



*School of Computer Science & Technology*  
*Harbin Institute of Technology*



# 编译原理

## Compiler Principles and Techniques

主讲：姜守旭 博士/教授/教学带头人/博导

助教：林世荣

办公室：综合楼808 办公电话：86403492-808

手机：13936168008

email: [jsx@hit.edu.cn](mailto:jsx@hit.edu.cn)

课程网站: <http://cst.hit.edu.cn/comp>

博客: <http://blog.hit.edu.cn/jsx>

答疑地点：综合楼808室

答疑时间：???





# 课程性质与特点

---

- 课程性质
  - 技术基础
- 基础知识要求
  - 高级程序设计语言，数据结构与算法，形式语言与自动机
- 主要特点
  - 既有理论，又有实践
  - 面向系统设计
  - 涉及程序的自动生成技术



# 教学目的——《编译原理》是一门非常好的课程

- Alfred V.Aho: 编写编译器的原理和技术具有十分普遍的意义，以至于在每个计算机科学家的研究生涯中，本课程中的原理和技术都会反复用到
- 本课程将兼顾语言的描述方法、设计与应用(形式化)
  - 能形式化就能自动化(抽象→符号化→机械化)
  - 可以使学生对程序设计语言具有更加深刻的理解
  - 体验实现自动计算的乐趣
- 涉及的是一个比较适当的抽象层面上的数据变换(既抽象又实际，既有理论又有实践)
- 一个相当规模的系统的设计
  - 总体结构
  - 若干具体的表示和变换算法



# 教学目的(续)

---

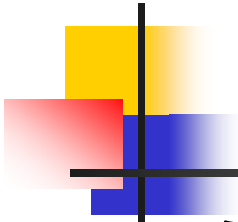
- 在**系统级**上认识算法、系统的设计
  - 具有把握系统的能力
  - 局部最优vs. 全局最优(木桶效用)
  - “自顶向下”和“自底向上”的系统设计方法
    - 对其思想、方法、实现的全方位讨论
- 进一步培养“**计算思维能力**”
  - 深入理解软件系统的非物理性质
  - 培养抽象思维能力和逻辑思维能力
  - 训练对复杂数据结构的设计和操纵能力



# 教学目的(续)

---

- 计算机专业最为恰当、有效的知识载体之一
- 综合运用下列课程所学知识
  - 高级程序设计语言
  - 汇编语言
  - 集合论与图论
  - 数据结构与算法
  - 计算机组成原理
  - 算法设计与分析
  - 形式语言与自动机



# 教学要求——课程要求

## ■ 知识要求

- 掌握编译程序的总体结构、编译程序各个组成部分的任务、编译过程各个阶段的工作原理、编译过程各个阶段所要解决的问题及其采用的方法和技术

## ■ 能力要求

1. 掌握程序变换基本概念、问题描述和处理方法
2. 增强理论结合实际能力
3. 修养“问题、形式化描述、计算机化”的问题求解过程
4. 使学生在系统级上认识算法和系统的设计，培养系统能力



# 教学要求——实验要求

---

## ■ 实验形式

- 分析、设计、编写、调试、测试程序
- 撰写实验报告
- 答辩

## ■ 实验内容

- 词法分析器的设计与实现 6学时
- 语法分析器的设计与实现 12学时
- 语义分析与中间代码生成 6学时



# 教学要求——实验目的

- 实验贯穿于理论、抽象和设计过程;
- 实验对软件的设计和实现、测试原理和方法起示范作用;
- 实验不仅仅是对理论的验证,重要的是**技术训练和能力培养**,包括动手能力、分析问题解决问题能力、表达能力、写作能力等的培养;
- 教学活动是教师和学生不断交流的过程, **实验是实现这个过程桥梁**,可以弥补课堂教学的不足,加深理论过程的理解,启发学生深入思考,敢于创新,达到良好的理论联系实际的教学效果。





# 教学要求——考试要求

---

## ■ 题型

- 选择、填空、判断、简答、证明、论述、设计、计算等

## ■ 重点和难点

- 会在各章的开始点明

## ■ 考试权重

- 出勤占5%
- 作业占5%
- 实验占20%
- 期末考试占70%

## ■ 考前答疑

- 考试前两天



# 教学方法

---

- 围绕一条主线展开
  - 编译过程的各个阶段
- 面向系统
  - 从系统的角度，引导大家逐步建立系统观和工程观，并学会折衷
- 启发式
  - 问题驱动，引导大家理解问题和方法的直观背景
  - 以学生为中心，注重课堂交互，鼓励大家多发问
- 面向应用
  - 引导大家了解技术、方法的应用背景
- 注重实践
  - 以编写一个小型语言编译器为目标



# 学习方法——教中学、做中学、创中学

## ■ 基于问题的学习（What-Why-how）

- 学习要以思考为基础
- 一般的学习只是一种模仿，而没有任何创用
- 思考由怀疑和答案组成，学习便是经常怀疑，经常随时发问。怀疑是智慧的大门，知道得越多，就越会发问，而问题就越多。所以，发问使人进步，发问和答案一样重要。

## ■ 基础知识是研究的工具

- 在独立思考之前，必须先有基础知识。所谓“获得基础知识”并不是形式上读过某门课程，而是将学过的东西完全弄懂(什么叫做精通C语言？)。



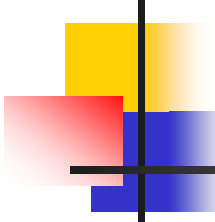
# 学习方法——教中学、做中学、创中学

## ■ 要敢于犯错误

- 学习的一种方法，经常还是唯一的方法，就在于首先犯错误。我们在学习，多数时间在通过犯错误学习。

## ■ 教学、学习是一个过程

- 是毛毛雨，需不断地滋润
- 教师在传授知识和技术的过程中，偶尔会传授教训，
- 但这种教训如果没有经过你的亲身体验，不会变成有用的经验。
- 知识没有教训作为根基，只能是纸上谈兵。
- 上课、读书、复习、做作业、讨论、做实验、自己编程序、上机调试排错...是绝对必要的



# 学习方法——教中学、做中学、创中学

---

## ■ 辅导答疑

- 充分利用好答疑时间，是与老师交流的机会，会获得意想不到的东西
- 没有经你思考的习题、问题最好暂时不问，否则收获不大
- 把老师看成朋友或者长者，这时除谈业务外，谈理想、人生、道德、责任、如何做人...

## ■ 把编译的每个阶段放到整个编译程序背景中学习



# 寄语

---

- 要主动学习
  - 不要苛求课程、老师和环境，他/她/它们只是资源
  - 目标确定后要善于利用各种资源
- 注重对自己能力的培养
- 学会做人，乐于助人，多为别人着想，可以获取友谊
  - 朋友是资源，可以终生受益
- 学会安排自己的时间
  - 时间就像海绵里的水，只要肯挤，总会有的。贵在恒。
- 学会利用各种资源提高自己
  - 学校的、家庭的、社会的.....
  - 上学期间利用资源的唯一目的就是提高自己
- 不要沉迷于网络聊天与游戏



# 主要内容

---

1. 引论
2. 高级语言及其文法
3. 词法分析
4. 自顶向下的语法分析
5. 自底向上的语法分析
6. 语法制导翻译与属性文法
7. 语义分析与中间代码生成
8. 符号表管理
9. 运行时的存储组织
10. 代码优化
11. 代码生成



2012-4-26

16





# Alfred V. Aho

---

**Alfred V. Aho**博士是哥伦比亚大学的劳伦斯科斯基曼计算机科学教授，于普林斯顿大学获得博士学位，**IEEE**、**ACM Fellow**，美国科学与艺术学院及国家工程学院院士，曾获得**IEEE**的冯·诺伊曼奖。“龙书”的第一作者，**AWK**的发明者之一。他目前的研究方向为量子计算、程序设计语言、编译器和算法等。他还赢得了2003年大学毕业生社群的最佳教师奖



*School of Computer Science & Technology*  
*Harbin Institute of Technology*



# 第一章 引论

**重点:** 教学目的, 教学要求, 学习方法, 课程的基本内容, 编译系统的结构, 编译程序的生成。  
**难点:** 编译程序的生成。





# 第1章 引论

---

1.1 程序设计语言

1.2 程序设计语言的翻译

1.3 编译程序的总体结构

1.4 编译程序的组织

1.5 编译程序的生成

1.6 本章小结

```
assume cs:code, ds:data
data segment
```

```
    dw 1234h,5678h
```

```
data ends
```

```
code segment
```

```
start: mov ax, data
```

```
    mov ds, ax
```

```
    mov ax, ds:[0]
```

```
    mov bx, ds:[2]
```

```
    mov cx, 0
```

```
    add cx, ax
```

```
    add cx, bx
```

```
    mov cx, ds:[4]
```

```
    mov ax, 4c00h
```

```
    int 21h
```

```
code ends
```

```
end start
```

```
int main
```

```
{
```

```
    int a,b,c;
```

```
    a=1234h;
```

```
    b=5678h;
```

```
    c=a+b;
```

```
    return 0;
```

```
}
```

```
A100
```

```
000 10
```

```
000 0000 (8B1E0200)
```

```
011 1001 0000 0000 0000 0000
```

```
390000)
```

```
000 0011 1100 1000 (03C8)
```

```
000 0011 1100 1011 (03CB)
```

```
000 1011 0000 1110 0000 0100
```

```
0000 0000 (8B0E0400)
```

```
1011 1000 0000 0000 0100 1100
```

```
(B8004C)
```

```
1100 1101 0010 0001 (CD21)
```

■ 如UNIX上的sh



# 程序设计语言的分类

---

- **强制式（命令式）语言(Imperative Language)**
  - 通过指明一系列可执行的运算及运算的次序来描述计算过程的语言；
  - **FORTRAN(段结构)、BASIC、Pascal(嵌套结构)、C.....**
  - 程序的层次性和抽象性不高



# 程序设计语言的分类

---

- 申述式语言 ( **Declarative Language** )
  - 着重描述要处理什么，而非如何处理的非命令式语言
  - 函数(应用)式语言(**Functional Language**)
    - 基本运算单位是函数，如LISP、ML.....
  - 逻辑式(基于规则)语言(**Logical Language**)
    - 基本运算单位是谓词，如Prolog, Yacc.....



# 程序设计语言的分类

---

- 面向对象语言(Object-Oriented Language)
  - 以对象为核心，如Smalltalk、C++、Java、Ada(程序包).....
  - 具有识认性（对象）、类别性（类）、多态性和继承性



## 1.2 程序设计语言的翻译

### ■ 翻译程序(Translator)

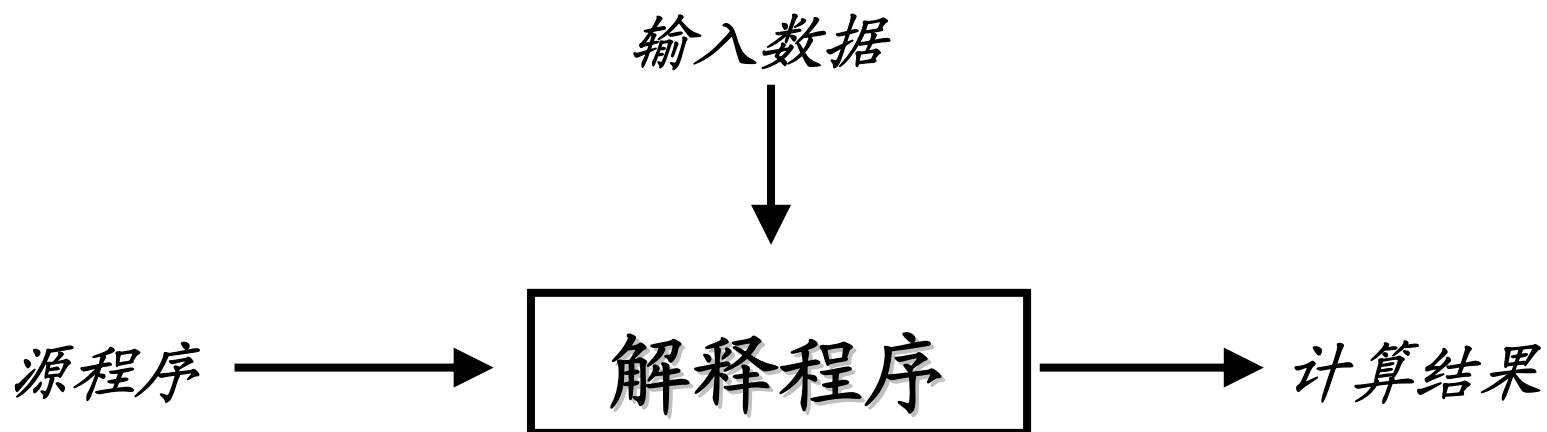
将某一种语言描述的程序(源程序——Source Program)翻译成等价的另一种语言描述的程序(目标程序——Object Program)的程序。

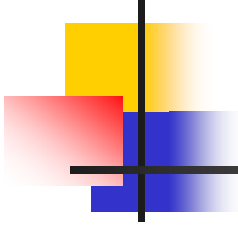




## 1.2 程序设计语言的翻译

- 解释程序(Interpreter)
  - 一边解释一边执行的翻译程序
  - 口译与笔译（单句提交与整篇提交）





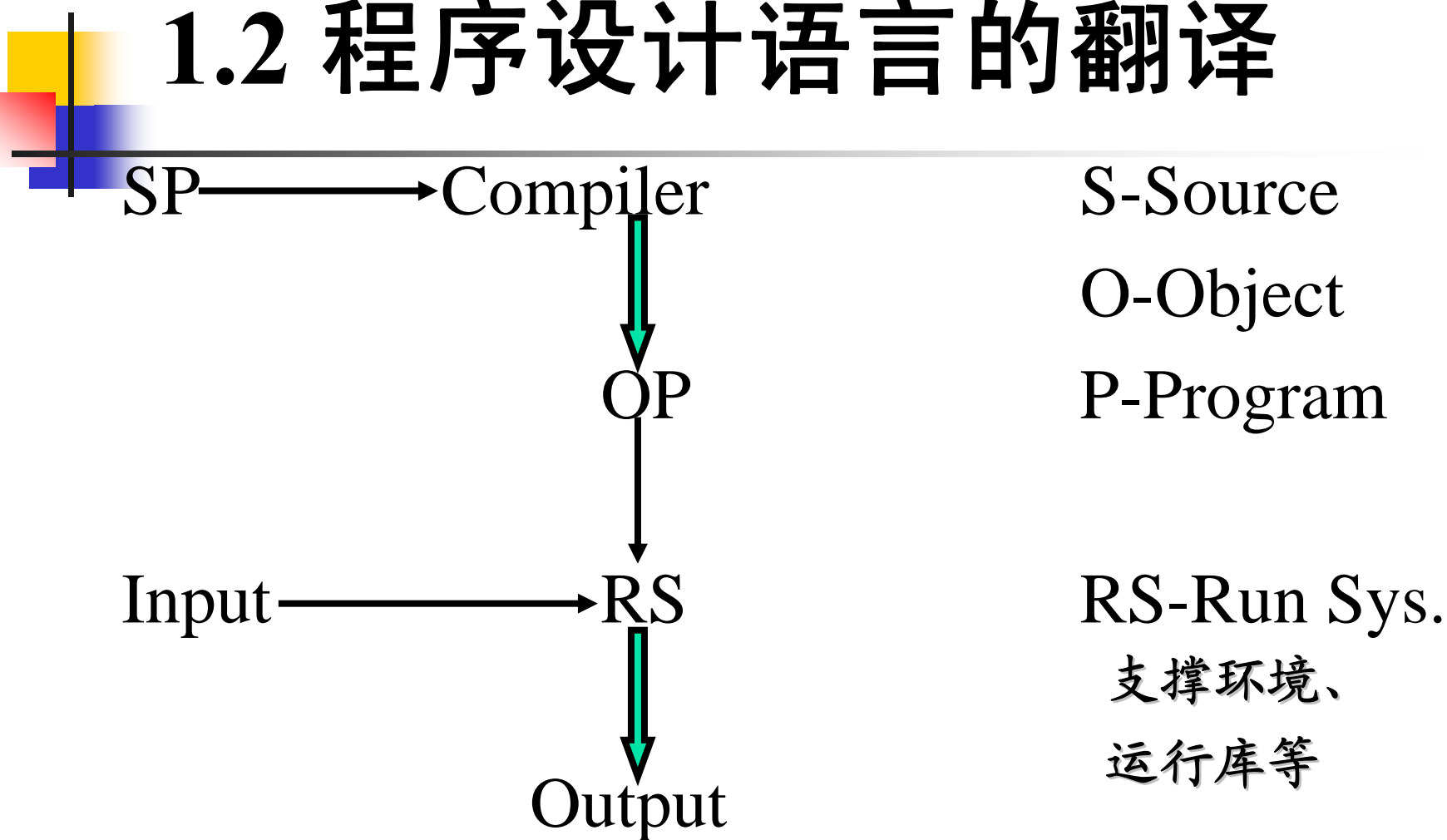
## 1.2 程序设计语言的翻译

### ■ 编译程序(Compiler)

- 将源程序完整地转换成机器语言程序或汇编语言程序，然后再处理、执行的翻译程序
- 高级语言程序 → 汇编/机器语言程序

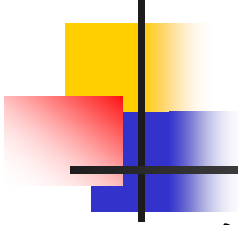


## 1.2 程序设计语言的翻译



### ■ 编译系统(Compiling System)

◆ 编译系统=编译程序+运行系统



# 1.2 程序设计语言的翻译

---

- 其它翻译程序：
  - 汇编程序(Assembler)
  - 交叉汇编程序(Cross Assembler)
  - 反汇编程序 ( Disassembler)
  - 交叉编译程序 ( Cross Compiler)
  - 反编译程序 ( Decompiler )
  - 可变目标编译程序 ( Retargetable Compiler)
  - 并行编译程序 ( Parallelizing Compiler )
  - 诊断编译程序 ( Diagnostic Compiler)
  - 优化编译程序 ( Optimizing Compiler)

# 1.2 程序设计语言的翻译—汇总

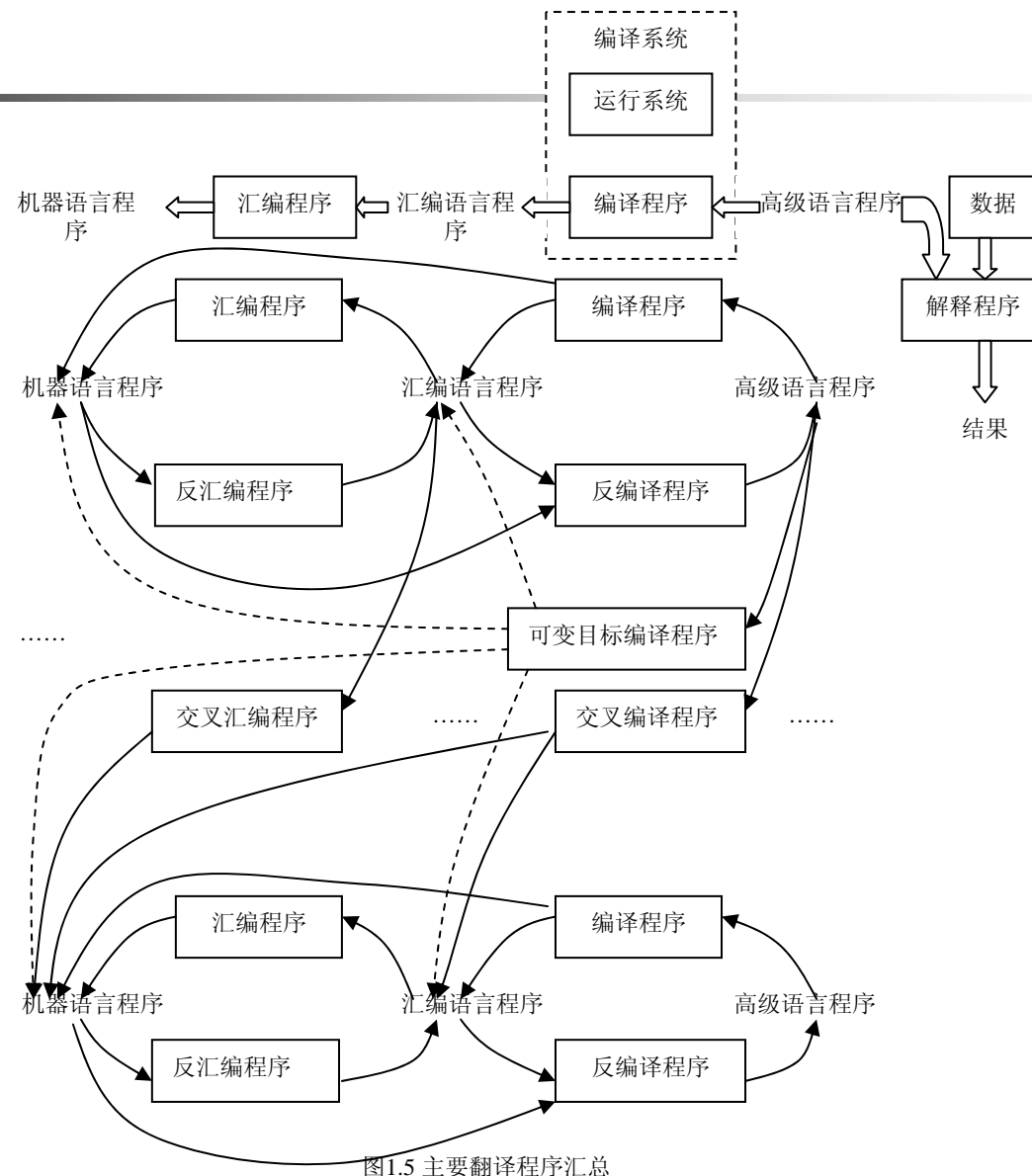
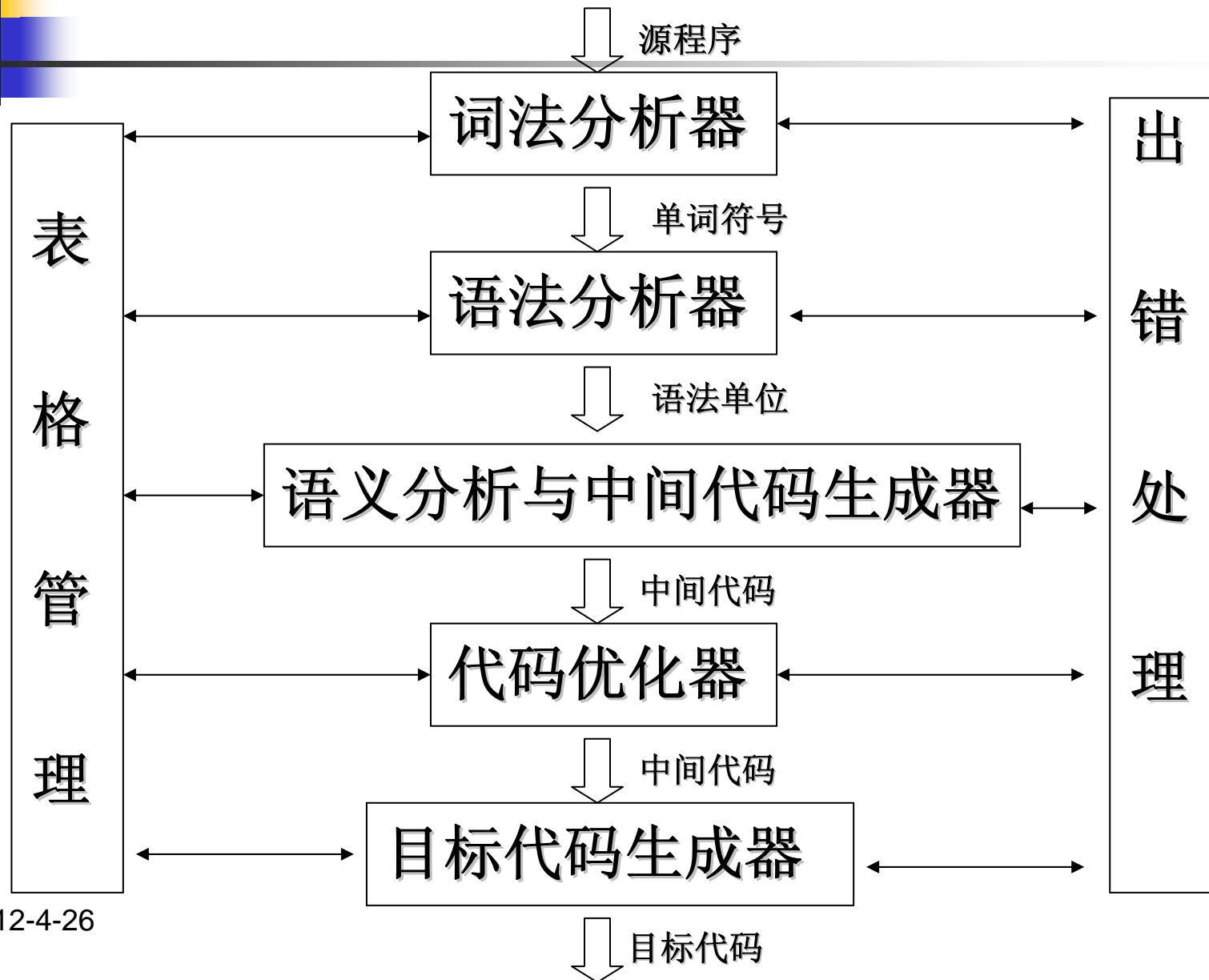


图1.5 主要翻译程序汇总

# 1.3 编译程序总体结构





# 1、词法分析

■ 例:

**sum=(10+20)\*(num+square);**

## 结果

- (标识符, sum)
- (赋值号, =)
- (左括号, (
- (整常数, 10)
- (加号, +)
- (整常数, 20)
- (右括号, ))
- (乘号, \*)
- (左括号, (
- (标识符, num)
- (加号, +)
- (标识符, square)
- (右括号, ))
- (分号, ;)



# 1、词法分析

---

- 词法分析由词法分析器(Lexical Analyzer)完成，词法分析器又称为扫描器(Scanner)
- 词法分析器从左到右扫描组成源程序的字符串，并将其转换成单词(记号—token)串；同时要：查词法错误，进行标识符登记——符号表管理。
- 输入：字符串
- 输出：(种别码，属性值)——序对
  - 属性值——token的机内表示





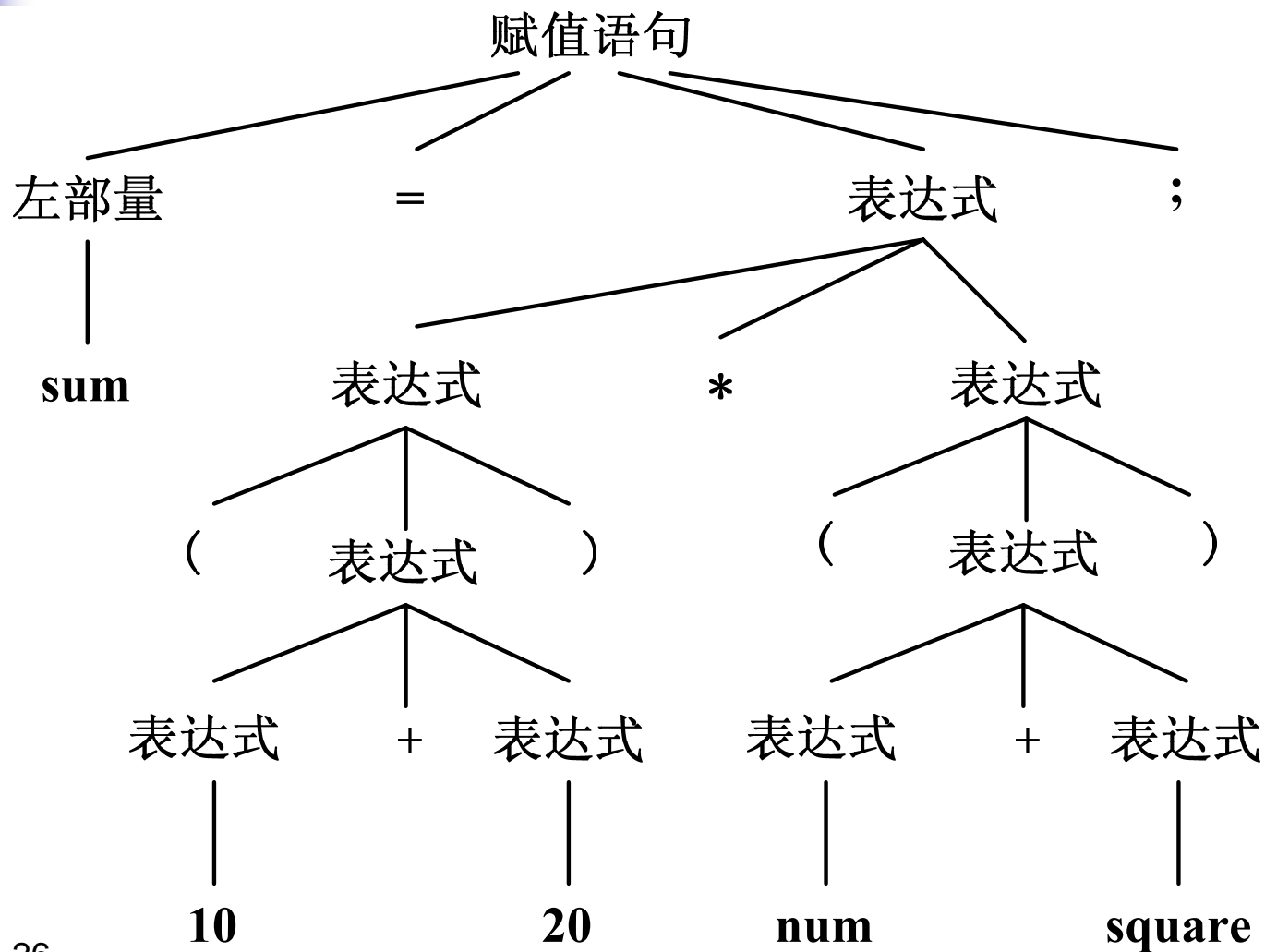
## 2、语法分析

---

- 语法分析由语法分析器(Syntax Analyzer)完成，语法分析器又叫Parser。
- 功能：
  - Parser实现“组词成句”
    - 将词组成各类语法成分：表达式、因子、项，语句，子程序...
  - 构造分析树
  - 指出语法错误
  - 指导翻译
- 输入：token序列
- 输出：语法成分

## 2、语法分析

`sum=(10+20)*(num+square);`





## 3、语义分析

---

- 语义分析(semantic analysis)一般和语法分析同时进行，称为**语法制导翻译**(syntax-directed translation)
- 功能：分析由语法分析器识别出来的语法成分的语义
  - 获取标识符的属性：类型、作用域等
  - 语义检查：运算的合法性、取值范围等
  - 子程序的静态绑定：代码的相对地址
  - 变量的静态绑定：数据的相对地址

## 4、中间代码生成

- 中间代码(intermediate Code)
- 例:  $\text{sum} = (10 + 20) * (\text{num} + \text{square});$

**后缀表示(逆波兰Anti-Polish Notation)**

$\text{sum } 10 \ 20 \ + \ \text{num } \text{square} \ + \ * =$

**前缀表示(波兰Polish Notation)**

$= \text{sum } * + 10 \ 20 + \text{num } \text{square}$

2012-4-26

**四元式表示**

(三地址码)

$(+, 10, 20, T_1)$

$(+, \text{num}, \text{square}, T_2)$

$(*, T_1, T_2, T_3)$

$(=, T_3, , \text{sum})$

**三元式表示**

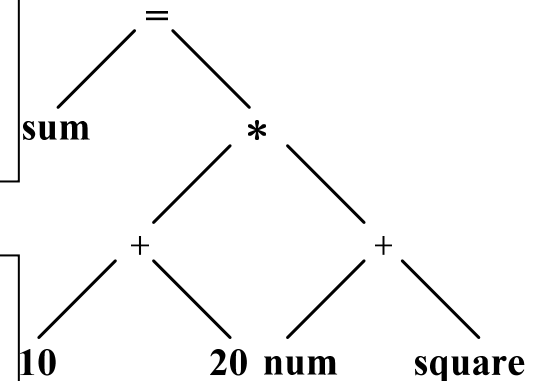
$(+, 10, 20)$

$(+, \text{num}, \text{square})$

$(*, (1), (2))$

$(=, \text{sum}, (3))$

**语法树**



36



# 波兰表示问题——Lukasiewicz 1929年发明

- 中缀表示(Infix notation): $(a + ^{(1)}b) * (-c + ^{(2)}d) + ^{(3)}e / f$
- 波兰表示 (Polish / Prefix / Parenthesis-free / Lukasiewicz notation) ——也就是前缀表示
  - $+ ^{(3)} * + ^{(1)} a b + ^{(2)} @ c d / e f$
- 逆波兰表示(Reverse Polish / Suffix / Postfix notation) ——也就是后缀表示
  - $a b + ^{(1)} c @ d + ^{(2)} * e f / + ^{(3)}$  运算顺序从左向右



## 4、中间代码生成

### ■ 中间代码的特点

- 简单规范
- 与机器无关
- 易于优化与转换

三地址码的另一种表示形式:

- $T_1 = 10 + 20$
- $T_2 = \text{num} + \text{square}$
- $T_3 = T_1 * T_2$
- $\text{sum} = T_3$

其它类型的语句

例: **printf**("hello")

**x := s**            (赋值)

**param x**        (参数)

**call f**          (函数调用)

注释

**s** 是 **hello** 的地址

**f** 是函数 **printf** 的地址



# 5、代码优化

---

- 代码优化(optimization)是指对中间代码进行优化处理，使程序运行能够尽量节省存储空间，更有效地利用机器资源，使得程序的运行速度更快，效率更高。当然这种优化变换必须是等价的。
  - 与机器无关的优化
  - 与机器有关的优化



# 与机器无关的优化

---

## ■ 局部优化

- 常量合并: 常数运算在编译期间完成, 如 $8+9*4$
- 公共子表达式的提取: 在基本块内进行的

## ■ 循环优化

- 强度削减
  - 用较快的操作代替较慢的操作
- 代码外提
  - 将循环不变计算移出循环





# 与机器有关的优化

---

- 寄存器的利用
  - 将常用量放入寄存器，以减少访问内存的次数
- 体系结构
  - **MIMD、SIMD、SPMD、向量机、流水机**
- 存储策略
  - 根据算法访存的要求安排：**Cache、并行存储体系——减少访问冲突**
- 任务划分
  - 按运行的算法及体系结构，划分子任务(**MPMD**)



## 6、目标代码生成

---

- 将中间代码转换成目标机上的机器指令代码或汇编代码
  - 确定源语言的各种语法成分的目标代码结构（机器指令组/汇编语句组）
  - 制定从中间代码到目标代码的翻译策略或算法
- 目标代码的形式
  - 具有绝对地址的机器指令
  - 汇编语言形式的目标程序
  - 模块结构的机器指令（需要链接程序）



# 7、表格管理

---

- 管理各种符号表(常数、标号、变量、过程、结构.....)，查、填（登记、查找）源程序中出现的符号和编译程序生成的符号，为编译的各个阶段提供信息。
  - 辅助语法检查、语义检查
  - 完成静态绑定、管理编译过程
- Hash表、链表等各种表的查、填技术
- “数据结构与算法”课程的应用



# 8、错误处理

---

- 进行各种错误的检查、报告、纠正，以及相应的续编译处理(如：错误的定位与局部化)
  - 词法：拼写.....
  - 语法：语句结构、表达式结构.....
  - 语义：类型不匹配、参数不匹配.....

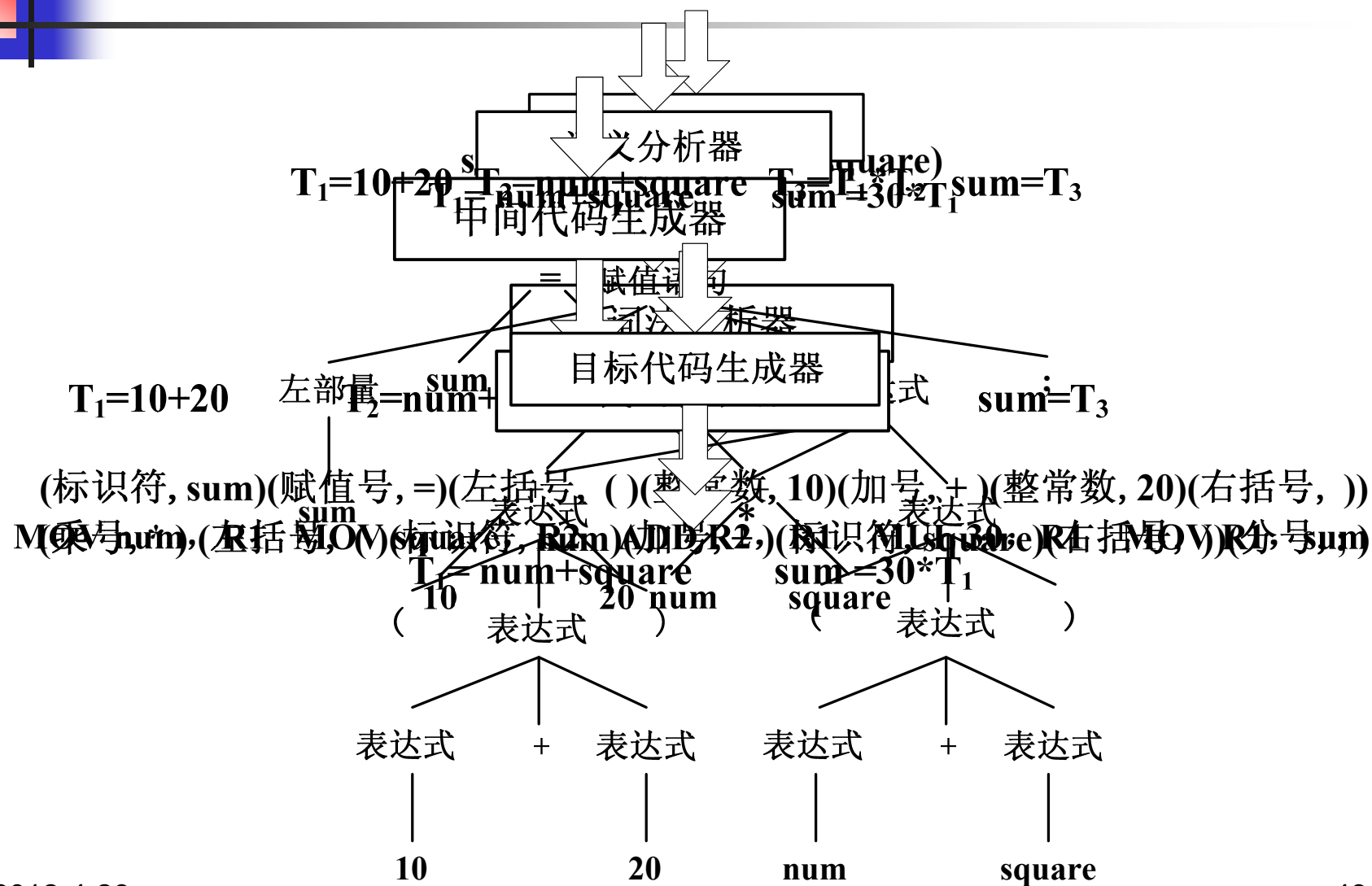


# 模块分类

---

- 分析：词法分析、语法分析、语义分析
- 综合：中间代码生成、代码优化、目标代码生成
- 辅助：符号表管理、出错处理
- 8项功能对应8个模块

# 语句 $\text{sum}=(10+20)*(\text{num}+\text{square});$ 的翻译过程





## 1.4 编译程序的组织

---

- 根据系统资源的状况、运行目标的要求.....等，可以将一个编译程序设计成多遍（Pass）扫描的形式，在每一遍扫描中，完成不同的任务。
  - 如：首遍构造语法树，二遍处理中间表示，增加信息等。
- 遍可以和阶段相对应，也可以和阶段无关
- 单遍代码不太有效



## 1.4 编译程序的组织

---

- 编译程序的设计目标
  - 规模小、速度快、诊断能力强、可靠性高、可移植性好、可扩充性好
  - 目标程序也要规模小、执行速度快
- 编译系统规模较大，因此可移植性很重要
  - 为了提高可移植性，将编译程序划分为前端和后端





## 1.4 编译程序的组织

---

### ■ 前端

- 与源语言有关、与目标机无关的部分
- 词法分析、语法分析、语义分析与中间代码生成、与机器无关的代码优化

### ■ 后端

- 与目标机有关的部分
- 与机器有关的代码优化、目标代码生成



## 1.5 编译程序的生成

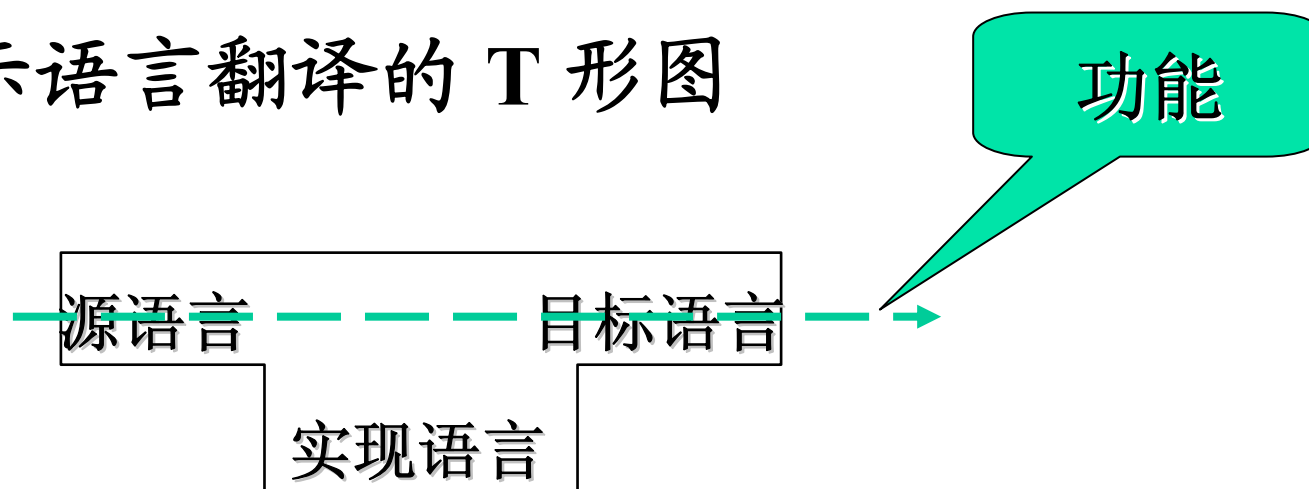
---

### ■ 如何实现编译器？

- 直接用可运行的代码编制——太费力！
- 自举-使用语言提供的功能来编译该语言自身。
- “第一个编译器是怎样被编译的？”

# 1. T形图

## ■ 表示语言翻译的 T 形图

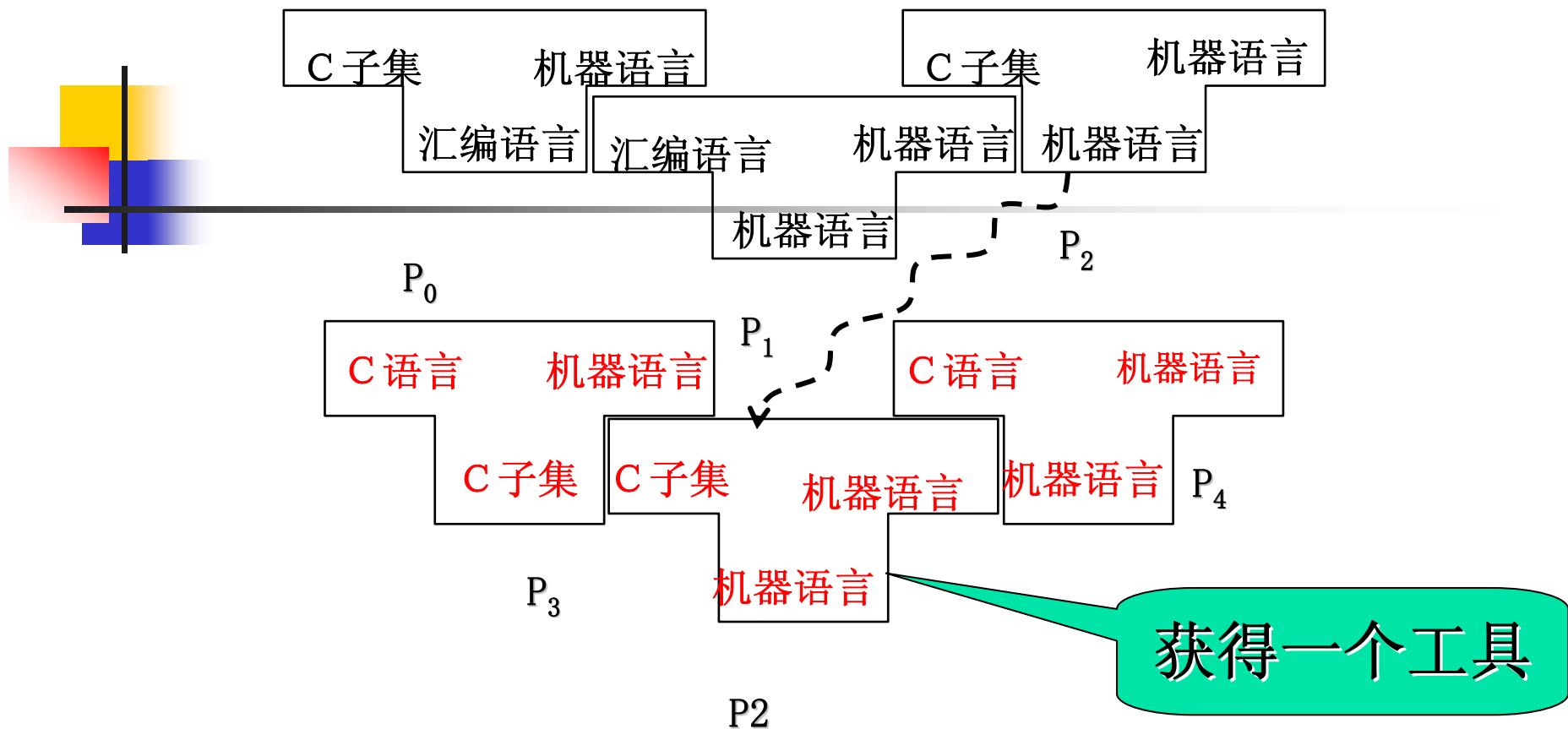




## 2. 自展

---

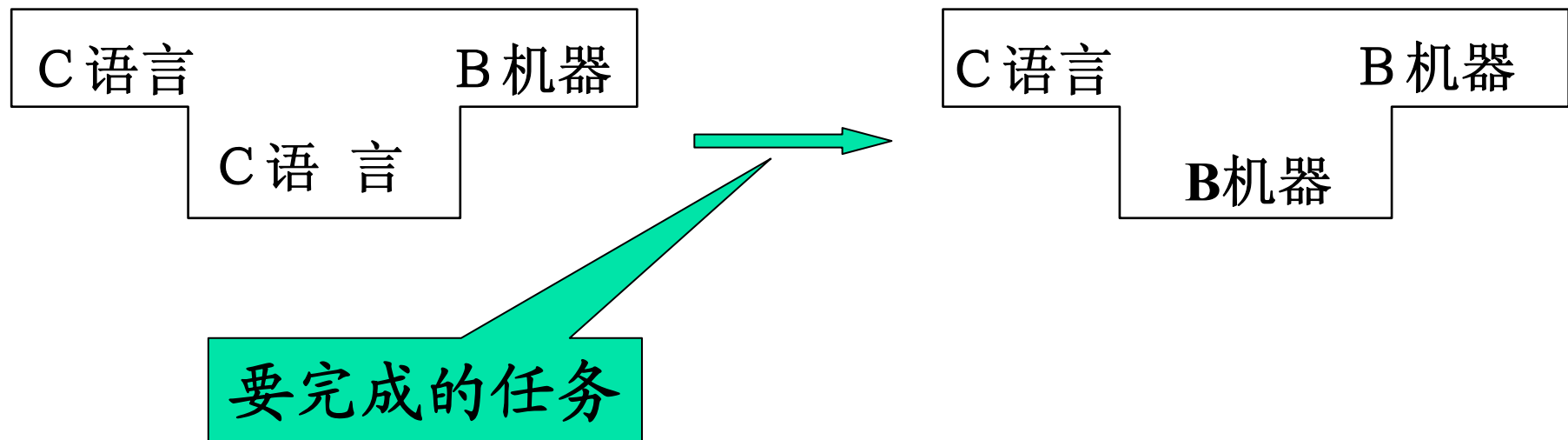
- 问题一：如何直接在一个机器上实现C语言编译器？
- 解决：
  - 用汇编语言实现一个C子集的编译程序( $P_0$ —人)
  - 用汇编程序处理该程序,得到( $P_2$ :可直接运行)
  - 用C子集编制C语言的编译程序( $P_3$ —人)
  - 用 $P_2$ 编译 $P_3$ , 得到 $P_4$



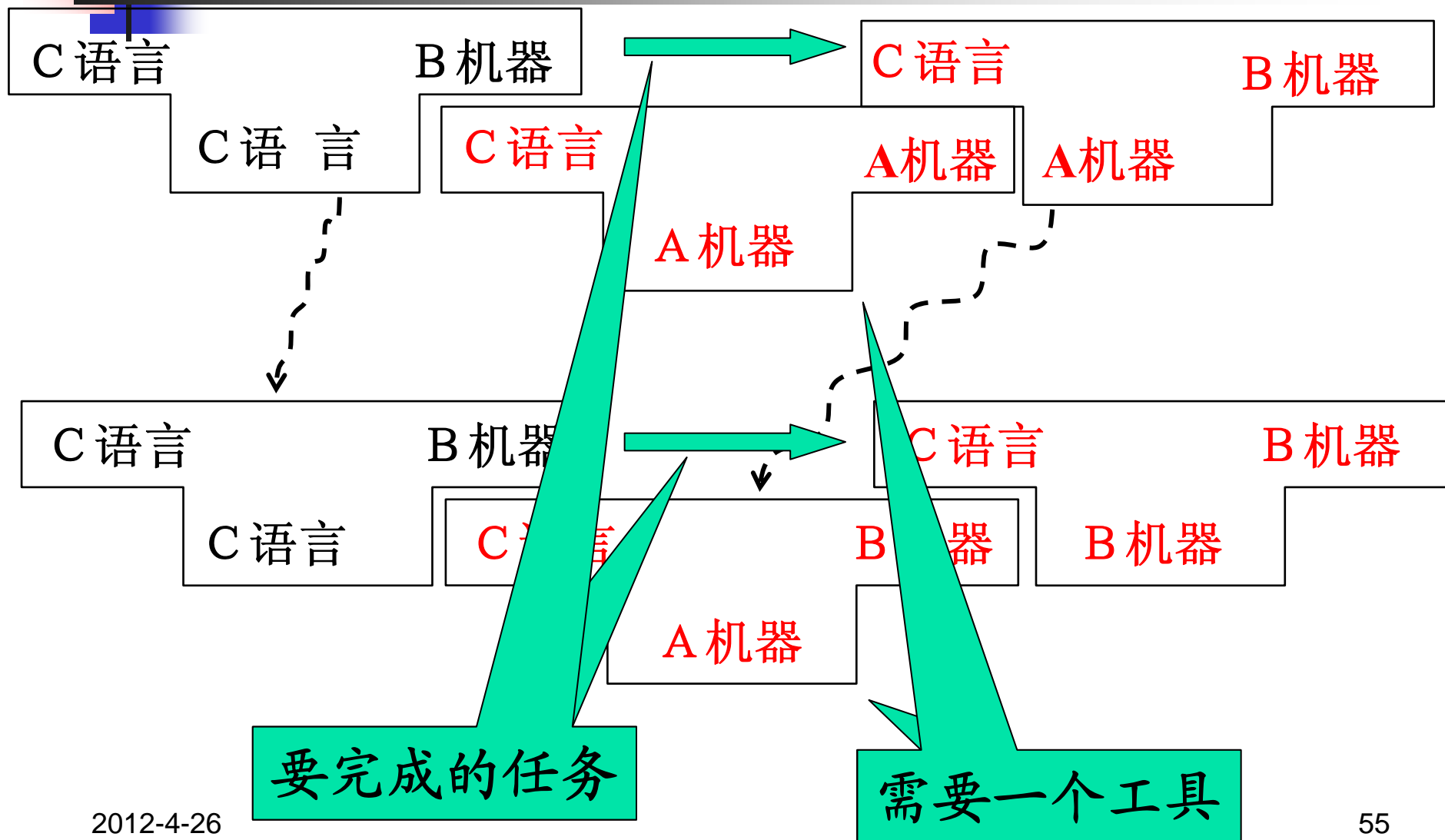
1. 用汇编语言实现一个 C 子集的编译程序( $P_0$ —人)
2. 用汇编程序( $P_1$ )处理该程序,得到( $P_2$ :可直接运行)
3. 用 C 子集编制 C 语言的编译程序( $P_3$ —人)
4. 用  $P_2$  编译  $P_3$ , 得到  $P_4$

### 3.移植

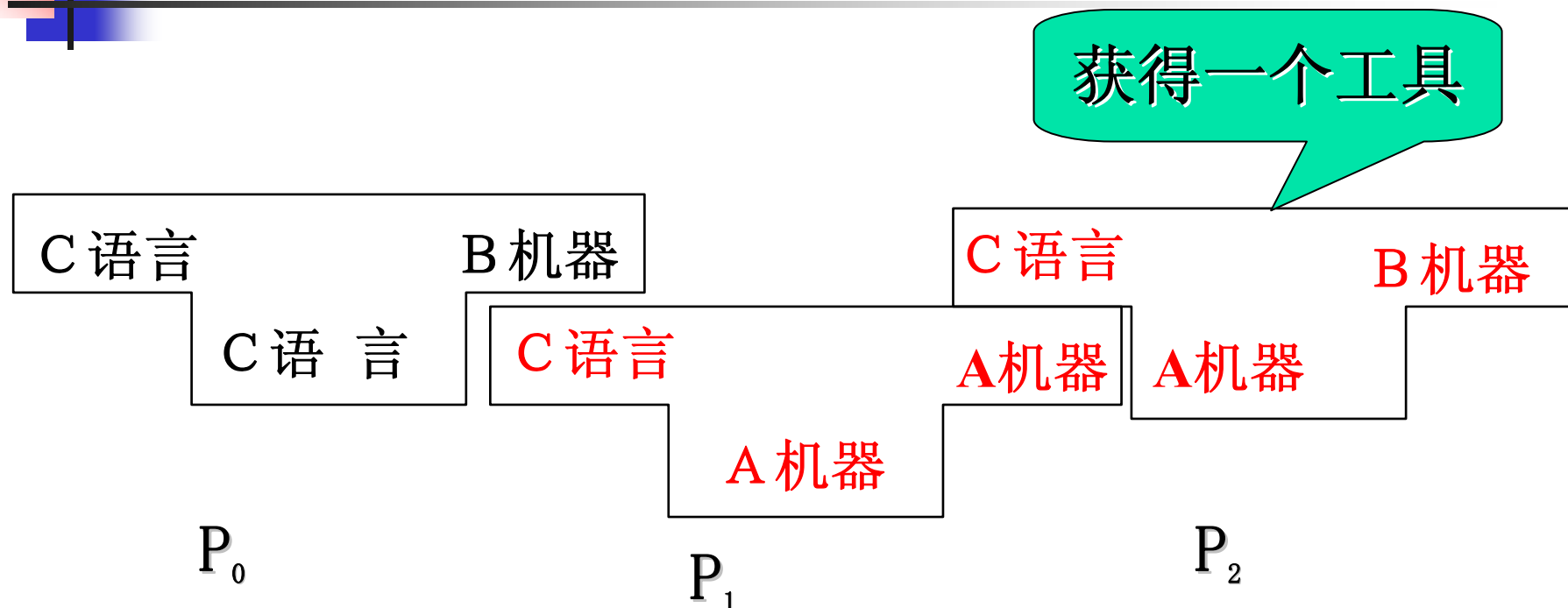
- 问题二：A机上有一个C语言编译器，是否可利用此编译器实现B机上的C语言编译器？
  - 条件：A机有C语言的编译程序
  - 目的：实现B机的C语言的编译



# 1)问题的分析

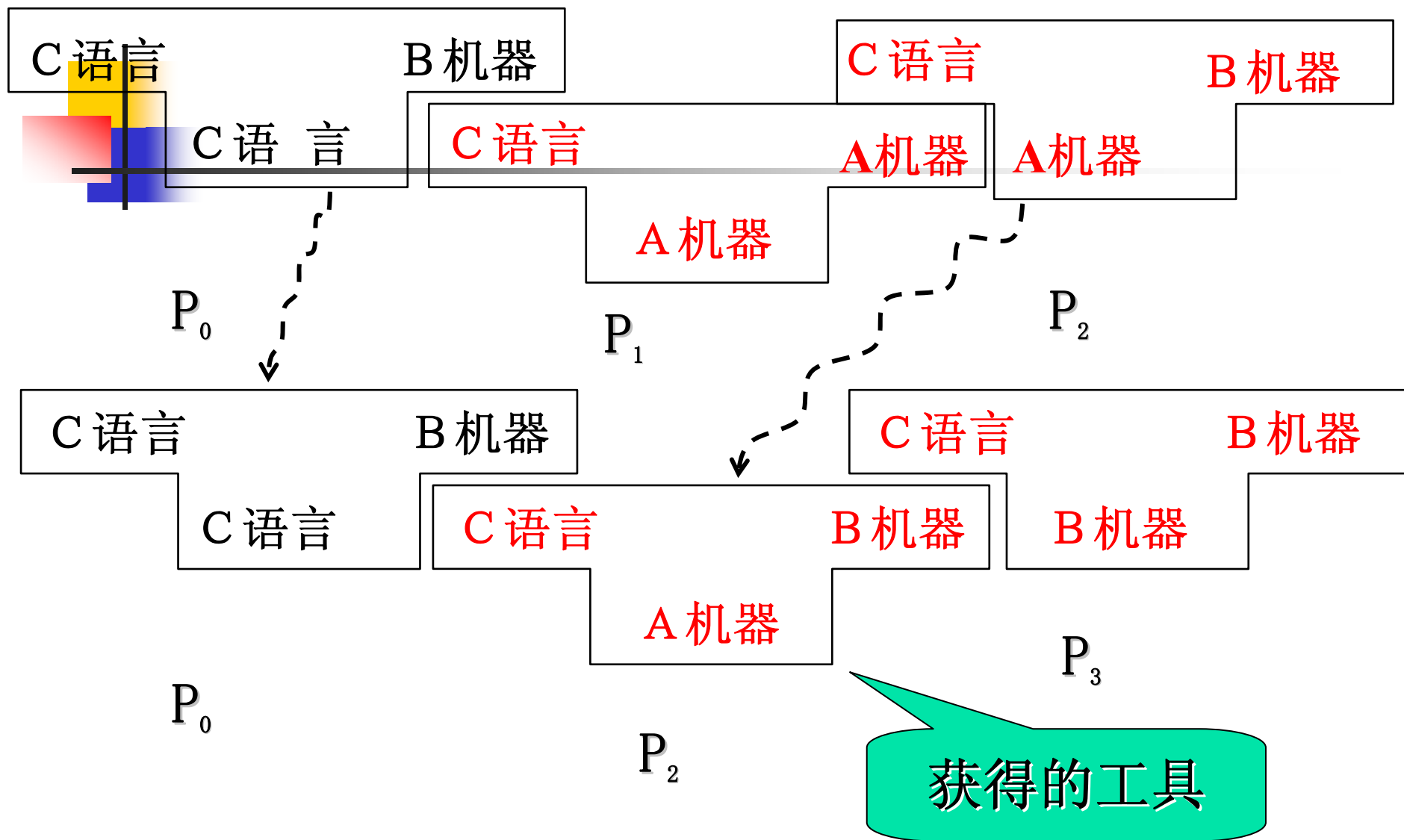


## 2)问题的解决办法



1. (人)用 C 语言编制B机的 C 编译程序 $P_0$  ( $C \rightarrow B$ )
2. (A 机的C编译 $P_1$ ) 编译 $P_0$ , 得到在A机上可运行的 $P_2$  ( $C \rightarrow B$ )

