍的方法是产生一个有用的中间字，然后用除法产生一个最终的表地址。

# 解决冲突的方法

- 冲突：变量名被映射到一个存储单元d中，而这个单元已被占用
- 开放地址法
  - 按照顺序d, d+1, ..., m, 1, 2, ..., d-1进行扫描，直到找到一个空闲的存储单元为止，或者在扫描完m个单元之后搜索停止
  - 在查找一个记录时，按同样的顺序扫描，或找到要找的记录、或找到一个空闲单元（从未使用过）为止
- 分离链表法
  - 将发生冲突的记录链到一个专门的溢出区，该溢出区与主区相分离
  - 为每一组冲突的记录设置一个链表，主区和溢出区的每一个记录都必须有一个链接字段。
  - 为节省存储空间，建立一个中间表（散列表），所有记录都存入溢出区，而主区（散列表）只有链域

# 开放地址、分离链表示意图



## 2. 块结构语言的符号表组织

### ■ 块结构语言：

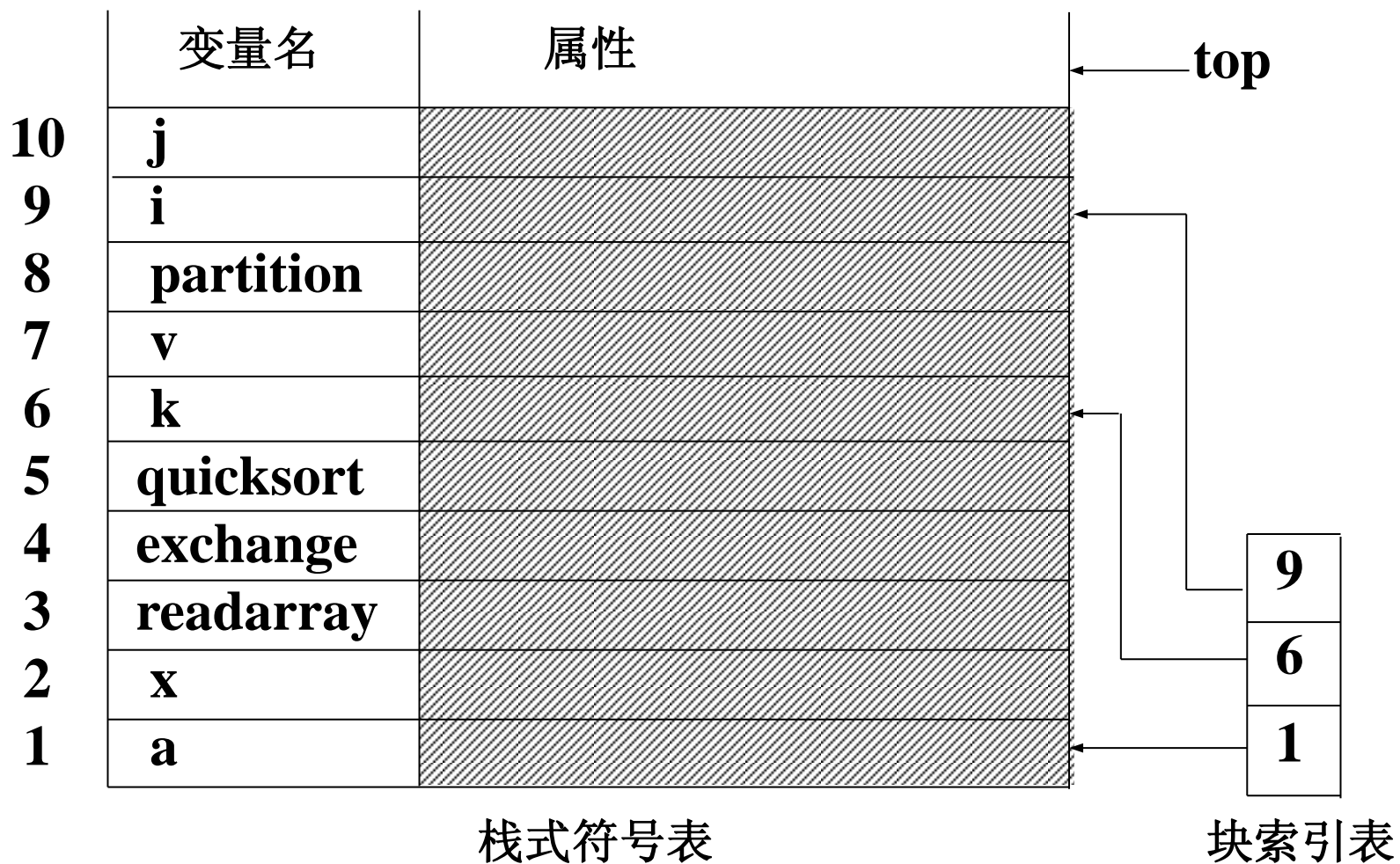
- 模块中可嵌套子块
- 子块中可以定义局部变量
- 每个块应有一个子表

### ■ 符号表组织

#### – 栈式符号表

- 当遇到变量声明时，将包含变量属性的记录入栈
- 当到达块结尾时，将该块中声明的所有变量的记录出栈
- 因为这些变量是局部于该块的，在块外已不可用的。

# 栈式符号表



# 操作

## ■ 插入

- 检查子表中是否有重名变量
  - 无，将新记录推入栈顶单元
  - 有，报告错误

## ■ 检索

- 从栈顶到栈底线性检索
  - 在当前子表中找到，局部变量
  - 在其他子表中找到，非局部名字
- 根据最近嵌套作用域原则，选择变量的记录

## ■ 定位

- 将下一个可用的栈顶单元的地址压入块索引表的顶端
- 块索引表的元素是指针，指向相应块的子表中第一个记录在栈中的位置

## ■ 重定位

- 有效地清除刚刚被编译完的块在栈式符号表中的所有记录
- 用块索引表顶端元素的值恢复栈顶指针top，完成重定位操作
- top指示符号表栈顶第一个空闲的记录存储单元。



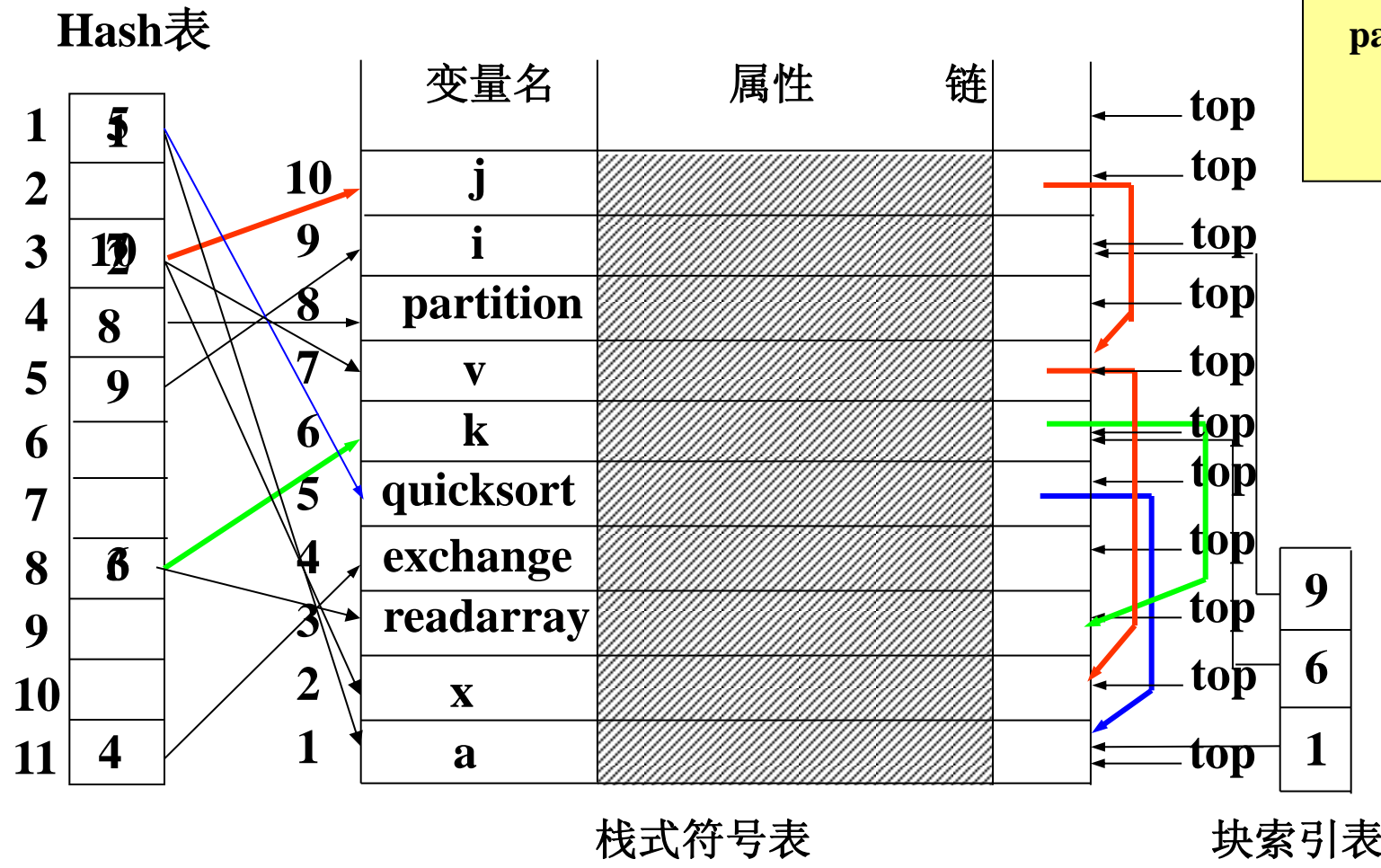
# 栈式散列符号表

- 假设散列表的大小为11，散列函数执行如下变换：

名字	映射到地址
<b>a、 quicksort</b>	<b>1</b>
<b>x、 v、 j</b>	<b>3</b>
<b>partition</b>	<b>4</b>
<b>i</b>	<b>5</b>
<b>k、 readarray</b>	<b>8</b>
<b>exchange</b>	<b>11</b>

# 栈式散列符号表示意图

a	1
x	3
readarray	8
exchange	11
quicksort	1
k	8
V	3
partition	4
i	5
j	3



# 操作

## ■ 插入

- 散列函数将变量名映射到散列表单元
  - 是否存在冲突？该表单元是否为空？
    - 无冲突：
      - 将栈指针的值记入该散列表单元
      - 将新记录推入栈顶
    - 有冲突
      - 检查冲突链中是否有同名变量
        - » 没有：将新记录插入冲突链的链头
        - » 有：检查同名变量是否属于当前子表
- 同名变量在栈中的位置  $\geq$  块索引表顶端元素的值？
- $\geq$ ：在当前子表中，报告错误
- $<$ ：不在当前子表中，将新记录插入冲突链的链头

# 名字引用时

## ■ 检索

- 散列函数将变量名映射到散列表单元
- 该散列表单元是否为空？
  - 空：名字未定义，报告错误
  - 不空：沿冲突链检索
    - 未找到：名字未定义，报告错误
    - 找到：名字在栈中的位置  $\geq$  块索引表顶端元素的值
      - $\geq$ ：局部名字
      - $<$ ：非局部名字

# 定位与重定位

## ■ 定位

- 当识别到一个新块时，要为之创建子表
- 将下一个可用的栈顶单元的地址压入块索引表的顶端
- 标识新块符号子表的开始位置

## ■ 重定位

- 当一个块编译完成时，它的有关记录必须逻辑上或物理上从符号表中除去。
- 用块索引表顶端单元的值确定要删除的栈单元
- 依次取出栈单元中的名字
  - 通过散列函数将该名字映射到散列表单元
  - 从链中把链头记录删除
  - 重复，直到新链头在栈中的位置 < 块索引表顶端单元的值
- 用块索引表顶端单元的值设置栈顶指针**TOP**

## 6.3 符号表的建立

- 处理声明语句时，编译程序的任务
  - 分离出每一个被声明的实体，并将它的名字填入符号表中
  - 尽可能多地将有关该实体的信息填入符号表
- 声明语句的形式
  - 类型关键字的位置
    - 标识符表的前面
    - 标识符表的后面
  - 标识符表
    - 单一实体
    - 多个同类型的实体
    - 不同种类的实体

# 源语言的说明

- 声明部分的文法

**$P \rightarrow D; S$**

**$D \rightarrow D; D$**

**$D \rightarrow \text{id}: T$**

**$D \rightarrow \text{proc id}; D; S$**

**$T \rightarrow \text{integer} \mid \text{real}$**

**$\mid \text{array} [\text{num}] \text{ of } T_1$**

**$\mid \uparrow T_1$**

**$\mid \text{record } D \text{ end}$**

- 所有名字必须先声明后引用
- 变量声明、每个声明语句声明一个名字
- 过程声明、过程声明允许嵌套

# 一、过程中的声明语句

## ■ 假定

- 最内层过程
- 暂时不考虑记录结构的声明

## ■ 声明语句的文法

**$P \rightarrow D; S$**

**$D \rightarrow D; D$**

**$D \rightarrow \text{id}: T$**

**$T \rightarrow \text{integer}$**

**$T \rightarrow \text{real}$**

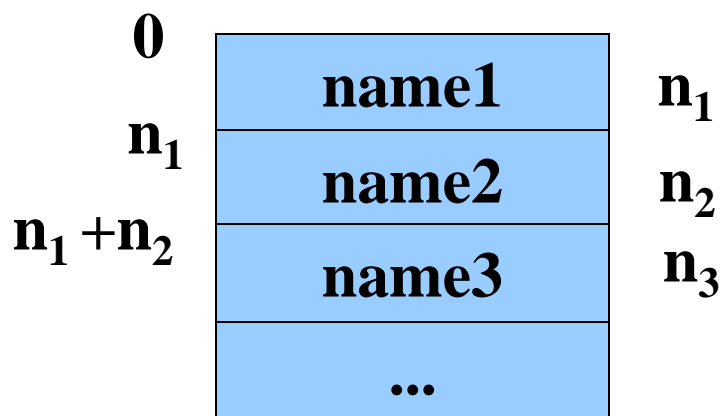
**$T \rightarrow \text{array} [\text{num}] \text{ of } T_1$**

**$T \rightarrow \uparrow T_1$**



# 设计翻译方案的目的

- 识别出被声明实体
- 记录实体的名字、类型、在数据区中的位置



- 引入：
  - 全局变量**offset**，记录下一个可用单元的相对地址
  - **T.width**：记录实体的域宽
  - **T.type**：记录实体的类型
  - **enter(id.name, T.type, offset)**

# 翻译方案1

i: integer;  
x: real;  
A: array[10] of integer

{ offset=0 }

$P \rightarrow \blacktriangle D; S$

$P \rightarrow MD$

$M \rightarrow \epsilon \quad \{ \text{offset}=0 \}$

$D \rightarrow D; D$

$D \rightarrow id:T \quad \{ \text{enter}(id.name, T.type, \text{offset});$   
 $\text{offset}=\text{offset}+T.width \}$

$T \rightarrow integer \quad \{ T.type=integer; T.width=4 \}$

$T \rightarrow real \quad \{ T.type=real; T.width=8 \}$

$T \rightarrow array [num] \text{ of } T_1 \quad \{ T.type=array(num.val, T_1.type);$   
 $T.width=num.val \times T_1.width \}$

$T \rightarrow \uparrow T_1 \quad \{ T.type=pointer(T_1.type); T.width=4 \}$

## 二、过程定义的处理

### ■ 作用域信息的保存

### ■ 文法

$P \rightarrow \blacksquare D; \bullet S$

$D \rightarrow D; D$

$D \rightarrow \text{id}; T$

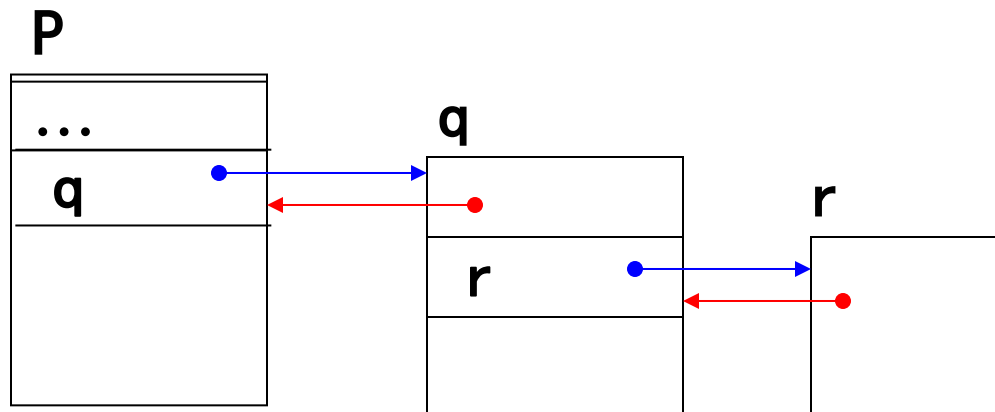
$D \rightarrow \text{proc id}; \blacksquare D; S \bullet$

$T \rightarrow \text{integer}$

$T \rightarrow \text{real}$

$T \rightarrow \text{array} [\text{num}] \text{ of } T_1$

$T \rightarrow \uparrow T_1$



创建主程序的符号表、初始化

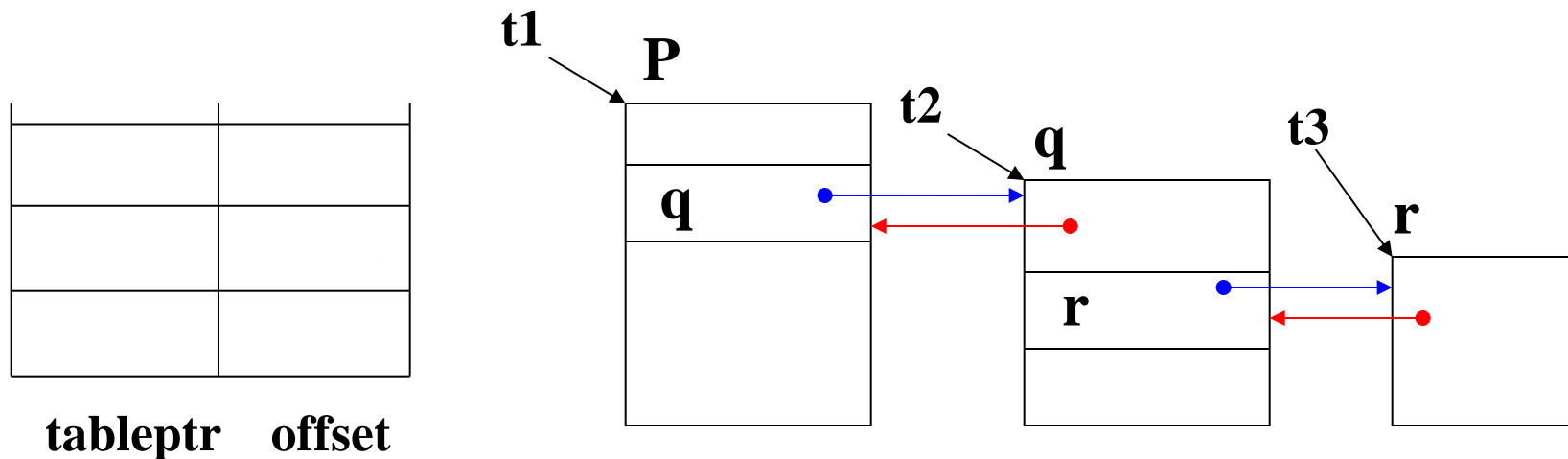
记录主程序中声明的变量所需要的空间

重定位操作：记录子过程中声明的局部变量所需要的空间，返回到外围过程

定位操作：为子过程id创建符号子表，并初始化

# 数据结构及过程

- 记录符号表嵌套关系的、便于操作的结构：栈



- 过程:

- maketable(previous)
- enter(table,name,type,offset)
- addwidth(table,width)
- enterproc(table,name,newtable)

# 翻译方案2

**P** → **M D; S** { addwidth(top(tableptr), top(offset));  
pop(tableptr); pop(offset) }

**M** →  $\epsilon$  { t= maketable(nil);  
push(t, tableptr); push(0, offset) }

**D** → **D; D**

**D** → **id: T** { enter(top(tableptr), id.name, T.type, top(offset));  
top(offset)=top(offset)+T.width }

**D** → **proc id; N D; S** { t=top(tableptr);  
addwidth(t, top(offset));  
pop(tableptr); pop(offset);  
enterproc(top(tableptr), id.name, t) }

**N** →  $\epsilon$  { t=maketable(top(tableptr));  
push(t, tableptr); push(0, offset) }

# 三、记录声明的处理

## ■ 文法

$P \rightarrow D; S$

$D \rightarrow D; D$

$D \rightarrow \text{id}: T$

$D \rightarrow \text{proc id}; D; S$

$T \rightarrow \text{record } \blacksquare D \text{ end } \bullet$

定位操作:

创建符号表

保存记录中各域的信息

重定位操作:

将各域的总域宽作为

该记录型的域宽,

返回记录声明所在过程

的符号表

## 翻译方案3

$T \rightarrow \text{record } \color{red}{L} D \text{ end} \quad \{ \quad T.\text{type} = \text{record}(\text{top}(\text{tableptr}));$   
 $\quad \quad \quad T.\text{width} = \text{top}(\text{offset});$   
 $\quad \quad \quad \text{pop}(\text{tableptr}); \text{pop}(\text{offset}) \quad \}$

$\color{red}{L} \rightarrow \epsilon \quad \{ \quad t = \text{mktable}(\text{nil});$   
 $\quad \quad \quad \text{push}(t, \text{tableptr}); \text{push}(0, \text{offset}) \quad \}$

# 举例

## ■ 声明

**x : integer;**

**q : record** ■

**i: integer;**

**x: real**

**end;** ●

**y: real;**

t →	x	integer	0
	q	record(t')	4
	y	real	16

t' →	i	integer	0
	x	real	4

t	24

**tableptr      offset**

**T.type=record(t')**  
**T.width=12**

## 6.4 类型检查

- 静态类型检查：由编译程序完成的检查
  - 动态类型检查：目标程序运行时完成的检查
    - 如果目标代码把每个对象的类型和该对象的值一起保存，那么任何检查都可以动态完成。
  - 一个健全的类型体制不需要动态检查类型错误
  - 如果一种语言的编译程序能够保证它所接受的程序不会有运行时的类型错误，则称这种语言是强类型语言。
  - 有些检查只能动态完成
- ```
char table[20];  
int i;  
计算table[i]
```



# 静态类型检查

- 设计类型检查程序时要考虑的因素：
  - 语法结构
  - 类型概念
  - 把类型指派给语言结构的规则
- Pascal、C语言报告中关于类型的描述：
  - 如果算术运算符加、减和乘的两个运算对象都是整型，那么结果是整型。
  - 一元运算符&的结果是指向运算对象所代表的实体的指针，如果运算对象的类型是‘...’，结果类型就是指向‘...’的指针。
- 暗示的概念：
  - 每一个表达式有一个类型
  - 类型有结构

# 类型体制

- 把类型表达式指派到程序各部分的一组规则
- 由类型检查程序实现
- 同一语言的不同编译程序使用的类型体制可能不同
  - 数组作为变元传递时，数组的下标集合可以指明，也可以不指明
  - 原因：不同的**Pascal**语言编译程序实现的类型体制不同
  - **UNIX**系统中，**lint**命令实现的类型体制比**C**语言编译程序本身所实现的更详细。

# 一、类型表达式

## ■ 类型表达式的定义：

- 基本类型是类型表达式

**boolean**、**char**、**integer**、和 **real**

错误类型**type\_error**、回避类型**void**

- 类型名是类型表达式，因为类型表达式可以命名。

- 类型构造器作用于类型表达式的结果仍是类型表达式

- 数组：如果**T**是类型表达式，那么 **array(I, T)** 是元素类型为**T**和下标集合为**I**的数组的类型表达式，**I**通常是一个整数域。

**var A:array[1..10] of integer;**

与**A**相关的类型表达式为：**array(1..10,integer)**

- 笛卡儿积：如果**T<sub>1</sub>**和**T<sub>2</sub>**是类型表达式，那么它们的笛卡儿积**T<sub>1</sub>×T<sub>2</sub>**也是类型表达式，假定×是左结合的。
- 记录：记录类型是它的各域类型的笛卡儿积

把类型构造器 **record** 作用于由域名和与之相关的类型表达式组成的二元组，就形成记录的类型表达式

如Pascal的程序片段：

```
type row=record
    address: integer;
    lexeme: array[1..15] of char
end;
var table: array[1..10] of row;
```

类型名row代表类型表达式：

**record((address×integer)×(lexeme×array(1..15,char)))**

和变量table相关的类型表达式为：

**array(1..10,row)**

- **指针**：如果T是类型表达式，那么 **pointer(T)** 是类型“指向类型T的对象的指针”的类型表达式。

**var p:↑row;**

与P相关的类型表达式为：

**pointer(row)**

- **函数**：从定义域类型**D**到值域类型**R**的映射  
类型由类型表达式  $D \rightarrow R$  表示。

**Pascal的内部函数 mod:**

**int×int→int**

用户定义的**Pascal**函数: **function f(a,b: char): ↑integer;**

**f**的类型表达式: **char×char→pointer(integer)**

**C 语言函数: int square(int x) { return x\*x }**

**square**的类型表达式: **integer→integer**

**函数g:**

参数是把整数映射成整数的函数

返回结果是和参数类型相同的另一函数

**g**的类型表达式为:

**(integer→integer)→(integer→integer)**

- 类型表达式可以包含变量（称为**类型变量**），变量的值是类型表达式。

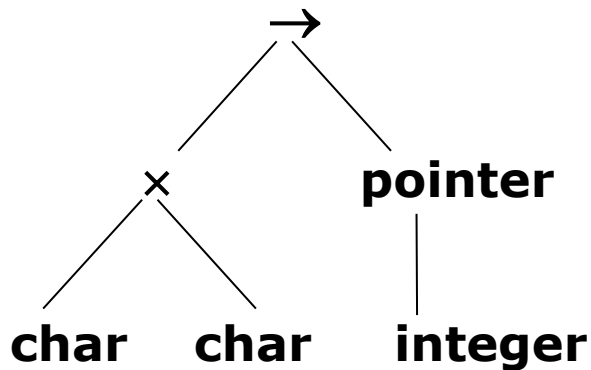
# 类型表达式的有向图

- 利用语法制导翻译技术为类型表达式构造树或dag
  - 内部结点：类型构造器
  - 叶结点：基本类型、类型名、或类型变量

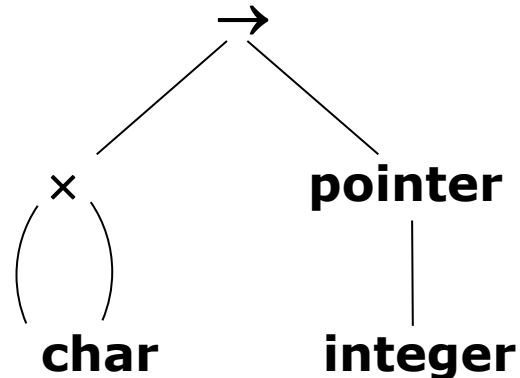
■ 如：

**char×char → pointer(integer)**

树：



dag:



## 二、类型等价

### ■ 关键：

精确地定义什么情况下两个类型表达式等价

### ■ 问题：

- 类型表达式可以命名，且这个名字可用于随后的类型表达式中
- 名字代表它自己？代表另一个类型表达式？
- 新的名称是一个类型表达式，这个表达式与名称所代表的类型表达式是否总是等价的？

# 1. 结构等价

- 如果类型表达式仅由类型构造器作用于基本类型组成，则两个类型表达式等价的自然概念是结构等价
  - 两个类型表达式要么是同样的基本类型
  - 要么是同样的构造器作用于结构等价的类型表达式。
- 两个类型表达式结构等价**当且仅当**它们完全相同

例如：

**integer 仅等价于 integer**

**pointer(integer) 仅等价于 pointer(integer)**



## 例：考虑如下Pascal声明

|                                                                                  |                                                                                  |                                                                                |                                                                                |
|----------------------------------------------------------------------------------|----------------------------------------------------------------------------------|--------------------------------------------------------------------------------|--------------------------------------------------------------------------------|
| <b>A: record</b><br><b>    i: integer</b><br><b>    f: real</b><br><b>end;</b> ✓ | <b>B: record</b><br><b>    i: integer</b><br><b>    f: real</b><br><b>end;</b> ✓ | <b>C: record</b><br><b>    f: real</b><br><b>    i: integer</b><br><b>end;</b> | <b>D: record</b><br><b>    x: real</b><br><b>    y: integer</b><br><b>end;</b> |
|----------------------------------------------------------------------------------|----------------------------------------------------------------------------------|--------------------------------------------------------------------------------|--------------------------------------------------------------------------------|

## 考虑如下C语言声明：

|                                                                                            |                                                                                                                    |                                                                                       |
|--------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| <b>struct recA</b><br><b>{</b><br><b>    int i;</b><br><b>    char c;</b><br><b>} a;</b> ✓ | <b>typedef struct</b><br><b>{</b><br><b>    int i;</b><br><b>    char c;</b><br><b>} recB;</b><br><b>recB b;</b> ✓ | <b>struct</b><br><b>{</b><br><b>    int i;</b><br><b>    char c;</b><br><b>} c;</b> ✓ |
|--------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|

# 测试两个类型表达式结构等价的算法

输入：两个类型表达式s和t

输出：如果s和t结构等价，则返回true，否则返回false

方法：

```
bool  sequiv(bexpr s, t)
```

```
{  
    if (s和t是同样的基本类型) return 1;  
    else if (s==array(s1,s2))&&(t==array(t1,t2))  
        return sequiv(s1,t1) && sequiv(s2,t2);  
    else if (s==s1×s2)&&(t==t1×t2)  
        return sequiv(s1,t1) && sequiv(s2,t2);  
    else if (s==pointer(s1))&&(t==pointer(t1))  
        return sequiv(s1,t1);  
    else if (s==s1→s2)&&(t==t1→t2)  
        return sequiv(s1,t1) && sequiv(s2,t2);  
    else return 0;  
}.
```

# 实际应用中结构等价概念的修改

- 当数组作为参数传递时，**数组的界**不作为类型的一部分
- 算法调整  
    **else if (s==array(s<sub>1</sub>,s<sub>2</sub>))&&(t==array(t<sub>1</sub>,t<sub>2</sub>))**  
        **return sequiv(s<sub>2</sub>,t<sub>2</sub>);**

## 2. 名字等价

- 有些语言(如Pascal、C等)允许用户定义类型名
- C类型定义和变量声明 (6.1)

```
typedef struct {  
    int age;  
    char name[20];  
} recA;  
typedef recA *recP;  
recP a;  
recP b;  
recA *c, *d;  
recA *e;
```

问题: **a、b、c、d和e**是否具有相同的类型?

关键: 类型表达式**recP**和**pointer(recA)**是否等价

回答: 依赖于具体的实现系统

原因: C语言报告中没有定义“类型等价”这个术语

# 名字等价把每个类型名看成是一个可区别的类型

- 两个类型表达式名字等价，当且仅当它们完全相同
- 所有的名字被替换后，两个类型表达式成为结构等价的类型表达式，那么这两个表达式结构等价。
- 声明中的变量及其类型表达式：

| 变量 | 类型表达式         |
|----|---------------|
| a  | recP          |
| b  | recP          |
| c  | pointer(recA) |
| d  | pointer(recA) |
| e  | pointer(recA) |

结构等价 { 名字等价 { 名字不等价

- 考虑名字等价的情形
- 考虑结构等价的情形

# 类型等价实例

- C语言使用介于名字等价和结构等价之间的一种类型等价形式，即对于struct和union采用名字等价，其他则采用结构等价。
- 有如下C语言声明：

```
struct {  
    short j;  
    char c;  
} x, y;  
struct {  
    short j;  
    char c;  
} b;
```

x、y 名字等价  
x、y与b名字不等价

Pascal语言采用了与C语言类似的规则，几乎所有类型构造器（如记录、数组、指针等）的应用都将建立一个新的不等价类型。

# 与声明6.1等效的声明6.2:

```
typedef struct {  
    int age;  
    char name[20];  
} recA;  
typedef recA *recP;  
typedef recA *recD;  
typedef recA *recE;  
recP a;  
recP b;  
recD c, d;  
recE e;
```

## ■ 名字等价

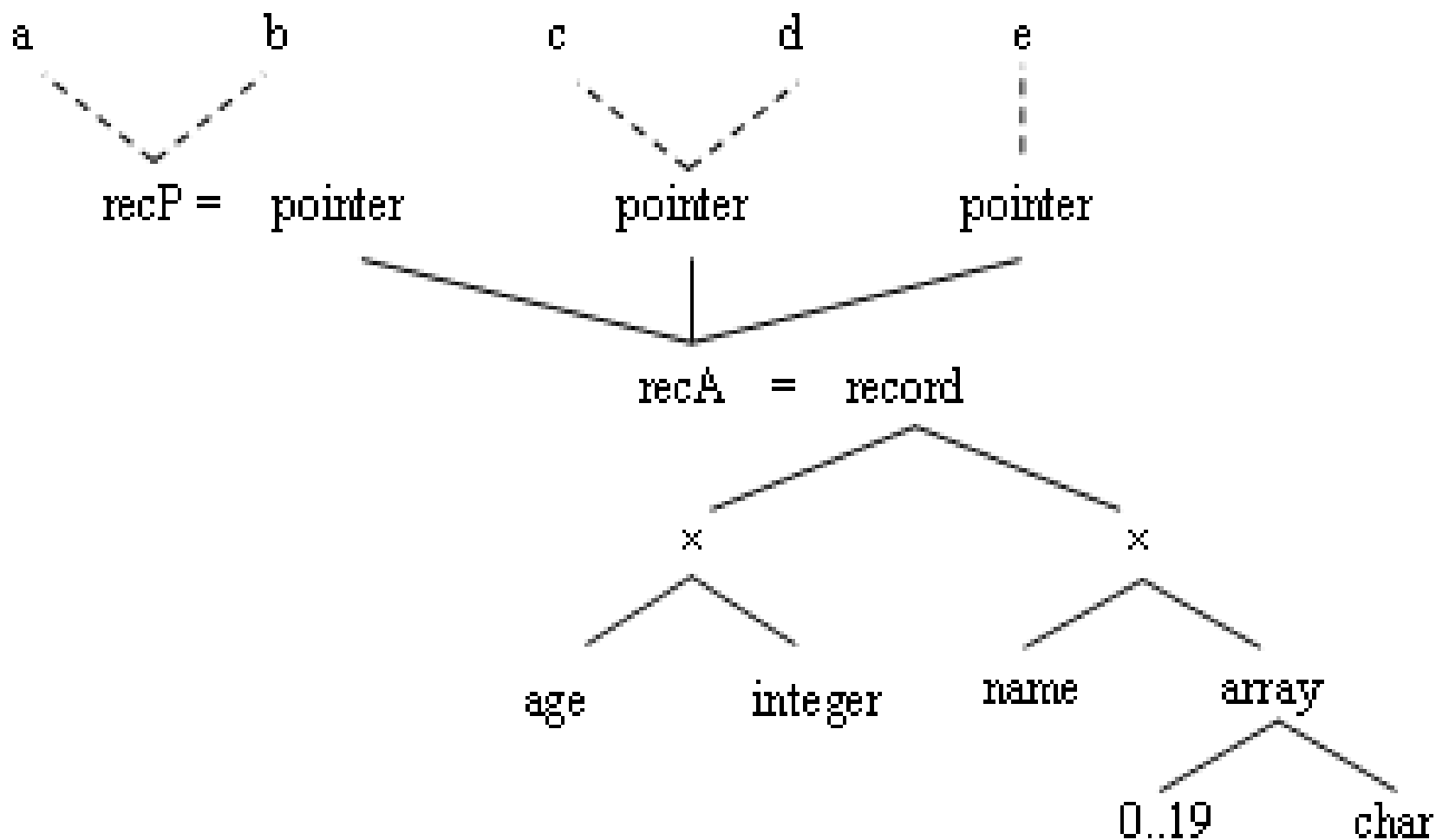
- a和b具有等价的类型
- c和d具有等价的类型
- a、c、和e具有不同的类型

# 类型图

- 类型图表示类型
- 构造方法：
  - 每当出现一个类型构造器或基本类型，就建立一个新的结点；
  - 每当出现一个新的类型名时，就建立一个叶结点；
  - 要跟踪该名字所代表的类型表达式。
- 在类型图中，如果两个类型表达式用相同的结点表示，则它们是名字等价的。



# 与声明6.2对应的类型图

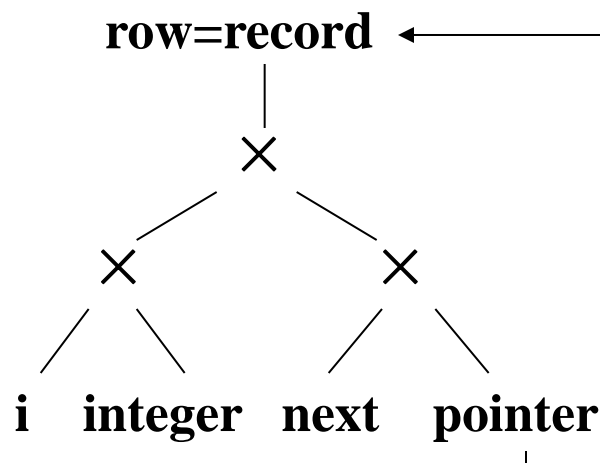
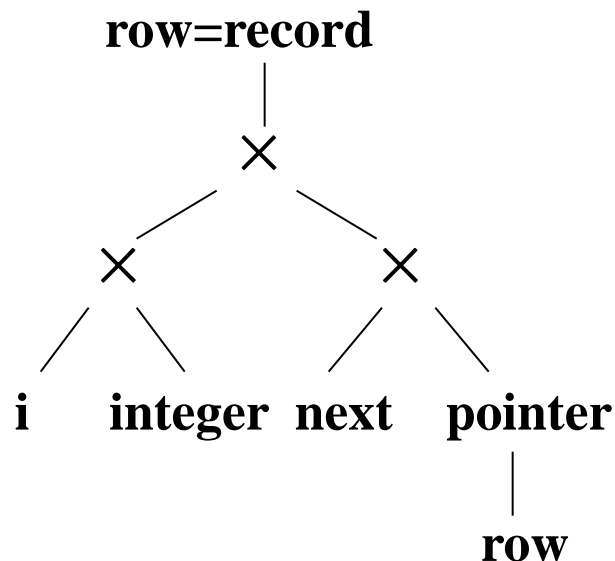
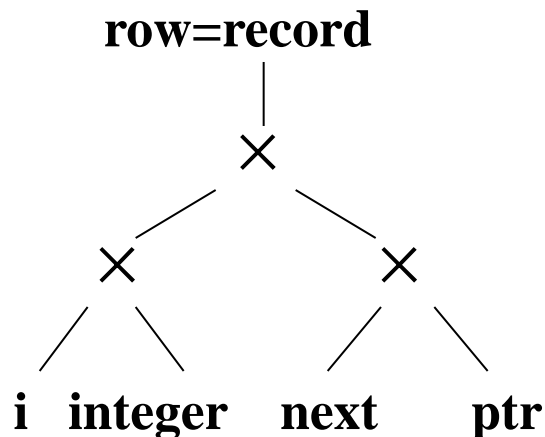