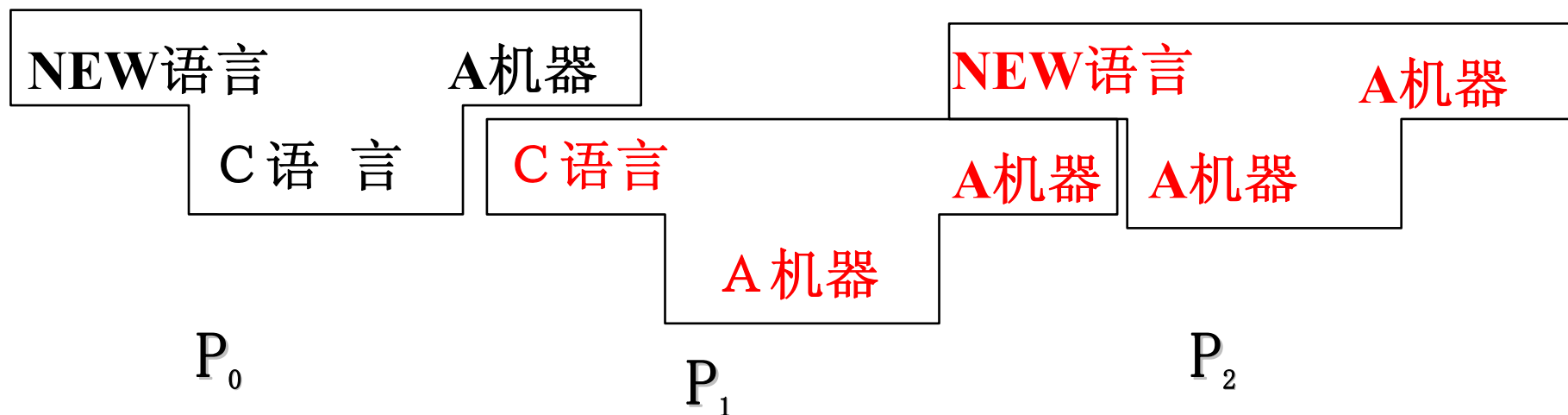




3. (A 机的  $P_2$ ) 编译  $P_0$ , 得到在 B 机上可运行的  $P_3$  ( $C \rightarrow B$ )

## 4.本机编译器的利用

- 问题三： A机上有一个C语言编译器，现要实现一个新语言NEW的编译器？能利用交叉编译技术么？
- 用C编写NEW的编译，并用C编译器编译它





## 5. 编译程序的自动生成

### 1) 词法分析器的自动生成程序



输入:

词法 ( 正规表达式 )

识别动作 ( C 程序段 )

输出:

`yyllex()` 函数



## 2) 语法分析器的自动生成程序

---



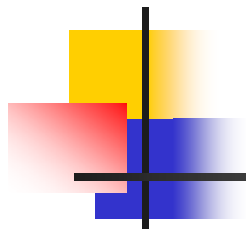
输入:

语法规则 (产生式)

语义动作 (C程序段)

输出:

`yyparse()` 函数



# 例1-1

---

DOS 命令 date 的输出格式

例: 9-3-1993、09-03-1993、9-03-93

语法

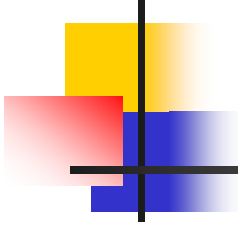
date → month - day - year

词法

month → DIGIT DIGIT | DIGIT

day → DIGIT DIGIT | DIGIT

year → DIGIT DIGIT | DIGIT DIGIT DIGIT DIGIT



## 例1-1 ( 续 )

---

语义

year(年)、month(月)、day(日)

语义约束条件

$$0 < \text{month.value} < 13$$
$$0 < \text{day.value} < 32, 31, 30$$
$$0 < \text{year.value} < 10000$$



## 1.6 本章小结

---

- 编译原理是一门非常好的课
- 程序设计语言及其发展
- 程序设计语言的翻译
- 编译程序的总体结构
- 编译程序的各个阶段
- 编译程序的组织与生成



*School of Computer Science & Technology*  
*Harbin Institute of Technology*



## 第二章 高级语言及其文法

**重点：**文法的定义与分类，CFG的语法树及二义性、  
程序设计语言的定义。

**难点：**程序设计语言的语义定义。







# 第2章 高级语言及其文法

---

**2.1 语言概述**

**2.2 基本定义**

**2.3 文法的定义**

**2.4 文法的分类**

**2.5 CFG的语法树**

**2.6 CFG的二义性**

**2.7 本章小结**



## 2.1 语言概述

---

语言是一定的群体用来进行  
信息交流的工具。



## 2.1 语言概述

---

- 信息交流的基础是什么？
  - 按照共同约定的生成规则和理解规则去生成“句子”和理解“句子”
  - 例：
    - “今节日上课始开译第一编”
    - “今日开始上第一节编译课”



## 2.1 语言概述

---

### ■ 语言的特征

#### ■ 自然语言(Natural Language)

- 是人与人的通讯工具
- 语义(semantics):环境、背景知识、语气、二义性——难以形式化

#### ■ 计算机语言(Computer Language)

- 计算机系统间、人机间通讯工具
- 严格的语法(Grammar)、语义(semantics)——易于形式化: 严格



## 2.1 语言概述

---

- 语言的描述方法——现状
  - 自然语言：自然、方便-非形式化
  - 数学语言（符号）：严格、准确-形式化
  - 形式化描述
    - 高度的抽象，严格的理论基础和方便的计算机表示。

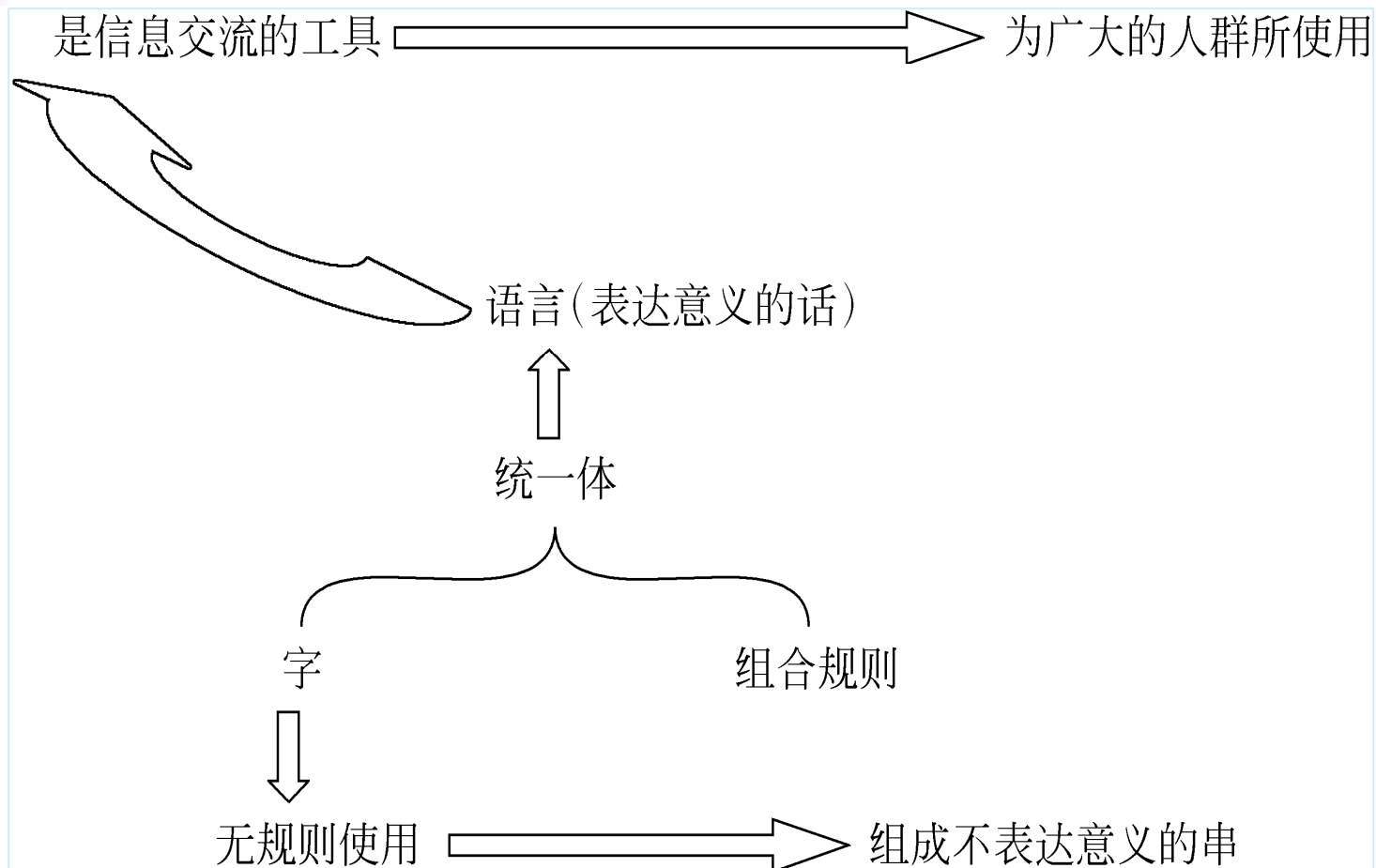


## 2.1 语言概述

---

- 语言——形式化的内容提取
  - 语言(Language): 满足一定条件的句子集合
  - 句子(Sentence): 满足一定规则的单词序列
  - 单词(Token): 满足一定规则的字符(Character)串
- 语言是字和组合字的规则
  - 例（自然语言：第译始二天课今开编上节）
  - 今天开始上第二节编译课

## 2.1 语言概述





## 2.1 语言概述

---

- 程序设计语言——形式化的内容提取
  - 程序设计语言(Programming Language): 组成程序的所有语句的集合。
  - 程序(Program): 满足语法规则的语句序列。
  - 语句(Sentence): 满足语法规则的单词序列。
  - 单词(Token): 满足词法规则的字符串。
- 例: 变量:=表达式
  - if 条件表达式 then 语句
  - while 条件表达式 do 语句
  - call 过程名(参数表)





## 2.1 语言概述

---

- 描述形式——文法
  - 语法——语句
    - 语句的组成规则
    - 描述方法：BNF范式、语法(描述)图
  - 词法——单词
    - 单词的组成规则
    - 描述方法：BNF范式、正规式



# 形式语言与自动机理论的产生与作用

---

- 语言学家Chomsky最初从产生语言的角度研究语言。
  - 1956年，通过抽象，他将语言形式地定义为是由一个字母表中的字母组成的一些串的集合。可以在字母表上按照一定的规则定义一个文法（Grammar），该文法所能产生的所有句子组成的集合就是该文法产生的语言。



# 形式语言与自动机理论的产生与作用

---

- 克林（Kleene）在1951年到1956年间，从识别语言的角度研究语言，给出了语言的另一种描述。
  - 克林是在研究神经细胞中，建立了自动机，他用这种自动机来识别语言：对于按照一定的规则构造的任一个自动机，该自动机就定义了一个语言，这个语言由该自动机所能识别的所有句子组成。



# 形式语言与自动机理论的产生与作用

---

- 1959年，Chomsky通过深入研究，将他本人的研究成果与克林的研究成果结合了起来，不仅确定了文法和自动机分别从生成和识别的角度去表达语言，而且证明了文法与自动机的等价性。



# 形式语言与自动机理论的产生与作用

---

- 20世纪50年代，人们用**巴科斯范式**（**Backus Nour Form** 或 **Backus Normal Form**，简记为 **BNF**）成功地对高级语言**ALGOL-60**进行了描述。实际上，巴科斯范式就是上下文无关文法（**Context Free Grammar**）的一种表示形式。这一成功，使得形式语言在20世纪60年代得到了大力的发展。



# 形式语言与自动机理论的产生与作用

- 形式语言与自动机理论除了在计算机科学领域中的直接应用外，更在计算学科人才的**计算思维的培养**中占有极其重要的地位
- 计算思维能力的培养，主要是由基础理论系列课程实现的，该系列主要由从数学分析开始到形式语言结束的一些数学和抽象程度比较高的内容的课程组成。
  - 它们构成的是一个梯级训练系统。在此系统中，连续数学、离散数学、计算模型等三部分内容要按阶段分开，三个阶段对应与本学科的学生在大学学习期间的思维方式和能力的变化与提高过程的三个步骤。



## 2.2 基本定义

---

- 定义2.1 **字母表** (Alphabet)  $\Sigma$  是一个非空有穷集合，字母表中的元素称为该字母表的一个字母 (Letter)，也叫字符 (Character)
- 例 以下是不同的字母表：
  - (1)  $\{a, b, c, d\}$
  - (2)  $\{a, b, c, \dots, z\}$
  - (3)  $\{0, 1\}$
  - (4) ASCII字母表



## 2.2 基本定义

- 定义2.2 设  $\Sigma_1$ 、 $\Sigma_2$  是两个字母表， $\Sigma_1$  与  $\Sigma_2$  的乘积 (Product) 定义为  $\Sigma_1 \Sigma_2 = \{ab | a \in \Sigma_1, b \in \Sigma_2\}$
- 例:  $\Sigma_1 = \{0,1\}$ ,  $\Sigma_2 = \{a,b\}$ ,  $\Sigma_1 \Sigma_2 = \{0a, 0b, 1a, 1b\}$
- 定义2.3 设  $\Sigma$  是一个字母表， $\Sigma$  的  $n$  次幂 (Power) 递归地定义为：
  - (1)  $\Sigma^0 = \{\varepsilon\}$
  - (2)  $\Sigma^n = \Sigma^{n-1} \Sigma \quad n \geq 1$
- 例:  $\Sigma_1^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$





## 2.2 基本定义

---

- 定义2.4 设  $\Sigma$  是一个字母表,  $\Sigma$  的**正闭包** (Positive Closure)定义为:

- $\Sigma^+ = \Sigma \cup \Sigma^2 \cup \Sigma^3 \cup \Sigma^4 \cup \dots$

- $\Sigma$  的**克林闭包** (Kleene Closure)为:

- $\Sigma^* = \Sigma^0 \cup \Sigma^+$

- $= \Sigma^0 \cup \Sigma \cup \Sigma^2 \cup \Sigma^3 \cup \dots$



## 2.2 基本定义

---

### ■ 例

$$\{0,1\}^+ = \{0, 1, 00, 01, 11, 000, 001, 010, 011, 100, \dots\}$$

$$\{a, b, c, d\}^+ = \{a, b, c, d, aa, ab, ac, ad, ba, bb, bc, bd, \dots, aaa, aab, aac, aad, aba, abb, abc, \dots\}$$



## 2.2 基本定义

---

### ■ 例

$$\{0,1\}^* = \{ \varepsilon, 0, 1, 00, 01, 11, 000, 001, 010, 011, 100, \dots \}$$

$$\{a, b, c, d\}^* = \{ \varepsilon, a, b, c, d, aa, ab, ac, ad, ba, bb, bc, bd, \dots, aaa, aab, aac, aad, aba, abb, abc, \dots \}$$



## 2.2 基本定义

---

- 定义2.5 设  $\Sigma$  是一个字母表,  $\forall x \in \Sigma^*$ ,  $x$  称为字母表  $\Sigma$  上的一个句子 (sentence),  $\varepsilon$  叫做  $\Sigma$  上的一个空句子。
- 定义2.6 设  $\Sigma$  是一个字母表, 对任意的  $x, y \in \Sigma^*$ ,  $a \in \Sigma$ , 句子  $xay$  中的  $a$  叫做  $a$  在该句子中的一个出现 (occurrence)。
- 定义2.7 设  $\Sigma$  是一个字母表,  $\forall x \in \Sigma^*$ , 句子  $x$  中字符出现的总个数叫做该字符串的长度 (length), 记作  $|x|$ 。



## 2.2 基本定义

---

■ 定义2.8 设  $\Sigma$  是一个字母表,  $\forall x, y \in \Sigma^*$ ,  $x, y$  的**并置**(concatenation)是这样一串, 该串是由串  $x$  直接连接串  $y$  所组成的。记作  $xy$ 。并置又叫做**连结**。

■ 对于  $n \geq 0$ , 串  $x$  的  $n$  次幂(power)定义为:

■ (1)  $x^0 = \varepsilon$ ;

■ (2)  $x^n = x^{n-1}x$ 。



## 2.2 基本定义

- 定义2.9 设  $\Sigma$  是一个字母表, 对  $\forall x, y, z \in \Sigma^*$ , 如果  $x=yz$ , 则称  $y$  是  $x$  的**前缀**(prefix), 如果  $z \neq \varepsilon$ , 则称  $y$  是  $x$  的**真前缀**(proper prefix);  $z$  是  $x$  的**后缀**(suffix), 如果  $y \neq \varepsilon$ , 则称  $z$  是  $x$  的**真后缀**(proper suffix)。
- 字母表  $\Sigma=\{a, b\}$  上的句子  $abaabb$  的前缀、后缀、真前缀和真后缀如下:
- 前缀:  $\varepsilon, a, ab, aba, abaa, abaab, abaabb$
- 真前缀:  $\varepsilon, a, ab, aba, abaa, abaab$
- 后缀:  $\varepsilon, b, bb, abb, aabb, baabb, abaabb$
- 真后缀:  $\varepsilon, b, bb, abb, aabb, baabb$



## 2.2 基本定义

■ 定义2.10 设  $\Sigma$  是一个字母表, 对  $\forall x, y, z, w, v \in \Sigma^*$ , 如果  $x=yz$ ,  $w=yv$ , 则称  $y$  是  $x$  和  $w$  的 **公共前缀** (common prefix)。如果  $x$  和  $w$  的任何公共前缀都是  $y$  的前缀, 则称  $y$  是  $x$  和  $w$  的 **最大公共前缀** (maximal common prefix)。如果  $x=zy$ ,  $w=vy$ , 则称  $y$  是  $x$  和  $w$  的 **公共后缀** (common suffix)。如果  $x$  和  $w$  的任何公共后缀都是  $y$  的后缀, 则称  $y$  是  $x$  和  $w$  的 **最大公共后缀** (maximal common suffix)。



## 2.2 基本定义

■ 定义2.11 设  $\Sigma$  是一个字母表, 对  $\forall w, x, y, z \in \Sigma^*$ , 如果  $w=xyz$ , 则称  $y$  是  $w$  的 **子串** (substring)。

■ 定义2.12 设  $\Sigma$  是一个字母表, 对  $\forall t, u, v, w, x, y, z \in \Sigma^*$ , 如果  $t=uyv, w=xyz$ , 则称  $y$  是  $t$  和  $w$  的 **公共子串** (common substring)。如果  $y_1, y_2, \dots, y_n$  是  $t$  和  $w$  的公共子串, 且  $|y_j| = \max\{|y_1|, |y_2|, \dots, |y_n|\}$ , 则称  $y_j$  是  $t$  和  $w$  的 **最大公共子串** (maximal common substring)。





## 2.2 基本定义

---

■ 定义2.13 设  $\Sigma$  是一个字母表,  $\forall L \subseteq \Sigma^*$ ,  $L$  称为字母表  $\Sigma$  上的一个语言 (Language),  $\forall x \in L$ ,  $x$  叫做  $L$  的一个句子。

■ 例: 字母表  $\{0, 1\}$  上的语言

$\{0, 1\}$

$\{00, 11\}$

$\{0, 1, 00, 11\}$

$\{0, 1, 00, 11, 01, 10\}$

$\{00, 11\}^*$

$\{01, 10\}^*$



## 2.2 基本定义

---

■ 2.14 设  $\Sigma_1$ ,  $\Sigma_2$  是字母表,  $L_1 \subseteq \Sigma_1^*$ ,  $L_2 \subseteq \Sigma_2^*$ , 语言  $L_1$  与  $L_2$  的乘积(product)是字母表  $\Sigma_1 \cup \Sigma_2$  上的一个语言, 该语言定义为:

$$L_1 L_2 = \{xy \mid x \in L_1, y \in L_2\}$$

## 2.2 基本定义

定义2.15 设  $\Sigma$  是一个字母表,  $\forall L \in \Sigma^*$ ,  **$L$ 的 $n$ 次幂**(power)是一个语言, 该语言定义为:

- (1) 当  $n=0$  是,  $L^n = \{ \varepsilon \}$ ;
- (2) 当  $n \geq 1$  时,  $L^n = L^{n-1}L$ 。

**$L$ 的正闭包**(positive closure) $L^+$ 是一个语言, 该语言定义为:

$$L^+ = L \cup L^2 \cup L^3 \cup L^4 \cup \dots$$

**$L$ 的克林闭包**(Kleene closure) $L^*$ 是一个语言, 该语言定义为:

$$L^* = L^0 \cup L \cup L^2 \cup L^3 \cup L^4 \cup \dots$$



## 2.3 文法的定义

---

# 如何实现语言结构的 形式化描述?

考虑赋值语句的形式:

左部量 = 右部表达式

$a = a + a$

$b = m[3] + b$

$m[1] = a + m[2]$

# 句子的组成规则

- $\langle \text{赋值语句} \rangle \rightarrow \langle \text{左部量} \rangle = \langle \text{右部表达式} \rangle$
- $\langle \text{左部量} \rangle \rightarrow \langle \text{简单变量} \rangle$
- $\langle \text{左部量} \rangle \rightarrow \langle \text{下标变量} \rangle$
- $\langle \text{简单变量} \rangle \rightarrow a$
- $\langle \text{简单变量} \rangle \rightarrow b$
- $\langle \text{简单变量} \rangle \rightarrow c$
- $\langle \text{下标变量} \rangle \rightarrow m[1]$
- $\langle \text{下标变量} \rangle \rightarrow m[2]$
- $\langle \text{下标变量} \rangle \rightarrow m[3]$
- $\langle \text{右部表达式} \rangle \rightarrow \langle \text{简单变量} \rangle \langle \text{运算符} \rangle \langle \text{简单变量} \rangle$
- $\langle \text{右部表达式} \rangle \rightarrow \langle \text{简单变量} \rangle \langle \text{运算符} \rangle \langle \text{下标变量} \rangle$
- $\langle \text{右部表达式} \rangle \rightarrow \langle \text{下标变量} \rangle \langle \text{运算符} \rangle \langle \text{简单变量} \rangle$
- $\langle \text{右部表达式} \rangle \rightarrow \langle \text{下标变量} \rangle \langle \text{运算符} \rangle \langle \text{下标变量} \rangle$
- $\langle \text{运算符} \rangle \rightarrow +$
- $\langle \text{运算符} \rangle \rightarrow -$



# 定义句子的规则的语法组成

——终结符号集，非终结符号集，语法规则，开始符号

非终结符号集  $V =$

$\{ \langle \text{赋值语句} \rangle, \langle \text{左部量} \rangle, \langle \text{右部表达式} \rangle, \langle \text{简单变量} \rangle, \langle \text{下标变量} \rangle, \langle \text{运算符} \rangle \}$

终结符号集  $T =$

$\{ a, b, c, m[1], m[2], m[3], +, - \}$

语法规则集  $P =$

$\{ \langle \text{赋值语句} \rangle \rightarrow \langle \text{左部量} \rangle = \langle \text{右部表达式} \rangle, \dots \}$

开始符号  $S = \langle \text{赋值语句} \rangle$



# 文法G的形式定义

---

定义2.16 文法 $G$ 为一个四元组:

$$G = (V, T, P, S)$$

- $V$ : 非终结符(Terminal)集
  - 语法变量(成分)——代表某个语言的各种子结构
- $T$ : 终结符(Variable)集
  - 语言的句子中出现的字符,  $V \cap T = \emptyset$
- $S$ : 开始符号(Start Symbol),  $S \in V$ 
  - 代表文法所定义的语言, 至少在产生式左侧出现一次



# 文法G的形式定义

---

- $P$ : 产生式(Product)集合

$\alpha \rightarrow \beta$ , 被称为产生式(定义式), 读作:  $\alpha$  定义为  $\beta$ 。其中  $\alpha \in (V \cup T)^+$ , 且  $\alpha$  中至少有  $V$  中元素的一个出现。  $\beta \in (V \cup T)^*$ 。  $\alpha$  称为产生式  $\alpha \rightarrow \beta$  的**左部**(Left Part),  $\beta$  称为产生式  $\alpha \rightarrow \beta$  的**右部**(Right Part)。

产生式定义各个语法成分的结构(组成规则)



## 例2.9 赋值语句的文法

- $V = \{ \langle \text{赋值语句} \rangle, \langle \text{左部量} \rangle, \langle \text{右部表达式} \rangle, \langle \text{简单变量} \rangle, \langle \text{下标变量} \rangle, \langle \text{运算符} \rangle \}$
- $T = \{ a, b, c, m[1], m[2], m[3], +, - \}$
- $P = \{ \langle \text{赋值语句} \rangle \rightarrow \langle \text{左部量} \rangle = \langle \text{右部表达式} \rangle, \langle \text{左部量} \rangle \rightarrow \langle \text{简单变量} \rangle, \langle \text{左部量} \rangle \rightarrow \langle \text{下标变量} \rangle, \langle \text{简单变量} \rangle \rightarrow a, \langle \text{简单变量} \rangle \rightarrow b, \langle \text{简单变量} \rangle \rightarrow c, \langle \text{下标变量} \rangle \rightarrow m[1], \langle \text{下标变量} \rangle \rightarrow m[2], \langle \text{下标变量} \rangle \rightarrow m[3], \langle \text{右部表达式} \rangle \rightarrow \langle \text{简单变量} \rangle \langle \text{运算符} \rangle \langle \text{简单变量} \rangle, \langle \text{右部表达式} \rangle \rightarrow \langle \text{简单变量} \rangle \langle \text{运算符} \rangle \langle \text{下标变量} \rangle, \langle \text{右部表达式} \rangle \rightarrow \langle \text{下标变量} \rangle \langle \text{运算符} \rangle \langle \text{简单变量} \rangle, \langle \text{右部表达式} \rangle \rightarrow \langle \text{下标变量} \rangle \langle \text{运算符} \rangle \langle \text{下标变量} \rangle, \langle \text{运算符} \rangle \rightarrow +, \langle \text{运算符} \rangle \rightarrow - \}$
- $S = \langle \text{赋值语句} \rangle$



## 例2.9 赋值语句的文法（续）

- 符号化之后:
- $G=(\{A, B, E, C, D, P\}, \{a, b, c, m[1], m[2], m[3], +, -\}, P, A)$ ,  
其中,  $P=\{A \rightarrow B=E, B \rightarrow C, B \rightarrow D, C \rightarrow a, C \rightarrow b, C \rightarrow c, D \rightarrow m[1], D \rightarrow m[2], D \rightarrow m[3], E \rightarrow COC, E \rightarrow COD, E \rightarrow DOC, E \rightarrow DOD, O \rightarrow +, O \rightarrow -\}$



# 产生式的简写

- 对一组有相同左部的产生式

$$\alpha \rightarrow \beta_1, \alpha \rightarrow \beta_2, \dots, \alpha \rightarrow \beta_n$$

可以简单地记为:

$$\alpha \rightarrow \beta_1 | \beta_2 | \dots | \beta_n$$

读作:  $\alpha$  定义为或者  $\beta_1$ , 或者  $\beta_2$ , ..., 或者  $\beta_n$ 。  
并且称它们为  $\alpha$  产生式。  $\beta_1, \beta_2, \dots, \beta_n$  称为  
**候选式(Candidate)**。

- 对一个文法, 只列出该文法的所有产生式, 且所列的第一个产生式的左部是该文法的开始符号。

**问题: 有了定义句子的规则, 如何判定某一句子是否属于某语言?**



# 句子的派生(推导)-从产生语言的角度

<赋值语句>

$\Rightarrow \langle \text{左部量} \rangle = \langle \text{右部表达式} \rangle$

$\Rightarrow \langle \text{简单变量} \rangle = \langle \text{右部表达式} \rangle$

$\Rightarrow a = \langle \text{右部表达式} \rangle$

$\Rightarrow a = \langle \text{简单变量} \rangle \langle \text{运算符} \rangle \langle \text{简单变量} \rangle$

$\Rightarrow a = a \langle \text{运算符} \rangle \langle \text{简单变量} \rangle$

$\Rightarrow a = a + \langle \text{简单变量} \rangle$

$\Rightarrow a = a + a$



# 句子的归约

---从识别语言的角度

$a = a + a$

$\Leftarrow a = a + \langle \text{简单变量} \rangle$

$\Leftarrow a = a \langle \text{运算符} \rangle \langle \text{简单变量} \rangle$

$\Leftarrow a = \langle \text{简单变量} \rangle \langle \text{运算符} \rangle \langle \text{简单变量} \rangle$

$\Leftarrow a = \langle \text{右部表达式} \rangle$

$\Leftarrow \langle \text{简单变量} \rangle = \langle \text{右部表达式} \rangle$

$\Leftarrow \langle \text{左部量} \rangle = \langle \text{右部表达式} \rangle$

$\Leftarrow \langle \text{赋值语句} \rangle$

## ■ 派生与均根据规则

# 基于产生式的变换--推导或归约

- 定义2.17 设 $G=(V, T, P, S)$ 是一个文法, 如果 $\alpha \rightarrow \beta \in P$ ,  $\gamma, \delta \in (V \cup T)^*$ , 则称 $\gamma \alpha \delta$ 在 $G$ 中直接推导出 $\gamma \beta \delta$ , 记作:  
 $\gamma \alpha \delta \xrightarrow{G} \gamma \beta \delta$
- 读作:  $\gamma \alpha \delta$ 在文法 $G$ 中直接推导出 $\gamma \beta \delta$ 。在不特别强调推导的直接性时, “直接推导”可以简称为推导(derivation), 有时我们也称推导为派生。
- 与之相对应, 也可以称 $\gamma \beta \delta$ 在文法 $G$ 中直接归约成 $\gamma \alpha \delta$ 。在不特别强调归约的直接性时, “直接归约”可以简称为归约(reduction)。由于这里实际是将 $\gamma \beta \delta$ 中的 $\beta$ 变成了 $\alpha$ , 而 $\gamma$ 和 $\delta$ 都没有变化, 所以又称将 $\beta$ 归约成 $\alpha$ 。



## (多步)推导

---

- $\alpha_0 \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n$
- 记为  $\alpha_0 \xRightarrow{n} \alpha_n$  (恰用  $n$  步)
- $\alpha_0 \xRightarrow{+} \alpha_n$  (至少一步)
- $\alpha_0 \xRightarrow{*} \alpha_n$  (若干步: 零步或多步)



# 文法 $G$ 产生的语言

- 设 $G=(V, T, P, S)$ 是一个文法, 对于 $\forall A \in V$ , 令
$$L(A)=\{x \mid A \xRightarrow{*} x \ \& \ x \in T^*\}$$

不难看出,  $L(A)$ 就是语法变量 $A$ 所代表的集合。

- 定义2.19 设有文法 $G=(V, T, P, S)$ , 则称
$$L(G)=\{w \mid S \xRightarrow{*} w \ \& \ w \in T^*\}$$

为文法 $G$ 产生的语言(language)。  $\forall w \in L(G)$ ,  $w$ 称为 $G$ 产生的一个句子(sentence)。

显然, 对于任意一个文法 $G$ ,  $G$ 产生的语言 $L(G)$ 就是该文法的开始符号 $S$ 所对应的集合 $L(S)$ 。





# 文法 $G$ 产生的语言(续)

---

$$L(G) = \{x \mid S \Rightarrow^* x \text{ and } x \in T^*\}$$

- 文法 $G$ 可以派生出多少个句子?
- 文法 $G$ 的作用——语言的有穷描述
  - 以有限的规则描述无限的语言现象
- 有限:
  - 产生式集合、终结符集合、非终结符集合
- 无限:
  - 可以导出无穷多个句子( $L$ 也可能是有穷)

# 推导/归约举例

$A \rightarrow B=E$

$B \rightarrow C \mid D$

$C \rightarrow a \mid b \mid c$

$D \rightarrow m[1] \mid m[2] \mid m[3]$

$E \rightarrow COC \mid COD \mid DOC \mid$

$DOD$

$O \rightarrow + \mid -$

$A \Rightarrow B=E$  有产生式  $A \rightarrow B=E$ ，所以可以将  $A$  换成  $B=E$

$\Rightarrow C=E$  有产生式  $B \rightarrow C$ ，所以可以将  $B=E$  中的  $B$  换成  $C$

$\Rightarrow a=E$  有产生式  $C \rightarrow a$ ，所以可以将  $C=E$  中的  $C$  换成  $a$

$\Rightarrow a=COD$  有产生式  $E \rightarrow COD$ ，所以可以将  $a=E$  中的  $E$  换成  $COD$

$\Rightarrow a=bOD$  有产生式  $C \rightarrow b$ ，所以可以将  $a=COD$  中的  $C$  换成  $b$

$\Rightarrow a=b+D$  有产生式  $O \rightarrow +$ ，所以可以将  $a=bOD$  的  $O$  换成  $+$

$\Rightarrow a=b+m[1]$  有产生式  $D \rightarrow m[1]$ ，所以可以将  $a=b+D$  的  $D$  换成  $m[1]$



# 句型与句子

---

- 定义2.20 设文法  $G=(V, \overset{*}{\Rightarrow} T, P, S)$ , 对于  $\forall \alpha \in (V \cup T)^*$ , 如果  $S \Rightarrow^* \alpha$ , 则称  $\alpha$  是  $G$  产生的一个句型(sentential form), 简称为 **句型**
- 对于任意文法  $G=(V, T, P, S)$ ,  $G$  产生的句子和句型的区别在于句子满足  $w \in T^*$ , 而句型满足  $\alpha \in (V \cup T)^*$



# 句型与句子

---

- 句子  $w$  是从  $S$  开始，在  $G$  中可以推导出来的终结符号行，它不含语法变量；
- 句型  $\alpha$  是从  $S$  开始，在  $G$  中可以推导出来的符号行，它可能含有语法变量；
- 句子一定是句型；但句型不一定是句子



## 2.4 文法的分类(Chomsky体系)

---

- 根据语言结构的复杂程度（形式语言）
  - 涉及文法的复杂程度、分析方法的选择
  - 反映文法描述语言的能力
- 如果 $G$ 满足文法定义的要求，则 $G$ 是0型文法（短语结构文法PSG: Phrase Structure Grammar）。
- $L(G)$ 为PSL。



# 1. 上下文有关文法(CSG)

---

- 如果对于 $\forall \alpha \rightarrow \beta \in P$ , 均有 $|\beta| \geq |\alpha|$ 成立 ( $S \rightarrow \varepsilon$ 除外), 则称  $G$  为 **1型文法**
  - 即: 上下文有关文法 (CSG——Context Sensitive Grammar)
- $L(G)$  为 1型/上下文有关/敏感语言(CSL)



## 2. 上下文无关文法(CFG)

---

- 如果对于  $\forall \alpha \rightarrow \beta \in P$ , 均有  $|\beta| \geq |\alpha|$ , 并且  $\alpha \in V$  成立, 则称  $G$  为 **2型文法**
  - 即: 上下文无关文法 (CFG: Context Free Grammar)
- $L(G)$  为 2型/上下文无关语言 (CFL)
  - CFG 能描述程序设计语言的多数语法成分



# 例 标识符的文法

---

$$\blacksquare S \rightarrow L|LT$$

$$T \rightarrow L|N|TL|TN$$

$$L \rightarrow a|b|c|d$$

$$N \rightarrow 0|1|2|3|4|5$$

$$\blacksquare S \rightarrow a|b|c|d$$

$$S \rightarrow aT|bT|cT|dT$$

$$T \rightarrow a|b|c|d|0|1|2|3|4|5$$

$$T \rightarrow aT|bT|cT|dT|0T$$

$$N \rightarrow 1T|2T|3T|4T|5T$$





### 3. 正规文法(RG)

---

- 设 $A, B \in V$ ,  $a \in T$ 或为 $\varepsilon$
  - 右线性(Right Linear)文法:  $A \rightarrow aB$ 或 $A \rightarrow a$
  - 左线性(Left Linear)文法:  $A \rightarrow Ba$ 或 $A \rightarrow a$
- 都是 3 型文法(正规文法 Regular Grammar -RG)
- $L(G)$ 为3型/正规集/正则集/正则语言 (RL)
    - 能描述程序设计语言的多数单词
    - 左、右线性文法不可混用



# 例 非CFL的文法

---

$L = \{a^n b^n c^n | n > 0\}$  的文法

$S \rightarrow aBC | aSBC$

$CB \rightarrow BC$

$aB \rightarrow ab$

$bB \rightarrow bb$

$bC \rightarrow bc$

$cC \rightarrow cc$

- “可以证明”不存在CFG  $G$ , 使  $L(G) = L$



# 非上下文无关的语言结构

---

- 程序设计语言的有些语言结构不能用上下文无关文法描述

- 例2.9

$$L_1 = \{wcw \mid w \in \{a, b\}^+\}$$

$aabcaab$  就是  $L_1$  的一个句子

这个语言是检查程序中标识符的  
声明应先于引用的抽象



# 非上下文无关的语言结构

---

- 例2.10

$$L_2 = \{a^n b^m c^n d^m \mid n, m \geq 0\}$$

- 它是检查过程声明的形参个数和过程引用的参数个数是否一致问题的抽象



# Chomsky体系——总结

---

$G = (V, T, P, S)$  是一个文法,  $\alpha \rightarrow \beta \in P$

\*  $G$  是0型文法,  $L(G)$  是0型语言;

---其能力相当于图灵机

\*  $|\alpha| \leq |\beta|$ :  $G$  是1型文法,  $L(G)$  是1型语言(除  $S \rightarrow \varepsilon$ );

---其识别系统是线性界限自动机

\*  $\alpha \in V$ :  $G$  是2型文法,  $L(G)$  是2型语言;

---其识别系统是不确定的下推自动机

\*  $A \rightarrow aB$  或  $A \rightarrow a$ :  $G$  是右线性文法,  $L(G)$  是3型语言

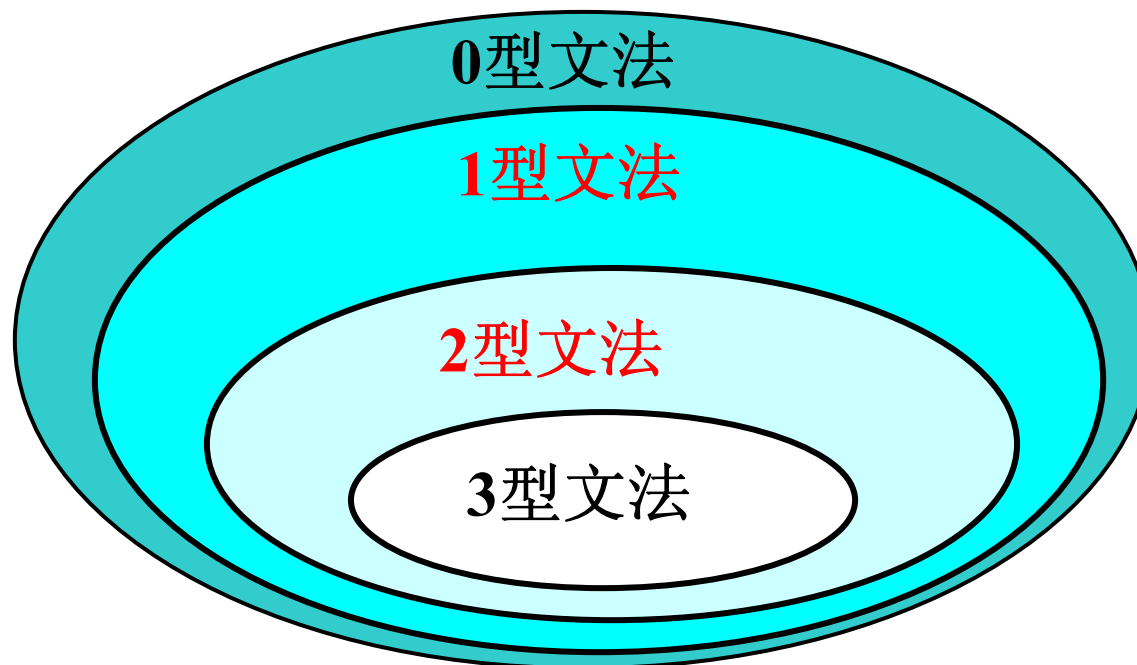
$A \rightarrow Ba$  或  $A \rightarrow a$ :  $G$  是左线性文法,  $L(G)$  是3型语言

---其识别系统是有穷自动机

# 文法的类型

四种文法之间的关系是将产生式作进一步限制而定义的。

四种文法之间的逐级“包含”关系如下：





# B N F 范式——Backus-Naur Form

## Backus-Normal Form

- $\alpha \rightarrow \beta$  表示为  $\alpha ::= \beta$
- 非终结符用“<”和“>”括起来
- 终结符：基本符号集
- 其他
  - $\beta(\alpha_1 | \alpha_2 \dots | \alpha_n) \equiv \beta \alpha_1 | \beta \alpha_2 \dots | \beta \alpha_n$
  - $[\alpha] \equiv \alpha | \varepsilon$
  - .....



# B N F 范式——Backus-Naur Form

## Backus-Normal Form

- 例 简单算术表达式 (只写产生式)
  - $\langle \text{算术表达式} \rangle ::= \langle \text{简单表达式} \rangle + \langle \text{简单表达式} \rangle$
  - $\langle \text{简单表达式} \rangle ::= \langle \text{简单表达式} \rangle * \langle \text{简单表达式} \rangle$
  - $\langle \text{简单表达式} \rangle ::= (\langle \text{简单表达式} \rangle)$
  - $\langle \text{简单表达式} \rangle ::= \text{id}$
- 即:  $\langle \text{算术表达式} \rangle ::= \langle \text{简单表达式} \rangle + \langle \text{简单表达式} \rangle \mid \langle \text{简单表达式} \rangle * \langle \text{简单表达式} \rangle \mid (\langle \text{简单表达式} \rangle) \mid \text{id}$
- 哪些是终结符? 哪些是变量?





## 2.5 CFG的语法树

---

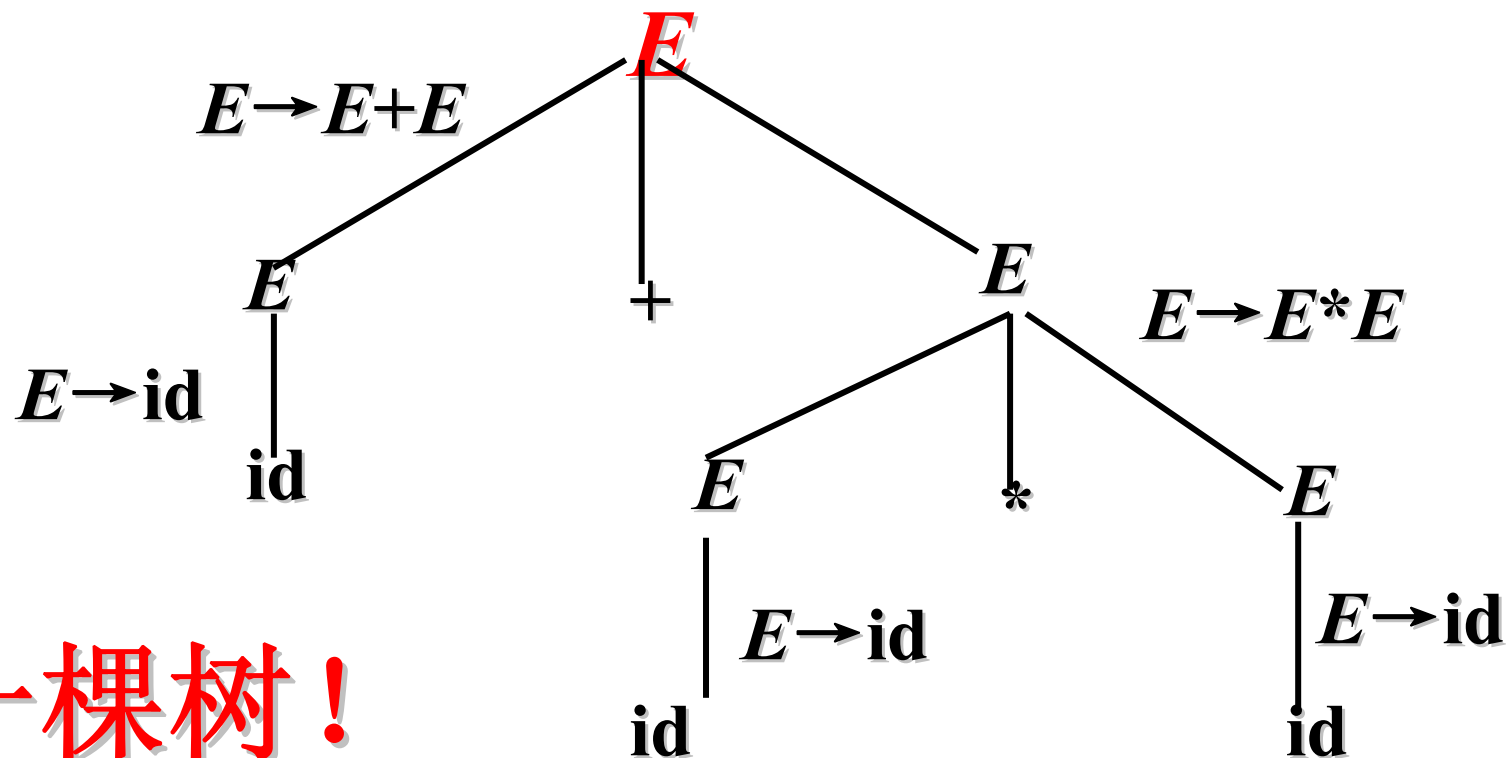
### Parse Tree

- 用树的形式表示句型的生成
  - 树根： 开始符号
  - 中间结点： 非终结符
  - 叶结点： 终结符或者非终结符
- 每个推导对应一个中间结点及其儿子——一个二级子树-直接短语
- 又称为**分析树**(parse tree)、**推导树**(derivation tree)、**派生树**(derivation tree)

# 例子结构的表示

(文法  $E \rightarrow E+E | E * E | (E) | id$  )

$E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+E * E \Rightarrow id+id * E \Rightarrow id+id * id$



一棵树！

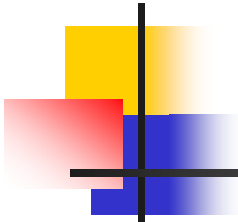
# 短语(Phrase)

■ 定义2.27 设有CFG  $G=(V, T, P, S)$ ,  $\exists \alpha$ ,  $\gamma, \beta \in (V \cup T)^*$ ,  $S \Rightarrow \gamma A \beta$ ,  $A \Rightarrow \alpha$ , 则称  $\alpha$  是句型  $\gamma \alpha \beta$  的相对于变量  $A$  的**短语**(phrase);

如果此时有  $A \Rightarrow \alpha$ , 则称  $\alpha$  是句型  $\gamma \alpha \beta$  的相对于变量  $A$  的**直接短语**(immediate phrase)

在无意义冲突时, 简称为句型  $\gamma \alpha \beta$  的直接短语。直接短语也叫做**简单短语**(simple phrase)。

■ 定义2.28 设有CFG  $G=(V, T, P, S)$ ,  $G$  的句型的最左直接短语叫做**句柄**(handle)。



## 例：(直接)短语

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

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

$$F \rightarrow (E) \mid \text{id}$$

$$E \Rightarrow E+T$$

$$\Rightarrow T+T$$

$$\Rightarrow F+T$$

$$\Rightarrow (E)+T$$

$$\Rightarrow (E+T)+T$$

$$\Rightarrow (E+T)+T$$

$$\Rightarrow (T+T)+T$$

$$\Rightarrow (F+T)+T$$

$$\Rightarrow (\text{id}+T)+T$$

$$\Rightarrow (a+T*F)+T$$

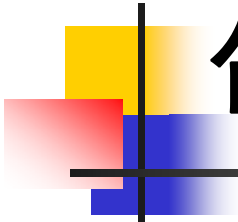
$$\Rightarrow (a+F*F)+T$$

$$\Rightarrow (a+b*F)+T$$

$$\Rightarrow (a+b*c)+T$$

$$\Rightarrow (a+b*c)+F$$

$$\Rightarrow (a+b*c)+d$$



# 句柄(Handle): 最左直接短语

$$\blacklozenge T \rightarrow T * F | F$$

$$\blacklozenge E \rightarrow E + T | T$$

$$\blacklozenge F \rightarrow ( E ) | \text{id}$$

$$E \Rightarrow E + T$$

$$\Rightarrow T + T$$

$$\Rightarrow F + T$$

$$\Rightarrow (E) + T$$

$$\Rightarrow (E + T) + T$$

$$\Rightarrow (E + T) + T$$

$$\Rightarrow (T + T) + T$$

$$\Rightarrow (F + T) + T$$

$$\Rightarrow (\text{id} + T) + T$$

$$\Rightarrow (a + T * F) + T$$

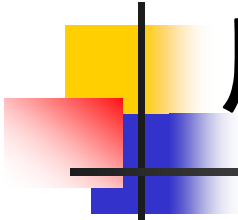
$$\Rightarrow (a + F * F) + T$$

$$\Rightarrow (a + b * F) + T$$

$$\Rightarrow (a + b * c) + T$$

$$\Rightarrow (a + b * c) + F$$

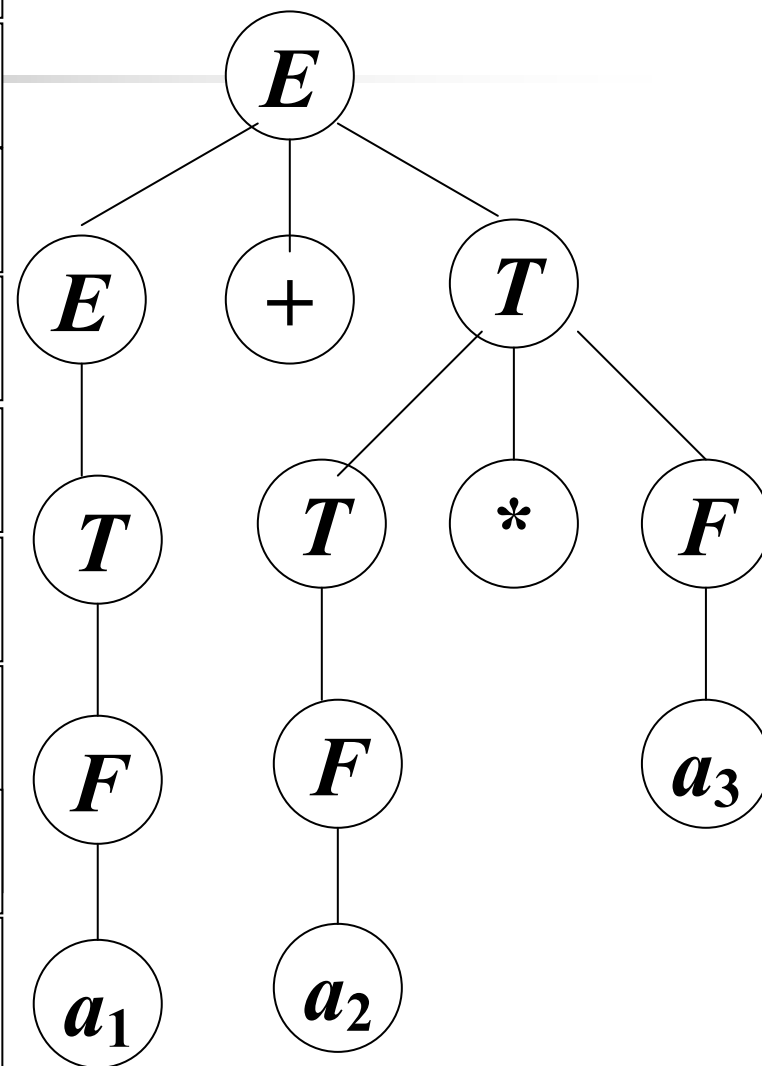
$$\Rightarrow (a + b * c) + d$$



# 用子树解释短语，直接短语，句柄

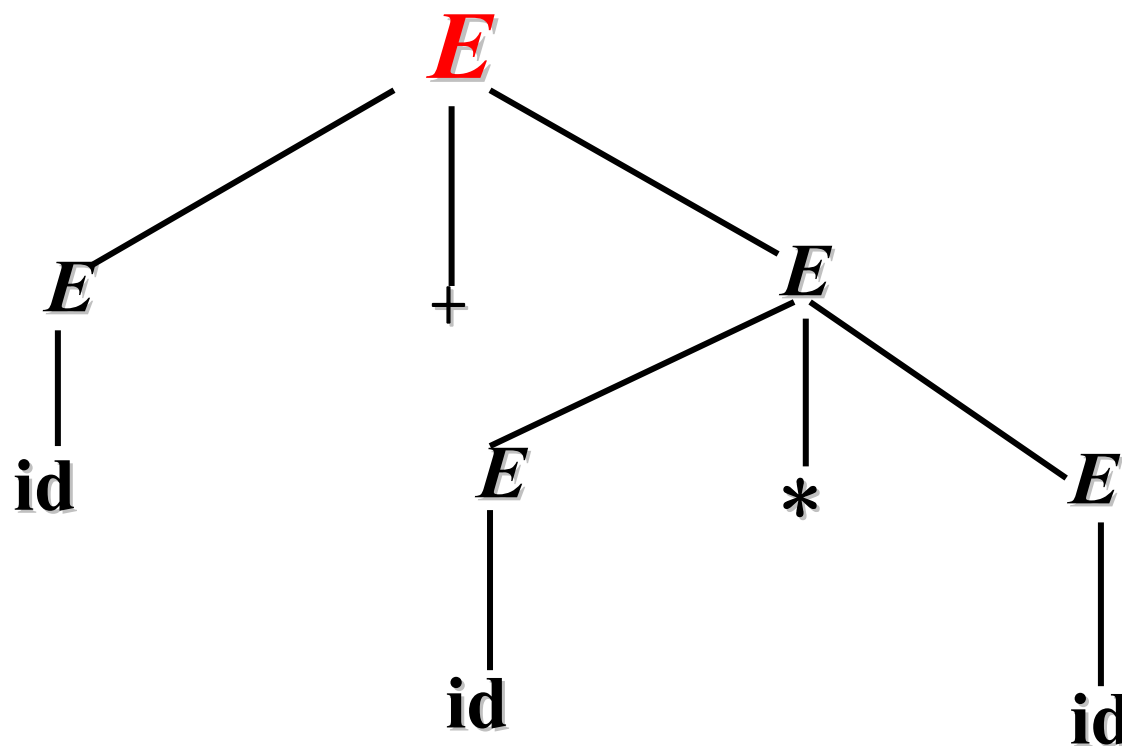
- **短语**：一棵子树的所有叶子自左至右排列起来形成一个相对于子树根的短语。
- **直接短语**：仅有父子两代的一棵子树，它的所有叶子自左至右排列起来所形成的符号串。
- **句柄**：一个句型的分析树中最左那棵只有父子两代的子树的所有叶子的自左至右排列。
- 例如，对表达式文法  $G[E]$  和句子  $a_1 + a_2 * a_3$ ，挑选出推导过程中产生的句型中的短语，直接短语，句柄

$E$	短语
$\Rightarrow \underline{E} + T$	$\underline{E} + T$
$\Rightarrow \underline{T} + T$	$\underline{T}, T + T$
$\Rightarrow \underline{F} + T$	$\underline{F}, F + T$
$\Rightarrow \underline{a_1} + T$	$\underline{a_1}, a_1 + T$
$\Rightarrow \underline{a_1} + T * F$	$\underline{a_1}, \underline{T * F}, a_1 + T * F$
$\Rightarrow \underline{a_1} + F *$	$\underline{a_1}, \underline{F}, F * F,$
$\Rightarrow \underline{a_1} + a_2 * F$	$\underline{a_1}, \underline{a_2}, a_1 + a_2 * F, a_2 * F$
$\Rightarrow \underline{a_1} + a_2 * a_3$	$\underline{a_1}, \underline{a_2}, \underline{a_3}, a_2 * a_3$ $a_1 + a_2 * a_3$



# 例 短语与分析树

(文法  $E \rightarrow E+E|E*E|(E)|id$  )



一棵子树的叶子！



# id+id\*id的不同推导 $E \rightarrow E+E | E * E | (E) | id$

$E \Rightarrow E * E$   
 $\Rightarrow E + E * E$   
 $\Rightarrow E + id * E$   
 $\Rightarrow id + id * E$   
 $\Rightarrow id + id * id$

不做限制

句型 (sentential Form)

(归约)

$E \Rightarrow^* id + id * id$

2012-4-26

$E \Rightarrow E + E$   
 $\Rightarrow id + E$   
 $\Rightarrow id + E * E$   
 $\Rightarrow id + id * E$   
 $\Rightarrow id + id * id$

施于最左变量

左句型 (left-~)

(最右归约)

$E \Rightarrow^5 id + id * id$

$E \Rightarrow E + E$   
 $\Rightarrow E + E * E$   
 $\Rightarrow E + E * id$   
 $\Rightarrow E + id * id$   
 $\Rightarrow id + id * id$

施于最右变量

右句型/规范句型

(canonical ~)

(最左/规范归约)

$E \Rightarrow^+ id + id * id$



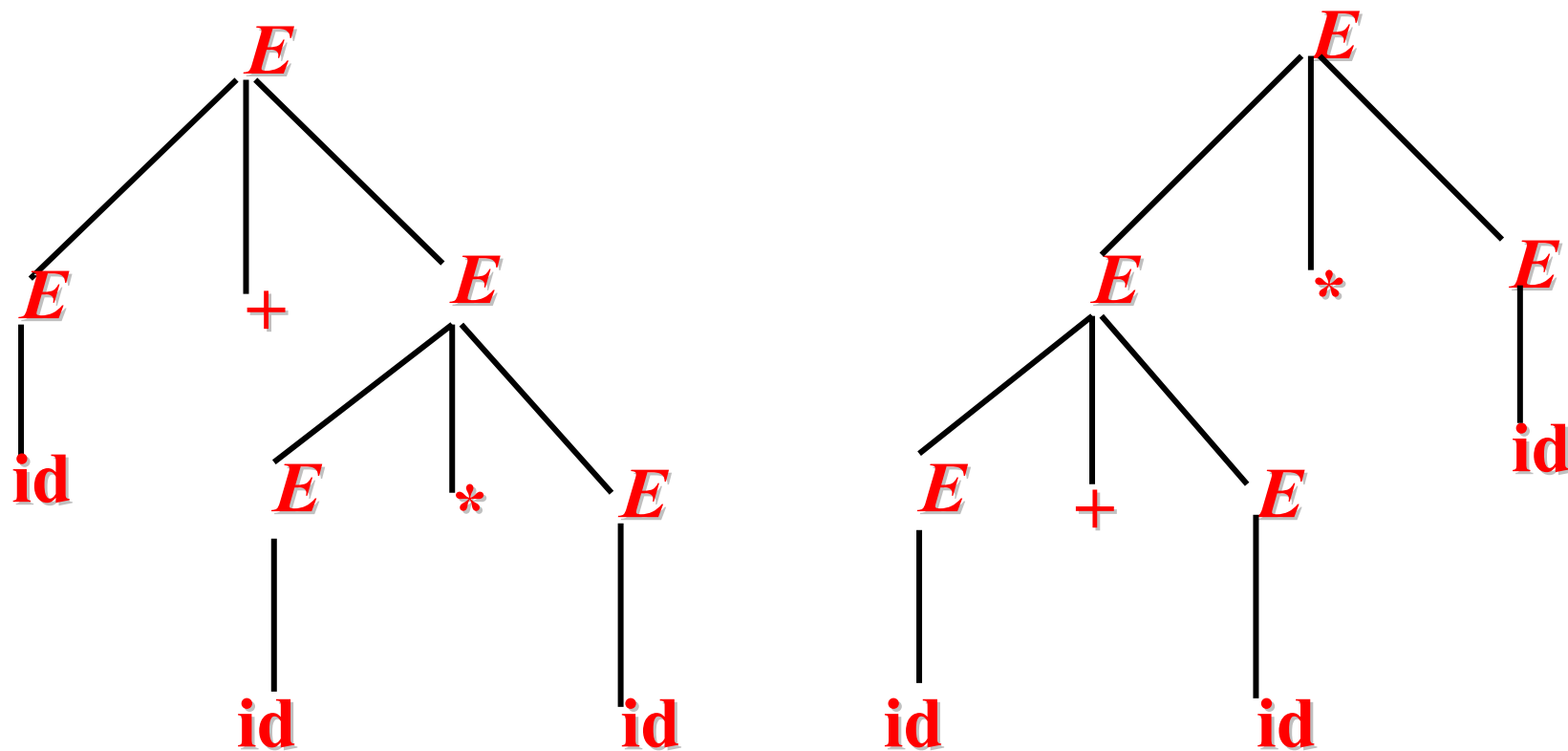
# 最左推导与最右推导

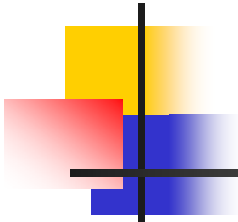
---

- **最左推导(Left-most Derivation)**
  - 每次推导都施加在句型的最左边的语法变量上。——与最右归约对应
- **最右推导(Right-most Derivation)**
  - 每次推导都施加在句型的最右边的语法变量上。——与最左归约（规范规约）对应的规范(Canonical)句型

## 2.6 CFG的二义性

- 对同一句子存在两棵语法分析树
  - 在理论上不可判定





# 文法的二义性

1. 描述一个句子的文法不是唯一的;
2. 对于一个句子的分析应是唯一的。

考虑表达式下面的文法  $G[E]$ , 其产生式如下:

$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

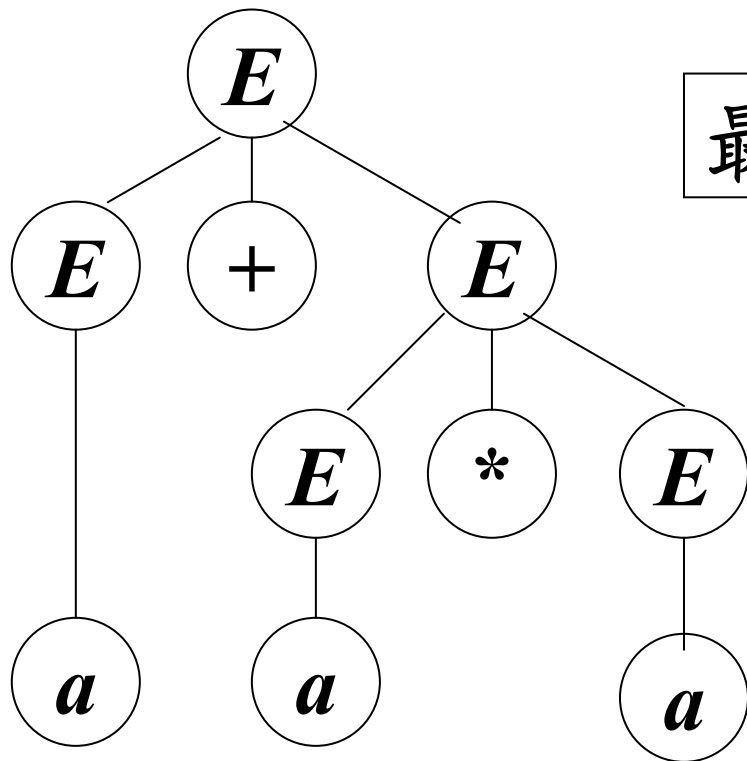
对于句子  $a + a * a$ , 有如下两个最左推导:

$$E \Rightarrow E + E \Rightarrow a + E \Rightarrow a + E * E \Rightarrow a + a * E \Rightarrow a + a * a$$

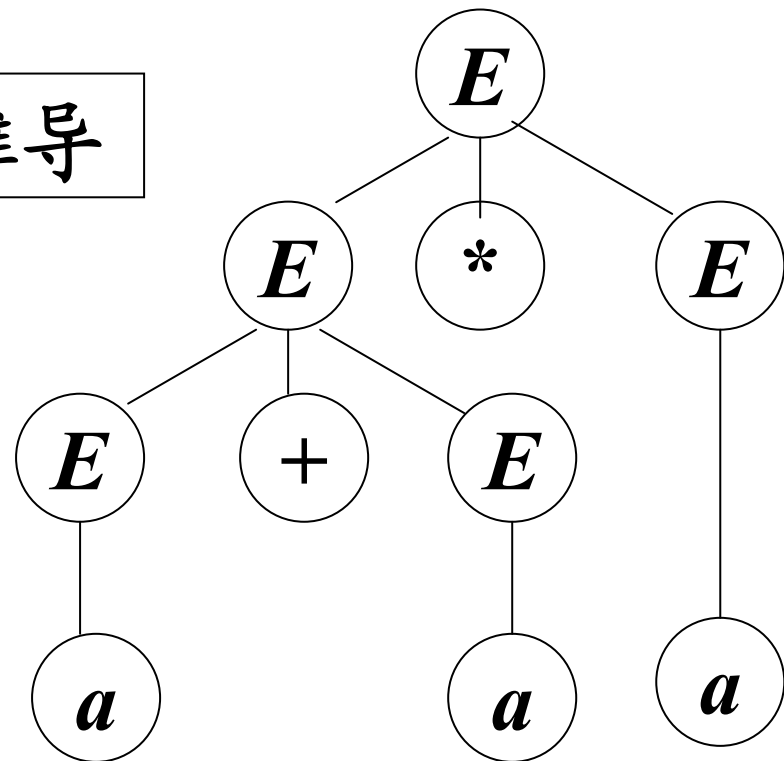
$$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow a + E * E \Rightarrow a + a * E \Rightarrow a + a * a$$

$$\begin{aligned}
 E &\Rightarrow E + E \Rightarrow a + E \\
 &\Rightarrow a + E * E \Rightarrow a + a * E \\
 &\Rightarrow a + a * a
 \end{aligned}$$

$$\begin{aligned}
 E &\Rightarrow E * E \Rightarrow E + E * E \\
 &\Rightarrow a + E * E \Rightarrow a + a * E \\
 &\Rightarrow a + a * a
 \end{aligned}$$



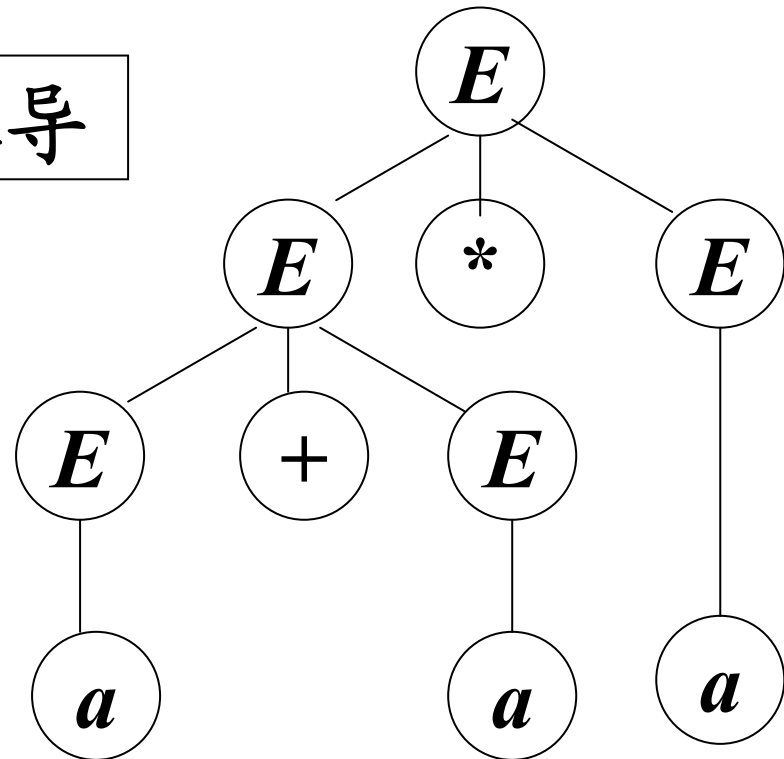
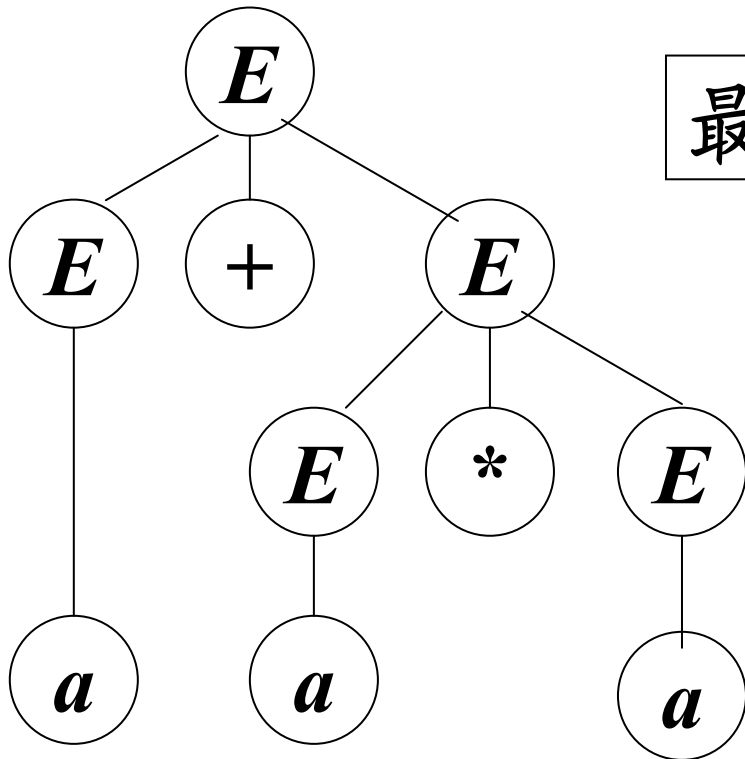
最左推导



$$\begin{aligned}
 E &\Rightarrow E + E \Rightarrow E + E * E \\
 &\Rightarrow E + E * a \Rightarrow E + a * a \\
 &\Rightarrow a + a * a
 \end{aligned}$$

$$\begin{aligned}
 E &\Rightarrow E * E \Rightarrow E * a \\
 &\Rightarrow E + E * a \Rightarrow E + a * a \\
 &\Rightarrow a + a * a
 \end{aligned}$$

最右推导





# 二义性 (ambiguity)的定义

- 如果一个文法的句子存在两棵分析树,那么,该句子是二义性的。如果一个文法包含二义性的句子,则称这个文法是二义性的;否则,该文法是无二义性的。几点说明:
- 1. 一般来说,程序语言存在无二义性文法,对于表达式来说,文法(2.1)是二义性的。对于条件语句,经常使用二义性文法描述它:
- $$S \rightarrow \text{if } \textit{expr} \text{ then } S$$

$$\quad \quad \quad \text{if } \textit{expr} \text{ then } S \text{ else } S$$
$$\quad \quad \quad \text{other}$$
- 二义性的句子:  $\text{if } e_1 \text{ then if } e_2 \text{ then } s_1 \text{ else } s_2$



## 描述if语句的无二义性文法的产生式

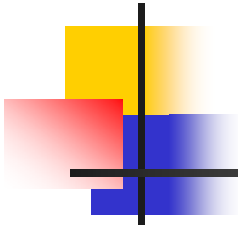
下面是

$$\begin{aligned} S &\rightarrow matched\_s \mid unmatched\_s \\ matched\_s &\rightarrow \text{if } expr \text{ then } matched\_s \\ &\quad \text{else } matched\_s \mid \text{other} \\ unmatched\_s &\rightarrow \text{if } expr \text{ then } S \\ &\quad \mid \text{if } expr \text{ then } matched\_s \\ &\quad \text{else } unmatched\_s \end{aligned}$$

它显然比较复杂，因此：

2. 在能驾驭的情况下，使用二义性文法。





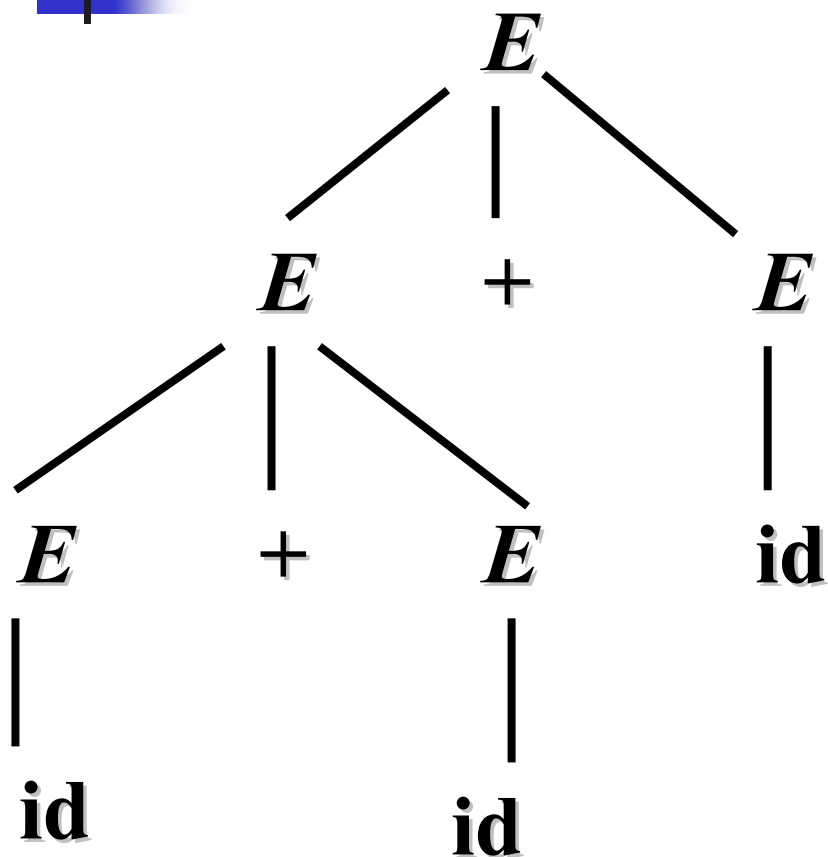
3. 对于任意一个上下文无关文法，不存在一个算法，判定它是无二义性的；但能给出一组充分条件，满足这组充分条件的文法是无二义性的。

4. 存在先天二义性语言。例如，

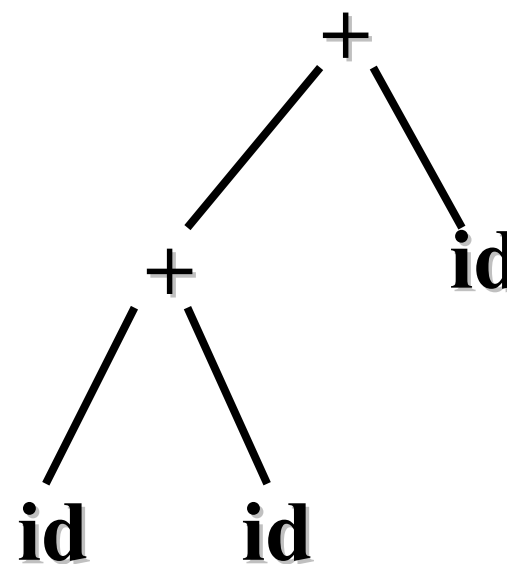
$$\{a^i b^i c^j \mid i, j \geq 1\} \cup \{a^i b^j c^j \mid i, j \geq 1\}$$

存在一个二义性的句子  $a^k b^k c^k$ 。

# 分析树与(抽象)语法树不同



(语法)分析树



(抽象)语法树



## 2.7 本章小结

---

- 语言及其描述
- 文法的基本概念
- Chomsky体系
- CFG的语法树
- 文法的二义性
- 语言的固有二义性



*School of Computer Science & Technology*  
*Harbin Institute of Technology*



## 第三章 词法分析

- 重点：**词法分析器的输入、输出，  
用于识别符号的状态转移图的构造，  
根据状态转移图实现词法分析器。
- 难点：**词法的正规文法表示、正规表达式表示、  
状态转移图表示，它们之间的转换。





# 第3章 词法分析

---

3.1 词法分析器的功能

3.2 单词的描述

3.3 单词的识别

3.4 词法分析程序的自动生成

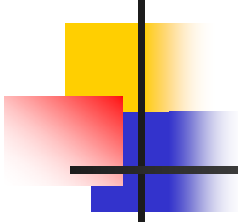
3.5 本章小结



## 3.1 词法分析器的功能

---

- 功能：输入源程序，输出单词符号(token)。  
即：把构成源程序的字符串转换成“等价的”  
单词(记号)序列
  - 根据词法规则识别及组合单词，进行词法检查
  - 对数字常数完成数字字符串到二进制数值的转换
  - 删去空格字符和注释



## 3.1.1 单词的分类与表示 & 3.1.2 词法分析器的输出

### 一、单词的种类

1. 关键字:也称基本字, begin、end、for、do...
2. 标识符:由用户定义,表示各种名字
3. 常数:整常数、实常数、布尔常数、字符串常数等
4. 运算符:算术运算符+、-、\*、/等; 逻辑运算符not、or与and等; 关系运算符=、<>、>=、<=、>和<等
5. 分界符: , 、 ; 、 ( 、 ) ...

## 二、单词的内部形式

表示单词的种类，可用整数编码或记忆符表示

不同的单词不同的值

种别	属性值
----	-----

几种常用的单词内部形式：

- 1、按单词种类分类
- 2、保留字和分界符采用一符一类
- 3、标识符和常数的单词值又为指示字（指针值）





# 1、按单词种类分类

---

单词名称	类别编码	单词值
标识符	1	内部字符串
无符号常数(整)	2	整数值
无符号浮点数	3	数值
布尔常数	4	0 或 1
字符串常数	5	内部字符串
保留字	6	保留字或内部编码
分界符	7	分界符或内部编码

## 2、保留字和分界符采用一符一类

单词名称	类别编码	单词值
标识符	1	内部字符串
无符号常数(整)	2	整数值
无符号浮点数	3	数值
布尔常数	4	0 或 1
字符串常数	5	内部字符串
BEGIN	6	—
END	7	—
FOR	8	—
DO	9	—
.....	.....	.....
:	20	—
+	21	—
*	22	—
,	23	—
(	.....	—

## 例3.1 语句if count>7 then result := 3.14 的单词符号序列

(IF, 0)

(ID, 指向count 的符号表入口)

(GT, 0)

(INT, 7)

(THEN, 0)

(ID, 指向result的符号表入口)

(ASSIGN, 0)

(REAL, 3.14)

(SEMIC, 0)

跟实现有关



## 3.1.3 源程序的输入缓冲与预处理

---

### ■ 超前搜索和回退

- 双字符运算符 ( $**$ ,  $/*$ ,  $:=$ , ...)

- DO 90 k=1, 10

- DO 90 k=1. 10

### ■ 缓冲区

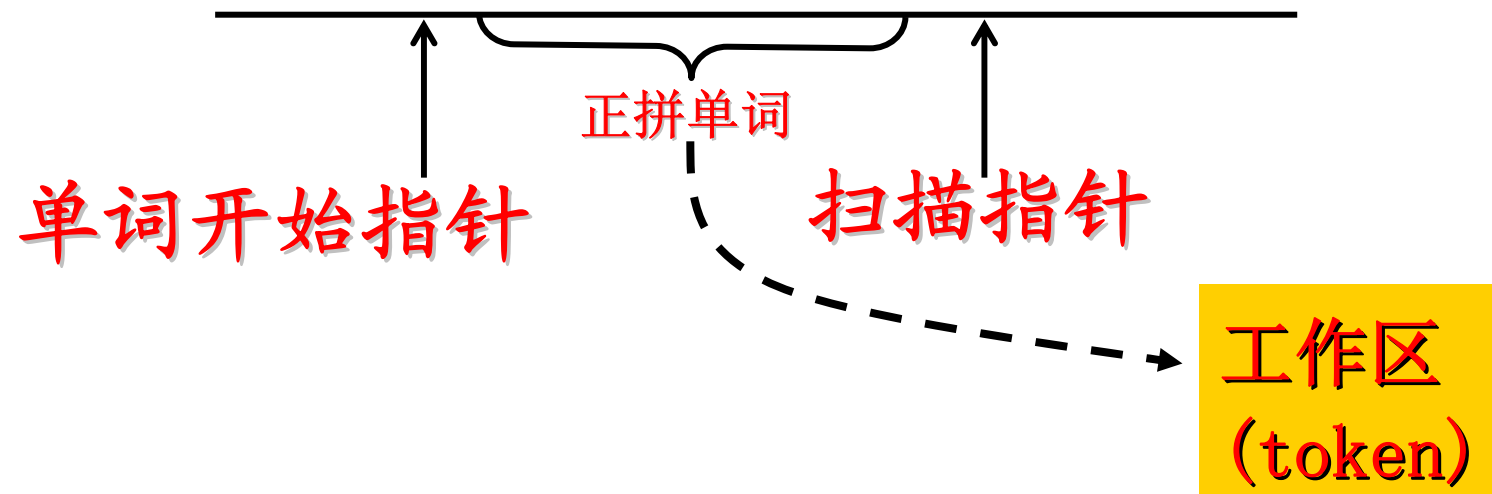
- 假定源程序存储在磁盘上，这样每读一个字符就需要访问一次磁盘，效率显然是很低的。

### ■ 空白字符的剔除

- 剔除源程序中的无用符号、空格、换行、注释等

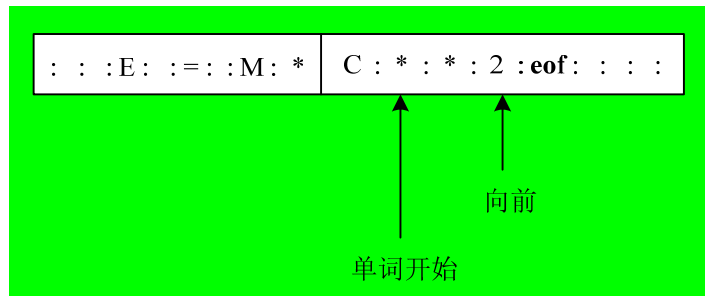
### 3.1.3 源程序的输入缓冲与预处理(续)

#### ■ 输入缓冲区

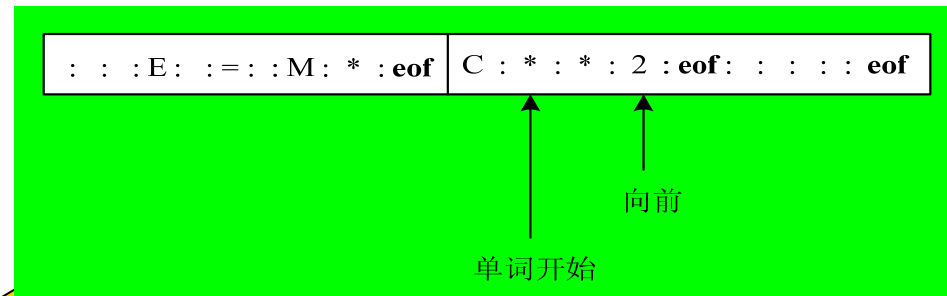


## 3.1.3 源程序的输入缓冲与预处理(续)

### 双缓冲区问题--超前扫描导致的效率问题



```
if forward在缓冲区第一部分末尾 then begin
  重装缓冲区第二部分;
  forward := forward + 1
end
else if forward在缓冲区第二部分末尾 then begin
  重装缓冲区第一部分;
  将forward移到缓冲区第一部分开始
end
else forward:= forward + 1;
```



```
forward := forward + 1;
if forward ↑ = eof then begin
  if forward在第一部分末尾 then begin
    重装第二部分;
    forward := forward + 1
  end
  else if forward在第二部分末尾 then begin
    重装第一部分;
    将forward 移到第一部分开始
  end
end
else /* eof 在表示输入结束 */
  终止词法分析
end
```

#### 大小问题

128Byte\*2|1024Byte\*2|4096Byte\*2

### 问题：如何设计和实现扫描器？



## 3.1.4 词法分析阶段的错误处理

---

1. 非法字符检查
2. 关键字拼写错误检查
3. 不封闭错误检查
4. 重复说明检查
5. 错误恢复与续编译

紧急方式恢复(panic-mode recovery)

反复删掉剩余输入最前面的字符，直到词法分析器能发现一个正确的单词为止。

## 3.1.5 词法分析器的位置

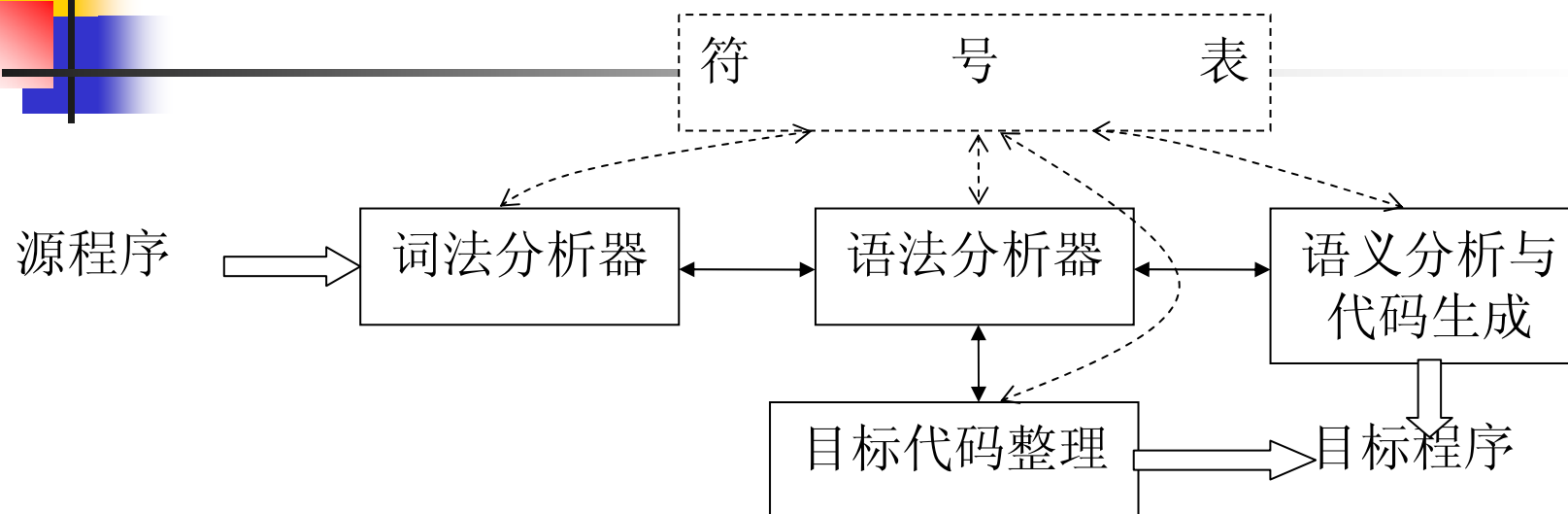


图3.4 以语法分析器为中心

- 以语法分析器为中心的优点：
  - 简化编译器的设计。
  - 提高编译器的效率。
  - 增强编译器的可移植性。



## 3.2 单词的描述

### 3.2.1 正则文法

- 正则文法  $G = (V, T, P, S)$  中, 对  $\forall \alpha \rightarrow \beta \in P$ ,  $\alpha \rightarrow \beta$  均具有形式  $A \rightarrow w$  或  $A \rightarrow wB$  ( $A \rightarrow w$  或  $A \rightarrow Bw$ ), 其中  $A, B \in V, w \in T^+$ .
- 例3.2 标识符的文法
  - 约定: 用 **digit** 表示数字: 0,1,2,...,9;  
用 **letter** 表示字母: A,B,...,Z,a,b,...,z
  - **$\langle id \rangle \rightarrow \langle letter \rangle \mid \langle id \rangle \langle digit \rangle \mid \langle id \rangle \langle letter \rangle$**
  - **$\langle letter \rangle \rightarrow A \mid B \mid \dots \mid Z \mid a \mid b \mid \dots \mid z$**
  - **$\langle digit \rangle \rightarrow 0 \mid 1 \mid 2 \mid \dots \mid 9$**



## 3.2.2 正则表达式

---

- 例3.2: 标识符的另一种表示
  - `letter(letter|digit)*`
  - `|` 表示“或”
  - `*` 表示Kleene闭包
  - `+` 表示正闭包
  - `?` 表示“0或1个”
  - `Letter`和`(letter|digit)*`的并列表示两者的连接
- 正则式`r`表示正则集,相应的正则集记为 `L(r)`

## 3.2.2 正则表达式(续)—RE

定义3.1 设  $\Sigma$  是一个字母表, 则  $\Sigma$  上的正则表达式及其所表示的正则语言可递归地定义如下:

- (1)  $\emptyset$  是  $\Sigma$  上的一个正则表达式, 它表示空集;
- (2)  $\varepsilon$  是  $\Sigma$  上的一个正则表达式, 它表示语言  $\{\varepsilon\}$ ;
- (3) 对于  $\forall a(a \in \Sigma)$ ,  $a$  是  $\Sigma$  上的一个正则表达式, 它表示的正则语言是  $\{a\}$ ;
- (4) 假设  $r$  和  $s$  都是  $\Sigma$  上的正则表达式, 它们表示的语言分别为  $L(r)$  和  $L(s)$ , 则:
  - ①  $(r)$  也是  $\Sigma$  上的正则表达式, 它表示的语言为  $L(r)$ ;
  - ②  $(r|s)$  也是  $\Sigma$  上的正则表达式, 它表示的语言为  $L(r) \cup L(s)$ ;
  - ③  $(r \bullet s)$  也是  $\Sigma$  上的正则表达式, 它表示的语言为  $L(r)L(s)$ ;
  - ④  $(r^*)$  也是  $\Sigma$  上的正则表达式, 它表示的语言为  $(L(r))^*$ ;
- (5) 只有经有限次使用上述规则构造的表达式才是  $\Sigma$  上的正则表达式。



# 正则表达式中的运算优先级

运算优先级和结合性:

- \*高于“连接”和|, “连接”高于|
- |具有交换律、结合律
- “连接”具有结合律、和对|的分配律
- ()指定优先关系

意义清楚时, 括号可以省略

例:

1.  $L((a|b)^*) = L((a^*|b^*)^*) = \{x | x \text{ 是 } a \text{ 和 } b \text{ 构成的符号串, 包括 } \varepsilon\}$

2.  $L(a|a^*b) = \{a, b, ab, aab, aaab, aaaab, \dots\}$



### 3.2.3 正则表达式与正则文法的等价性

---

#### ■ 正则表达式与正则文法等价

- 对任意一个正则表达式，存在一个定义同一语言的正则文法
- 对任意一个正则文法，存在一个定义同一语言的正则表达式

# 根据正则文法构造等价的正则表达式

- 问题：给定正则文法 $G$ ，构造一个正则表达式 $r$ ，使得 $L(r) = L(G)$
- 基本思路
  - 为正则文法的每个产生式构造一个正则表达式方程式，这些方程式中的变量是文法 $G$ 中的语法变量，各变量的系数是正则表达式，简称为方程式。从而得到一个联立方程组。
  - 用代入消元法消去联立方程组中除开始符号外的其他变量，最后得到关于开始符号 $S$ 的解： $S = r$ ， $r$ 即为所求的正则表达式。

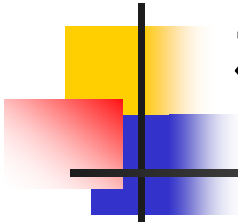
# 根据正则文法构造等价的正则表达式

## ■ 具体步骤

(1) 根据正则文法  $G$  构造正则表达式联立方程组。

假设正则文法  $G$  是右线性的，其每个产生式的右部只含有一个终结符，则有如下方程式构造规则：

- ① 对形如  $A \rightarrow a_1 | a_2 | \dots | a_m$  的产生式，构造方程式  $A = a_1 | a_2 | \dots | a_m$ 。其中可以有形如  $A \rightarrow \varepsilon$  的产生式；
- ② 对形如  $A \rightarrow a_1 A | a_2 A | \dots | a_m A$  的产生式，构造方程式  $A = (a_1 | a_2 | \dots | a_m)^* A$ ；
- ③ 对形如  $A \rightarrow a_1 B | a_2 B | \dots | a_m B$  的产生式，构造方程式  $A = (a_1 | a_2 | \dots | a_m) B$ ，其中  $B \neq A$ 。



## 根据正则文法构造等价的正则表达式

(2)解联立方程组，求等价的正则表达式 $r$   
用代入消元法逐个消去方程组中除开始符号 $S$ 外的其他变量，最后即可得到关于开始符号 $S$ 的解。代入消元规则如下：

- ① 如果有 $A=r_1B|r_2B|\dots|r_nB$ ，则用 $A=(r_1|r_2|\dots|r_n)B$ 替换之，其中 $B \neq A$ ；
- ② 如果有 $A=t_1A|t_2A|\dots|t_mA$ ，则用 $A=(t_1|t_2|\dots|t_m)^*A$ 替换之；



## 根据正则文法构造等价的正则表达式

- ③ 如果有  $A=(r_1|r_2|\dots|r_n)B$ ,  $B=(t_1|t_2|\dots|t_m)C$ , 则用  $A=(r_1|r_2|\dots|r_n)(t_1|t_2|\dots|t_m)C$  替换之, 其中  $B \neq A$ ;  
如果有  $A=(r_1|r_2|\dots|r_n)B$ ,  $B=(t_1|t_2|\dots|t_m)$ , 则用  $A=(r_1|r_2|\dots|r_n)(t_1|t_2|\dots|t_m)$  替换之, 其中  $B \neq A$ ;
- ④ 对  $A=(t_1|t_2|\dots|t_m)^*A$  且  $A=(r_1|r_2|\dots|r_n)B$ , 其中  $B \neq A$ , 则用  $A=(t_1|t_2|\dots|t_m)^*(r_1|r_2|\dots|r_n)B$  替换之;  
对  $A=(t_1|t_2|\dots|t_m)^*A$  且  $A=r_1|r_2|\dots|r_n$  则用  $A=(t_1|t_2|\dots|t_m)^*(r_1|r_2|\dots|r_n)$  替换之;

## 根据正则文法构造等价的正则表达式

⑤ 如果有  $A = \beta_1$ 、 $A = \beta_2$ 、...、 $A = \beta_h$ ，则用  $A = \beta_1 | \beta_2 | \dots | \beta_h$  代替之。

如果最后得到的关于  $S$  的方程式为如下形式，

$$S = \alpha_1 | \alpha_2 | \dots | \alpha_h$$

则将方程式右边所有其中仍然含有语法变量的  $\alpha_i (1 \leq i \leq n)$  删除，得到的结果就是与  $G$  等价的正则表达式；如果任意的  $\alpha_i (1 \leq i \leq n)$  均含有语法变量，则  $\emptyset$  就是与  $G$  等价的正则表达式。

# 根据正则文法构造等价的正则表达式

## ■ 例3.6 将如下文法 $G$ 转换成相应的正则表达式

$$S \rightarrow aS|aB$$

$$B \rightarrow bB|bC|aB|bS$$

$$C \rightarrow cC|c$$

### 1.列方程组

$$\blacksquare S=a^*S \quad S=aB \quad B=(a|b)^*B \quad B=bC$$

$$\blacksquare B=bS \quad C=c^*C \quad C=c$$

### 2.代入法解方程组

$$\blacksquare C=c^*c \quad B=bc^*c|bc \quad B=(a|b)^*(bc^*c|bc)$$

$$\blacksquare B=(a|b)^*bS \quad S=a^*aB$$

$$\blacksquare S=a^*a(a|b)^*bS|a^*a(a|b)^*(bc^*c|bc)$$

$$\blacksquare S=(a^*a(a|b)^*b)^*a^*a(a|b)^*(bc^*c|bc)$$

$$\blacksquare \text{如果用正闭包表示, 则为 } (a^+(a|b)^*b)^*a^+(a|b)^*(bc^+|bc)$$



# 将正则表达式转换成等价的正则文法

- 问题：给定  $\Sigma$  上的一个正则表达式  $r$ ，根据  $r$  构造正则文法  $G$ ，使得  $L(G)=L(r)$
- 定义3.3 设字母表为  $\Sigma$ ， $\{A、B、...、C\}$  为语法变量集合，对于  $\Sigma$  上的任意正则表达式  $r$ ，形如  $A \rightarrow r$  的式子称为正则定义式；如果  $r$  是  $\Sigma$  中的字母和用正则定义式定义的变量组成的正则表达式，则形如  $A \rightarrow r$  的式子称为正则定义式。



# 将正则表达式转换成等价的正则文法

---

- 按如下方法构造正则定义式，并逐步将其转换成正则文法
- 引入开始符号 $S$ ，从如下正则定义式开始
  - $S \rightarrow r$
- 按如下规则将 $S \rightarrow r$ 分解为新的正则定义式，在分解过程中根据需要引入新的语法变量



## 将正则表达式转换成等价的正则文法

■  $A \rightarrow r$  是正则定义式，则对  $A \rightarrow r$  的分解规则如下：

(1) 如果  $r = r_1 r_2$ ，则将  $A \rightarrow r$  分解为  $A \rightarrow r_1 B$ ， $B \rightarrow r_2$ ， $B \in V$ ；

(2) 如果  $r = r_1^* r_2$ ，则将  $A \rightarrow r$  分解为  $A \rightarrow r_1 A$ ， $A \rightarrow r_2$ ；

(3) 如果  $r = r_1 | r_2$ ，则将  $A \rightarrow r$  分解为  $A \rightarrow r_1$ ， $A \rightarrow r_2$ 。

不断应用分解规则(1)到(3)对各个正则定义式进行分解，直到每个正则定义式右端只含一个语法变量(即符合正则文法产生式的形式)为止。

## 例3.9 正则表达式到正则文法的转换

$$\bullet a(a|b)^*(\varepsilon | ((.|_)(a|b)(a|b)^*))$$

$$= a(a|b)^*$$

$$|a(a|b)^*.(a(a|b)^*|b(a|b)^*)$$

$$|a(a|b)^*_(a(a|b)^*|b(a|b)^*)$$

$$=aA|aB$$

$$A=aA|bA|\varepsilon \quad B=aB|bB|.C|_C$$

$$C=aA|bA$$

$$S \rightarrow aA|aB \quad A \rightarrow aA|bA|\varepsilon$$

$$B \rightarrow aB|bB|.C|_C$$

$$C \rightarrow aA|bA$$



## 例 3.10 标识符定义的转换

---

- 引入  $S$

$$S \rightarrow \text{letter} (\text{letter}|\text{digit})^*$$

- 分解为

$$S \rightarrow \text{letter } A$$

$$A \rightarrow (\text{letter}|\text{digit})A \mid \varepsilon$$

- 执行连接对 $|$ 的分配律

$$S \rightarrow \text{letter } A$$

$$A \rightarrow \text{letter } A|\text{digit } A \mid \varepsilon$$





# 高级语言词法的简单描述

---

- 词法

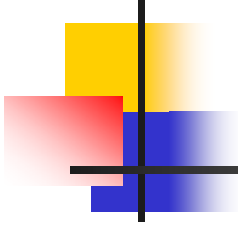
- 单词符号的文法，用来描述高级语言中的：  
标识符、常数、运算符、分界符、关键字

- 参考教材P73-77，了解如何定义高级语言中的整数、实数.....等的相应正则文法。

# 例 3.7 某简易语言的词法

——正则定义式

词法规则	单词种别	属性
$\langle \text{标识符} \rangle \rightarrow \langle \text{字母} \rangle (\langle \text{字母} \rangle   \langle \text{数字} \rangle)^*$	<i>IDN</i>	符号表入口
$\langle \text{无符号整数} \rangle \rightarrow \langle \text{数字} \rangle (\langle \text{数字} \rangle)^*$	<i>NUM</i>	数值
$\langle \text{赋值符} \rangle \rightarrow :=$	<i>ASG</i>	无
其它单词 $\rightarrow$ 字符本身	单词名称	无



# 变换为正规文法

---

$\langle \text{标识符} \rangle \rightarrow \text{letter} \langle \text{标识符尾} \rangle$

$\langle \text{标识符尾} \rangle \rightarrow \varepsilon \mid \text{letter} \langle \text{标识符尾} \rangle \mid \text{digit} \langle \text{标识符尾} \rangle$

$\langle \text{整数} \rangle \rightarrow \text{digit} \langle \text{整数尾} \rangle$

$\langle \text{整数尾} \rangle \rightarrow \varepsilon \mid \text{digit} \langle \text{整数尾} \rangle$


$\langle \text{赋值号} \rangle \rightarrow :=$

$\langle \text{加号} \rangle \rightarrow +$

$\langle \text{等号} \rangle \rightarrow =$

...

(其它：实数、算术运算符、关系运算符、分号、括号等)



问题：如何  
识别记号？

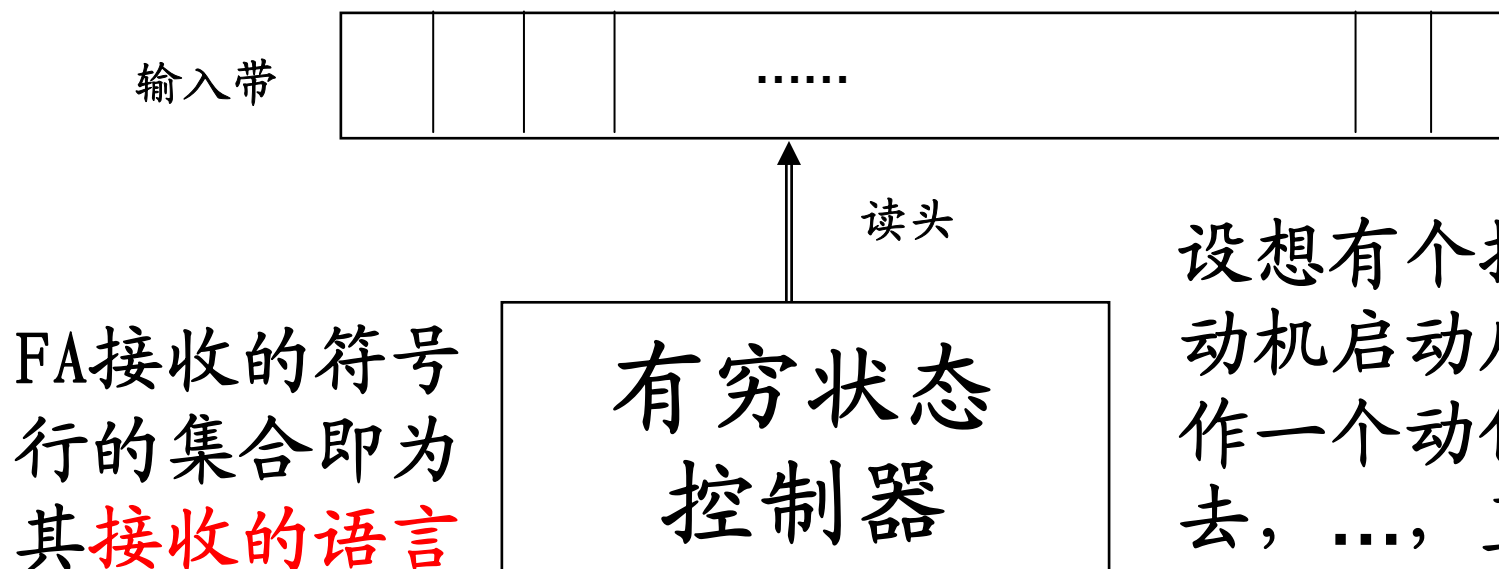


## 3.2.4 有穷状态自动机

---

- 具有离散输入输出的系统的数学模型
- 具有有穷个内部状态
- 系统只需根据当前所处的状态和面临的输入就能确定后继的行为，处理完当前输入后系统的状态将发生变化
- 具有初始状态和终止状态
- 例：电梯、文本编辑程序、词法分析程序.....

# 有穷自动机的物理模型



设想有个按钮，自动机启动后一个动作一个动作地做下去，...，直到没有输入。如果停在终态，接收；如果停在非终态，不接收

$[p, a] \rightarrow q$ ，读头前进一格



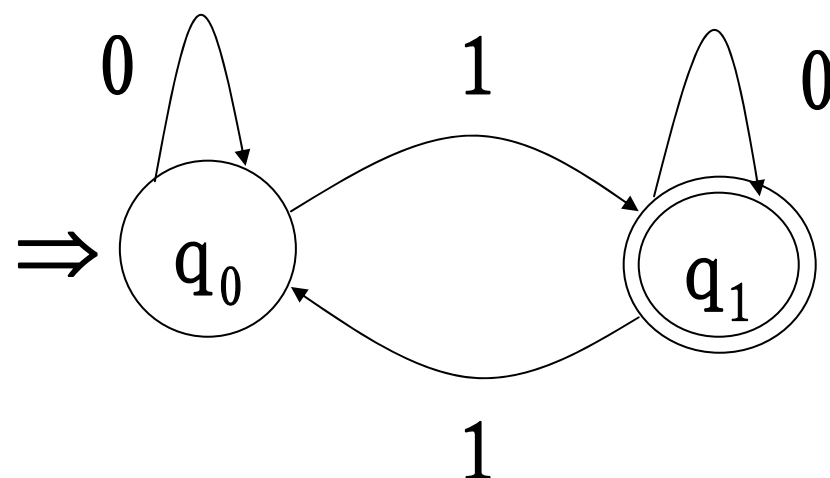
# 有穷自动机的用处

---

- 有穷自动机是许多重要类型的硬件和软件的有用模型
  - 数字电路的设计和检查软件
  - 典型编译器的词法分析器
  - 扫描大量文本来发现单词、短语或其他模式的出现的软件
  - 所有只有有穷个不同状态的系统（如通信协议或安全交换信息的协议）的验证软件

# 例： 一个奇偶校验器

测试输入中1的个数的奇偶性，并且只接收含有奇数个1的那些输入串。



注意：状态有记忆功能，记住输入串的部分特征。

问题：有穷自动机的形式描述？

关键是如何描述动作？



# 确定的有穷自动机的形式定义

定义3.4 一个**确定的有穷自动机**  $M$  (记作DFA  $M$ ) 是一个五元组  $M = (Q, \Sigma, \delta, q_0, F)$ , 其中

- ①  $Q$  是一个有穷状态集合。
- ②  $\Sigma$  是一个字母表, 它的每个元素称输入符号。
- ③  $q_0 \in Q$ ,  $q_0$  称为初始状态。
- ④  $F \subseteq Q$ ,  $F$  称为终止状态集合。
- ⑤  $\delta$  是一个从  $Q \times \Sigma$  到  $Q$  的单值映射

$$\delta(p, a) = q \quad (p, q \in Q, a \in \Sigma)$$

表示当前状态为  $p$ , 输入符号为  $a$  时, 自动机将转换到下一个状态  $q$ ,  $q$  称为  $p$  的一个后继。





# DFA的表示

---

例 设 DFA  $M = ( \{ 0, 1, 2, 3 \}, \{ a, b \}, \delta, 0, \{ 3 \} )$

其中:

$$\delta(0, a) = 1, \quad \delta(1, a) = 3$$

$$\delta(2, a) = 1, \quad \delta(3, a) = 3$$

$$\delta(0, b) = 2, \quad \delta(1, b) = 2$$

$$\delta(2, b) = 3, \quad \delta(3, b) = 3$$

✓ 一个 DFA 有三种表示:

(1) 转换函数;

(2) 转移矩阵;

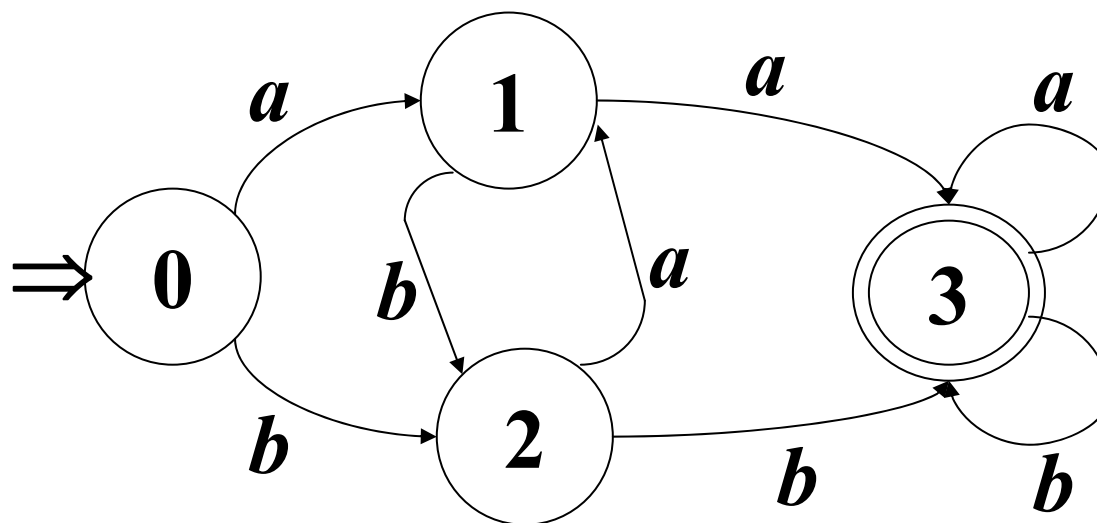
(3) 状态转换图。

## 转移矩阵

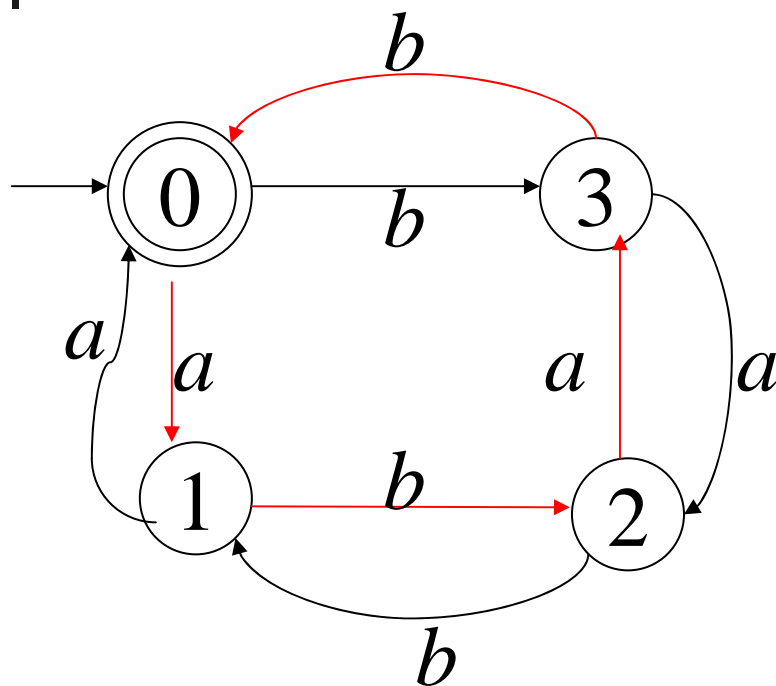
	<i>a</i>	<i>b</i>
0	1	2
1	3	2
2	1	3
3	3	3

易存储

## 状态转换图



# DFA $M$ 接受的语言



从状态转换图看，从初态出发，沿任一条路径到达接受状态，这条路径上的弧上的标记符号连接起来构成的符号串被接受。

如: *abab*

问题：如何形式描述DFA接收的语言？



# DFA $M$ 接受的语言

---

如果对所有  $w \in \Sigma^*$ ,  $a \in \Sigma$ ,  $q \in Q$  以下述方式  
递归地扩展  $\delta$  的定义  $\hat{\delta}(q, \varepsilon) = q$ ,

$$\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a),$$

则  $M$  所接收的语言为:

$$L(M) = \{w \mid w \in \Sigma^*, \text{ 且 } \delta(q_0, w) \in F\}$$

对于上页例中的 DFA  $M$  和  $w=baa$ ,

$$\delta(0, baa) = \delta(2, aa) = \delta(1, a) = 3$$



# 非确定的有穷自动机NFA $M$

定义3.6 非确定的有穷自动机 $M$ 是一个五元组

$$M = (Q, \Sigma, \delta, q_0, F)$$

其中  $Q, \Sigma, q_0, F$  的意义和DFA的定义一样，而  $\delta$  是一个从  $Q \times \Sigma \cup \{\varepsilon\}$  到  $Q$  的子集的映射，即  $\delta: Q \times S \rightarrow 2^Q$ ，其中  $S = \Sigma \cup \{\varepsilon\}$ 。

类似于DFA，NFA  $M$  亦可用状态转换图表示，同样也可以定义NFA  $M$  接受的语言。



# DFA $M$ 的模拟算法

---

输入：以eof结尾的串 $x$ ，DFA  $M = (Q, \Sigma, \delta, q_0, F)$ ;

输出：如果 $M$ 接受 $x$ 则输出“yes”，如果 $M$ 不接受 $x$ 则输出“no”

步骤：

```
1  $s = q_0$ ;  
2  $c = \text{getchar}(x)$ ;  
3 while ( $c \neq \text{eof}$ ) {  
4    $s = \text{move}(s, c)$ ;  
5    $c = \text{getchar}(x)$ ;  
6 }  
7 if  $s \in F$  return “yes”  
8 else return “no”;
```



## 3.2.5 状态转换图

- 定义3.7 设 $M=(Q, \Sigma, \delta, q_0, F)$ 为一个有穷状态自动机, 满足如下条件的有向图被称为 $M$ 的状态转换图(transition diagram):
  - (1)  $q \in Q \Leftrightarrow q$ 是该有向图中的一个顶点;
  - (2)  $\delta(q, a)=p \Leftrightarrow$ 图中有一条从顶点 $q$ 到顶点 $p$ 的标记为 $a$ 的弧;
  - (3)  $q \in F \Leftrightarrow$ 标记为 $q$ 的顶点被用双层圈标出;
  - (4) 用标有start的箭头指出 $M$ 的开始状态。

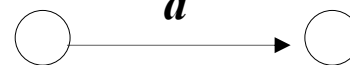
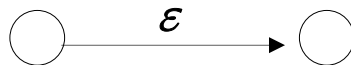
## 3.2.6 正则表达式转换为状态转换图



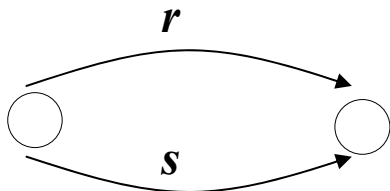
(a)  $\emptyset$ 对应的状态转换图



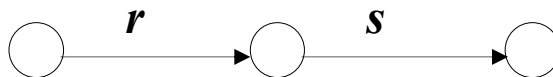
(b)  $\varepsilon$ 对应的状态转换图



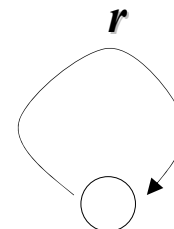
(c)  $a$ 对应的状态转换图



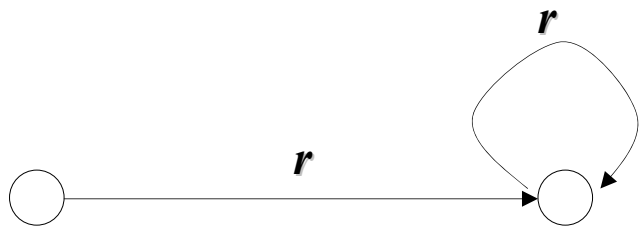
(d)  $r | s$ 对应的状态转换图



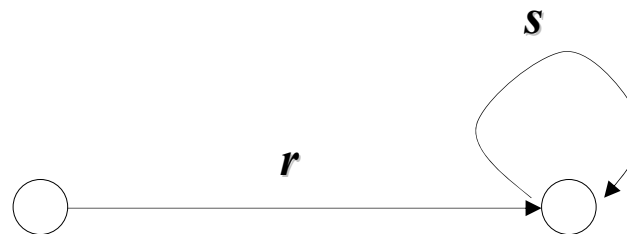
(e)  $rs$ 对应的状态转换图



(f)  $r^*$ 对应的状态转换图



(g)  $r^+$ 对应的状态转换图



(h)  $rs^*$ 对应的状态转换图

图3.8 典型正则表达式对应的状态转换图





## 3.2.6 正则表达式转换为状态转换图

### ■ 转换过程如下:

- 设置一个开始状态和一个终止状态，从开始状态到终止状态引一条标记为待转换表达式的边；
- 检查图中边的标记，如果相应的标记不是字符、 $\emptyset$ 、 $\varepsilon$ 或用“|”连接的字符和  $\varepsilon$ ，则根据规则(1)-(8)进行替换，直到图中不再存在不满足要求的边。按照习惯，如果一条边上标记的是 $\emptyset$ ，这个边就不用画出来。

## 3.3 单词的识别

### 3.3.1 有穷状态自动机与单词识别的关系

- 有穷状态自动机和正则文法等价，考虑到状态转换图的直观性，我们从状态转换图出发来考虑词法分析器的设计。
- 允许在状态转换图的边上标记像digit、letter这样意义明确的符号，other表示例外情况



### 3.3.1 有穷状态自动机与单词识别的关系

---

- 考虑到在识别单词的过程中需要执行一些动作，所以将这些动作标记标在基本的状态转换图上。
- 如果到达终止状态，则意味着读入了一个与当前单词无关的字符，由于这个无关字符是下一个单词的开始符号，所以必须回退一个字符。状态上的\*表示向前指针必须回退一个字符。



## 例 3.14 不同进制无符号整数的识别

---

八进制数: (OCT, 值)

■  $\text{oct} \rightarrow 0(0|1|2|3|4|5|6|7)(0|1|2|3|4|5|6|7)^*$

十进制数: (DEC, 值)

■  $\text{dec} \rightarrow (1|\dots|9)(0|\dots|9)^* | 0$

十六进制数: (HEX, 值)

■  $\text{hex} \rightarrow 0\text{x}(0|1|\dots|9|\text{a}|\dots|\text{f}|\text{A}|\dots|\text{F})(0|\dots|9|\text{a}|\dots|\text{f}|\text{A}|\dots|\text{F})^*$