### 3. 类型表示中的环

- 指针和结点的类型定义用Pascal语言描述如下:

```
type ptr = ↑row;  
row = record  
    i: integer;  
    next: ptr;  
end;
```

- row的类型表达式:



# C 采用结构等价

## ■ 考虑声明:

```
typedef struct cell_1 *link_1;
typedef struct cell_2 *link_2;
struct cell_1 {
    int info;
    link_1 next;
};
struct cell_2 {
    int info;
    link_2 next;
};
```

- 如果开始的时候就假定 **link\_1** 和 **link\_2** 是结构等价的, 那么就很容易地得出 **cell\_1** 和 **cell\_2** 是结构等价的, 并且可以得出 **link\_1** 和 **link\_2** 它们本身是等价的。
- 结构名是其类型的一部分, 于是, 在测试结构等价时, 当遇到结构构造符 **struct** 时, 测试停止, 结果被比较的类型或者由于它们有同样的命名结构类型而等价, 或者不等价。

## 6.5 一个简单类型检查程序的说明

### ■ 利用语法制导方法说明类型体制

一、语言说明

二、确定标识符的类型

三、表达式的类型检查

四、语句的类型检查

五、类型转换

# 一、语言说明

## ■ 源语言文法:

$P \rightarrow D;S$

$D \rightarrow D;D \mid id:T$

$T \rightarrow \text{char} \mid \text{integer} \mid \text{boolean}$

$\mid \text{array}[\text{num}] \text{ of } T \mid \uparrow T \mid T_1' \rightarrow T_2$

$S \rightarrow id:=E \mid \text{if } E \text{ then } S_1 \mid \text{while } E \text{ do } S_1 \mid S_1;S_2$

$E \rightarrow \text{literal} \mid \text{num} \mid id$

$\mid id_1 < id_2 \mid E_1 \text{ and } E_2$

$\mid E_1 \text{ mod } E_2 \mid E_1[E_2] \mid E \uparrow \mid E_1(E_2)$

## ■ 该文法产生的一个程序:

```
i: integer  
k: integer;  
i:=7;  
k:=k mod i
```

## 二、确定标识符的类型

### ■ 语言中的类型：

#### – 基本类型：

**char、integer、和boolean**  
**type\_error**和**void**

#### – 构造类型：

数组、指针和函数

### ■ 假定：

#### – 数组下标从**1**开始

类型**array[256] of char**的类型表达式是：  
**array(1..256,char)**

#### – 前缀算符**↑**用于建立指针类型

**↑integer**的类型表达式是**pointer(integer)**

#### – 函数类型由定义域类型和值域类型确定

# 翻译方案中

## 确定并保存标识符类型的部分

引入综合属性**T.type**，记录类型表达式

$P \rightarrow D; S$

$D \rightarrow D; D$

$D \rightarrow id:T \{ \text{addtype}(id.entry, T.type) \}$

$T \rightarrow char \{ T.type = char \}$

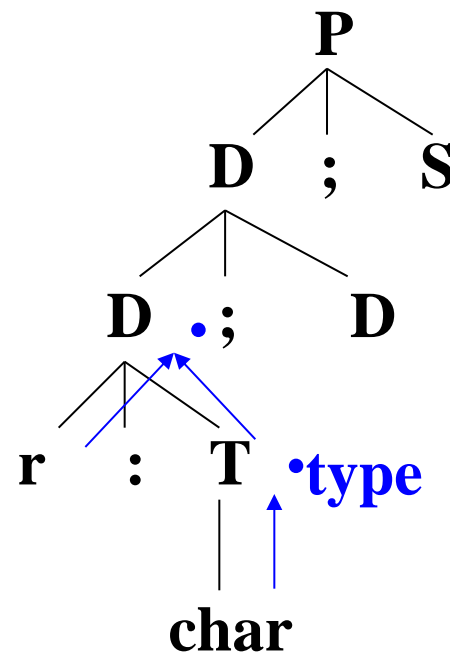
$T \rightarrow integer \{ T.type = integer \}$

$T \rightarrow boolean \{ T.type = boolean \}$

$T \rightarrow array[num] \text{ of } T_1 \{ T.type = array(num.val, T_1.type) \}$

$T \rightarrow \uparrow T_1 \{ T.type = pointer(T_1.type) \}$

$T \rightarrow T_1 \rightarrow T_2 \{ T.type = T_1.type \rightarrow T_2.type \}$



下标缺省为1

# 三、表达式的类型检查

综合属性E.type，类型体制指派给E产生的表达式的类型表达式

$E \rightarrow \text{literal}$  {  $E.\text{type} = \text{char}$  }

$E \rightarrow \text{num}$  {  $E.\text{type} = \text{integer}$  }

$E \rightarrow \text{id}$  {  $E.\text{type} = \text{lookup}(\text{id}.\text{entry})$  }

$E \rightarrow \text{id}_1 > \text{id}_2$  { if (lookup(id<sub>1</sub>.entry) == integer) &&

$E \rightarrow E_1 \text{ and } E_2$  (lookup(id<sub>2</sub>.entry) == integer) ||

$E \rightarrow E_1 \text{ mod } E_2$  (lookup(id<sub>1</sub>.entry) == char) &&

$E \rightarrow E_1 [E_2]$  (lookup(id<sub>2</sub>.entry) == char)

$E \rightarrow E_1 \uparrow$   $E.\text{type} = \text{boolean};$

else  $E.\text{type} = \text{type\_error};$  }

$E \rightarrow E_1 (E_2)$



$E \rightarrow E_1 \text{ and } E_2$  { if ( $E_1.\text{type} == \text{Boolean}$ ) \&\&  
( $E_2.\text{type} == \text{Boolean}$ )  
     $E.\text{type} = \text{Boolean};$   
    else  $E.\text{type} = \text{type\_error};$  }

$E \rightarrow E_1 \text{ mod } E_2$  { if ( $E_1.\text{type} == \text{integer}$ ) \&\&  
( $E_2.\text{type} == \text{integer}$ )  
     $E.\text{type} = \text{integer};$   
    else  $E.\text{type} = \text{type\_error};$  }

$E \rightarrow E_1 [E_2]$  { if ( $E_1.\text{type} == \text{array}(s,t)$ ) \&\&  
( $E_2.\text{type} == \text{integer}$ )  
     $E.\text{type} = t;$

$E \rightarrow E_1 \uparrow$   
else  $E.\text{type} = \text{type\_error};$  }

$E \rightarrow E_1 (E_2)$

$E \rightarrow E_1 \uparrow$

```
{ if (E1.type==pointer(t))  
    E.type=t;  
  else E.type=type_error; }
```

$E \rightarrow E_1 (E_2)$

```
{ if (E1.type==s→t) && (E2.type==s)  
    E.type=t;  
  else E.type=type_error; }
```

## 四、语句的类型检查

$S \rightarrow id := E$     { if (lookup(id.entry) == E.type)  
                    S.type = void;  
                    else S.type = type\_error; }

$S \rightarrow \text{if } E \text{ then } S_1$     { if (E.type == boolean)  
                            S.type = S<sub>1</sub>.type;  
                            else S.type = type\_error; }

$S \rightarrow \text{while } E \text{ do } S_1$     { if (E.type == boolean)  
                            S.type = S<sub>1</sub>.type;  
                            else S.type = type\_error; }

$S \rightarrow S_1 ; S_2$     { if (S<sub>1</sub>.type == void) && (S<sub>2</sub>.type == void)  
                    S.type = void ;  
                    else S.type = type\_error; }

$P \rightarrow D ; S$     { if (S.type == void) P.type = void;  
                    else P.type = type\_error }

# 五、类型转换

- 例如：

```
int x=5;
```

```
x=2.1+x/2;
```

- 等价于： `x=int(2.1+double(x/2))`， 结果： `x=4`

- 如果从一种数据类型转换成另一种数据类型可以由编译程序自动完成，则这种类型转换是隐式的，隐式转换也叫做**强制转换**。

- 如果转换必须由程序员显式地写在源程序中，则这种转换叫做**显式转换**。

- 显式的类型转换对类型检查程序来说好象函数调用

# 常数的隐式转换

- for  $I:=1$  to  $N$  do  $X[I]:=1$
- 执行时间:  $48.4N\mu s$
  
- for  $I:=1$  to  $N$  do  $X[I]:=1.0$
- 执行时间:  $5.4N\mu s$
  
- 编译程序为第一个语句产生的目标代码含运行时的例程调用, 该例程把1的整型表示转换为实型表示
- 大多数编译程序都在编译时把1转换为1.0

## 6.6 类型检查有关的其他主题

- 一、函数和运算符的重载
- 二、多态函数
- 三、错误处理

# 一、函数和运算符的重载

- 在数学中，加法运算符‘+’是重载的
  - 当**A**和**B**是整数、实数、复数或矩阵时，表达式**A+B**中的‘+’具有不同的含义。
- 重载符号的含义依赖于上下文
- 在重载符号的引用点，其含义能够确定到唯一，叫做**重载的消除**。
  - 重载的消除有时也称为**算符的辨别**，因为它确定了运算符表示哪种运算。
- 大多数语言中，算术运算符是重载的，并且它们的重载可以通过查看其操作对象来消除。

# 类型集合

- 仅通过检查函数的参数类型并不一定能消除重载
  - 因为一个子表达式可能有不止一个类型，而是有一个可能的类型集合
  - 上下文要提供足够的信息来缩小这个集合到唯一的类型

- 例：用C++语言描述的三个重载的函数min的定义：

min\_1: `int min(int x, int y) { return x<y ? x : y; }`

min\_2: `double min(double x, double y) { return x<y ? x : y; }`

min\_3: `int min(int x, int y, int z)`  
`{ return x<y ? (x<z? x:z) : (y<z ? y:z); }`

- 调用：

(1) `min(3, 7);` // call min\_1

(2) `min(3.3, 7.7);` // call min\_2

(3) `min(3, 7, 5);` // call min\_3

(4) `min(3.3, 7);` // call min\_???

`min(3, 7);` // call min\_1  
`min(3.3, 7.0);` // call min\_2  
取决于类型转换规则  
C++, 都可以  
Java, 后者  
Ada, 都不可以



- 增加如下定义后，调用 (4) 在Ada和C++中均合法：

```
min_4: double min(int x, double y)
    { return x<y ? (double)x : y; }
```

```
min_5: double min(double x, int y)
    { return x<y ? x : (double) y; }
```

- 调用：(5) `min(3, 7, 5.2);` // call min\_???  
在Ada和Java语言中不合法，在C++语言中合法

# C++语言中，大多数运算符都是重载的

- 对于运算符“+”：
  - 作用于两个整型数，结果仍为整型；
  - 也可以作用于两个实型数，结果仍为实型；
  - 用于连接两个字符串。
- 因此，“+”的类型表达式有：
  - `integer × integer → integer`
  - `float × float → float`
  - `string × string → string`

# C++语言中的重载运算符

- `+`、`-`、`*`、`/`、`%`、`^`、`&`、`|`、`~`、`!`、`=`、`++`、`--`、`+=`、`-=`、`/=`、`%=`、`^=`、`&=`、`|=`、`<<`、`>>`、`>>=`、`<<=`、`<`、`>`、`==`、`!=`、`<=`、`>=`、`&&`、`||`、`->*`、`,`、`->`、`[]`、`()`、`new`、`delete`等。
- 通过关键字`operator`可以重载运算符，如`operator+`、`operator<`等。

## 二、多态函数

- 多态函数允许变元有不同的类型，即函数体中的语句可以在不同类型的变元下执行。
- 数组索引、函数引用和指针操作通常是多态的
- 原因：没有限于特定类型的数组、函数或指针
- C语言参考手册中关于指针'&'的论述：  
“如果运算对象的类型是'...'，那么结果类型是指向'...'的指针”。
- 因为任何类型都可以代替'...'，所以C语言的算符'&'是多态的。

# 1. 多态函数应用举例

- C++使用函数模板实现多态函数，以对不同类型的数据进行相同的处理。
- 如求一个数的平方的函数模板可以定义如下：

```
template <class T>
T power(T x)
{ return x*x;}
```
- 调用power (2)、power (2. 5)、power (x)等都是合法的。
- 函数模板只是对函数的描述，编译程序并不为其产生任何可执行代码，只有当遇到函数调用时，编译程序自动将模板中的类型参数T用实参的类型来替换，生产一个重载函数，该重载函数称为模板函数。
- 如当遇到函数调用power (2. 5)时，将函数模板中出现T的地方用double替换，生成如下的一个模板函数：

```
double power(double x)
{ return x*x; }
```

# 用C++语言描述的对n个数按递增顺序排序的多态函数

```
#include "iostream.h"
/* 定义函数模板 */
template <class T>
void sort (T a[], int n) {
    int i, j, k;
    T x;
    for(i=0; i<n-1; i++) {
        k=i;
        for(j=i+1; j<n; j++)
            if (a[k]>a[j]) k=j;
        if (i!=k) { x=a[i]; a[i]=a[k];
                    a[k]=x; }
    }
}
```

```
void main( )
{
    int i, a[8]={4, 6, 1, 3, 8, 2, 7, 5};
    double d[5]={5.5, 8.8, 2.2, 7.7, 3.3};
    /* 对int型数组排序 */
    sort(a, 8);
    for (i=0; i<10; i++) cout << a[i] <<endl;
    /* 对double型数组排序 */
    sort(d, 5);
    for (i=0; i<5; i++) cout << d[i] <<endl;
}
```

## 2. 类型变量与类型推导

- 为便于讨论未知类型，可以引入变量来表示类型表达式，这样的变量称为类型变量。
- 用希腊字母 $\alpha$ 、 $\beta$ 、...等来表示类型变量
- 在不要标识符的声明先于使用的语言中，类型变量的一个重要应用是检查标识符前后使用的一致性
- 检查程序跟踪未声明的程序变量的使用情况
- 从语言结构的使用方式推导其类型称为类型推导
- 例如：推导表达式  $a[i]+i$  中各名字的类型
  - 如分别为 $a$ 、 $i$ 分配类型变量 $\alpha$ 和 $\beta$ 。
  - 类型检查程序在扫描到 ‘[ ]’时，可以知道， $a$ 是数组类型
  - 由于 $i$ 出现在数组下标的位置，可以推断 $i$ 是整数，所以： $\beta=\text{int}$ 。
  - 再根据表达式是一个加法表达式，算符 ‘+’要求两个运算分量必须有相同的类型，因此可知数组元素也是整数类型。
  - 所以： $\alpha=\text{array}(\text{int}, \text{int})$

# 从函数体推导函数类型

```
type link = ↑cell;  
cell = record  
    info: integer;  
    next: link  
end;
```

proc 是一个过程  
变元的个数?  
变元的类型?

```
procedure example (nptr: link; proc: procedure);
```

```
begin  
    while nptr <> nil do  
        begin  
            proc(nptr);  
            nptr := nptr↑.next  
        end  
    end  
end;
```

proc 用于链表的每个元素  
可以为表元中的整型变量  
置初值、或打印其值等  
proc 的类型必是  $\text{link} \rightarrow \text{void}$



# 类型推导

- 从语言结构的使用方法推导出其类型

- 例:

**function def(p);** ← 对P的类型一无所知，用 $\beta$ 表示

**begin**

**return p $\uparrow$**  ← P是指向某类型 $\alpha$ 的指针类型， $\beta = \text{pointer}(\alpha)$

**end;**

函数**def**的返回值的类型:

$\alpha$

**def**的类型表达式:

对任何类型 $\alpha$ ， $\text{pointer}(\alpha) \rightarrow \alpha$

# 三、错误恢复

- 类型错误**不直观**
- 出错处理会影响类型检查的规则
- 包含出错处理可能使类型体制超出说明正确程序所需要的类型体制
- 类型检查程序必须应付**缺少信息**的局面
- **使用的技术**类似于支持变量隐式声明的语言所需要的技术

# 小结

## ■ 语义分析的概念

- 编译的一个重要任务、检查语义的合法性
- 符号表的建立和管理
- 语义检查

## ■ 符号表

- 何时创建
- 内容
- 操作
  - 检索、插入
  - 定位、重定位
- 组织形式
  - 非块结构语言
  - 块结构语言

# 小结（续）

## ■ 静态语义检查

- 类型检查
- 控制流合法性检查
- 同名变量检查
- 关联名字检查
- 利用语法制导的翻译技术实现

## ■ 类型体制

- 语法结构、类型概念、把类型指派给语言结构的规则
- 类型表达式的定义
- 类型表达式的等价
  - 结构等价
  - 名字等价
  - 类型表达式结构等价的测试算法

# 小结（续）

- 简单类型检查程序的说明
  - 声明语句的类型检查
  - 表达式的类型检查
  - 语句的类型检查
  - 类型转换
- 类型检查有关的其他主题
  - 函数和运算符的重载
  - 多态函数
  - 类型推导

# 作业 (P. 216)

- 6. 2
- 6. 3
- 6. 9
- 6. 10
- 6. 12
- 6. 13
  
- 程序设计 3

# 第7章 运行环境



*LI Wensheng, SCST, BUPT*

知识点：活动记录、控制栈  
栈式存储分配  
非局部名字的访问  
参数传递方式

# 运行环境

- 程序运行时刻的环境，即运行中程序的信息是怎样存储和访问的。
- 执行过程中，程序中数据的存取是通过对应的存储单元来进行的。
- 存储组织与管理
  - ◆ 早期的计算机上，存储管理工作是由程序员自己来完成
  - ◆ 有了高级语言之后，程序中使用的存储单元都由标识符来表示，它们对应的内存地址由编译程序在编译时或由其生成的目标程序在运行时进行分配。
- 存储的组织及管理是编译程序要完成的一个复杂而又十分重要的工作。



# 运行环境

7.1 程序运行时的存储组织

7.2 存储分配策略

7.3 访问非局部名字

7.4 参数传递机制

小 结

# 7.1 程序运行时的存储组织

概念：过程与活动

- 一、程序运行空间的划分
- 二、控制栈与活动记录
- 三、作用域及名字绑定

# 概念：过程与活动

## ■ 过程的定义

- ◆ 一个声明语句
- ◆ 把一个标识符和一个语句联系起来
- ◆ 标识符是过程名，语句是过程体。

## ■ 过程的分类

- ◆ 过程：没有返回值的過程
- ◆ 函数：有返回值的過程
- ◆ 也可以把函数、一个完整的程序看作过程

## ■ 过程引用：过程名出现在一个可执行语句中

## ■ 参数：

- ◆ 形参、实参

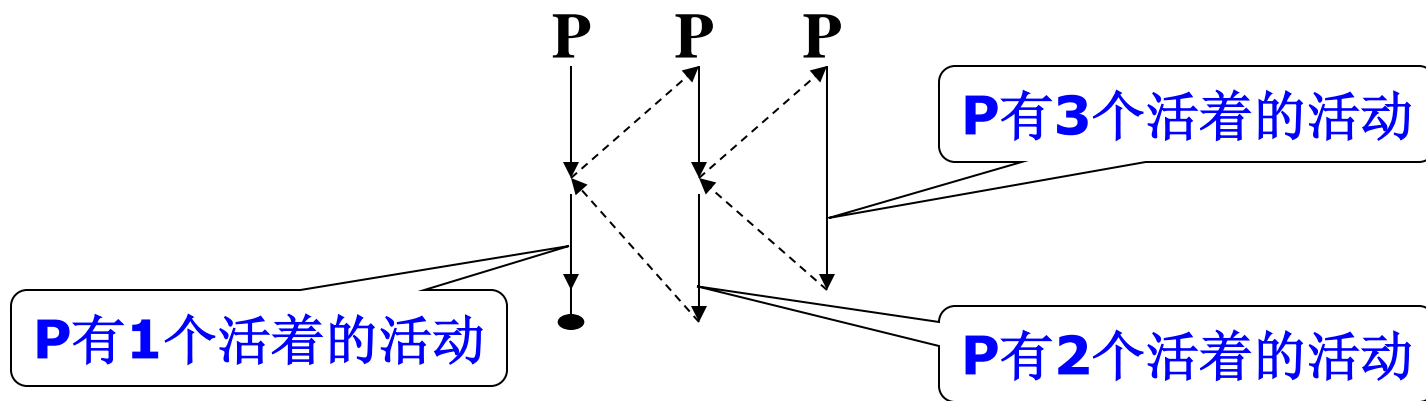
# 活动

## ■ 活动

- ◆ 一个过程的每次执行称为它的一次活动
- ◆ 如果一个过程在执行中，则称它的这次活动是**活着的**

## ■ 过程与活动

- ◆ 过程是一个静态概念，活动是一个动态概念
- ◆ 过程与活动之间可以是**1:1**、**1:m**的关系
- ◆ 递归过程，同一时刻可能有若干个活动是活着的
- ◆ 每个活动都有自己独立的存储空间/数据空间



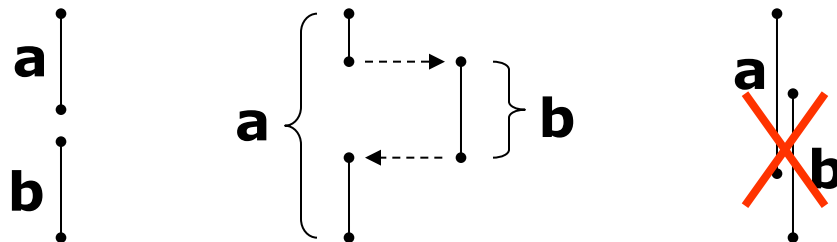
# 活动的生存期

## ■ 程序执行时, 过程之间的控制流

- ◆ 是连续的
- ◆ 过程的每一次执行都是从过程体的起点开始, 最后控制返回到直接跟随本次调用点的位置。

## ■ 活动的生存期

- ◆ 过程体的执行中, 第一步和最后一步之间的一系列步骤的执行时间
- ◆ 过程**P**的一次活动的生存期, 包括执行过程**P**所调用的过程的时间, 以及这些过程所调用的过程的时间
- ◆ 如果**a**和**b**是过程的活动, 那么它们的生存期要么是不重叠, 要么是嵌套的。

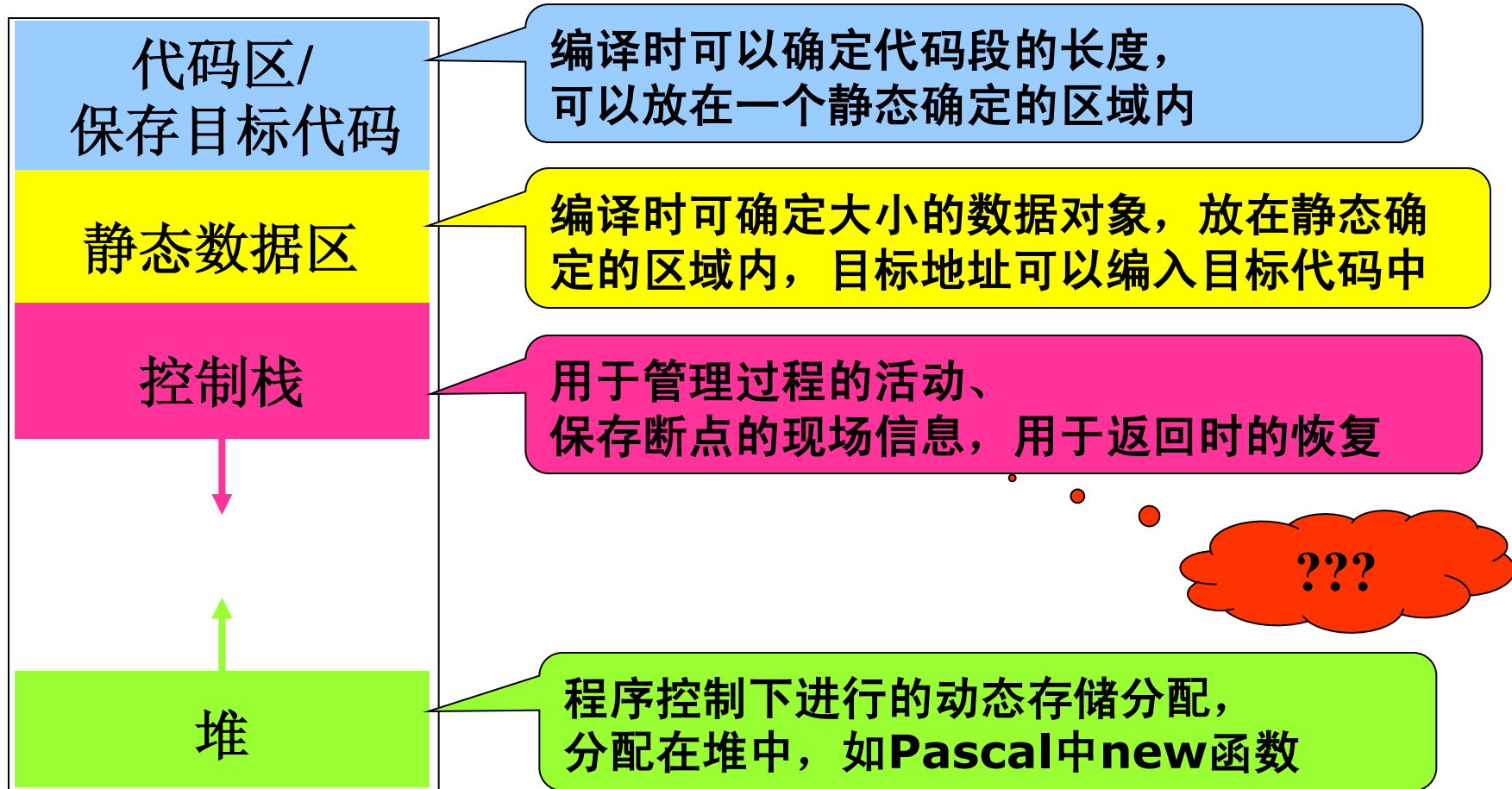


## ■ 递归过程:

- ◆ 如果一个过程, 它的不同活动的生存期可以嵌套, 则这个过程是递归的
- ◆ 直接递归、间接递归

# 一、程序运行空间的划分

- 编译程序在编译源程序时，向操作系统申请一块内存区域，以便被编译程序在其中运行



## 二、控制栈与活动记录

- **控制栈：用于保存控制信息的栈。**
  - ◆ 程序执行过程中使用控制栈来保存活动的生存踪迹及活动的运行环境。
- **活动记录：一个连续的存储块**
  - ◆ 记录过程在一次执行中所需要的信息
- **通常，活动记录分配在控制栈中（像Pascal、C）**
  - ◆ 当一个过程被调用时，被调用过程的一次新的活动被激活，在栈顶为该活动创建一个新的活动记录来保存其环境信息；
  - ◆ 当活动结束，控制从被调用过程返回时，释放该活动记录，使调用过程的活动记录成为栈顶活动记录，即恢复调用过程的执行环境。

# 活动记录的内容

|       |                                     |
|-------|-------------------------------------|
| 返回值   | 本活动返回给调用过程的值                        |
| 实参区域  | 调用过程提供给本活动的实参值                      |
| 控制链   | 指向调用过程的活动记录的指针，用于本活动结束时的恢复          |
| 访问链   | 指向直接外围过程的最近一次活动的活动记录的指针，用于对非局部名字的访问 |
| 机器状态域 | 保存断点的现场信息，寄存器、PSW等                  |
| 局部数据区 | 在本次活动中，为过程中定义的局部变量分配的存储空间           |
| 临时数据区 | 存放中间计算结果                            |

根据确定每个域所需空间大小的时间早晚安排其位置。

(1) 早：中间 晚：两头 (2) 用于通信：前面 自己用的：后面



# 活动记录中内容的安排原则

- 大小能够较早确定的区域放在活动记录的中间，大小较晚才能确定、并且变化较多的区域放在活动记录的两头。
  - ◆ 控制链、访问链、机器状态域，是编译器设计的一部分，编译器构造时就可以确定它们的大小，所以把这些区域放在活动记录的中间。
  - ◆ 参数域放在前面，便于调用过程进行参数传递，同时，被调用过程也可很方便地进行访问。
  - ◆ 返回值域放在最前面，便于调用过程可以根据自己的栈指针访问该区域，取回返回值。
  - ◆ 局部数据/临时数据安排在最后，其大小变化不会影响到活动记录中其他数据的存取。并且调用过程也无权访问被调用过程中的局部数据。

# 局部数据的安排

## ■ 常识:

- ◆ 程序运行时使用连续的存储空间
- ◆ 内存可编址的最小单位是字节
- ◆ 一个机器字由若干个字节组成
- ◆ 一个名字所需存储空间的大小由其类型决定
- ◆ 需多个字节表示的数据对象，存放在连续字节的存储块中，第一个字节的地址作为它的地址

## ■ 局部数据的安排

- ◆ 局部数据区是在编译过程中检查声明语句时安排的
- ◆ 长度可变的数据对象，放在该区域之外

## ■ 数据对象的存储安排受目标机器编址限制的影响

# 编址限制的影响

- 整数加法指令可能要求整数的地址能够被4整除

- 要求地址对齐  
如：`x:integer;`  
`y:char;`  
`z:integer;`

如果`char`占一个字节，`integer`占2个字节，`x`、`y`、`z`共需要5个字节。

如果要求从双字节地址分配，则需要为这三个变量分配6个字节，

- 为求分配上的全局统一，而多余出来的无用空间叫做填塞（**padding**）

# 三、作用域及名字绑定

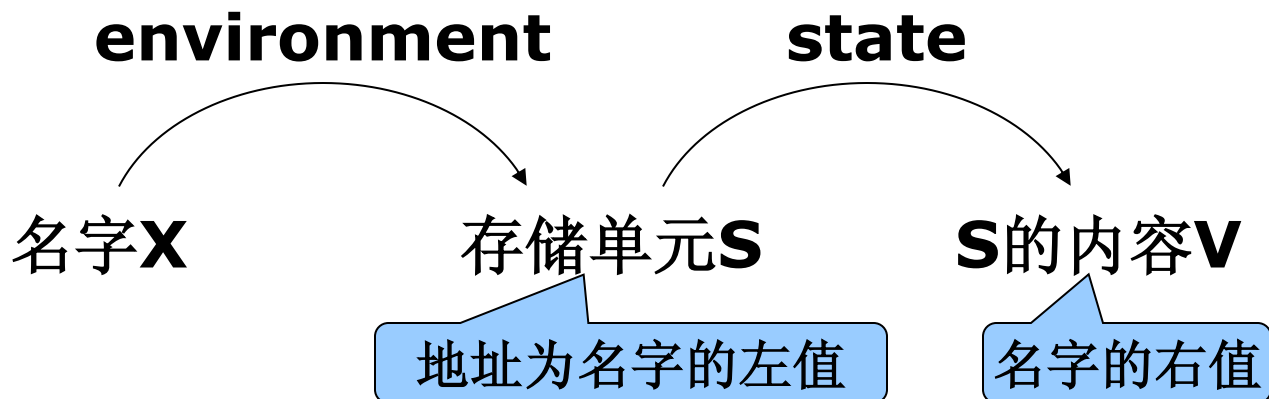
- 声明是一个把信息与名字联系起来的语法结构
  - ◆ 显式声明（如**PASCAL**中的声明：**var i:integer**）
  - ◆ 隐含声明（如**FORTRAN**程序）
- 在一个程序的不同部分可能有对同一个名字的相互独立的声明
- 一个声明起作用的程序部分称为该声明的作用域
- 语言的作用域规则决定了当这样的名字在程序正文中出现时应该使用哪一个声明
- **Pascal**中的名字遵循“最近嵌套原则”
- 编译过程中，名字的作用域信息记录在符号表中

# 名字的绑定

- 把名字对应到存储单元的过程
- 名字与存储单元的对应关系:

◆ **1:1** 一个活动中的名字与其存储单元之间

◆ **1:m** 一个递归过程中的名字与其存储单元之间



- 当**environment**把一个存储单元**S**与一个名字**X**联系起来时，称**X**受限**S**。
- **S**的大小取决于**X**的类型

# 与存储组织与管理有关的其他问题

- 名字的存储空间如何组织、名字绑定的方法等主要取决于对以下问题的回答：
  - ◆ 过程是否可递归？
  - ◆ 当控制从过程的一次活动返回时，对局部名字的值如何处理？
  - ◆ 过程是否可以引用非局部的名字？
  - ◆ 过程调用时如何传递参数？
  - ◆ 过程是否可以作为参数传递？
  - ◆ 过程可否作为结果被返回？
  - ◆ 存储空间能否在程序控制下进行动态分配？
  - ◆ 存储空间是否必须显式地归还？

## 7.2 存储分配策略

- 运行时刻存储空间的划分，除目标代码外，其余三种数据空间采用的存储分配策略是不同的。
    - ◆ 静态存储分配：编译时对所有数据对象分配存储空间
    - ◆ 栈式存储分配：运行时把存储器作为栈进行管理，数据对象分配在栈中
    - ◆ 堆式存储分配：运行时把存储器组织成堆结构，对用户提出的存储空间的申请与归还进行存储分配与回收。
- 一、静态存储分配
  - 二、栈式存储分配
  - 三、堆式存储分配

# 一、静态存储分配

- 条件：源程序中出现的各种数据所需要的存储空间的大小在**编译时可以确定**
- 存储分配：编译时, 为他们分配**固定的**存储空间
- 运行时：**总是**使用这些存储空间
  - ◆ 过程每次被激活，同一名字都使用相同的存储空间
  - ◆ 允许局部名字的值在活动结束后被保留下来
  - ◆ 当控制再次进入时，局部名字的值即上次离开时的值
- 数据对象：
  - ◆ 每个过程的活动记录的位置及大小
  - ◆ 记录中每一个名字所占用的存储空间的位置及大小
  - ◆ 数据在运行时刻的地址可以填入到目标代码中

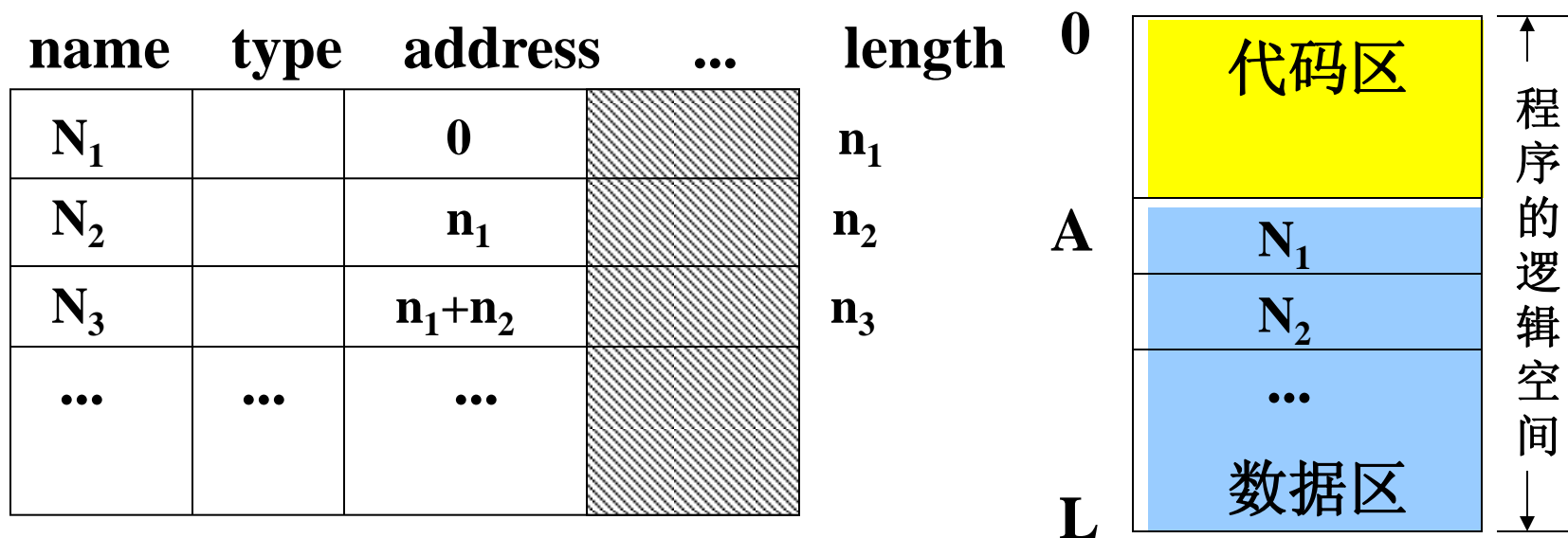


# 静态存储分配策略对源语言的限制

- 数据对象的大小和它们在内存中的位置必须在编译时都能够确定
- 不允许过程递归调用
  - ◆ 因为使用静态存储分配，一个过程里声明的局部数据在该过程的所有活动中都结合到同一个地址。
- 不能建立动态数据结构
  - ◆ 因为没有在运行时进行存储分配的机制。

# 静态存储分配策略的实现

- 编译器处理声明语句时，每遇到一个变量名就创建一个符号表条目，填入相应的属性，包括目标地址。
- 每个变量所需空间的大小由其类型确定，并且在编译时刻是已知的。
- 根据名字出现的先后顺序，连续分配空间



# Fortran程序举例

## PROGRAM CNSUME

```
CHARACTER *50 BUF
INTEGER NEXT
CHARACTER C, PRDUCE
DATA NEXT /1/, BUF /' '/
```

6

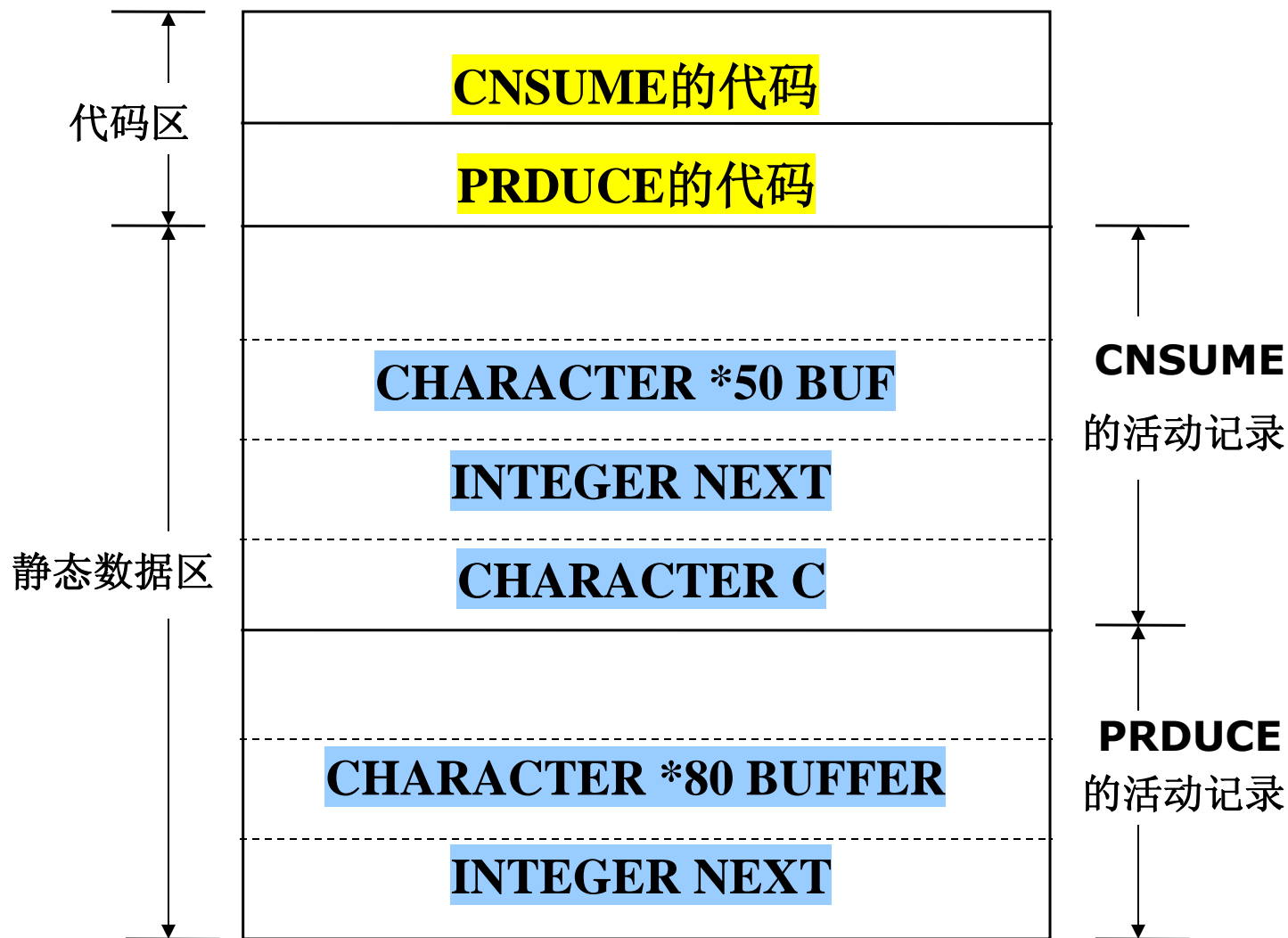
```
C=PRDUCE()
BUF (NEXT:NEXT)=C
NEXT=NEXT+1
IF (C .NE. ' ') GOTO 6
WRITE (*, '(A)') BUF
END
```

## CHARACTER FUNCTION PRDUCE()

```
CHARACTER *80 BUFFER
INTEGER NEXT
SAVE BUFFER, NEXT
DATA NEXT /81/
```

```
IF (NEXT .GT. 80) THEN
    READ (*, '(A)') BUFFER
    NEXT=1
END IF
PRDUCE=BUFFER (NEXT:NEXT)
NEXT=NEXT+1
END
```

# 存储空间分配

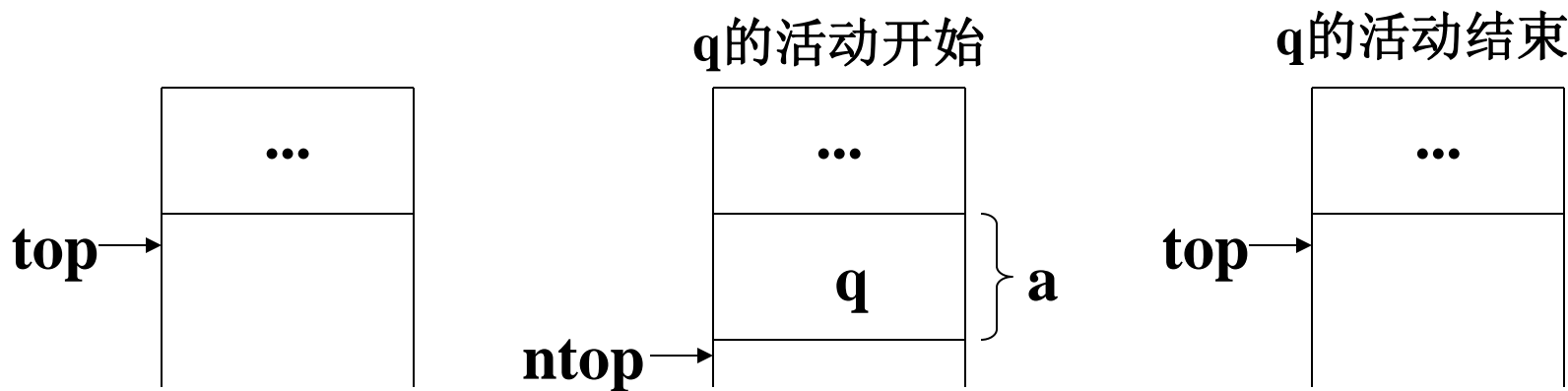


## 二、栈式存储分配

### ■ 存储空间被组织成栈

### ■ 存储管理

- ◆ 活动开始时，与活动相应的活动记录入栈，局部变量的存储空间分配在该活动记录中。同一过程中声明的名字在不同的活动中被结合到不同的存储空间。
- ◆ 活动结束时，活动记录出栈，分配给局部名字的存储空间被释放。名字的值将丢失，不可再用。

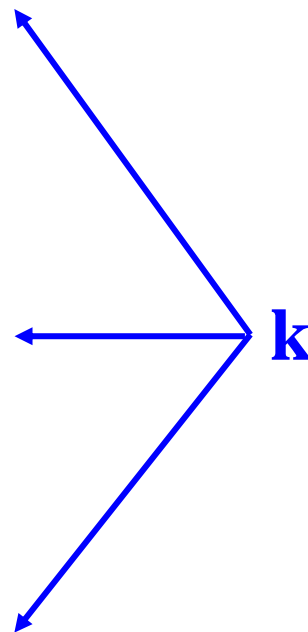
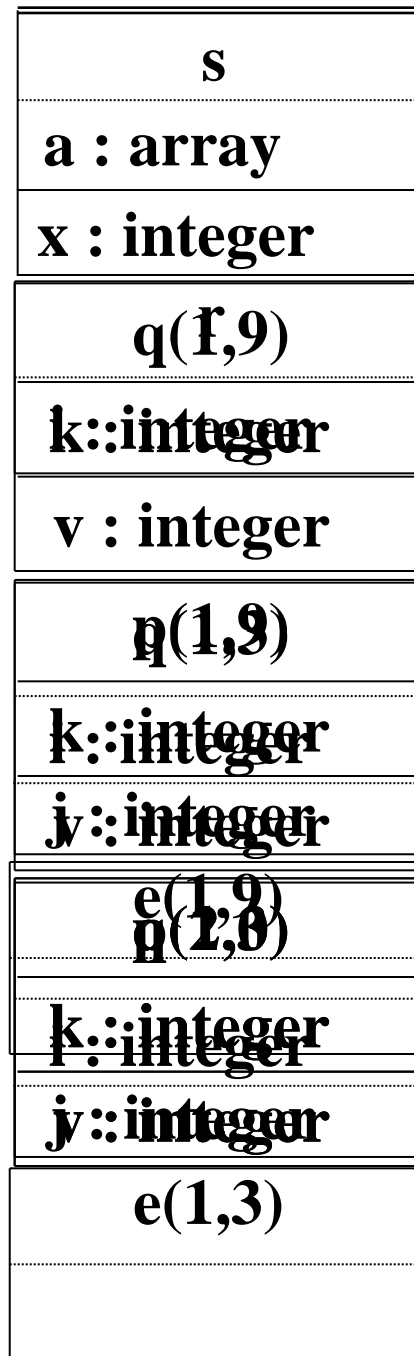
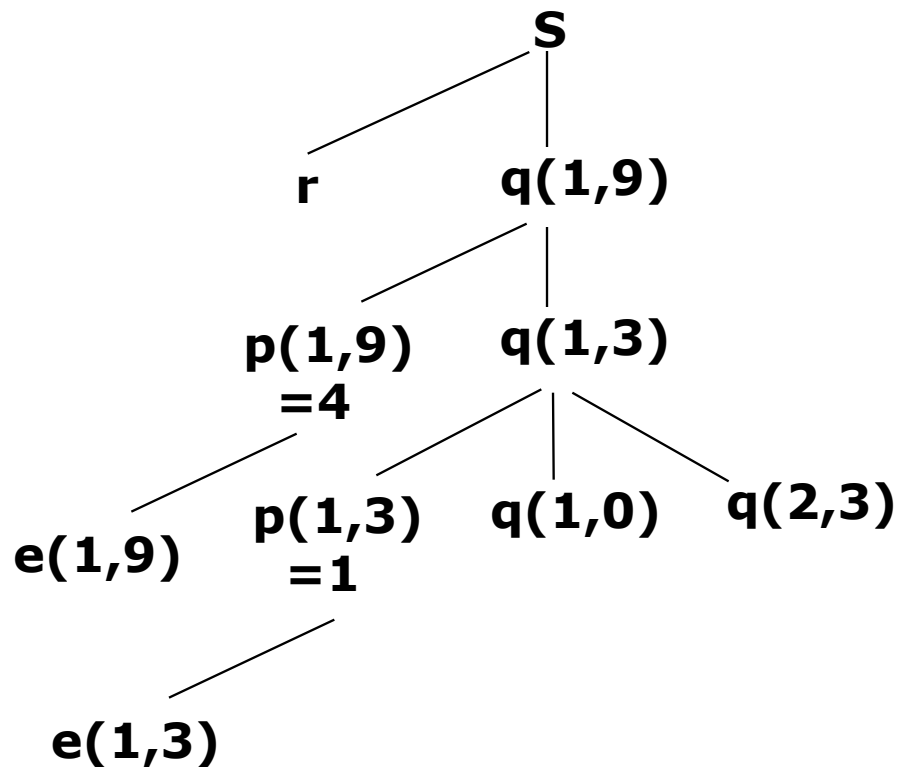


# 对读入的数据进行排序的PASCAL程序

```
program sort(input, output);  
  var a: array[0..10] of integer;  
      x: integer;  
  procedure readarray;  
    var i: integer;  
    begin  
      for i:=1 to 9 do read(a[i])  
    end;  
  prcedure exchange(i, j: integer);  
  begin  
    x:=a[i]; a[i]:=a[j]; a[j]:=x  
  end;
```

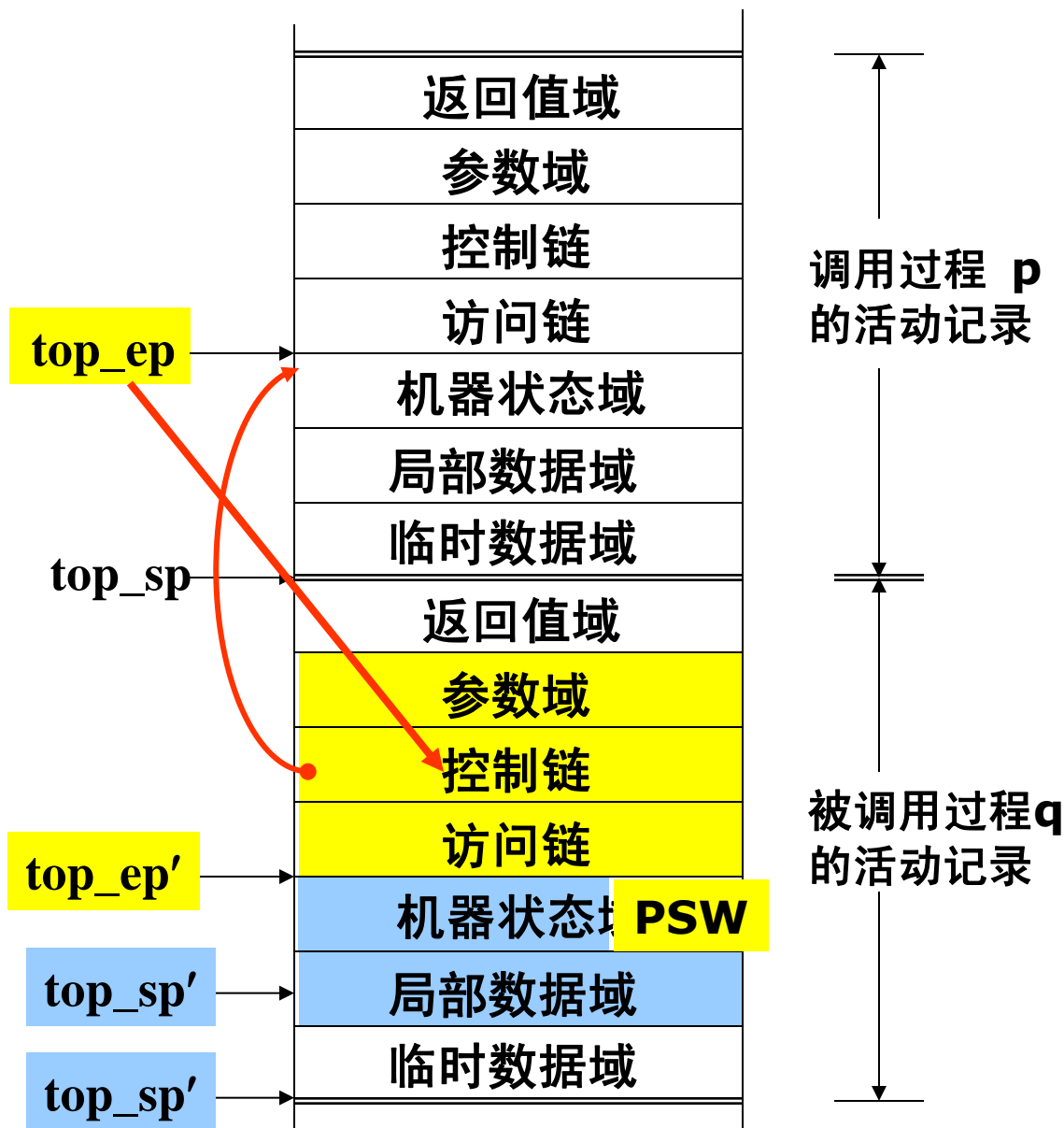
```
  procedure quicksort(m, n: integer);  
    var k, v: integer;  
    function partition(y, z: integer): integer;  
      var i, j: integer;  
      begin  
        ... a ...;  
        ... v ...;  
        exchange(i, j);  
      end;  
    begin if (n>m) then begin  
      i:=partition(m, n);  
      quicksort(m,i-1);  
      quicksort(i+1, n)  
    end  
  end {quicksort};  
begin a[0]:=-999; a[10]=999;  
  readarray; quicksort(1, 9)  
end {sort}.
```

# 控制栈的变化举例



# 调用序列

- 除局部数据外，活动记录中还有实现过程调用和返回的控制信息
- 活动记录的入栈，实现了控制从调用过程到被调用过程的转移
- 控制的转移由一段代码（即**调用序列**）来实现





# 调用序列的安排

- 参数传递：
  - ◆ p计算实参的值，写入q的活动记录的参数域；
- 控制信息设置：
  - ◆ p将返回地址写入q的活动记录的机器状态域中
  - ◆ p将当前的top\_ep的值写入q的活动记录的控制链域
  - ◆ p为q建立访问链
  - ◆ p设置新的top\_ep的值(指向q的活动记录中某个位置)
- 进入q的代码(goto语句)
- q保存寄存器的值、以及其他机器状态信息
- q增加top\_sp的值，初始化局部变量
- 开始执行

# 返回序列

- **q**把返回值写入自己活动记录的返回值域
- **q**恢复断点状态：
  - ◆ 寄存器的值
  - ◆ **top\_sp**和**top\_ep**的值
  - ◆ 机器状态
- 根据返回地址返回到**p**的代码中（**goto**语句）
- **p**把返回值取入自己的活动记录中
- **p**继续执行

# 调用序列与活动记录

## ■ 区别：

- ◆ 活动记录是一块连续的存储区域，保存一个活动所需的全部信息，与活动一一对应。
- ◆ 调用序列是一段代码，完成活动记录的入栈，实现控制从调用过程到被调用过程的转移。
- ◆ 调用序列逻辑上是一个整体，物理上被分成两部分，分属于调用过程和被调用过程。

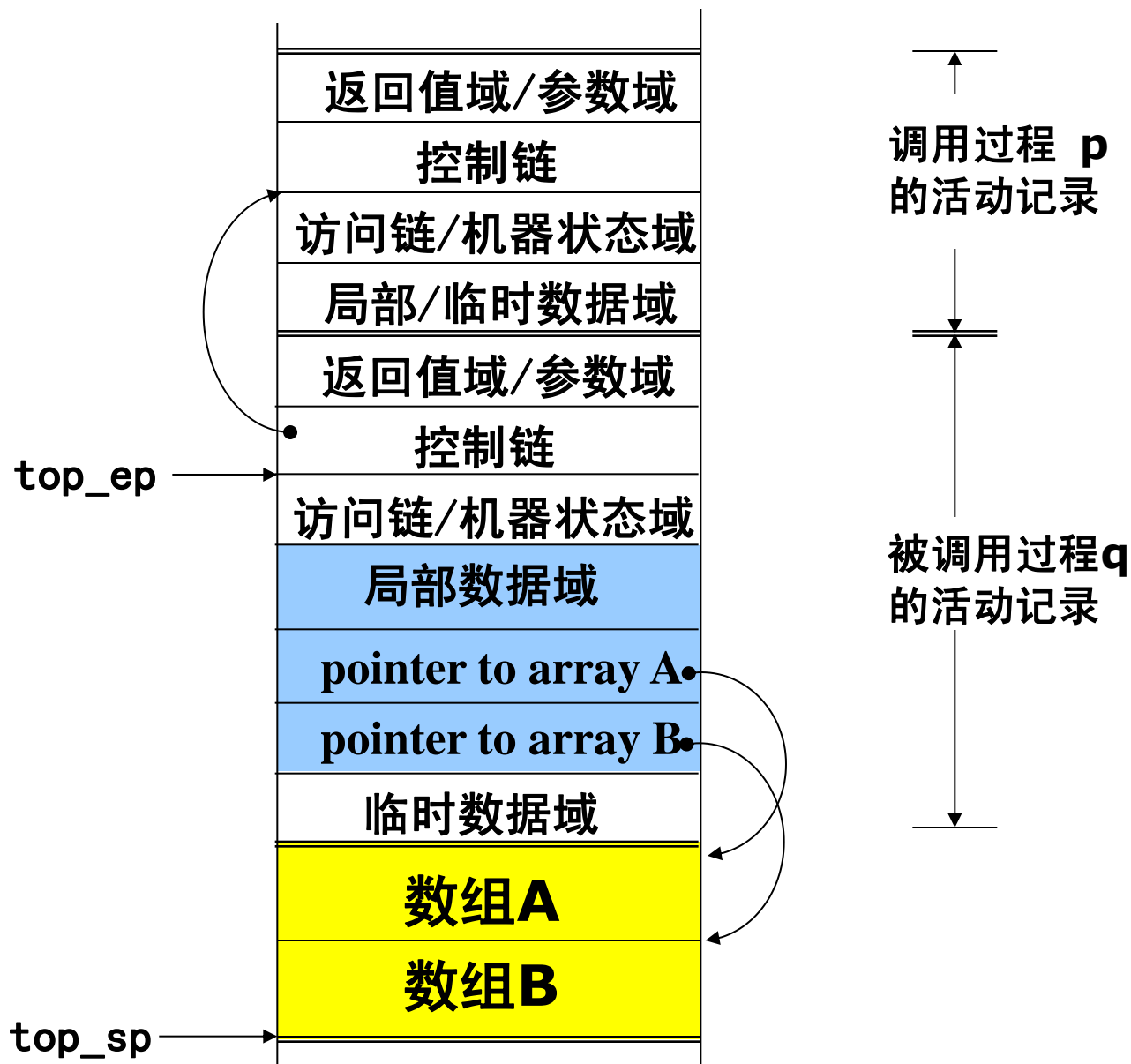
## ■ 联系：

- ◆ 调用序列的实现与活动记录中内容的安排有密切关系

# 可变长数据的处理

- 某些语言允许由实参的值决定被调用过程中局部数组的大小——可变数组
- 可变数组的大小只有到执行过程调用时才能确定
- 编译时可以确定数组的个数
- 活动记录中只设置相应于可变数组的指针，而不包括这些数组的数据空间
- 可变数组的数据空间放在活动记录之外

# 可变数组的空间分配



# 三、堆式存储分配

- 如果具体的存储需求在编译时刻可以确定  
——采取静态存储分配策略
  - 如果某些存储需求在编译时不能确定，但在程序执行期间，在程序的入口点上可以知道  
——采用栈式存储分配策略
  - 栈式存储分配策略不能处理的存储需求：
    - ◆ 程序设计语言中的某些数据结构的存储需求
    - ◆ 活动停止时局部名字的值必须被保存下来
    - ◆ 被调用过程的活动生存期超过调用过程的生存期，这种语言的过程间的控制流不能用活动树正确地描述。
- 共性：**活动记录的释放不需要遵循先进后出的原则

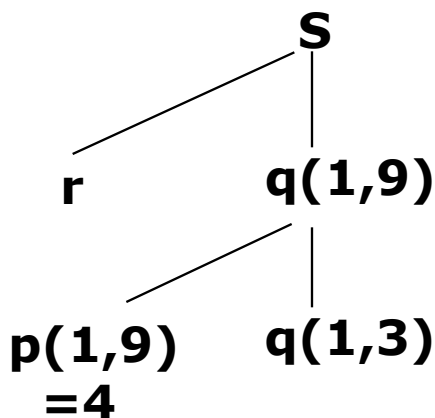
# 堆式存储分配与栈式存储分配的比较

## ■ 相同点:

- ◆ 动态存储分配

## ■ 不同点:

- ◆ 组织形式: 栈、堆
- ◆ 释放顺序:  
栈: 先进后出  
堆: 任意



## 7.3 访问非局部名字

- 对非局部名字的引用取决于**作用域规则**
  - 静态作用域规则：词法作用域规则、**最近嵌套**规则
  - 动态作用域规则：由运行时最近的活动决定可应用到一个名字上的声明
- 对非局部名字的访问通过**访问链**实现
- 关键：访问链如何创建、使用、维护

一、程序块

二、非嵌套过程的静态作用域

三、嵌套过程的静态作用域

四、动态作用域



# 一、程序块

## ■ 程序块的基本结构

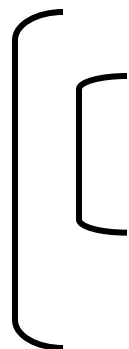
**begin**

声明语句

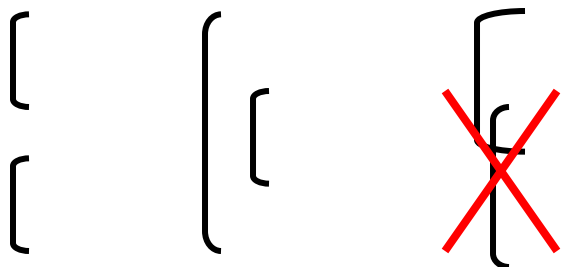
语句序列

**end**

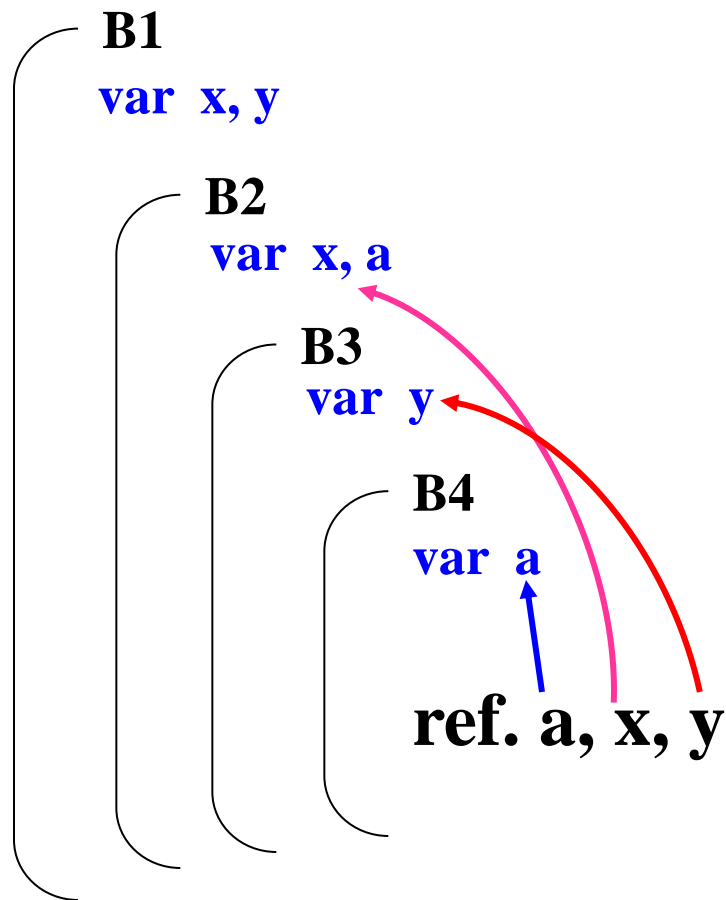
## ■ 块可以嵌套



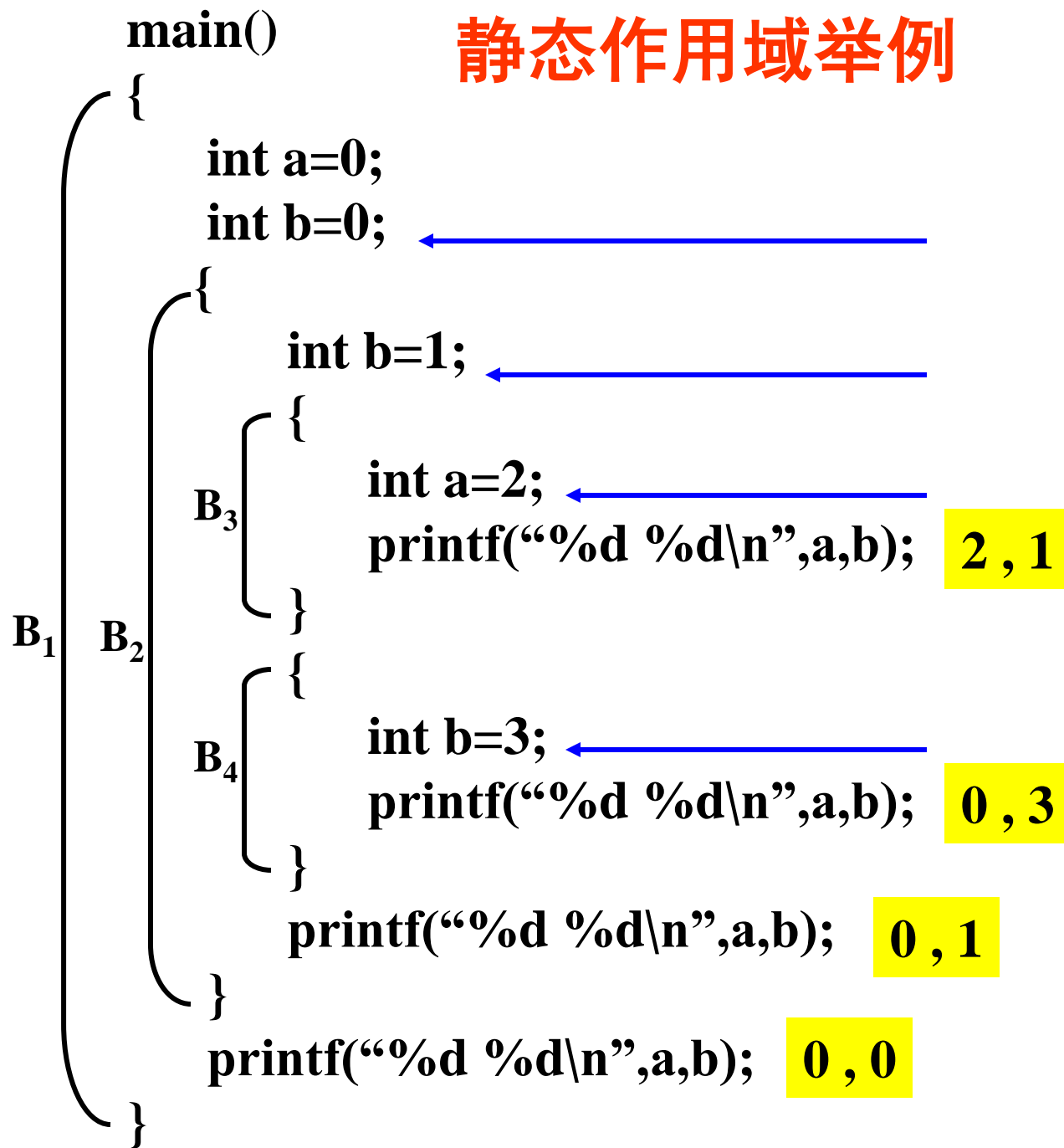
## ■ 块之间的关系



## ■ 最近嵌套规则



# 静态作用域举例



|            |
|------------|
| <b>a=0</b> |
| <b>b=0</b> |
| <b>b=1</b> |
| <b>b=3</b> |

## 二、非嵌套过程的静态作用域

- 过程定义不允许嵌套
  - ◆ 独立的
  - ◆ 顺序的
- 声明语句的位置
  - ◆ 过程/函数内部
  - ◆ 所有过程之外
- 在一个过程中引用的名字
  - ◆ 局部的
  - ◆ 全局的

```
int a[11];
readarray()
{ ... a ... }
int partition(int y, int z)
{ ... a ... }
quicksort(int m, int n)
{ int i; ...
  i=partition(m,n);
  quicksort(m,i-1);
  quicksort(i+1,n); }
main ()
{ readarray;
  quicksort(0,10); }
```

# 变量的存储分配

## ■ 全局变量

- ◆ 静态地进行分配
- ◆ 分配在静态数据区中
- ◆ 编译时知道它们的位置，可以将全局变量对应的存储单元的地址编入目标代码中

## ■ 局部变量

- ◆ 动态地进行分配
- ◆ 分配在活动记录中
- ◆ 栈式存储分配
- ◆ 通过对栈指针的偏移访问当前活动记录中的局部名字

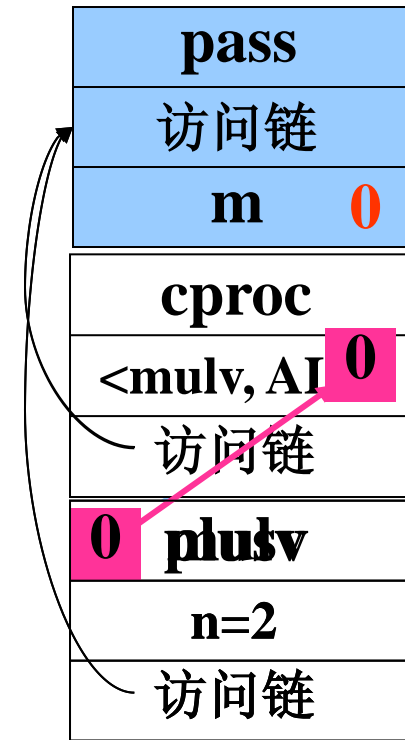


# 过程作为参数传递、作为结果返回

- 对非局部名字采用静态存储分配
- 前提（源语言）
  - ◆ 静态作用域规则
  - ◆ 过程定义不允许嵌套
- 效果
  - ◆ 一个过程的非局部名，对其他过程也是非局部的
  - ◆ 非局部名字的存储空间由编译程序静态地进行分配
  - ◆ 它的静态地址在所有的过程中都可以引用
  - ◆ 过程对非局部名字的引用与过程是如何激活的无关
- 结果
  - ◆ 无论过程作为参数传递、还是作为结果返回，其中对非局部名字的引用，都是对静态分配给它的存储单元进行访问。

# 示例

```
int m;  
int plusv( int n)  
{ return m+n; }  
int mulv( int n)  
{ return m*n; }  
void cproc( int pform( int n) )  
{ cout << pform(2) <<endl ; }  
void main( ) {  
    m=0;  
    cproc(plusv);  
    cproc(mulv);  
}
```



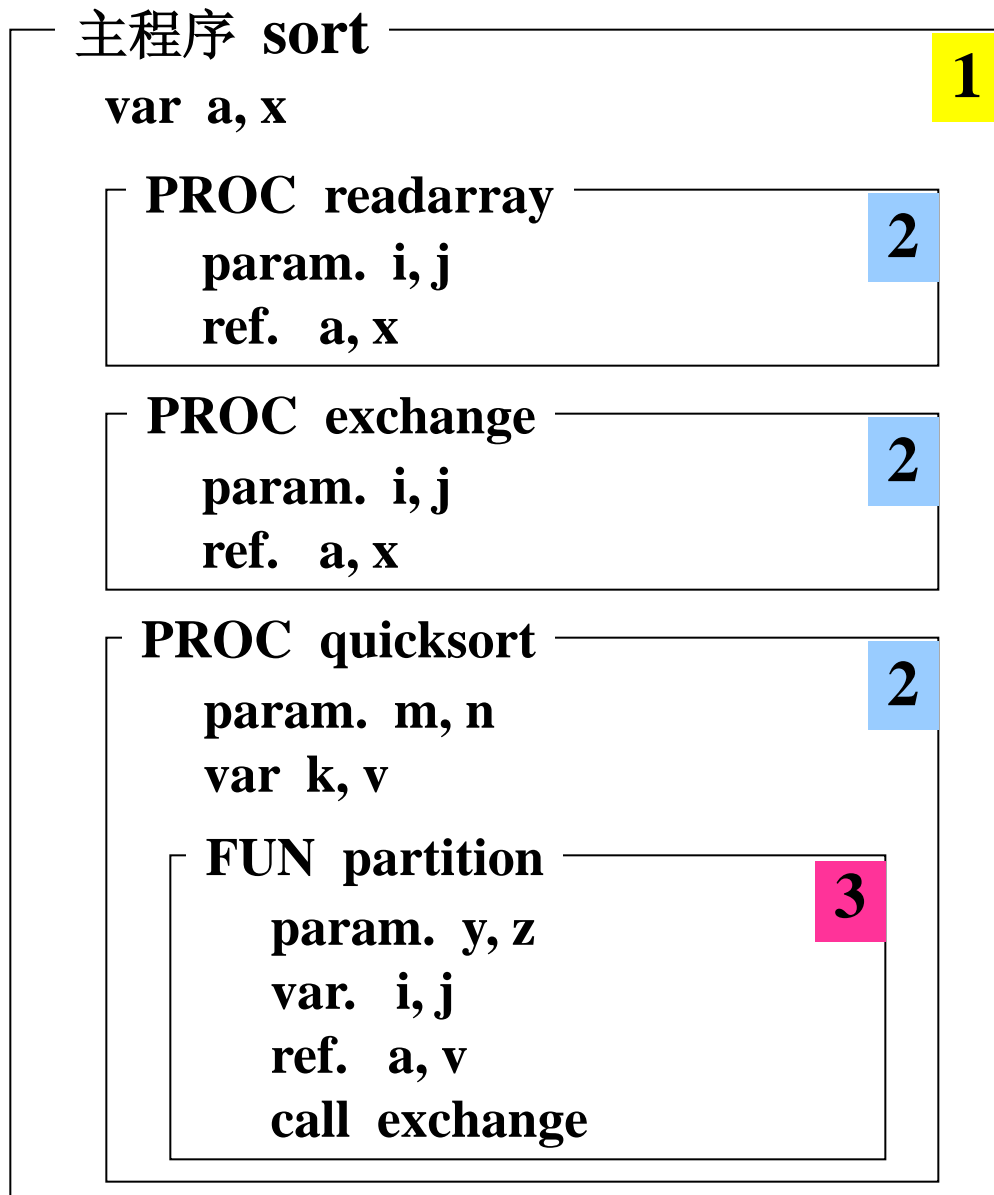
# 三、嵌套过程的静态作用域

- 过程定义允许嵌套
- 最近嵌套规则

```
Program sort(input,output);  
  var a: array[0..10] of integer;  
      x: integer;  
  procedure readarray;  
    var i: integer;  
    begin ... a ... end;  
  prcedure exchange(i,j:integer);  
    begin  
      x:=a[i]; a[i]:=a[j]; a[j]:=x  
    end;
```

```
procedure quicksort(m,n:integer);  
  var k,v: integer;  
  function partition(y,z:integer):integer;  
    var i,j: integer;  
    begin ... a ...;  
          ... v ...;  
          exchange(i,j);  
          ...  
    end;  
    begin ... end {quicksort};  
begin ... ...end {sort}.
```

# 过程及名字的嵌套关系



## ■ 嵌套深度

- ◆ 主程序：1
- ◆ 每进入一个过程，深度加1

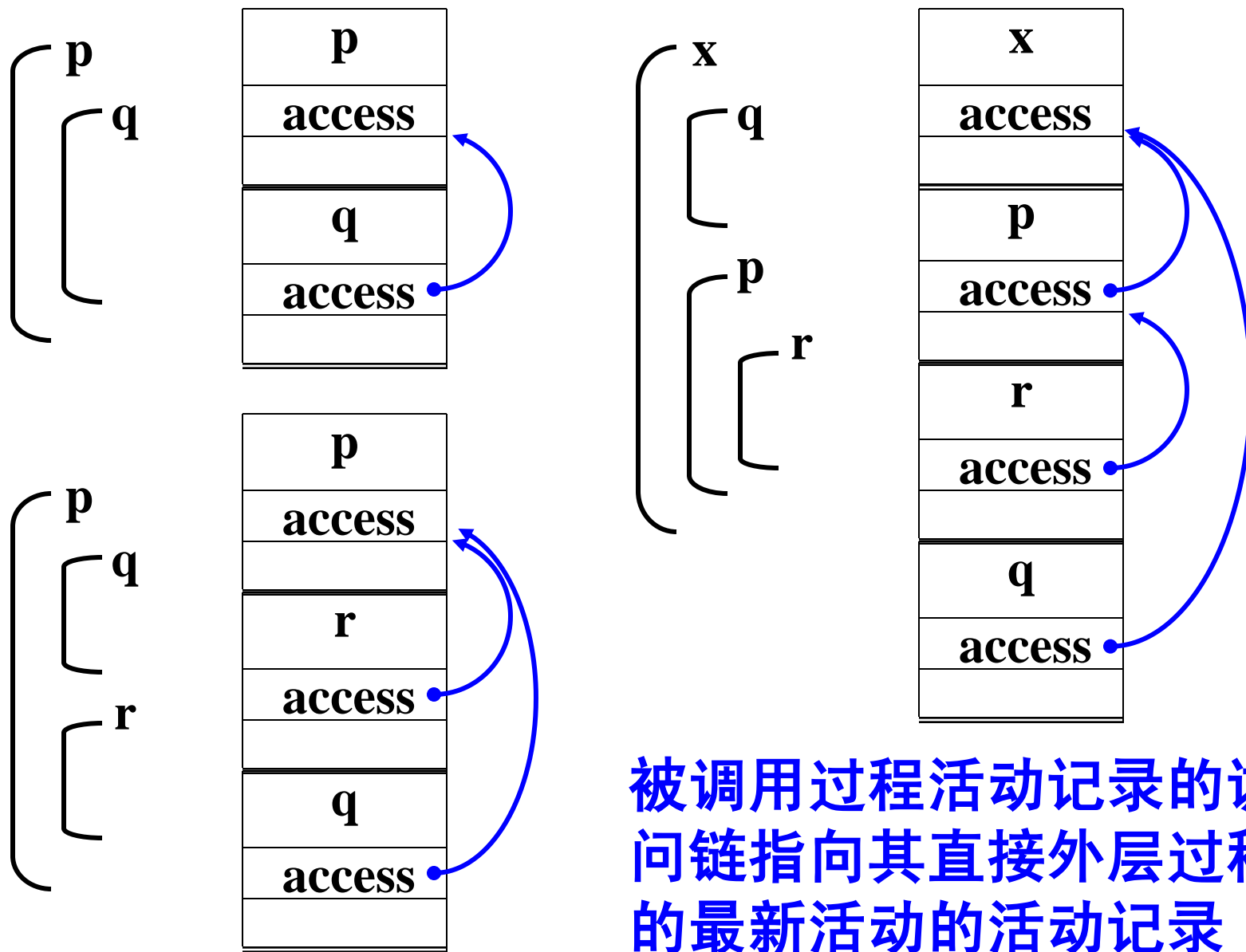
## ■ 名字的嵌套深度

- 声明时所在过程的嵌套深度



# 访问链

- 实现嵌套过程的静态作用域规则
- 通过访问链可以实现对非局部名字的访问



# 访问链的使用

- 过程 $p$ 引用非局部名字 $a$ ，嵌套深度分别为 $n_p$ ， $n_a$

- ◆  $n_a < n_p$

- 当控制处于 $p$ 中时， $p$ 的活动记录在栈顶

- 访问链的使用

该值在编译时可以计算出来

- ◆ 从栈顶活动记录出发，沿访问链前进  $n_p - n_a$  步

- ◆ 到达 $a$ 的声明所在过程的最新活动记录

- ◆ 在该活动记录中，相对于访问链的某个固定偏移处即 $a$ 的存储位置

该值在编译时可以计算出来

- 符号表中，变量名字的目标地址：

- $\langle \text{嵌套深度}, \text{偏移量} \rangle$

- $p$ 中非局部名字 $a$ 的地址：

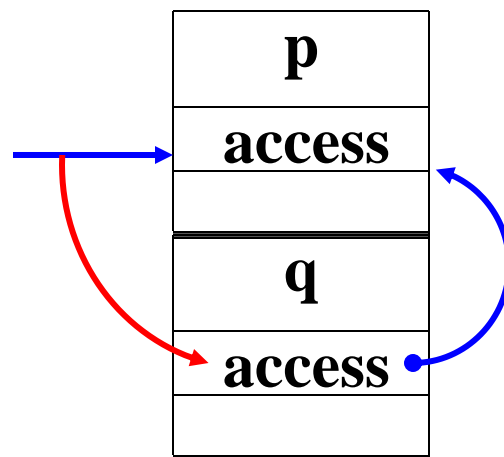
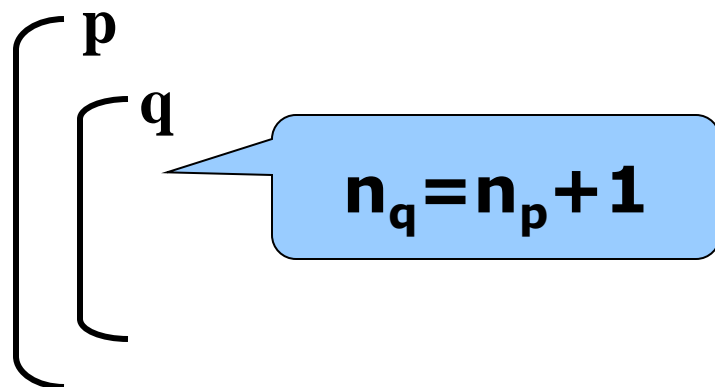
- $\langle n_p - n_a, a \text{在活动记录中相对于访问链的偏移} \rangle$