# 识别不同进制数的状态图

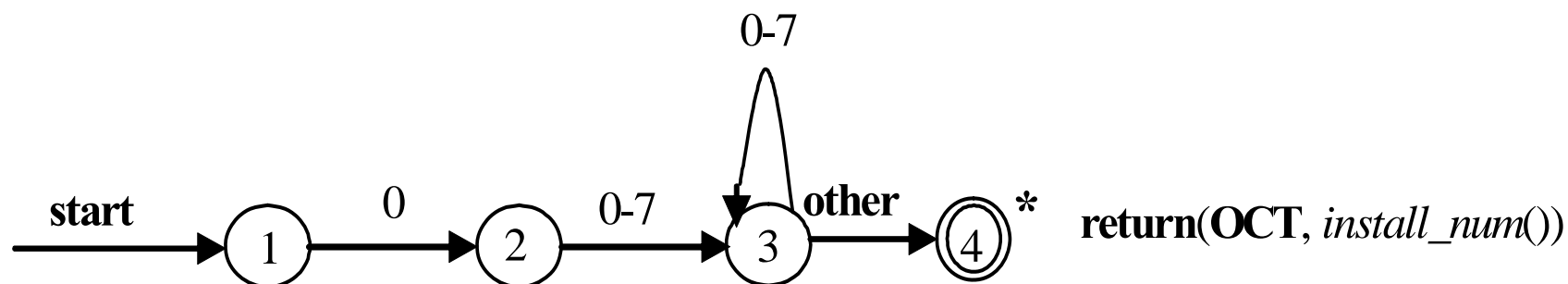


图 3.11 识别八进制无符号整数的状态转换图

# 识别不同进制数的状态图

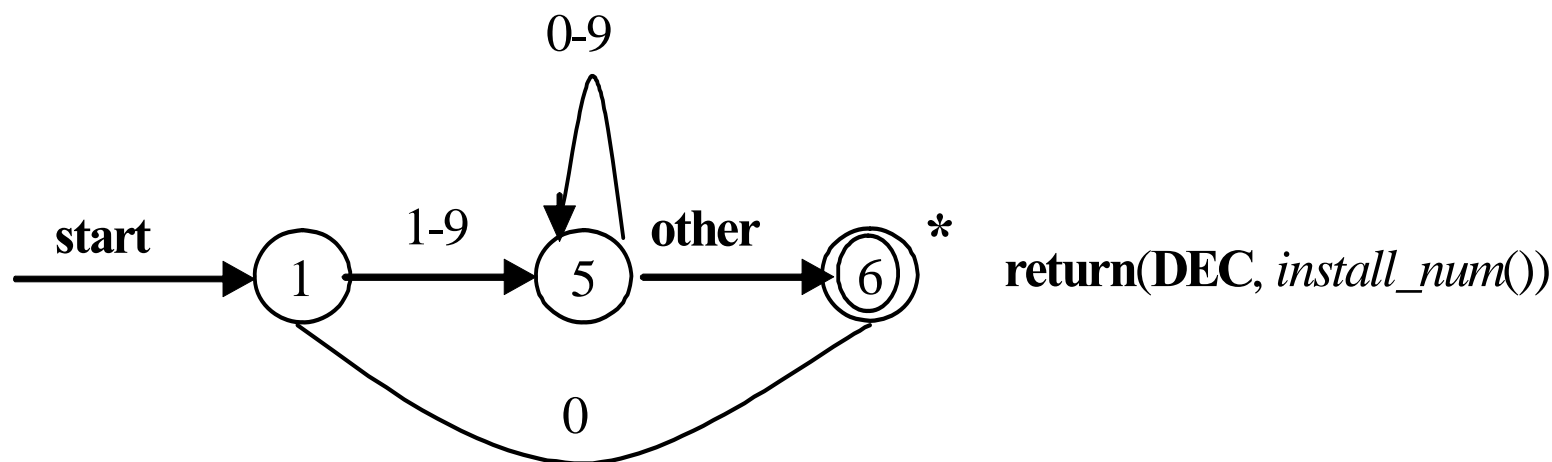


图 3.12 识别十进制无符号整数的状态转换图

# 识别不同进制数的状态图

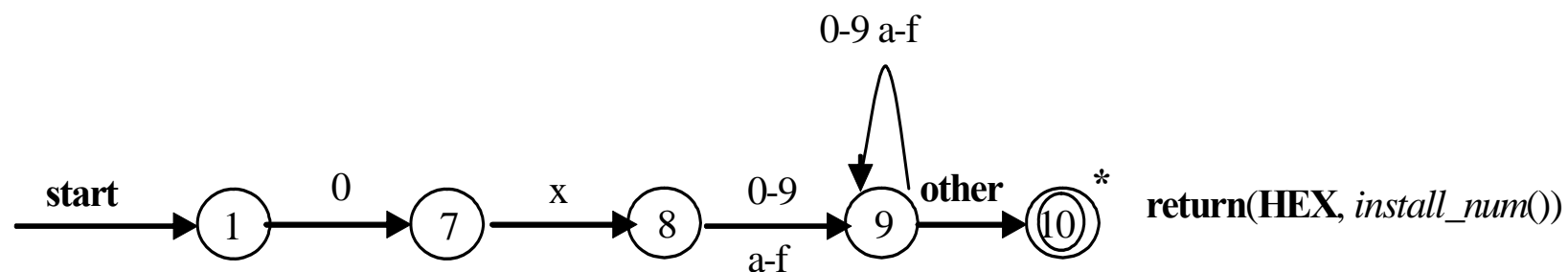


图 3.13 识别十六进制无符号整数的状态转换图

# 识别不同进制数的状态图

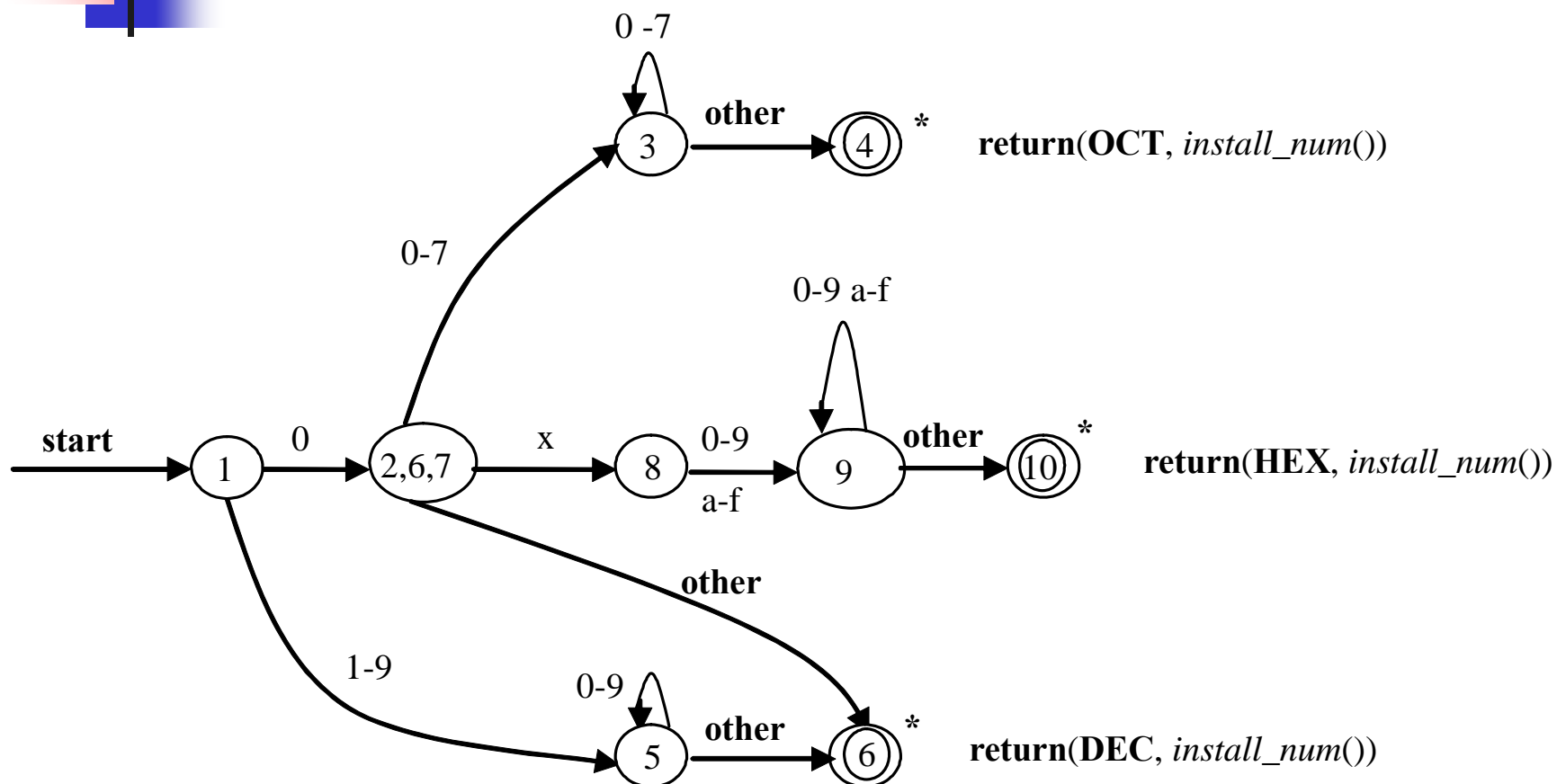


图3.14 识别C语言不同进制无符号整数的状态转换图

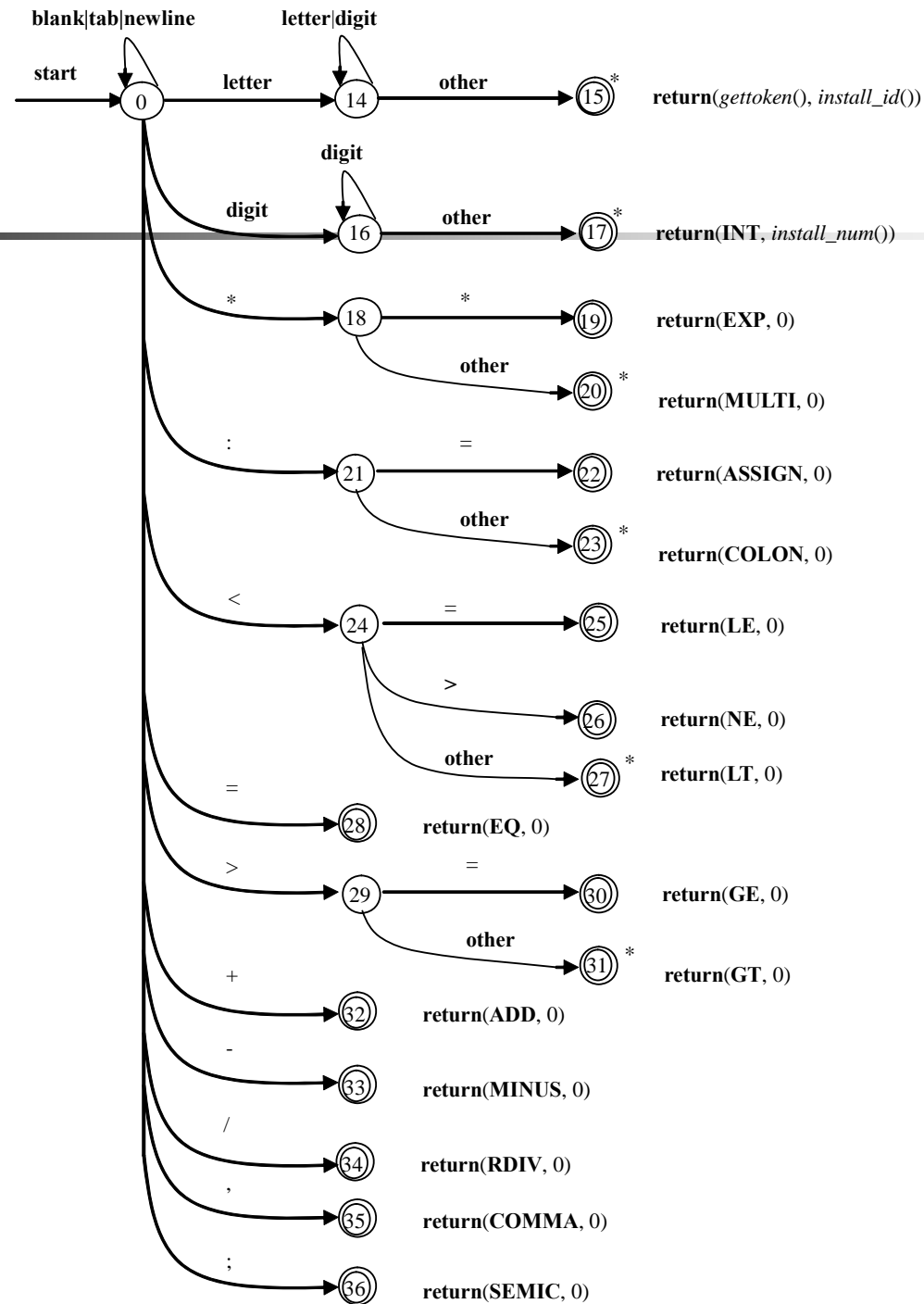
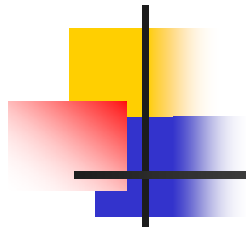




## 3.3.2 单词识别的状态转换图表示

---

- $\langle \text{id} \rangle \rightarrow \text{letter} \langle \text{id\_left} \rangle$
- $\langle \text{id\_left} \rangle \rightarrow \varepsilon \mid \text{letter} \langle \text{id\_left} \rangle \mid \text{digit} \langle \text{id\_left} \rangle$
- $\langle \text{int} \rangle \rightarrow \text{digit} \langle \text{int\_left} \rangle$
- $\langle \text{int\_left} \rangle \rightarrow \varepsilon \mid \text{digit} \langle \text{int\_left} \rangle$
- $\langle \text{assignment} \rangle \rightarrow :=$
- $\langle \text{relop} \rangle \rightarrow < \mid <= \mid = \mid < > \mid > \mid >=$
- $\langle \text{op} \rangle \rightarrow + \mid - \mid * \mid / \mid **$
- $\langle \text{delimiter} \rangle \rightarrow : \mid , \mid ;$





# 利用状态转换图识别单词

---

- (1) 从初始状态出发;
- (2) 读入一个字符;
- (3) 按当前字符转入下一状态;
- (4) 重复 (2)-(3) 直到无法继续转移为止。

在遇到读入的字符是单词的分界符时，若当前状态是终止状态，说明读入的字符组成了一个单词；否则，说明输入字符串 $w$ 不符合词法规则。



# 利用状态转换图识别单词

- (1) 从初始状态出发;
  - (2) 读入一个字符;
  - (3) 按当前字符转入下一状态;
  - (4) 重复 (2)-(3) 直到无法继续转移为止。
- 在遇到读入的字符是单词的分界符时，若当前状态是终止状态，说明读入的字符组成了一个单词；否则，说明输入字符串  $w$  不符合词法规则。
  - 如果从状态转换图的初始状态出发，分别沿着所有可能的路径到达终止状态，并将每条路径上的标记依次连接成字符串，则可以得到状态转换图能够识别的所有单词，这些单词组成的集合也就是状态转换图识别的语言。
  - 读入字符  $a$  时从状态  $A$  转换到状态  $B$  正好对应着一步推导过程，即  $A \Rightarrow aB$ ，边正好与产生式  $(A \rightarrow aB)$  相对应



# 由正则文法构造状态转换图

- (1) 以每个语法变量(或其编号)为状态结点, 开始符号对应初始状态 $S$ ;
- (2) 增设一个终止状态  $T$ ;
- (3) 对 $G$ 中每个形如 $A \rightarrow aB$ 的产生式, 从状态 $A$ 到状态 $B$ 画一条有向弧, 并标记为 $a$ ;
- (4) 对 $G$ 中每个形如 $A \rightarrow a$ 的产生式, 从状态 $A$ 到终止状态 $T$ 画一条标记为 $a$ 的有向弧;
- (5) 对 $G$ 中每个形如 $A \rightarrow \varepsilon$ 的产生式, 从状态 $A$ 到终止状态 $T$ 画一条标记为any的有向弧, any表示 $T$ 中的任何符号。



### 3.3.3 几种典型的单词识别问题

---

- 标识符的识别
- 关键字的识别
- 常数的识别
- 算符和分界符的识别
- 回退



## 3.3.4 状态转换图的实现

- 如果将状态转换图看成是单词的识别规则库的话，则单词识别程序从当前状态(最初为初始状态)出发，读入一个输入字符后，将首先查询该规则库。
- 重复以下过程，直至到达某个终止状态。
  - 如果从当前状态出发有一条边上标记了刚刚读入的输入字符，则单词识别程序将转入这条边所指向的那个状态，并再读入一个输入字符；
  - 否则调用出错处理程序；
  - 将从初始状态到该终止状态所经历的路径上的字符所组成的字符串作为一个单词输出；
  - 并将当前状态重新置为开始状态，以便进行下一个单词的识别；
  - 如果读完输入字符流后仍未进入某个终止状态则调用出错处理程序。



## 3.3.5 词法分析程序的编写

---

- 状态转移图——教材P93图3.15
- 状态转移图的实现——教材P105图3.22
- 词法分析程序 *token\_scan*( )
  - 输入：字符流
  - 输出：
    - *symbol*: 单词种别
    - *attr*: 属性（全局变量）





# 数据结构与子例程

## ■ 数据结构

- *ch* 字符变量，存放当前读入的输入字符
- *token* 字符串变量，存放构成单词的字符串
- *symbol* 单词种别（词法分析子程序的返回值）
- *attr* 属性（全局变量）

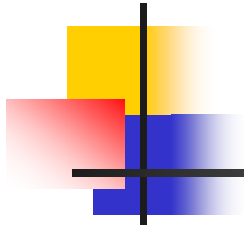
## ■ 子例程

- *install\_id(token)*:将*token*存入符号表，返回入口指针
- *getchar()*:从输入缓冲区中读入一个字符放入*ch*
- *retract()*:将向前指针回退一个字符，将*ch*置为空白符
- *copytoken()*:返回输入缓冲区中从开始指针  
*lexeme\_beginning*到向前指针*forward*之间的字符串
- *isLetter()* *isalpha()* *isalnum()*

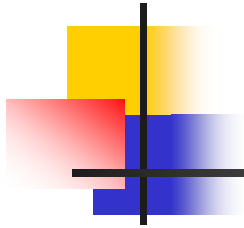


## 图3.15的状态转换图的实现算法

- *token token\_scan()*
- { *char ch*;
- *char\* token*;
- *ch = getchar()*;
- while (*ch == blank || ch == tab || ch == newline*) {
- *ch = getchar()*;
- *lexeme\_beginning++*;
- }
- if (*isalpha(ch)*) {*ch = getchar()*;
- while (*isalnum(ch)*)
- *ch = getchar()*;
- *retract(1)*;
- *token = copytoken()*;
- return(*gettoken(token), install\_id(token)*);}



- else
- if (*isdigit(ch)*) {
- *ch = getchar();*
- while (*isdigit(ch)*)
- *ch = getchar();*
- *retract(1);*
- *token = copytoken();*
- return(INT, *install\_num(token)*);
- }
- else
- switch(*ch*)
- {
- case '\*': *ch = getchar();*
- if(*ch == '\*'*) return(EXP, 0);
- else {
- *retract(1);*
- return(MULTI, 0);
- } break;



```
■ case ':': ch = getchar();  
■         if(ch == '=') return(ASSIGN, 0);  
■         else { retract(1); return(COLON, 0);  
■         } break;  
■ case '<': ch = getchar();  
■         if(ch == '=') return(LE, 0);  
■         else if(ch == '>') return(NE, 0);  
■     else { retract(1); return(LT, 0);  
■         } break;  
■ case '=': return(EQ, 0); break;  
■ case '>': ch = getchar();  
■         if(ch == '=') return(GE, 0);  
■         else { retract(1); return(GT, 0);  
■         } break;  
■ case '+': return(PLUS, 0); break;  
■ case '-': return(MINUS, 0); break;  
■ case '/': return(RDIV, 0); break;  
■ case ',': return(COMMA, 0); break;  
■ case ';': return(SEMIC, 0); break;  
■ default: error_handle(); break;}return;}
```



# 需要说明的问题

---

- 缓冲区预处理，超前搜索
- 关键字的处理，符号表的实现
- `Install_id()`：查找效率，算法的优化实现
- 词法错误处理
- 由于高级语言的词组成的集合为3型语言，所以，这里讨论的词法分析技术可以用于处理所有的3型语言，也就是所有的可以用3型文法描述的语言。  
如：信息检索系统的查询语言、命令语言等

## 3.4 词法分析程序的自动生成

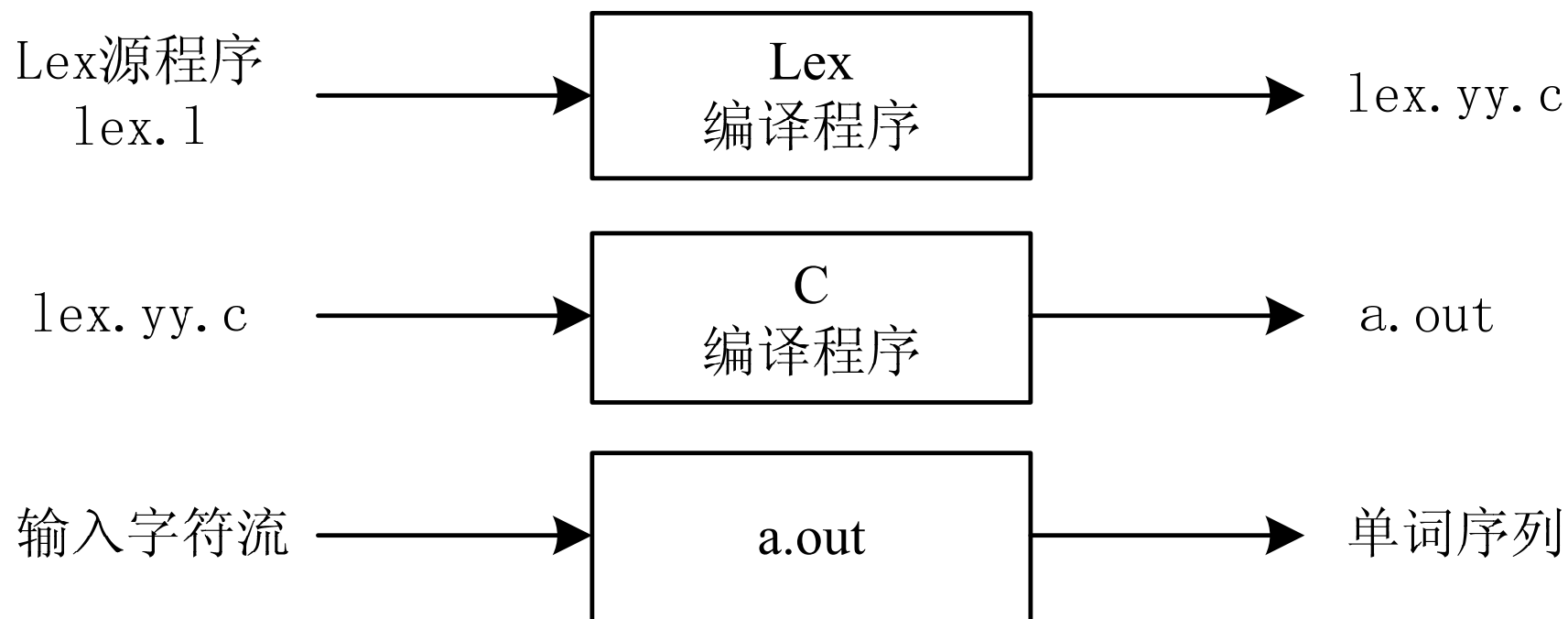


图 3.23 利用Lex建立词法分析程序的过程

## 3.4.1 Lex源程序

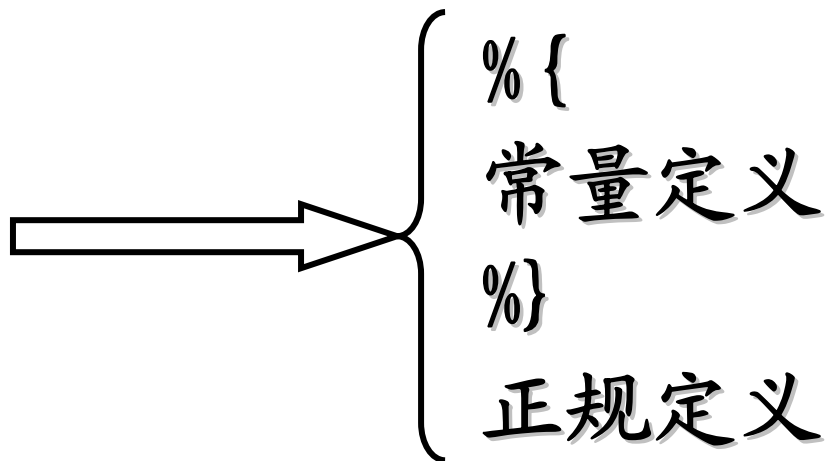
声明部分  
(正规定义式)

%%

识别规则部分  
(识别规则)

%%

辅助过程部分





## 3.4.1 Lex源程序

---

### 1、正规定义式

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

**digit**  $\rightarrow$  0|1|2|...|9

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

**integer**  $\rightarrow$  digit(digit)\*

### 2、识别规则

正规式	动作描述
-----	------

<b>token<sub>1</sub></b>	<b>{action<sub>1</sub>}</b>
--------------------------	-----------------------------

<b>token<sub>2</sub></b>	<b>{action<sub>2</sub>}</b>
--------------------------	-----------------------------

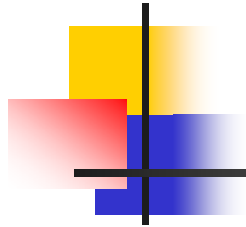
.....

<b>token<sub>n</sub></b>	<b>{action<sub>n</sub>}</b>
--------------------------	-----------------------------





• <b>%{</b>		• <b>delim</b>	<b>[\t\n]</b>
• <b>#include &lt;stdio.h&gt;</b>		• <b>ws</b>	<b>[delim]+</b>
• <b>#include "y.tab.h"</b>		• <b>letter</b>	<b>[a-zA-Z]</b>
• <b>#define ID</b>	<b>1</b>	• <b>digit</b>	<b>[0-9]</b>
• <b>#define INT</b>	<b>2</b>	• <b>id</b>	<b>{letter}({letter} {digit})*</b>
• <b>#define EXP</b>	<b>3</b>	• <b>number</b>	<b>{digit}+</b>
• <b>#define MULTI</b>	<b>4</b>	• <b>%%</b>	
• <b>#define COLON</b>	<b>5</b>	• <b>{ws}</b>	<b>;</b>
• <b>#define EQ</b>	<b>6</b>	• <b>begin</b>	<b>return(BEGIN);</b>
• <b>#define NE</b>	<b>7</b>	• <b>end</b>	<b>return(END);</b>
• <b>#define LE</b>	<b>8</b>	• <b>if</b>	<b>return(IF);</b>
• <b>#define GE</b>	<b>9</b>	• <b>then</b>	<b>return(THEN);</b>
• <b>#define LT</b>	<b>10</b>	• <b>else</b>	<b>return(ELSE);</b>
• <b>#define GT</b>	<b>11</b>	• <b>do</b>	<b>return(DO);</b>
• <b>#define PLUS</b>	<b>12</b>	• <b>program</b>	<b>return(PROGRAM);</b>
• <b>#define MINUS</b>	<b>13</b>	• <b>{id}</b>	<b>{yyval = install_id(); return(ID);}</b>
• <b>#define RDIV</b>	<b>14</b>	• <b>{number}</b>	<b>{yyval = install_num();</b>
• <b>#define COMMA</b>	<b>15</b>		<b>return(INT);}</b>
• <b>#define SEMIC</b>	<b>16</b>		
• <b>#define RELOP</b>	<b>17</b>		
• <b>#define ASSGIN</b>	<b>18</b>		
• <b>int line_no = 1;  %{</b>	<b>2012-4-26</b>		



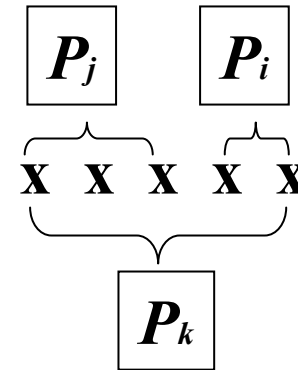
■	"<"	{yyval =LT; return(RELOP);}
■	"<="	{yyval =LE; return(RELOP);}
■	"="	{yyval =EQ; return(RELOP);}
■	">"	{yyval =GT; return(RELOP);}
■	">="	{yyval =GE; return(RELOP);}
■	"<>"	{yyval =NE; return(RELOP);}
■	"+"	return(PLUS);
■	"_"	return(MINUS);
■	"*"	return(MULTI);
■	"/"	return(RDIV);
■	"**"	return(EXP);
■	":"	return(COLON);
■	":="	return(ASSGIN);
■	","	return(COMMA);
■	";"	return(SEMIC);
■	\n	line_no++;
■	.	{ fprintf (stderr, "'%c' (0%o): illegal charcter at
	line	%d\n", yytext[0], yytext[0], line_no); }
■	%%	
■	install_id()	
■	{.....}	
■	install_num()	
■	{.....}	

如: begin: =

# LEX二义性问题的两条原则

## 1.最长匹配原则

在识别单词过程中，有一字符串  
根据最长匹配原则，应识别为这是一个符合 $P_k$ 规则的单词，  
而不是 $P_j$ 和 $P_i$ 规则的单词。



## 2.优先匹配原则

如果有一字符串有两条规则可以同时匹配时，那么用规则序列中位于前面的规则相匹配，所以排列在最前面的规则优先权最高。

## 3.4.2 Lex的实现原理

Lex的功能是根据Lex源程序构造一个词法分析程序，该词法分析器实质上是一个有穷自动机。

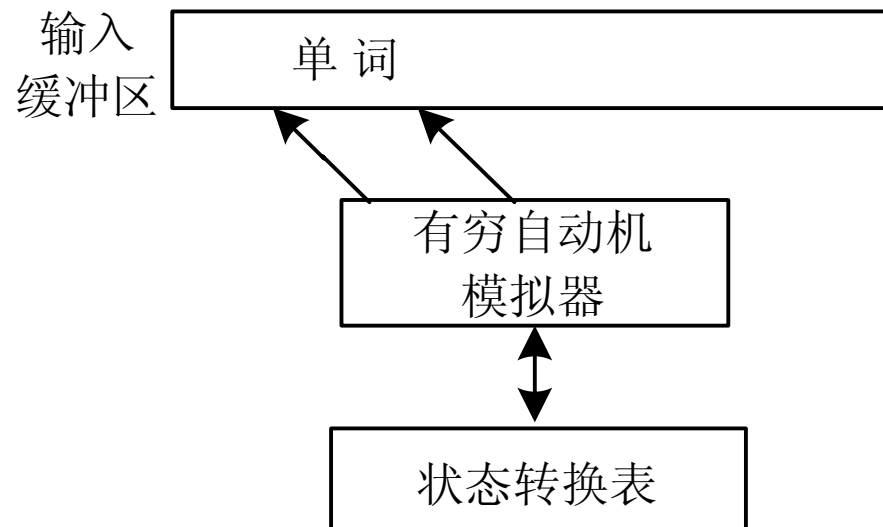


图 3.24 Lex生成的词法分析器结构

Lex的功能是根据Lex源程序生成状态转换矩阵和控制程序



# 三点说明

---

- 1) 以上是Lex的构造原理，虽然是原理性的，但据此就不难将Lex构造出来。
- 2) 所构造出来的Lex是一个通用的工具，用它它可以生成各种语言的词法分析程序，只需要根据不同的语言书写不同的LEX源文件就可以了。
- 3) Lex不但能自动生成词法分析器，而且也可以产生多种模式识别器及文本编辑程序等



# 本章小结

---

- 词法分析器接收表示源程序的“平滑字符流”，输出与之等价的单词序列；
- 单词被分成多个种类，并被表示成(种别，属性值)的二元组形式；
- 为了提高效率，词法分析器使用缓冲技术，而且在将字符流读入缓冲区时，是经过剔除注解、无用空白符等预处理后的结果；



# 本章小结

---

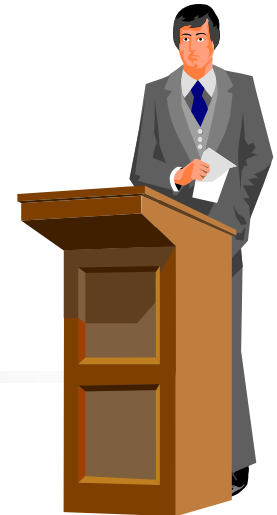
- 单词的识别相当于正则语言的识别;
- 词法的等价描述形式有正则文法、有穷状态自动机、正则表达式, 其中有穷状态自动机可以用状态转换图表示;
- 实现词法分析器时状态转换图是一个很好的设计工具, 根据该图, 容易构造出相应的分析程序;
- 使用恰当的形式化描述, 可以实现词法分析器的自动生成, Lex就是一种自动生成工具。



*School of Computer Science & Technology*  
*Harbin Institute of Technology*



# 第四章 自顶向下的 语法分析



**重点:** 自顶向下分析的基本思想, 预测分析器总体结构, 预测分析表的构造, 递归下降分析法基本思想, 简单算术表达式的递归下降分析器。

**难点:** FIRST 和 FOLLOW 集的求法, 对它们的理解以及在构造 LL(1) 分析表时的使用。递归子程序法中如何体现分析的结果。





# 第4章 自顶向下的语法分析

---

**4.1 语法分析概述**

**4.2 自顶向下的语法分析面临的问题  
与文法的改造**

**4.3 预测分析法**

**4.4 递归下降分析法**

**4.5 本章小结**

# 语法分析的功能和位置

■ **语法分析** (syntax analysis) 是编译程序的核心部分，其任务是检查词法分析器输出的单词序列是否是源语言中的句子，亦即是否符合源语言的语法规则。

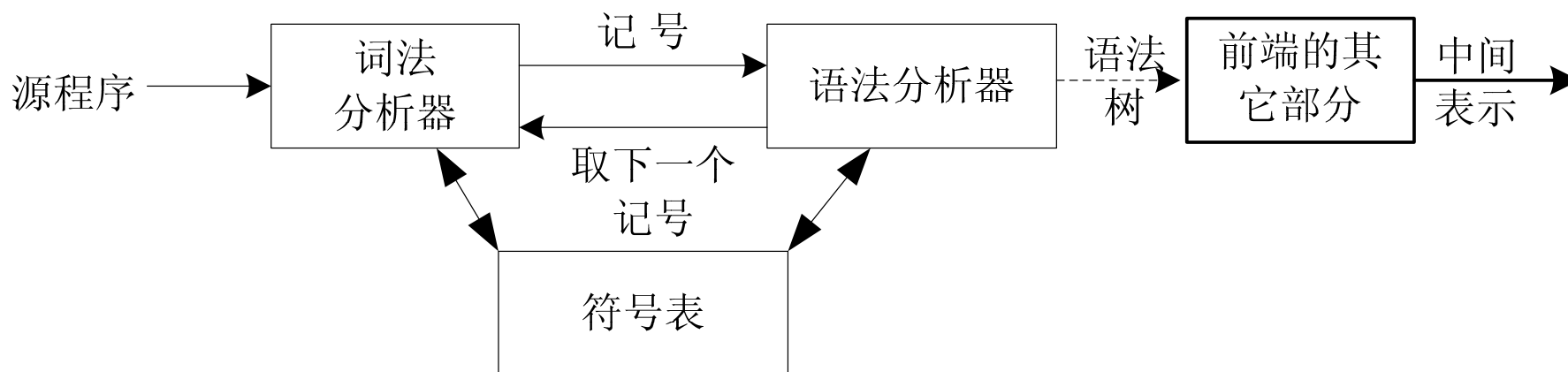


图4.1 语法分析器在编译器中的位置

# 4.1 语法分析概述

从文法产生语言的角度

- 自顶向下  
Top Down

递归子程序法

预测分析法 (LL (1))

从根开始，逐步为某语句构造一棵语法树

从自动机识别语言的角度

- 自底向上  
Bottom Up

算符优先分析法

相反，将一句子归约为开始符号

LR (0)、SLR (1)、LR (1)、LALR (1)

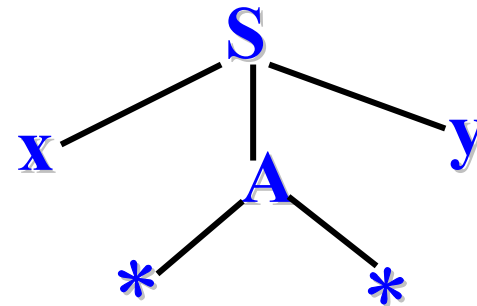
问题：解决确定性问题！

假定文法是压缩的：即删除了单位产生式和无用产生式。

## 4.2 自顶向下的语法分析面临的问题与文法的改造

- 自顶向下语法分析的基本思想
  - 从文法的开始符号出发，寻求所给的输入符号串的一个最左推导。
  - 从树根S开始，构造所给输入符号串的语法树
- 例:设有  $G: S \rightarrow xAy \quad A \rightarrow **|*$ , 输入串:  $x**y$

$S \Rightarrow xAy$   
 $\Rightarrow x**y$



## 4.2.1 自顶向下分析面临的问题

### 1. 二义性问题

- 对于文法  $G$ ，如果  $L(G)$  中存在一个具有两棵或两棵以上分析树的句子，则称  $G$  是二义性的。也可以等价地说：如果  $L(G)$  中存在一个具有两个或两个以上最左(或最右)推导的句子，则  $G$  是二义性文法。
- 如果一个文法  $G$  是二义性的，假设  $w \in L(G)$  且  $w$  存在两个最左推导，则在对  $w$  进行自顶向下的语法分析时，语法分析程序将无法确定采用  $w$  的哪个最左推导。
- $G_{exp}$ :  
$$E \rightarrow E+T \mid E-T \mid T$$
$$T \rightarrow T * F \mid T / F \mid F$$
$$F \rightarrow F \uparrow P \mid P$$
$$P \rightarrow c \mid id \mid (E)$$

解决办法：改造文法，引入新的文法变量

## 4.2.1 自顶向下分析面临的问题

### 2.回溯问题

- 文法中每个语法变量A的产生式右部称为A的候选式，如果A有多个候选式存在公共前缀，则自顶向下的语法分析程序将无法根据当前输入符号准确地选择用于推导的产生式，只能试探。当试探不成功时就需要退回到上一步推导，看A是否还有其它的候选式，这就是回溯(backtracking)。
- Ge:  $E \rightarrow T \quad E \rightarrow E+T \quad E \rightarrow E-T \quad T \rightarrow F \quad T \rightarrow T * F$   
 $T \rightarrow T / F \quad F \rightarrow (E) \quad F \rightarrow id$
- 例如：考虑为输入串id+id\*id建立最左推导

## 4.2.1 自顶向下分析面临的问题

### 2. 回溯问题

- $E \Rightarrow T$  (4.1)

- $E \Rightarrow T \Rightarrow F$  (4.2)

- $E \Rightarrow T \Rightarrow F \Rightarrow (E)$  (4.3)

- $E \Rightarrow T \Rightarrow F \Rightarrow \text{id}$  (4.4)

- $E \Rightarrow T \Rightarrow T^* F$  (4.5)

- .....

4.2.2节我们将采用提取左因子的方法来改造文法，以便减少推导过程中回溯现象的发生，当然，单纯通过提取左因子无法彻底避免回溯现象的发生。



## 4.2.1 自顶向下分析面临的问题

### 3. 左递归引起的无穷推导问题

- 假设 $A$ 是文法 $G$ 的某个语法变量，如果存在推导 $A \xRightarrow{+} \alpha A \beta$ ，则称文法 $G$ 是递归的，当 $\alpha = \varepsilon$ 时称之为左递归；如果 $A \xRightarrow{+} \alpha A \beta$ 至少需要两步推导，则称文法 $G$ 是间接递归的，当 $\alpha = \varepsilon$ 时称之为间接左递归；如果文法 $G$ 中存在形如 $A \rightarrow \alpha A \beta$ 的产生式，则称文法 $G$ 是直接递归的，当 $\alpha = \varepsilon$ 时称之为直接左递归。
- **Ger:**  $E \rightarrow T \quad E \rightarrow E+T \quad E \rightarrow E-T \quad T \rightarrow F \quad T \rightarrow T * F$   
 $T \rightarrow T / F \quad F \rightarrow (E) \quad F \rightarrow \text{id}$
- 考虑为输入串 $\text{id}+\text{id}*\text{id}$ 建立一个最左推导
- $E \Rightarrow E+T \Rightarrow E+T+T \Rightarrow E+T+T+T \Rightarrow \dots$



## 4.2.2 对上下文无关文法的改造

### 1. 消除二义性

- 改造的方法就是通过引入新的语法变量等，使文法含有更多的信息。其实，许多二义性文法是由于概念不清，即语法变量的定义不明确导致的，此时通过引入新的语法变量即可消除文法的二义性。
- $\langle stmt \rangle \rightarrow \text{if } \langle expr \rangle \text{ then } \langle stmt \rangle$
- $\quad \quad \quad | \text{if } \langle expr \rangle \text{ then } \langle stmt \rangle \text{ else } \langle stmt \rangle$
- $\quad \quad \quad | \text{other} \quad \quad \quad (4.7)$
- 根据if语句中else与then配对情况将其分为配对的语句和不配对的语句两类。上述if语句的文法没有对这两个不同的概念加以区分，只是简单地将它们都定义为 $\langle stmt \rangle$ ，从而导致该文法是二义性的。



## 4.2.2 对上下文无关文法的改造

引入语法变量 *<unmatched\_stmt>* 来表示不配对语句， *<matched\_stmt>* 表示配对语句

- $\langle stmt \rangle \rightarrow \langle matched\_stmt \rangle$   
                  |  $\langle unmatched\_stmt \rangle$
- $\langle matched\_stmt \rangle \rightarrow \text{if } \langle expr \rangle \text{ then}$   
                   $\langle matched\_stmt \rangle \text{ else } \langle matched\_stmt \rangle$   
                  | other
- $\langle unmatched\_stmt \rangle \rightarrow \text{if } \langle expr \rangle \text{ then } \langle stmt \rangle$   
                  |  $\text{if } \langle expr \rangle \text{ then } \langle matched\_stmt \rangle \text{ else}$   
                   $\langle unmatched\_stmt \rangle$

## 4.2.2 对上下文无关文法的改造

### 2.消除左递归

- 直接左递归的消除(转换为右递归)
- 引入新的变量 $A'$ ，将左递归产生式 $A \rightarrow A \alpha \mid \beta$ 替换为 $A \rightarrow \beta A' \quad A' \rightarrow \alpha A' \mid \varepsilon$
- $E \rightarrow E+T \mid T \quad T \rightarrow T * F \mid F \quad F \rightarrow (E) \mid \text{id}$ 替换为：
- $E \rightarrow TE' \quad E' \rightarrow +TE' \mid \varepsilon \quad T \rightarrow FT' \quad T' \rightarrow *FT' \mid \varepsilon$   
 $F \rightarrow (E) \mid \text{id}$
- 一般地，假设文法 $G$ 中的语法变量 $A$ 的所有产生式如下： $A \rightarrow A \alpha_1 \mid A \alpha_2 \mid \dots \mid A \alpha_n \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_m$   
其中， $\beta_i (i=1,2,\dots,m)$ 不以 $A$ 打头。则用如下的产生式代替 $A$ 的所有产生式即可消除其直接左递归： $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_m A' \quad A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_n A' \mid \varepsilon$

## 4.2.2 对上下文无关文法的改造

算法4.1 消除左递归。

输入：不含循环推导和  $\varepsilon$ -产生式的文法  $G$ ;

输出：与  $G$  等价的无左递归文法;

步骤:

1. 将  $G$  的所有语法变量排序(编号), 假设排序后的语法变量记为  $A_1, A_2, \dots, A_n$ ;
2. for  $i \leftarrow 1$  to  $n$  {
3.     for  $j \leftarrow 1$  to  $i-1$  {
4.         用产生式  $A_i \rightarrow \alpha_1 \beta \mid \alpha_2 \beta \mid \dots \mid \alpha_k \beta$  代替每个形如  $A_i \rightarrow A_j \beta$  的产生式,
- 其中,  $A_j \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_k$  是所有的当前  $A_j$  产生式;
5.     }
6.     消除  $A_i$  产生式中的所有直接左递归
7. }

## 4.2.2 对上下文无关文法的改造

### ■ 3. 提取左因子

- 对每个语法变量 $A$ ，找出它的两个或更多候选式的最长公共前缀 $\alpha$ 。如果 $\alpha \neq \varepsilon$ ，则用下面的产生式替换所有的 $A$ 产生式

$A \rightarrow \alpha \beta_1 | \alpha \beta_2 | \dots | \alpha \beta_n | \gamma_1 | \gamma_2 | \dots | \gamma_n$ ，其中  $\gamma_1, \gamma_2, \dots, \gamma_n$  表示所有不以  $\alpha$  开头的候选式：

$$A \rightarrow \alpha A' | \gamma_1 | \gamma_2 | \dots | \gamma_n$$

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

- 其中， $A'$ 是新引入的语法变量。反复应用上述变换，直到任意语法变量都没有两个候选式具有公共前缀为止。请读者自行给出这个变换的算法。



## 4.2.3 LL(1)文法

问题：什么样的文法对其句子才能进行确定的自顶向下分析？

- 确定的自顶向下分析首先从文法的开始符号出发，每一步推导都根据当前句型的最左语法变量 $A$ 和当前输入符号 $a$ ，选择 $A$ 的某个候选式 $\alpha$ 来替换 $A$ ，并使得从 $\alpha$ 推导出的第一个终结符恰好是 $a$ 。
- 当 $A$ 有多个候选式时，当前选中的候选式必须是惟一的。
- 第一个终结符是指符号串的第一个符号，并且是终结符号，可以称为首终结符号。在自顶向下的分析中，它对选取候选式具有重要的作用。为此引入首符号集的概念。



## 4.2.3 LL(1)文法

1. 假设  $\alpha$  是文法  $G=(V, T, P, S)$  的符号串, 即  
 $\alpha \in (V \cup T)^*$ , 从  $\alpha$  推导出的串的首符号集记作

$\text{FIRST}(\alpha): \xRightarrow{*}$

$\text{FIRST}(\alpha) \xRightarrow{*} \{a \mid \alpha \xRightarrow{*} a\beta, a \in T, \beta \in (V \cup T)^*\}$ 。

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

3. 如果文法  $G$  中的所有  $A$  产生式为  $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_m$ ,

且  $\varepsilon \notin \text{FIRST}(\alpha_1) \cup \text{FIRST}(\alpha_2) \cup \dots \cup \text{FIRST}(\alpha_n)$

且对  $\forall i, j, 1 \leq i, j \leq m; i \neq j$ , 均有

$\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \emptyset$  成立, 则可以对  $G$  的句子  
进行确定的自顶向下分析 26



## 4.2.3 LL(1)文法

---

如果存在 $A \rightarrow \varepsilon$ 这样的产生式，则需定义FOLLOW( $A$ )

$\forall A \in V$ 定义 $A$ 的后续符号集为：

1. FOLLOW( $A$ ) =  $\{a | S \xRightarrow{*} \alpha A a \beta, a \in T, \alpha, \beta \in (V \cup T)^*\}$

2. 如果 $A$ 是某个句型的最右符号，则将结束符#  
添加到FOLLOW( $A$ )中

3. 如果 $\alpha_j \xRightarrow{*} \varepsilon$ ，则如果对 $\forall i (1 \leq i \leq m; i \neq j)$ ，  
FIRST( $\alpha_i$ )  $\cap$  FOLLOW( $A$ ) =  $\emptyset$ 均成立，则可以对 $G$   
的句子进行确定的自顶向下分析





## 4.2.3 LL(1)文法

---

如果 $G$ 的任意两个具有相同左部的产生式 $A \rightarrow \alpha | \beta$ 满足下列条件:

1. 如果 $\alpha$ 、 $\beta$ 均不能推导出 $\varepsilon$ , 则

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

2.  $\alpha$ 和 $\beta$ 至~~多~~<sup>\*</sup>有一个能推导出 $\varepsilon$ ;

3. 如果 $\beta \Rightarrow \varepsilon$ , 则 $\text{FIRST}(\alpha) \cap \text{FOLLOW}(A) = \emptyset$

则称 $G$ 为 $LL(1)$ 文法。

第一个 $L$ 代表从左向右扫描输入符号串, 第二个 $L$ 代表产生最左推导, 1代表在分析过程中执行每步推导都要向前查看一个输入符号

# LL(1)文法的判定

算法4.2 计算FIRST(X)。

输入：文法  $G=(V, T, P, S)$ ,  $X \in (V \cup T)$ ;

输出：FIRST(X);

步骤：

1. FIRST(X) =  $\emptyset$ ;
2. if  $(X \in T)$  then FIRST(X) := {X} ;
3. if  $X \in V$  then  
begin
4. if  $(X \rightarrow \varepsilon \notin P)$  then FIRST(X) := FIRST(X)  $\cup$  {a |  $X \rightarrow a... \in P$ };
5. if  $(X \rightarrow \varepsilon \in P)$  then FIRST(X) := FIRST(X)  $\cup$  {a |  $X \rightarrow a... \in P$ }  $\cup$  {  $\varepsilon$  };
- end
6. 对  $\forall X \in V$ , 重复如下的过程7-10, 直到所有FIRST集不变为止。
7. if  $(X \rightarrow Y... \in P \text{ and } Y \in V)$  then FIRST(X) :=  
FIRST(X)  $\cup$  (FIRST(Y) - {  $\varepsilon$  });
8. if  $(X \rightarrow Y_1...Y_n \in P \text{ and } Y_1...Y_{i-1} \overset{*}{\Rightarrow} \varepsilon)$  then
9. for  $k=2$  to  $i$  do FIRST(X) := FIRST(X)  $\cup$  (FIRST( $Y_k$ ) - {  $\varepsilon$  });
10. if  $Y_1...Y_n \overset{*}{\Rightarrow} \varepsilon$  then FIRST(X) := FIRST(X)  $\cup$  {  $\varepsilon$  };



# LL(1)文法的判定

---

算法4.3 计算FIRST( $\alpha$ )。

输入：文法 $G=(V, T, P, S)$ ,  $\alpha \in (V \cup T)^*$ ,  $\alpha = X_1 \dots X_n$ ;

输出：FIRST( $\alpha$ );

步骤：

1. 计算FIRST( $X_1$ );
2. FIRST( $\alpha$ ) := FIRST( $X_1$ ) - {  $\varepsilon$  };
3.  $k := 1$ ;
4. while (  $\varepsilon \in \text{FIRST}(X_k)$  and  $k < n$  ) do begin
5.     FIRST( $\alpha$ ) := FIRST( $\alpha$ )  $\cup$  (FIRST( $X_{k+1}$ ) - {  $\varepsilon$  });
6.      $k := k + 1$  end
7. if ( $k = n$  and  $\varepsilon \in \text{FIRST}(X_k)$ ) then  
    FIRST( $\alpha$ ) := FIRST( $\alpha$ )  $\cup$  {  $\varepsilon$  };



## 例 表达式文法的语法符号的FIRST 集

$\text{FIRST}(F) = \{ (, \text{id} \}$

$\text{FIRST}(T) = \text{FIRST}(F) = \{ (, \text{id} \}$

$\text{FIRST}(E) = \text{FIRST}(T) = \{ (, \text{id} \}$

$\text{FIRST}(E') = \{ +, \varepsilon \}$

$\text{FIRST}(T') = \{ *, \varepsilon \}$

$\text{FIRST}(+) = \{ + \}, \text{FIRST}(*) = \{ * \}$

$\text{FIRST}( ( ) ) = \{ ( \}$

$\text{FIRST}( ) ) = \{ ) \}$

$\text{FIRST}(\text{id}) = \{ \text{id} \}$

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid \text{id}$



# LL(1)文法的判定

算法4.4 计算FOLLOW集。

输入：文法 $G=(V, T, P, S)$ ,  $A \in V$ ;

输出：FOLLOW( $A$ );

步骤：

1. 对 $\forall X \in V$ , FOLLOW( $S$ ) :=  $\emptyset$ ;
2. FOLLOW( $S$ ) :=  $\{\#\}$ , #为句子的结束符;
3. 对 $\forall X \in V$ , 重复下面的第4步到第5步, 直到所有FOLLOW集不变为止。
4. 若 $A \rightarrow \alpha B \beta \in P$ , 则  
FOLLOW( $B$ ) := FOLLOW( $B$ )  $\cup$  FIRST( $\beta$ ) -  $\{\varepsilon\}$ ;
5. 若 $A \rightarrow \alpha B$ 或 $A \rightarrow \alpha B \beta \in P$ , 且 $\beta \xRightarrow{\varepsilon}$ ,  $A \neq B$ , 则  
FOLLOW( $B$ ) := FOLLOW( $B$ )  $\cup$  FOLLOW( $A$ );

## 例 表达式文法的语法变量的 FOLLOW 集

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

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

$F \rightarrow (E) \mid id$

$FIRST(F) = \{ (, id \}$

$FIRST(T) = FIRST(F) = \{ (, id \}$

$FIRST(E) = FIRST(T) = \{ (, id \}$

$FIRST(E') = \{ +, \varepsilon \}$

$FIRST(T') = \{ *, \varepsilon \}$

$FOLLOW(E) = \{ \#, ) \}$

$FOLLOW(E') = FOLLOW(E) = \{ \#, ) \}$

$FOLLOW(T) = FIRST(E') \cup FOLLOW(E) \cup FOLLOW(E') = \{ +, ), \# \}$

$FOLLOW(T') = FOLLOW(T) = \{ +, ), \# \}$

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

# 表达式文法是 LL(1) 文法

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

考察

- $E'$  :  $+$  不在  $FOLLOW(E') = \{ ), \# \}$
- $T'$  :  $*$  不在  $FOLLOW(T') = \{ +, ), \# \}$
- $F$ :  $($  和  $id$  不同



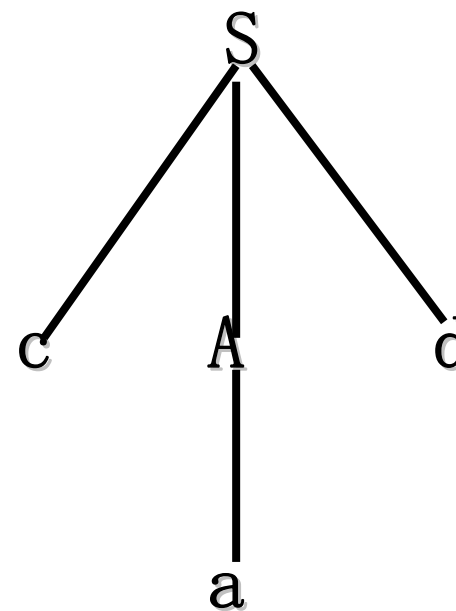
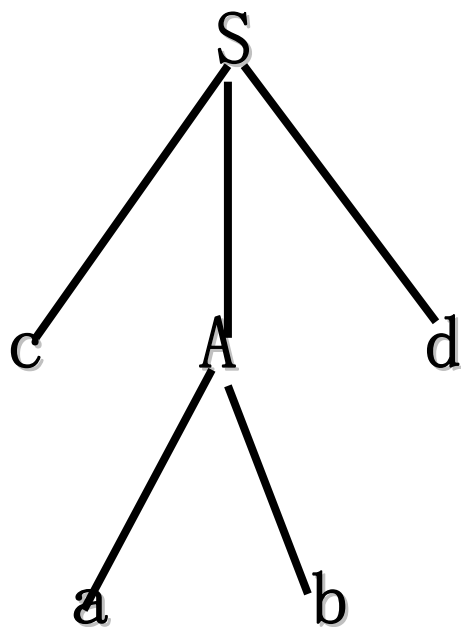
例 对文法

■  $S \rightarrow cAd$

■  $A \rightarrow ab|a$

输入 **cad** 的分析

# 非 LL(1)文法的不确定性







# 不确定性的解决方法

---

## 1) 采用回溯算法

- 过于复杂，效率低下

## 2) 改写文法

- 将非LL(1)文法改写为等价的LL(1)文法
- 无法改写时:
  - 增加其它的判别因素
  - 文法过于复杂，无法用自顶向下方法处理

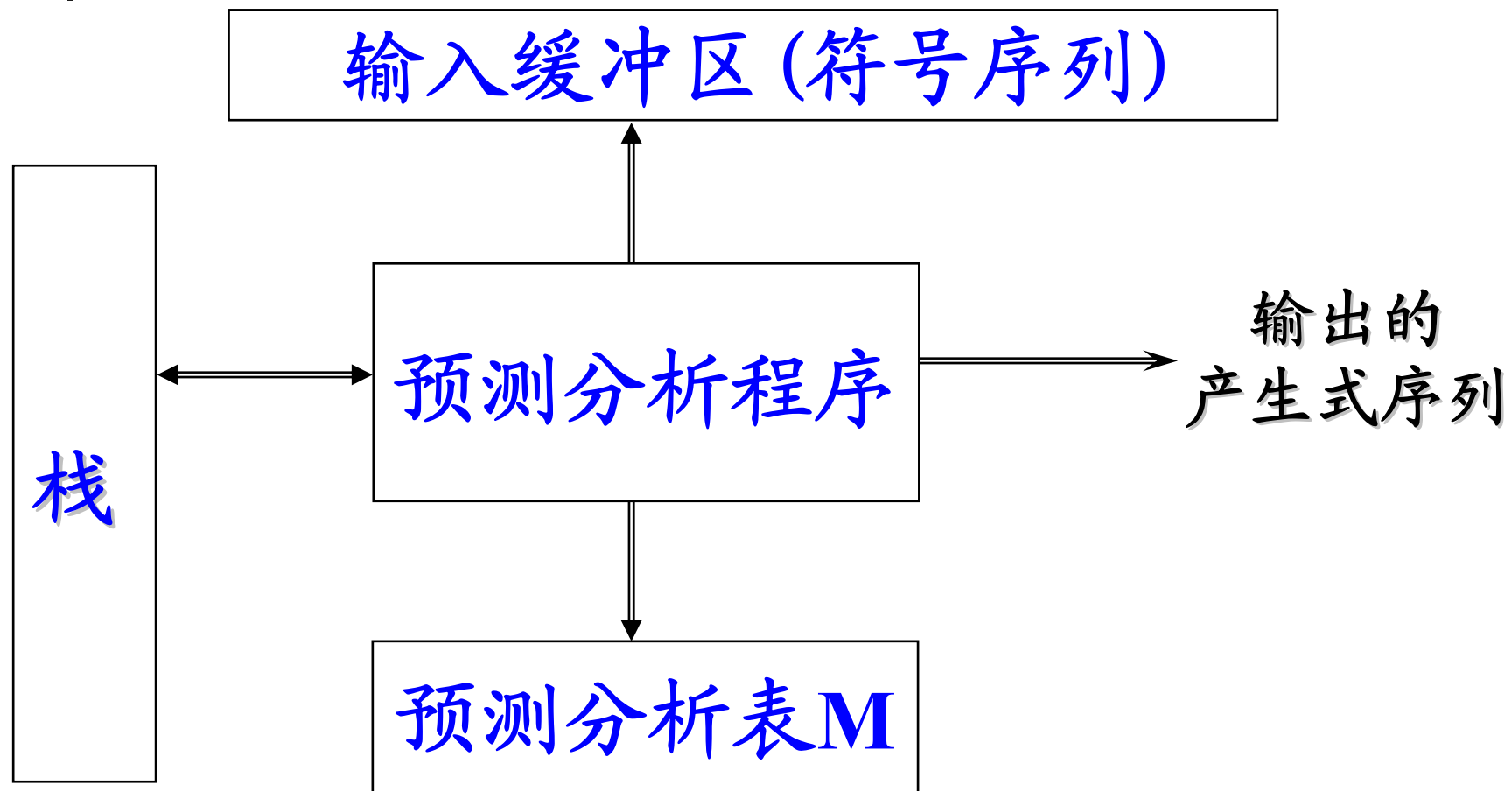


## 4.3 预测分析法

---

- 系统维持一个分析表和一个分析栈，根据当前扫描到的符号，选择当前语法变量（处于栈顶）的候选式进行推导——希望找到相应输入符号串的最左推导。
- 一个通用的控制算法
- 一个分析栈，#为栈底符号
- 一个输入缓冲区，#为输入串结束符
- 一个统一形式的分析表M
  - 不同语言使用内容不同的分析表

## 4.3.1 预测分析器的构成





# 系统的执行与特点

---

- 在系统启动时，输入指针指向输入串的第一个字符，分析栈中存放着栈底符号#和文法的开始符号。
- 根据栈顶符号A和读入的符号a，查看分析表M, 以决定相应的动作。
- 优点：
  - 1) 效率高
  - 2) 便于维护、自动生成
- 关键——分析表M的构造

# 预测分析程序的总控程序

算法4.5 预测分析程序的总控程序。

输入：输入串 $w$ 和文法 $G=(V, T, P, S)$ 的分析表 $M$ ;

输出：如果 $w$ 属于 $L(G)$ ，则输出 $w$ 的最左推导，否则报告错误;

步骤:

1. 将栈底符号 $\#$ 和文法开始符号 $S$ 压入栈中;
2. repeat
3.      $X :=$ 当前栈顶符号;
4.      $a :=$ 当前输入符号;
5.     if  $X \in T \cup \{\#\}$  then
6.         if  $X = a$  then
7.             {if  $X \neq \#$  then begin
8.                 将 $X$ 弹出栈;
9.                 前移输入指针
10.             end}



# 预测分析程序的总控程序

---

```
11.                                     else error
12.                                     else
13.                                     if  $M[X, a]=Y_1Y_2...Y_k$  then begin
14.                                     将 $X$ 弹出栈;
15.                                     依次将 $Y_k, ..., Y_2, Y_1$ 压入栈;
16.                                     输出产生式 $X \rightarrow Y_1Y_2...Y_k$ 
17.                                     end
18.                                     else error
19. until  $X=\#$ 
```



## 例4.10 考虑简单算术表达式文法的实现

---

$\text{FOLLOW}(E') = \{ ), \# \}$

$\text{FOLLOW}(T') = \{ +, ), \# \}$

$\text{FIRST}(TE') = \{ (, \text{id} \}$

$\text{FIRST}(+TE') = \{ + \}$

$\text{FIRST}(FT') = \{ (, \text{id} \}$

$\text{FIRST}(*FT') = \{ * \}$

$\text{FIRST}((E)) = \{ ( \}$

$\text{FIRST}(\text{id}) = \{ \text{id} \}$

$E \rightarrow TE' \quad E' \rightarrow +TE' \mid \varepsilon \quad T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon \quad F \rightarrow (E) \mid \text{id}$

# 简单算术表达式文法的预测分析表

非终结符	输入符号					
	id	+	*	(	)	#
<b>E</b>	$\rightarrow \mathbf{TE'}$			$\rightarrow \mathbf{TE'}$		
<b>E'</b>		$\rightarrow \mathbf{+TE}$			$\rightarrow \boldsymbol{\varepsilon}$	$\rightarrow \boldsymbol{\varepsilon}$
<b>T</b>	$\rightarrow \mathbf{FT'}$			$\rightarrow \mathbf{FT'}$		
<b>T'</b>		$\rightarrow \boldsymbol{\varepsilon}$	$\rightarrow \mathbf{*FT'}$		$\rightarrow \boldsymbol{\varepsilon}$	$\rightarrow \boldsymbol{\varepsilon}$
<b>F</b>	$\rightarrow \mathbf{id}$			$\rightarrow \mathbf{(E)}$		



# 对输入串id+id\*id进行分析的过程

(在黑板上同时画出语法树)

栈	输入缓冲区	输出
#E	id+id*id#	
#E' T	id+id*id#	$E \rightarrow TE'$
#E' T' F	id+id*id#	$T \rightarrow FT'$
#E' T' id	id+id*id#	$F \rightarrow id$
#E' T'	+id*id#	
#E'	+id*id#	$T' \rightarrow \varepsilon$
#E' T+	+id*id#	$E' \rightarrow +TE'$
#E' T	id*id#	



#E' T

id\*id#

#E' T' F

id\*id#

$T \rightarrow FT'$

#E' T' id

id\*id#

$F \rightarrow id$

#E' T'

\*id#

#E' T' F\*

\*id#

$T' \rightarrow *FT'$

#E' T' F

id#

#E' T' id

id#

$F \rightarrow id$

#E' T'

#

#E'

#

$T' \rightarrow \varepsilon$

#

#

$E' \rightarrow \varepsilon$

输出的产生式序列形成了最左推导对应的分析树



## 4.3.2 预测分析表的构造算法

算法4.6 预测分析表( $LL(1)$ 分析表)的构造算法。

输入：文法 $G$ ;

输出：分析表 $M$ ;

步骤:

1. 对 $G$ 中的任意一个产生式 $A \rightarrow \alpha$ , 执行第2步和第3步;
2. for  $\forall a \in \text{FIRST}(\alpha)$ , 将 $A \rightarrow \alpha$ 填入 $M[A, a]$ ;
3. if  $\varepsilon \in \text{FIRST}(\alpha)$  then  $\forall a \in \text{FOLLOW}(A)$ , 将 $A \rightarrow \alpha$ 填入 $M[A, a]$ ;  
if  $\varepsilon \in \text{FIRST}(\alpha) \&\# \in \text{FOLLOW}(A)$  then 将 $A \rightarrow \alpha$ 填入 $M[A, \#]$ ;
4. 将所有无定义的 $M[A, b]$ 标上出错标志。



# 预测分析法的实现步骤

---

1. 构造文法

2. 改造文法：消除二义性、消除左递归、提取左因子

3. 求每个候选式的FIRST集和变量的FOLLOW集

4. 检查是不是 LL(1) 文法

若不是 LL(1),说明文法的复杂性超过自顶向下方法的分析能力, 需要附加新的“信息”

5. 构造预测分析表

6. 实现预测分析器



## 4.3.3 预测分析中错误的处理

- 对语法变量 $A$ ，如果 $M[A,a]$ 无定义，并且 $a$ 属于 $FOLLOW(A)$ ，则增加 $M[A,a]$ 为“同步点”(synch)，同步记号选择方法如下：
  - 把 $FOLLOW(A)$ 的所有符号放入语法变量 $A$ 的同步记号集合中。
  - 把高层结构的开始符号加到低层结构的同步记号集合中。
  - 把 $FIRST(A)$ 的符号加入 $A$ 的同步记号集合。
  - 如果语法变量可以产生空串，若出错时栈顶是这样的语法变量，则可以使用产生空串的产生式。
  - 如果符号在栈顶而不能匹配，则弹出此符号。

## 4.4 递归下降分析法—

### 一个设想

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

$$F \rightarrow (E) \mid \text{id}$$

1. 对应每个变量设置一个处理子程序:

$$A \rightarrow X_1 X_2 \dots X_k \dots X_n$$

(1) 当遇到 $X_k$ 是终极符号时直接进行匹配;

(2) 当遇到 $X_k$ 是语法变量时就调用 $X$ 对应的处理子程序.