# 访问链的建立

- 在调用序列中由调用过程 $p$ 创建被调用过程 $q$ 的访问链
- 过程 $p$ 和 $q$ 的嵌套深度分别为 $n_p$ 和 $n_q$
- $q$ 的活动记录中访问链的建立依赖于 $q$ 与 $p$ 的关系
  - ◆  $q$ 嵌套在 $p$ 中
    - $n_q > n_p$
  - ◆  $q$ 不嵌套在 $p$ 中
    - $n_q = n_p$
    - $n_q < n_p$

# q嵌套在p中—— $n_q > n_p$

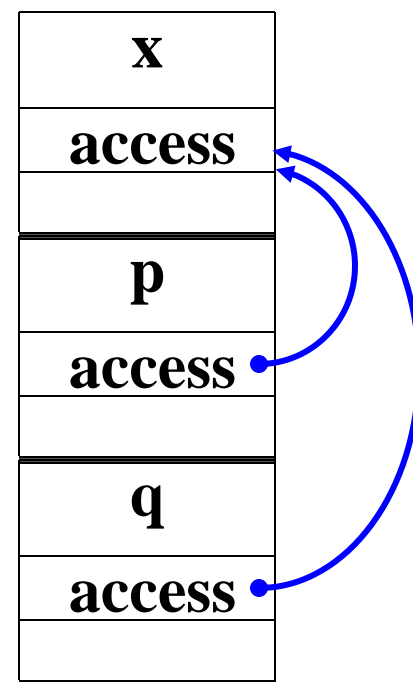
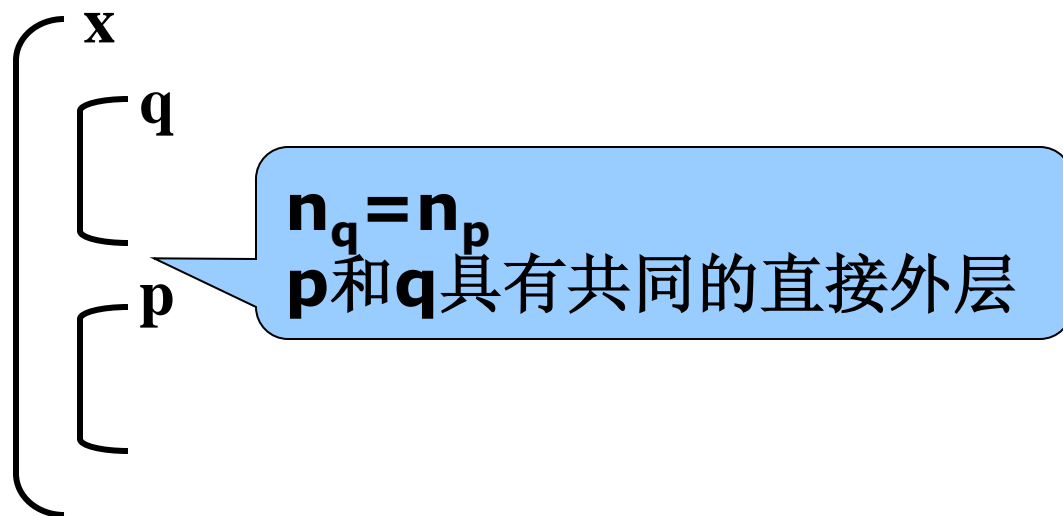
- 静态文本中q与p的关系:



- q活动记录中的访问链指向栈中刚好在其前面的p的活动记录的访问链
- 调用序列中, p把top-ep的值写入q的活动记录的访问链

# q不嵌套在p中—— $n_q = n_p$

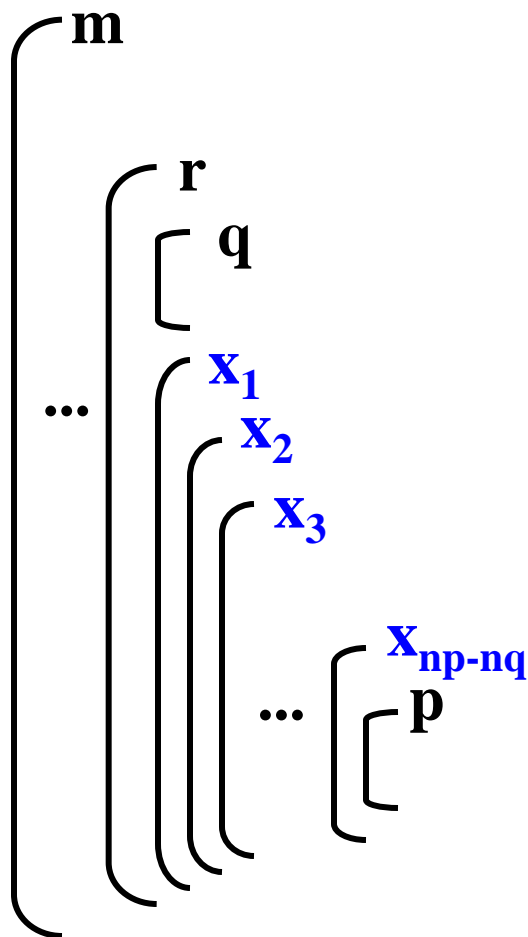
- 静态文本中q与p的关系:



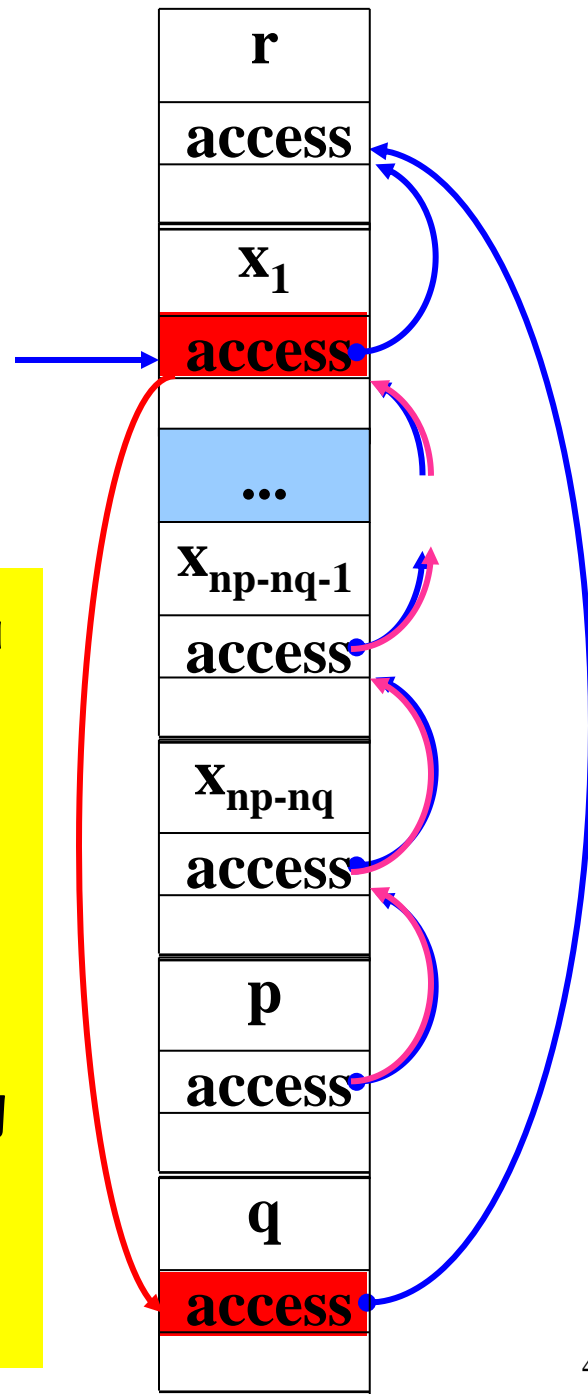
- q活动记录中的访问链与p的活动记录的访问链指向栈中刚好在p前面的x的活动记录访问链
- 调用序列中，p把自己的访问链的值复制<sup>复制</sup>到q的活动记录的访问链

# q不嵌套在p中—— $n_q < n_p$

## ■ 静态文本中q与p的关系:



- 从p的活动记录出发，沿访问链前进 $n_p - n_q$ 步；
- 找到 $x_1$ 的活动记录访问链
- 把 $x_1$ 的访问链的值复制到q的活动记录中访问链域



# 访问链举例

(a)

|               |
|---------------|
| <b>s</b>      |
| <b>access</b> |
| <b>a, x</b>   |
| <b>q(1,9)</b> |
| <b>access</b> |
| <b>k, v</b>   |

(b)

|               |
|---------------|
| <b>s</b>      |
| <b>access</b> |
| <b>a, x</b>   |
| <b>q(1,9)</b> |
| <b>access</b> |
| <b>k, v</b>   |
| <b>q(1,9)</b> |
| <b>access</b> |
| <b>k, v</b>   |

(c)

|               |
|---------------|
| <b>s</b>      |
| <b>access</b> |
| <b>a, x</b>   |
| <b>q(1,9)</b> |
| <b>access</b> |
| <b>k, v</b>   |
| <b>q(1,3)</b> |
| <b>access</b> |
| <b>k, v</b>   |
| <b>p(1,3)</b> |
| <b>access</b> |
| <b>i, j</b>   |

(d)

|               |
|---------------|
| <b>s</b>      |
| <b>access</b> |
| <b>a, x</b>   |
| <b>q(1,9)</b> |
| <b>access</b> |
| <b>k, v</b>   |
| <b>q(1,3)</b> |
| <b>access</b> |
| <b>k, v</b>   |
| <b>p(1,3)</b> |
| <b>access</b> |
| <b>i, j</b>   |
| <b>e(1,3)</b> |
| <b>access</b> |
|               |

# display表

- 目的：为了提高访问非局部名字的速度
- **display表**：
  - ◆ 指针数组d
  - ◆ 每一个指针指向一个活动记录
  - ◆ **d[i]**指向嵌套深度为i的过程的最新活动的活动记录
  - ◆ 全程数组
  - ◆ 元素个数在编译时刻根据过程的最大嵌套深度确定
  - ◆ 静态存储分配
  - ◆ 控制栈中，具有相同嵌套深度j的各活动记录，从靠近栈顶的最新活动记录开始，通过访问链链成一个链表
  - ◆ **d[j]**是该链表的头指针



|      |  |
|------|--|
| d[3] |  |
| d[2] |  |
| d[1] |  |

|        |
|--------|
| S      |
| NIL    |
| a, x   |
| q(1,9) |
| NIL    |
| k, v   |

|      |  |
|------|--|
| d[3] |  |
| d[2] |  |
| d[1] |  |

|           |
|-----------|
| S         |
| NIL       |
| a, x      |
| q(1,9)    |
| NIL       |
| k, v      |
| q(1,9)    |
| save d[2] |
| k, v      |

|      |  |
|------|--|
| d[3] |  |
| d[2] |  |
| d[1] |  |

|           |
|-----------|
| S         |
| NIL       |
| a, x      |
| q(1,9)    |
| NIL       |
| k, v      |
| q(1,3)    |
| save d[2] |
| k, v      |
| p(1,3)    |
| NIL       |
| i, j      |

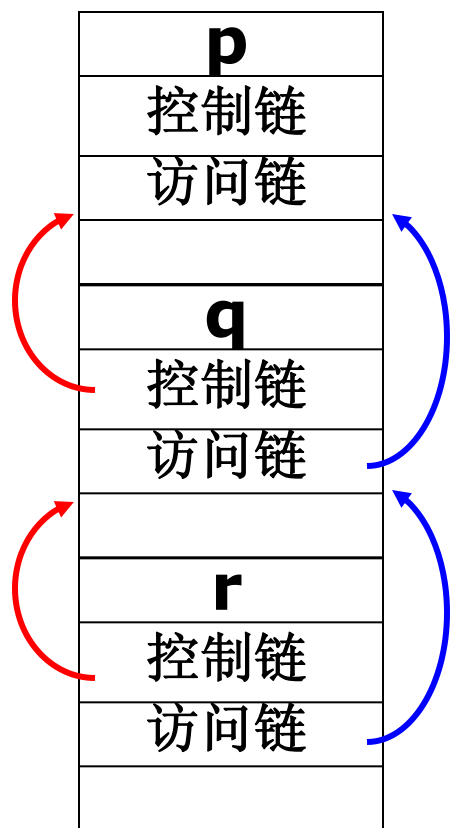
|      |  |
|------|--|
| d[3] |  |
| d[2] |  |
| d[1] |  |

|           |
|-----------|
| S         |
| NIL       |
| a, x      |
| q(1,9)    |
| NIL       |
| k, v      |
| q(1,3)    |
| save d[2] |
| k, v      |
| p(1,3)    |
| NIL       |
| i, j      |
| e(1,3)    |
| save d[2] |
|           |

## d表应用举例

## 四、动态作用域

- 名字的作用域由程序执行过程中，过程之间的调用与被调用关系决定
- 如过程p调用q，q又调用r



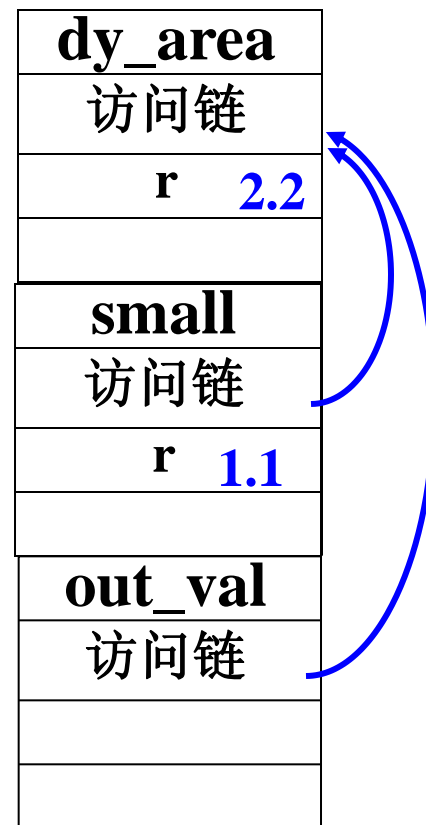
- 访问链的指向：从被调用过程指向调用过程
- 访问链与控制链的指向相同
- 为了访问非局部名字，可能会沿控制链/访问链进入栈的深处——**深访问**

# 举例

## ■ 程序

```
program dy_area(input,output);  
  var r: real;  
  procedure out_val;  
    begin write(r:5:3) end;  
  procedure small;  
    var r: real;  
    begin r:=1.1; out_val end;  
begin  
  r:=2.2; ←  
  out_val; small; writeln;  
  out_val; small; writeln;  
end.
```

## ■ 静态作用域情况

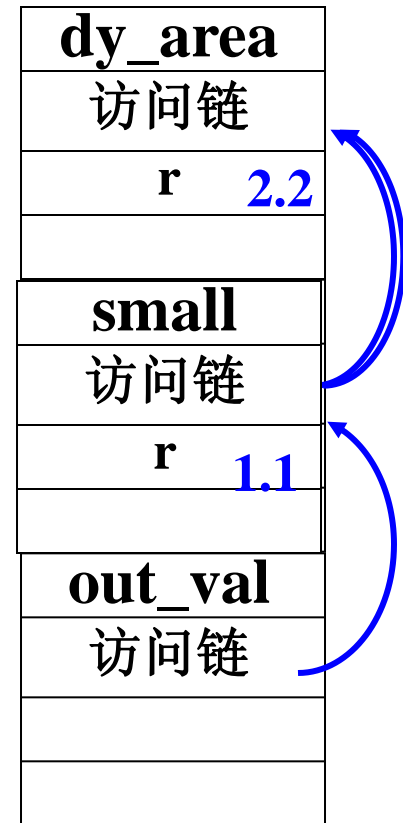


2.2      2.2  
2.2      2.2

# 动态作用域情况

## ■ 程序

```
program dy_area(input,output);  
  var r: real;  
  procedure out_val;  
    begin write(r:5:3) end;  
  procedure small;  
    var r: real;  
    begin r:=1.1; out_val end;  
begin  
  r:=2.2; ←  
  out_val; small; writeln;  
  out_val; small; writeln;  
end.
```



2.2      1.1  
2.2      1.1

# 浅访问方式

- 动态作用域下，名字可以采用“浅访问”方式进行存储分配
- 做法：
  - ◆ 为每个名字在静态分配的存储空间中保存它的值
  - ◆ 当控制进入过程 $p$ 时， $p$ 的活动记录入栈， $p$ 中的局部名字 $a$ 接管静态分配给该名字的存储单元，而将该名字的当前值保存在 $p$ 的活动记录中。
  - ◆ 控制在活动 $p$ 中时，对名字 $a$ 的访问总是对静态分配给 $a$ 的存储单元进行存取操作。
  - ◆ 当 $p$ 的活动结束时，再用保存的值恢复 $a$ 的存储单元。

## 7.4 参数传递机制

### ■ 过程之间通信的方式:

- 文件
- 非局部名字
- 参数

一、传值调用

二、引用调用

三、复制恢复

四、传名调用

```
program reference(input,output);  
  var  a, b : integer;  
  procedure swap(x, y : integer);  
    var  temp : integer;  
    begin  
      temp:=x;  
      x:=y;  
      y:=temp  
    end;  
  begin  
    a:=1; b:=2;  
    swap(a, b);  
    writeln('a=', a);  
    writeln('b=', b)  
  end.
```

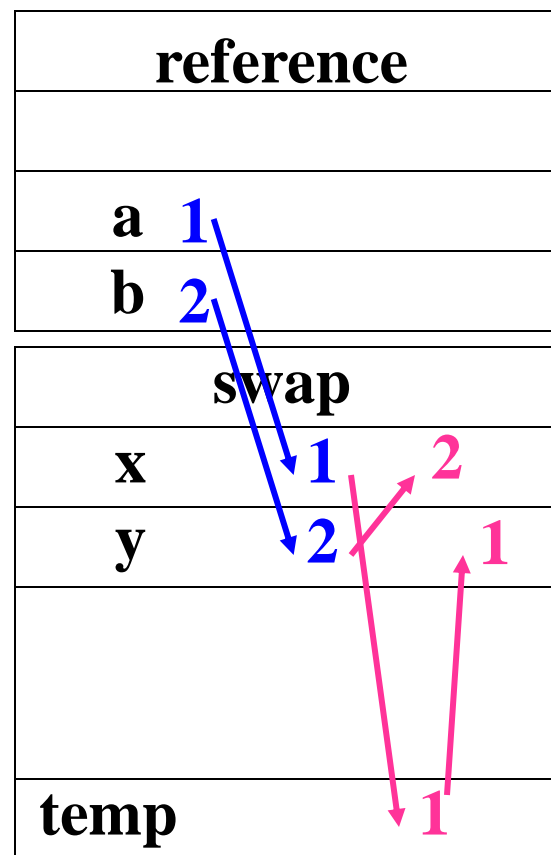
# 一、传值调用

- 最一般、最简单的参数传递方法
- 先计算出实参的值，然后将其右值传递给被调用过程，参数值在过程执行时如同常数。
- 如：有函数声明：  

```
int max(int x, int y) { return x>y? x : y ; }
```
- `max (5, 3+4) :`
  - ◆ 将x替换为5，y替换为7，得到：5>7? 5:7。
- 用相应的实参的值替代过程体中出现的所有形参。
- 是C++和Pascal语言的内置机制，本质上，也是C语言和Java语言参数传递的唯一机制。
- 在这些语言中，参数被看作是过程的局部变量，初值由调用时的实参给出。
- 在过程中，参数和局部变量一样可以被赋值，但其结果不影响过程体之外的变量的值。

# 传值调用的实现

- 调用过程对实参求值，并把实参的**右值**写入被调用过程活动记录的形参存储单元中。
- 被调用过程在形参上的操作不影响调用过程活动记录中的值
- 执行结果  
    **a=1**  
    **b=2**





# 参数是指针类型的情况

- 传值调用并不意味着参数的使用一定不会影响过程体外变量的值。
- 如果参数的类型为指针或引用，参数的值就是一个地址
  - ◆ 通过它可以改变过程体外部的内存值。如，有C语言函数  

```
void init_ptr(int* p)
{ *p=3; }
```
  - ◆ 对参数p的直接赋值不会改变过程体外的实参的值。如：  

```
void init_ptr(int* p)
{ p=(int*) malloc(sizeof(int)); }
```
- 在一些语言中，某些值是隐式指针或引用
  - ◆ 如C语言中的数组是隐式指针（指向数组的第一个位置）
  - ◆ 可以使用数组参数来改变存储在数组中的值。如：  

```
void init_array_0(int p[])
{ p[0]=0; }
```

## 二、引用调用

- 原则上要求：实参必须是已经分配了存储空间的变量。
- 调用过程把一个指向实参存储单元的指针传递给被调用过程的相应形参
- 在目标代码中，被调用过程通过传递给形参的指针间接地引用实参，因此，可以把形参看成是实参的别名，任何对形参的引用就是对相应实参的引用。
- Fortran语言中唯一的参数传递机制。
- 在Pascal语言中，通过在形参前加关键字var来指定采用引用调用机制。

```
procedure inc_1(var x: integer);  
begin x:=x+1 end;
```

- 在C++中，通过在形参的类型关键字后加符号 ‘&’来指明采用引用调用机制，如：

```
void inc_1(int& x) { x++ ; }
```

# 引用调用

- C语言可以通过传递引用或显式指针来实现引用调用的效果
- C语言使用 ‘&’ 指示变量的地址，使用操作 ‘\*’ 撤销引用指针。
- 如：

```
void inc_1(int* x)           // C 模拟引用调用
{ (*x)++; }

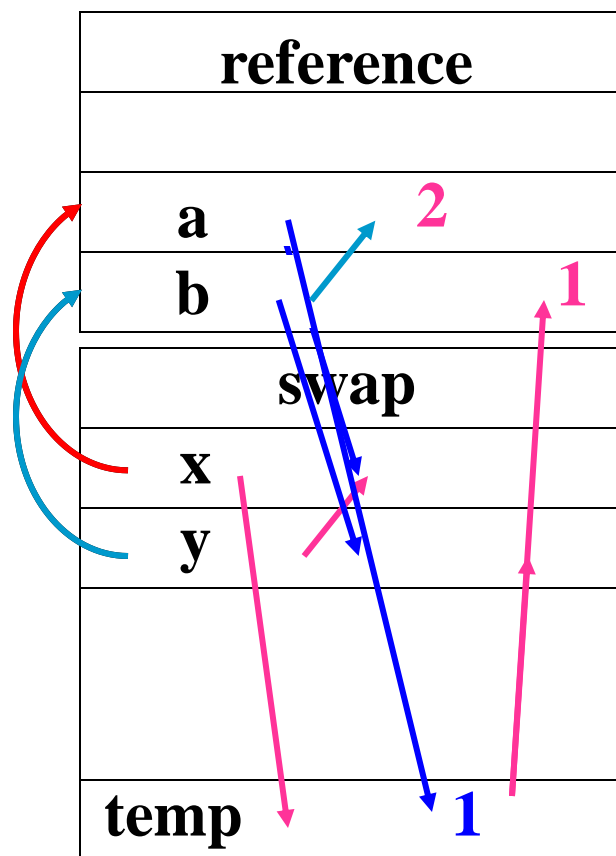
.....
int a;

.....
inc_1(&a);
```

# 引用调用的实现

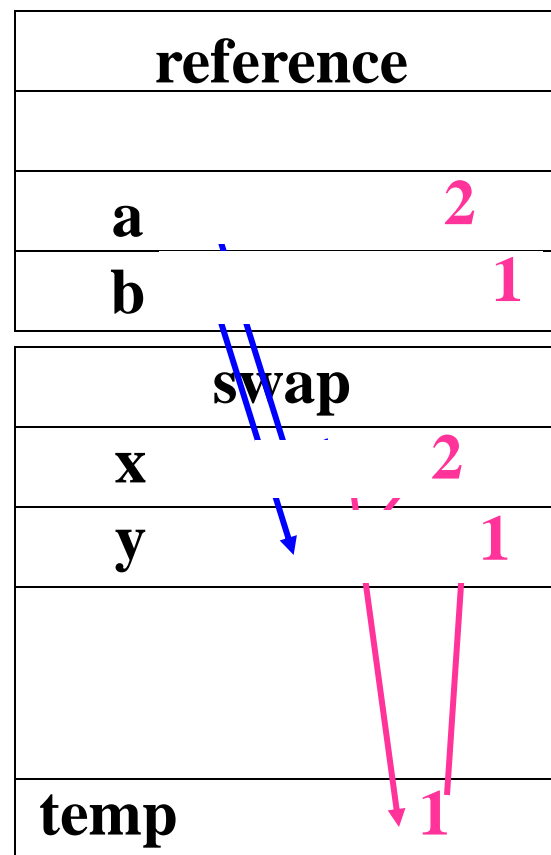
- 调用过程对实参求值，并把实参的左值写入被调用过程活动记录的形参存储单元中。
- 若实参是表达式，计算表达式的值，并把它存入临时存储单元，然后传递这个单元的地址。
- 被调用过程通过形参间接地引用实参
- 执行结果

**a=2      b=1**



### 三、复制恢复

- 传值调用和引用调用的一种混合形式
- 调用过程计算实参，实参的右值被传递到被调用过程中。要求在调用之前求出实参的左值。
- 从被调用过程返回时，把形参的现行右值复制到相应实参的左值中。只有具有左值的实参被复制下来



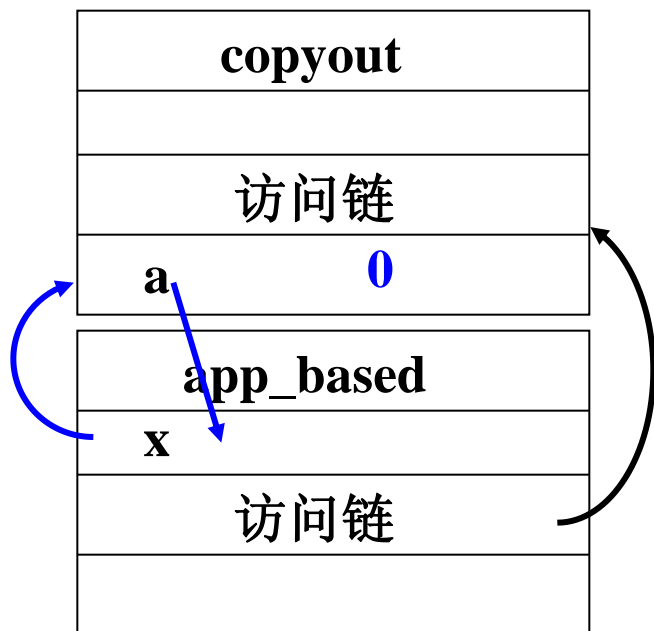
- 执行结果

**a=2    b=1**

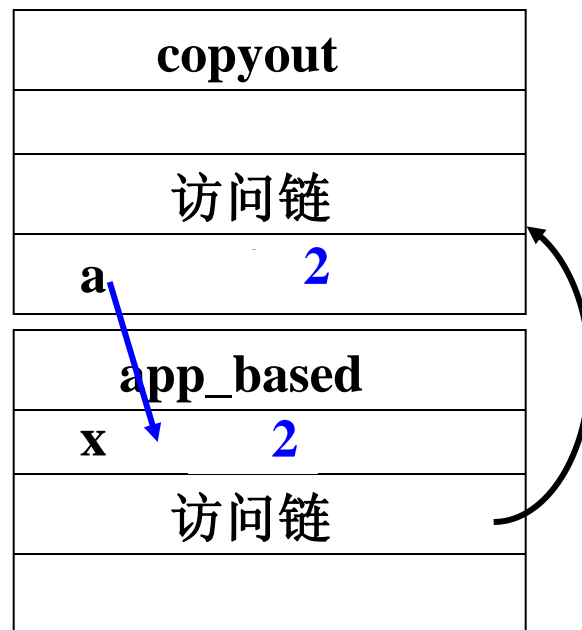
# 引用调用与复制恢复的区别

```
program copyout(input,output);  
  var a: integer;  
  procedure app_based(var x:integer);  
    begin x:=2; a:=0 end;  
begin a:=1; app_based(a); writeln(a) end.
```

## ■ 引用调用情况



## ■ 复制恢复情况



# 引用调用与复制恢复的区别

```
program cmp_example(input, output);  
var a: integer;  
procedure inc_1(x, y: integer);  
    begin x:=x+1; y:=y+1 end;  
begin  
a:=1;  
inc_1(a, a);  
end.
```

- 引用调用情况:  $a=3$
- 复制恢复:  $a=2$

## 四、传名调用

- ALGOL60所定义的一种特殊的传递方式
- 宏扩展：
  - ◆ 把过程当作宏，用被调用过程的过程体替换调用语句
  - ◆ 用实参的字面形式替换相应的形参
- 注意：
  - ◆ 被调用过程中的局部名字不能与调用过程中的名字重名
  - ◆ 在做宏扩展之前，对被调用过程中的每一个名字都系统地重新命名
  - ◆ 为了保持实参的完整性，可以用括号把实参括起来。



# 实现传名调用方式的基本方法

## ■ 展开结果

```
begin  
  a:=1; b:=2;  
  temp:=a;  
  a:=b;  
  b:=temp;  
  writeln('a=', a);  
  writeln('b=', b)  
end.
```

- 进入被调用过程之前不对实参求值
- 在过程体中，当用到相应的形参时，才求值。
- 实现
  - 结果程序中对每个实参都需要编制一个单独的子程序，称为参数子程序。
  - 每当过程体中用到相应的形参时，就调用这个程序。
  - 调用时，若实参不是变量，形参替换程序就计算实参，并返回此值所在的地址。过程体中每当引用形参时，就调用参数子程序，利用所送回的地址去引用该值。

# 例:

```
int i;  
int B[2];  
void P(int v)  
{  
    i=0; v=v+10; B[i]=10;  
    i=1; v=v+10; B[i]=10;  
}  
main()  
{  
    B[0]=10; B[1]=20;  
    i=0; P(B[i]);  
    cout<<B[0]<<'\ '<<B[1];  
}
```

- 传值调用
- 引用调用
- 复制恢复
- 传名

# 小结

## ■ 基本概念

- ◆ 静态文本、活动
- ◆ 活动的生存期、活动树、控制栈、活动记录
- ◆ 运行时内存的划分
- ◆ 声明的作用域、名字的结合

## ■ 存储分配策略

- ◆ 静态存储分配
- ◆ 栈式存储分配
  - 控制链、访问链
  - 调用序列、返回序列
- ◆ 堆式存储分配

# 小结（续）

## ■ 非局部名字的访问

- ◆ 静态作用域规则
- ◆ 非嵌套过程的作用域规则、存储分配策略
- ◆ 嵌套过程的作用域规则
  - 嵌套深度
  - 访问链的指向
  - 访问链的使用
  - 访问链的建立

## ■ 动态作用域

- ◆ 访问链的指向
- ◆ 深访问
- ◆ 浅访问

# 小结（续）

## ■ 参数传递

- ◆ 传值调用
- ◆ 引用调用
- ◆ 复制恢复
  - 引用调用与复制恢复的不同
- ◆ 传名调用

# 作业 (P. 251)

- 7. 1
- 7. 2
- 7. 3
- 7. 4
- 7. 6

# 第8章 中间代码生成



*LI Wensheng, SCST, BUPT*

知识点： 三地址代码  
语句的翻译  
布尔表达式的翻译  
回填技术

# 中间代码生成

## ■ 中间代码生成程序的任务

把经语法分析、语义分析后得到的源程序的中间表示翻译成中间代码表示

## ■ 采用中间代码作为过渡的优点

便于编译程序的建立和移植

便于进行与机器无关的代码优化工作

## ■ 缺点

增加了I/O操作、效率下降

## ■ 中间代码生成程序的位置





# 中间代码生成

- 8. 1 中间代码形式
- 8. 2 赋值语句的翻译
- 8. 3 布尔表达式的翻译
- 8. 4 控制语句的翻译
- 8. 5 标号和转移语句的翻译
- 8. 6 CASE语句的翻译
- 8. 7 过程调用语句的翻译
- 小 结

# 8.1 中间代码形式

## 一、图形表示

- ◆ 语法树
- ◆ dag图

## 二、三地址代码

- ◆ 三地址语句的形式
- ◆ 三地址语句的种类
- ◆ 三地址语句的实现

# 一、图形表示

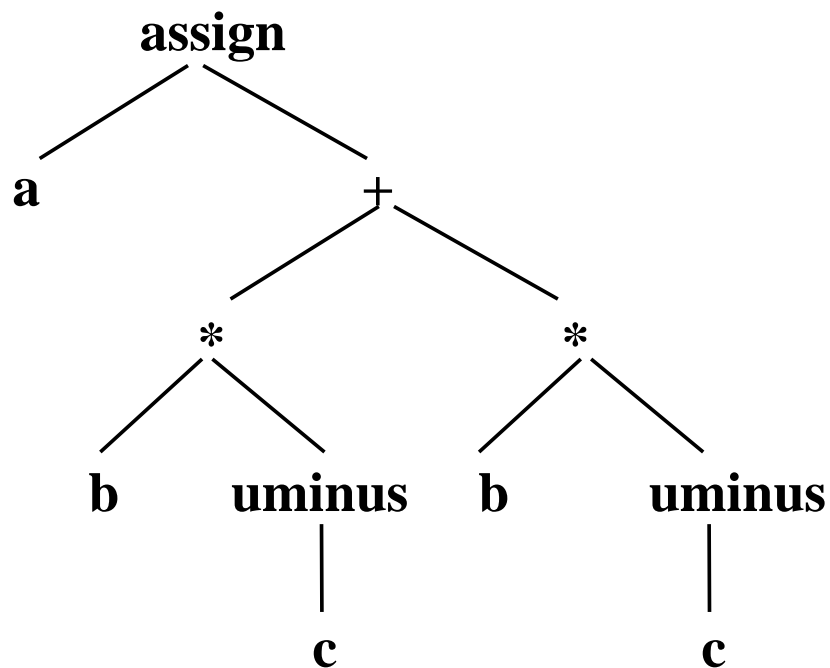
- 语法树
  - ◆ 描绘了源程序的天然层次结构
- dag图
  - ◆ 以更紧凑的方式给出了同样的信息
  - ◆ 因为公共子表达式被标识出来了

# 为赋值语句产生语法树的语法制导定义

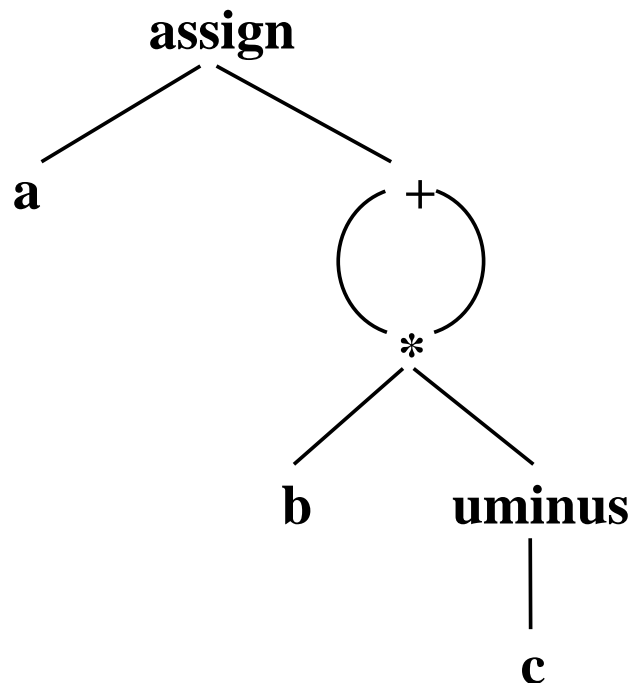
| 产生式                        | 语义规则                                                               |
|----------------------------|--------------------------------------------------------------------|
| $S \rightarrow id := E$    | $S.nptr = \text{manode}(':', \text{mkleaf}(id, id.entry), E.nptr)$ |
| $E \rightarrow E_1 + T$    | $E.nptr = \text{makenode}('+', E_1.nptr, T.nptr)$                  |
| $E \rightarrow T$          | $E.nptr = T.nptr$                                                  |
| $T \rightarrow T_1 * F$    | $T.nptr = \text{makenode}('*', T_1.nptr, F.nptr)$                  |
| $T \rightarrow F$          | $T.nptr = F.nptr$                                                  |
| $F \rightarrow (E)$        | $F.nptr = E.nptr$                                                  |
| $F \rightarrow id$         | $F.nptr = \text{makeleaf}(id, id.entry)$                           |
| $F \rightarrow \text{num}$ | $F.nptr = \text{makeleaf}(\text{num}, \text{num.val})$             |

# 赋值语句 $a := b * -c + b * -c$ 的图表示法

语法树表示:



dag图形表示:



## 二、三地址代码

### ■ 三地址代码

- ◆ 三地址语句组成的序列
- ◆ 类似于汇编语言的代码
- ◆ 语句可以有标号
- ◆ 有控制流语句

### ■ 三地址语句的形式: $x := y \text{ op } z$

- ◆  $x$  可以是名字、临时变量
- ◆  $op$  代表运算符号, 如定点、浮点、或逻辑运算符等
- ◆  $y$ 、 $z$  可以是名字、常数、或临时变量

# 三地址语句的种类

## ■ 赋值语句

**$x := y \text{ op } z$**

**$x := \text{op } y$**

**$x := y$**

## ■ 转移语句

**goto L**

**if x relop y goto L**

## ■ 过程调用语句

**param x**

**call p, n**

## ■ 数组赋值语句

**$x := y[i]$**

**$x[i] := y$**

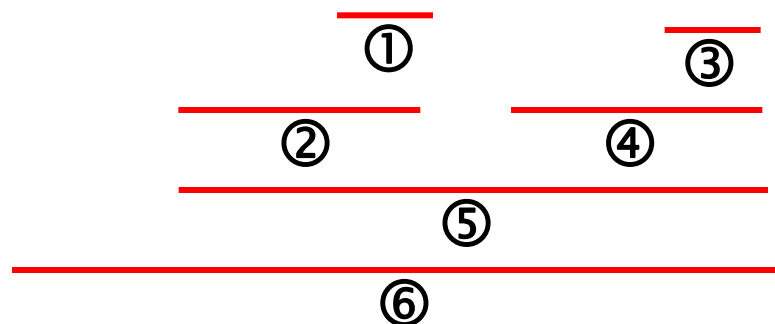
## ■ 指针赋值语句

**$x := \&y$**

**$x := *y$**

**$*x := y$**

# 赋值语句 $a := b * -c + b * -c$ 的三地址代码



## ■ 对应语法树的代码

$t_1 := -c$   
 $t_2 := b * t_1$   
 $t_3 := -c$   
 $t_4 := b * t_3$   
 $t_5 := t_2 + t_4$   
 $a := t_5$

## ■ 对应dag的代码

$t_1 := -c$   
 $t_2 := b * t_1$   
 $t_5 := t_2 + t_2$   
 $a := t_5$



# 三地址语句的实现——四元式

## ■ 四元式

(**op**, **arg1**, **arg2**, **result**)

(**op**, **arg1**, , **result**)

(**param**, **arg1**, , )

(**goto**, , , 语句标号)

## ■ 赋值语句**a:=b\*-c+b\*-c**的四元式表示

|      | <b>op</b>     | <b>arg1</b>          | <b>arg2</b>          | <b>result</b>        |
|------|---------------|----------------------|----------------------|----------------------|
| (14) | <b>uminus</b> | <b>c</b>             |                      | <b>t<sub>1</sub></b> |
| (15) | <b>*</b>      | <b>b</b>             | <b>t<sub>1</sub></b> | <b>t<sub>2</sub></b> |
| (16) | <b>uminus</b> | <b>c</b>             |                      | <b>t<sub>3</sub></b> |
| (17) | <b>*</b>      | <b>b</b>             | <b>t<sub>3</sub></b> | <b>t<sub>4</sub></b> |
| (18) | <b>+</b>      | <b>t<sub>2</sub></b> | <b>t<sub>4</sub></b> | <b>t<sub>5</sub></b> |
| (19) | <b>:=</b>     | <b>t<sub>5</sub></b> |                      | <b>a</b>             |

# 三地址语句的实现——三元式

- 三元式：（**op**, **arg1**, **arg2**）
  - ◆ 为避免把临时变量名也存入符号表，可不引入临时变量
  - ◆ 把由一个语句计算出来的中间结果直接提供给引用它的语句
  - ◆ 用计算中间结果的语句的指针代替存放中间结果的临时变量
- 赋值语句**a:=b\*-c+b\*-c**的三元式表示

|      | op     | arg1 | arg2 |
|------|--------|------|------|
| (14) | uminus | c    |      |
| (15) | *      | b    | (14) |
| (16) | uminus | c    |      |
| (17) | *      | b    | (16) |
| (18) | +      | (15) | (17) |
| (19) | :=     | a    | (18) |

# 语句 $x[i] := y$ 和 $x := y[i]$ 的三元式序列

## ■ 语句 $x[i] := y$

|     | op     | arg1 | arg2 |
|-----|--------|------|------|
| (0) | $[] =$ | $x$  | $i$  |
| (1) | assign | (0)  | $y$  |

## ■ 语句 $x := y[i]$

|     | op     | arg1 | arg2 |
|-----|--------|------|------|
| (0) | $= []$ | $y$  | $i$  |
| (1) | assign | $x$  | (0)  |

# 三地址语句的实现——间接三元式

## ■ 间接三元式

- ◆ 间接码表：为三元式序列建立的一个指针数组，其每个元素依次指向三元式序列中的一项

## ■ 赋值语句 $a:=b*-c+b*-c$ 的间接三元式表示

间接码表

|     | 语句   |
|-----|------|
| (0) | (14) |
| (1) | (15) |
| (2) | (16) |
| (3) | (17) |
| (4) | (18) |
| (5) | (19) |

三元式

|      | op     | arg1 | arg2 |
|------|--------|------|------|
| (14) | uminus | c    |      |
| (15) | *      | b    | (14) |
| (16) | uminus | c    |      |
| (17) | *      | b    | (16) |
| (18) | +      | (15) | (17) |
| (19) | assign | a    | (18) |

## 8.2 赋值语句的翻译

- 假定赋值语句出现的环境可用下面的文法描述：

$P \rightarrow M D; S$

$M \rightarrow \varepsilon$

$D \rightarrow D; D \mid D \rightarrow \text{id}: T \mid D \rightarrow \text{proc id}; N D; S$

$N \rightarrow \varepsilon$

$T \rightarrow \text{integer} \mid \text{real}$   
 $\quad \mid \text{array} [\text{num}] \text{ of } T_1$   
 $\quad \mid \uparrow T_1$   
 $\quad \mid \text{record } D \text{ end}$

$S \rightarrow \text{id} := E$

$E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid \text{id}$

# 一、仅涉及简单变量的赋值语句

## ■ 文法

**$S \rightarrow \text{id} := E$**

**$E \rightarrow E_1 + E_2$**

**$E \rightarrow E_1 * E_2$**

**$E \rightarrow -E_1$**

**$E \rightarrow (E_1)$**

**$E \rightarrow \text{id}$**

■ 函数 **newtemp**

■ 属性 **E.place**

■ 函数 **lookup(e)**

# 翻译方案1

**$S \rightarrow id := E$**  {  $p = \text{lookup}(id.name);$   
if ( $p \neq \text{nil}$ )  $\text{emit}(p := E.place);$   
else  $\text{error}();$  }

**$E \rightarrow E_1 + E_2$**  {  $E.place = \text{newtemp};$   
 $\text{emit}(E.place := E_1.place + E_2.place)$  }

**$E \rightarrow E_1 * E_2$**  {  $E.place = \text{newtemp};$   
 $\text{emit}(E.place := E_1.place * E_2.place)$  }

**$E \rightarrow -E_1$**  {  $E.place = \text{newtemp};$   
 $\text{emit}(E.place := \text{'uminus'} E_1.place)$  }

**$E \rightarrow (E_1)$**  {  $E.place = E_1.place$  }

**$E \rightarrow id$**  {  $p = \text{lookup}(id.name);$   
if ( $p \neq \text{nil}$ )  $E.place = p;$   
else  $\text{error}();$  }

# 翻译方案2

(考虑类型)

```
S→id:=E { p=lookup(id.name);  
    if (p!=nil) {  
        t=gettype(p);  
        if (t==E.type) {  
            emit(p ':=' E.place);  
            S.type=void;  
        };  
        else if (E.type 可转换为t) {  
            u=newtemp;  
            emit(u ':=' 类型转换符 E.place);  
            emit(p ':=' u);  
            S.type=void;  
        }  
        else S.type=type_error;  
    };  
else error(); }
```



```

E→E1+E2 { E.place=newtemp;
    if (E1.type==integer) && (E2.type==integer) {
        emit(E.place ':=' E1.place '+' E2.place);
        E.type=integer;
    };
    else if (E1.type==real) && (E2.type==real) {
        emit(E.place ':=' E1.place 'real+' E2.place);
        E.type=real;
    };
    else if (E1.type==integer) && (E2.type==real) {
        u=newtemp;
        emit(u ':=' 'inttoreal' E1.place);
        emit(E.place ':=' u 'real+' E2.place);
        E.type=real;
    };
    else if (E1.type==real) && (E2.type==integer) {
        u=newtemp;
        emit(u ':=' 'inttoreal' E2.pace);
        emit(E.place ':=' E1.place 'real+' u);
        E.type=real;
    };
    else E.type=type_error; }

```

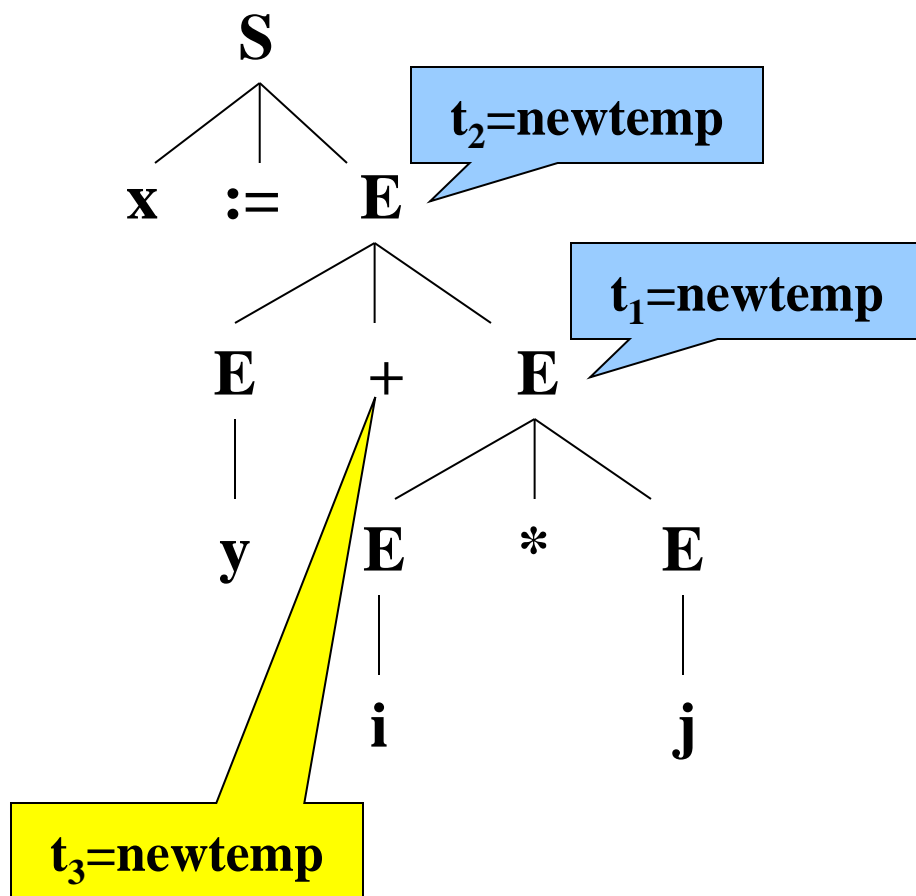
**$E \rightarrow -E_1$     {     $E.place = newtemp;$   
                   $emit(E.place \text{ '}' := \text{ 'uminus' } E_1.place);$   
                   $E.type = E_1.type$     }**

**$E \rightarrow (E_1)$     {     $E.place = E_1.place;$   
                   $E.type = E_1.type$     }**

**$E \rightarrow id$         {     $p = lookup(id.name);$   
                  if ( $p \neq nil$ ) {  
                       $E.place = p;$   
                       $E.type = gettype(p);$   
                  };  
                  else error();    }**

# 翻译赋值语句 $x := y + i * j$

- 假定 $x$ 和 $y$ 的类型为 $\text{real}$ ， $i$ 和 $j$ 的类型为 $\text{integer}$



- 三地址代码:

$t_1 := i \text{ int} * j$

$t_3 := \text{inttoreal } t_1$

$t_2 := y \text{ real} + t_3$

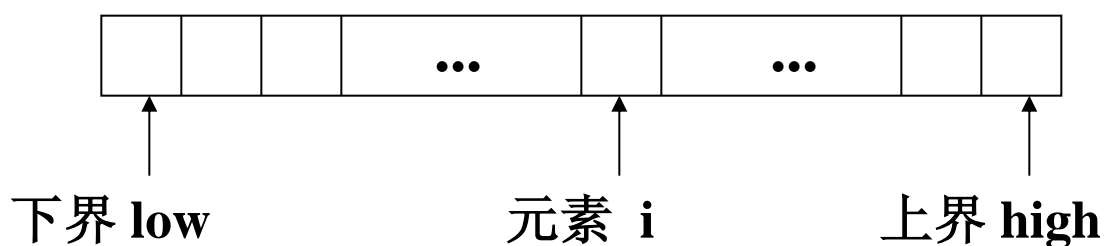
$x := t_2$

## 二、涉及数组元素的赋值语句

### 1. 计算数组元素的地址

- 数组元素存储在一个连续的存储块中，根据数组元素的下标可以快速地查找每个元素。

#### ■ 一维数组A



基址: base  
域宽: w  
元素个数:  $high - low + 1$

数组元素A[i]的位置:

$$\begin{aligned} & \text{base} + (i - \text{low}) \times w \\ &= i \times w + \text{base} - \text{low} \times w \end{aligned}$$

## ■ 二维数组A

|           |           |           |            |
|-----------|-----------|-----------|------------|
| $a_{11}$  | $a_{12}$  | $a_{1j}$  | $a_{1n2}$  |
| $a_{21}$  | $a_{22}$  | $a_{2j}$  | $a_{2n2}$  |
| ...       |           |           |            |
| $a_{i1}$  | $a_{i2}$  | $a_{ij}$  | $a_{in2}$  |
| ...       |           |           |            |
| $a_{n11}$ | $a_{n12}$ | $a_{n1j}$ | $a_{n1n2}$ |

存储方式:

按行存放

按列存放

每维的下界:  $low_1$ 、 $low_2$

每维的上界:  $high_1$ 、 $high_2$

每维的长度:  $n_1 = high_1 - low_1 + 1$

$n_2 = high_2 - low_2 + 1$

域宽:  $w$

基址:  $base$

数组元素 $A[i,j]$ 的位置:

$$\begin{aligned} & base + ((i - low_1) \times n_2 + (j - low_2)) \times w \\ & = (i \times n_2 + j) \times w + base - (low_1 \times n_2 + low_2) \times w \end{aligned}$$

## ■ K维数组A

每维的下界:  $\text{low}_1$ 、 $\text{low}_2$ 、 $\dots$ 、 $\text{low}_k$

每维的长度:  $n_1$ 、 $n_2$ 、 $\dots$ 、 $n_k$

存储方式: 按行存放

数组元素 $A[i_1, i_2, \dots, i_k]$ 的位置:

$$\begin{aligned} & ((\dots((i_1 \times n_2 + i_2) \times n_3 + i_3) \dots) \times n_k + i_k) \times w \\ & + \text{base} - ((\dots((\text{low}_1 \times n_2 + \text{low}_2) \times n_3 + \text{low}_3) \dots) \times n_k + \text{low}_k) \times w \end{aligned}$$

递归计算:

$$e_1 = i_1$$

$$e_2 = e_1 \times n_2 + i_2$$

$$e_3 = e_2 \times n_3 + i_3$$

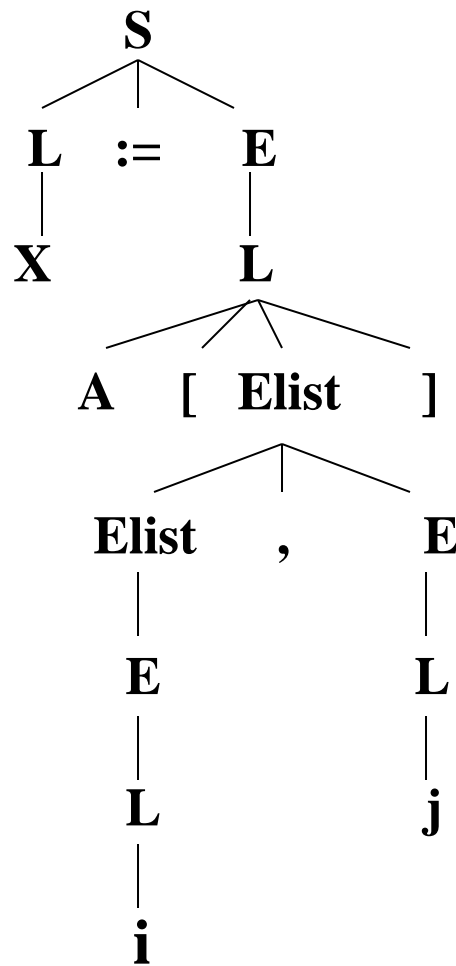
$$e_k = e_{k-1} \times n_k + i_k$$

## 2. 涉及数组元素的赋值语句的翻译 ——L属性定义

语句  $X := A[i, j]$  的分析树

### ■ 赋值语句的文法:

- (1)  $S \rightarrow L := E$
- (2)  $E \rightarrow E_1 + E_2$
- (3)  $E \rightarrow (E_1)$
- (4)  $E \rightarrow L$
- (5)  $L \rightarrow id$
- (6)  $L \rightarrow id [ Elist ]$
- (7)  $Elist \rightarrow E$
- (8)  $Elist \rightarrow Elist_1 , E$



# 属性及函数设计

## ■ L 综合属性L.place和L.offset

### ◆ 简单变量:

**L.offset=null**

**L.place=符号表入口指针**

### ◆ 数组元素:

**L.offset=计算公式第一项**

**L.place=计算公式第二项**

## ■ E 综合属性E.place, 保存E值的变量在符号表中的位置

## ■ Elist 继承属性array, 综合属性ndim, place

### ◆ Elist.array: 数组名在符号表中的位置

### ◆ Elist.ndim: 目前已经识别出的下标表达式的个数

### ◆ Elist.place: 保存递推公式中 $e_m$ 值的临时变量在符号表中的位置

## ■ 函数

### ◆ limit(array, j): 返回array指向的数组第 j 维的长度

### ◆ invariant(array): 返回array指向的数组的地址计算公式中的不变项



# L属性定义翻译方案

**$S \rightarrow L := E$**  { if (L.offset == null) /\* L是简单变量 \*/  
                  emit(L.place ':= ' E.place );  
                  else emit(L.place '[' L.offset ']' ':= ' E.place) }

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

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

**$E \rightarrow L$**  { if (L.offset == null) /\* L是简单变量 \*/  
                  E.place = L.place;  
                  else {  
                      E.place = newtemp;  
                      emit(E.place ':= ' L.place '[' L.offset ']' );  
                  }  
                  }

**L**→**id** { L.place=id.place; L.offset=null }

**L**→**id**[ { Elist.array=id.place }  
**Elist**]

{ L.place=newtemp;  
emit( L.place ':=' Elist.array '-' invariant(Elist.array));  
L.offset=newtemp;  
emit(L.offset ':=' w '×' Elist.place) }

**Elist**→**E** { Elist.place=E.place;  
Elist.ndim=1 }

$$e_1 = i_1$$

$$\begin{aligned} e_2 &= e_1 \times n_2 + i_2 \\ e_3 &= e_2 \times n_3 + i_3 \\ e_k &= e_{k-1} \times n_k + i_k \end{aligned}$$

**Elist**→ { Elist<sub>1</sub>.array=Elist.array }

**Elist**<sub>1</sub>,**E** { t=newtemp; m=Elist<sub>1</sub>.ndim+1;  
emit(t ':=' Elist<sub>1</sub>.place '×' limit(Elist<sub>1</sub>.array,m));  
emit(t ':=' t '+' E.place);  
Elist.place=t;  
Elist.ndim=m }

# 举例 翻译语句 $X := A[y, z]$

已知：

设A为一个 $10 \times 20$ 的数组，即  $n_1 = 10$ ， $n_2 = 20$ ；

并设域宽  $w = 4$ ；

数组的第一个元素为A[1,1]，

则有  $low_1 = 1$ ， $low_2 = 1$

所以：

$$(low_1 \times n_2 + low_2) \times w = (1 \times 20 + 1) \times 4 = 84$$

问题：

将赋值语句  $x := A[y, z]$  翻译为三地址代码。

# $x := A[y, z]$

## ■ 产生三地址代码:

$t_1 := y \times 20$

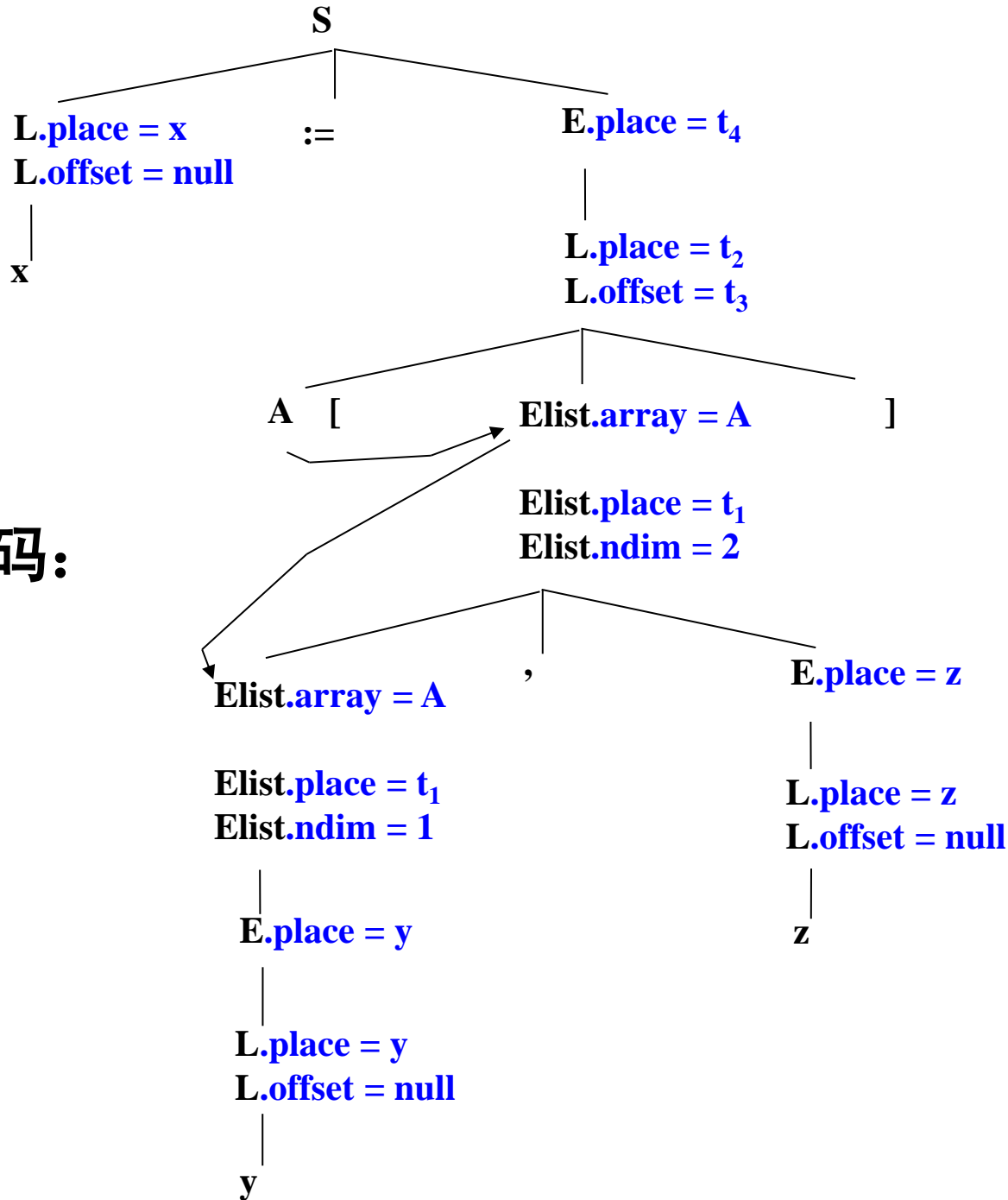
$t_1 := t_1 + z$

$t_2 := A - 84$

$t_3 := 4 \times t_1$

$t_4 := t_2[t_3]$

$x := t_4$



### 3. 涉及数组元素的赋值语句 ——S属性定义

$S \rightarrow L := E$

$E \rightarrow E_1 + E_2$

$E \rightarrow (E_1)$

$E \rightarrow L$

$L \rightarrow \text{id} \mid \text{id} [ \text{Elist} ]$

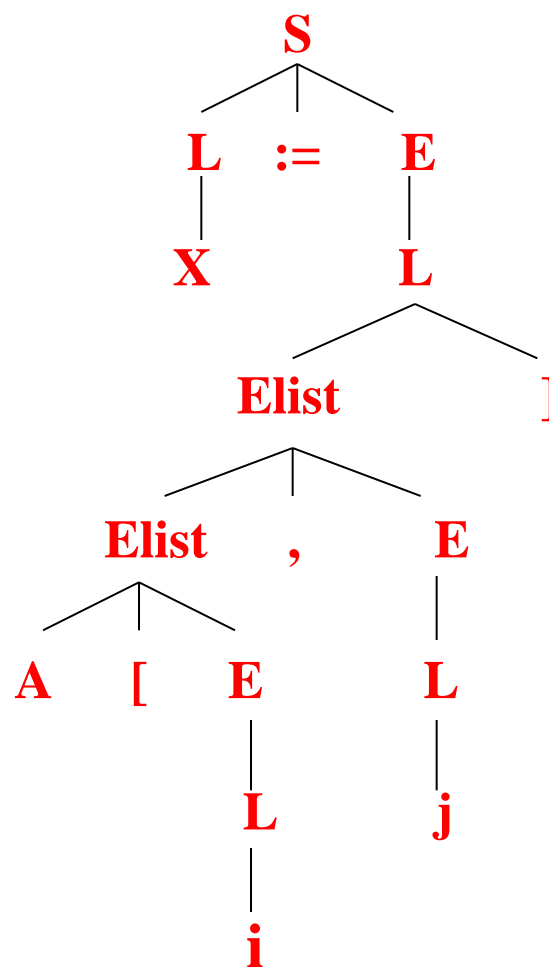
$\text{Elist} \rightarrow \text{Elist}_1, E \mid E$

改写文法:

$L \rightarrow \text{id} \mid \text{Elist} ]$

$\text{Elist} \rightarrow \text{Elist}_1, E \mid \text{id}[E$

语句  $X := A [ i, j ]$  的分析树



# 属性及函数设计

## ■ L 综合属性L.place和L.offset

### ◆ 简单变量:

**L.offset=null**

**L.place=符号表入口指针**

### ◆ 数组元素:

**L.offset=计算公式第一项**

**L.place=计算公式第二项**

## ■ E 综合属性E.place, 保存E值的变量在符号表中的位置

## ■ Elist 综合属性Elist.array, ndim, place

### ◆ Elist.array: 数组名在符号表中的位置

### ◆ Elist.ndim: 目前已经识别出的下标表达式的个数

### ◆ Elist.place: 保存递推公式中 $e_m$ 值的临时变量在符号表中的位置

## ■ 函数

### ◆ limit(array, j): 返回array指向的数组第j维的长度

### ◆ invariant(array): 返回array指向的数组的地址计算公式中的不变项

# S属性定义翻译方案

**$S \rightarrow L := E$**  { if (L.offset == null) /\* L是简单变量 \*/  
                  emit(L.place ':= ' E.place );  
          else emit(L.place '[' L.offset ']' ':= ' E.place); }

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

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

**$E \rightarrow L$**  { if (L.offset == null) /\* L是简单变量 \*/  
                  E.place = L.place;  
          else { E.place = newtemp;  
                  emit(E.place ':= ' L.place '[' L.offset ']' ); }  
          }

**L**→**id** { L.place=id.place; L.offset=null }

**Elist**→**id**[**E** { Elist.place=E.place;  
Elist.ndim=1;  
Elist.array=id.place }

$$e_1 = i_1$$

**Elist**→**Elist**<sub>1</sub>,**E** { t=newtemp;  
m=Elist<sub>1</sub>.ndim+1;  
emit(t ':=' Elist<sub>1</sub>.place '×' limit(Elist<sub>1</sub>.array,m));  
emit(t ':=' t '+' E.place);  
Elist.array=Elist<sub>1</sub>.array;  
Elist.place=t;  
Elist.ndim=m }

$$e_2 = e_1 \times n_2 + i_2$$

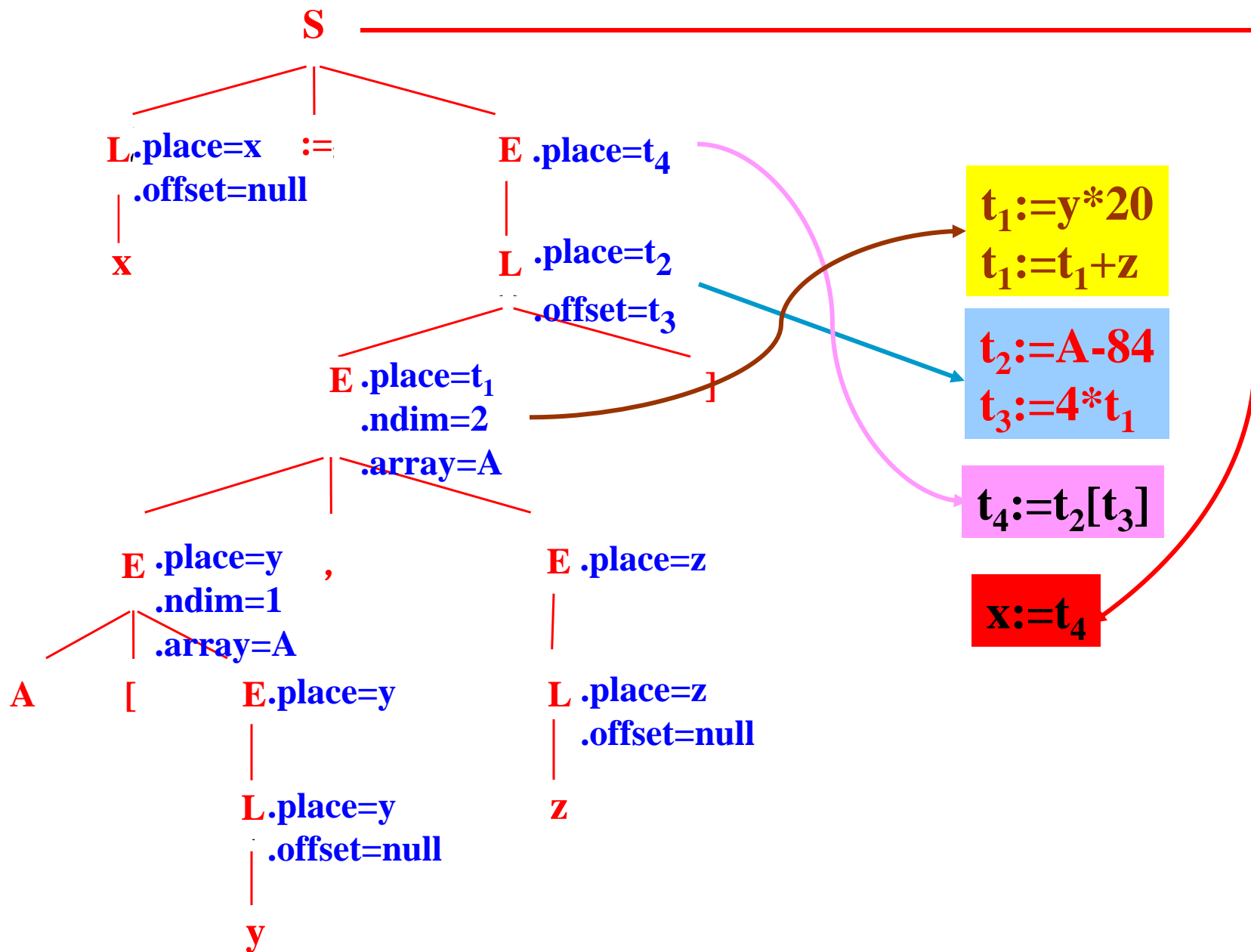
$$e_3 = e_2 \times n_3 + i_3$$

$$e_k = e_{k-1} \times n_k + i_k$$

**L**→**Elist**] { L.place=newtemp;  
emit( L.place ':=' Elist.array '-' invariant(Elist.array));  
L.offset=newtemp;  
emit(L.offset ':=' w '×' Elist.place) }



# 赋值语句 $x := A[y, z]$ 的分析树



# 三、记录中域的访问

## ■ 声明:

```
p: record
  info: integer;
  x: real
end;
```

name      type      address

|     |           |    |
|-----|-----------|----|
| ... |           |    |
| p   | record(t) | 48 |
| ... |           |    |

|      |      |    |
|------|------|----|
| nil  |      | 12 |
| info | int  | 0  |
| x    | real | 4  |

## ■ 引用

p.info+1

## ■ 编译器的动作

lookup(p)

Gettype

根据t, 找到记录的符号表

根据info在表中找

## 8.3 布尔表达式的翻译

- 布尔表达式的作用
  - ◆ 计算逻辑值
  - ◆ 用作控制语句中的条件表达式
- 产生布尔表达式的文法

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

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

$E \rightarrow \text{not } E$

$E \rightarrow (E)$

$E \rightarrow \text{id relop id}$

$E \rightarrow \text{true}$

$E \rightarrow \text{false}$

# 一、翻译布尔表达式的方法

## ■ 布尔表达式的值的表示方法

### ◆ 数值表示法:

**1 — true      0 — false**

**非0 — true      0 — false**

### ◆ 控制流表示法:

利用控制流到达程序中的位置来表示 **true** 或 **false**

## ■ 布尔表达式的翻译方法

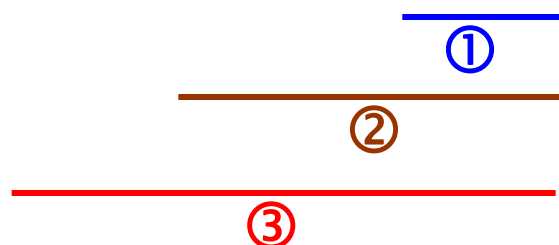
### ◆ 数值方法

### ◆ 控制流方法

## 二、数值表示法

- 布尔表达式的求值类似于算术表达式的求值

- 例如:  $a \text{ or } b \text{ and not } c$



- 三地址代码

$t_1 := \text{not } c$

$t_2 := b \text{ and } t_1$

$t_3 := a \text{ or } t_2$

- 关系表达式  $a < b$

等价于:

if  $a < b$

then 1

else 0

- $a < b$  的三地址代码:

100: if  $a < b$  goto 103

101:  $t := 0$

102: goto 104

103:  $t := 1$

104:

# 翻译方案

## ■ 属性、函数及变量说明

属性**E.place**: 存放布尔表达式E的值的临时变量  
(临时变量在符号表中的入口位置)

函数**emit**: 产生并输出一条三地址语句

变量**nextstat**: 输出序列中下一条三地址语句的位置  
**emit**之后, **nextstat**自动加1。

## ■ 翻译方案

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

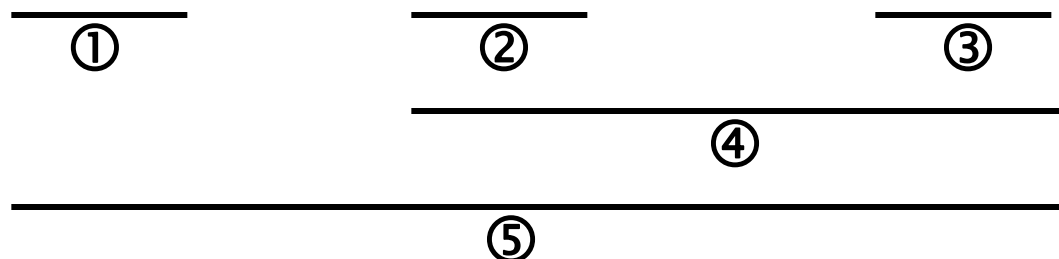
$E \rightarrow (E_1) \quad \{ E.place = E_1.place \}$

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

$E \rightarrow \text{true} \quad \{ E.place = \text{newtemp};$   
                     $\text{emit}(E.place \text{ ':='} '1') \}$

$E \rightarrow \text{false} \quad \{ E.place = \text{newtemp};$   
                     $\text{emit}(E.place \text{ ':='} '0') \}$

举例:  $a < b$  or  $c < d$  and  $e < f$



100: if  $a < b$  goto 103

101:  $t_1 := 0$

102: goto 104

103:  $t_1 := 1$

104: if  $c < d$  goto 107

105:  $t_2 := 0$

106: goto 108

107:  $t_2 := 1$

108: if  $e < f$  goto 111

109:  $t_3 := 0$

110: goto 112

111:  $t_3 := 1$

112:  $t_4 := t_2$  and  $t_3$

113:  $t_5 := t_1$  or  $t_4$

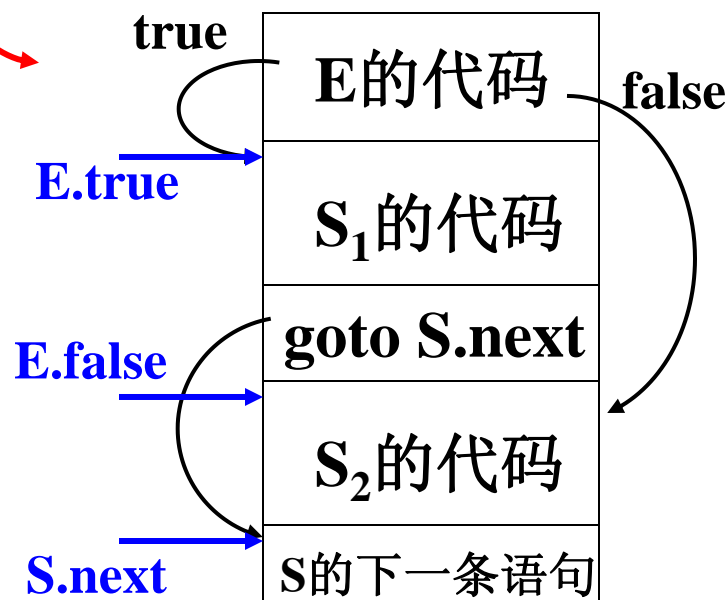
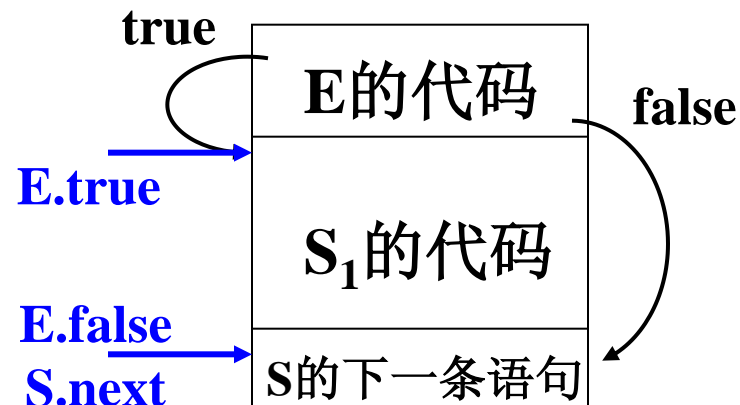
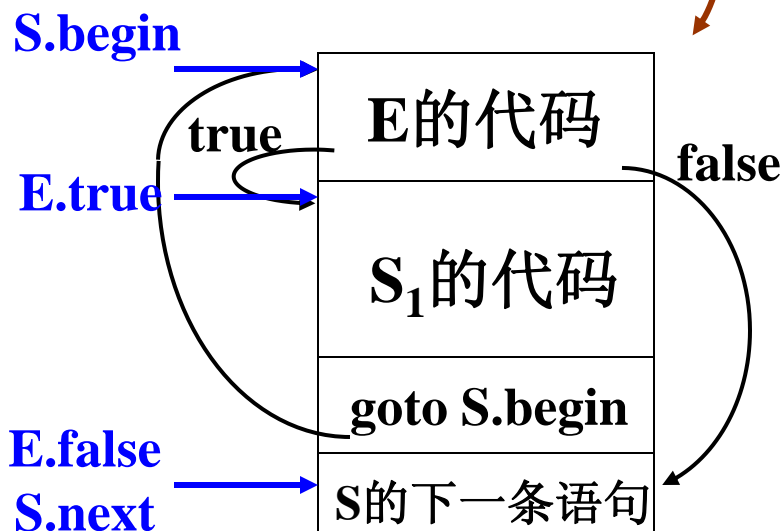


# 三、控制流表示法

## ■ 控制语句

$S \rightarrow$  if E then  $S_1$   
| if E then  $S_1$  else  $S_2$   
| while E do  $S_1$

## ■ 代码结构



# 属性说明

## ■ 继承属性：三地址语句标号

**E.true:** E值为真时应执行的第一条语句的标号

**E.false:** E值为假时应执行的第一条语句的标号

**S.next:** 紧跟在语句S之后的三地址语句的标号

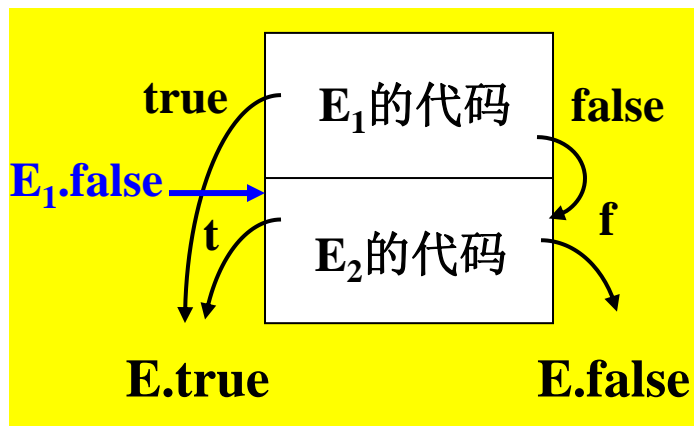
**S.begin:** 语句S的第一条三地址语句的标号

# 布尔表达式的控制流翻译

- 布尔表达式被翻译为一系列**条件转移**和**无条件转移**三地址语句
- 这些语句转移到的位置是**E.true**、**E.false**之一
- 例如 **a < b** 翻译为：  
    **if a < b goto E.true**  
    **goto E.false**
- 属性说明
  - ◆ 继承属性
    - E.true**: E为真时转移到的三地址语句的标号
    - E.false**: E为假时转移到的三地址语句的标号

# 布尔表达式的代码结构

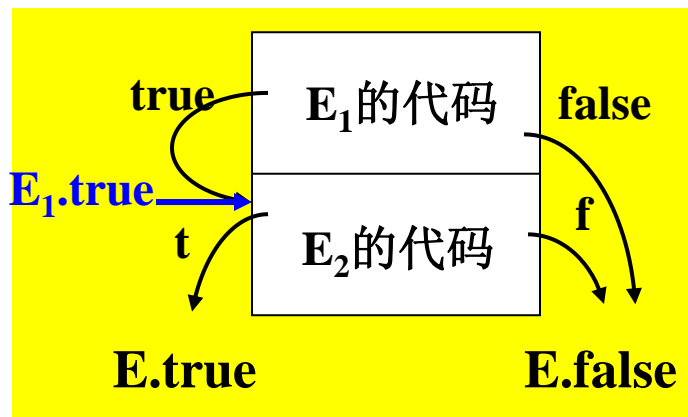
$E \rightarrow E_1 \text{ or } E_2$



$E \rightarrow \text{not } E_1$



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

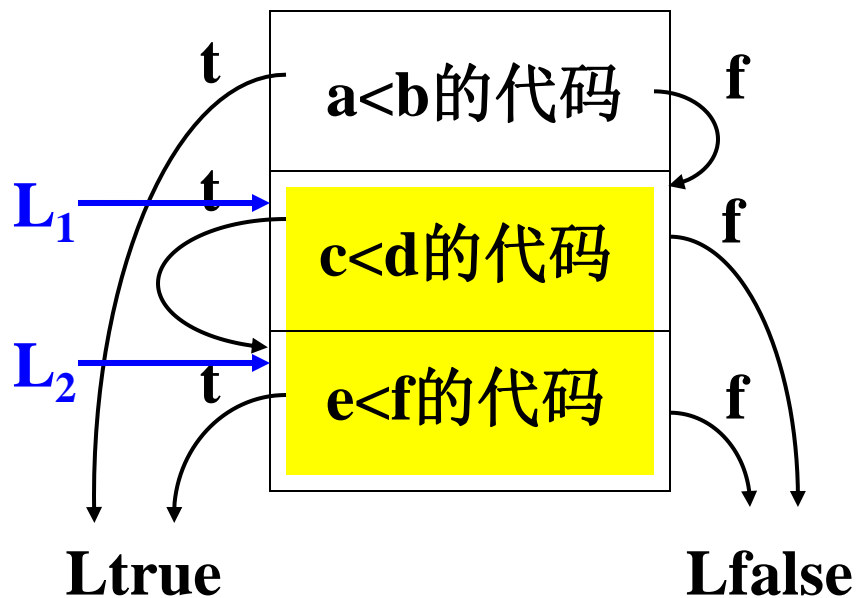


$E \rightarrow \text{id}_1 \text{ relop id}_2$

'if' id<sub>1</sub>.place relop.op id<sub>2</sub>.place 'goto' E.true  
'goto' E.false

# 例：用控制流方法翻译布尔表达式

## $a < b$ or $c < d$ and $e < f$



```
if a<b goto Ltrue
goto L1
L1: if c<d goto L2
      goto Lfalse
L2: if e<f goto Ltrue
      goto Lfalse
```

# 布尔表达式的翻译

## ■ 两遍扫描的翻译技术

1. 生成分析树
2. 为分析树加注释——翻译

## ■ 一遍扫描的分析和翻译技术

问题：当生成某些转移指令时，目标地址可能还不知道

## ■ 解决办法

回填技术（backpatching）

- ◆ 先产生没有填写目标标号的转移指令；
- ◆ 建立一个链表，把转向这个目标的所有转移指令的标号填入该链表；
- ◆ 目标地址确定后，再把该地址填入该链表中记录的所有转移指令中。

# 利用回填技术翻译布尔表达式

## ■ 说明

- ◆ 为明确起见，生成的中间代码用四元式表示
- ◆ 四元式存放在数组中
- ◆ 用数组下标表示三地址语句的标号

## ■ 综合属性

- ◆ **E.truelist**: 记录转移到E的真出口的指令链表的指针
- ◆ **E.falselist**: 记录转移到E的假出口的指令链表的指针
- ◆ **M.quad**: M所标识的三地址语句的地址

## ■ 变量

- ◆ **nextquad**: 下一个可用的四元式地址  
(产生的下一条三地址语句在数组中的位置)

# 函数说明

## ■ **makelist(i):**

- ◆ 建立新链表，其中只包括四元式指令在数组中的下标*i*；
- ◆ 返回所建链表的指针。

## ■ **merge( $p_1, p_2$ ):**

- ◆ 合并由指针 $p_1$ 和 $p_2$ 所指向的两个链表
- ◆ 返回结果链表的指针。

## ■ **backpatch( $p, i$ ):**

- ◆ 用目标地址*i*回填  $p$ 所指链表中记录的每一条转移指令。

## ■ **emit(S)**

- ◆ 产生一条三地址语句*S*，并写入输出数组中
- ◆ 该函数执行完后，变量 **nextquad** 加 1



例:

**.truelist={100, 104}**

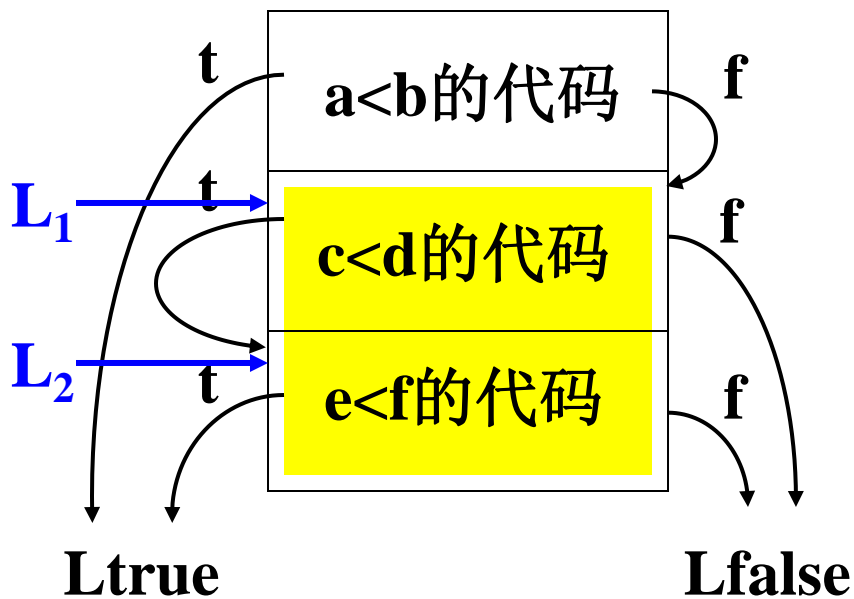
**.falselist={103, 105}**

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

①      ②      ③

④

⑤



```
100: if a<b goto —
101: goto 102
102: if c<d goto 104
103: goto —
104: if c<d goto —
105: goto —
```

# 翻译方案

## ■ 布尔表达式的文法

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

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

$E \rightarrow \text{not } E_1$

$E \rightarrow (E_1)$

$E \rightarrow \text{id}_1 \text{ relop id}_2$

$E \rightarrow \text{true}$

$E \rightarrow \text{false}$

$\mathbf{M} \rightarrow \varepsilon$

## ■ 标记非终结符号 **M**

- 标识布尔表达式  $E_2$  的开始位置
- 用属性 **M.quad** 记录其所标识的布尔表达式的第一条三地址语句的地址
- 相应于产生式  $\mathbf{M} \rightarrow \varepsilon$  的动作:  
**M.quad = nextquad**

$E \rightarrow E_1 \text{ or } ME_2$  { backpatch( $E_1$ .falselist, M.quad);  
 $E$ .truelist= merge( $E_1$ .truelist,  $E_2$ .truelist);  
 $E$ .falselist= $E_2$ .falselist }

$E \rightarrow E_1 \text{ and } ME_2$  { backpatch( $E_1$ .truelist, M.quad);  
 $E$ .truelist= $E_2$ .truelist;  
 $E$ .falselist= merge( $E_1$ .falselist,  $E_2$ .falselist) }

$E \rightarrow \text{not } E_1$  {  $E$ .truelist= $E_1$ .falselist;  $E$ .falselist= $E_1$ .truelist }

$E \rightarrow (E_1)$  {  $E$ .truelist= $E_1$ .truelist;  $E$ .falselist= $E_1$ .falselist }

$E \rightarrow id_1 \text{ relop } id_2$  {  $E$ .truelist=makelist(nextquad);  
 $E$ .falselist=makelist(nextquad+1);  
 emit('if'  $id_1$ .place relop.op  $id_2$ .place 'goto —');  
 emit('goto —') }

$E \rightarrow \text{true}$  {  $E$ .truelist=makelist(nextquad); emit('goto —') }

$E \rightarrow \text{false}$  {  $E$ .falselist=makelist(nextquad); emit('goto —') }

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

# 利用翻译方案翻译布尔表达式

**$a < b$  or  $c < d$  and  $e < f$**

假定变量**nextquad**的初值为**100**

100: if  $a < b$  goto —

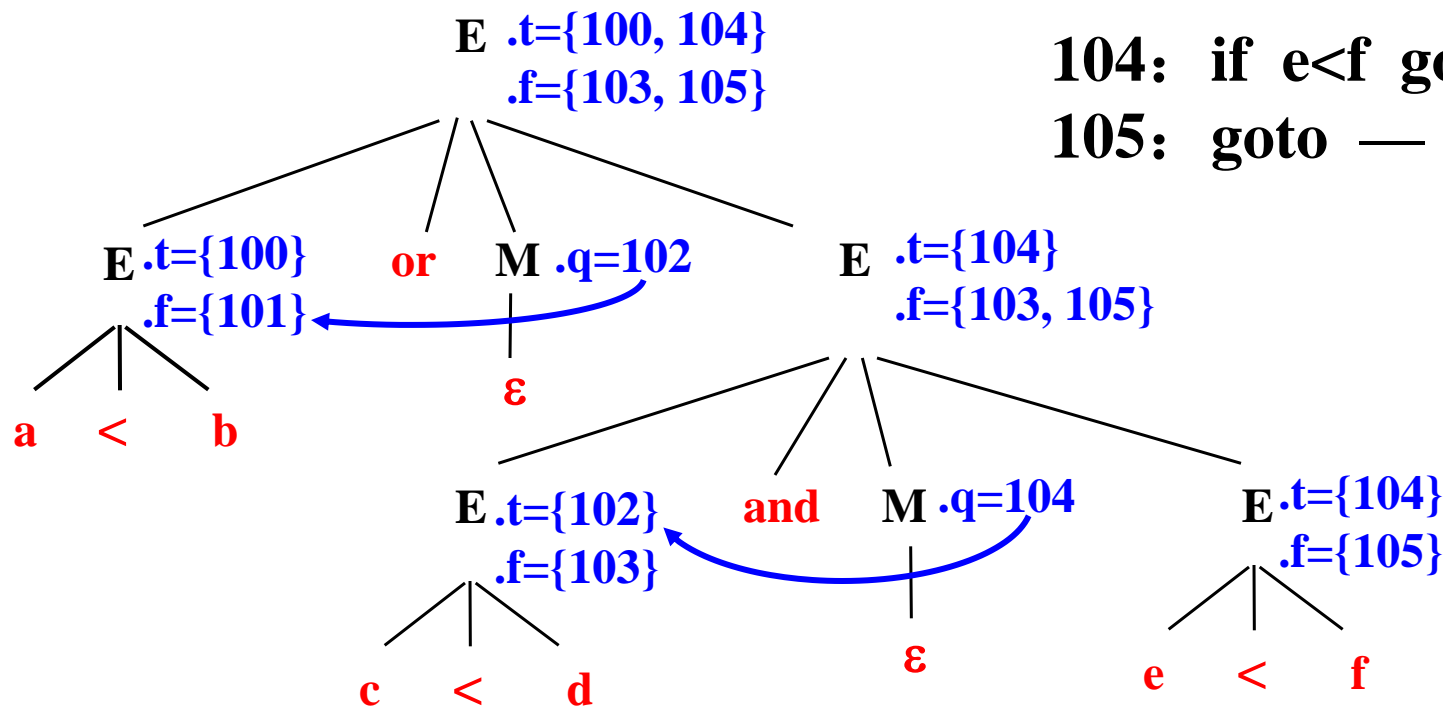
101: goto **102**

102: if  $c < d$  goto **104**

103: goto —

104: if  $e < f$  goto —

105: goto —



## 8.4 控制语句的翻译

### ■ 文法

$S \rightarrow \text{if } E \text{ then } M S_1$

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

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

$S \rightarrow \text{begin } L \text{ end}$

$S \rightarrow A$

$L \rightarrow L_1; M S$

$L \rightarrow S$

$M \rightarrow \epsilon$

$N \rightarrow \epsilon$

属性:

**E.truelist**

**E.falselist**

**M.quad**

**N.nextlist**

**S.nextlist**

变量: **nextquad**

函数:

**makelist**

**backpatch**

**merge**

**emit**

转移到下一条语句  
的指令链表的指针

▲ 记录变量 **nextquad** 的当前, 以便回填转移到此的指令

◆ 产生一条不完整的goto指令, 并记录下它的位置

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

$S \rightarrow \text{if } E \text{ then } M_1 \ S_1 \ N \ \text{else } M_2 \ S_2$   
 $\qquad\qquad\qquad \{ \text{backpatch}(E.\text{truelist}, M_1.\text{quad});$   
 $\qquad\qquad\qquad \text{backpatch}(E.\text{falselist}, M_2.\text{quad});$   
 $\qquad\qquad\qquad S.\text{nextlist} = \text{merge}(S_1.\text{nextlist}, N.\text{nextlist}, S_2.\text{nextlist}) \}$

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

$N \rightarrow \epsilon \ \{ \ N.\text{nextlist} = \text{makelist}(\text{nextquad}); \ \text{emit}(\text{'goto' } \text{---}) \}$

$S \rightarrow \text{while } M_1 \ E \ \text{do } M_2 \ S_1 \{ \text{backpatch}(S_1.\text{nextlist}, M_1.\text{quad});$   
 $\qquad\qquad\qquad \text{backpatch}(E.\text{truelist}, M_2.\text{quad});$   
 $\qquad\qquad\qquad S.\text{nextlist} = E.\text{falselist};$   
 $\qquad\qquad\qquad \text{emit}(\text{'goto' } M_1.\text{quad}) \}$

$S \rightarrow \text{begin } L \ \text{end} \ \{ \ S.\text{nextlist} = L.\text{nextlist} \}$

$S \rightarrow A \ \{ \ S.\text{nextlist} = \text{makelist}() \}$

$L \rightarrow L_1; \ M \ S \ \{ \text{backpatch}(L_1.\text{nextlist}, M.\text{quad}); \ L.\text{nextlist} = S.\text{nextlist} \}$

$L \rightarrow S \ \{ \ L.\text{nextlist} = S.\text{nextlist} \}$

⑧

④

⑦



## A<sub>1</sub>的代码

116

117

127

127

129

~~.t={127}~~  
~~.f={128}~~

```

127:  if a<b goto 129
128:  goto —
129:  A3的代码
138:
139:  goto 127

```

## 8.5 标号和转移语句的翻译

### ■ 标号的出现有两种形式

- ◆ **L: S**      —— 定义性出现
- ◆ **goto L**    —— 引用性出现

### ■ 两种情况

– 先定义、后引用



– 先引用、后定义

```
...  
L: S  
...  
goto L
```

```
...  
goto L  
...  
L: S
```



# 先定义后引用的情况

L:  S  
...  
goto L 

1. 查找当前符号表
2. 找到同名标识符，出错处理
3. 否则，将该标号L插入符号表

S的第一条三地址语句的位置

| name | type  | exist | address    |
|------|-------|-------|------------|
| L    | label | t     | nextquad的值 |
|      |       |       |            |

1. 根据标号L查找当前符号表
2. 取出L所标识的三地址语句的位置 A
3. 生成指令 (goto, —, —, A)

# 先引用后定义的情况

...

goto L

...

goto L

...

L: S

1. 根据标号L查找当前符号表
2. 若未找到，则：  
将标号L插入符号表  
生成目标地址未定义的goto语句  
将该语句的地址p填入L的表项中
3. 若找到，则：  
生成目标地址未定义的goto语句  
将该语句的地址q插在链首

| name | type  | exist | address |
|------|-------|-------|---------|
| L    | label | t     | A       |

1. 根据标号L查找当前符号表
2. 将“未定义”改为“已定义”
3. 用S的第一条三地址语句的地址A回填链表中的goto语句

q: (goto, —, —, A)

p: (goto, —, —, A)

# 两种情况统一考虑

## ■ goto L

### ◆ 根据名字L查找符号表

### ◆ 若未找到L，则

- 将L插入符号表中，置类型为“label”，置存在标志为“f”
- 为该语句生成无目标地址的三地址语句(**goto**, —, —, —)
- 将该三地址语句的地址q记入符号表中L的地址域中

### ◆ 找到L，则

- 如果L的类型不是label，则出错处理
- 若L的类型为label且“已定义”，则  
以L标识的三地址语句的地址A作为目标地址生成转移指令  
(**goto**, —, —, A)
- 若L的类型为label且“未定义”，则说明  
前面已出现过“**goto L**”，  
为该语句生成无目标地址的三地址语句(**goto**, —, —, —)，  
并将该三地址语句插入回填链的链首

## ■ L: S

### ◆ 根据名字L查找符号表

### ◆ 找到L，则

- L的类型不是label，出错处理；
- L的类型为label且“已定义”，出错处理；
- L的类型为label，但“未定义”，则：

置定义标志为“t”

用变量nextquad的值回填“地址域”中记录的链表中的三地址语句

### ◆ 未找到L，则

- 将L插入符号表中，置类型为“label”，置存在标志为“t”
- 将S的三地址代码的始址(变量nextquad的值)填入“地址域”

## ■ 请思考：

如何实现“goto语句只允许将控制从结构内转移出，而不允许转移入”？

## 8.6 CASE语句的翻译

### ■ CASE语句的语法结构

```
switch E
begin
  case  $V_1$ :  $S_1$ 
  case  $V_2$ :  $S_2$ 
  ...
  case  $V_{n-1}$ :  $S_{n-1}$ 
  default:  $S_n$ 
end
```

### ■ 生成的代码的功能

1. 对开关表达式求值;
2. 在 $V_1$ 、...、 $V_n$ 中找与E值匹配的值, 若找不到, 则让“缺省值”与之匹配;
3. 执行与 $V_i$ 相联系的语句 $S_i$ 。

# CASE语句的代码结构

对E求值的代码

把求值结果置于t中的代码

if  $t \neq V_1$  goto  $L_2$

$S_1$ 的代码

goto next

$L_2$ : if  $t \neq V_2$  goto  $L_3$

$S_2$ 的代码

goto next

$L_3$ : ...

$L_{n-1}$ : if  $t \neq V_{n-1}$  goto  $L_n$

$S_{n-1}$ 的代码

goto next

$L_n$ :  $S_n$ 的代码

next:

控制结构复杂  
goto语句生成时不完整

对E求值的代码

把求值结果置于t中的代码

goto test

$L_1$ :  $S_1$ 的代码

goto next

$L_2$ :  $S_2$ 的代码

goto next

$L_3$ : ...

$L_{n-1}$ :  $S_{n-1}$ 的代码

goto next

$L_n$ :  $S_n$ 的代码

goto next

test: if  $t = V_1$  goto  $L_1$

if  $t = V_2$  goto  $L_2$

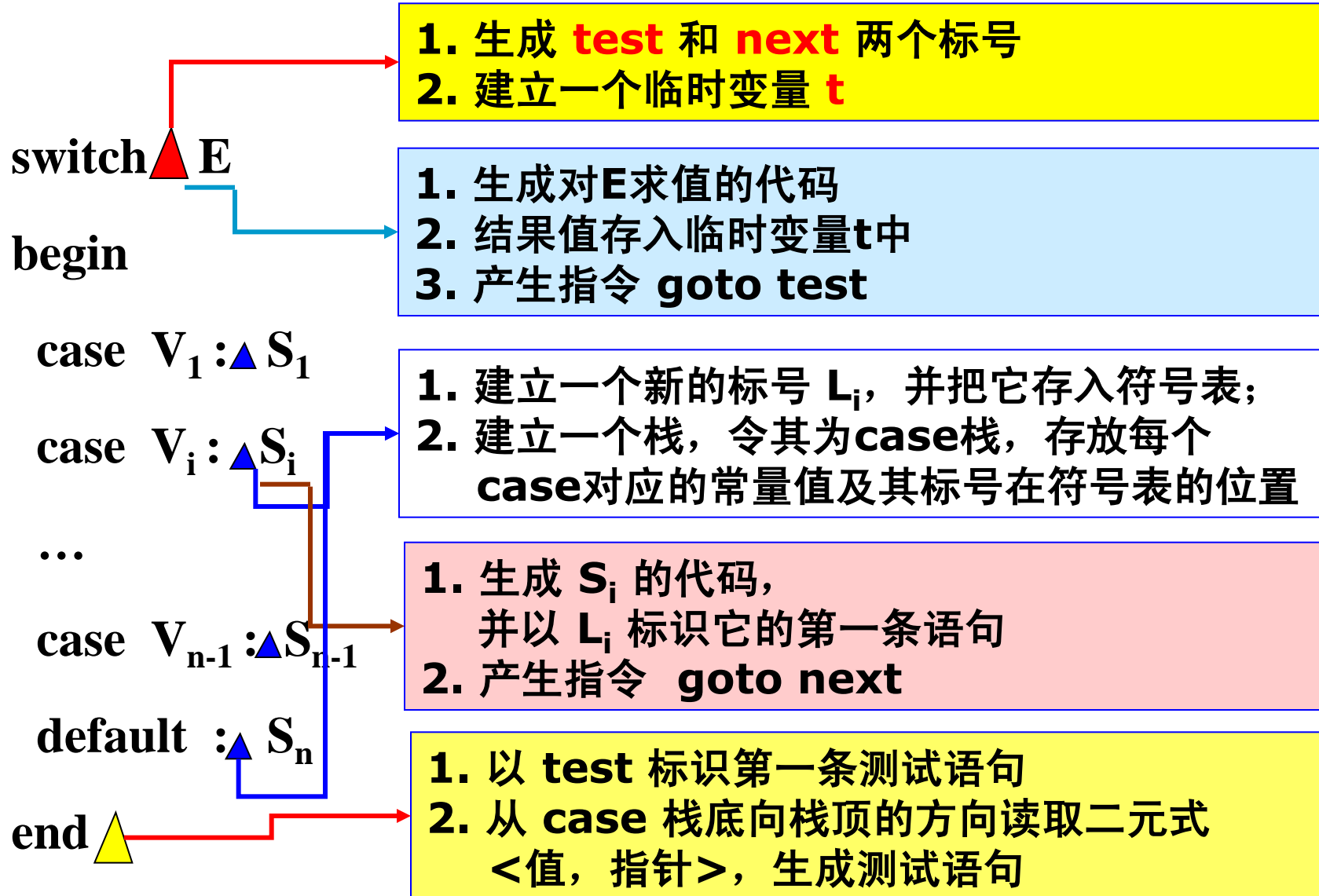
...

if  $t = V_{n-1}$  goto  $L_{n-1}$

goto  $L_n$

next:

# CASE语句的翻译



# case栈

底

符号表

| 值         | 指针        |
|-----------|-----------|
| $V_1$     | $L_1$     |
| $V_2$     | $L_2$     |
| ...       | ...       |
| $V_{n-1}$ | $L_{n-1}$ |
| $t$       | $L_n$     |
|           |           |

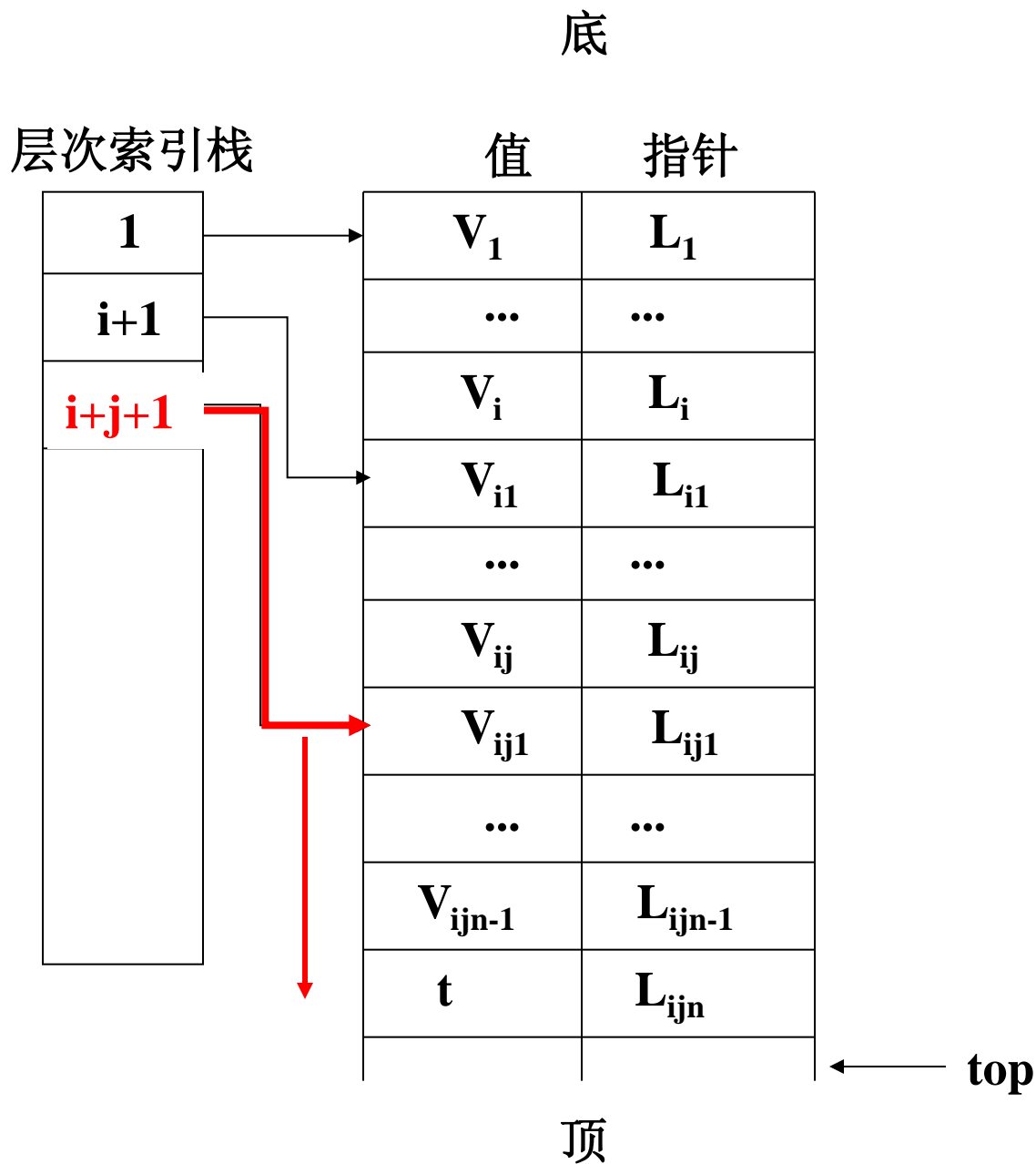
| name      | type  | address  |
|-----------|-------|----------|
| $L_1$     | label | nextstat |
| $L_2$     | label | nextstat |
| ...       |       |          |
| $L_{n-1}$ | label | nextstat |
| $L_n$     | label | nextstat |

**test: if  $t=V_1$  goto  $L_1$**   
**if  $t=V_2$  goto  $L_2$**   
**...**  
**if  $t=V_{n-1}$  goto  $L_{n-1}$**   
**goto  $L_n$**

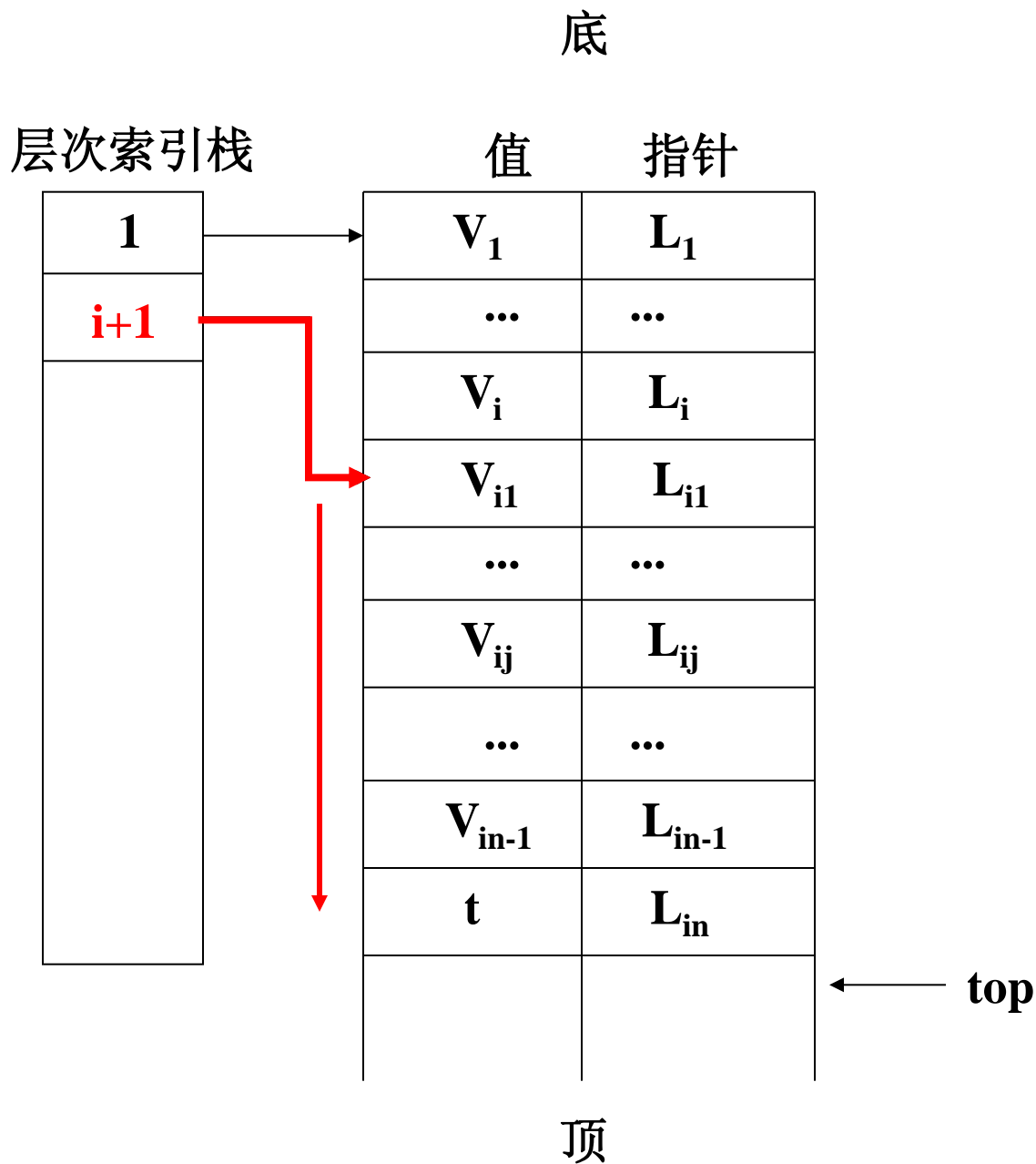
顶



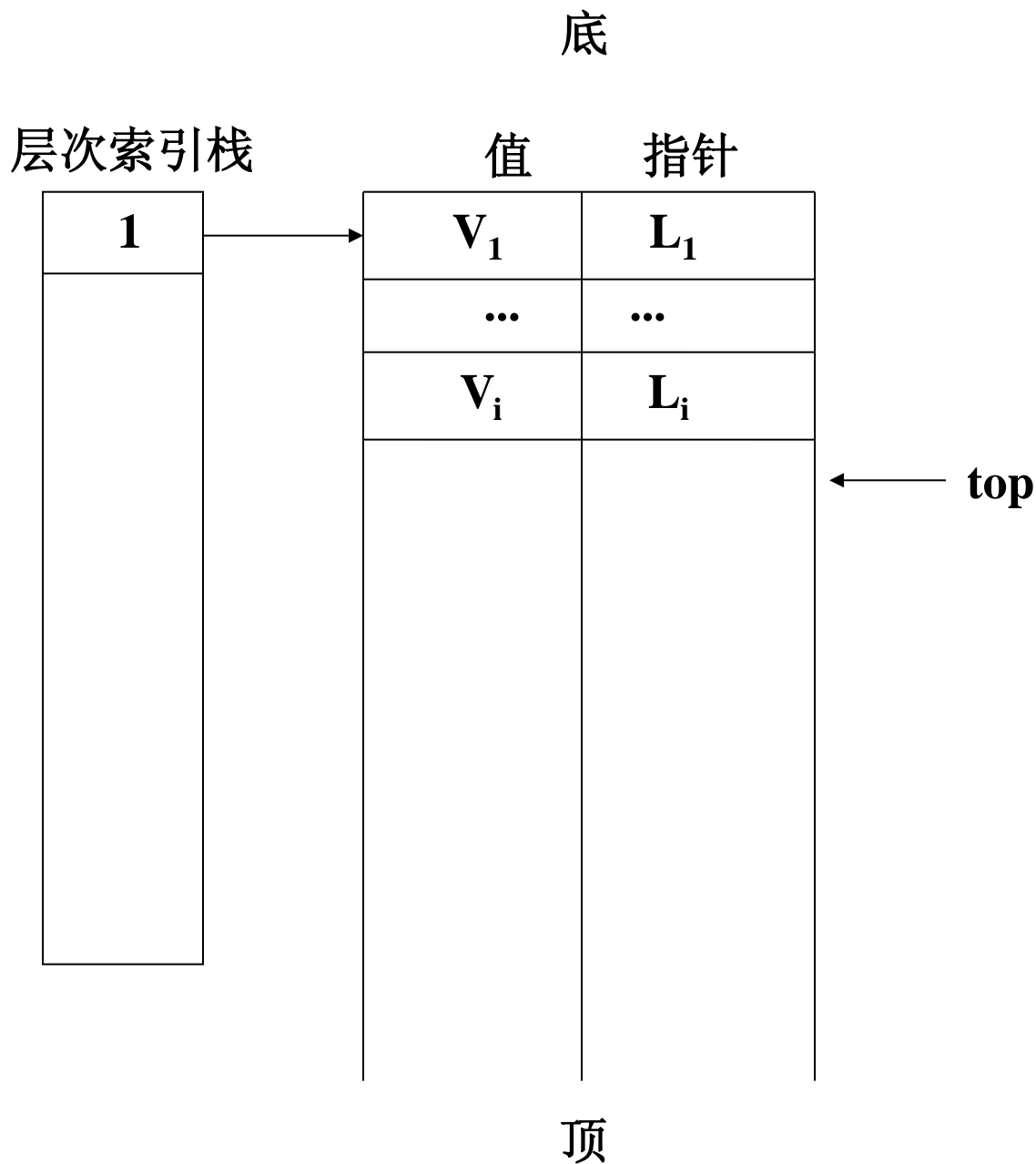
# 多层 case 语句嵌套的情况



# 多层 case 语句嵌套的情况



# 多层 case 语句嵌套的情况



## 8.7 过程调用语句的翻译

- 过程调用语句的文法

**$S \rightarrow \text{call id}(\text{Elist})$**

**$\text{Elist} \rightarrow \text{Elist}, E$**

**$\text{Elist} \rightarrow E$**

- 生成的三地址代码的形式

**param  $E_1$**

**param  $E_2$**

**...**

**param  $E_n$**

**call id, n**

# 翻译思路

- 在处理实参的过程中，生成对参数表达式求值的代码，并记住每个实参的地址
- 在生成call语句之前，按顺序为每个实参生成一条param语句
- 数据结构：队列
- 语义动作
  - ◆  $\text{arglist} \rightarrow \text{arglist}, E$   
将表达式E的存放地址E.place放入队列queue中
  - ◆  $S \rightarrow \text{call id}(\text{arglist})$   
对队列queue中的每一项生成一条param语句，并让这些语句接在对参数表达式求值的那些语句之后。