2. 要求处理子程序是可以递归调用的



## 4.4.1 递归下降分析法的基本思想

---

例4.14 对于产生式 $E' \rightarrow +TE'$ ，与 $E'$ 对应的子程序可以按如下方式来编写：

```
procedure  $E'$ 
begin
    match('+');
     $T$ ;           /*调用识别 $T$ 的过程*/
     $E'$            /*调用识别 $E'$ 的过程*/
end;
```



## 4.4.1 递归下降分析法的基本思想

---

其中，服务子程序 $match$ 用来匹配当前的输入记号，其代码为：

```
procedure match(t:token);  
begin  
    if lookhead=t then  
        lookhead:=nexttoken;  
    else error          /*调用出错处理程序*/  
end;
```





## 4.4.2 语法图和递归子程序法

---

- 状态转换图（语法图）是非常有用的设计工具
- 语法分析器和词法分析器的状态转换图不同
  - 每个非终结符对应一个状态转换图，边上的标记是记号和非终结符
  - 记号上的转换意味着如果该记号是下一个输入符号，就应进行转换
  - 非终结符A上的转换是对与A对应的过程的调用

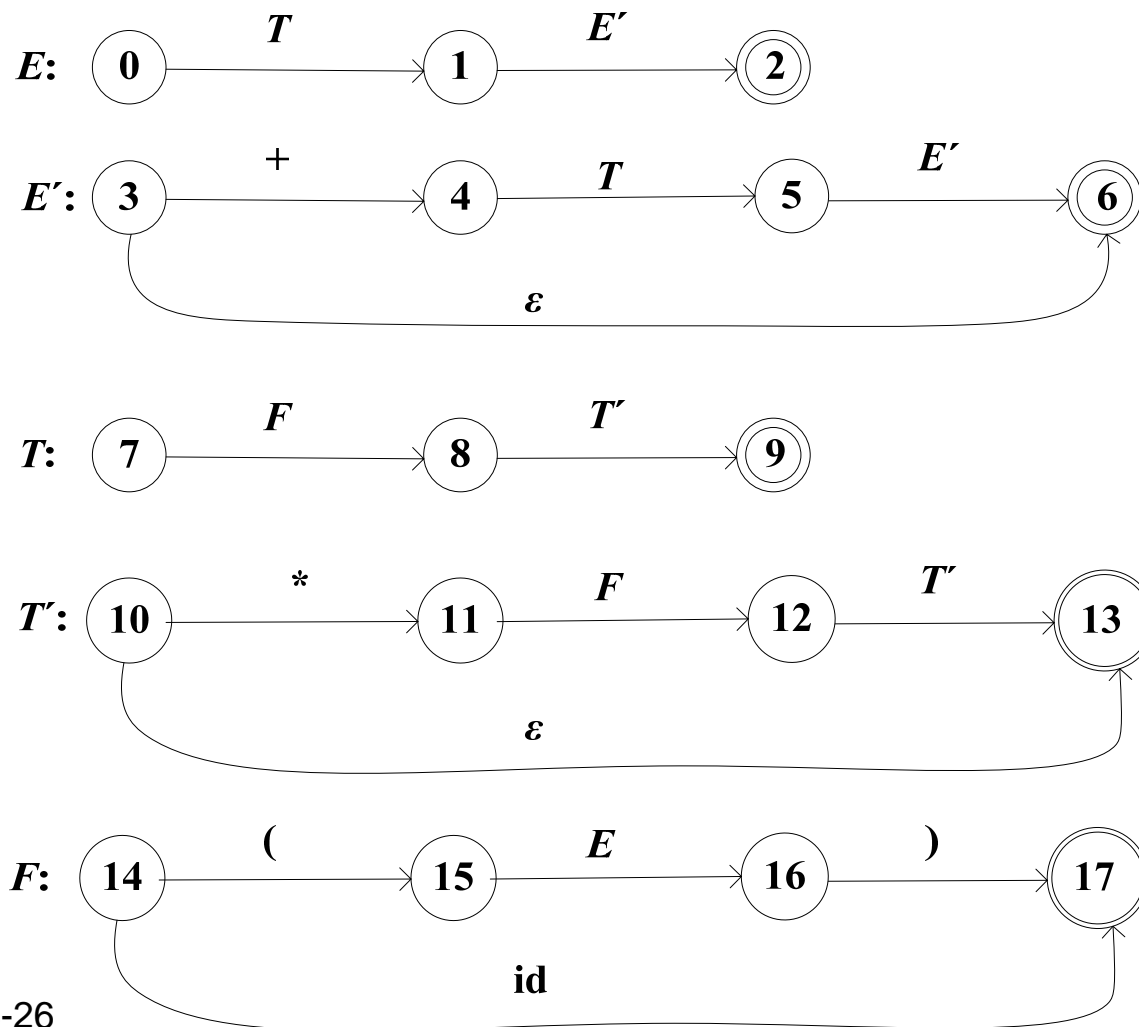


## 4.4.2 语法图和递归子程序法

---

- 从文法构造语法图，对每个非终结符A执行如下操作
  - 创建一个开始状态和一个终止状态（返回状态）
  - 对每个产生式 $A \rightarrow X_1 X_2 \dots X_n$ ，创建一条从开始状态到终止状态的路径，边上的标记分别为 $X_1$ ,  $X_2$ ,  $\dots$ ,  $X_n$

## 例4.15 简单表达式文法的语法图



$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$



### 4.4.3 基于语法图的语法分析器工作方式

---

- 初始时，分析器进入状态图的开始状态，输入指针指向输入符号串的第一个符号。
- 如果经过一些动作后，它进入状态 $s$ ，且从状态 $s$ 到状态 $t$ 的边上标记了终结符 $a$ ，此时下一个输入符又正好是 $a$ ，则分析器将输入指针向右移动一位，并进入状态 $t$ 。



### 4.4.3 基于语法图的语法分析器工作方式

- 另一方面，如果边上标记的是非终结符A，则分析器进入A的初始状态，但不移动输入指针。一旦到达A的终态，则立刻进入状态t，事实上，分析器从状态s转移到状态t时，它已经从输入符号串“读”了A（调用A对应的过程）。
- 最后，如果从s到t有一条标记为  $\varepsilon$  的边，那么分析器从状态s直接进入状态t而不移动输入指针。

## 4.4.4 语法图的化简与实现

### (1) 左因子提取

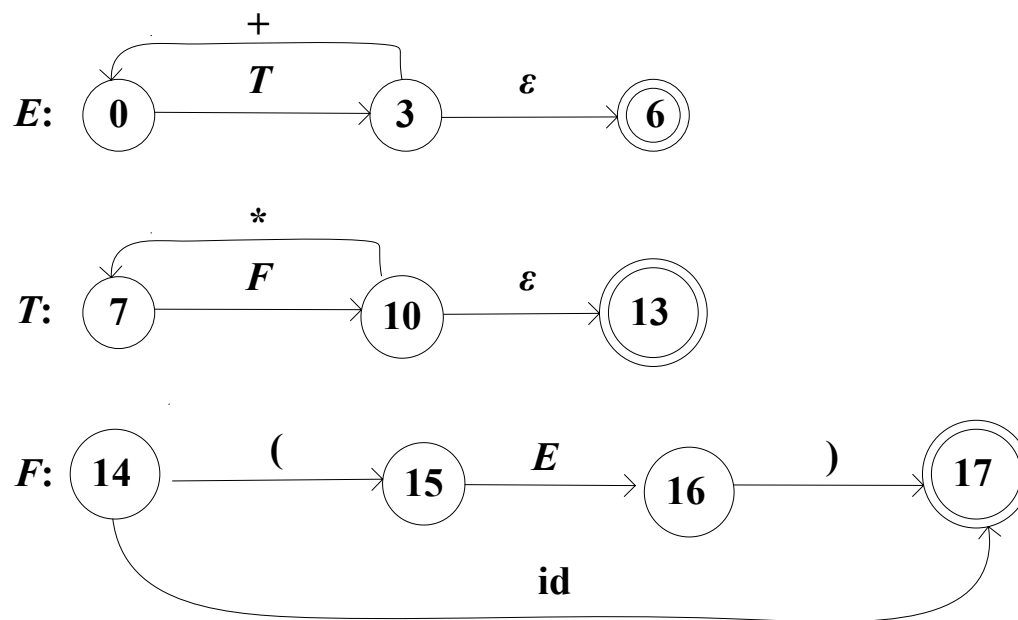
将形如 $A \rightarrow YX|YZ$ 的产生式替换为 $A \rightarrow Y(X|Z)$ ;

### (2) 右因子提取

将形如 $A \rightarrow YX|ZX$ 的产生式替换为 $A \rightarrow (Y|Z)X$ ;

### (3) 尾递归消除

将形如 $X \rightarrow YX|Z$ 的产生式替换为 $X \rightarrow Y^*Z$ 。





## 例4.16 简单算术表达式的语法分析器

- E 的子程序( $E \rightarrow T(+T)^*$ )

**procedure E;**

**begin**

**T;**                      T 的过程调用

**while lookahead='+' do**

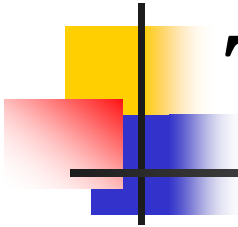
**begin**                      当前符号等于 + 时

**match('+');**              处理终结符 +

**T**                      T 的过程调用

**end**

**end;**                      lookahead: 当前符号



## T 的子程序 ( $T \rightarrow F (*F) *$ )

---

**procedure T;**

**begin**

**F;**                      F 的过程调用

**while lookahead='\*' then**

**begin**                      当前符号等于 \* 时

**match('\*');**              处理终结符 \*

**F**                      F 的递归调用

**end**

**end;**





# F 的子程序 ( $F \rightarrow (E)|id$ )

---

```
procedure F;  
begin  
  if lookahead='(' then  
    begin          当前符号等于 (  
      match('(');  处理终结符 (  
      E;           E 的递归调用  
      match(')');  处理终结符 )  
    end  
  else if lookahead=id then  
    match(id)      处理终结符 id  
  else error      出错处理  
end
```



# 主程序

---

begin

lookhead:=nexttoken;    调词法分析程序  
E                            E的过程调用

end

## 服务子程序

procedure match(t:token);

begin

if lookhead=t then

lookhead:=nexttoken

else error

出错处理程序

end;



## 4.4.5 递归子程序法的实现步骤

---

- 1) 构造文法;
- 2) 改造文法: 消除二义性、消除左递归、提取左因子;
- 3) 求每个候选式的FIRST集和语法变量的FOLLOW集;
- 4) 检查 $G$ 是不是  $LL(1)$  文法, 若 $G$ 不是  $LL(1)$ 文法, 说明文法 $G$ 的复杂性超过了自顶向下方法的分析能力, 需要附加新的“信息”;
- 5) 按照 $LL(1)$ 文法画语法图;
- 6) 化简语法图;
- 7) 按照语法图为每个语法变量设置一个子程序。



# 递归子程序法的优缺点分析

---

- 优点:

- 1) 直观、简单、可读性好
- 2) 便于扩充

- 缺点:

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

- 从递归子程序法及FIRST与FOLLOW集看如何进一步用好当前的输入符号?



# 本章小结

---

1. 自顶向下分析法和自底向上分析法分别寻找输入串的最左推导和最左归约
2. 自顶向下分析会遇到二义性问题、回溯问题、左递归引起的无穷推导问题，需对文法进行改造：消除二义性、消除左递归、提取公共左因子
3.  $LL(1)$ 文法是一类可以进行确定分析的文法，利用FIRST集和FOLLOW集可以判定某个上下文无关文法是否为 $LL(1)$ 文法



## 本章小结

---

4.  $LL(1)$ 文法可以用 $LL(1)$ 分析法进行分析。
5. 递归下降分析法根据各个候选式的结构为每个非终结符编写一个子程序。
6. 使用语法图可以方便地进行递归子程序的设计。



*School of Computer Science & Technology*  
*Harbin Institute of Technology*



## 第五章 自底向上的 语法分析

**重点：**自底向上分析的基本思想，算符优先分析法的基本思想，简单算符优先分析法。LR 分析器的基本构造思想，LR分析算法，规范句型活前缀及其识别器——DFA，LR (0)分析表的构造，SLR (1)分析表的构造，LR (1)分析表的构造。

**难点：**求FIRSTOP 和LASTOP，算符优先关系的确定，算符优先分析表的构造，素短语与最左素短语的概念。规范句型活前缀，LR (0)项目集闭包与项目集规范族，它们与句柄识别的关系，活前缀与句柄的关系，LR (1)项目集闭包与项目集规范族。



# 第5章 自底向上的语法分析

---

5.1 自底向上的语法分析概述

5.2 算符优先分析法

5.3 LR分析法

5.4 语法分析程序的自动生成工具Yacc

5.5 本章小结





## 5.1 自底向上的语法分析概述

---

### ■思想

- 从输入串出发，反复利用产生式进行归约，如果最后能得到文法的开始符号，则输入串是句子，否则输入串有语法错误。

### ■核心

- 寻找句型中的当前归约对象——“句柄”进行归约，用不同的方法寻找句柄，就可获得不同的分析方法

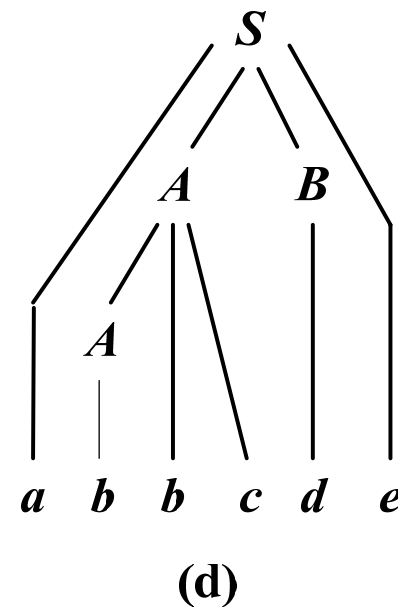
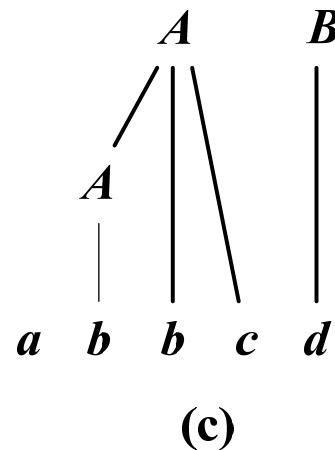
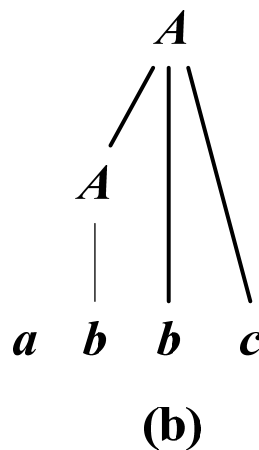
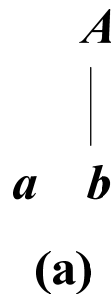
## 例5.1 一个简单的归约过程

■ 设文法G为:

$S \rightarrow aABe$      $A \rightarrow Abc|b$      $B \rightarrow d$

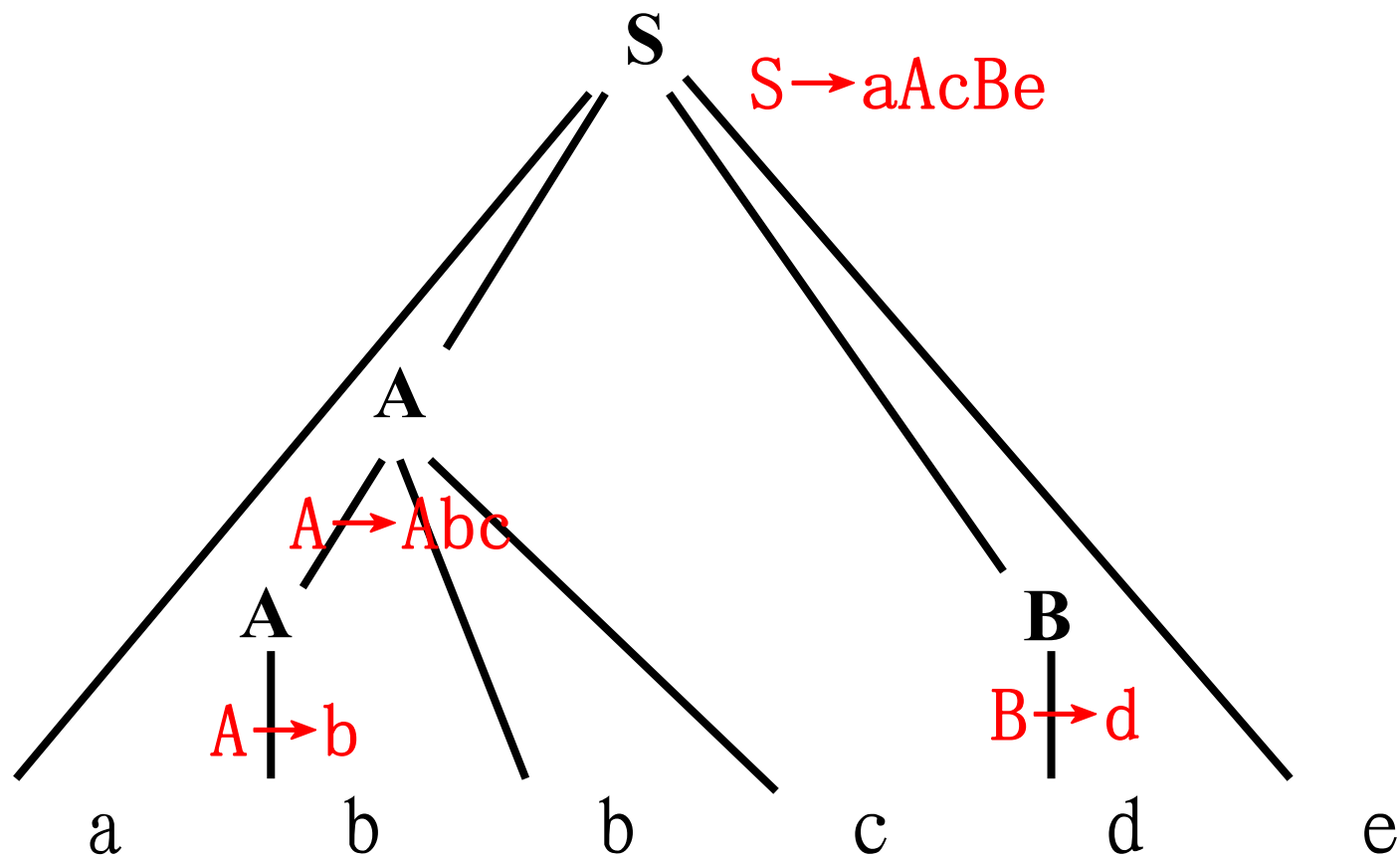
句子分析:

$a\underline{b}bcde$   
 $\Leftarrow a\underline{A}bcde$   
 $\Leftarrow a\underline{A}de$   
 $\Leftarrow a\underline{AB}e$   
 $\Leftarrow S$



语法树的形成过程

# 语法分析树的生成演示





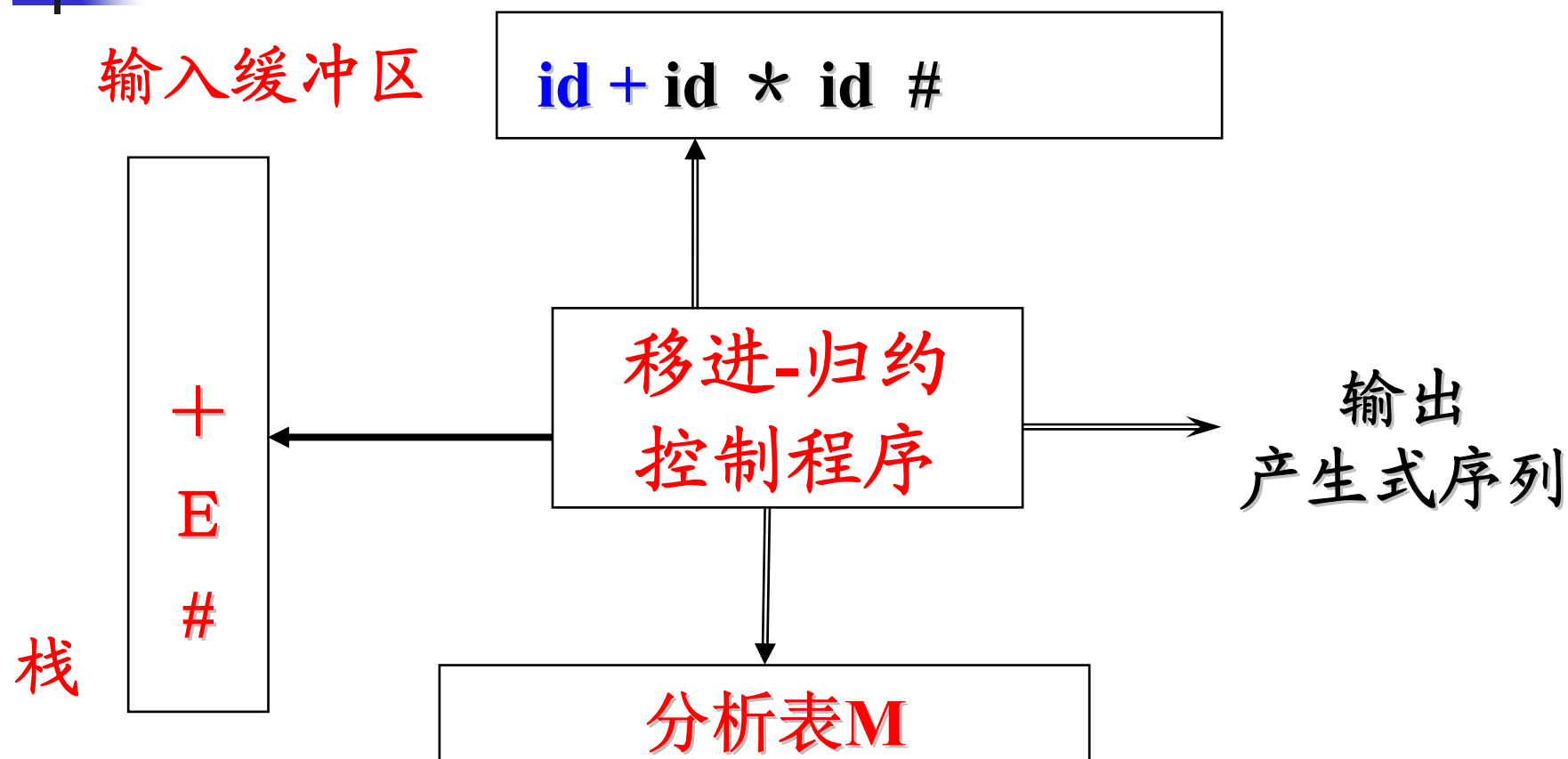
## 5.1.1 移进-归约分析

---

### ■ 系统框架

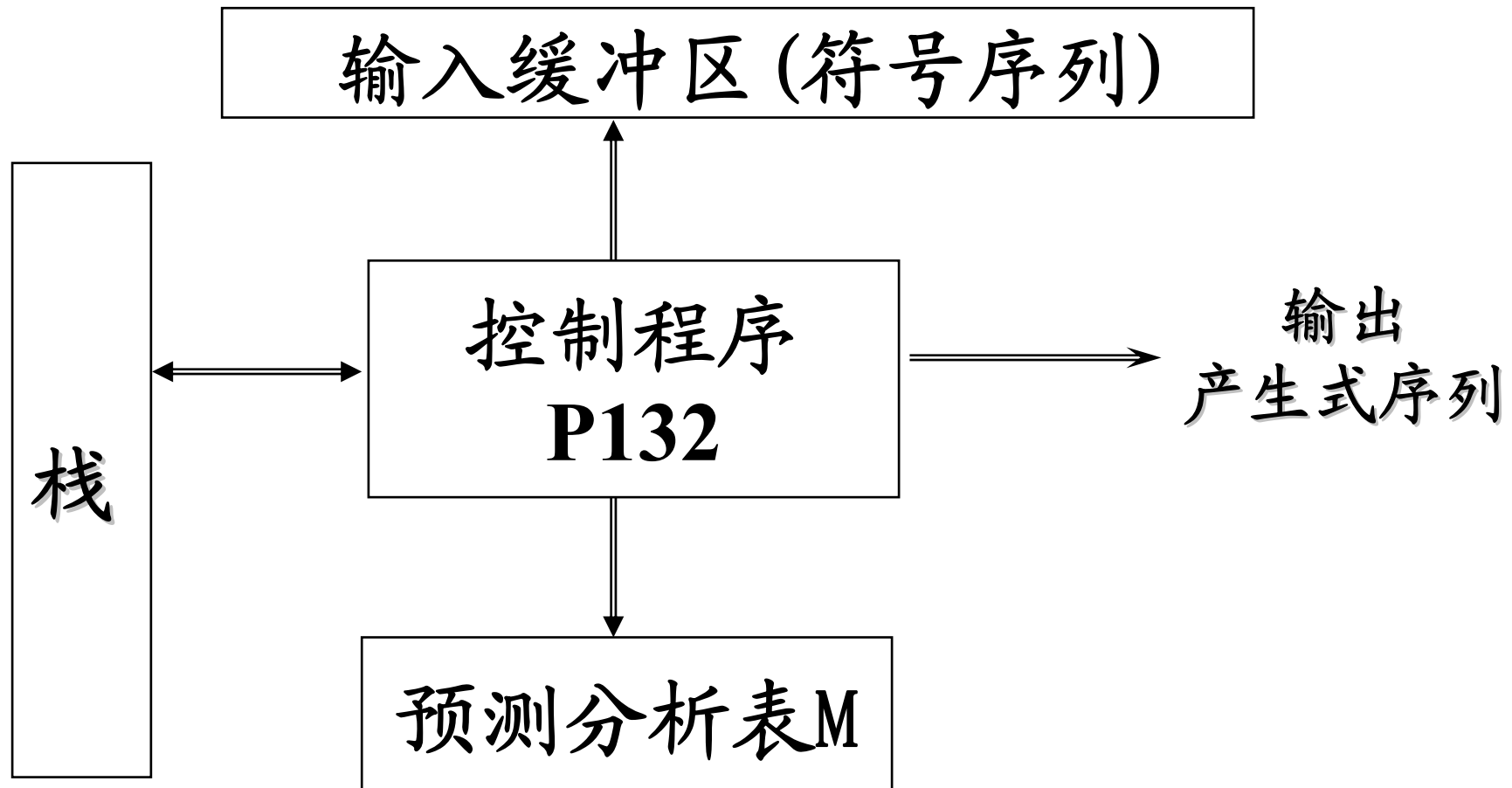
- 采用表驱动的方式实现
- 输入缓冲区：保存输入符号串
- 分析栈：保存语法符号——已经得到的那部分分析结果
- 控制程序：控制分析过程，输出分析结果——产生式序列
- 格局：栈+输入缓冲区剩余内容=“句型”

# 移进-归约语法分析器的总体结构



栈内容+输入缓冲区内容 = # “当前句型” #

# 与LL(1)的体系结构比较





# 移进-归约分析的工作过程

---

- 系统运行
  - 开始格局
    - 栈: #; 输入缓冲区: w#
  - 存放已经分析出来的结果, 并将读入的符号送入栈, 一旦句柄在栈顶形成, 就将其弹出进行归约, 并将结果压入栈
    - 问题: 系统如何发现句柄在栈顶形成?
  - 正常结束: 栈中为 #S, 输入缓冲区只有 #

输出结果表示: **例5.2  $E \rightarrow E + E | E * E | (E) | id$**   
用产生式序列表示语法分析树

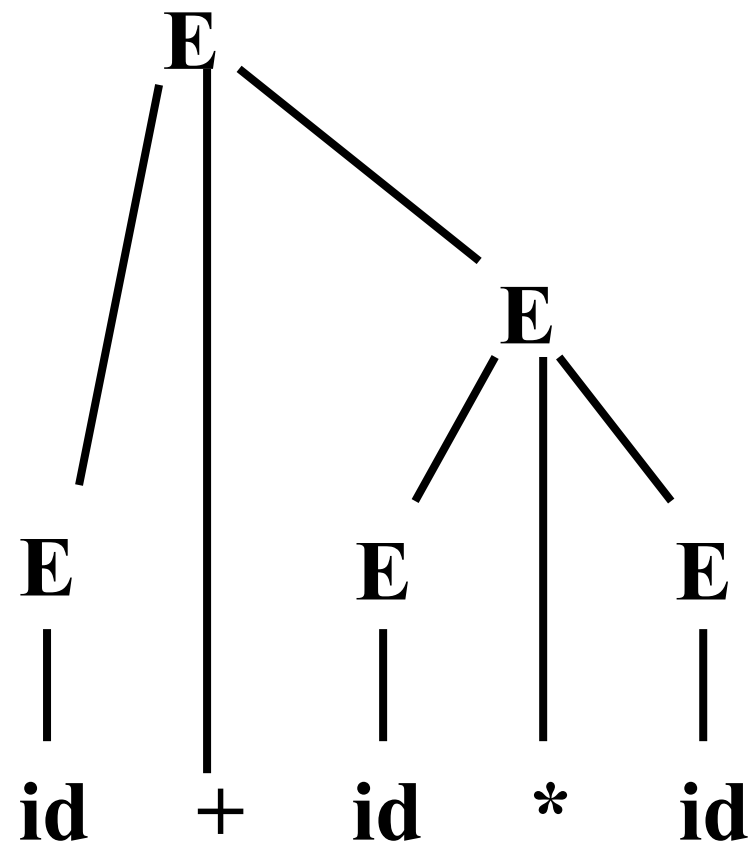
$E \rightarrow id$

$E \rightarrow id$

$E \rightarrow id$

$E \rightarrow E * E$

$E \rightarrow E + E$





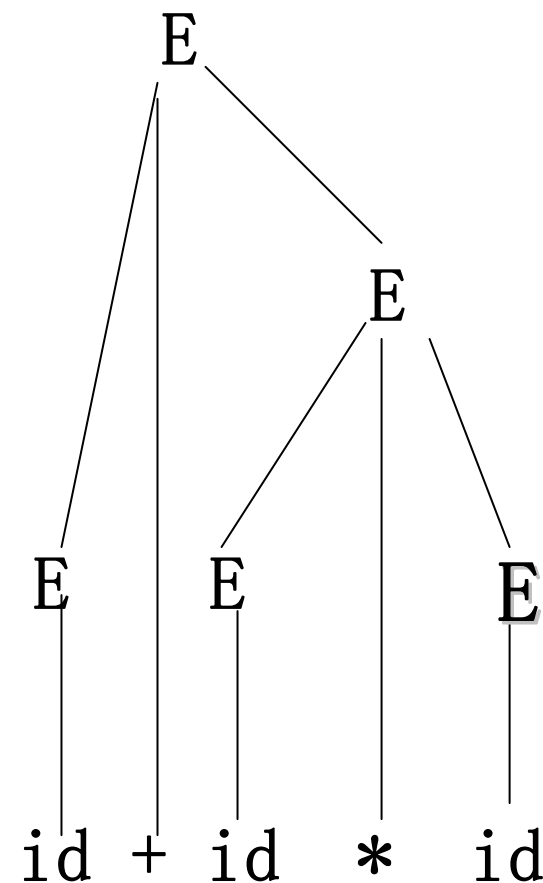
动作

栈

输入缓冲区

1)		#	$id_1 + id_2 * id_3 \#$
2)	移进	$\#id_1$	$+id_2 * id_3 \#$
3)	归约 $E \rightarrow id$	$\#E$	$+id_2 * id_3 \#$
4)	移进	$\#E +$	$id_2 * id_3 \#$
5)	移进	$\#E + id_2$	$*id_3 \#$
6)	归约 $E \rightarrow id$	$\#E + E$	$*id_3 \#$
7)	移进	$\#E + E *$	$id_3 \#$
8)	移进	$\#E + E * id_3$	$\#$
9)	归约 $E \rightarrow id$	$\#E + E * E$	$\#$
10)	归约 $E \rightarrow E * E$	$\#E + E$	$\#$
11)	归约 $E \rightarrow E + E$	$\#E$	$\#$
12)	接受		

例5.2 分析过程





# 分析器的四种动作

---

- 1) 移进: 将下一输入符号移入栈
  - 2) 归约: 用产生式左侧的非终结符替换栈顶的句柄 (某产生式右部)
  - 3) 接受: 分析成功
  - 4) 出错: 出错处理
- ?? 决定移进和归约的依据是什么——回头看是否可以找到答案



# 移进-归约分析中的问题

---

- 1) 移进归约冲突
  - 例5.2中的 6) 可以移进 \* 或按产生式  $E \rightarrow E+E$  归约



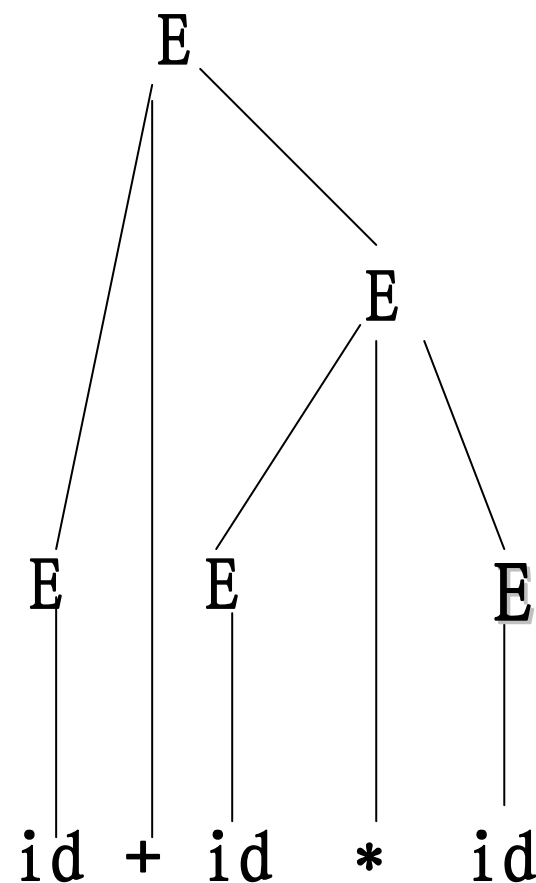
动作

栈

输入缓冲区

1)			#	$id_1 + id_2 * id_3 \#$
2)	移进		$\#id_1$	$+id_2 * id_3 \#$
3)	归约	$E \rightarrow id$	$\#E$	$+id_2 * id_3 \#$
4)	移进		$\#E +$	$id_2 * id_3 \#$
5)	移进		$\#E + id_2$	$*id_3 \#$
6)	归约	$E \rightarrow id$	$\#E + E$	$*id_3 \#$
7)	移进		$\#E + E *$	$id_3 \#$
8)	移进		$\#E + E * id_3$	$\#$
9)	归约	$E \rightarrow id$	$\#E + E * E$	$\#$
10)	归约	$E \rightarrow E * E$	$\#E + E$	$\#$
11)	归约	$E \rightarrow E + E$	$\#E$	$\#$
12)	接受			

例5.2分析过程





# 移进-归约分析中的问题

---

## 1) 移进归约冲突

- 例5.2中的 6) 可以移进 \* 或按产生式  $E \rightarrow E+E$  归约

## 2) 归约归约冲突

- 存在两个可用的产生式
- 各种分析方法处理冲突的方法不同
- 如何识别句柄?
  - 如何保证找到的直接短语是最左的? 利用栈
  - 如何确定句柄的开始处与结束处?



## 5.1.2 优先法

- 根据归约的先后次序为句型中相邻的文法符号规定优先关系
  - 句柄内相邻符号同时归约，是同优先的
  - 句柄两端符号的优先级要高于句柄外与之相邻的符号
- $a_1 \dots a_{i-1} \prec a_i \equiv a_{i+1} \equiv \dots \equiv a_{j-1} \equiv a_j \succ a_{j+1} \dots a_n$
- 定义了这种优先关系之后，语法分析程序就可以通过  $a_{i-1} \prec a_i$  和  $a_j \succ a_{j+1}$  这两个关系来确定句柄的头和尾了



## 5.1.3 状态法

- 根据句柄的识别状态（句柄是逐步形成的）
  - 用状态来描述不同时刻下形成的那部分句柄
  - 因为句柄是产生式的右部，可用产生式来表示句柄的不同识别状态
- 例如：  $S \rightarrow bBB$  可分解为如下识别状态
  - $S \rightarrow .bBB$  移进b
  - $S \rightarrow bB.B$  等待归约出B
  - $S \rightarrow b.BB$  等待归约出B
  - $S \rightarrow bBB.$  归约
- 采用这种方法，语法分析程序根据当前的分析状态就可以确定句柄的头和尾，并进行正确的归约

。



## 5.2 算符优先分析法

---

- 算术表达式分析的启示

- 算符优先关系的直观意义

- $+$   $\lessdot$   $*$                        $+$  的优先级低于  $*$

- $($   $\equiv$   $)$                        $($  的优先级等于  $)$

- $+$   $\rhd$   $+$                        $+$  的优先级高于  $+$

- 方法

- 将句型中的终结符号当作“算符”，借助于算符之间的优先关系确定句柄



# 算术表达式文法的再分析

■  $E \rightarrow E + E$

■  $E \rightarrow E - E$

■  $E \rightarrow E * E$

■  $E \rightarrow E / E$

■  $E \rightarrow (E)$

■  $E \rightarrow id$

从如何去掉二义性,看  
对算符优先级的利用

句型的特征:

$(E + E) * (E - E) / E / E + E * E * E$

■  $E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow (E) \mid id$



# 算符文法

---

■如果文法  $G = (V, T, P, S)$  中不存在形如

$$A \rightarrow \alpha BC \beta$$

的产生式，则称之为算符文法(OG —Operator Grammar)

即：如果文法  $G$  中不存在具有相邻非终结符的产生式，则称为算符文法。

## 5.2.1 算符优先文法

- 定义5.1 假设 $G$ 是一个不含 $\varepsilon$ -产生式的文法,  $A$ 、 $B$ 和 $C$ 均是 $G$ 的语法变量,  $G$ 的任何一对终结符 $a$ 和 $b$ 之间的**优先关系**定义为:
  - (1)  $a \equiv b$ , 当且仅当文法 $G$ 中含有形如 $A \rightarrow \dots ab \dots$ 或 $A \rightarrow \dots aBb \dots$ 的产生式;
  - (2)  $a \lessdot b$ , 当且仅当文法 $G$ 中含有形如 $A \rightarrow \dots aB \dots$ 的产生式, 而且 $Bb \dots$ 或 $BCb \dots$ ;
  - (3)  $a \gtrdot b$ , 当且仅当文法 $G$ 中含有形如 $A \rightarrow \dots Bb \dots$ 的产生式, 而且 $B \dots a$ 或 $B \dots aC$ ;
  - (4)  $a$ 与 $b$ 无关系, 当且仅当 $a$ 与 $b$ 在 $G$ 的任何句型中都不相邻。
- **问题: 什么是算符优先文法?**



## 5.2.1 算符优先文法

---

- 设  $G = (V, T, P, S)$  为 OG, 如果  $\forall a, b \in V_T$ ,  $a \equiv b, a \lessdot b, a \gtrdot b$  至多有一个成立, 则称之为算符优先文法(OPG — Operator Precedence Grammar)

——在无  $\varepsilon$  产生式的算符文法  $G$  中, 如果任意两个终结符之间至多有一种优先关系, 则称为算符优先文法。

## 5.2.2 算符优先矩阵的构造

### ■ 优先关系的确定

#### ■ 根据优先关系的定义

- $a \lessdot b \Leftrightarrow A \rightarrow \dots aB \dots \in P$  且  $(B \Rightarrow^+ b \dots \text{或者 } B \Rightarrow^+ Cb \dots)$
- 需要求出非终结符B派生出的第一个终结符集
- $a \gtrdot b \Leftrightarrow A \rightarrow \dots Bb \dots \in P$  且  $(B \Rightarrow^+ \dots a \text{或者 } B \Rightarrow^+ \dots aC)$
- 需要求出非终结符B派生出的最后一个终结符集

### ■ 设 $G = (V, T, P, S)$ 为OG, 则定义

- $\text{FIRSTOP}(A) = \{b | A \Rightarrow^+ b \dots \text{或者 } A \Rightarrow^+ Bb \dots, b \in T, B \in V\}$
- $\text{LASTOP}(A) = \{b | A \Rightarrow^+ \dots b \text{或者 } A \Rightarrow^+ \dots bB, b \in T, B \in V\}$



# 算符优先关系矩阵的构造

---

- $A \rightarrow \dots ab\dots ; A \rightarrow \dots aBb\dots$ , 则  $a \equiv b$
- $A \rightarrow \dots aB\dots$ , 则对  $\forall b \in \text{FIRSTOP}(B), a \lessdot b$
- $A \rightarrow \dots Bb\dots$ , 则对  $\forall a \in \text{LASTOP}(B), a \gtrdot b$
- if  $A \rightarrow B\dots \in P$ , then  $\text{FIRSTOP}(B) \subseteq \text{FIRSTOP}(A)$
- if  $A \rightarrow \dots B \in P$ , then  $\text{LASTOP}(B) \subseteq \text{LASTOP}(A)$
- 问题: 编程求 **FIRSTOP**、**LASTOP**



# 算符优先关系矩阵的构造

■  $A \rightarrow X_1 X_2 \dots X_n$

① 如果  $X_i X_{i+1} \in TT$  则:  $X_i \equiv X_{i+1}$

② 如果  $X_i X_{i+1} X_{i+2} \in TVT$  则:  $X_i \equiv X_{i+2}$

③ 如果  $X_i X_{i+1} \in TV$  则:  $\forall a \in \text{FIRSTOP}(X_{i+1})$ ,  
 $X_i \lessdot a$

④ 如果  $X_i X_{i+1} \in VT$  则:  $\forall a \in \text{LASTOP}(X_i)$ ,  
 $a \rhd X_{i+1}$

## 例 5.6 表达式文法的算符优先关系

	+	-	*	/	(	)	id	#
+	≠	≠	<	<	<	≠	<	≠
-	≠	≠	<	<	<	≠	<	≠
*	≠	≠	≠	≠	<	≠	<	≠
/	≠	≠	≠	≠	<	≠	<	≠
(	<	<	<	<	<	≡	<	
)	≠	≠	≠	≠		≠		≠
id	≠	≠	≠	≠		≠		≠
#	<	<	<	<	<		<	acc





## 5.2.3 算符优先分析算法

---

### ■ 原理

- 识别句柄并归约
- 各种优先关系存放在算符优先分析表中
- 利用  $\succ$  识别句柄尾，利用  $\prec$  识别句柄头，分析栈存放已识别部分，比较栈顶和下一输入符号的关系，如果是句柄尾，则沿栈顶向下寻找句柄头，找到后弹出句柄，归约为非终结符。

例5.7  $E \rightarrow E+T|E-T|T$      $T \rightarrow T*F|T/F|F$      $F \rightarrow (E)|id$ ,  
试利用算符优先分析法对  $id+id$  进行分析

步骤	栈	输入串	优先关系	动作
1	#	$id_1+id_2\#$		
2	# $id_1$	$+id_2\#$	$\# \lessdot id_1$	移进 $id_1$
3	#F	$+id_2\#$	$\# \lessdot id_1 \rhd +$	用 $F \rightarrow id$ 归约
4	#F+	$id_2\#$	$\lessdot$	移进 +
5	#F+ $id_2$	#	$\lessdot$	移进 $id_2$
6	#F+F	#	$+ \lessdot id_2 \rhd \#$	用 $F \rightarrow id$ 归约
7	#E	#	$\# \lessdot + \rhd \#$	用 $E \rightarrow E+T$ 归约



# 问题

---

- 有时未归约真正的句柄 (F)
- 不是严格的最左归约
- 归约的符号串有时与产生式右部不同
- 仍能正确识别句子的原因
  - OPG未定义非终结符之间的优先关系，不能识别由单非终结符组成的句柄
  - 定义算符优先分析过程识别的“句柄”为**最左素短语**LPP (Leftmost Prime Phase)



# 素短语与最左素短语

- 什么是短语？当前我们要找什么样的短语？——至少有一个算符
- $S \Rightarrow^* \alpha A \beta$  and  $A \Rightarrow^+ \gamma$ ， $\gamma$  至少含一个终结符，且不含更小的含终结符的短语，则称  $\gamma$  是句型  $\alpha \gamma \beta$  的相对于变量  $A$  的素短语(Prime Phrase)
- 句型的至少含一个终结符且不含其它素短语的短语

# 例

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

句型  $T+T * F+i$  的短语有

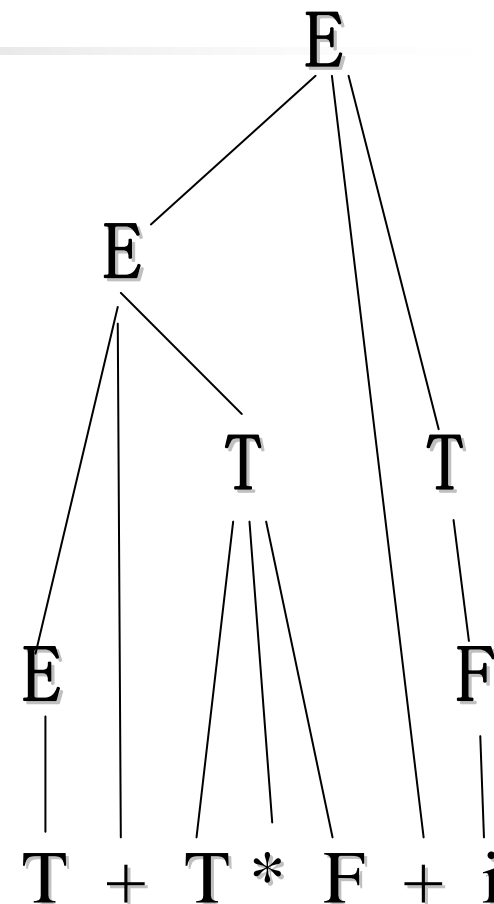
$T \quad T * F \quad i \quad T+T * F \quad T+T * F+i$

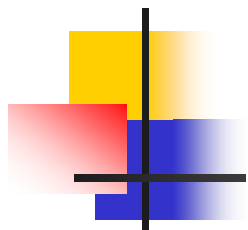
其中的素短语为

$T * F \quad i$

$T * F$  为最左素短语，是被归约的对象

问题：按照文法  $E \rightarrow E+E \mid E * E \mid (E) \mid id$ ，  
求  $i+E*i+i$  的短语和素短语





文法:  $E \rightarrow E + E \mid E * E$

$E \rightarrow (E) \mid id$

句型  $i + E * i + i$  的短语

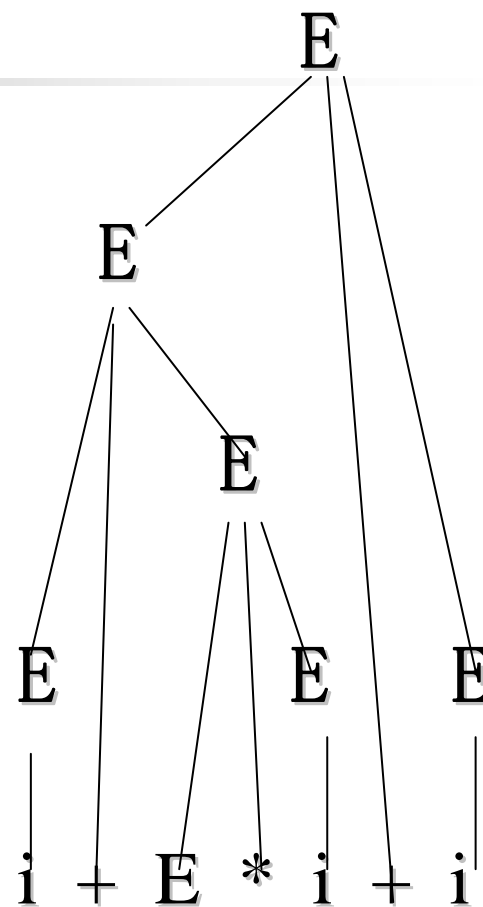
$i \quad i \quad E * i \quad i$

$i + E * i \quad i + E * i + i$

其中的素短语为

$i \quad i \quad i$

问题: 归约过程中如何发现“  
中间句型”的最左素短语?





# 素短语与最左素短语

---

设句型的一般形式为

$$\#N_1a_1N_2a_2\ldots N_na_n\# (N_i \in V \cup \{\varepsilon\}, a_i \in V_T)$$

它的最左素短语是满足下列条件的最左子串

$$N_ia_iN_{i+1}a_{i+1}\ldots N_ja_jN_{j+1}$$

其中:  $a_{i-1} \prec a_i$ ,

$$a_i \equiv a_{i+1} \equiv \ldots \equiv a_{j-1} \equiv a_j ,$$

$$a_j \succ a_{j+1}$$



# 算符优先分析的实现

---

## ■ 系统组成

- 移进归约分析器 + 优先关系表

## ■ 分析算法

- 参照输入串、优先关系表，完成一系列归约，生成语法分析树——输出产生式



# 算符优先分析算法

## 算法5.3 算符优先分析算法。

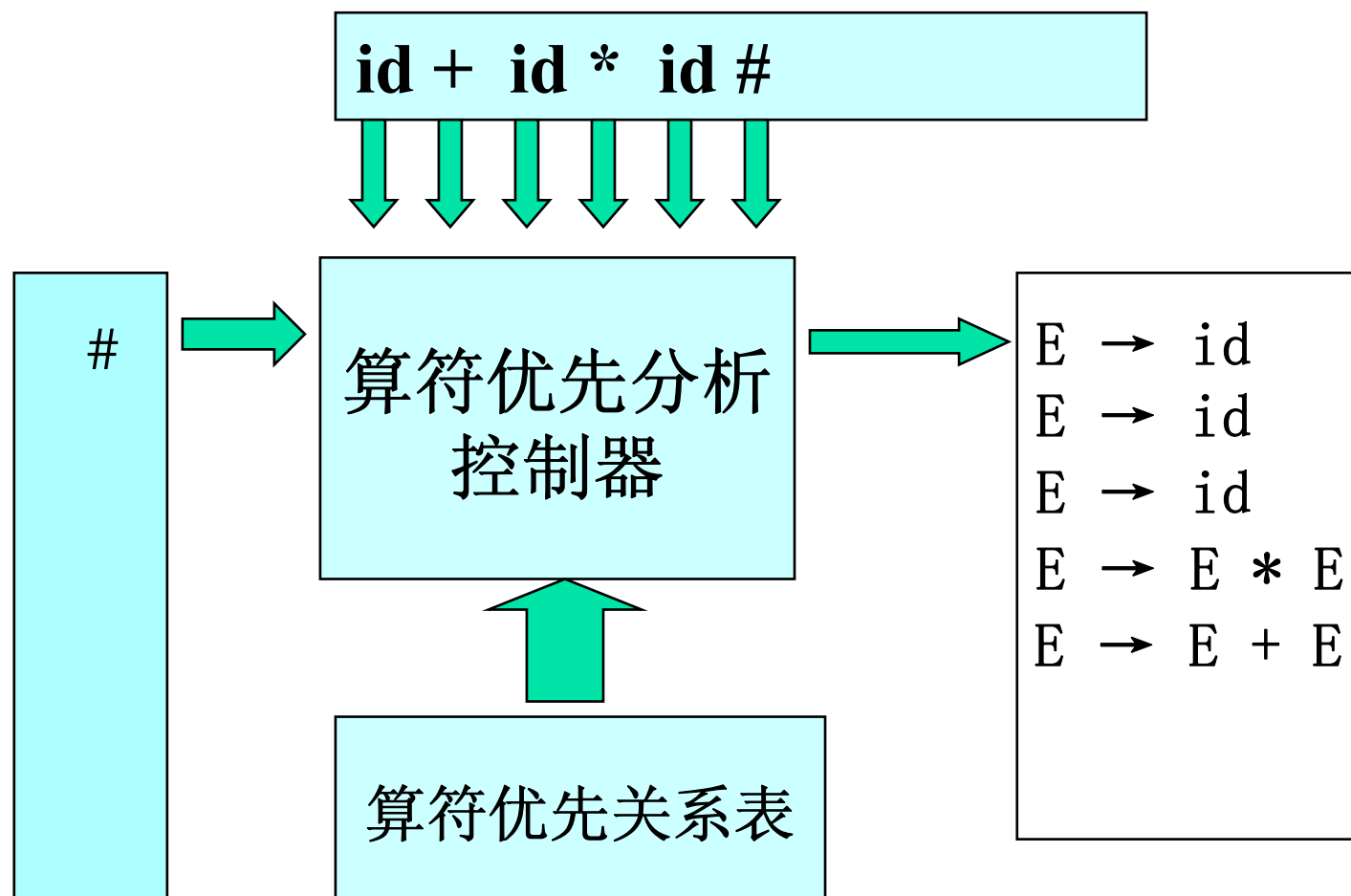
输入：文法  $G=(V, T, P, S)$ ，输入字符串  $w$  和优先关系表；

输出：如果  $w$  是一个句子则输出一个分析树架子，否则指出错误；

步骤：

```
begin  S[1]:='#';  i:=1;
      repeat 将下一输入符号读入R;
        if S[i] ∈ T then j:=i else j:=i-1;
        while S[j] ⋈ R do begin
          repeat  Q:=S[j];
            if S[j-1] ∈ T then j:=j-1 else j:=j-2
          until S[j] ⋈ Q;
          将S[j+1]...S[i]归约为N; i:=j+1; S[i]:=N end;
          if S[j] ⋈ R or S[j] ≡ R then begin i:=i+1; S[i]:=R end
        else error
      until i=2 and R='#' end;
```

# id+id\*id 的分析过程





## 5.2.4 优先函数

- 为了节省存储空间 ( $n^2 \rightarrow 2n$ ) 和便于执行比较运算, 用两个优先函数  $f$  和  $g$ , 它们是从终结符号到整数的映射。对于终结符号  $a$  和  $b$  选择  $f$  和  $g$ , 使之满足:
  - $f(a) < g(b)$ , 如果  $a \prec b$
  - $f(a) = g(b)$ , 如果  $a \equiv b$
  - $f(a) > g(b)$ , 如果  $a \succ b$ 。
- 损失
  - 错误检测能力降低
    - 如:  $id \succ id$  不存在, 但  $f(id) > g(id)$  可比较

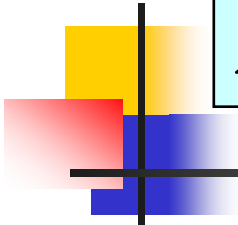


表5.2 对应的优先函数:

	+	-	*	/	(	)	id	#
<i>f</i>	2	2	4	4	0	4	4	0
<i>g</i>	1	1	3	3	5	0	5	0

- 1) 构造优先函数的算法不是唯一的。
- 2) 存在一组优先函数，那就存在无穷组优先函数。



# 优先函数的构造

算法5.4 优先函数的构造。

输入：算符优先矩阵；

输出：表示输入矩阵的优先函数，或指出其不存在；

步骤：

1. 对 $\forall a \in T \cup \{\#\}$ ，建立以 $fa$ 和 $ga$ 为标记的顶点；
2. 对 $\forall a, b \in T \cup \{\#\}$ ，若 $a \succ b$ 或者 $a \equiv b$ ，则从 $fa$ 至 $gb$ 画一条有向弧；若 $a \prec b$ 或者 $a \equiv b$ ，则从 $gb$ 至 $fa$ 画一条有向弧；
3. 如果构造的有向图中有环路，则说明不存在优先函数；如果没有环路，则对 $\forall a \in T \cup \{\#\}$ ，将 $f(a)$ 设为从 $fa$ 开始的最长路经的长度，将 $g(a)$ 设为从 $ga$ 开始的最长路经的长度。

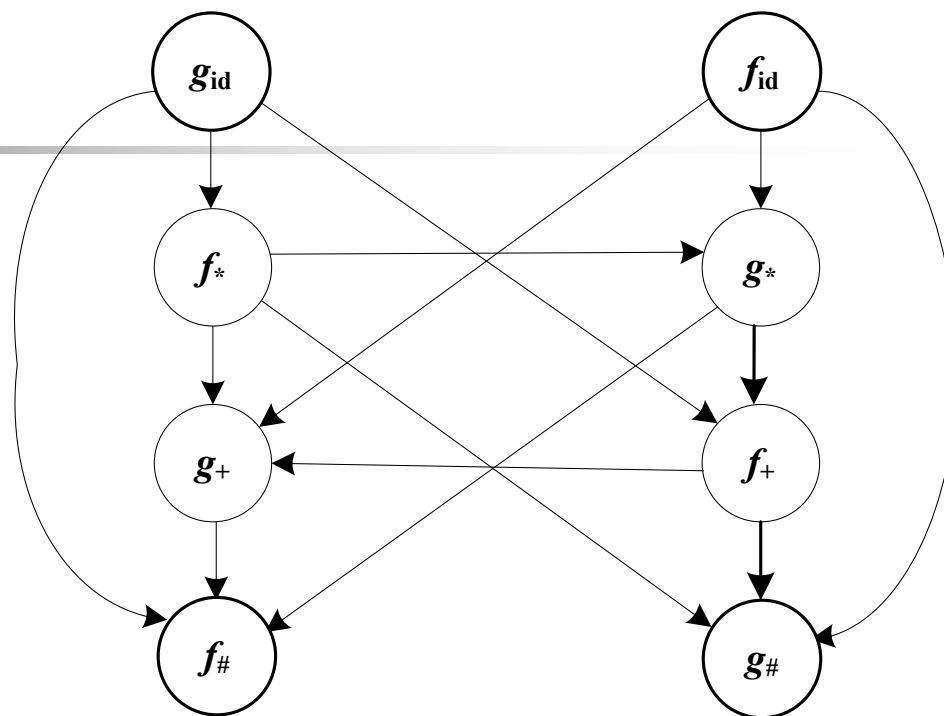
# 例5.10

$G_{es} : E \rightarrow E + T \mid T$   
 $T \rightarrow T * F \mid F$   
 $F \rightarrow \text{id}$

	+	*	id	#
+	$\succ$	$\prec$	$\prec$	$\succ$
*	$\succ$	$\succ$	$\prec$	$\succ$
id	$\succ$	$\succ$		$\succ$
#	$\prec$	$\prec$	$\prec$	