# 属性及函数设计

- 继承属性**arglist.queue**，保存参数队列的首指针
- 全程变量**count**记录已经加入队列中的参数的个数
- 函数**makequeue()**，创建一个空队列，返回队列首指针
- 过程**append(queue, p)**，将**p**加入到由**queue**指向的队列的队尾
- 函数**get(queue)**，返回队首的参数地址，并将它移出队列，使下一个参数成为队首参数。

# 翻译方案

```
S → call id ( { arglist.queue=makequeue(); count=0 }  
               arglist) { if (count!=0)  
                           while (count>0) {  
                               t=get(arglist.queue);  
                               emit('param' t);  
                               count--;  
                           };  
                           emit('call' id.place); }
```

```
arglist → { arglist1.queue=arglist.queue }  
          arglist1, E { append(arglist.queue, E.place);  
                       count++ }
```

```
argist → E { append(arglist.queue, E.place);  
            count=1 }
```

# 小结

## ■ 中间语言

### ◆ 图形表示

- 树、dag

### ◆ 三地址代码

- 三地址语句的形式:  $x := y \text{ op } z$
- 三地址语句的种类
  - 简单赋值语句
  - 转移语句
  - 过程调用语句
  - 涉及数组元素的赋值语句
  - 涉及指针的赋值语句
- 三地址语句的具体实现
  - 三元式、四元式、间接三元式
  - 比较



# 小结 (续1)

## ■ 赋值语句的翻译

- ◆ 文法（赋值语句出现的环境）
- ◆ 仅涉及简单变量的赋值语句的翻译
- ◆ 涉及数组元素的赋值语句的翻译
  - 计算数组元素的地址
- ◆ 访问记录中的域

## ■ 布尔表达式的翻译

- ◆ 数值方法
- ◆ 控制流方法：代码结构
- ◆ 回填技术
  - 思想、问题、方法
  - 与链表操作有关的函数
    - **makelist**
    - **merge**
    - **backpatch**
  - 属性设计
  - 布尔表达式的翻译

# 小结 (续2)

- 控制语句的翻译
- 转移语句的翻译
- **CASE**语句的翻译
  - ◆ 语法结构、功能
  - ◆ 三地址代码结构
  - ◆ 翻译过程、**CASE**栈
- 过程调用语句的翻译

# 作业 (P. 292)

- 8. 3
- 8. 4
- 8. 6

# 第9章 目标代码生成



*LI Wensheng, SCST, BUPT*

知识点：基本块、程序流图

下次引用信息

代码生成算法

# § 9 目标代码生成

9.1 代码生成概述

9.2 基本块与流图

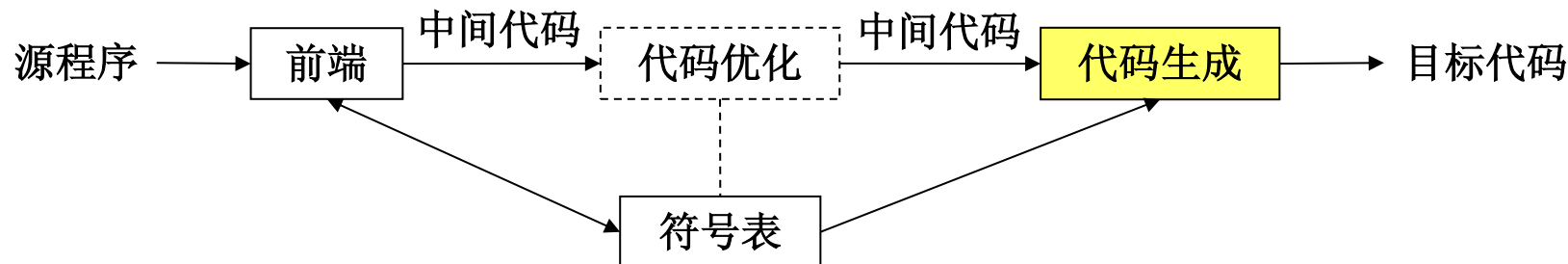
9.3 一个简单的代码生成程序

# 9.1 代码生成概述

- 目标代码生成程序的任务
  - ◆ 将前端产生的中间代码转换为等价的目标代码
- 对目标代码生成程序的要求：
  - ◆ 正确
  - ◆ 高质量
    - 1. 有效地利用目标机器的资源
    - 2. 所生成的目标代码应高效地运行
- 本节内容：
  - ◆ 代码生成程序
  - ◆ 代码生成程序设计有关的问题

# 一、代码生成程序

## ■ 代码生成程序在编译模型中的位置



## ■ 代码生成程序的输入

- 中间代码：经过语法分析/语义检查之后得到的、正确的
- 符号表
  - 记录了与名字有关的信息
  - 决定中间表示中的名字所代表的数据对象的运行地址

## ■ 假定：前期工作结果正确、可信

- 中间代码足够详细、必要的类型转换符已正确插入、明显的语义错误已经发现、且正确恢复

# 代码生成程序(续)

- 代码生成程序的输出：目标代码
- 目标代码形式
  - ◆ 绝对机器代码
    - 可把代码放在内存中固定的地方、立即执行
  - ◆ 可重定位机器代码
    - **.obj (DOS)、.o (UNIX)**
    - 开发灵活，允许各子模块单独编译
    - 由连接装配程序将它们连接在一起，生成可执行文件
  - ◆ 汇编代码



## 二、代码生成程序设计有关的问题

- 代码生成程序的具体细节依赖于目标机器和操作系统
- 所有的代码生成程序固有的问题
  - ◆ 存储管理
  - ◆ 指令选择
  - ◆ 寄存器分配
  - ◆ 计算顺序的选择

# 存储管理

- 从名字到存储单元的转换由前端和代码生成程序共同完成
- 三地址代码中的名字
  - ◆ 指向该名字在符号表中位置的指针
- 符号表中的信息
  - ◆ 在处理声明语句时填入
  - ◆ “类型”决定了它的域宽
  - ◆ “地址”确定该名字在过程的数据区域中的相对位置
  - ◆ 上述信息用于确定中间代码中的名字对应的数据对象在运行时的地址

# 例如：中间代码与目标代码的对应

|       | 四元式       | 地址   | 长度 |
|-------|-----------|------|----|
|       | ...       |      |    |
| → 100 | ( , , , ) | n    | 12 |
| 101   | ( , , , ) | n+12 | 8  |
| 102   | ( , , , ) | n+20 | 16 |
| 103   | ( , , , ) | n+36 | 4  |
|       | ...       |      |    |

|      | 目标代码           |
|------|----------------|
| 0    | ...            |
| n    | 四元式100<br>的机器码 |
| n+12 | 四元式101<br>的机器码 |
| n+20 | 四元式102<br>的机器码 |
| n+36 | 四元式103<br>的机器码 |
| n+40 | ...            |

## ■ 对于四元式j: goto i

◆  $i < j$  四元式i的地址已有，可以直接生成机器指令

◆  $i > j$  将四元式j的地址记入与i相关的链表中，等待回填

# 指令选择

- 机器指令系统的性质决定了指令选择的难易程度

- ◆ 一致性
- ◆ 完整性
- ◆ 指令的执行速度
- ◆ 机器的特点

- 对每一类三地址语句，可以设计它的代码框架

如  $x := y + z$  的代码框架可以是

```
MOV  y, R0
ADD  z, R0
MOV  R0, x
```

$a := b + c$   
 $d := a + e$

```
MOV  b, R0
ADD  c, R0
MOV  R0, a
MOV  a, R0
ADD  e, R0
MOV  R0, d
```

$a := a + 1$       **INC a**

```
MOV  a, R0
ADD  #1, R0
MOV  R0, a
```

# 寄存器分配

- 选出要使用寄存器的变量
  - ◆ 局部范围内
  - ◆ 在程序的某一点上
- 寄存器指派
  - ◆ 可用寄存器
    - 专用寄存器
    - 通用寄存器
    - 寄存器对
  - ◆ 把寄存器指派给相应的变量
    - 变量需要什么样的寄存器
    - 操作需要什么样的寄存器

# 计算次序的选择

- 计算次序影响目标代码的效率

- 如：

- ◆ RISC体系结构的一种通用的流水线限制是：从内存中取出存入寄存器的值在随后的几个周期内是不能用的。在这几个周期期间，可以调出不依赖于该寄存器值的指令来执行，如果找不到这样的指令，则这些周期就会被浪费。所以，对于具有流水线限制的体系结构，选择合适的计算次序是必需的。

- ◆ 有些计算顺序可以用较少的寄存器来保留中间结果

- 代码生成程序的设计原则

- ◆ 能够正确地生成代码

- ◆ 易于实现、便于测试和维护

## 9.2 基本块与流图

### ■ 基本块

- ◆ 具有**原子性**的一组连续语句序列。
- ◆ 控制从第一条语句流入，从最后一条语句流出，中途没有停止或分支

### ■ 如：

$t_1 := a * a$   
 $t_2 := b * b$   
 $t_3 := t_1 + t_2$

### 基本块：

$t_1 := a * a$   
 $t_2 := a * b$   
 $t_3 := 2 * t_2$   
 $t_4 := t_1 + t_3$   
 $t_5 := b * b$   
 $t_6 := t_4 + t_5$

# 基本块的划分方法

- 确定入口语句，下面的语句是入口语句：
  - ◆ 三地址代码的第一条语句
  - ◆ **goto**语句转移到的语句
  - ◆ 紧跟在**goto**语句后面的语句
- 确定基本块，与每一个入口语句相应的基本块：
  - ◆ 从一个入口语句（**含该语句**）到下一个入口语句（**不含**）之间的语句序列
  - ◆ 从一个入口语句（**含该语句**）到停止语句（**含该语句**）之间的语句序列



# Pascal 程序片断:

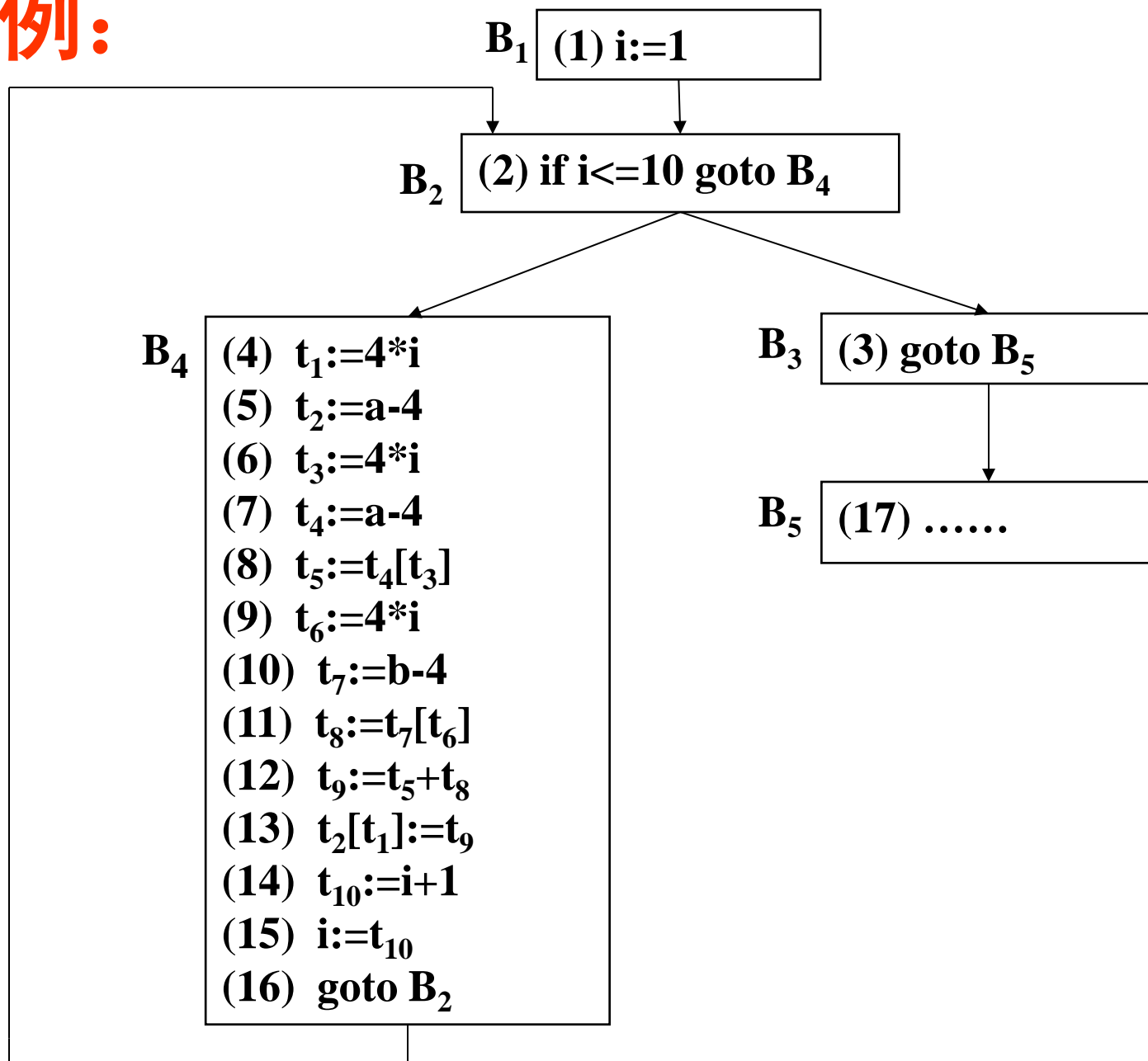
```
i:=1;  
while (i<=10) do  
begin  
    a[i]:=a[i]+b[i];  
    i:=i+1  
end
```

|                               |                |
|-------------------------------|----------------|
| → (1) i:=0                    | B <sub>1</sub> |
| → (2) if i<=10 goto (4)       | B <sub>2</sub> |
| → (3) goto (17)               | B <sub>3</sub> |
| → (4) t1:=4*i                 | B <sub>4</sub> |
| (5) t2:=a-4                   |                |
| (6) t3:=4*i                   |                |
| (7) t4:=a-4                   |                |
| (8) t5:=t4[t3] /* t5=a[i] */  |                |
| (9) t6:=4*i                   |                |
| (10) t7:=b-4                  |                |
| (11) t8:=t7[t6] /* t8=b[i] */ |                |
| (12) t9:=t5+t8                |                |
| (13) t2[t1]:=t9               |                |
| (14) t10:=i+1                 |                |
| (15) i:=t10                   |                |
| (16) goto (2)                 |                |
| → (17) ...                    | B <sub>5</sub> |

# 流图

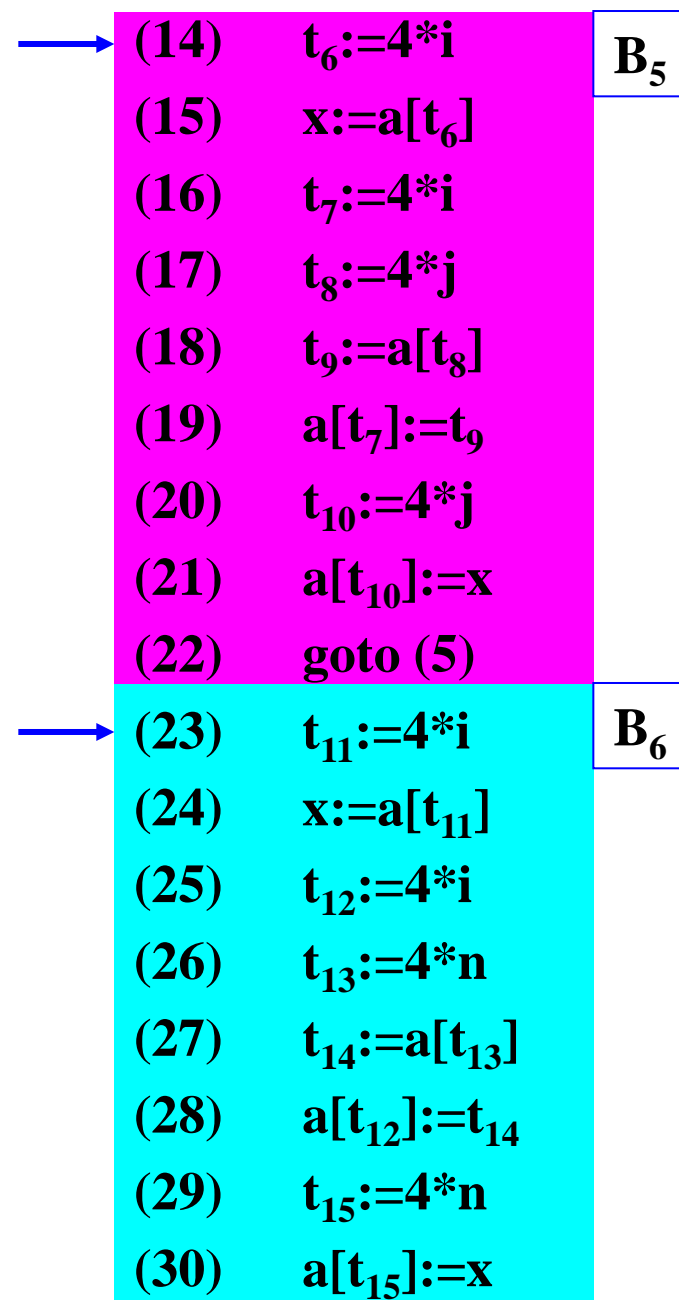
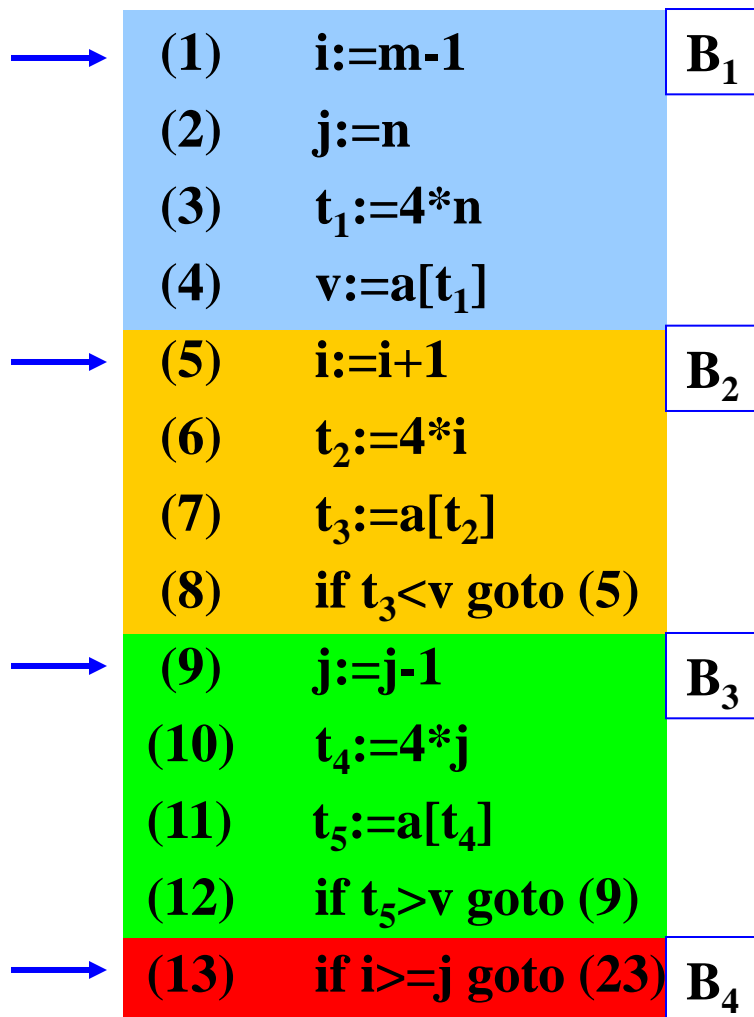
- 把控制信息加到基本块集合中，形成程序的有向图，称为**流图**（控制流图）
- 流图的**结点**是基本块
- 如果一个结点基本块的入口语句是程序的第一条语句，则称此基本块结点为**首结点**。
- 如果在某个执行序列中，基本块 $B_2$ 紧跟在基本块 $B_1$ 之后执行，则从 $B_1$ 到 $B_2$ 有一条有向边， $B_1$ 是 $B_2$ 的**前驱**， $B_2$ 是 $B_1$ 的**后继**。即如果：
  - ◆ 有一个条件/无条件转移语句从 $B_1$ 的最后一条语句转移到 $B_2$ 的第一条语句；
  - ◆  $B_1$ 的最后一条语句不是转移语句，并且在程序的语句序列中， $B_2$ 紧跟在 $B_1$ 之后。

# 流图示例:

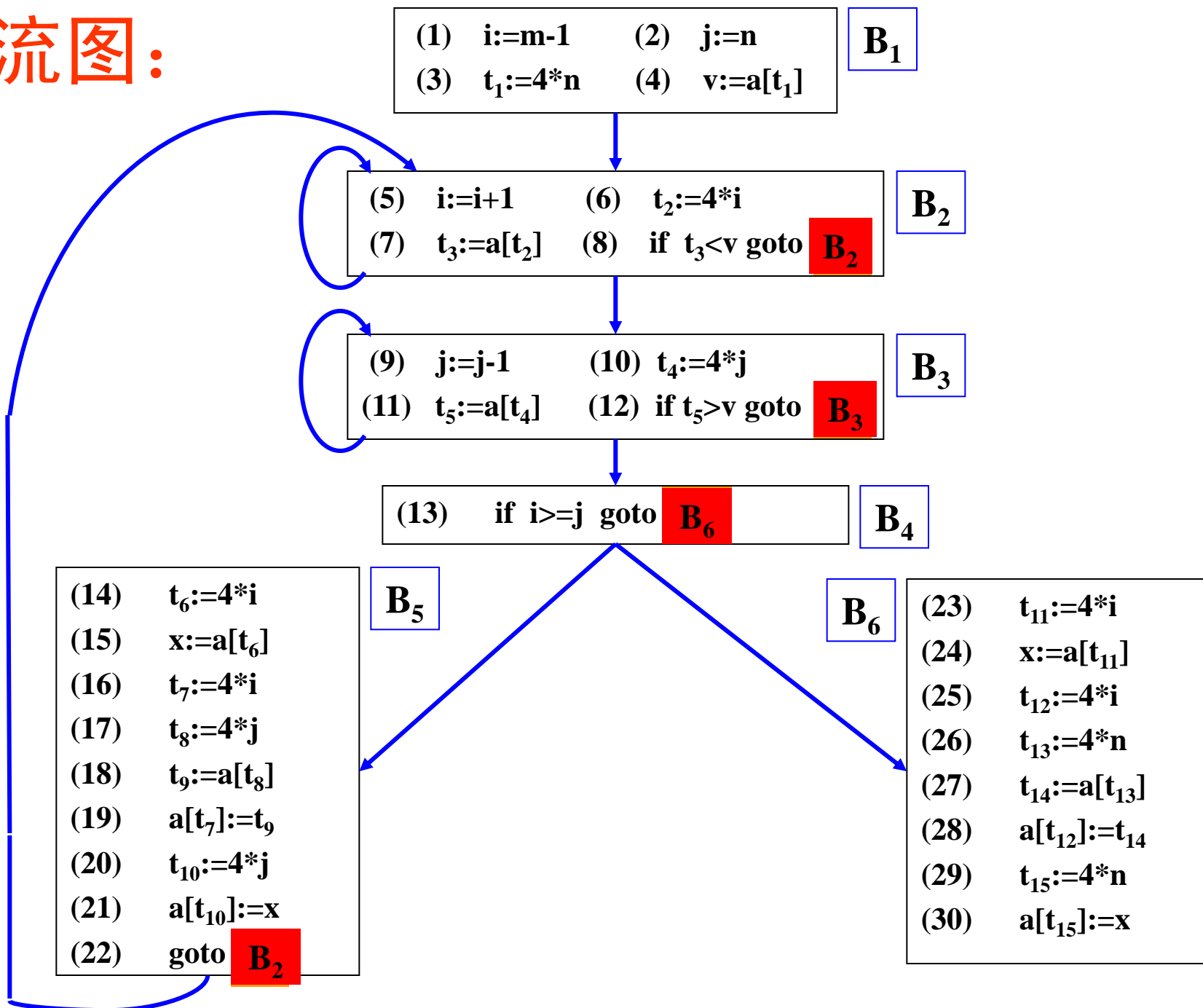


# 举例

## 基本块划分:



# 流图:



## 9.3 一个简单的代码生成程序

- 依次处理基本块中的每条三地址语句
- 考虑在基本块内充分利用寄存器的问題
  - ◆ 当生成计算某变量值的目标代码时，尽可能让变量的值保存在寄存器中（而不产生把该变量的值存入内存单元的指令），直到该寄存器必须用来存放其它的变量值，或已到达基本块的出口为止；
  - ◆ 后续的目标代码尽可能引用变量在寄存器中的值。
- 在基本块之间如何充分利用寄存器的问題比较复杂，简单起见，在离开基本块时，把有关变量在寄存器中的当前值存放到内存单元中去。
- 代码生成时需考察许多情形，如下次引用信息、活跃信息、当前值的存放位置等，在不同的情况下生成的代码也不同。

# 本节内容

- 目标机器概述
- 下次引用信息的计算
- 代码生成算法

# 一、目标机器

- 设计代码生成程序的必要条件：熟悉目标机器

- 一般信息

- ◆ 编址方式：

- 按字节编址
    - 每个字有4个字节

- ◆ 寄存器：

- $n$ 个通用寄存器： $R_0$ 、 $R_1$ 、 $R_{n-1}$

- ◆ 指令形式：

- **OP S, D**

其中 **OP: MOV、ADD、SUB**

**S: 源操作数**

**D: 目的操作数**



# 寻址方式

| 地址形式  | 汇编方式         | 地址                             | 占用存储空间   |
|-------|--------------|--------------------------------|----------|
| 绝对地址  | <b>M</b>     | <b>M</b>                       | <b>1</b> |
| 寄存器   | <b>R</b>     | <b>R</b>                       | <b>0</b> |
| 变址    | <b>c(R)</b>  | <b>c+contents(R)</b>           | <b>1</b> |
| 间接寄存器 | <b>*R</b>    | <b>contents(R)</b>             | <b>0</b> |
| 间接变址  | <b>*c(R)</b> | <b>contents(c+contents(R))</b> | <b>1</b> |
| 立即数   | <b>#c</b>    | <b>常数c</b>                     | <b>1</b> |

## ■ 指令开销

= 指令所占用存储单元字数

= **1** + **S**寻址方式占用字数 + **D**寻址方式占用字数

# 举例

- **MOV R<sub>0</sub>, R<sub>1</sub>**
  - ◆ 将寄存器R<sub>0</sub>的内容复制到R<sub>1</sub>中
  - ◆ 开销: 1
- **MOV R<sub>5</sub>, M**
  - ◆ 将寄存器R<sub>5</sub>的内容放到存储单元M中
  - ◆ 开销: 2
- **ADD #1, R<sub>3</sub>**
  - ◆ 将寄存器R<sub>3</sub>的内容增加1
  - ◆ 开销: 2
- **SUB 4(R<sub>0</sub>), \*12(R<sub>1</sub>))**
  - ◆ 将地址为(contents(12+contents(R<sub>1</sub>)))的单元中的值减去contents(4+contents(R<sub>0</sub>)), 结果仍存放到地址为(contents(12+contents(R<sub>1</sub>)))的单元中。
  - ◆ 开销: 3

# 三地址语句 $a := b + c$ 的代码

(1) **MOV b, R<sub>0</sub>**

**ADD c, R<sub>0</sub>**

**MOV R<sub>0</sub>, a**

指令开销为6

(2) **MOV b, a**

**ADD c, a**

指令开销为6

(3) 假定R<sub>1</sub>和R<sub>2</sub>中分别包含b和c的值，b的值以后不再需要：

**ADD R<sub>2</sub>, R<sub>1</sub>**

**MOV R<sub>1</sub>, a**

指令开销为3

(4) 假定R<sub>0</sub>、R<sub>1</sub>和R<sub>2</sub>中分别存放了a、b和c的地址：

**MOV \*R<sub>1</sub>, \*R<sub>0</sub>**

**ADD \*R<sub>2</sub>, \*R<sub>0</sub>**

指令开销为2

## 二、下次引用信息

- 在把三地址代码转换成为目标代码时，遇到的一个重要问题：

如何充分利用寄存器？

- 基本思路：

- ◆ 在一个基本块范围内考虑
- ◆ 把在基本块内还要被引用的变量的值尽可能保存在寄存器中
- ◆ 把在基本块内不再被引用的变量所占用的寄存器尽早地释放

- 如：翻译语句  $x := y \text{ op } z$

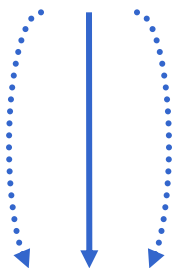
- ◆  $x$ 、 $y$ 、 $z$ 在基本块中是否还会被引用？
- ◆ 在哪些三地址语句中被引用？

# 下次引用信息

三地址语句序列:

i:  $x := 1$

语句i对变量x定值



没有对变量x定值的其他语句

j:  $y := x \text{ op } z$

语句j引用x在语句i处定的值

**j 是三地址语句 i 中 x 的下次引用信息**

## ■ 假定

- 讨论在一个基本块内的引用信息
- 所有的变量在基本块出口处都是活跃的
- 符号表中含有记录下次引用信息和活跃信息的域

# 算法

输入：组成基本块的三地址语句序列

输出：基本块中各名字的下次引用信息

方法：

1. 把基本块中各变量在符号表中的下次引用信息域置为“**无下次引用**”、活跃信息域置为“**活跃**”。
2. 从基本块出口到入口**由后向前**依次处理各语句，对每个三地址语句  **$i: x := y \text{ op } z$** ，依次执行下述步骤：
  - a) 把当前符号表中变量  **$x$**  的下次引用信息和活跃信息附加到语句  **$i$**  上；
  - b) 把符号表中  **$x$**  的下次引用信息和活跃信息分别置为“无下次引用”和“非活跃”；
  - c) 把当前符号表中变量 **$y$ 和 $z$** 的下次引用信息和活跃信息附加到语句  **$i$**  上；
  - d) 把符号表中 **$y$ 和 $z$** 的下次引用信息均置为  **$i$** ，活跃信息均置为“活跃”。

abcd  
次序  
不能  
颠倒  
!

# 计算B4中变量的下次引用信息

## ■ 初始化符号表:

B<sub>4</sub>

- (4)  $t_1 := 4 * i$
- (5)  $t_2 := a - 4$
- (6)  $t_3 := 4 * i$
- (7)  $t_4 := a - 4$
- (8)  $t_5 := t_4[t_3]$
- (9)  $t_6 := 4 * i$
- (10)  $t_7 := b - 4$
- (11)  $t_8 := t_7[t_6]$
- (12)  $t_9 := t_5 + t_8$
- (13)  $t_2[t_1] := t_9$
- (14)  $t_{10} := i + 1$
- (15)  $i := t_{10}$
- (16) goto B<sub>2</sub>

| 变量              | 下次 | 活跃 |
|-----------------|----|----|
| i               | 无  | 活  |
| a               | 无  | 活  |
| b               | 无  | 活  |
| t <sub>1</sub>  | 无  | 活  |
| t <sub>2</sub>  | 无  | 活  |
| t <sub>3</sub>  | 无  | 活  |
| t <sub>4</sub>  | 无  | 活  |
| t <sub>5</sub>  | 无  | 活  |
| t <sub>6</sub>  | 无  | 活  |
| t <sub>7</sub>  | 无  | 活  |
| t <sub>8</sub>  | 无  | 活  |
| t <sub>9</sub>  | 无  | 活  |
| t <sub>10</sub> | 无  | 活  |

# 从出口到入口依次检查每条语句

|   |      |   |
|---|------|---|
| i | (4)  | 活 |
| a | (5)  | 活 |
| b | (10) | 活 |

B<sub>4</sub>

|                          |                 |          |            |             |
|--------------------------|-----------------|----------|------------|-------------|
| (4) $t_1 := 4 * i$       | $t_1$ (13) 活    | i (6) 活  |            |             |
| (5) $t_2 := a - 4$       | $t_2$ (13) 活    | a (7) 活  |            |             |
| (6) $t_3 := 4 * i$       | $t_3$ (8) 活     | i (9) 活  |            |             |
| (7) $t_4 := a - 4$       | $t_4$ (8) 活     | a 无      | 活          |             |
| (8) $t_5 := t_4[t_3]$    | $t_5$ (12) 活    | $t_4$ 无  | 活          | $t_3$ 无 活   |
| (9) $t_6 := 4 * i$       | $t_6$ (11) 活    | i (14) 活 |            |             |
| (10) $t_7 := b - 4$      | $t_7$ (11) 活    | b 无      | 活          |             |
| (11) $t_8 := t_7[t_6]$   | $t_8$ (12) 活    | $t_7$ 无  | 活          | $t_6$ 无 活   |
| (12) $t_9 := t_5 + t_8$  | $t_9$ (13) 活    | $t_5$ 无  | 活          | $t_8$ 无 活   |
| (13) $t_2[t_1] := t_9$   | $t_2$ 无         | 活        | $t_1$ 无    | 活 $t_9$ 无 活 |
| (14) $t_{10} := i + 1$   | $t_{10}$ (15) 活 | i 无      | 非活         |             |
| (15) $i := t_{10}$       | i: 无            | 非活       | $t_{10}$ 无 | 非活          |
| (16) goto B <sub>2</sub> |                 |          |            |             |



# 三、代码生成算法

## ■ 基本思路:

- ◆ 依次把三地址语句转换为目标语言语句
- ◆ 在基本块范围内，充分利用寄存器
- ◆ 尽可能让变量的值保存在寄存器中
- ◆ 后续的代码尽可能引用变量在寄存器中的值
- ◆ 离开基本块时，把有关变量在寄存器中的值送到它的存储单元中

**MOV R, M**

# 考虑引用信息

- 代码生成时，要考察不同的情况
- 不同的情况下，生成的目标代码也不同
- 如  $a := b + c$

(1)  $b$ 的值在 $R_i$ 中， $c$ 的值在 $R_j$ 中，且 $b$ 不再活跃

**ADD  $R_j, R_i$                       开销为 1**

(2)  $b$ 的值在 $R_i$ 中， $c$ 的值在存储单元 $Mc$ 中，且 $b$ 不再活跃

**ADD  $Mc, R_i$                       开销为 2**

或 **MOV  $Mc, R_j$**

**ADD  $R_j, R_i$                       开销为 3**

# 数据结构

## ■ 寄存器描述器

- ◆ 记录每个寄存器的当前内容
- ◆ 开始时，寄存器描述器指示所有的寄存器均为空
- ◆ 代码生成过程中，每个寄存器在任一给定时刻将保留0个或多个名字的值。

## ■ 地址描述器

- ◆ 记录某时刻一个名字的当前值存放的位置，可能是：
  - 一个寄存器地址
  - 一个栈地址
  - 一个存储单元地址
  - 或这些地址的一个集合
- ◆ 这些信息可以存放在符号表中，用来确定对一个名字的存取方式

# 函数getreg

- 输入：三地址语句  $x := y \text{ op } z$

寄存器描述器和名字的地址描述器

- 输出：地址L（L或者是寄存器，或者是存储单元）

- 算法

(1)若y的值在R中，且该R中不含其它名字的值，并且以后y不再活跃，没有下次引用信息，则返回R作为L。否则，转(2)。

(2)有空R时，就返回一个空R作为L。否则，转(3)。

(3)x在块中有下次引用，或op是一个需要寄存器的算符，则找一个已被占用的R。如果R的值尚未在存储单元中，用指令MOV R, M将R的值存放到一个存储单元M中，并且更新地址描述器为M，返回R。

如果R同时保存有几个变量的值，则对每一个需要存储的变量值都应产生一条MOV指令。转(4)。

(4)返回x的存储单元Mx作为L。

# 代码生成算法

输入：基本块的三地址语句    输出：基本块的目标代码

方法：对基本块中每个三地址语句  $x := y \text{ op } z$  执行以下操作

(1)  $L := \text{getreg}(i: x := y \text{ op } z)$

(2) 查看  $y$  的地址描述器，以确定  $y$  的值存放的当前位置  $y'$

- 若  $y$  的值同时存放在存储器和寄存器中，那么选择寄存器作为  $y'$
- 如果  $y$  的值在寄存器  $L$  中，更新  $y$  的地址描述器（ $y$  不在  $L$  中）和  $L$  的寄存器描述器（ $L$  中不含  $y$  的值），转(3)。
- 如果  $y$  的值不在  $L$  中，则生成指令：  $\text{MOV } y', L$

(3) 生成指令：  $\text{op } z', L$

更新  $x$  的地址描述器（ $x$  的值在  $L$  中），如果  $L$  为寄存器，更新  $L$  的寄存器描述器（ $L$  中只含  $x$  的值）

(4) 若  $y/z$  的当前值没有下次引用，在块的出口非活跃，并且在寄存器中，则更新寄存器描述器及  $y/z$  的地址描述器（此后，这些寄存器不再包含  $y$  和/或  $z$  的值）

# 代码生成算法（续1）

- 若三地址语句是  $x := op\ y$ 
  - ◆ 与  $x := y\ op\ z$  类似
- 若三地址语句是  $x := y$ 
  - ◆ 若  $y$  的值在  $R$  中，更新  $R$  的寄存器描述器和  $x$  的地址描述器，以记录  $x$  的值仅在保存  $y$  的那个寄存器中。
    - 若  $y$  没有下次引用，且在块的出口非活跃，则该寄存器不再保留  $y$  的值。
  - ◆ 若  $y$  的值在存储器  $My$  中，则调用函数 `getreg` 来找一个存放  $x$  值的地址  $L$ 
    - 若  $L$  是寄存器，生成指令 `MOV y, L`，并更新  $L$  的寄存器描述器和  $x$ 、 $y$  的地址描述器。
    - 若  $L$  是内存地址，生成指令 `MOV My, Mx`，更新  $x$  的地址描述器。

# 代码生成算法（续2）

- 在基本块的出口，使用MOV指令把那些在块出口是活跃的，且当前值还不在存储单元中的名字的值存储到它们的存储器地址中。
  - ◆ 使用寄存器描述器确定哪些名字的当前值仍保留在寄存器中
  - ◆ 使用地址描述器确定其中哪些名字的当前值还不在存储单元里
  - ◆ 使用活跃变量信息来确定是否需要存储其当前值

# 举例

- 考虑赋值语句  $x := a + b * c$
- 三地址语句序列:  
     $t := b * c$   
     $u := a + t$   
     $x := u$
- 假定在基本块的出口 $x$ 是活跃的
- 有两个寄存器 $R_0$ 和 $R_1$



# 翻译过程

| 语句            | 生成的代码                                                                  | 寄存器描述器                                           | 地址描述器                                              |
|---------------|------------------------------------------------------------------------|--------------------------------------------------|----------------------------------------------------|
|               |                                                                        | 寄存器全空                                            |                                                    |
| <b>t:=b*c</b> | <b>MOV b, R<sub>0</sub></b><br><b>MUL c, R<sub>0</sub></b>             | <b>R<sub>0</sub>含t</b>                           | <b>t在R<sub>0</sub>中</b>                            |
| <b>u:=a+t</b> | <b>MOV a, R<sub>1</sub></b><br><b>ADD R<sub>0</sub>, R<sub>1</sub></b> | <b>R<sub>0</sub>含t</b><br><b>R<sub>1</sub>含u</b> | <b>t在R<sub>0</sub>中</b><br><b>u在R<sub>1</sub>中</b> |
| <b>x:=u</b>   |                                                                        | <b>R<sub>1</sub>含x</b>                           | <b>x在R<sub>1</sub>中</b>                            |
|               | <b>MOV R<sub>1</sub>, X</b>                                            |                                                  | <b>x在R<sub>1</sub>和内存单元中</b>                       |
|               |                                                                        |                                                  |                                                    |