$G_{es}$ 的优先矩阵

2012-4-26



	+	*	id	#
$f$	2	4	4	0
$g$	1	3	5	0

根据 $G_{es}$ 的优先矩阵建立的  
 有向图和优先函数

310



## 5.2.5 算符优先分析的出错处理

- (1) 栈顶的终结符号和当前输入符号之间不存在任何优先关系；
  - (2) 发现被“归约对象”，但该“归约对象”不能满足归约要求。
- 对于第(1)种情况，为了进行错误恢复，必须修改栈、输入或两者都修改。
  - 对于优先矩阵中的每个空白项，必须指定一个出错处理程序，而且同一程序可用在多个地方。
  - 对于第(2)种情况，由于找不到与“归约对象”匹配的产生式右部，分析器可以继续将这些符号弹出栈，而不执行任何语义动作。



# 算符优先分析法小结

---

## ■ 优点

- 简单、效率高
- 能够处理部分二义性文法

## ■ 缺点

- 文法书写限制大——强调算符之间的优先关系的唯一性
- 占用内存空间大
- 不规范、存在查不到的语法错误
- 算法在发现最左素短语的尾时，需要回头寻找对应的头



## 5.3 LR分析法

### 5.3.1 LR分析算法

#### ■ LR (k) 分析法可分析LR (k) 文法产生的语言

- L : 从左到右扫描输入符号

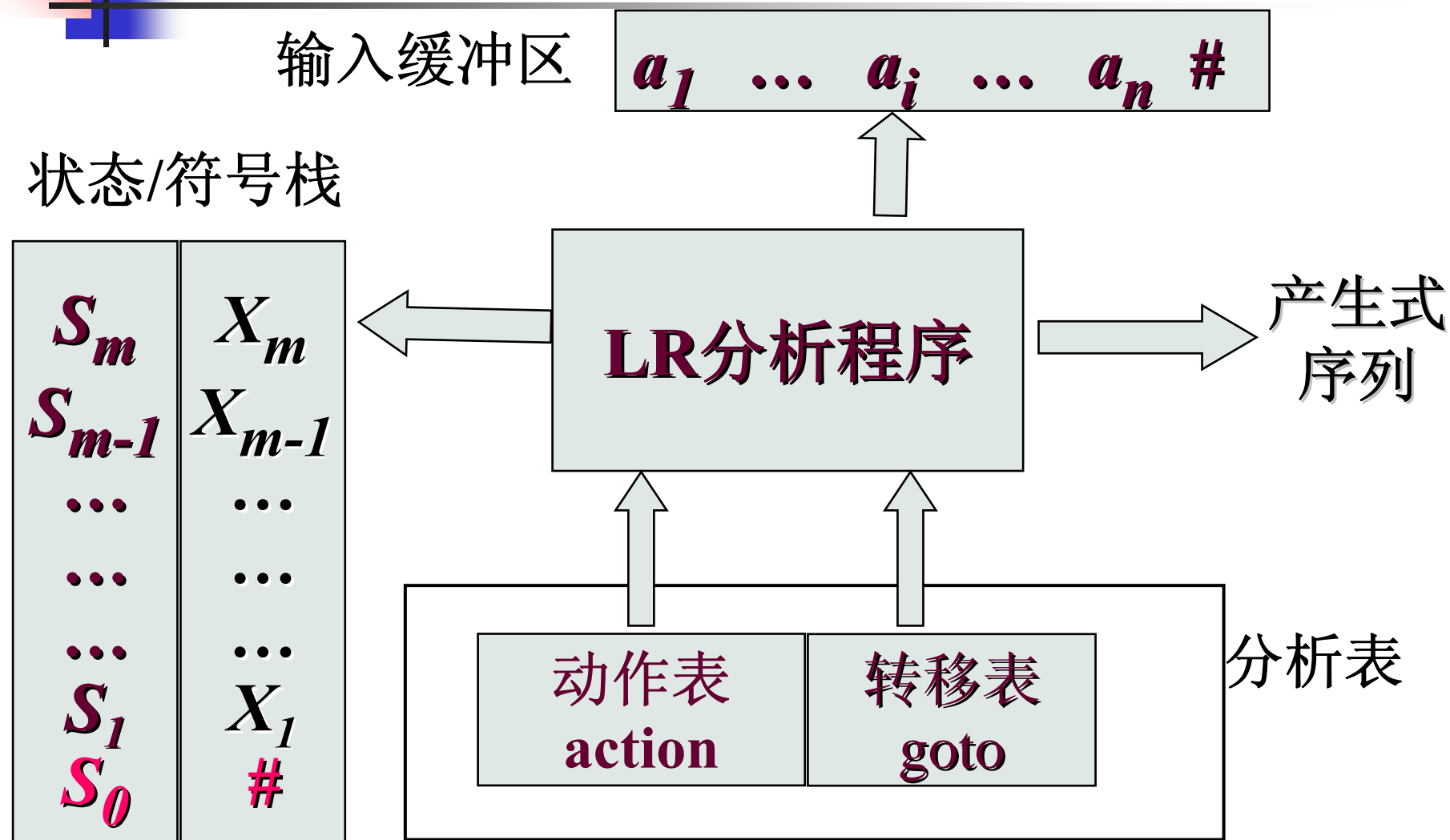
- R : 最右推导对应的最左归约

- k : 超前读入k个符号, 以便确定归约用的产生式

- 使用语言的文法描述内涵解决句柄的识别问题, 从语言的**形式描述**入手, 为语法分析器的**自动生成**提供了前提和基础

■ 分析器根据当前的状态, 并至多向前查看k个输入符号, 就可以确定是否找到了句柄, 如果找到了句柄, 则按相应的产生式归约, 如果未找到句柄则移进输入符号, 并进入相应的状态

# LR语法分析器的总体结构



# LR 分析表: $\text{action}[s,a]$ ; $\text{goto}[s,X]$

约定:

sn:将符号a、状态n压入栈

rn:用第n个产生式进行归约

LR(0)、SLR(1)、  
LR(1)、LALR(1)  
将以不同的原则  
构造这张分析表

状态	动作表 action			转移表 goto	
	a	b	#	S	B
0	s3	s4		1	2
1			acc		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		



# LR分析器的工作过程

---

- 书上的下式（格局）

$(s_0s_1\dots s_m, X_1X_2\dots X_m, a_ia_{i+1}\dots a_n\#)$

- 在这里表示为

$$\begin{array}{ccc} s_0s_1\dots s_m & & \\ \#X_1\dots X_m & a_ia_{i+1}\dots a_n\# & \end{array}$$

# LR分析器的工作过程

## ■ 1. 初始化

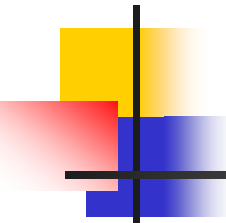
$s_0$   
#  $a_1 a_2 \dots a_n \#$       对应“句型”  $a_1 a_2 \dots a_n$

## ■ 2. 在一般情况下，假设分析器的格局如下：

$s_0 s_1 \dots s_m$   
#  $X_1 \dots X_m a_i a_{i+1} \dots a_n \#$       对应“句型”  $X_1 \dots X_m a_i a_{i+1} \dots a_n$

■ ① If  $\text{action}[s_m, a_i] = \text{si}(\text{shift } i)$  then 格局变为

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



$$\begin{array}{l} S_0 S_1 \dots S_m \\ \# X_1 \dots X_m \quad a_i a_{i+1} \dots a_n \# \end{array}$$

② If  $\text{action}[s_m, a_i] = \text{ri}(\text{Reduce } i)$  then 表示用第*i*个产生式  $A \rightarrow X_{m-(k-1)} \dots X_m$  进行归约，格局变为

$$\begin{array}{l} S_0 S_1 \dots S_{m-k} \\ \# X_1 \dots X_{m-k} A \quad a_i a_{i+1} \dots a_n \# \end{array}$$

查goto表，如果  $\text{goto}[s_{m-k}, A] = i$  then 格局变为

$$\begin{array}{l} S_0 S_1 \dots S_{m-k} \quad i \\ \# X_1 \dots X_{m-k} A \quad a_i a_{i+1} \dots a_n \# \end{array}$$

③ If  $\text{action}[s_m, a_i] = \text{acc}$  then 分析成功

④ If  $\text{action}[s_m, a_i] = \text{err}$  then 出现语法错

# LR分析算法

## 算法5.5 LR分析算法。

输入：文法 $G$ 的LR分析表和输入串 $w$ ;

输出：如果 $w \in L(G)$ ，则输出 $w$ 的自底向上分析，否则报错;

步骤：

1. 将#和初始状态 $S_0$ 压入栈，将 $w\#$ 放入输入缓冲区;
2. 令输入指针 $ip$ 指向 $w\#$ 的第一个符号;
3. 令 $S$ 是栈顶状态， $a$ 是 $ip$ 所指向的符号;
4. repeat
5.   if  $action[S,a]=S_i$  then     /\*  $S_i$ 表示移进 $a$ 并转入状态 $i$ \*/
6.     begin
7.         把符号 $a$ 和状态 $i$ 先后压入栈;
8.         令 $ip$ 指向下一输入符号
9.     end

```
10. elseif action[S,a]=rk then /* ri表示按第k
    个产生式 $A \rightarrow \beta$ 归约 */
11.   begin
12.       从栈顶弹出 $2*|\beta|$ 个符号;
13.       令 $S'$ 是现在的栈顶状态;
14.       把 $A$ 和 $goto[S',A]$ 先后压入栈中;
15.       输出产生式  $A \rightarrow \beta$ 
16.   end
17. elseif action[S,a]= acc then
18.   return
19. else
20.   error();
```





# 例5.12

## 分析表

文法

- 1)  $S \rightarrow BB$
- 2)  $B \rightarrow aB$
- 3)  $B \rightarrow b$

请跟随讲解，快速抄下右侧的表格！

状态	动作表 action			转移表 goto	
	a	b	#	S	B
0	s3	s4		1	2
1			acc		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

## 栈 输入 动作说明

0  
# bab# action(0,b)=s4  
04  
#b ab# action(4,a)=r3  
0  
#B ab# goto(0,B)=2  
02  
#B ab# action(2,a)=s3  
023  
#Ba b# action(3,b)=s4  
0234  
#Bab # action(4,#)=r3  
023  
#BaB # goto(3,B)=6

## **bab** 的分析过程:

- 1)  $S \rightarrow BB$
- 2)  $B \rightarrow aB$
- 3)  $B \rightarrow b$

0236  
#BaB # action(6,#)=r2  
02  
#BB # goto(2,B)=5  
025  
#BB # action(5,#)=r1  
  
0  
#S # goto(0,S)=1  
01  
#S # action(1,#)=acc

# 规范句型活前缀

- 分析栈中内容+剩余输入符号=规范句型
  - 分析栈中内容为某一句型的前缀
- 来自分析栈的活前缀(Active Prefix)
  - 不含句柄右侧任意符号的规范句型的前缀
- 例:  $\text{id} + \text{id} * \text{id}$  的分析中

- 句型  $\text{E} + \text{id} . * \text{id}$  和  $\text{E} + \text{E} * . \text{id}$   

$\underbrace{\text{E} + \text{id} .}_{\text{活前缀}}$

$\underbrace{\text{E} + \text{E} * .}_{\text{活前缀}}$

$$S \Rightarrow_{\text{rm}}^* \alpha A w \Rightarrow_{\text{rm}} \alpha \beta_1 \beta_2 w$$

# 规范句型活前缀

- 规范归约所得到的规范句型(Canonical Sentential Form)的活前缀是出现在分析栈中的符号串，所以，不会出现句柄之后的任何字符，而且相应的后缀正是输入串中还未处理的终结符号串。
- 活前缀与句柄的关系
  - 包含句柄  $A \rightarrow \beta$ .
  - 包含句柄的部分符号  $A \rightarrow \beta_1 \cdot \beta_2$
  - 不含句柄的任何符号  $A \rightarrow \cdot \beta$

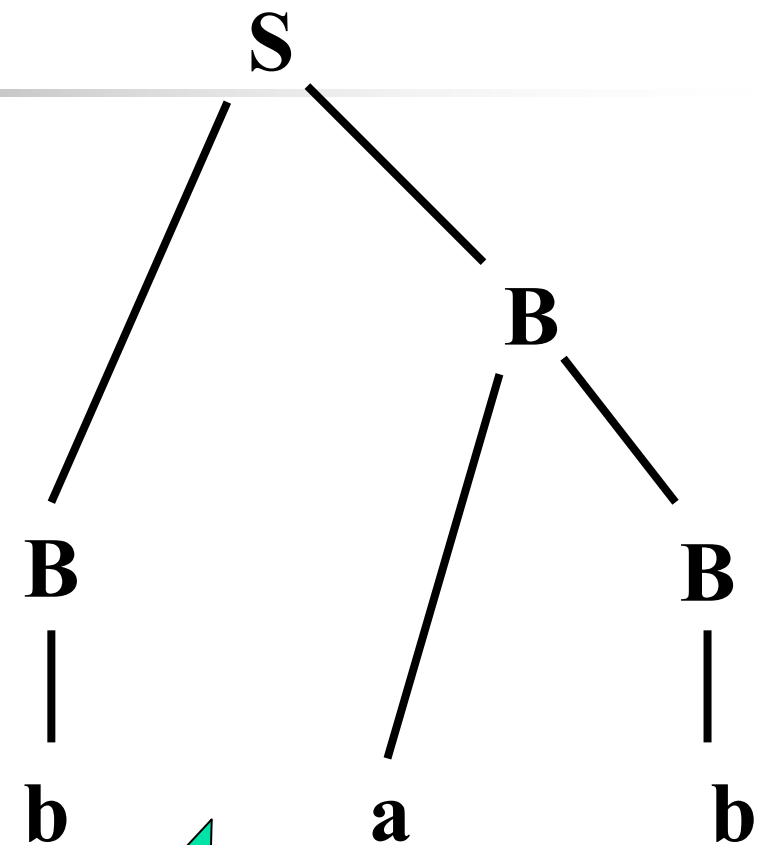


## 5.3.2 LR(0)分析表的构造

- LR(0)项目——从产生式寻找归约方法
  - 右部某个位置标有圆点的产生式称为相应文法的**LR(0)项目** (Item)
  - 例  $S \rightarrow .bBB$     $S \rightarrow bB.B$     $S \rightarrow b.BB$     $S \rightarrow bBB.$
  - 归约 (Reduce) 项目:  $S \rightarrow aBB.$
  - 移进 (Shift) 项目:  $S \rightarrow .bBB$
  - 待约项目:  $S \rightarrow b.BB$     $S \rightarrow bB.B$

# 项目的意义

- 用项目表示分析的进程(句柄的识别状态)
- 方法: 在产生式右部加一圆点以分割已获取的内容和待获取的内容: 构成句柄



$S \rightarrow B . B$   
 $B \rightarrow . a B$



# 拓广(Augmented)文法

---

- 需要一个对“归约成S”的表示（只有一个接受状态）
- 文法  $G = (V, T, P, S)$  的拓广文法  $G'$ :
  - $G' = (V \cup \{S'\}, T, P \cup \{S' \rightarrow S\}, S')$
  - $S' \notin V$
  - 对应  $S' \rightarrow .S$ （分析开始）和  $S' \rightarrow S.$ （分析成功）
- 例5.13
  - 0)  $S' \rightarrow S$
  - 1)  $S \rightarrow BB$
  - 2)  $B \rightarrow aB$
  - 3)  $B \rightarrow b$

- 
- 问题：如何设计能够指导分析器运行，并且能够根据当前状态（栈顶）确定句柄——归约对象的头——的装置

构造识别G的所有规范句型活前缀的DFA





# 项目集闭包的计算

项目集  $I$  的闭包 ( Closure )

$$\text{CLOSURE}(I) = I \cup \{B \rightarrow \cdot \gamma \mid A \rightarrow \alpha \cdot B \beta \in I, B \rightarrow \gamma \in P\}$$

算法

**J:=I;**

**repeat**

$$\mathbf{J = J \cup \{B \rightarrow \cdot \eta \mid A \rightarrow \alpha \cdot B \beta \in J, B \rightarrow \eta \in P\}}$$

**until J不再扩大**



# 闭包之间的转移

---

- 后继项目 ( Successive Item )
  - $A \rightarrow \alpha . X \beta$  的后继项目是  $A \rightarrow \alpha X . \beta$
- 闭包之间的转移
  - $go(I, X) = CLOSURE(\{A \rightarrow \alpha X . \beta \mid A \rightarrow \alpha . X \beta \in I\})$



# 状态转移的计算

---

- 确定在某状态遇到一个文法符号后的状态转移目标

```
function GO(I, X);  
begin  
    J:= $\emptyset$ ;  
    for I中每个形如 $A \rightarrow \alpha.X\beta$ 的项目 do  
        begin J:=J  $\cup$   $\{A \rightarrow \alpha X.\beta\}$  end;  
    return CLOSURE(J)  
end;
```

# 识别拓广文法所有规范句型活前缀的DFA

- 识别文法  $G = (V, T, P, S)$  的拓广文法  $G'$  的所有规范句型活前缀的DFA：

$$M = (C, V \cup T, go, I_0, C)$$

- $I_0 = \text{CLOSURE}(\{S' \rightarrow .S\})$
- $C = \{I_0\} \cup \{I \mid \exists J \in C, X \in V \cup T, I = go(J, X)\}$

称为  $G'$  的 **LR(0)项目集规范族** (Canonical Collection)

# 计算LR(0)项目集规范族 C

即：分析器状态集合

**begin**

**$C := \{\text{closure}(\{ S' \rightarrow .S \})\};$**

**repeat**

**for  $\forall I \in C, \forall X \in V \cup T$**

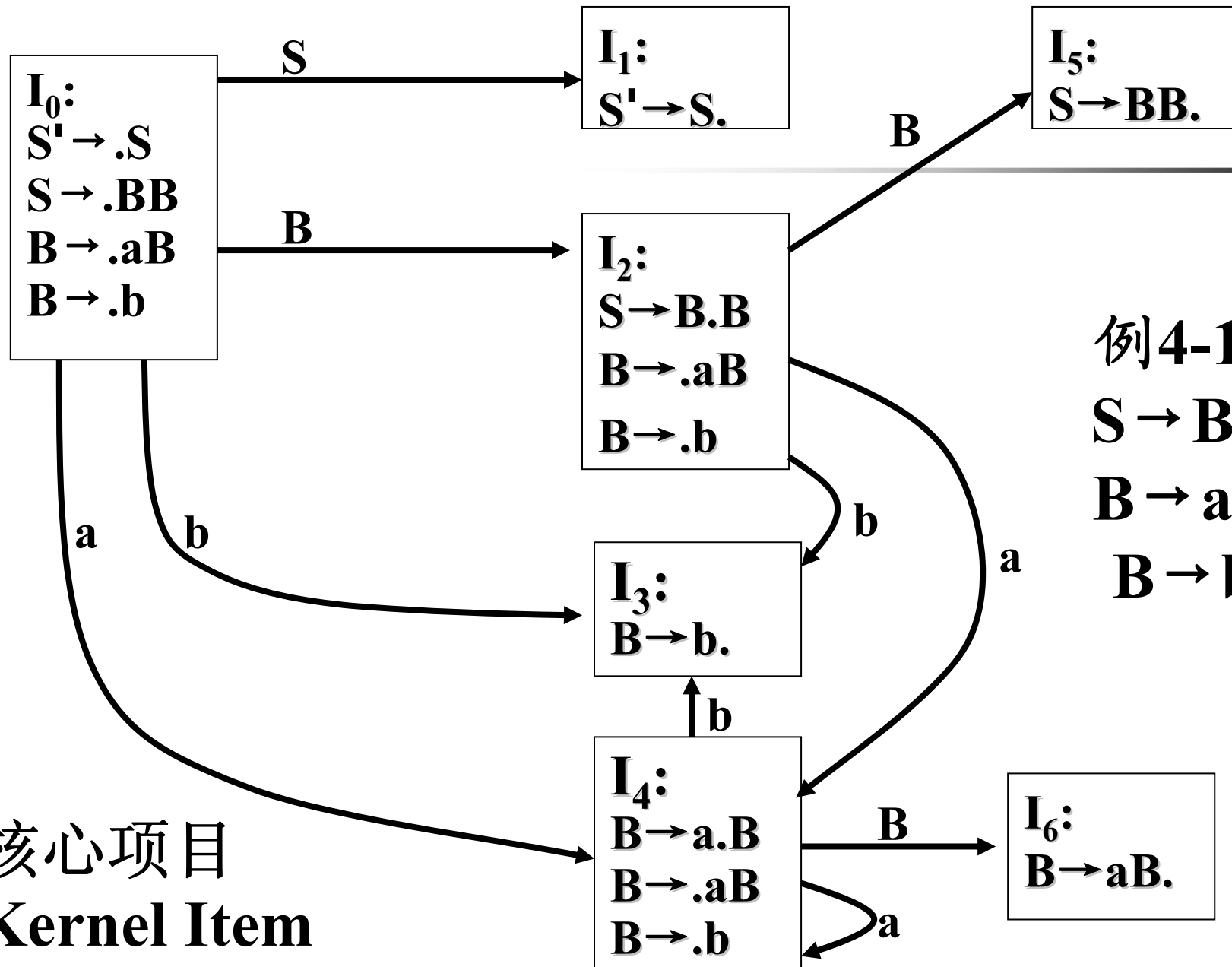
**if  $\text{go}(I, X) \neq \Phi$  &  $\text{go}(I, X) \notin C$  then**

**$C = C \cup \{\text{go}(I, X)\}$**

**until C不变化**

**end.**

# 核心项目 Kernel Item



例4-13  
 $S \rightarrow BB$   
 $B \rightarrow aB$   
 $B \rightarrow b$

# LR(0)分析表的构造算法

算法5.6 LR(0)分析表的构造。

输入：文法  $G=(V, T, P, S)$  的拓广文法  $G'$ ;

输出：  $G'$  的 LR(0) 分析表，即 *action* 表和 *goto* 表;

步骤：

1. 令  $I_0 = \text{CLOSURE}(\{S' \rightarrow .S\})$ ，构造  $G'$  的 LR(0) 项目集规范族  $C = \{I_0, I_1, \dots, I_n\}$

2. 让  $I_i$  对应状态  $i$ ， $I_0$  对应状态 0，0 为初始状态。

3. for  $k=0$  to  $n$  do begin

(1) if  $A \rightarrow \alpha.a\beta \in I_k$  &  $a \in T$  &  $\text{GO}(I_k, a) = I_j$  then  $\text{action}[k, a] := Sj$ ;

(2) if  $A \rightarrow \alpha.B\beta \in I_k$  &  $B \in V$  &  $\text{GO}(I_k, B) = I_j$  then  $\text{goto}[k, B] := j$ ;

(3) if  $A \rightarrow \alpha. \in I_k$  &  $A \rightarrow \alpha$  为  $G$  的第  $j$  个产生式 then

for  $\forall a \in T \cup \{\#\}$  do  $\text{action}[k, a] := rj$ ;

(4) if  $S' \rightarrow S. \in I_k$  then  $\text{action}[k, \#] := \text{acc}$  end;

4. 上述(1)到(4)步未填入信息的表项均置为 error。



# LR(0)不是总有效的

---

$(S' \rightarrow S)$

1)  $S \rightarrow A|B$

2)  $A \rightarrow aAc$

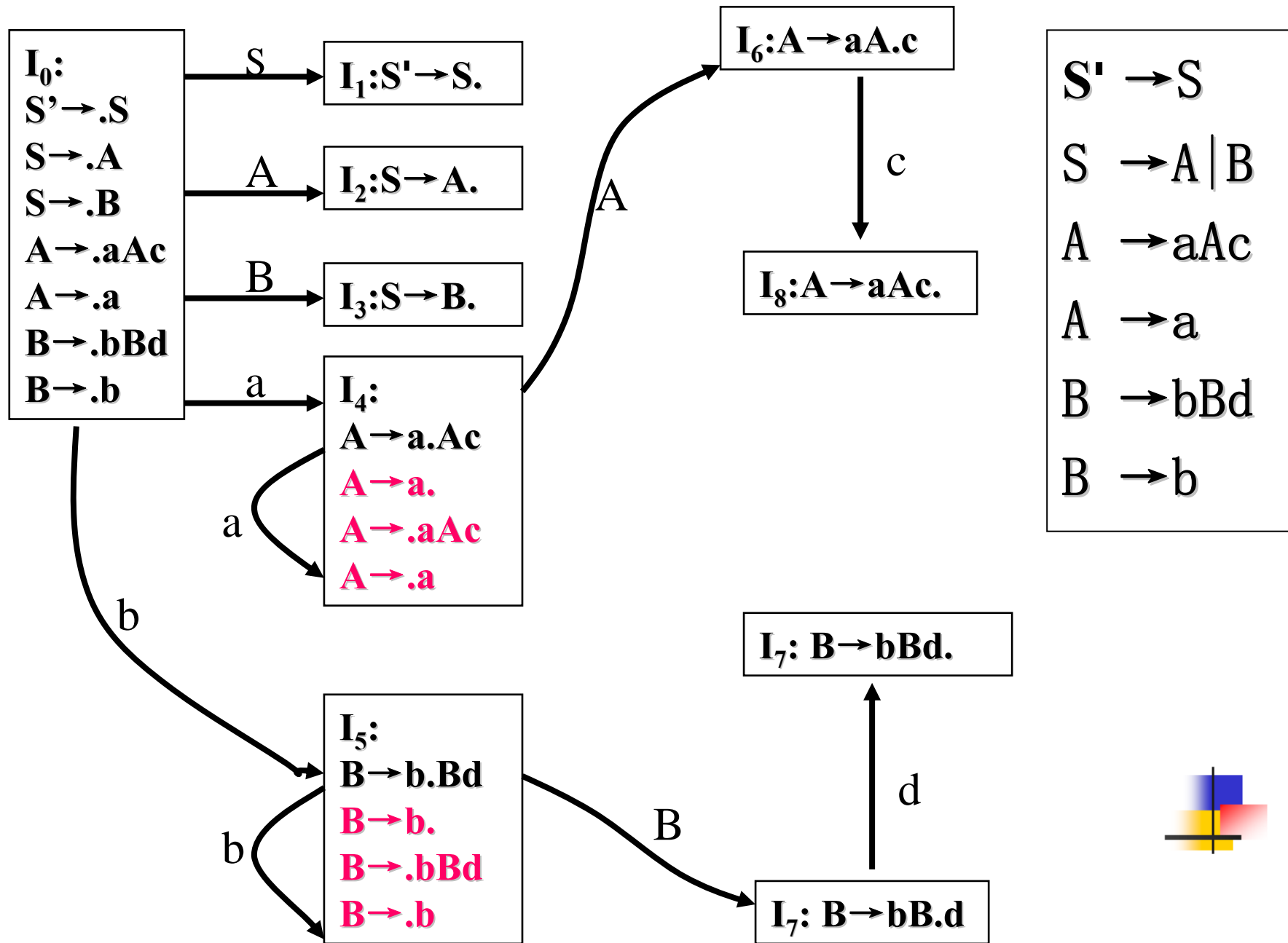
3)  $A \rightarrow a$

4)  $B \rightarrow bBd$

5)  $B \rightarrow b$

上下文无关文法  
不是都能用  
LR(0)方法进行  
分析的，也就是  
说，CFG不总是  
LR(0)文法.



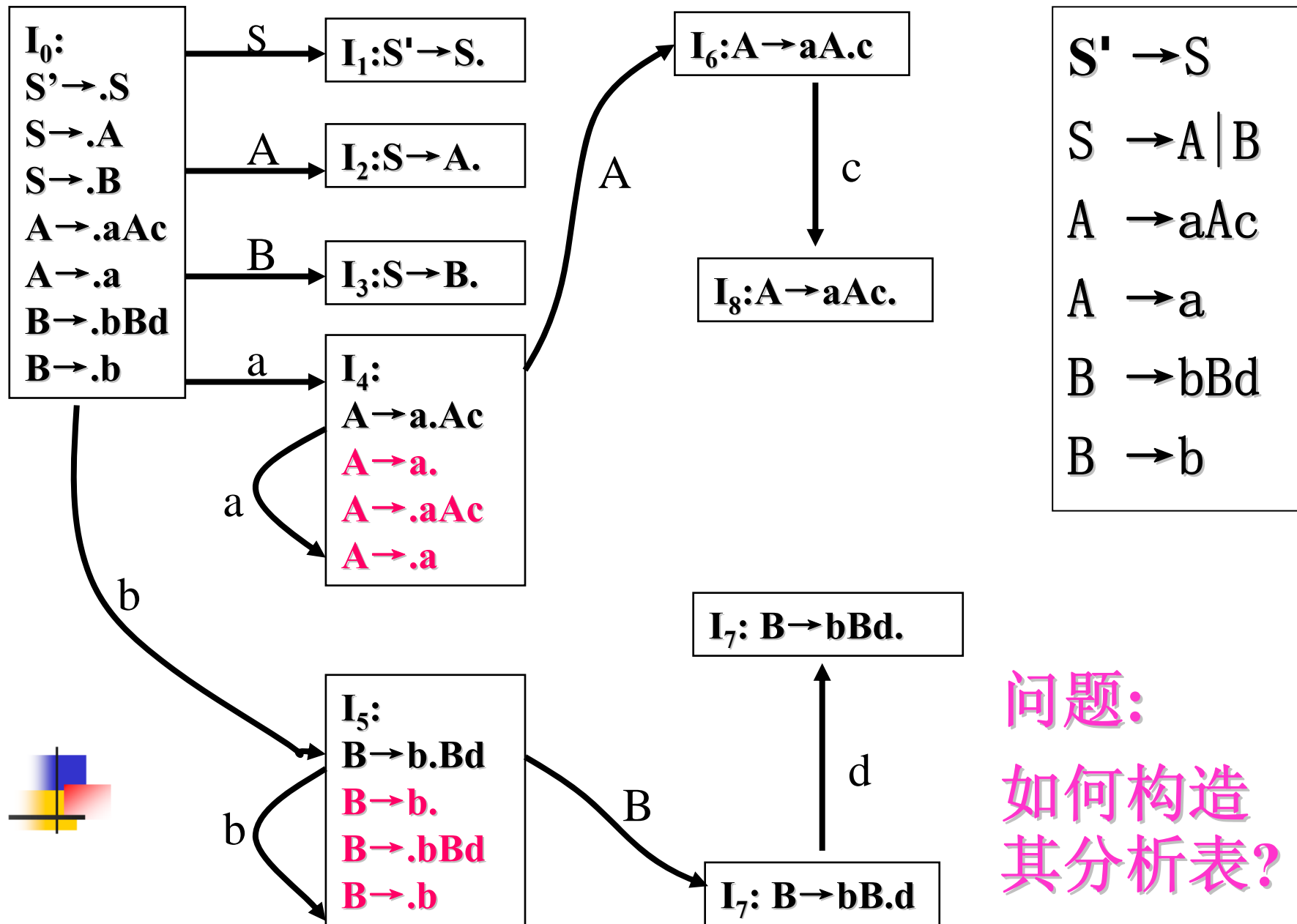




# 项目集 I 的相容

---

- 如果 I 中至少含两个归约项目，则称 I 有归约—归约冲突 (**Reduce/Reduce Conflict**)
- 如果 I 中既含归约项目，又含移进项目，则称 I 有移进—归约冲突 (**Shift/Reduce Conflict**)
- 如果 I 既没有归约—归约冲突，又没有移进—归约冲突，则称 I 是相容的(**Consistent**)，否则称 I 是不相容的
- 对文法 G，如果  $\forall I \in C$ , 都是相容的，则称 G 为 **LR(0)** 文法



问题:  
如何构造  
其分析表?

## 5.3.3 SLR(1)分析表的构造算法

算法5.6  $LR(0)$ 分析表的构造。

输入：文法  $G=(V, T, P, S)$  的拓广文法  $G'$ ;

输出：  $G'$  的  $LR(0)$  分析表，即 *action* 表和 *goto* 表;

步骤：

1. 令  $I_0 = \text{CLOSURE}(\{S' \rightarrow .S\})$ ，构造  $G'$  的  $LR(0)$  项目集规范族  $C = \{I_0, I_1, \dots, I_n\}$

2. 让  $I_i$  对应状态  $i$ ， $I_0$  对应状态 0，0 为初始状态。

3. for  $k=0$  to  $n$  do begin

(1) if  $A \rightarrow \alpha.a\beta \in I_k$  &  $a \in T$  &  $\text{GO}(I_k, a) = I_j$  then  $\text{action}[k, a] := Sj$ ;

(2) if  $A \rightarrow \alpha.B\beta \in I_k$  &  $B \in V$  &  $\text{GO}(I_k, B) = I_j$  then  $\text{goto}[k, B] := j$ ;

(3) if  $A \rightarrow \alpha. \in I_k$  &  $A \rightarrow \alpha$  为  $G$  的第  $j$  个产生式 then

for  $\forall a \in \text{FOLLOW}(A)$  do  $\text{action}[k, a] := rj$ ;

(4) if  $S' \rightarrow S. \in I_k$  then  $\text{action}[k, \#] := \text{acc}$  end;

4. 上述(1)到(4)步未填入信息的表项均置为 error。

# 识别表达式文法的所有活前缀的DFA

## 拓广文法

0)  $E' \rightarrow E$

1)  $E \rightarrow E + T$

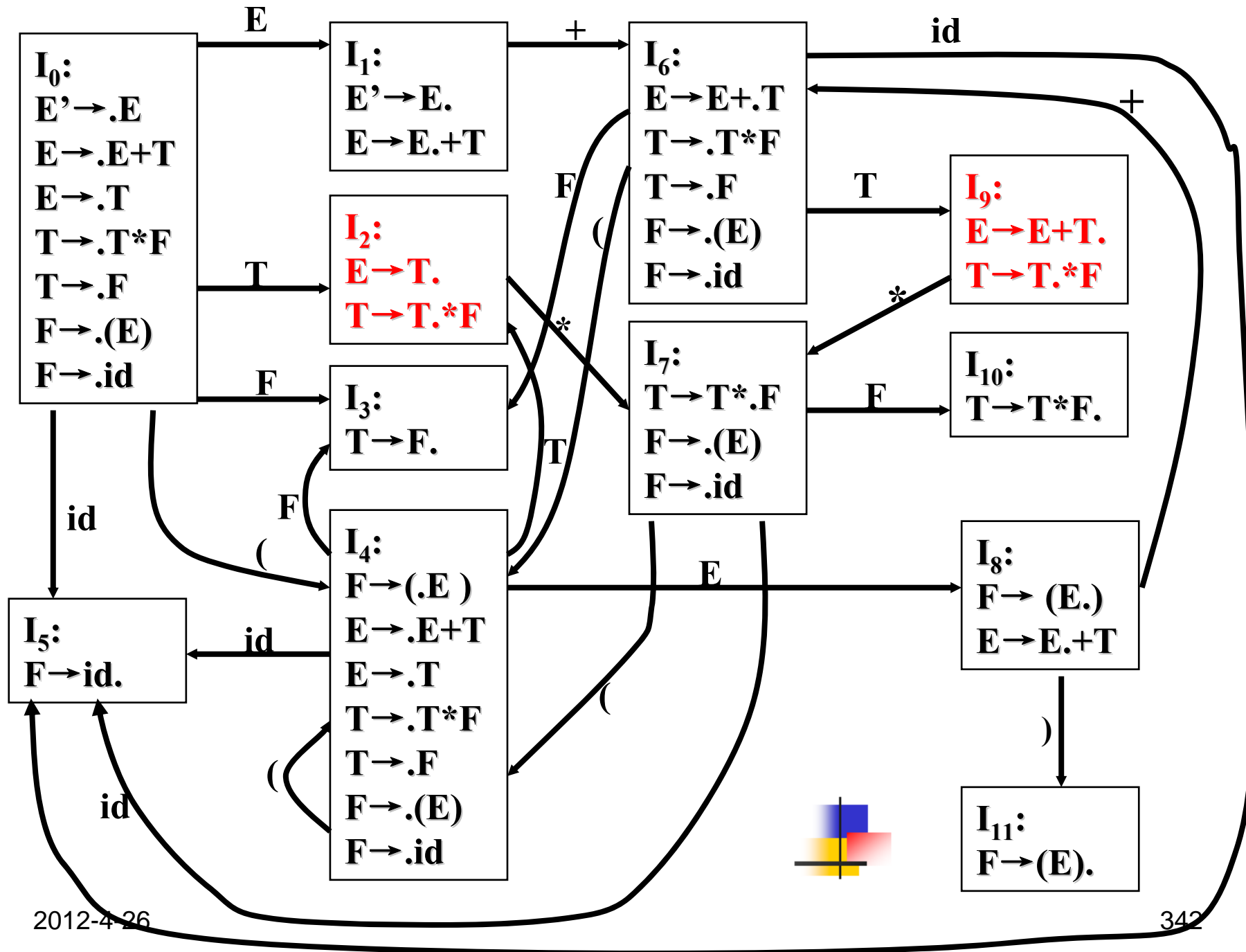
2)  $E \rightarrow T$

3)  $T \rightarrow T * F$

4)  $T \rightarrow F$

5)  $F \rightarrow (E)$

6)  $F \rightarrow \text{id}$



# 表达式文法的 LR(0)分析表含有冲突

状态	ACTION					
	id	+	*	(	)	#
2	r2	r2	r2/s7	r2	r2	r2
3	r4	r4	r4	r4	r4	r4
	...					
5	r6	r6	r6	r6	r6	r6
	...					
9	r1	r1	r1/s7	r1	r1	r1
10	r3	r3	r3	r3	r3	r3
11	r5	r5	r5	r5	r5	r5

- 在状态 2、9 采用归约，出现移进归约冲突



# 表达式文法的SLR(1)分析表

## ■ 求非终结符的 FIRST 集 和 FOLLOW 集

- $\text{FIRST}(F) = \{ \text{id}, ( \}$
- $\text{FIRST}(T) = \{ \text{id}, ( \}$
- $\text{FIRST}(E) = \{ \text{id}, ( \}$
- $\text{FOLLOW}(E) = \{ ), +, \# \}$
- $\text{FOLLOW}(T) = \{ ), +, \#, * \}$
- $\text{FOLLOW}(F) = \{ ), +, \#, * \}$

1)  $E \rightarrow E + T$

2)  $E \rightarrow T$

3)  $T \rightarrow T * F$

4)  $T \rightarrow F$

5)  $F \rightarrow (E)$

6)  $F \rightarrow \text{id}$



状态	ACTION						GOTO		
	i d	+	*	(	)	#	E	T	F
0	s 5			s 4			1	2	3
1		s 6				a c c			
2		r 2	s 7		r 2	r 2			
3		r 4	r 4		r 4	r 4			
4	s 5			s 4			8	2	3
5		r 6	r 6		r 6	r 6			
6	s 5			s 4				9	3
7	s 5			s 4					1 0
8		s 6			s 1 1				
9		r 1	s 7		r 1	r 1			
1 0		r 3	r 3		r 3	r 3			
1 1		r 5	r 5		r 5	r 5			

si 表示移进到状态i, ri 表示用i号产生式归约



# SLR(1) 分析的特点

---

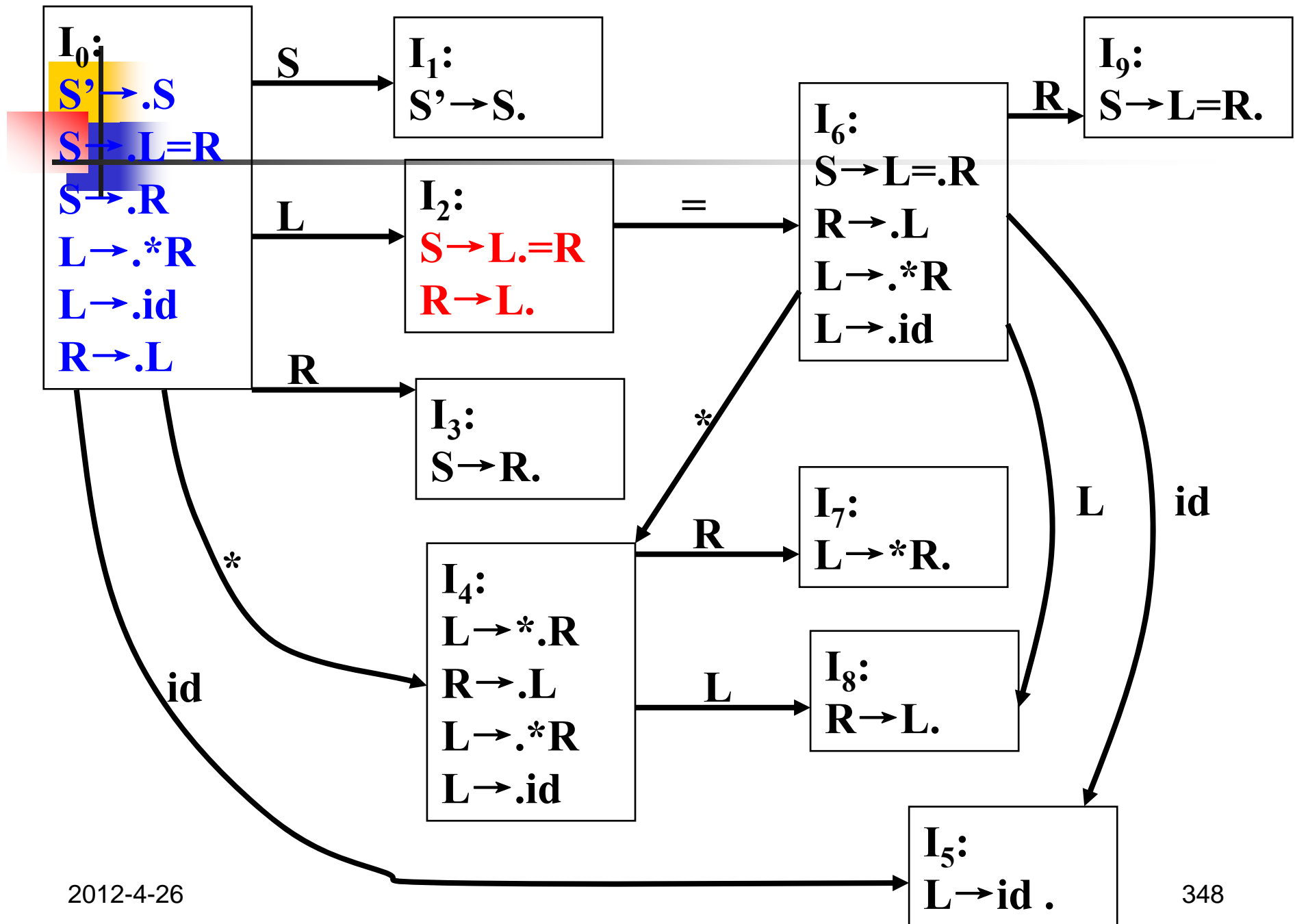
- 描述能力强于 LL(1)
  - SLR(1)还考虑Follow集中的符号
  - LL(1) 仅考虑产生式的首符号
- SLR(1) 文法: SLR(1)分析表无冲突的CFG



# SLR(1)分析的局限性

---

- 如果 SLR(1) 分析表仍有多重入口（移进归约冲突或归约归约冲突），则说明该文法不是 SLR(1) 文法；
- 说明仅使用 LR(0) 项目集和 FOLLOW 集还不足以分析这种文法



# SLR分析中的冲突——需要更强的分析方法

$$I_2 = \{S \rightarrow L.=R, R \rightarrow L.\}$$

- 输入符号为 = 时，出现了移进归约冲突：

$$S \rightarrow L.=R \in I_2 \text{ and } \text{go}(I_2, =) = I_6$$

$$\Rightarrow \text{action}[2, =] = \text{Shift } 6$$

$$R \rightarrow L. \in I_2 \text{ and } = \in \text{FOLLOW}(R) = \{=, \# \}$$

$$\Rightarrow \text{action}[2, =] = \text{Reduce } R \rightarrow L$$

- 说明该文法不是SLR(1)文法，分析这种文法需要更多的信息。

# SLR分析中存在冲突的原因

- SLR (1) 只孤立地考察输入符号是否属于归约项目  $A \rightarrow \alpha$  相关联的集合 FOLLOW (A)，而没有考察符号串  $\alpha$  所在规范句型的“上下文”。
- 所以试图用某一产生式  $A \rightarrow \alpha$  归约栈顶符号串  $\alpha$  时，不仅要向前扫描一个输入符号，还要查看栈中的符号串  $\delta \alpha$ ，只有当  $\delta Aa$  的确构成文法某一规范句型的活前缀时才能用  $A \rightarrow \alpha$  归约。亦即要考虑归约的有效性：
- 问题：怎样确定  $\delta Aa$  是否是文法某一规范句型的活前

缀



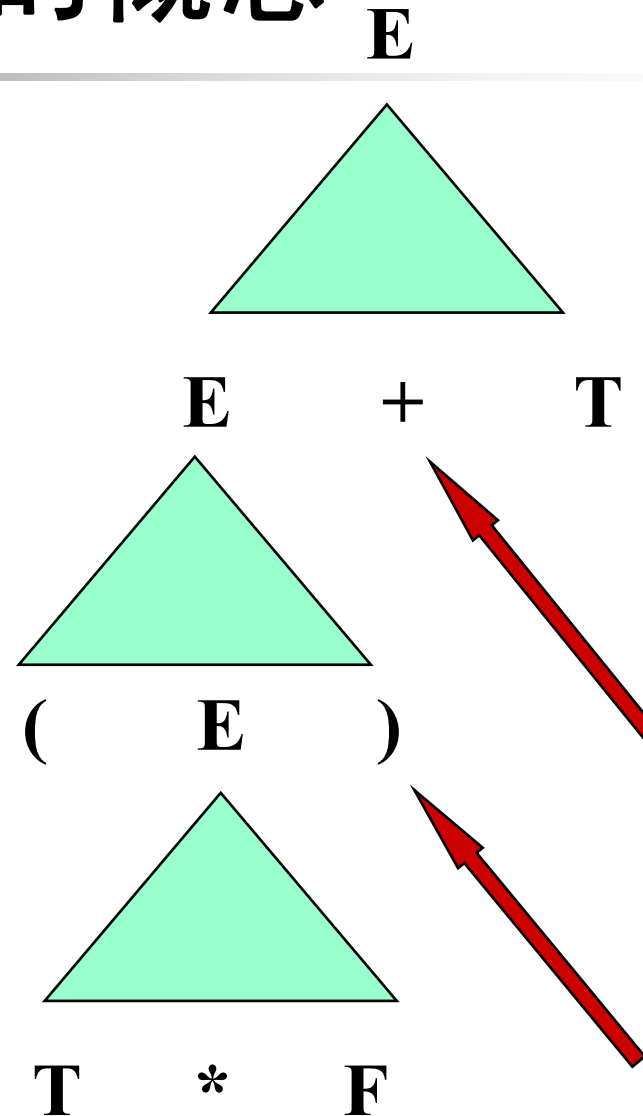
## 5.3.4 LR(1)分析表的构造

---

- LR(0)不考虑后继符(搜索符), SLR(1)仅在归约时考虑后继符(搜索符), 因此, 对后继符(搜索符)所含信息量的利用有限, 未考虑栈中内容。
- 希望在构造状态时就考虑后继符(搜索符)的作用: 考虑对于产生式  $A \rightarrow \alpha$  的归约, 不同使用位置的  $A$  会要求不同的后继符号

# 后继符(搜索符)的概念

- 不同的归约中有不同的后继符。
- 特定位置的后继符是 FOLLOW 集的子集







# LR(k) 项目

---

## ■ 定义5.11

- $[A \rightarrow \alpha . \beta , a_1 a_2 \dots a_k]$  为 **LR (k) 项目**，根据圆点所处位置的不同又分为三类：
  - 归约项目：  $[A \rightarrow \alpha . , a_1 a_2 \dots a_k]$
  - 移进项目：  $[A \rightarrow \alpha . a \beta , a_1 a_2 \dots a_k]$
  - 待约项目：  $[A \rightarrow \alpha . B \beta , a_1 a_2 \dots a_k]$
- 利用LR(k)项目进行(构造)LR(k)分析(器)，当  $k=1$  时，为LR(1)项目，相应的分析叫LR(1)分析(器)



# LR(1) 项目的有效性

## ■ 形式上

- 称LR(1)项目  $[A \rightarrow \alpha \cdot \beta, a]$  对活前缀  $\gamma = \delta \alpha$  是有效的, 如果存在规范推导
- $S \Rightarrow^* \delta A w \Rightarrow \delta \alpha \beta w$
- 其中  $a$  为  $w$  的首字符, 如果  $w = \varepsilon$ , 则  $a = \#$

- 与LR(0)文法类似, 识别文法全部活前缀的DFA的每一状态也是用一个LR(1)项目集来表示, 为保证分析时, 每一步都在栈中得到规范句型的活前缀, 应使每一个LR(1)项目集仅由若干个对相应活前缀有效的项目组成



# 识别文法全部活前缀的DFA

---

## ■ LR(1) 项目集族的求法

- **CLOSURE (I)** : 求I的闭包, 目的是为了合并某些状态, 节省空间
- **GO (I, X)** : 转移函数



# 闭包的计算

---

- **CLOSURE(I)的计算**
  - (核心位置:  $A \rightarrow \alpha . B \beta$ ,  $a$  扩展成闭包)
- 同时考虑可能出现的后继符
  - $b \in \text{FIRST}(\beta a)$



# 闭包的计算

---

- 如果  $[A \rightarrow \alpha . B \beta , a]$  对  $\gamma = \delta \alpha$  有效

/\*即存在  $S \Rightarrow^* \delta A a x \Rightarrow \delta \alpha \beta a x$ \*/

- 假定  $\beta a x \Rightarrow^* b y$ ，则对任意的  $B \rightarrow \eta$  有：
  - $[B \rightarrow . \eta , b]$  对  $\gamma = \delta \alpha$  也是有效的，其中
  - $b \in \text{FIRST} ( \beta a )$



# 闭包的计算

---

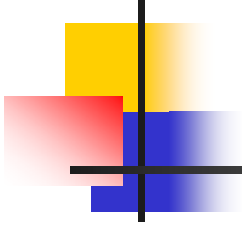
**$J := I;$**

**repeat**

**$J = J \cup \{ [B \rightarrow \cdot \eta, b] \mid [A \rightarrow \alpha \cdot \underline{B} \underline{\beta}, a] \in J, \\ b \in \underline{\text{FIRST}}(\underline{\beta a}) \}$**

**until J 不再扩大**

- 当  $\beta \Rightarrow^+ \varepsilon$  时，此时  $b=a$  叫继承的后继符，否则叫自生的后继符



# 状态 I 和文法符号 X 的转移函数

---

$\text{go}(I, X) =$

$\text{closure}([A \rightarrow \alpha X. \beta, \mathbf{a}] \mid [A \rightarrow \alpha .X \beta, \mathbf{a}] \in I)$

# 计算LR(1)项目集规范族 C

即：分析器状态集合

$C = \{I_0\} \cup \{I \mid \exists J \in C, X \in V \cup T, I = \text{go}(J, X)\}$  称为  $G'$  的 LR(1)项目集规范族 (算法: P181)

begin

$C := \{\text{closure}(\{S' \rightarrow \cdot S, \#\})\};$

repeat

for  $\forall I \in C, \forall X \in V \cup T$

if  $\text{go}(I, X) \neq \Phi$  &  $\text{go}(I, X) \notin C$  then

$C = C \cup \text{go}(I, X)$

until C不变化

end.





# 识别活前缀的关于LR(1) 的DFA

---

- 识别文法  $G = (V, T, P, S)$  的拓广文法  $G'$  的所有活前缀的DFA  $M = (C, V \cup T, go, I_0, C)$ 
  - $I_0 = \text{CLOSURE}(\{S' \rightarrow .S, \#\})$
- 如果CFG  $G$ 的LR(1)分析表无冲突则称 $G$ 为LR(1)文法

# LR(1) 分析表的构造

1. 令  $I_0 = \text{CLOSURE}(\{S' \rightarrow .S\})$ , 构造  $C = \{I_0, I_1, \dots, I_n\}$ , 即  $G'$  的 LR(1) 项目集规范族。
2. 从  $I_i$  构造状态  $i$ , 0 为初始状态。

for  $k=0$  to  $n$  do

begin

(1) if  $[A \rightarrow \alpha .a \beta, b] \in I_k$  &  $a \in T$  &  $\text{GO}(I_k, a) = I_j$  then  
     $\text{action}[k, a] := S_j$ ;

(2) if  $\text{GO}(I_k, B) = I_j$  &  $B \in V$  then  $\text{goto}[k, B] := j$ ;

(3) if  $[A \rightarrow \alpha ., a] \in I_k$  &  $A \rightarrow \alpha$  为  $G'$  的第  $j$  个产生式 then  
     $\text{action}[k, a] := r_j$ ;

(4) if  $[S' \rightarrow S., \#] \in I_k$  then  $\text{action}[k, \#] := \text{acc}$ ;

end

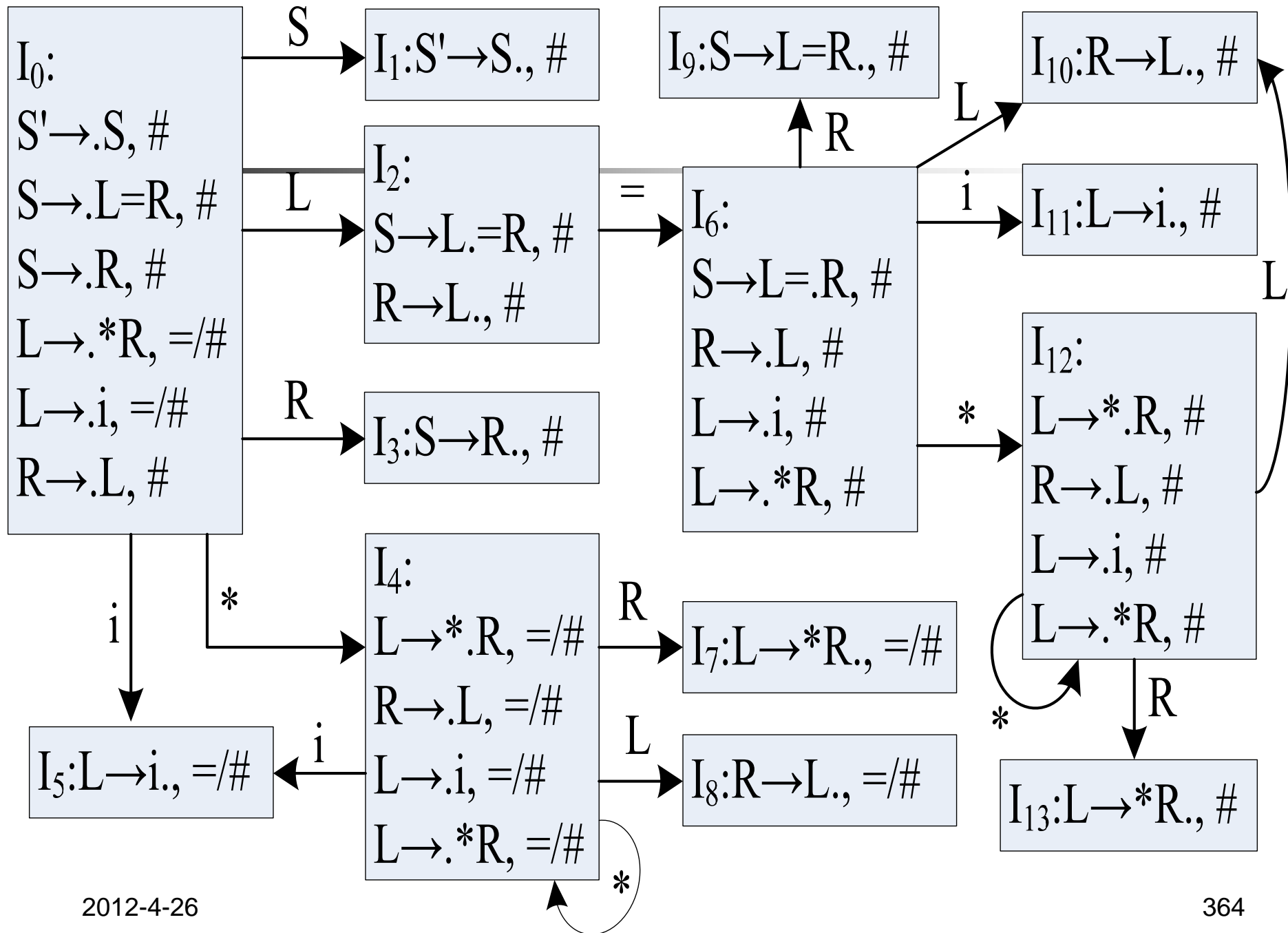
- 上述(1)到(4)步未填入信息的表项均置为 error。

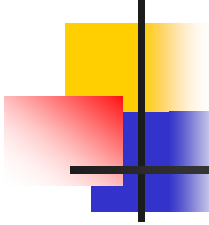


# LR(1) 分析表的构造

---

- 与LR(0)的不同点主要在归约动作的选择:
  - LR(0) 分析考虑所有终结符
  - SLR(1) 分析参考 FOLLOW 集
  - LR(1) 分析仅考虑 LR(1)项目中的后继符





## 5.3.5 LALR(1)分析表的构造

- LR(1)对应的C太大
- 问题：是否可以将某些闭包/状态合并？
  - 不同的LR(1)项目闭包可能有相同的LR(0)项目，但后继符可能不同——同心
  - 合并后可能带来归约归约冲突
  - 合并那些不会带来冲突的同心的LR(1)闭包/状态
- ( lookahead-LR )
  - 在不带来移进归约冲突的条件下，合并状态，重构分析表



# LALR(1) 的分析能力

---

- 强于 SLR(1)

- 合并的后继符仍为 FOLLOW 集的子集

- 局限性

- 合并中不出现归约-归约冲突
  - 如果CFG G的LALR(1)分析表无冲突则称G为LALR(1)文法

## 5.3.6 二义性文法的应用

**$I_1:$**

**$E' \rightarrow E.$**

**$E \rightarrow E . + E$**

**$E \rightarrow E . * E$**

**$I_7:$**

**$E \rightarrow E + E.$**

**$E \rightarrow E . + E$**

**$E \rightarrow E . * E$**

**$I_8:$**

**$E \rightarrow E * E.$**

**$E \rightarrow E . + E$**

**$E \rightarrow E . * E$**

- 采用二义性文法，可以减少结果分析器的状态数，并能减少对单非终结符（ $E \rightarrow T$ ）的归约。
- 在构造分析表时采用消除二义性的规则(按优先级)



## 5.3.6 二义性文法的应用

$I_4$ :  
 $S \rightarrow iS.eS$   
 $S \rightarrow iS.$

选择移进else,  
以便让它与前面  
的then配对





## 5.3.7 LR分析中的出错处理

- 当分析器处于某一状态 $S$ ，且当前输入符号为 $a$ 时，就以符号对 $(S, a)$ 查 $LR$ 分析表，如果分析表元素 $action[S, a]$ 为空(或出错)，则表示检测到了一个语法错误。
- 紧急方式的错误恢复：从栈顶开始退栈，直至发现在特定语法变量 $A$ 上具有转移的状态 $S$ 为止，然后丢弃零个或多个输入符号，直至找到符号 $a \in FOLLOW(A)$ 为止。接着，分析器把状态 $goto[S, A]$ 压进栈，并恢复正常分析。



# LR分析的基本步骤

---

- 1、编写拓广文法，求Follow集
- 2、求识别所有活前缀的DFA
- 3、构造LR分析表

## 5.4 语法分析程序的自动生成工具Yacc

### YSP(Yacc Specification)

%{变量定义：头文件和全局变量

%开始符号

词汇表： %Token  $n_1, n_2, \dots$  (自动定义种别码)

%Token  $n_1, i_1$  (用户指定种别码)

.....

%Token  $n_h, i_h$  (用户指定种别码)

类型说明    %type

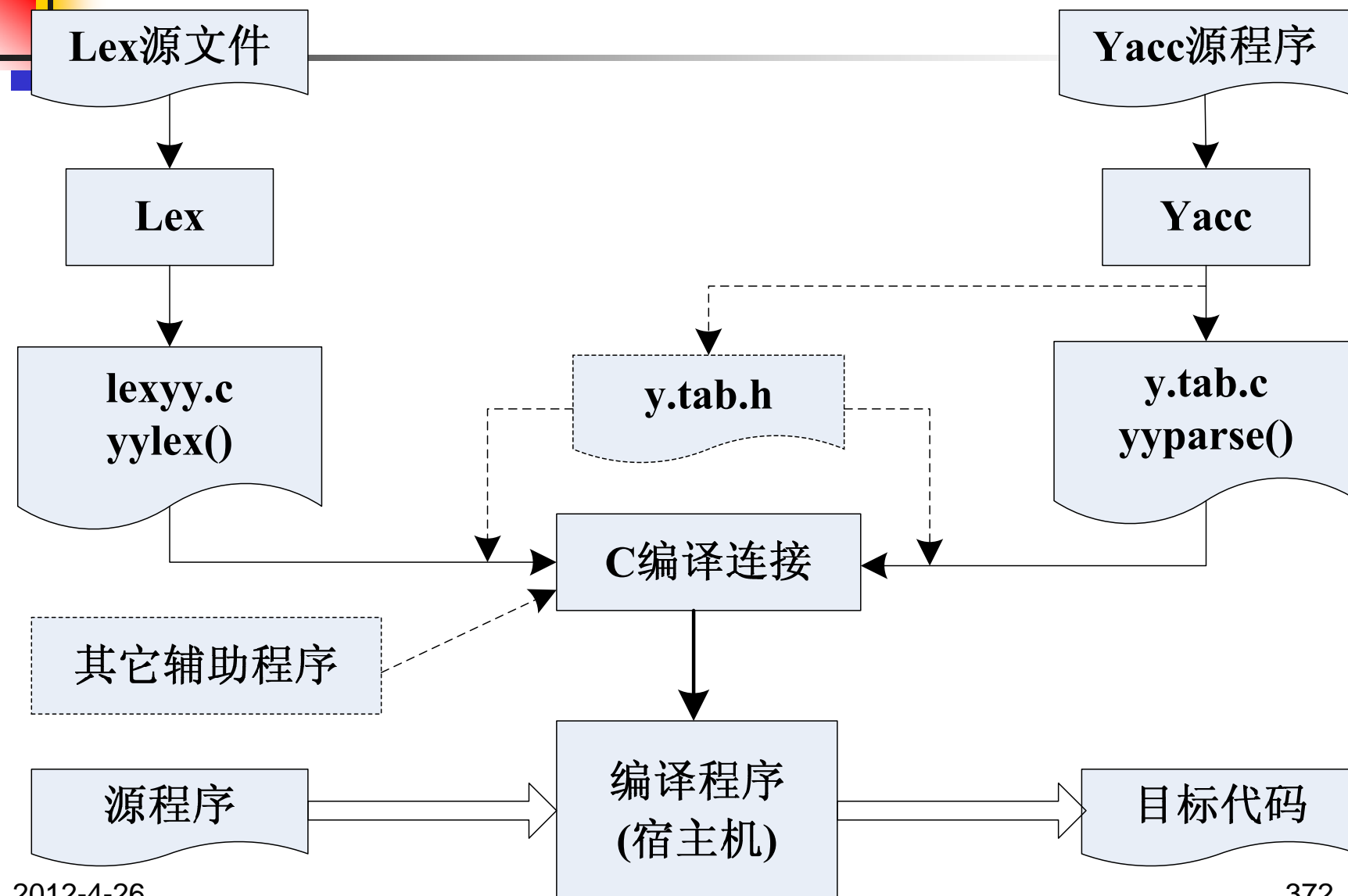
其它说明%}

%%规则部分    给出文法规则的描述

%%程序部分    扫描器和语义动作程序

■ 输出：LALR(1)分析器

# 用Yacc和Lex合建编译程序





# 本章小结

- 自底向上的语法分析从给定的输入符号串 $w$ 出发，自底向上地为其建立一棵语法分析树。
- 移进-归约分析是最基本的分析方式，分为优先法和状态法。
- 算符优先分析法是一种有效的方法，通过定义终结符号之间的优先关系来确定移进和归约。
- $LR$ 分析法有着更宽的适应性。该方法通过构建识别规范句型活前缀的DFA来设计分析过程中的状态。可以将 $LR$ 分析法分成 $LR(0)$ 、 $SLR(1)$ 、 $LR(1)$ 、 $LALR(1)$ 。
- 通过增加附加的信息可以解决一些二义性问题。
- Yacc是 $LALR(1)$ 语法分析器的自动生成工具。



*School of Computer Science & Technology*  
*Harbin Institute of Technology*



# 第六章 语法制导翻译与 属性文法

**重点:** 语法制导翻译的基本思想, 语法制导定义, 翻译模式, 自顶向下翻译, 自底向上翻译。

**难点:** 属性的意义, 对综合属性, 继承属性, 固有属性的理解, 属性计算, 怎么通过属性来表达翻译。





# 第6章 语法制导翻译与属性文法

---

## 6.1 语法制导翻译概述

## 6.2 语法制导定义

## 6.3 属性计算

## 6.4 翻译模式

## 6.5 本章小结



# 问题

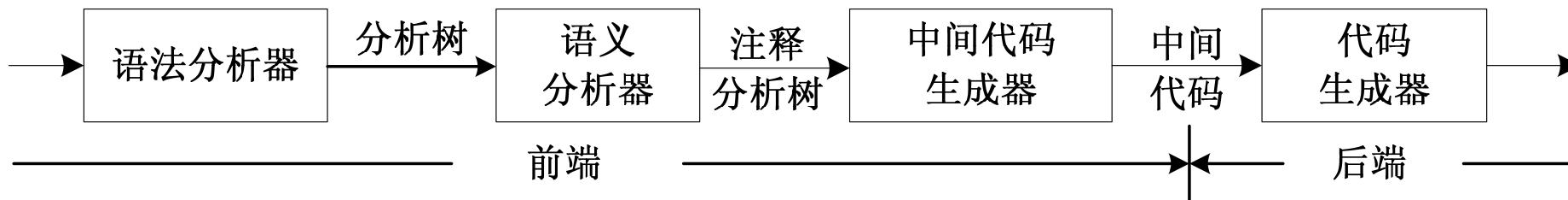
---

- 为什么进行词法和语法分析?
- 用  $A \rightarrow \alpha$  进行归约表达的是什么意思?
- 看: operand+term
- $E \rightarrow E_1 + T$
- $E_1$  的值 +  $T$  的值的结果作为  $E$  的值——即: 取来  $E_1$  的值和  $T$  的值做加法运算, 结果作为  $E$  的值
  - $E.val = E_1.val + T.val$



## 6.1 语法制导翻译概述

- 为了提高编译程序的可移植性，一般将编译程序划分为前端和后端。
  - 前端通常包括词法分析、语法分析、语义分析、中间代码生成、符号表的建立，以及与机器无关的中间代码优化等，它们的实现一般不依赖于具体的目标机器。
  - 后端通常包括与机器有关的代码优化、目标代码的生成、相关的错误处理以及符号表的访问等。





## 6.1 语法制导翻译概述

- 语义分析器的主要任务是检查各个语法结构的静态语义，称为静态语义分析或静态检查
  - 类型检查：操作数和操作符的类型是否相容；
  - 控制流检查：控制流转向的目标地址是否合法；
  - 惟一性检查：对象是否被重复定义；
  - 关联名检查：同一名字的多次特定出现是否一致。
- 将静态检查和中间代码生成结合到语法分析中进行的技术称为语法制导翻译。



## 6.1 语法制导翻译概述

### ■ 语法制导翻译的基本思想

- 在进行语法分析的同时，完成相应的语义处理。也就是说，一旦语法分析器识别出一个语法结构就要立即对其进行翻译，翻译是根据语言的语义进行的，并通过调用事先为该语法结构编写的语义子程序来实现。
- 对文法中的每个产生式附加一个/多个语义动作(或语义子程序)，在语法分析的过程中，每当需要使用一个产生式进行推导或归约时，语法分析程序除执行相应的语法分析动作外，还要执行相应的语义动作(或调用相应的语义子程序)。



## 6.1 语法制导翻译概述

---

### ■ 语义子程序的功能

- 指明相应产生式中各个文法符号的具体含义，并规定了使用该产生式进行分析时所应采取的语义动作(如传送或处理语义信息、查填符号表、计算值、生成中间代码等)。
- 语义信息的获取和加工是和语法分析同时进行的，而且这些语义信息是通过文法符号来携带和传递的。



## 6.1 语法制导翻译概述

---

- 一个文法符号 $X$ 所携带的语义信息称为 $X$ 的语义属性，简称为属性，它是根据翻译的需要设置的(对应分析树结点的数据结构)，主要用于描述语法结构的语义。
  - 一个变量的属性有类型、层次、存储地址等
  - 表达式的属性有类型、值等。



## 6.1 语法制导翻译概述

- 属性值的计算和产生式相关联，随着语法分析的进行，执行属性值的计算，完成语义分析和翻译的任务。
  - $E \rightarrow E_1 + E_2$        $E.val := E_1.val + E_2.val$
- 语法结构具有规定的语义
- 问题：如何根据被识别出的语法成分进行语义处理？
  - 亦即怎样将属性值的计算及翻译工作同产生式相关联？



# 典型处理方法一

## ■ 语法制导定义

- 通过将属性与文法符号关联、将语义规则与产生式关联来描述语言结构的翻译方案
- 对应每一个产生式编写一个语义子程序，当一个产生式获得匹配时，就调用相应的语义子程序来实现语义检查与翻译

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

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

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

## ■ 适宜在完成归约的时候进行



# 典型处理方法二

## ■ 翻译模式

- 通过将属性与文法符号关联，并将语义规则插入到产生式的右部来描述语言结构的翻译方案
- 在产生式的右部的适当位置，插入相应的语义动作，按照分析的进程，执行遇到的语义动作
- $D \rightarrow T \{ L.inh := T.type \} L$
- $T \rightarrow \text{int} \{ T.type := \text{integer} \}$
- $T \rightarrow \text{real} \{ T.type := \text{real} \}$
- $L \rightarrow \{ L_1.inh := L.inh \} L_1, \text{id} \{ \text{addtype}(\text{id.entry}, L.inh) \}$
- $L \rightarrow \text{id} \{ \text{addtype}(\text{id.entry}, L.inh) \}$





## 6.2 语法制导定义

- 语法制导定义是附带有属性和语义规则的上下文无关文法
  - 属性是与文法符号相关联的语义信息
  - 语义规则是与产生式相关联的语义动作
- 语法制导定义是基于语言结构的语义要求设计的，类似于程序设计，语法制导定义中的属性类似于程序中用到的数据结构，用于描述语义信息，语义规则类似于计算，用于收集、传递和计算语义信息的。
- 属性通常被保存在分析树的相关节点中



# 概念术语

---

- 综合属性：节点的属性值是通过分析树中该节点或其子节点的属性值计算出来的
- 继承属性：节点的属性值是由该节点、该节点的兄弟节点或父节点的属性值计算出来的
- 固有属性：通过词法分析直接得到的属性
- 依赖图：描述属性之间依赖关系的图，根据语义规则来构造
- 注释分析树：节点带有属性值的分析树



# 语法制导定义的形式

- 在一个语法制导定义中,  $\forall A \rightarrow \alpha \in P$  都有与之相关联的一套语义规则, 规则形式为

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

$f$  是一个函数, 而且或者

1.  $b$  是  $A$  的一个综合属性并且  $c_1, c_2, \dots, c_k$  是  $\alpha$  中的符号的属性, 或者

2.  $b$  是  $\alpha$  中某个符号的一个继承属性并且  $c_1, c_2, \dots, c_k$  是  $A$  或  $\alpha$  中的任何文法符号的属性。

这两种情况下, 都说属性  $b$  依赖于属性  $c_1, c_2, \dots, c_k$



## 例6.1 台式计算器的语法制导定义

---

产生式

$L \rightarrow En$

$E \rightarrow E_1 + T$

$E \rightarrow T$

$T \rightarrow T_1 * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow \text{digit}$

语义规则

$\text{print}(E.val)$  (可看作是 $L$ 的虚属性)

$E.val := E_1.val + T.val$

$E.val := T.val$

$T.val := T_1.val + F.val$

$T.val := F.val$

$F.val := E.val$

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



# S-属性定义

- 只含综合属性的语法制导定义称为**S-属性定义**
- 对于S-属性定义，通常使用自底向上的分析方法，在建立每一个结点处使用语义规则来计算综合属性值，即在用哪个产生式进行归约后，就执行那个产生式的S-属性定义计算属性的值，从叶结点到根结点进行计算。
- 没有副作用的语法制导定义有时又称为**属性文法**，属性文法的语义规则单纯根据常数和其它属性的值来定义某个属性的值



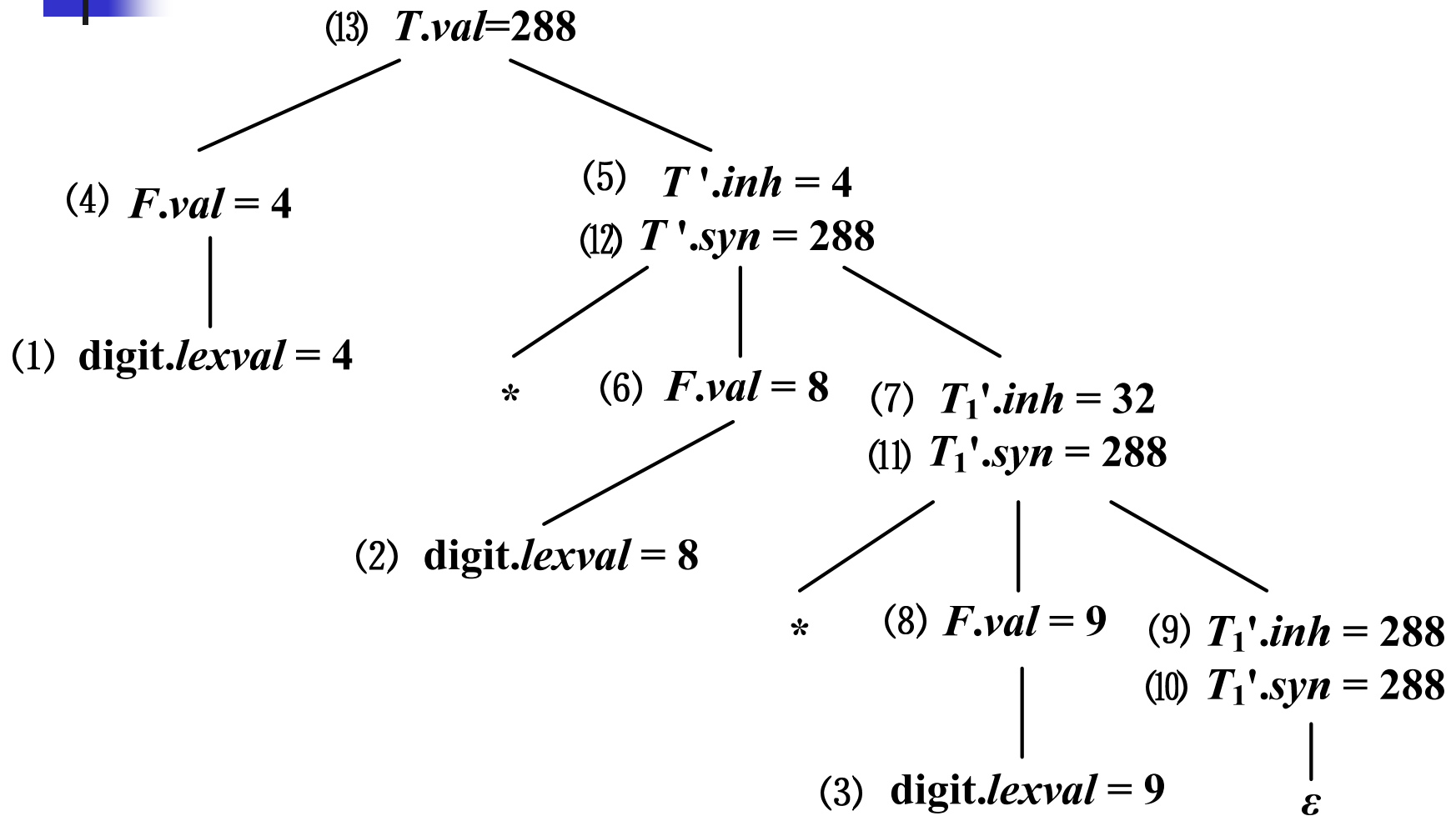
# 继承属性

- 当分析树的结构同源代码的抽象语法不“匹配”时，继承属性将非常有用。下面的例子可以说明怎样用继承属性来解决这种不匹配问题，产生这种不匹配的原因是因为文法通常是为语法分析而不是为翻译设计的。
- 例6.2
  - 考虑如何在自顶向下的分析过程中计算 $3*5$ 和 $4*8*9$ 这样的表达式项
  - 消除左递归之后的算数表达式文法的一个子集：
$$T \rightarrow FT' \quad T' \rightarrow *FT' \quad T' \rightarrow \varepsilon \quad F \rightarrow \text{digit}$$

## 表6.3 为适于自顶向下分析的文法设计的语法制导定义

产生式	语义规则
$T \rightarrow FT'$	$T'.inh := F.val$ $T.val := T'.syn$
$T' \rightarrow *FT_1'$	$T_1'.inh := T'.inh \times F.val$ $T'.syn := T_1'.syn$
$T' \rightarrow \varepsilon$	$T'.syn := T'.inh$
$F \rightarrow \text{digit}$	$F.val := \text{digit.lexval}$

# 4\*8\*9的注释分析树





## 表6.3中语法制导定义对应的翻译模式

- 如果对 $4*8*9$ 进行自顶向下的语法制导翻译，则 $val$ 的值的计算顺序为(1)(2)(3)(4)(5)(6)(7)(8)(9)(10)(11)(12)(13)
- 根据上述对 $val$ 值的计算顺序，可以将表6.3中的语法制导定义转换成如下的翻译模式
- $T \rightarrow F\{T'.inh := F.val\}T'\{T.val := T'.syn\}$
- $T' \rightarrow *F\{T_1'.inh := T'.inh \times F.val\}T_1'\{T'.syn := T_1'.syn\}$
- $T' \rightarrow \varepsilon\{T'.syn := T'.inh\}$
- $F \rightarrow \text{digit}\{F.val := \text{digit.lexval}\}$



## 6.3 属性计算

- 语义规则定义了属性之间的依赖关系，这种依赖关系将影响属性的计算顺序
- 为了确定分析树中各个属性的计算顺序，我们可以用图来表示属性之间的依赖关系，并将其称为依赖图
- 如果依赖图中没有回路，则利用它可以很方便地求出属性的计算顺序。
- 注释分析树只是给出了属性的值，而依赖图则可以帮助我们确定如何将这属性值计算出来。



## 6.3.1 依赖图

- 所谓依赖图其实就是一个有向图，用于描述分析树中节点的属性和属性间的相互依赖关系，称为分析树的依赖图。
- 每个属性对应依赖图中的一个节点，如果属性 $b$ 依赖于属性 $c$ ，则从属性 $c$ 的节点有一条有向边指向属性 $b$ 的节点。
- 属性间的依赖关系是根据相应的语义规则确定的。



# 依赖图的构造方法

---

**for** 分析树的每个节点  $n$  **do**

**for** 与节点  $n$  对应的文法符号的每个属性  $a$  **do**

        在依赖图中为  $a$  构造一个节点;

**for** 分析树的每个节点  $n$  **do**

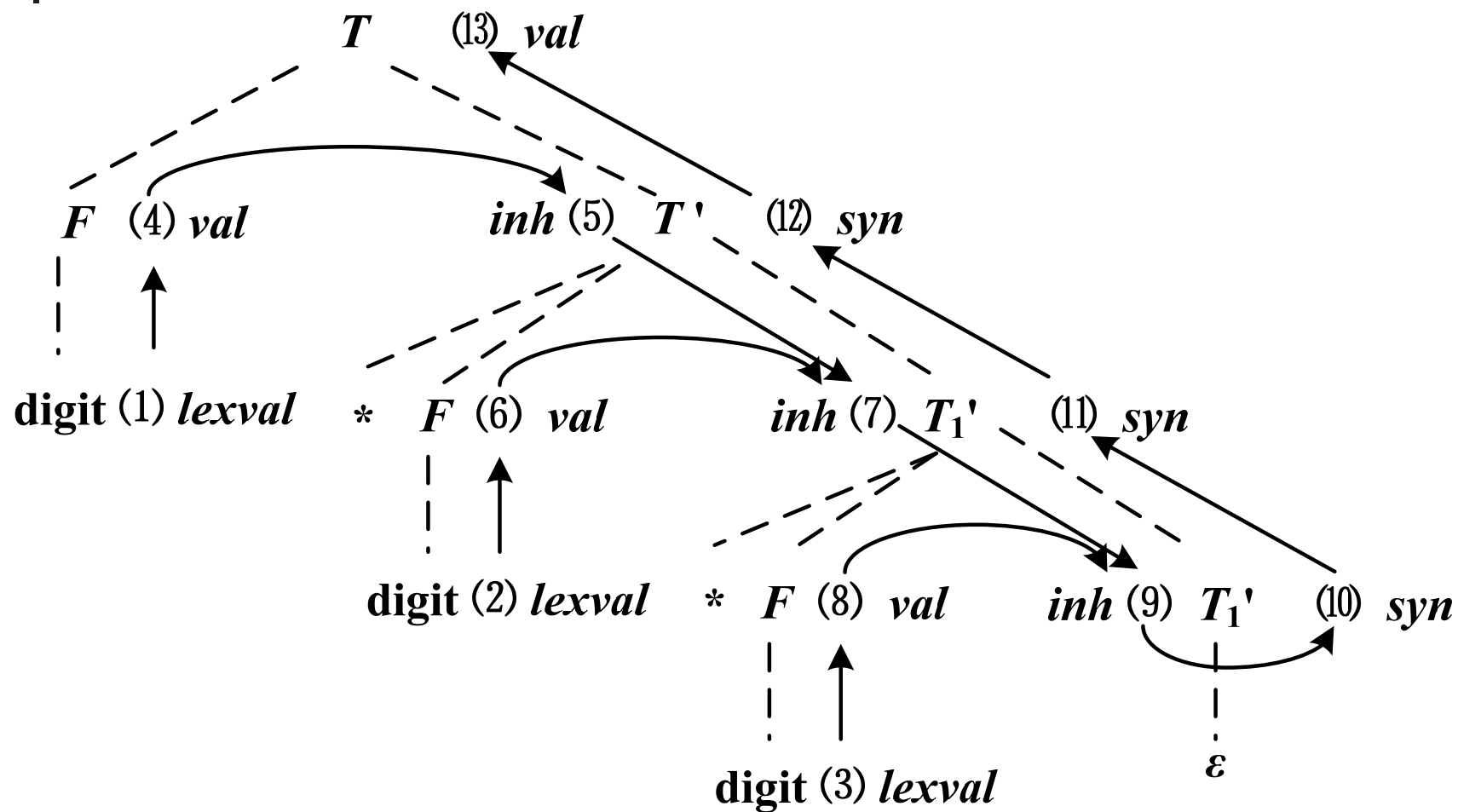
**for** 节点  $n$  所用产生式对应的每条语义规则

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

**for**  $i := 1$  to  $k$  **do**

            构造一条从节点  $c_i$  到节点  $b$  的有向边;

## 例6.3 图6.3中注释分析树的依赖图





## 6.3.2 属性的计算顺序

### ■ 拓扑排序

- 一个无环有向图的拓扑排序是图中结点的任何顺序  $m_1, m_2, \dots, m_k$ , 使得边必须是从序列中前面的结点指向后面的结点, 也就是说, 如果  $m_i \rightarrow m_j$  是  $m_i$  到  $m_j$  的一条边, 那么在序列中  $m_i$  必须出现在  $m_j$  的前面。
- 若依赖图中无环, 则存在一个拓扑排序, 它就是属性值的计算顺序。
- 例6.4 图6.4的拓扑排序为:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13

## 6.3.2 属性的计算顺序

### ■ 根据拓扑排序得到的翻译程序

- $a4:=4$                        $a5:=a4$                        $a6:=8$
- $a7:=a5 \times a6$        $a8:=9$                        $a9:=a7 \times a8$
- $a10:=a9$                $a11:=a10$                        $a12:=a11$
- $a13:=a12$

- 上述属性计算方法又称为**分析树法**，这种方法在编译时需要显式地构造分析树和依赖图，所以编译的时空效率比较低，而且如果分析树的依赖图中存在回路的话，这种方法将会失效。
- 这种方法的优点是可以多次遍历分析树，从而使得属性的计算不依赖于所采用的语法分析方法以及属性间严格的依赖关系。



# 计算语义规则的其他方法

## ■ 基于规则的方法

- 在构造编译器时，用手工或专门的工具来分析语义规则,确定属性值的计算顺序。

## ■ 忽略语义规则的方法

- 在分析过程中翻译，那么计算顺序由分析方法来确定而表面上与语义规则无关。这种方法限制了能被实现的语法制导定义的种类。

## ■ 这两种方法都不必显式构造依赖图，因此时空效率更高。





# S-属性定义

- 定义6.1 只含综合属性的语法制导定义称为S-属性定义，又称为S-属性文法。
- 如果某个语法制导定义是S-属性定义，则可以按照自下而上的顺序来计算分析树中节点的属性。
- 一种简单的属性计算方法是对分析树进行后根遍历，并在最后一次遍历节点 $N$ 时计算与节点 $N$ 相关联的属性。
  - $postorder(N) \{$
  - for  $N$ 的每个子节点 $M$ (从左到右)  $postorder(M);$
  - 计算与节点 $N$ 相关联的属性;
  - $\}$



# L-属性定义

- 定义6.2 一个语法制导定义被称为*L*-属性定义，当且仅当它的每个属性或者是综合属性，或者是满足如下条件的继承属性：设有产生式 $A \rightarrow X_1X_2...X_n$ ，其右部符号 $X_i (1 \leq i \leq n)$ 的继承属性只依赖于下列属性：
  - (1)  $A$ 的继承属性；
  - (2) 产生式中 $X_i$ 左边的符号 $X_1, X_2, ..., X_{i-1}$ 的综合属性或继承属性；
  - (3)  $X_i$ 本身的综合属性或继承属性，但前提是 $X_i$ 的属性不能在依赖图中形成回路。
- *L*-属性定义又称为*L*-属性文法。



## 表6.3 L-属性定义示例

产生式	语义规则
$T \rightarrow FT'$	$T'.inh := F.val$ $T.val := T'.syn$
$T' \rightarrow *FT_1'$	$T_1'.inh := T'.inh \times F.val$ $T'.syn := T_1'.syn$
$T' \rightarrow \varepsilon$	$T'.syn := T'.inh$
$F \rightarrow \text{digit}$	$F.val := \text{digit.lexval}$

## 例6.7 不是L-属性定义的语法制导定义

产生式	语义规则
$A \rightarrow BC$	$A.syn := B.b$ $B.inh := f(C.c, A.syn)$

语义规则 $B.inh := f(C.c, A.syn)$ 定义了一个继承属性，所以整个语法制导定义就不是S-属性定义了。此外，虽然这条语义规则是合法的属性定义规则，但不满足L-属性定义的要求。这是因为：属性 $B.inh$ 的定义中用到了属性 $C.c$ ，而 $C$ 在产生式的右部处在 $B$ 的右边。虽然在L-属性定义中可以使用兄弟节点的属性来定义某个属性，但这些兄弟节点必须是左兄弟节点才行。因此，该语法制导定义也不是L-属性定义。



# L-属性定义中的属性计算

---

```
visit(N) {  
  for N的每个子节点M(从左到右) {  
    计算节点M的继承属性;  
    visit (M);  
  }  
  计算节点N的综合属性;  
};
```

## 6.3.5 属性计算示例—抽象语法树的构造

**抽象语法树**是表示程序层次结构的树，它把分析树中对语义无关紧要的成分去掉，是分析树的抽象形式，也称作语法结构树，或结构树。

语法树是常用的一种**中间表示**形式。

把语法分析和翻译分开。语法分析过程中完成翻译有许多优点，但也有一些不足：

1. 适于语法分析的文法可能不完全反映语言成分的自然层次结构；
2. 由于语法分析方法的限制，对分析树结点的访问顺序和翻译需要的访问顺序不一致。





# 构造表达式的语法树

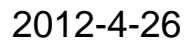
构造表达式的语法树使用的函数

1. *mknode*(*op*, *left*, *right*) 建立一个标记为*op*的运算符结点，两个域*left*和*right*分别是指向左右运算对象的指针。
2. *mkleaf*(*id*, *entry*) 建立一个标记为*id*的标识符结点，其域*entry*是指向该标识符在符号表中相应表项的指针。
3. *mkleaf*(*num*, *val*) 建立一个标记为*num*的数结点，其域*val*用于保存该数的值。



# 构造表达式语法树的语法制导定义

产生式	语义规则
(1) $T \rightarrow T_1 * F$	$T.node := mknode('*', T_1.node, F.node)$
(2) $T \rightarrow T_1 / F$	$T.node := mknode('/', T_1.node, F.node)$
(3) $T \rightarrow F$	$T.node := F.node$
(4) $F \rightarrow (E)$	$F.node := E.node$
(5) $F \rightarrow id$	$F.node := mkleaf(id, id.entry)$
(6) $F \rightarrow num$	$F.node := mkleaf(num, num.val)$





## 3\*x/y的抽象语法树的构造步骤

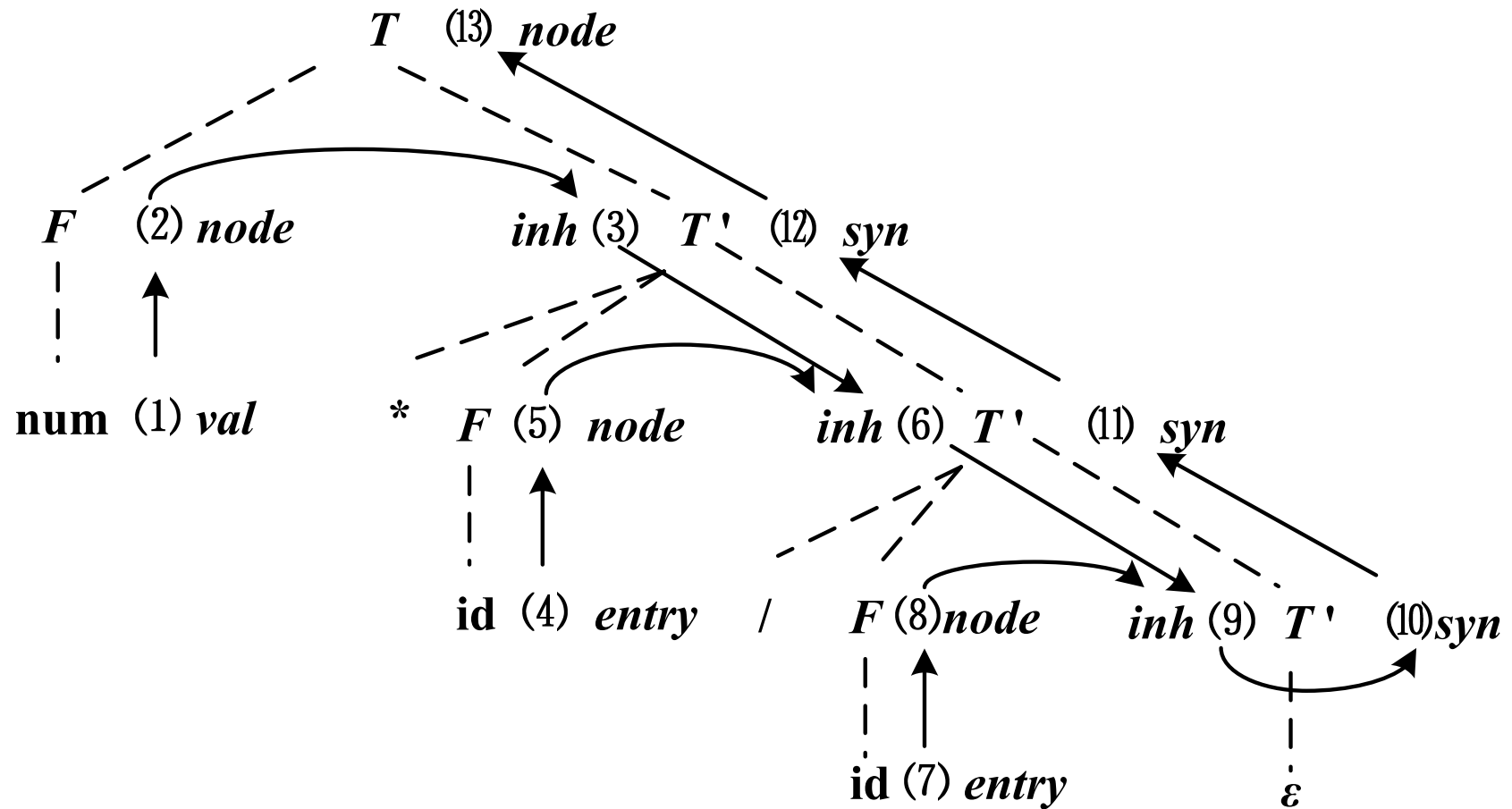
```
p1 := mkleaf(num, 3);  
p2 := mkleaf(id, entry-x);  
p3 := mknode('*', p1, p2);  
p4 := mkleaf(id, entry-y);  
p5 := mknode('/', p3, p4);
```

图6.5的语法树是自底向上构造的，对于那些为便于进行自顶向下分析而设计的文法来说，使用同样的步骤照样可以建立图6.5中的抽象语法树。当然，分析树的结构可能大不相同，而且可能需要引入继承属性来传递语义信息。

# 在自顶向下分析过程中构造语法树

产生式	语义规则
(1) $T \rightarrow FT'$	$T.node := T'.syn$ $T'.inh := F.node$
(2) $T' \rightarrow *FT_1'$	$T_1'.inh := mknode('*', T'.inh, F.node)$ $T'.syn := T_1'.syn$
(3) $T' \rightarrow /FT_1'$	$T_1'.inh := mknode('/', T'.inh, F.node)$ $T'.syn := T_1'.syn$
(4) $T' \rightarrow \varepsilon$	$T'.syn := T'.inh$
(5) $F \rightarrow (E)$	$F.node := E.node$
(6) $F \rightarrow id$	$F.node := mkleaf(id, id.entry)$
(7) $F \rightarrow num$	$F.node := mkleaf(num, num.val)$

# 根据表6.6的语法制导定义构造的语法树





## 6.4 翻译模式

- 定义

**翻译模式**是语法制导定义的一种便于实现的书写形式。其中属性与文法符号相关联，语义规则或语义动作作用花括号 { } 括起来，并可被插入到产生式右部的任何合适的位置上。

这是一种语法分析和语义动作交错的表示法，它表达在按深度优先遍历分析树的过程中何时执行语义动作。

**翻译模式给出了使用语义规则进行计算的顺序。**  
可看成是分析过程中翻译的注释。



## 例6.10 一个简单的翻译模式

将中缀表达式翻译成后缀表达式:

$$E \rightarrow TR$$

$$R \rightarrow \text{addop } T \{ \text{print}(\text{addop.lexeme}) \} R_1 \mid \varepsilon$$

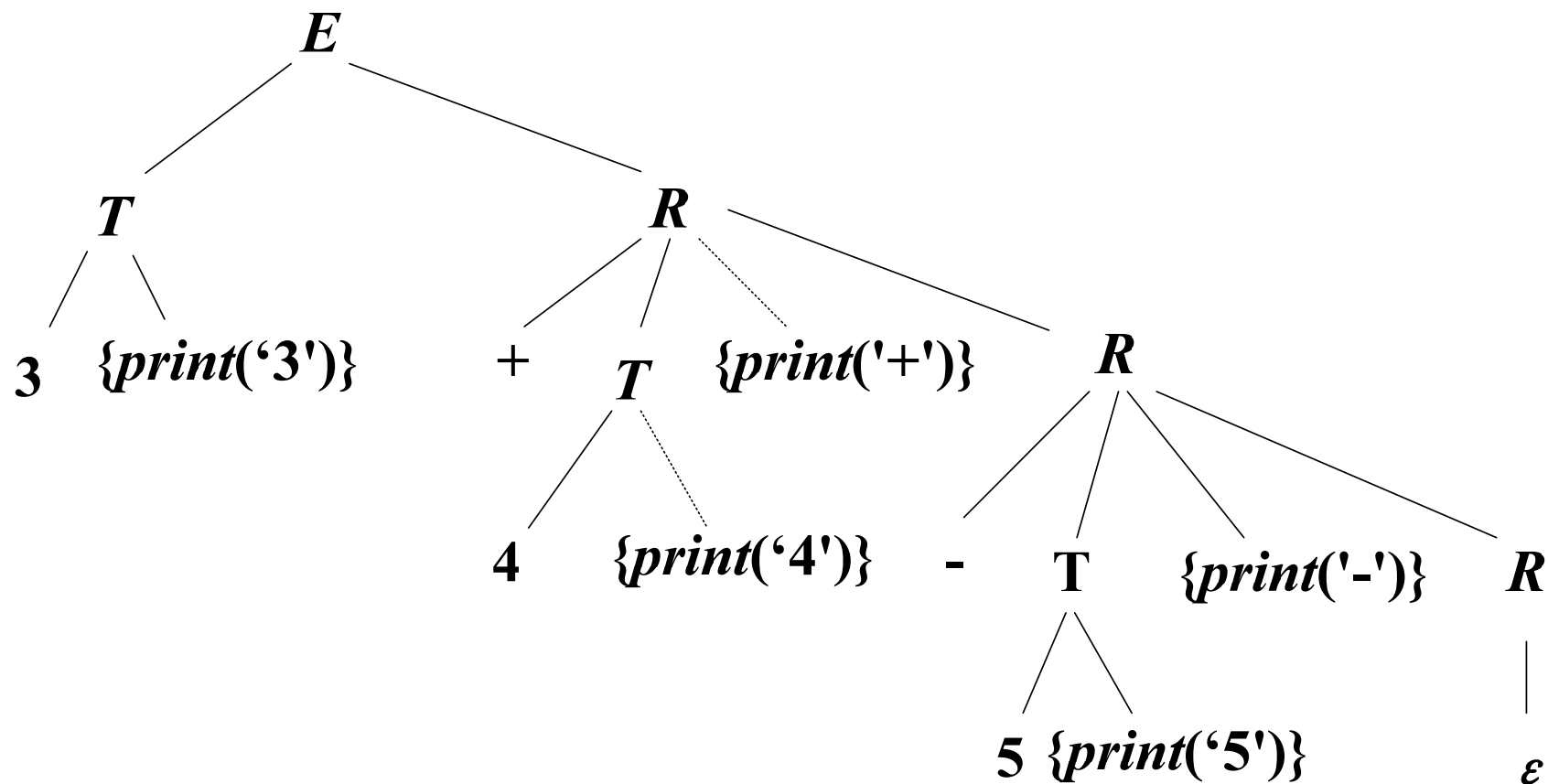
$$T \rightarrow \text{num} \{ \text{print}(\text{num.val}) \}$$

把语义动作看成终结符号，输入3+4-5,其分析树如图6.8，当按深度优先遍历它，执行遍历中访问的语义动作，将输出

3 4 + 5 -

它是输入表达式3+4-5的后缀式。

图6.8 3+4-5的带语义动作的分析树





# 翻译模式的设计

——根据语法制导定义

## ● 前提——语法制导定义是L-属性定义

保证语义动作不会引用还没计算出来的属性值

### 1. 只需要综合属性的情况

为每一个语义规则建立一个包含赋值的动作，并把该动作放在相应的产生式右部的末尾。

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

转换成：

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

# 翻译模式的设计

——根据语法制导定义

## 2. 既有综合属性又有继承属性

- 产生式右边的符号的继承属性必须在这个符号以前的动作中计算出来。
- 一个动作不能引用这个动作右边符号的综合属性。
- 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以后才能计算。计算这种属性的动作通常可放在产生式右端的末尾。

下面的翻译模式不满足要求:

$$S \rightarrow A_1 A_2 \quad \{A_1 \cdot in := 1; \quad A_2 \cdot in := 2\}$$
$$A \rightarrow a \quad \{ \textit{print}(A \cdot in) \} \quad /* A \cdot in \text{尚无定义} */$$

例6.11 从 $L$ -属性制导定义建立一个满足上面要求的翻译模式。

使用文法产生的语言是数学排版语言EQN

$E \text{ sub } 1 \cdot val$

编排结果



## $B$ 表示盒子

- (1)  $B \rightarrow B_1 B_2$  代表两个相邻盒子的并列，且  $B_1$  位于  $B_2$  的左边。
  - (2)  $B \rightarrow B_1 \text{ sub } B_2$  代表盒子  $B_1$  后随下标盒子  $B_2$ ，下标盒子  $B_2$  以较小的字体和较低的位置出现。
  - (3)  $B \rightarrow (B_1)$  代表一个由括号括起来的盒子  $B_1$ ，主要是为了对多个盒子或下标进行分组。在EQN中，使用花括号进行分组，此处使用圆括号是为了避免跟语义动作外面的花括号产生冲突。
  - (4)  $B \rightarrow \text{text}$  代表文本字符串，即任意字符组成的串。
- 该文法是二义性的文法，将“并列”和“下标”看成是左结合的，并令“下标”的优先级高于“并列”的话，则可以对该文法所描述的语言进行自底向上的语法分析。



## 属性设置

- (1) **point size**用于表示盒子中文本的尺寸(以点来计算, 也就是字号)。如果标准盒子的尺寸为 $p$ , 则下标盒子的尺寸为 $0.7 \times p$ 。属性 $B.ps$ 表示盒子 $B$ 的尺寸, 该属性是继承属性。
- (2) 每个盒子都有一个基线(**baseline**), 用来表示每个文本底部的垂直位置。
- (3) **height**用来表示从盒子的顶部到基线的距离。属性 $B.ht$ 表示盒子 $B$ 的高度 $height$ , 该属性是综合属性。
- (4) **depth**用来表示从基线到盒子底部的距离。用属性 $B.dp$ 表示盒子 $B$ 的深度 $depth$ , 该属性也是综合属性。

# 表6.7 对盒子进行排版的语法制导定义

产生式	语义规则
(1) $S \rightarrow B$	$B.ps := 10$ $S.ht := B.ht$ $S.dp := B.dp$
(2) $B \rightarrow B_1 B_2$	$B_1.ps := B.ps$ $B_2.ps := B.ps$ $B.ht := \max(B_1.ht, B_2.ht)$ $B.dp := \max(B_1.dp, B_2.dp)$
(3) $B \rightarrow B_1 \text{ sub } B_2$	$B_1.ps := B.ps$ $B_2.ps := 0.7 \times B.ps$ $B.ht := \max(B_1.ht, B_2.ht - 0.25 \times B.ps)$ $B.dp := \max(B_1.dp, B_2.dp + 0.25 \times B.ps)$
(4) $B \rightarrow (B_1)$	$B_1.ps := B.ps$ $B.ht := B_1.ht$ $B.dp := B_1.dp$
(5) $B \rightarrow \text{text}$ 2012-4-26	$B.ht := \text{getheight}(B.ps, \text{text.lexval})$ $B.dp := \text{getdepth}(B.ps, \text{text.lexval})$



## 从表6.7构造的翻译模式

$$S \rightarrow \{B.ps := 10\} B \{S.ht := B.ht; S.dp := B.dp\}$$
$$B \rightarrow \{B_1.ps := B.ps\} B_1 \{B_2.ps := B.ps\}$$
$$B_2 \{B.ht := \max(B_1.ht, B_2.ht)\}$$
$$B \rightarrow \{B_1.ps := B.ps\} B_1 \text{sub} \{B_2.ps := 0.7 \times B.ps\}$$
$$B_2 \{B.ht := \max(B_1.ht, B_2.ht - 0.25 \times B.ps);$$
$$B.dp := \max(B_1.dp, B_2.dp + 0.25 \times B.ps); \}$$
$$B \rightarrow (\{B_1.ps := B.ps\} B_1) \{B.ht := B_1.ht; B.dp := B_1.dp; \}$$
$$B \rightarrow \text{text} \{B.ht := \text{getheight}(B.ps, \text{text.lexval});$$
$$B.dp := \text{getdepth}(B.ps, \text{text.lexval}) \}$$



## 从表6.6构造的翻译模式

$$T \rightarrow F \{T'.inh := F.node\} T' \{T.node := T'.syn\}$$
$$T' \rightarrow *F \{T_1'.inh := mknode('*', T'.inh, F.node)\}$$
$$T_1' \{T'.syn := T_1'.syn\}$$
$$T' \rightarrow /F \{T_1'.inh := mknode('/', T'.inh, F.node)\}$$
$$T_1' \{T'.syn := T_1'.syn\}$$
$$T' \rightarrow \varepsilon \{T'.syn := T'.inh\}$$
$$F \rightarrow (E) \{F.node := E.node\}$$
$$F \rightarrow id \{F.node := mkleaf(id, id.entry)\}$$
$$F \rightarrow num \{F.node := mkleaf(num, num.val)\}$$




## 6.4.2 S-属性定义的自底向上计算

在分析栈中使用一个附加的域来存放综合属性值。下图为一个带有综合属性值域的分析栈：

<i>stack</i>		<i>val</i>
...	...	
<i>X</i>		<i>X.x</i>
<i>Y</i>		<i>Y.y</i>
<i>Z</i>		<i>Z.z</i>
...	...	

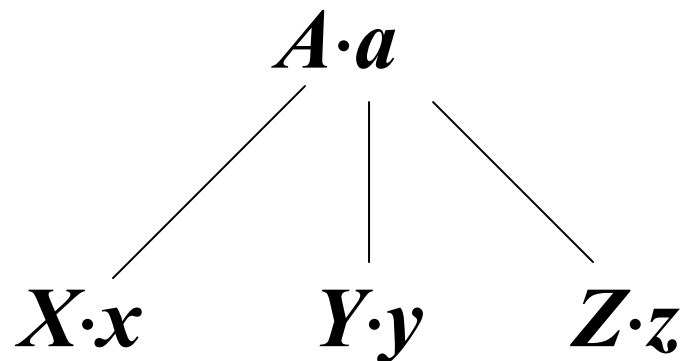
*top* →

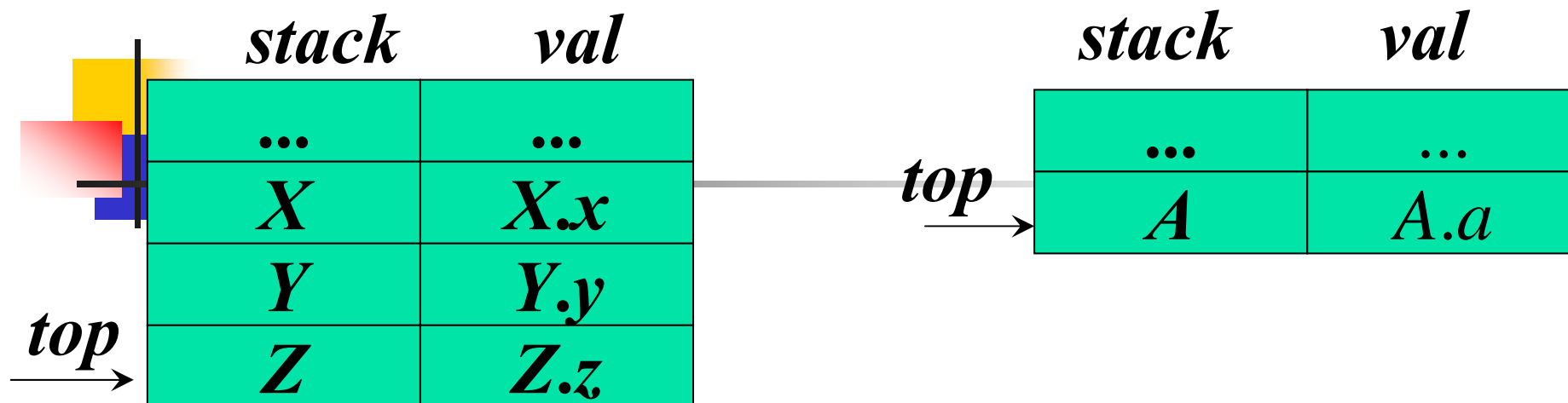
↓


$$\forall A \rightarrow \alpha \quad b := f(c_1, c_2, \dots, c_k)$$

$b$ 是 $A$ 的综合属性， $c_i (1 \leq i \leq k)$ 是 $\alpha$ 中符号的属性。  
综合属性的值是在自底向上的分析过程中，每次归约之前进行计算的。

$$A \rightarrow XYZ \quad A \cdot a := f(X \cdot x, Y \cdot y, Z \cdot z)$$





实现时，将定义式  $A.a := f(X.x, Y.y, Z.z)$  (抽象) 变成  $stack[ntop].val := f(stack[top-2].val, stack[top-1].val, stack[top].val)$  (具体可执行代码)。

在执行代码段之前执行：

$ntop := top - r + 1$  ——  $r$  是句柄的长度

执行代码段后执行：  $top := ntop;$

# 例6.14 用LR分析器实现台式计算器

——与表6.2对比

$L \rightarrow En\{\text{print}(\text{stack}[\text{top}-1].\text{val}); \text{top} := \text{top}-1;\}$

$E \rightarrow E_1 + T\{\text{stack}[\text{top}-2].\text{val} := \text{stack}[\text{top}-2].\text{val} + \text{stack}[\text{top}].\text{val};$   
 $\text{top} := \text{top}-2;\}$

$E \rightarrow T$

$T \rightarrow T_1 * F\{\text{stack}[\text{top}-2].\text{val} := \text{stack}[\text{top}-2].\text{val} \times \text{stack}[\text{top}].\text{val};$   
 $\text{top} := \text{top}-2;\}$


$T \rightarrow F$

$F \rightarrow (E)\{\text{stack}[\text{top}-2].\text{val} := \text{stack}[\text{top}-1].\text{val}; \text{top} := \text{top}-2;\}$

$F \rightarrow \text{digit}$

# 表6.8 翻译输入 $6+7*8n$ 上的移动序列

输入	<i>state</i>	<i>val</i>	使用的产生式
$6+7*8n$	-	-	
$+7*8n$	6	6	
$+7*8n$	F	6	$F \rightarrow \text{digit}$
$+7*8n$	T	6	$T \rightarrow F$
$+7*8n$	E	6	$E \rightarrow T$
$7*8n$	$E+$	$6-$	
$*8n$	$E+7$	$6-7$	



<b>*8n</b>	<b>E+F</b>	<b>6-7</b>	<b>F → digit</b>
<b>*8n</b>	<b>E+T</b>	<b>6-7</b>	<b>T → F</b>
<b>8n</b>	<b>E+T*</b>	<b>6-7-</b>	
<b>n</b>	<b>E+T*8</b>	<b>6-7-8</b>	
<b>n</b>	<b>E+T*F</b>	<b>6-7-8</b>	<b>F → digit</b>
<b>n</b>	<b>E+T</b>	<b>6-56</b>	<b>T → T*F</b>
<b>n</b>	<b>E</b>	<b>62</b>	<b>E → E+T</b>
	<b>En</b>	<b>62</b>	
	<b>L</b>	<b>62</b>	<b>L → En</b>



# S-属性定义小结

---

采用自底向上分析，例如 $LR$ 分析，首先给出 $S$ -属性定义，然后，把 $S$ -属性定义变成可执行的代码段，这就构成了翻译程序。象一座建筑，语法分析是构架，归约处有一个“挂钩”，**语义分析和翻译的代码段（语义子程序）**就挂在这个钩子上。这样，随着语法分析的进行，归约前调用相应的语义子程序，完成翻译的任务。

## 6.4.3 L-属性定义的自顶向下翻译

- 用翻译模式构造自顶向下的翻译。

### 1. 从翻译模式中消除左递归

对于一个翻译模式，若采用自顶向下分析，必须消除左递归和提取左公因子，在改写基础文法时考虑属性值的计算。

对于自顶向下语法分析，假设语义动作是在与之处于同一位置的文法非终结符被展开时执行的，而且**不考虑继承属性的处理（很简单）**。



# 只有简单语义动作时的左递归消除

- 例6.15 考虑如下将中缀表达式翻译为后缀表达式的翻译模式中的两个产生式:

$$E \rightarrow E_1 + T \{ \text{print}(' + '); \}$$

$$E \rightarrow T$$



$$E \rightarrow TR$$

$$R \rightarrow +T \{ \text{print}(' + '); \} R$$

$$R \rightarrow \varepsilon$$



# S-属性定义的左递归消除

- 设有如下左递归翻译模式:

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

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

假设每个非终结符都有一个综合属性，用相应的小写字母表示， $g$ 和 $f$ 是任意函数。

- 消除左递归后，文法转换成

$$A \rightarrow X R$$

$$R \rightarrow Y R \mid \varepsilon$$


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

## S-属性定义的左递归消除

- 再考虑语义动作，翻译模式变为：

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

经过转换的翻译模式使用 $R$ 的继承属性 $i$ 和综合属性 $s$ 。

转换前后的结果是一样的，

为什么？

引入继承属性 $R.i$ 来收集应用函数 $g$ 的计算结果。其初始值为  
 $A.a := f(X.x)$

引入综合属性 $R.s$ 在 $R$ 结束生成 $Y$ 时复制 $R.i$ 的值。



输入:  $XY_1Y_2$

$$A \cdot a = g(g(f(X \cdot x), Y_1 \cdot y), Y_2 \cdot y)$$

$$A \cdot a = g(f(X \cdot x), Y_1 \cdot y)$$

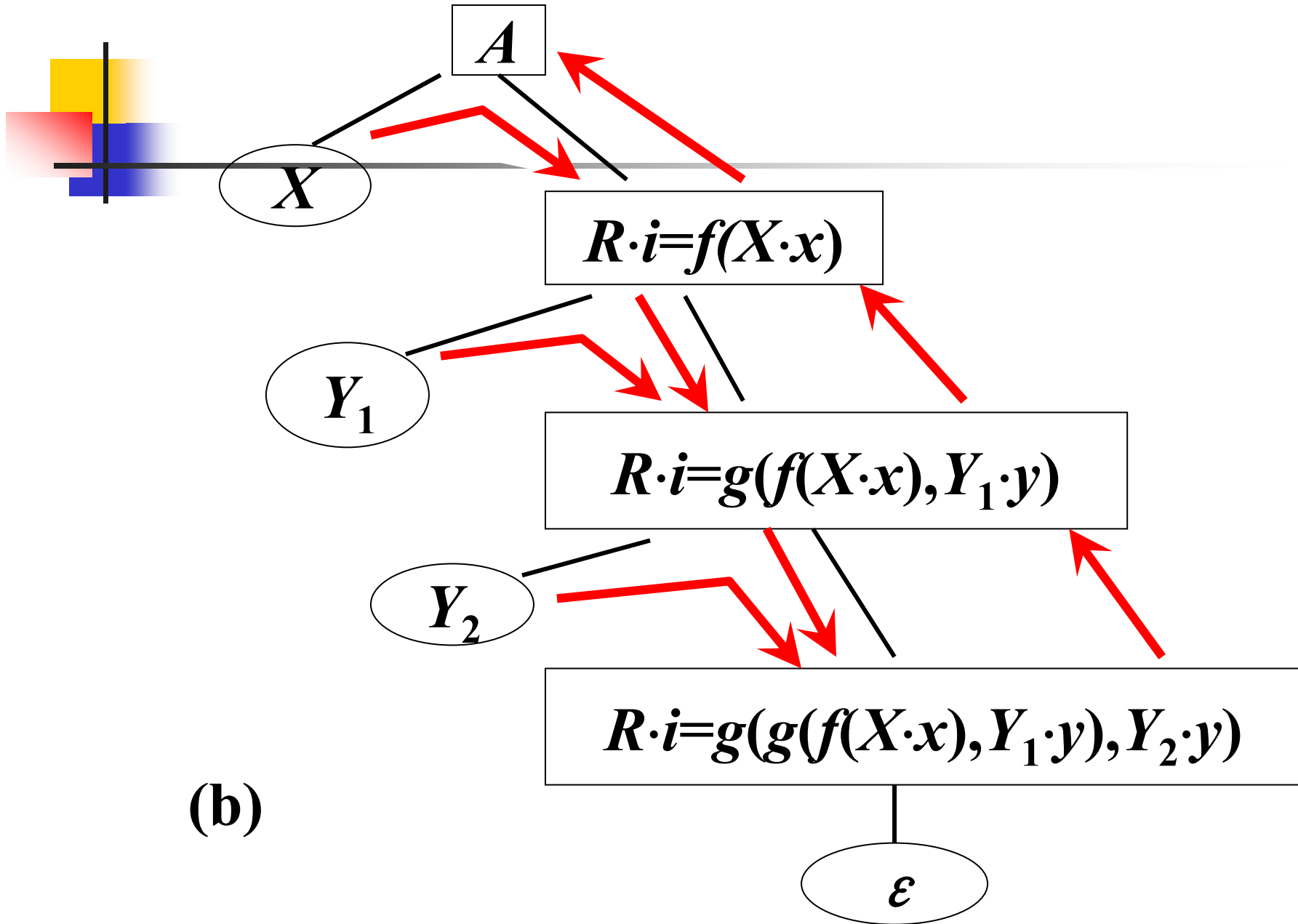
$$A \cdot a = f(X \cdot x)$$

$Y_2$

$Y_1$

$X$

(a)



(b)

## 2. $L$ -属性定义的递归下降翻译法

### $L$ -属性定义的递归下降翻译器的构造:

1. 对每个非终结符 $A$ 构造一个函数 $A$ ，将非终结符 $A$ 的各个继承属性当作函数 $A$ 的形式参数，而将非终结符 $A$ 的综合属性集当作函数 $A$ 的返回值，为了便于讨论，假设每个非终结符只具有一个综合属性。
2. 在函数 $A$ 的过程体中，不仅要进行语法分析，而且要处理相应的语义属性。函数 $A$ 的代码首先根据当前输入确定用哪个产生式展开 $A$ ，然后按照3中所给的方法对各产生式进行编码。

## 2. $L$ -属性定义的递归下降翻译法

3. 与每个产生式对应的程序代码的工作过程为：  
按照从左到右的次序，依次对产生式右部的记号、  
非终结符和语义动作执行如下的动作：

- 1) 对带有综合属性 $x$ 的符号 $X$ ，将 $x$ 的值保存在 $X.x$ 中，并生成一个函数调用来匹配 $X$ ，然后继续读入下一个输入符号；
- 2) 对非终结符 $B$ ，生成一个右部带有函数调用的赋值语句 $c:=B(b1,b2,...,bk)$ ，其中， $b1,b2,...,bk$ 是代表 $B$ 的继承属性的变量， $c$ 是代表 $B$ 的综合属性的变量；
- 3) 对于语义动作，将其代码复制到语法分析器中，并将对属性的引用改为对相应变量的引用。



## 例 6.16

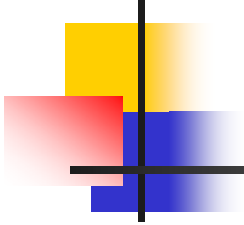
```
function T: ↑ syntax_tree_node;  
function T'(inh: ↑ syntax_tree_node): ↑ syntax_tree_node;  
function F: ↑ syntax_tree_node;  
function T: ↑ syntax_tree_node;  
    var node, syn: ↑ syntax_tree_node;  
    begin  
        node := F;  
        syn := T'(node);  
        return syn  
    end;  
end;
```



```

function  $T'$ (inh:  $\uparrow$  syntax_tree_node):  $\uparrow$  syntax_tree_node;
var node, inh1, syn1:  $\uparrow$  syntax_tree_node; oplexeme: char;
begin if lookahead = '*' then begin
    /* 匹配产生式  $T' \rightarrow *FT'*$  */
    oplexeme := lexval;
    match('*'); node := F;
    inh1 := mknode('*', inh, node);
    syn1 :=  $T'$ (inh1); syn := syn1
end
else if lookahead = '/' then begin
    /* 匹配产生式  $T' \rightarrow /FT'*$  */
    oplexeme := lexval;
    match('/'); node := F;
    inh1 := mknode('/', inh, node);
    syn1 :=  $T'$ (inh1); syn := syn1
end else syn := inh; return syn end;

```



```
function F: ↑ syntax_tree_node;  
var node: ↑ syntax_tree_node;  
begin  if lookahead = '(' then begin  
        /* 匹配产生式  $F \rightarrow (E)$  */  
    match('(');  node := E;  
    match(')')  
end  
else if lookahead = id then begin  
        /* 匹配产生式  $F \rightarrow \text{id}$  */  
    match(id);  node := mkleaf(id, id.entry)  end  
else if lookahead = num then begin  
        /* 匹配产生式  $F \rightarrow \text{num}$  */  
    match(num);  node := mkleaf(num, num.val)  
end  return node  
end;
```



## 3.L-属性定义的 $LL(1)$ 翻译法

---

- 预先在源文法中的相应位置上嵌入语义动作符号(每个对应一个语义子程序), 用于提示语法分析程序, 当分析到达这些位置时应调用相应的语义子程序。
- 带有语义动作符号的文法又叫翻译文法。



## 3.L-属性定义的LL(1)翻译法

### ■ 与递归子程序法的区别与联系

- 都是在扫描过程中进行产生式的推导，同时在适当的地方加入语义动作。
- 递归子程序法将语义动作溶入分析程序；LL(1)分析法则由语义子程序完成相应的翻译。
- 递归子程序法隐式地使用语义栈；LL(1)分析法则用显式的语义栈（程序自身控制对栈的操作）。