# 为数组元素赋值语句生成目标代码

- 考虑两种语句形式： $a := b[i]$  和  $a[i] := b$
- 假定数组采用静态存储分配
  - ◆ 基址已知
  - ◆ 下标  $i$  存放的位置不同，生成的目标代码也不同

| i的位置       | $a := b[i]$                          | $a[i] := b$                          |
|------------|--------------------------------------|--------------------------------------|
| i在 $R_i$ 中 | MOV b( $R_i$ ), R<br>2               | MOV b, a( $R_i$ )<br>3               |
| i在 $M_i$ 中 | MOV $M_i$ , R<br>MOV b(R), R<br>4    | MOV $M_i$ , R<br>MOV b, a(R)<br>5    |
| i在栈中       | MOV $S_i(A)$ , R<br>MOV b(R), R<br>4 | MOV $S_i(A)$ , R<br>MOV b, a(R)<br>5 |

# 为指针赋值语句生成目标代码

- 考虑两种语句形式： $a := *p$  和  $*p := a$
- $P$ 存放的位置不同，生成的目标代码也不同

| p的位置       | $a := *p$                           | $*p := a$                           |
|------------|-------------------------------------|-------------------------------------|
| p在 $R_p$ 中 | MOV $*R_p, R$<br>1                  | MOV $a, *R_p$<br>2                  |
| p在 $M_p$ 中 | MOV $M_p, R$<br>MOV $*R, R$<br>3    | MOV $M_p, R$<br>MOV $a, *R$<br>4    |
| p在栈中       | MOV $S_p(A), R$<br>MOV $*R, R$<br>3 | MOV $a, R$<br>MOV $R, *S_p(A)$<br>4 |

# 为条件语句生成目标代码

## ■ if a<b goto L

实现:

- a-b的结果送入寄存器R
- 判断R的值为正、负、还是零
- 若为负，则转移到L

## ■ 利用条件码表示计算结果或存入寄存器R的值为正、负、还是零

如: if a<b goto L

目标代码:

```
CMP a, b  
CJ< L
```

语句序列

```
x:=a+b  
if x<0 goto L
```

目标代码:

```
MOV a, R0  
ADD b, R0  
MOV R0, x  
CJ< L
```

# 小 结

## ■ 设计代码生成程序时要考虑的问题

- ◆ 输入、输出
- ◆ 存储管理、寄存器分配
- ◆ 目标机器相关问题（指令、寄存器、编址方式、寻址能力、寻址模式等）
- ◆ 指令选择、计算顺序选择

## ■ 基本块和流图

- ◆ 基本块：具有原子性的语句序列
- ◆ 基本块的划分：入口语句的确定
- ◆ 流图：有向图，结点：基本块，边：控制流

# 小结（续）

- 下次引用信息
  - ◆ 作用
  - ◆ 计算方法
- 代码生成程序
  - ◆ 寄存器描述器
  - ◆ 地址描述器
  - ◆ 寄存器分配函数**getreg**
  - ◆ 代码生成算法

# 作业 (P. 312)

- 9. 1
- 9. 2 (4)

# 第10章 代码优化



*LI Wensheng, SCST, BUPT*

知识点：基本块优化  
循环优化  
窥孔优化



# 代码优化

- 10.1 优化概述
- 10.2 基本块优化
- 10.3 循环优化
- 10.4 窥孔优化
- 小结

# 10.1 优化概述

## ■ 代码优化程序的任务

- ◆ 将前端产生的中间代码转换为等价的目标代码

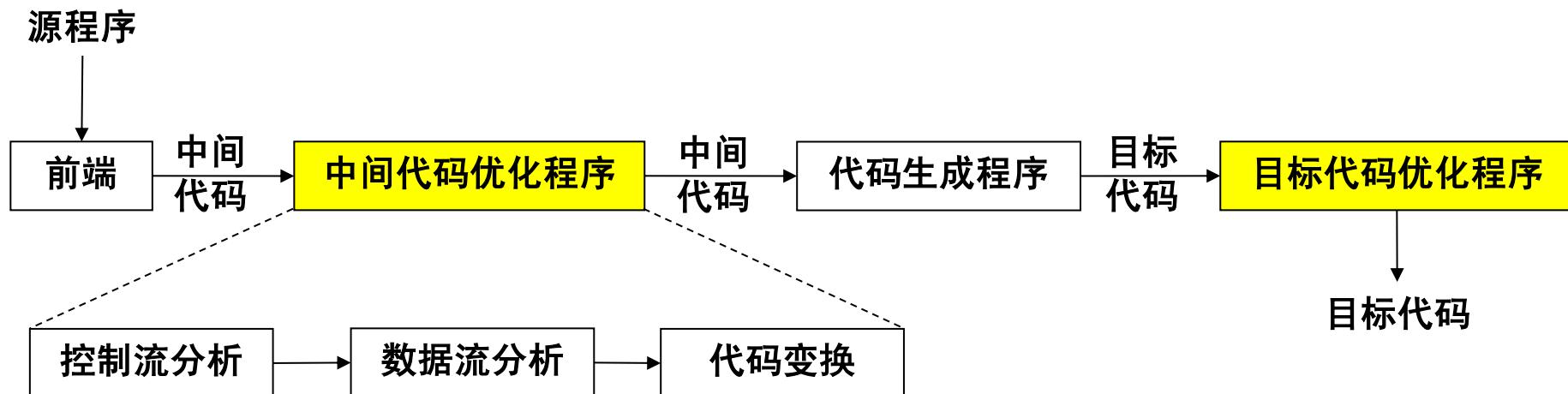
## ■ 代码优化程序的要求

- ◆ 等价变换
- ◆ 提高目标代码的执行速度
- ◆ 减少目标代码占用的空间

## ■ 代码优化程序的地位

- ◆ 目标代码生成之前的中间代码优化
- ◆ 目标代码生成之后的目标代码优化

# 代码优化程序的位置



# 优化的主要种类

## ■ 基本块优化

- ◆ 基本块内进行的优化
- ◆ 常数合并与传播、冗余子表达式的删除、复制传播、削弱计算强度、死代码的删除等

## ■ 循环优化

- ◆ 在循环语句所生成的中间代码序列上进行的优化
- ◆ 循环展开、代码外提、削弱计算强度、删除归纳变量等

## ■ 全局优化

- ◆ 在非线性程序段上（含多个基本块）进行的优化

## ■ 窥孔优化

- ◆ 在目标代码上进行的优化
- ◆ 删除冗余的传送指令、删除死代码、控制流优化、强度削弱及代数化简等

## 10.2 基本块优化

- 一、常数合并及常数传播
- 二、删除冗余的公共表达式
- 三、复制传播
- 四、削弱计算强度
- 五、改变计算次序

# 一、常数合并及常数传播

- 常数合并：将能在编译时计算出值的表达式用其相应的值替代

$x=2+3+y$  可代之以：  $x=5+y$

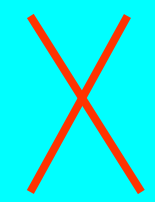
- 常数传播：用在编译时已知的变量值代替程序正文中对这些变量的引用

$PI:=3.14;$


$D\text{-to-R}:= 0.01744$

```
i:=0
10: i:=i+1
...
if i<10 goto 10
```

```
i:=0
10: i:=0+1
...
if i<10 goto 10
```



```
...
a[i]:=9.0
...
a[j]:=3.0
b:=a[i]
```



# 常数合并的实现

- 在符号表中增加两个信息域
  - ◆ 标志域：指示当前是否存在与该变量相关的常数。
  - ◆ 常数域：如果常数存在，则该域存放的即是与该变量相应的当前常数值。
- 常数合并时，注意事项：
  - ◆ 不能将结合律与交换律用于浮点表达式，因为浮点运算的精度有限，这两条定律并非是恒真的。
  - ◆ 不应将任何附加的错误引入

## 二、删除冗余的公共表达式

- 在一个基本块中，当第一次对表达式E求值之后，如果E中的变量都没有改变，再次对E求值，则除E的第一次出现之外，其余的E都是冗余的公共表达式。
- 删除冗余的公共表达式，用第一次出现时的求值结果代替重复求值的结果。

|     |               |
|-----|---------------|
| (1) | <b>a:=b+c</b> |
| (2) | <b>b:=a-d</b> |
| (3) | <b>c:=b+c</b> |
| (4) | <b>d:= b</b>  |



# 举例

B<sub>4</sub>

(4)  $t_1 := 4 * i$

(5)  $t_2 := a - 4$

(6)  $t_3 := 4 * i$

(7)  $t_4 := a - 4$

(8)  $t_5 := t_4[t_3]$

(9)  $t_6 := 4 * i$

(10)  $t_7 := b - 4$

(11)  $t_8 := t_7[t_6]$

(12)  $t_9 := t_5 + t_8$

(13)  $t_2[t_1] := t_9$

(14)  $t_{10} := i + 1$

(15)  $i := t_{10}$

(16) goto B<sub>2</sub>

(4)  $t_1 := 4 * i$

(5)  $t_2 := a - 4$

(6')  $t_3 := t_1$

(7')  $t_4 := t_2$

(8)  $t_5 := t_4[t_3]$

(9')  $t_6 := t_1$

(10)  $t_7 := b - 4$

(11)  $t_8 := t_7[t_6]$

(12)  $t_9 := t_5 + t_8$

(13)  $t_2[t_1] := t_9$

(14)  $t_{10} := i + 1$

(15)  $i := t_{10}$

(16) goto B<sub>2</sub>

# 三、复制传播

- 为减少重复计算，可以利用复制传播来删除公共表达式
- 思想：在复制语句 $f:=g$ 之后，尽可能用 $g$ 代替 $f$

(4)  $t_1:=4*i$

(5)  $t_2:=a-4$

(6')  $t_3:=t_1$

(7')  $t_4:=t_2$

(8)  $t_5:=t_4[t_3]$

(9')  $t_6:=t_1$

(10)  $t_7:=b-4$

(11)  $t_8:=t_7[t_6]$

(12)  $t_9:=t_5+t_8$

(13)  $t_2[t_1]:=t_9$

(14)  $t_{10}:=i+1$

(15)  $i:=t_{10}$

(16) goto  $B_2$

(4)  $t_1:=4*i$

(5)  $t_2:=a-4$

(6')  $t_3:=t_1$

(7')  $t_4:=t_2$

(8')  $t_5:=t_2[t_1]$

(9')  $t_6:=t_1$

(10)  $t_7:=b-4$

(11')  $t_8:=t_7[t_1]$

(12)  $t_9:=t_5+t_8$

(13)  $t_2[t_1]:=t_9$

(14)  $t_{10}:=i+1$

(15)  $i:=t_{10}$

(16) goto  $B_2$

# 删除死代码

- **死代码：** 如果对一个变量 $x$ 求值之后却不引用它的值，则称对 $x$ 求值的代码为死代码。
- **死块：** 控制流不可到达的块称为死块。
  - ◆ 如果一个基本块是在某一条件为真时进入执行的，经数据流分析的结果知该条件恒为假，则此块是死块。
  - ◆ 如果一个基本块是在某个条件为假时才进入执行，而该条件却恒为真，则这个块也是死块。
- 在确定一个基本块是死块之前，需要检查转移到该块的所有转移语句的条件。
- 死块的删除，可能使其后继块成为无控制转入的块，这样的块也成为死块，同样应该删除。

# 删除死代码举例

(4)  $t_1 := 4 * i$

(5)  $t_2 := a - 4$

(6')  $t_3 := t_1$

(7')  $t_4 := t_2$

(8')  $t_5 := t_2[t_1]$

(9')  $t_6 := t_1$

(10)  $t_7 := b - 4$

(11')  $t_8 := t_7[t_1]$

(12)  $t_9 := t_5 + t_8$

(13)  $t_2[t_1] := t_9$

(14)  $t_{10} := i + 1$

(15)  $i := t_{10}$

(16) goto  $B_2$

(4)  $t_1 := 4 * i$

(5)  $t_2 := a - 4$

(8')  $t_5 := t_2[t_1]$

(10)  $t_7 := b - 4$

(11')  $t_8 := t_7[t_1]$

(12)  $t_9 := t_5 + t_8$

(13)  $t_2[t_1] := t_9$

(14)  $t_{10} := i + 1$

(15)  $i := t_{10}$

(16) goto  $B_2$

(4)  $t_1 := 4 * i$

(5)  $t_2 := a - 4$

(8')  $t_5 := t_2[t_1]$

(10)  $t_7 := b - 4$

(11')  $t_8 := t_7[t_1]$

(12)  $t_9 := t_5 + t_8$

(13)  $t_2[t_1] := t_9$

(15')  $i := i + 1$

(16) goto  $B_2$

## 四、削弱计算强度

- 对基本块的代数变换：对表达式用代数上等价的形式替换，以便使复杂的运算变换成为简单的运算。

$$\mathbf{x} := \mathbf{y} ** 2$$

可以用代数上等价的乘式（如： $\mathbf{x} := \mathbf{y} * \mathbf{y}$ ）代替

- $\mathbf{x} := \mathbf{x} + 0$  和  $\mathbf{x} := \mathbf{x} * 1$ 
  - ◆ 执行的运算没有任何意义
  - ◆ 应将这样的语句应从基本块中删除。

# 五、改变计算次序

- 考虑语句序列:

$t_1 := b + c$

$t_2 := x + y$

如果这两个语句是互不依赖的，即  $x$ 、 $y$  均不为  $t_1$ ， $b$ 、 $c$  均不为  $t_2$ ，则交换这两个语句的位置不影响基本块的值。

- 对基本块中的临时变量重新命名不会改变基本块的值，如：语句  $t := b + c$

改成语句  $u := b + c$

把块中出现的所有  $t$  都改成  $u$ ，不改变基本块的值。

## 10.3 循环优化

- 对循环结构所生成的中间代码可划分为如下几个部分：
- **初始化部分**：循环控制变量被赋予一个初值，此部分组成一个基本块，它在循环体中的语句开始之前执行一次。
- **调节部分**：循环控制变量增加或减少一个特定的量，可把这部分视为构成该循环的最后一个基本块。
- **测试部分**：测试循环控制变量以判定终止条件是否满足。这部分的位置依赖于循环语句的性质，若循环语句允许循环体执行0次，则在执行循环体之前进行测试，若循环语句要求循环体至少执行1次，则在执行循环体之后进行测试。
- **循环体**：由需要重复执行的语句构成的一个或多个基本块组成。
- 循环结构中的**调节部分**和**测试部分**也可以与**循环体**中的其他语句一起出现在基本块中。

# 循环优化的主要技术

- 一、循环展开
- 二、代码外提/频度削弱
- 三、削弱计算强度
- 四、删除归纳变量



# 一、循环展开

- 将构成循环体的代码（不包括调节和测试部分）重复产生多次，而不仅一次。产生的次数可以在编译时确定。
- 以空间换时间的优化过程
- 要进行循环展开，必须：
  - ◆ 必须识别出循环结构，而且循环的初值、终值、以及步长的值都必须能够在编译时确定。
  - ◆ 作出空间与时间的权衡，是否可接受，如不可接受，则将此循环作为一个循环结构继续编译。
  - ◆ 如果可以接受，则重复产生循环体直到所需要的次数，
- 在重复产生代码时，必须确保每次重复产生时，都对循环变量进行了正确的合并。

考虑C语言的循环语句:

```
for (i=0;i<10;i++)  
    x[i]=0;
```

■ 生成三地址代码:

100: i:=0

101: if i<10 goto 103

102: goto 107

103: t:=4\*I

104: x[t]:=0

105: i:=i+1

106: goto 101

107:

x[0]:=0

x[4]:=0

...

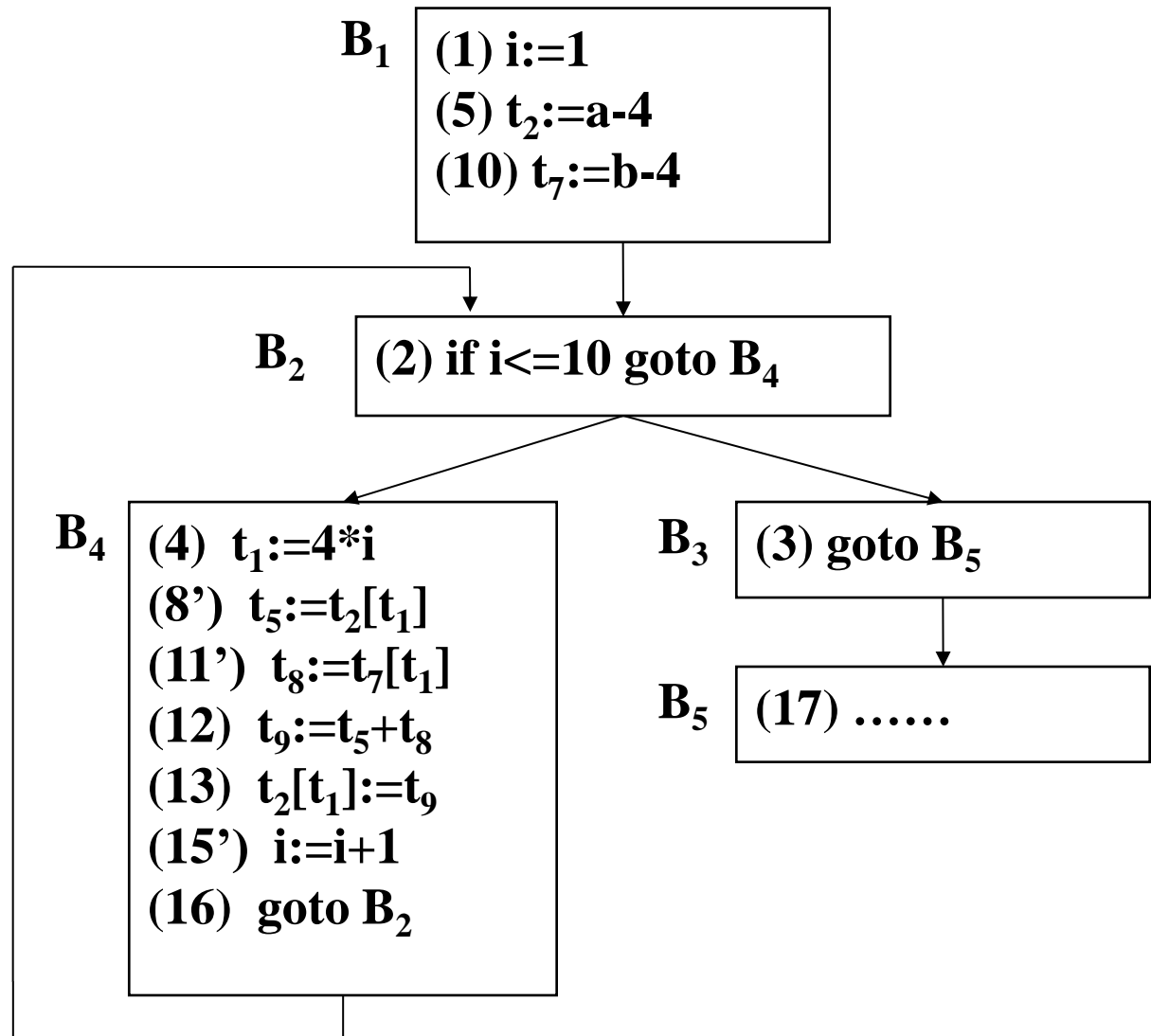
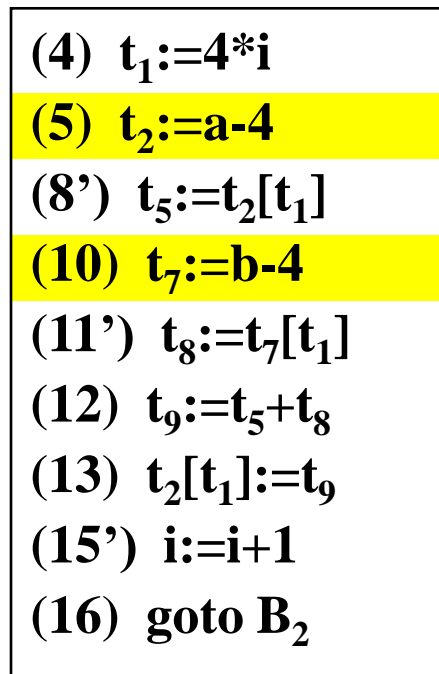
x[36]:=0

■ 循环展开:

## 二、代码外提/频度削弱

- 减少循环中代码总数的一个重要方法
  - 对于包含在循环里的基本块，如果其中的某些代码与循环无关，则可将这样的代码移到循环之外以减少其计算频度，通常移到循环的前边。
  - 如C语言程序中的语句：  
while (i<=limit-2) {  
    ...  
}
- ```
t:=limit-2;  
while (i<=t) {  
    ...  
}
```
- 如果limit的值在循环过程中保持不变，则limit-2的计算与循环无关

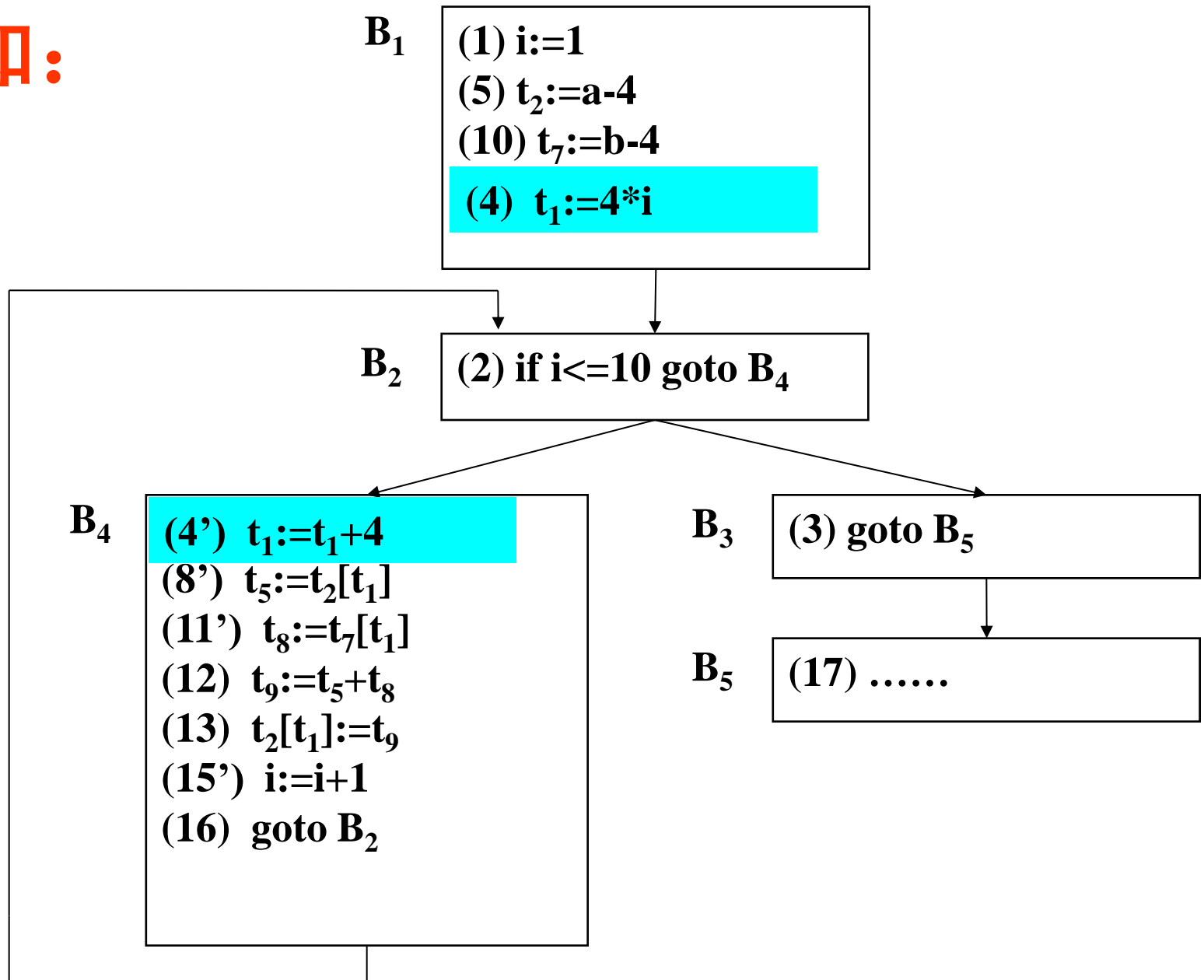
# 例如:



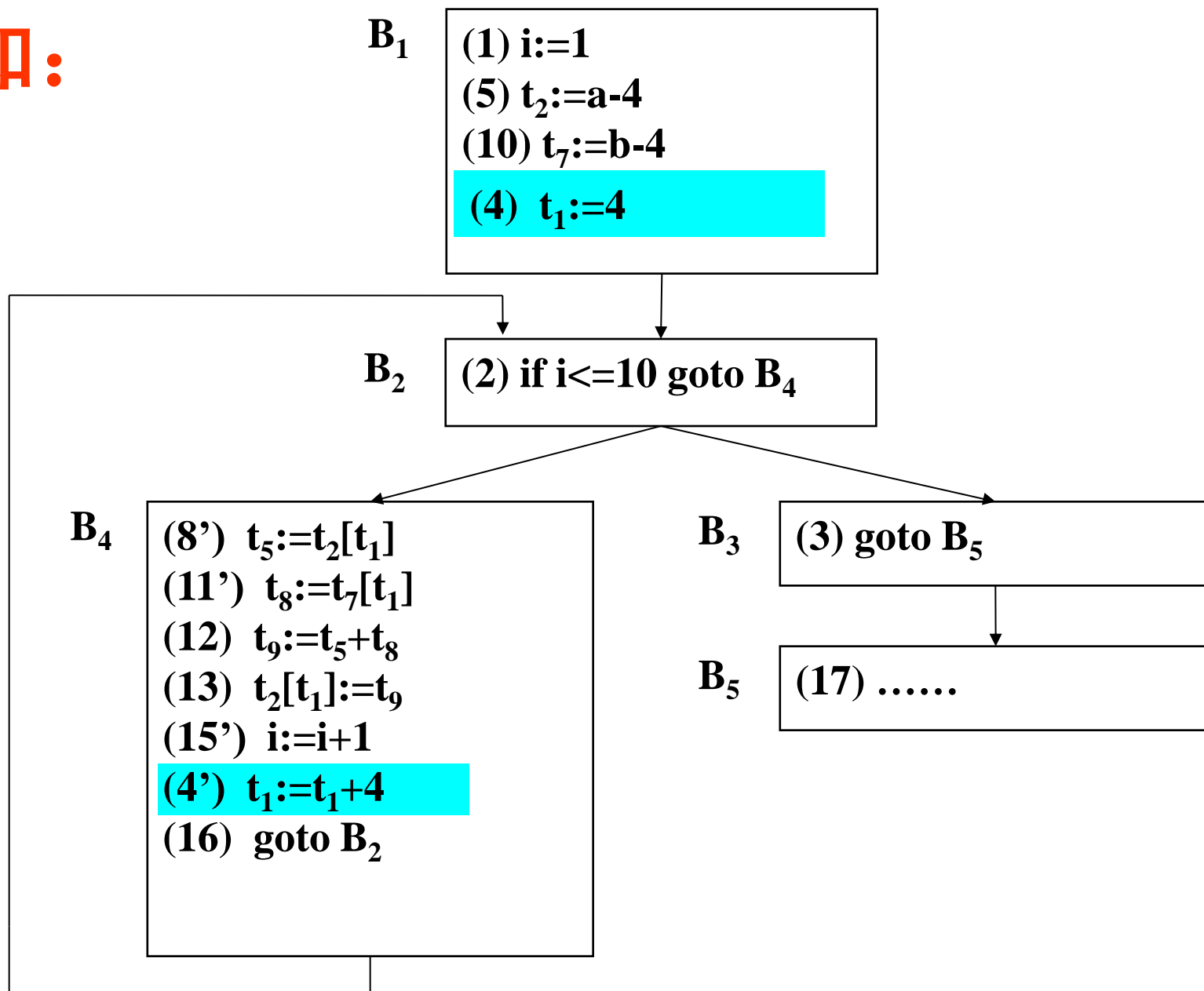
### 三、削弱计算强度

- 将一种类型的运算代之以另一种类型的需要较少执行时间的运算。
- 大多数计算机上乘法运算比加法运算需要更多的执行时间。
- 如可用 '+' 代替 '\*', 则可节省许多时间, 特别是当这种替代发生在循环中时更是如此。

例如:



例如:



## 四、归纳变量的删除

- 如果循环中对变量**I**只有唯一的形如  $\mathbf{I} := \mathbf{I} + \mathbf{C}$  的赋值，并且**C**为循环不变量，则称**I**为循环中的基本归纳变量。
- 如果**I**是循环中的一个基本归纳变量，**J**在循环中的定值总可以化归为**I**的同一线性函数，即  $\mathbf{J} := \mathbf{C1} * \mathbf{I} + \mathbf{C2}$ ，这里**C1**和**C2**都是循环不变量，则称**J**是归纳变量，并称**J**与**I**同族。
- 如：基本块**B4**中
  - ◆ **i**是基本归纳变量
  - ◆  $\mathbf{t_1} := 4 * \mathbf{i}$
  - ◆ **t<sub>1</sub>**是与**i**同族的归纳变量

**B<sub>4</sub>**

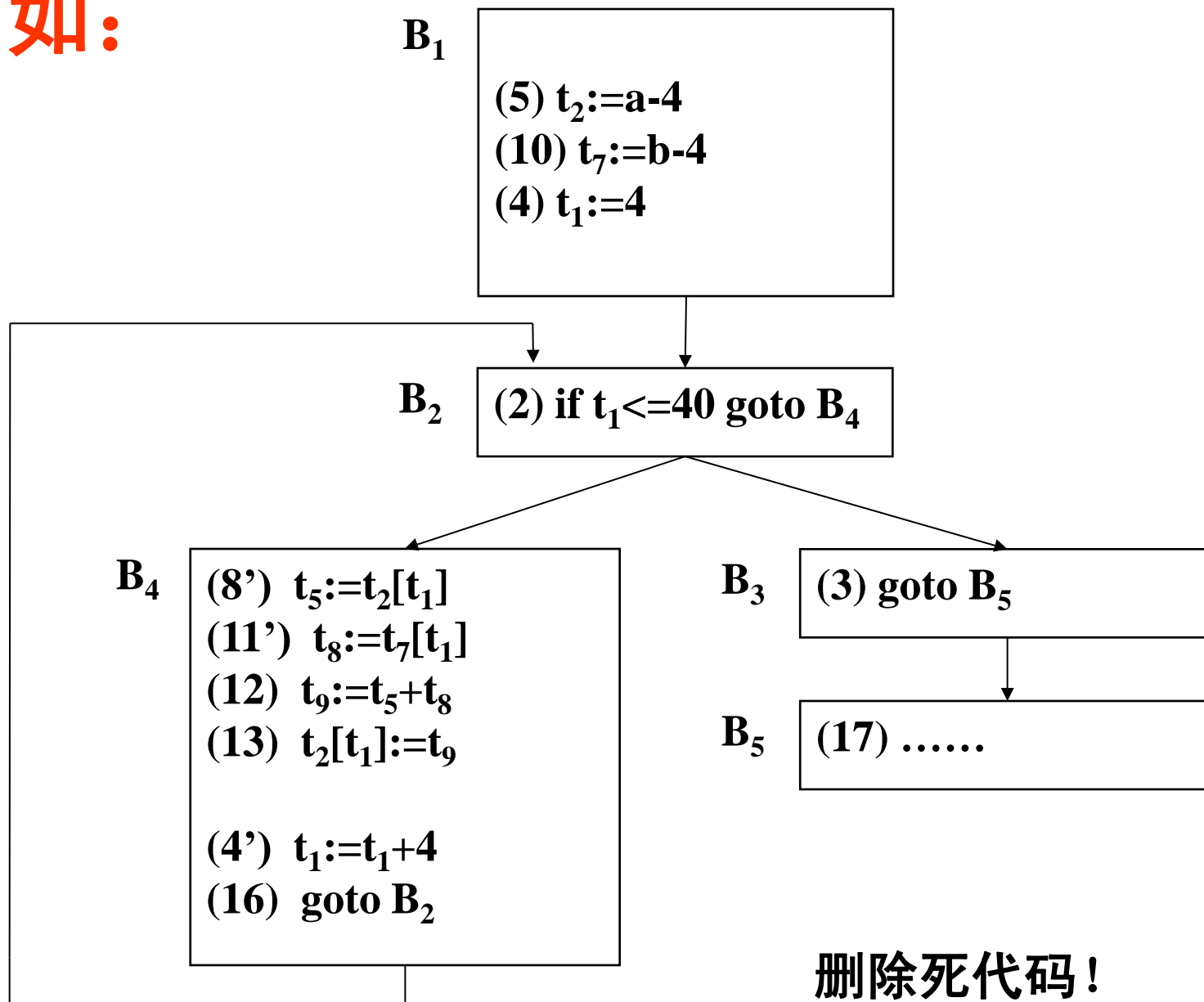
(8')  $\mathbf{t_5} := \mathbf{t_2[t_1]}$   
(11')  $\mathbf{t_8} := \mathbf{t_7[t_1]}$   
(12)  $\mathbf{t_9} := \mathbf{t_5} + \mathbf{t_8}$   
(13)  $\mathbf{t_2[t_1]} := \mathbf{t_9}$   
(15')  $\mathbf{i} := \mathbf{i} + 1$   
(4')  $\mathbf{t_1} := \mathbf{t_1} + 4$   
(16) goto **B<sub>2</sub>**



# 归纳变量的删除(续)

- 通常，一个基本归纳变量除用于其自身的递归定值外，往往只在循环中用于计算其他归纳变量的值、以及用来控制循环的进行。
- 由于 $t_1$ 和 $i$ 之间具有线性函数关系： $t_1=4*i$   
所以， $i \leq 20$  与  $t_1 \leq 80$  是等价的。  
因此，可以用  $t_1 \leq 40$  来替换  $i \leq 10$   
语句(2)变换为：if  $t_1 \leq 40$  goto  $B_4$

例如:



删除死代码!

## 10.4 窥孔优化

- 目标代码局部改进的简单有效技术
- 在目标程序上设置一个可移动的小窗口，称为窥孔，通过窥孔只能看到目标代码中有限的几条指令，只要有可能，就代之以较短或较快的等价的指令序列。
- 窥孔中的代码可以不连续
- 特点：每个改进都可能会带来新的改进机会
- 常作为改进目标代码质量的技术，也可以用于中间代码的优化

# 典型的窥孔优化技术

- 一、删除冗余的传送指令
- 二、删除死代码
- 三、控制流优化
- 四、强度削弱及代数化简

# 一、删除冗余的传送指令

- 如果窥孔中出现下列指令：
  - (1) **MOV  $R_0$ , a**
  - (2) **MOV a,  $R_0$**
- 通常可以删除指令(2)
- 如果指令(2)带有标号，则不能保证指令(2)紧跟在指令(1)之后执行，所以，在这种情况下，就不能删除指令(2)。
- 当这两个语句在同一基本块中时，删除指令(2)是安全的。

## 二、删除死代码

- 程序中控制流不可到达的一段代码称为死代码。
- 如果无条件转移指令的下一条指令没有标号，则它是死代码，应该删除。这种操作有时会连续进行，从而删除一串指令。
- 若条件转移语句中的条件表达式的值是个常量，则生成的目标代码势必有一个分支成为死代码。
- 如：为了调试一个较大的C语言程序，通常需要在程序里插入一些用于跟踪调试的语句，当调试完成之后，可能不删除这些语句，而只令其成为死代码。

# 例：程序里插入的跟踪调试语句

```
#define debug 0
```

```
...
```

```
if debug {
```

```
...          /* 输出调试信息 */
```

```
}
```

- 这一结构翻译出的中间代码可能是：

```
if debug=1 goto L1
```

```
goto L2
```

```
L1: ...          /* 输出调试信息 */
```

```
L2: ...
```

- 需要把从if到L<sub>1</sub>所标识的全部语句删除

# 三、控制流优化

- 连续跳转的goto语句:

goto L<sub>2</sub>

...

L<sub>1</sub>: goto L<sub>2</sub>

- 条件转移语句:

if a < b goto L<sub>2</sub>

...

L<sub>1</sub>: goto L<sub>2</sub>

- 如果控制结构为:

goto L<sub>1</sub>

...

L<sub>1</sub>: if a < b goto L<sub>2</sub>

L<sub>3</sub>: ...

- 如果只有一个转移到L<sub>1</sub>,  
且L<sub>1</sub>前边是一无条件转移指令:

if a < b goto L<sub>2</sub>

goto L<sub>3</sub>

...

L<sub>3</sub>: ...



## 四、强度削弱及代数化简

- 用等价的执行速度较快的机器指令代替执行速度慢的指令，以削弱计算强度。
- 特定的目标机器上，某些机器指令比其它一些指令执行要快得多，因而常作为较慢指令的特例。
- 如：
  - ◆ 用  $x * x$  实现  $x^2$  比调用指数例程一定要快得多。
  - ◆ 用移位操作实现定点数乘以或除以 2 的幂运算肯定会快些。
  - ◆ 浮点数除以常数用乘以常数近似实现会快些等。
- 窥孔优化时，有许多代数化简可以尝试，但经常出现的代数恒等式只有少数几个，如：  
$$x := x + 0 \quad \text{或}$$
$$x := x * 1$$
- 在简单的中间代码生成算法中经常出现这样的语句，它们很容易由窥孔优化删除。

# 利用机器的特点

- 目标机器可能有高效实现某些专门操作的硬指令，找出允许使用这些指令的情况可以使执行时间有可观的缩短。如：
  - ◆ 某些机器有自动增1或自动减1的寻址模式，它们在使用运算对象前或后，对它加1或减1。
  - ◆ 在参数传递的压栈和退栈时，用这些模式可大大改进代码质量。
  - ◆ 这些模式还可以用于实现语句 $i:=i+1$ 的代码等。

# 小 结

- 代码优化程序的功能
  - ◆ 等价变换
  - ◆ 执行时间
  - ◆ 占用空间
- 代码优化程序的组织
  - ◆ 控制流分析
  - ◆ 数据流分析
  - ◆ 代码变换
- 优化种类
  - ◆ 基本块优化
  - ◆ 循环优化
  - ◆ 窥孔优化

# 小结（续）

- 基本块优化的主要技术
  - ◆ 常数合并与常数传播
  - ◆ 删除冗余的公共表达式
  - ◆ 复制传播
  - ◆ 删除死代码
  - ◆ 削弱计算强度
  - ◆ 改变计算次序

# 小结（续）

## ■ 循环优化的主要技术

- ◆ 循环展开
- ◆ 代码外提/频度削弱
- ◆ 削弱计算强度
- ◆ 归纳变量的删除

## ■ 窥孔优化的主要技术

- ◆ 冗余指令的删除
- ◆ 控制流优化
- ◆ 代数化简
- ◆ 强度削弱
- ◆ 利用机器的特点

# 作业 (P. 336)

- 10. 2
- 10. 4
- 10. 5