**注：语义与语法栈不同步。**



## 例6.17

---

- 对于图6.11的翻译模式，设置两个栈，一个是分析栈，一个是语义栈。

$$(1) T \rightarrow F\{e1\}T'\{e2\}$$

$$(2) T' \rightarrow *F\{e3\}T_1'\{e4\}$$

$$(3) T' \rightarrow /F\{e5\}T_1'\{e4\}$$

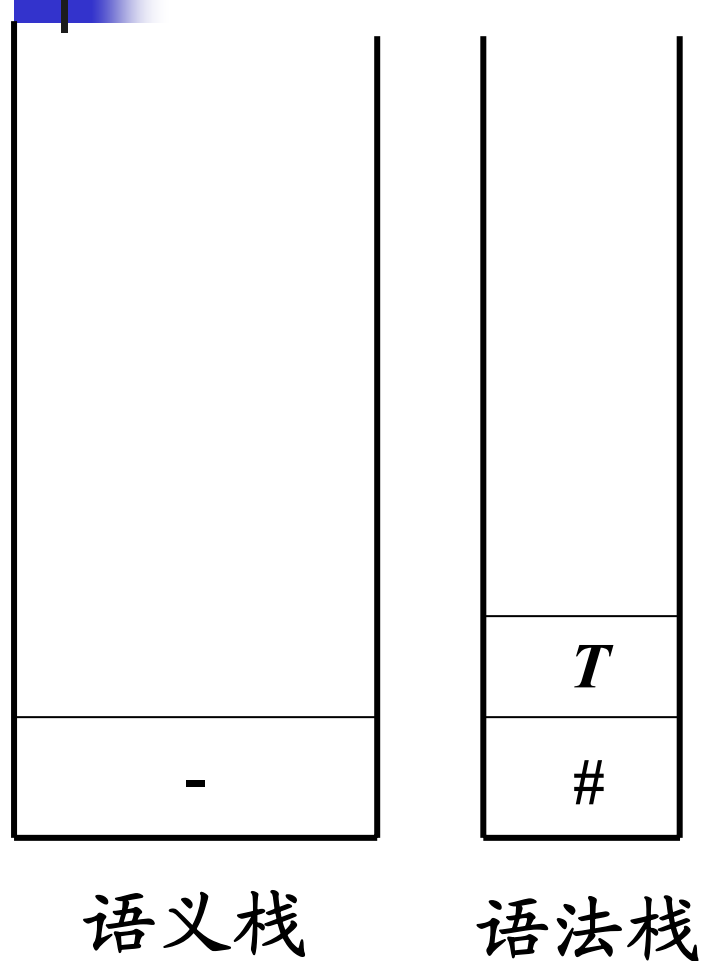
$$(4) T' \rightarrow \varepsilon \{e6\}$$

$$(5) F \rightarrow (E)\{e7\}$$

$$(6) F \rightarrow \text{id}\{e8\}$$

$$(7) F \rightarrow \text{num}\{e9\}$$

## 例6.17 对输入串 $3*x/y\#$ 的翻译



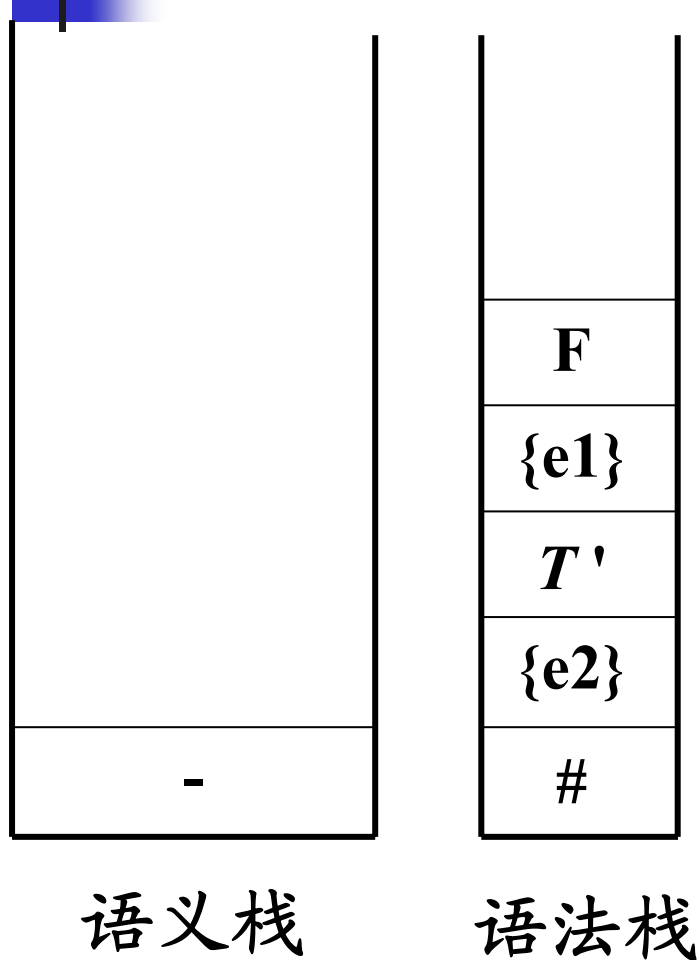
输入串  $3*x/y\#$



语法分析动作和语义操作

### 1. 初始化

## 例6.17 对输入串 $3*x/y\#$ 的翻译



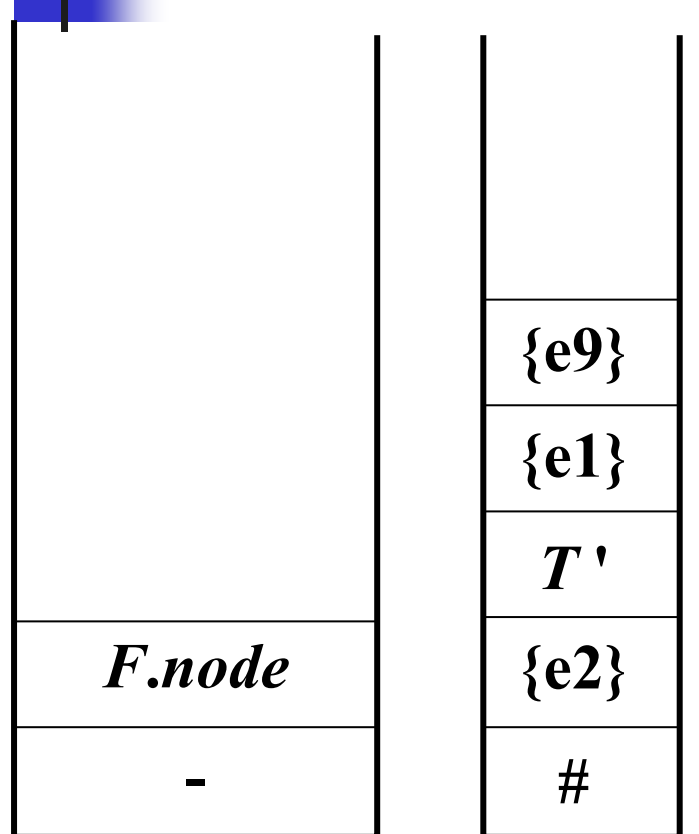
输入串  $3*x/y\#$



语法分析动作和语义操作

2. 选产生式①的右部进栈

## 例6.17 对输入串 $3*x/y\#$ 的翻译



语义栈

语法栈

输入串  $3*x/y\#$

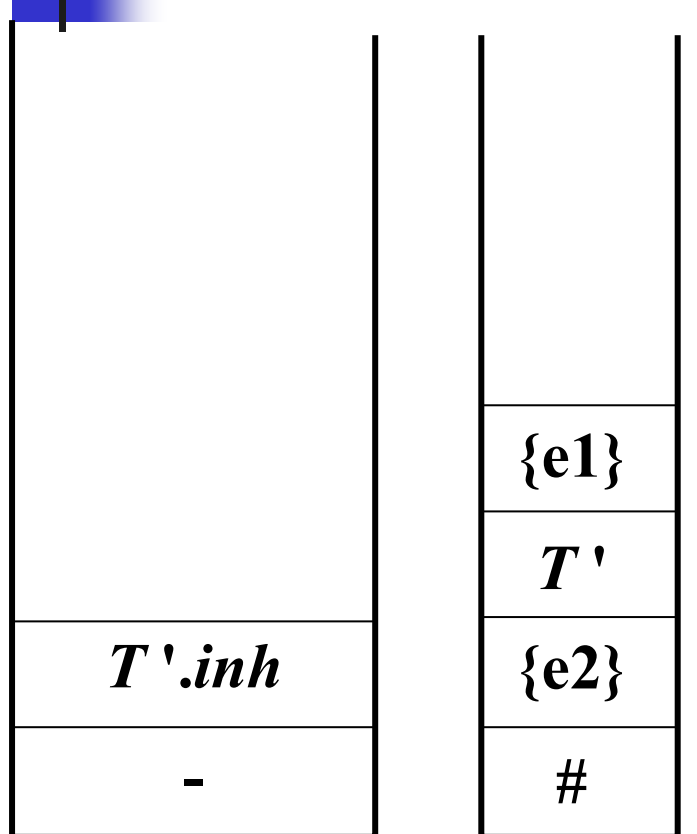


语法分析动作和语义操作

3.选择产生式(7), $num_3$ 不进栈, 调用{e9}, 调用{e9}后, 叶结点 $F.node$ 被压入语义栈



## 例6.17 对输入串 $3*x/y\#$ 的翻译



语义栈

语法栈

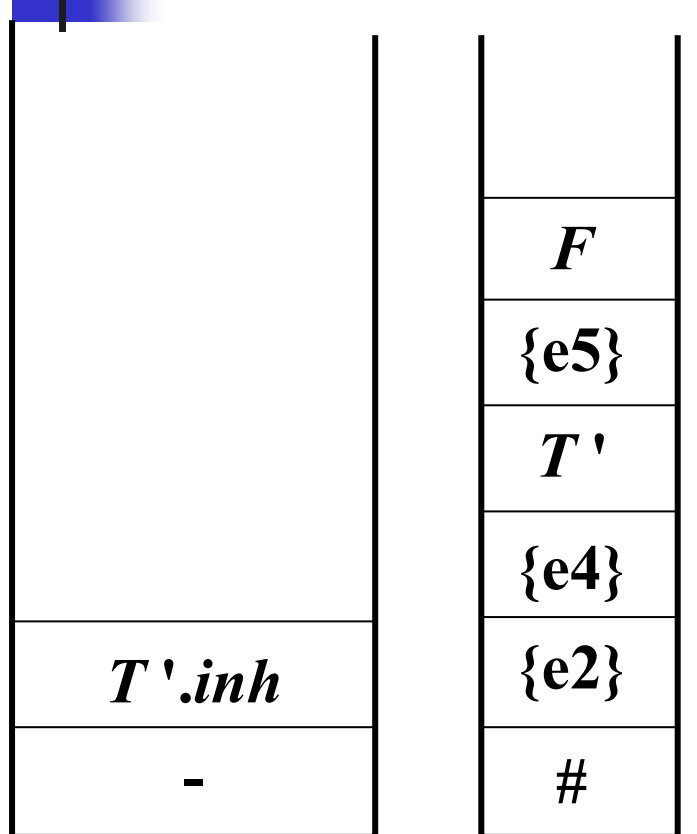
输入串  $3*x/y\#$



语法分析动作和语义操作

4. 执行动作{e1},  $F.node$ 出栈,  
 $T'.inh$ 被压入栈

## 例6.17 对输入串 $3*x/y\#$ 的翻译



语义栈

语法栈

输入串  $3*x/y\#$



语法分析动作和语义操作

5.选择产生式(3),\*不进栈

## 例6.17 对输入串 $3*x/y\#$ 的翻译

	{e8}
	{e5}
	$T'$
$F.node$	{e4}
$T'.inh$	{e2}
-	#

语义栈

语法栈

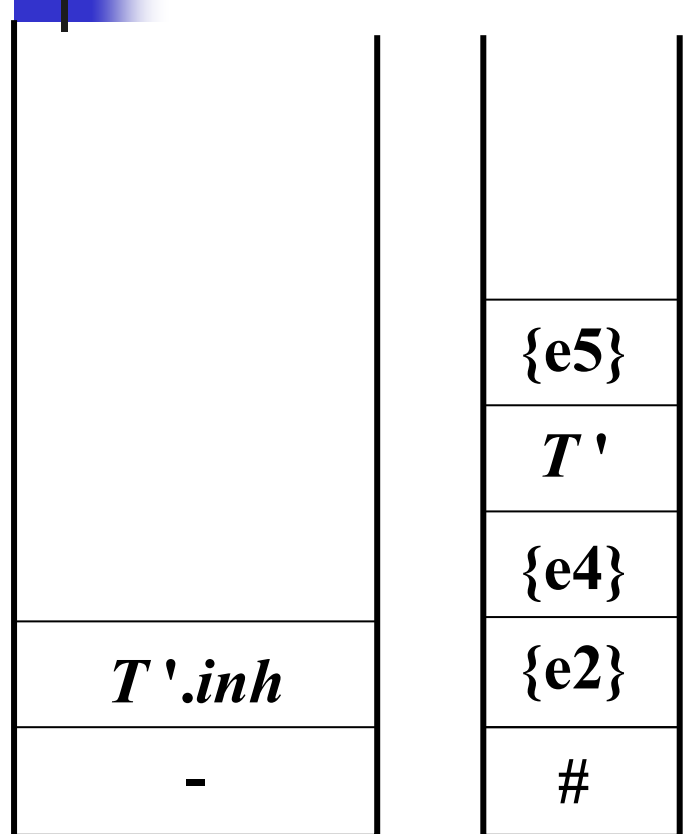
输入串  $3*x/y\#$



语法分析动作和语义操作

6.选择产生式(6), $id_x$ 不进栈, 调用{e8}, 调用{e8}后, 叶结点 $F.node$ 被压入语义栈

## 例6.17 对输入串 $3*x/y\#$ 的翻译



语义栈

语法栈

输入串  $3*x/y\#$



语法分析动作和语义操作

7. 执行动作{e5},  $F.node$ 和 $T'.inh$ 均被弹出栈, 新的 $T'.inh$ 被压入栈

## 例6.17 对输入串 $3*x/y$ 的翻译

	$F$
	$\{e3\}$
	$T'$
	$\{e4\}$
	$\{e4\}$
$T'.inh$	$\{e2\}$
-	#

语义栈

语法栈

输入串  $3*x/y\#$



语法分析动作和语义操作

8.选择产生式(2), /不进栈

## 例6.17 对输入串 $3*x/y\#$ 的翻译

$F.node$
$T'.inh$
-

语义栈

{e8}
{e3}
$T'$
{e4}
{e4}
{e2}
#

语法栈

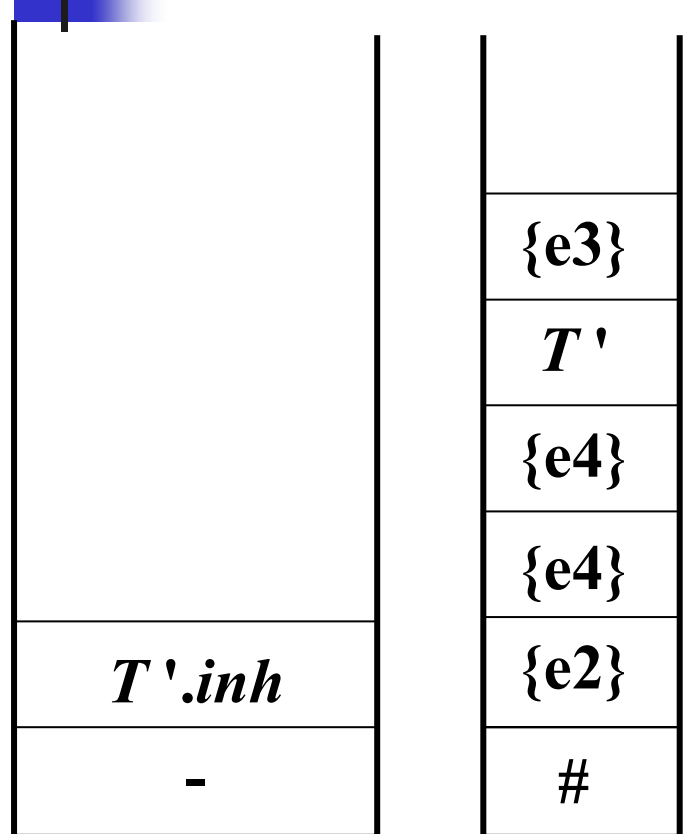
输入串  $3*x/y\#$



语法分析动作和语义操作

9.选择产生式(6), $id_y$ 不进栈, 调用 {e8}, 调用 {e8}后, 叶结点 $F.node$ 被压入语义栈

## 例6.17 对输入串 $3*x/y\#$ 的翻译



语义栈

语法栈

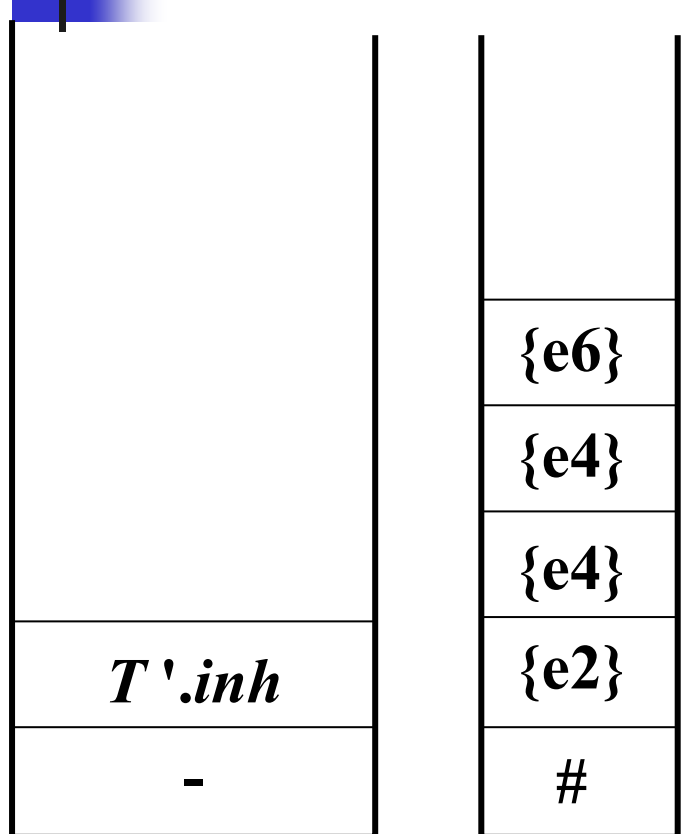
输入串  $3*x/y\#$



语法分析动作和语义操作

**10.**执行动作{e3},  $F.node$ 和 $T'.inh$ 均被弹出栈, 新的 $T'.inh$ 被压入栈

## 例6.17 对输入串 $3*x/y\#$ 的翻译



语义栈

语法栈

输入串  $3*x/y\#$

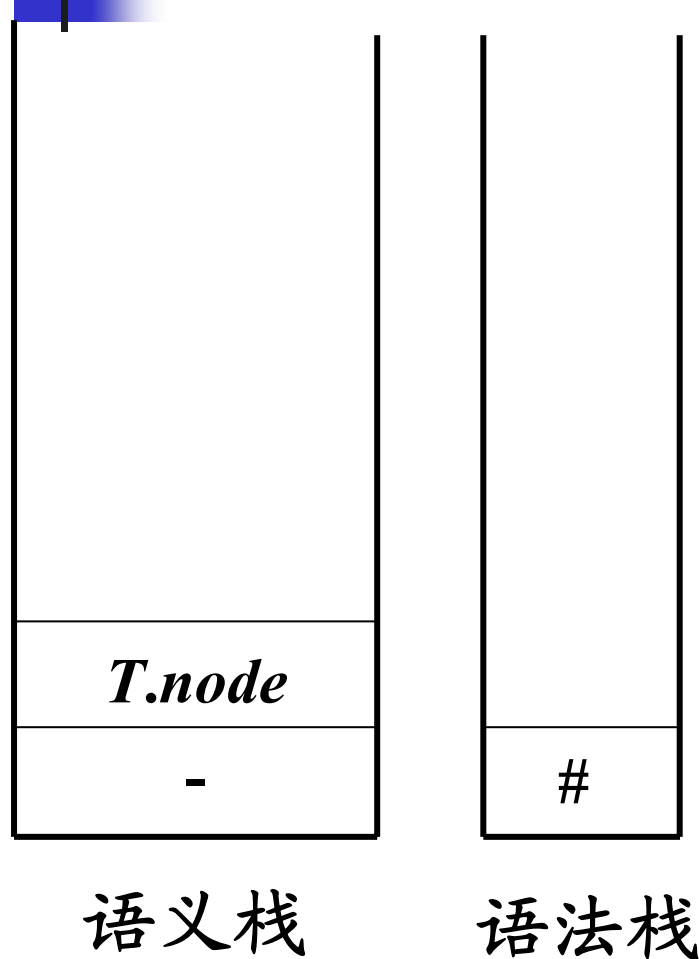


语法分析动作和语义操作

11. 选择产生式(4),  $T'.inh$ 被弹出栈,  $T'.syn$ 被压入栈



## 例6.17 对输入串 $3*x/y\#$ 的翻译



输入串  $3*x/y\#$



语法分析动作和语义操作

12.依次执行动作

$\{e6\}, \{e4\}, \{e4\}, \{e2\}$ , 最终语义栈中只有  $T.node$ , 代表  $3*x/y$  的语法树的根结点

## 6.4.4 L-属性定义的自底向上翻译

---

在自底向上分析的框架中实现 $L$ 属性定义的方法

- 它能实现任何基于 $LL(1)$ 文法的 $L$ 属性定义。
- 也能实现许多（但不是所有的）基于 $LR(1)$ 文法的 $L$ 属性定义。

## 6.4.4 L-属性定义的自底向上翻译(续)

- (1) 首先像6.4.1所介绍的那样构造翻译模式，它将计算继承属性的动作嵌入在非终结符的前面，而将计算综合属性的动作放在产生式的末尾。
- (2) 在每个嵌入动作处引入一个标记性非终结符(marker nonterminals)。不同位置所对应的标记是不同的，每个标记性非终结符 $M$ 都有一个形如 $M \rightarrow \varepsilon$ 的产生式。

## 6.4.4 L-属性定义的自底向上翻译(续)

(3) 如果标记性非终结符 $M$ 取代了某个产生式 $A \rightarrow \alpha \{a\} \beta$ 中的动作 $a$ , 则按如下方式将 $a$ 修改为 $a'$ , 并将动作 $\{a'\}$ 放在产生式 $M \rightarrow \varepsilon$ 的末尾。

① 为 $M$ 设置继承属性来复制动作 $a$ 所需要的 $A$ 或 $\alpha$ 中符号的继承属性;

② 以与动作 $a$ 相同的方式计算属性, 只不过要将这些属性置为 $M$ 的综合属性。

## 6.4.4 L-属性定义的自底向上翻译(续)

- 与 $M \rightarrow \varepsilon$ 相关联的语义动作可能需要用到没出现在该产生式中的文法符号的属性。不过，由于要在 $LR$ 分析栈中实现所有的语义动作，所以在分析栈中 $M$ 下面的某个已知位置总能找到所需的属性。
- 例如，假设在某个 $LL(1)$ 文法中有一个形如 $A \rightarrow BC$ 的产生式， $B$ 的继承属性 $B.inh$ 是从 $A$ 的继承属性 $A.inh$ 按照公式 $B.inh := f(A.inh)$ 来计算的，亦即翻译模式可能包含如下片断：

$$A \rightarrow \{B.inh := f(A.inh);\}BC$$

## 6.4.4 L-属性定义的自底向上翻译(续)

- 根据上面的论述，为了在自底向上的分析过程中计算  $B.inh := f(A.inh)$ ，需要引入一个标记性非终结符  $M$ ，并为其设置一个继承属性  $M.inh$  用来复制  $A$  的继承属性，而且还要用  $M$  的综合属性  $M.syn$  代替  $B.inh$ ，于是，该翻译模式片段将被修改为如下形式：

$$A \rightarrow MBC$$

$$M \rightarrow \varepsilon \{M.inh := A.inh; M.syn := f(M.inh)\}$$

## 6.4.4 L-属性定义的自底向上翻译(续)

- 注意，执行 $M$ 的语义规则时， $A.inh$ 是不可用的，但实际上，实现时会把每个非终结符 $X$ 的继承属性都放在堆栈中紧靠在 $X$ 将被归约出来的位置之下。于是，当将 $\varepsilon$ 归约为 $M$ 时， $A.inh$ 恰好就在它的下面。随 $M$ 保存在栈中的 $M.syn$ 的值，也就是 $B.inh$ 的值，亦将被放在紧靠在 $B$ 将被归约出来的位置之下，需要的时候同样是可用的。

## 6.4.4 L-属性定义的自底向上翻译(续)

- 给定一个L-属性定义，假设它的每个非终结符 $A$ 都有一个继承属性 $A.inh$ ，而且每个文法符号 $X$ 都有一个综合属性 $X.syn$ 。
- 如果 $X$ 是终结符，则其综合属性值就是词法分析器所返回的词法值。
- 假设分析栈仍是 $stack$ ，语义栈仍为 $val$ 。如果 $stack[i]$ 代表文法符号 $X$ ，则 $stack[i].val$ 将保存其综合属性 $X.syn$ 。
- 为了在自底向上的分析过程计算继承属性，需要对每个产生式 $A \rightarrow X_1 \dots X_n$ 引入 $n$ 个标记性非终结符 $M_1, \dots, M_n$ ，并用 $A \rightarrow M_1 X_1 \dots M_n X_n$ 代替该产生式。



## 6.4.4 L-属性定义的自底向上翻译(续)

- 归约标记性非终结符 $M_j$ 时，它出现在产生式 $A \rightarrow M_1 X_1 \dots M_n X_n$ 中，从而可以确定为计算继承属性 $X_j.inh$ 所需要的那些属性的位置： $A.inh$ 在 $stack[top-2j+2].val$ 中， $X_1.inh$ 在 $stack[top-2j+3].val$ 中， $X_1.syn$ 在 $stack[top-2j+4].val$ 中， $X_2.inh$ 在 $stack[top-2j+5].val$ 中，依次类推。于是，就可以计算出 $X_j.inh$ ，并将其存放在 $stack[top+1].val$ 中，归约后它成为新的栈顶。

## 6.4.4 L-属性定义的自底向上翻译(续)

- 归约非标记符号
- 譬如用产生式 $A \rightarrow M_1X_1 \dots M_nX_n$ 进行归约。在这种情况下，只需要计算综合属性 $A.syn$ ，因为 $A.inh$ 已经被计算出来了，而且放在栈中 $A$ 将要插入的位置之下。
- 很明显，进行归约时，计算 $A.syn$ 所需要的其它属性也都在栈中的已知位置，即 $X_j (1 \leq j \leq n)$ 所在的位置。

## 6.4.4 L-属性定义的自底向上翻译(续)

- 下面的化简可以减少标记性非终结符的个数，其中第2条还可以避免左递归文法中的分析冲突：

(1) 如果 $X_j$ 没有继承属性，则不需要使用标记 $M_j$ 。当然，如果省略了 $M_j$ ，属性在栈中的预期位置就会改变，但是分析器可以很容易地适应这种变化。

(2) 如果 $X_1.inh$ 存在，但它是由复制规则 $X_1.inh := A.inh$ 计算的，此时可以省略 $M_1$ 。因为 $A.inh$ 存放在栈中紧挨在 $X_1$ 下面的地方，所以该值也可同时作为 $X_1.inh$ 的值。

## 6.4.4 L-属性定义的自底向上翻译(续)

例6.18 数学排版语言EQN

$S \rightarrow \{B.ps := 10\}$   
 $B \quad \{S.ht := B.ht; S.dp := B.dp\}$

$B \rightarrow \{B_1.ps := B.ps\}$   
 $B_1 \quad \{B_2.ps := B.ps\}$   
 $B_2 \quad \{B.ht := \max(B_1.ht, B_2.ht)\}$

$B \rightarrow \{B_1.ps := B.ps\}$   
 $B_1$   
sub  $\{B_2.ps := 0.7 \times B.ps\}$   
 $B_2 \quad \{B.ht := \text{disp}(B_1.ht, B_2.ht)$   
 $\quad B.dp := \max(B_1.dp, B_2.dp)\}$

$B \rightarrow \text{text}\{B.ht := \text{getheight}(B.ps, \text{text.lexval});$   
 $\quad B.dp := \text{getdepth}(B.ps, \text{text.lexval})\}$

保证在 $B$ 的子树被归约时,  $B.ps$ 的值出现在分析栈中的已知位置

归约 $B_1$ 之前,  $B.ps$ 可以在栈中找到, 所以 $B_1.ps := B.ps$ 可以省略。但归约 $B_2$ 之前, 无法确定其前有几个 $B_1$ , 因此, 无法预测 $B.ps$ 在栈中的位置。

## 6.4.4 L-属性定义的自底向上翻译(续)

- 由于存在一个继承属性和两个综合属性，所以语义栈 $val$ 需要被扩展为 $val_1$ 、 $val_2$ 和 $val_3$ 
  - $val_1$ 用于保存继承属性 $ps$ 的值
  - $val_2$ 和 $val_3$ 分别用于保存综合属性 $ht$ 和 $dp$ 的值
  - 假设分析栈仍为 $stack$
  - $top$ 和 $ntop$ 分别是归约前和归约后栈顶的下标。

## 6.4.4 L-属性定义的自底向上翻译(续)

产生式	语义规则
$S \rightarrow LB$	$B.ps := L.syn; S.ht := B.ht; S.dp := B.dp$
$L \rightarrow \varepsilon$	$L.syn := 10$
$B \rightarrow B_1 MB_2$	$B_1.ps := B.ps; M.inh := B.ps;$ $B_2.ps := M.syn; B.ht := \max(B_1.ht, B_2.ht)$
$M \rightarrow \varepsilon$	$M.syn := M.inh$
$B \rightarrow B_1 \text{ sub } NB_2$	$B_1.ps := B.ps; N.inh := B.ps;$ $B_2.ps := N.syn; B.ht := \text{disp}(B_1.ht, B_2.ht);$ $B.dp := \max(B_1.dp, B_2.dp)$
$N \rightarrow \varepsilon$	$N.syn := 0.7 \times N.inh$
$B \rightarrow \text{text}$	$B.ht := \text{getheight}(B.ps, \text{text.lexval});$ $B.dp := \text{getdepth}(B.ps, \text{text.lexval})$

## 6.4.4 L-属性定义的自底向上翻译(续)

产生式	代码段
$S \rightarrow LB$	$stack[ntop].val_2 := stack[top].val_2$ $stack[ntop].val_3 := stack[top].val_3$
$L \rightarrow \varepsilon$	$stack[ntop].val_1 := 10$
$B \rightarrow B_1 MB_2$	$stack[ntop].val_2 := \max(stack[top-2].val_2, stack[top].val_2)$ $stack[ntop].val_3 := \max(stack[top-2].val_3, stack[top].val_3)$
$M \rightarrow \varepsilon$	$stack[ntop].val_1 := stack[top-1].val_1$
$B \rightarrow B_1 \text{ sub } NB_2$	$stack[ntop].val_2 :=$ $\max(stack[top-3].val_2, stack[top].val_2 - 0.25 \times stack[top-4].val_1)$ $stack[ntop].val_3 :=$ $\max(stack[top-3].val_3, stack[top].val_3 + 0.25 \times stack[top-4].val_1)$
$N \rightarrow \varepsilon$	$stack[ntop].val_1 := 0.7 \times stack[top-2].val_1$
$B \rightarrow \text{text}$	$stack[ntop].val_2 := \text{getheight}(stack[top-1].val_1, \text{text.lexval})$ $stack[ntop].val_3 := \text{getdepth}(stack[top-1].val_1, \text{text.lexval})$



# 本章小结

---

- 语法分析中进行静态语义检查和中间代码生成的技术称为语法制导翻译技术。
- 为了通过将语义属性关联到文法符号、将语义规则关联到产生式，有效地将语法和语义关联起来，人们引入了语法制导定义。没有副作用的语法制导定义又称为属性文法。





# 本章小结

- 为相应的语法成分设置表示语义的属性，属性的值是可以计算的，根据属性值计算的关联关系，将其分成综合属性和继承属性，根据属性文法中所含的属性将属性文法分成S-属性文法和L-属性文法。
- 如果不仅将语义属性关联到文法符号、将语义规则关联到产生式，而且还通过将语义动作嵌入到产生式的适当位置来表达该语义动作的执行时机，这就是翻译模式。翻译模式给语义分析的实现提供了更好的支持。



# 本章小结

---

- 注释分析树和相应的依赖图是属性值的关联关系和计算顺序的表达形式，语义关系可以使用抽象语法树表示。
- 依据语法分析方法有自底向上的和自顶向下的，语法制导翻译既可以按照自底向上的策略进行，也可以按照自顶向下的策略进行。



*School of Computer Science & Technology*  
*Harbin Institute of Technology*



# 第7章 语义分析与中间代码生成

**重点:** 三地址码, 各种语句的目标代码结构、语法制导定义与翻译模式。

**难点:** 布尔表达式的翻译, 对各种语句的目标代码结构、语法制导定义与翻译模式的理解。





# 第7章 语义分析与中间代码生成

---

- 7.1 中间代码的形式
- 7.2 声明语句的翻译
- 7.3 赋值语句的翻译
- 7.4 类型检查
- 7.5 控制结构的翻译
- 7.6 回填
- 7.7 **switch**语句的翻译
- 7.8 过程调用和返回语句的翻译
- 7.9 输入输出语句的翻译
- 7.10 本章小结



# 7.1 中间代码的形式

---

- 中间代码的作用
  - 过渡：经过语义分析被译成中间代码序列
- 中间代码的形式
  - 中间语言的语句
- 中间代码的优点
  - 形式简单、语义明确、独立于目标语言
  - 便于编译系统的实现、移植、代码优化
- 常用的中间代码
  - 语法树(6.3.5节)
  - 逆波兰表示、三地址码(三元式和四元式)、DAG图表示



## 7.1.1 逆波兰表示

---

- 中缀表达式的计算顺序不是运算符出现的自然顺序，而是根据运算符间的优先关系来确定的，因此，从中缀表达式直接生成目标代码一般比较麻烦。
- 波兰逻辑学家J. Lukasiewicz于1929年提出了后缀表示法，其优点为：表达式的运算顺序就是运算符出现的顺序，它不需要使用括号来指示运算顺序。



## 7.1.1 逆波兰表示

---

- 例7.1 下面给出的是一些表达式的中缀、前缀和后缀表示。

中缀表示

$a+b$

$a*(b+c)$

$(a+b)*(c+d)$

$a:=a*b+c*d$

前缀表示

$+ab$

$*a+bc$

$*+ab+cd$

$:=a+*ab*cd$

后缀表示

$ab+$

$abc+*$

$ab+cd+*$

$abc*bd*+: =$



## 7.1.2 三地址码

---

- 所谓三地址码，是指这种代码的每条指令最多只能包含三个地址，即两个操作数地址和一个结果地址。
- 如 $x+y*z$ 三地址码为： $t_1 := y*z$      $t_2 := x+t_1$
- 三地址码中地址的形式：
  - 名字、常量、编译器生成的临时变量。





## 7.1.2 三地址码

- 例7.2 赋值语句  $a := (-b) * (c + d) - (c + d)$  的三地址码如图7.1所示

```
t1 := minus b  
t2 := c + d  
t3 := t1 * t2  
t4 := c + d  
t5 := t3 - t4  
a := t5
```

图7.1  $a := (-b) * (c + d) - (c + d)$  的三地址码



## 7.1.2 三地址码

---

1. 形如 $x := y \text{ op } z$ 的赋值指令;
2. 形如 $x := \text{op } y$ 的赋值指令;
3. 形如  $x := y$ 的复制指令;
4. 无条件跳转指令  $\text{goto } L$ ;
5. 形如  $\text{if } x \text{ goto } L$  (或  $\text{if false } x \text{ goto } L$ ) 的条件跳转指令;
6. 形如  $\text{if } x \text{ relop } y \text{ goto } L$  的条件跳转指令;
7. 过程调用和返回使用如下的指令来实现:
  - $\text{param } x$  用来指明参数;
  - $\text{call } p, n$  和  $y = \text{call } p, n$  用来表示过程调用和函数调用;
  - $\text{return } y$  表示过程返回;
8. 形如  $x := y[i]$  和  $x[i] := y$  的变址复制指令;
9. 形如  $x := \&y$ 、 $x := *y$  和  $*x := y$  的地址和指针赋值指令。

# 四元式

- 四元式是一种比较常用的中间代码形式，它由四个域组成，分别称为op、arg1、arg2和result。op是一个一元或二元运算符，arg1和arg2分别是op的两个运算对象，它们可以是变量、常量或编译器生成的临时变量，运算结果则放入result中。

	op	arg <sub>1</sub>	arg <sub>2</sub>	result
0	minus	b		t <sub>1</sub>
1	+	c	d	t <sub>2</sub>
2	*	t <sub>1</sub>	t <sub>2</sub>	t <sub>3</sub>
3	+	c	d	t <sub>4</sub>
4	-	t <sub>3</sub>	t <sub>4</sub>	t <sub>5</sub>
5	assign	t <sub>5</sub>		a
	...			

# 三元式

- 为了节省临时变量的开销，有时也可以使用只有三个域的三元式来表示三地址码。三元式的三个域分别称为op，arg1和arg2，op，arg1和arg2的含义与四元式类似，区别只是arg1和arg2可以是某个三元式的编号(图7.2(b)中用圆括号括起来的数字)，表示用该三元式的运算结果作为运算对象。

	op	arg <sub>1</sub>	arg <sub>2</sub>
0	minus	b	
1	+	c	d
2	*	(0)	(1)
3	+	c	d
4	-	(2)	(3)
5	assign	a	(4)
	...		

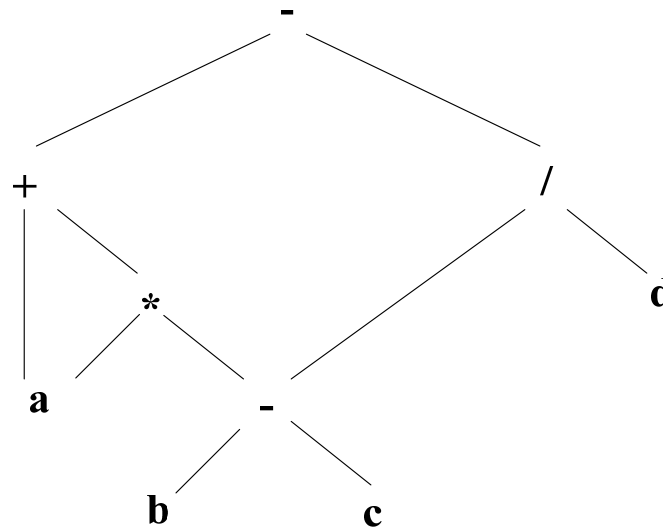
# 生成三地址码的语法制导定义

产生式	语义规则
$S \rightarrow id := E$	$S.code := E.code    gencode(id.addr' := 'E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr := newtemp; E.code := E_1.code    E_2.code    gencode(E.addr' := 'E_1.addr' + ' E_2.addr)$
$E \rightarrow E_1 * E_2$	$E.addr := newtemp; E.code := E_1.code    E_2.code    gencode(E.addr' := 'E_1.addr' * ' E_2.addr)$
$E \rightarrow -E_1$	$E.addr := newtemp; E.code := E_1.code    gencode(E.addr' := ''uminus' E_1.addr)$
$E \rightarrow (E_1)$	$E.addr := E_1.addr; E.code := E_1.code$
$E \rightarrow id$	$E.addr := id.addr; E.code := ''$
$E \rightarrow num$	$E.addr := num.val; E.code := ''$

属性 code 表示生成的代码

## 7.1.3 图表示

- 类似于表达式的抽象语法树一样，在dag(directed acyclic graph)中，每个节点对应一个运算符，代表表达式的一个子表达式，其子节点则与该运算符的运算对象相对应，叶节点对应的是变量或者常量，可以看成是原子运算。
- 利用dag可以很容易地消除公共子表达式
- 例7.3 表达式 $a+a*(b-c)-(b-c)/d$ 的dag如图7.5所示。





# 生成dag的语法制导定义

产生式	语义规则
(1) $E \rightarrow E_1 + T$	$E.node := mknode('+', E_1.node, T.node)$
(2) $E \rightarrow E_1 - T$	$E.node := mknode('-', E_1.node, T.node)$
(3) $E \rightarrow T$	$E.node := T.node$
(4) $T \rightarrow T_1 * F$	$T.node := mknode('*', T_1.node, F.node)$
(5) $T \rightarrow T_1 / F$	$T.node := mknode('/', T_1.node, F.node)$
(6) $T \rightarrow F$	$T.node := F.node$
(7) $F \rightarrow (E)$	$F.node := E.node$
(8) $F \rightarrow id$	$F.node := mkleaf(id, id.entry)$
(9) $F \rightarrow num$	$F.node := mkleaf(num, num.val)$



## 7.2 声明语句的翻译

---

- 声明语句的作用

- 为程序中用到的变量或常量名指定类型

- 类型的作用

- 类型检查：类型检查的任务是验证程序运行时的行为是否遵守语言的类型的规定，也就是是否符合该语言关于类型的相关规则。
- 辅助翻译：编译器从名字的类型可以确定该名字在运行时所需要的存储空间。在计算数组引用的地址、加入显式的类型转换、选择正确版本的算术运算符以及其它一些翻译工作时同样需要用到类型信息。

- 编译的任务

- 在符号表中记录被说明对象的属性(种别、类型、相对地址、作用域……等)，为执行做准备





## 7.2.1 类型表达式

- 类型可以具有一定的层次结构，因此用类型表达式来表示。类型表达式的定义如下：
  1. 基本类型是类型表达式。

典型的基本类型包括`boolean`、`char`、`integer`、`real`及`void`等。
  2. 类型名是类型表达式。
  3. 将类型构造符`array`应用于数字和类型表达式所形成的表达式是类型表达式。

如果 $T$ 是类型表达式，那么`array( $I$ ,  $T$ )`就是元素类型为 $T$ 、下标集为 $I$ 的数组类型表达式。
  4. 如果 $T1$ 和 $T2$ 是类型表达式，则其笛卡尔乘积 $T1 \times T2$ 也是类型表达式。



## 7.2.1 类型表达式

5. 类型构造符`record`作用于由域名和域类型所形成的表达式也是类型表达式。记录`record`是一种带有命名域的数据结构，可以用来构成类型表达式。例如，下面是一段Pascal程序段：

- `type row = record`
- `address: integer;`
- `lexeme: array[1..15] of char`
- `end;`
- `var table : array [1..10] of row;`
- 该程序段声明了表示下列类型表达式的类型名`row`:
  - `record ((address × integer) × (lexeme × array (1..15, char)))`



## 7.2.1 类型表达式

6. 如果 $T$ 是类型表达式, 那么 $pointer(T)$ 也是类型表达式, 表示“指向类型为 $T$ 的对象的指针”。
  - 函数的类型可以用类型表达式 $D \rightarrow R$ 来表示。考虑如下的Pascal声明:
    - `function  $f(a,b: char): \uparrow integer$ ;`
  - 其定义域类型为 $char \times char$ , 值域类型为 $pointer(integer)$ 。所以函数 $f$ 的类型可以表示为如下的类型表达式:
    - $char \times char \rightarrow pointer(integer)$
7. 类型表达式可以包含其值为类型表达式的变量。



## 7.2.2 类型等价

- 许多 **类型检查的规则** 都具有如下的形式：
  - if两个类型表达式等价then返回一种特定类型else返回 *type\_error*.
- 如果用图来表示类型表达式，当且仅当下列条件之一成立时，称两个类型  $T_1$  和  $T_2$  是 **结构等价** 的：
  - $T_1$  和  $T_2$  是相同的基本类型；
  - $T_1$  和  $T_2$  是将同一类型构造符应用于结构等价的类型上形成的；
  - $T_1$  是表示  $T_2$  的类型名。
- 如果将类型名看作只代表它们自己的话，则上述条件中的前两个将导致类型表达式的 **名字等价**
  - 两个类型表达式名字等价当且仅当它们完全相同



## 7.2.3 声明语句的文法

---

- $P \rightarrow \text{prog id (input, output) } D ; S$
- $D \rightarrow D ; D \mid List : T \mid \text{proc id } D ; S$
- $List \rightarrow List1, id \mid id$
- $T \rightarrow \text{integer} \mid \text{real} \mid \text{array } C \text{ of } T_1 \mid \uparrow T_1 \mid \text{record } D$
- $C \rightarrow [\text{num}] C \mid \varepsilon$
- $D$  ——程序说明部分的抽象
- $S$  ——程序体部分的抽象
- $T$  ——类型的抽象，需要表示成类型表达式
- $C$  ——数组下标的抽象

# 语义属性、辅助过程与全局变量的设置

- 文法变量T(类型)的语义属性
  - type: 类型(表达式)
  - width: 类型所占用的字节数
- 辅助子程序
  - enter: 将变量的类型和地址填入符号表中
  - array: 数组类型处理子程序
- 全局变量
  - offset: 已分配空间字节数, 用于计算相对地址

## 7.2.4 过程内声明语句的翻译

$P \rightarrow \{ \text{offset} := 0 \} D$

$P \rightarrow MD$

$M \rightarrow \varepsilon \quad \{ \text{offset} := 0 \}$

$D \rightarrow D ; D$

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

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

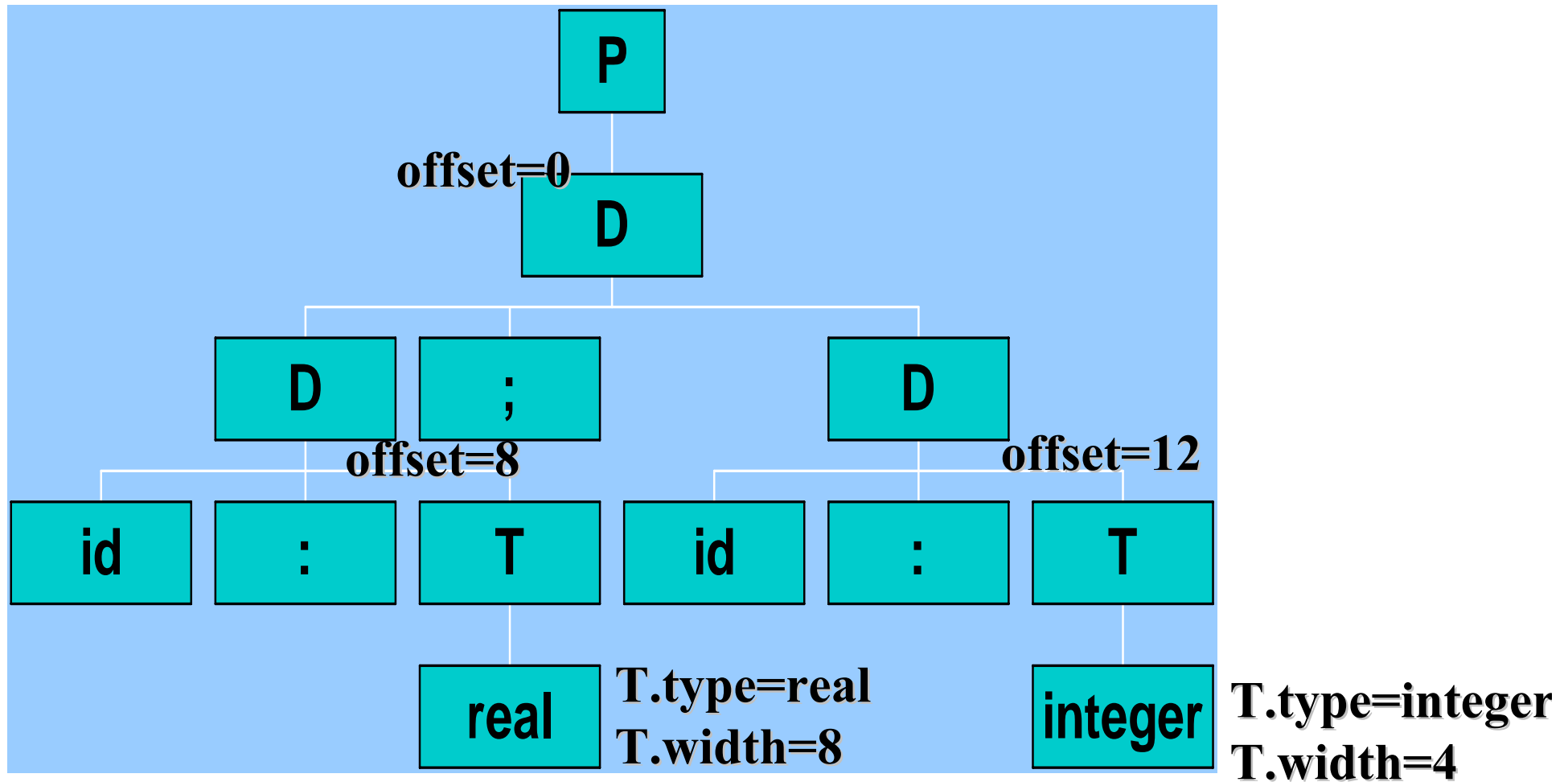
$T \rightarrow \text{real} \quad \{ T.\text{type} := \text{real}; \quad T.\text{width} := 8 \}$

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

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

# 例 x:real; i:integer 的翻译

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



`enter(x,real,0)`

2012-4-26

`enter(i,integer,8)`

496



# 例 x:real; i:integer 的翻译



$P \Rightarrow \{\text{offset}:=0\}D$

$\Rightarrow \{\text{offset}:=0\}D;D$

$\Rightarrow \{\text{offset}:=0\}x:T\{\text{enter}(x,T.\text{type},\text{offset});$   
 $\quad \text{offset}:=\text{offset}+T.\text{width}\};D$

$\Rightarrow \{\text{offset}:=0\}x:\text{real}\{T.\text{type}:=\text{real};T.\text{width}:=8\}$   
 $\quad \{\text{enter}(x,T.\text{type},\text{offset});\text{offset}:=\text{offset}+T.\text{width}\};D$

$\Rightarrow x:\text{real}\{(x,\text{real},0);\text{offset}:=8\};D$

$\Rightarrow x:\text{real}\{(x,\text{real},0);\text{offset}:=8\};i:T$   
 $\quad \{\text{enter}(i.\text{name},T.\text{type},\text{offset}); \text{offset}:=\text{offset}+T.\text{width}\}$

$\Rightarrow x:\text{real}\{(x,\text{real},0);\text{offset}:=8\};i:\text{integer}\{T.\text{type}:=\text{integer};$   
 $T.\text{width}:=4\}\{\text{enter}(i,T.\text{type},\text{offset});\text{offset}:=\text{offset}+T.\text{width}\}$

$\Rightarrow x:\text{real}\{(x,\text{real},0)\};i:\text{integer}\{(i,\text{integer},8);\text{offset}:=12\}$



## 7.2.5 嵌套过程中声明语句的翻译

---

- 所讨论语言的文法

$P \rightarrow \text{prog id (input, output) D ; S}$

$D \rightarrow D ; D \mid \text{id : T} \mid \text{proc id D ; S}$

- 语义动作作用到的函数

- **mktable(previous):** 创建一个新的符号表;

- **enter(table, name, type, offset)**

- **addwidth(table, width):** 符号表的大小;

- **enterproc(table, name, newtable)**

在table指向的符号表中为name建立一个新表项;

## 7.2.5 嵌套过程中声明语句的翻译

$P \rightarrow \text{prog id (input,output) } \mathbf{M} \text{ D; S \{addwidth (top(tblptr), top(offset)); pop(tblptr); pop(offset) \}}$

$M \rightarrow \varepsilon \quad \{t := \text{mktable(nil);}$   
 $\quad \text{push(t,tblptr); push(0,offset) \}$

$D \rightarrow D_1 ; D_2$

$D \rightarrow \text{proc id ; } \mathbf{N} \text{ D}_1 ; \text{S \{t := top(tblptr);}$   
 $\quad \text{addwidth(t,top(offset));pop(tblptr);pop(offset);}$   
 $\quad \text{enterproc(top(tblptr),id.name,t) \}$

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

$N \rightarrow \varepsilon \quad \{t := \text{mktable(top(tblptr) );}$   
 $\quad \text{push(t,tblptr); push(0,offset) \}$

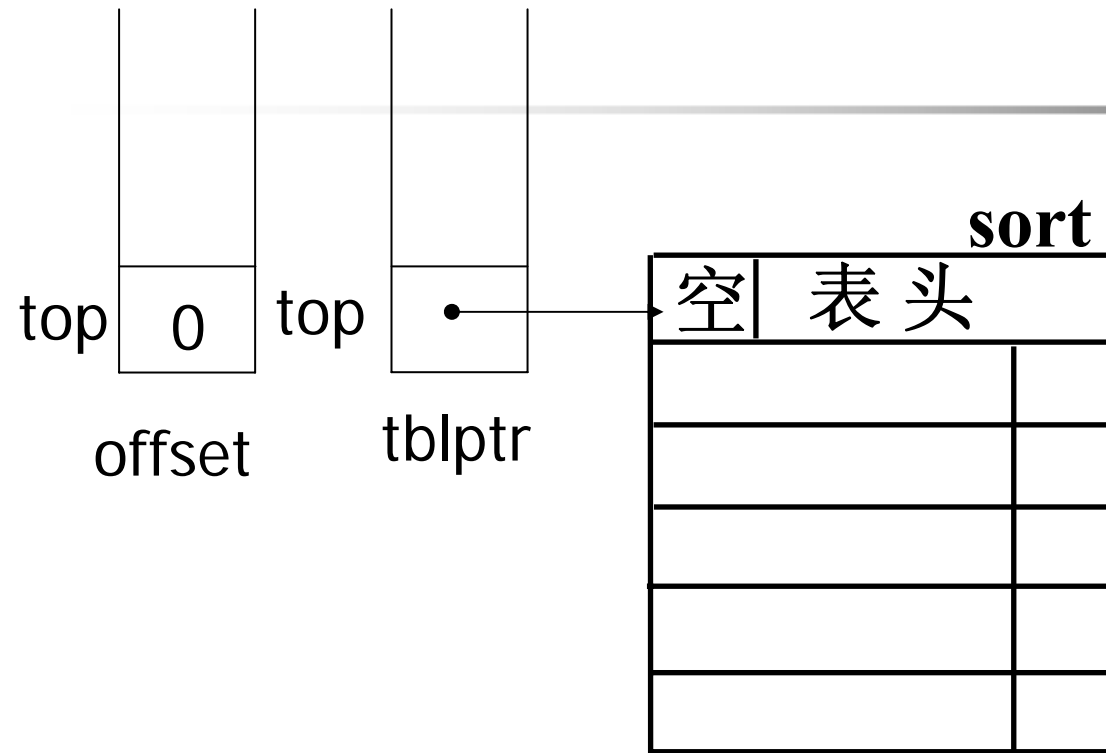
```

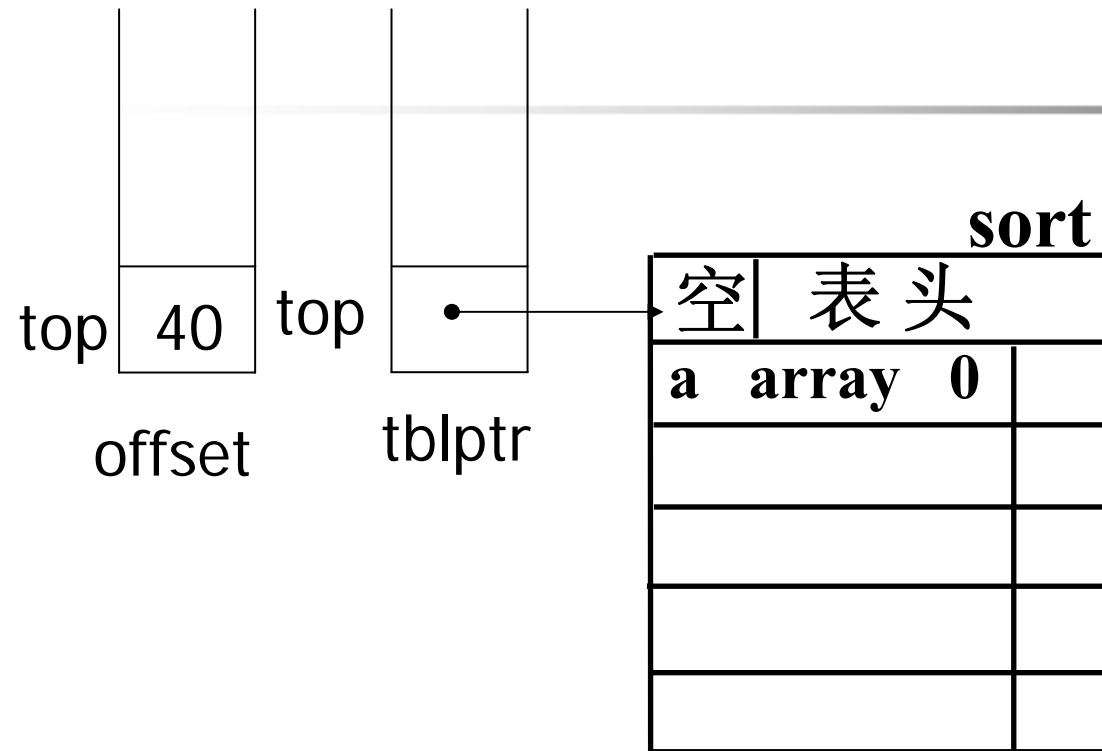
program sort(input,output);
  var a:array[0..10] of integer;
  x:integer;
procedure readarray;
  var i:integer; begin ...a...end;
procedure exchange(i,j:integer);
  begin x:=a[i];a[i]:=a[j];a[j]:=x;end;
procedure quicksort(m,n:integer);
  var k,v:integer;
  function partition(y,z:integer):integer;
    var i,j:integer;
    begin ...a...
      ...v...
    ...exchange(i,j)...end;
  begin ... end;
begin ... end;

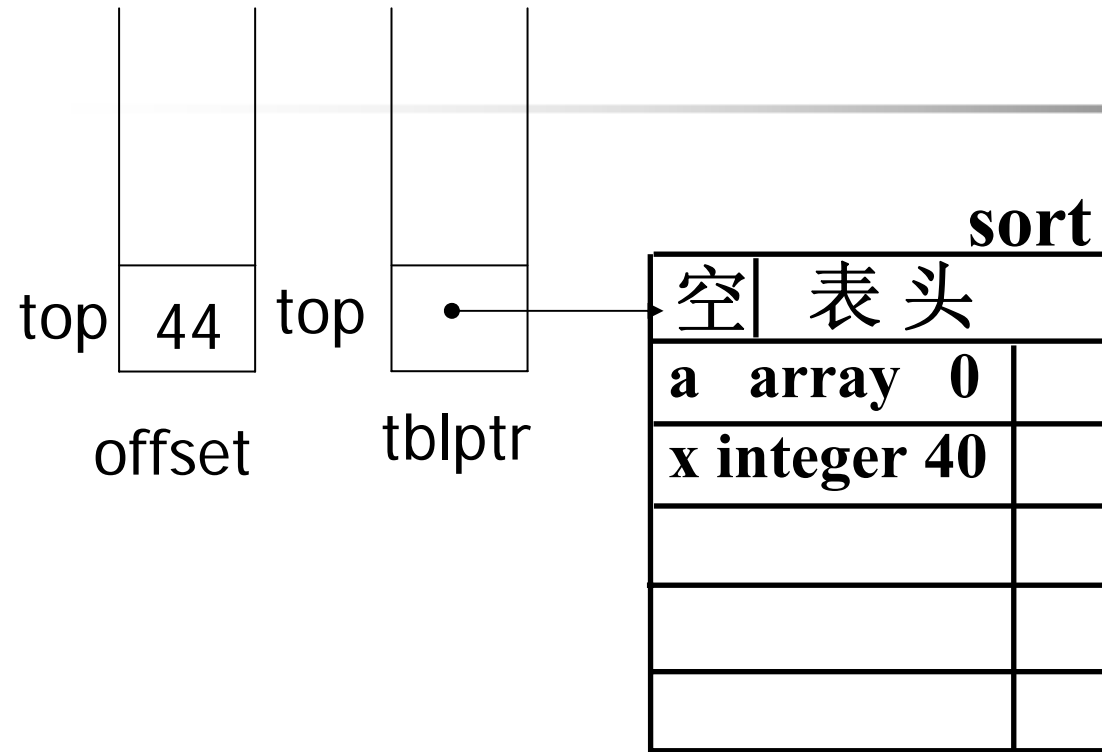
```

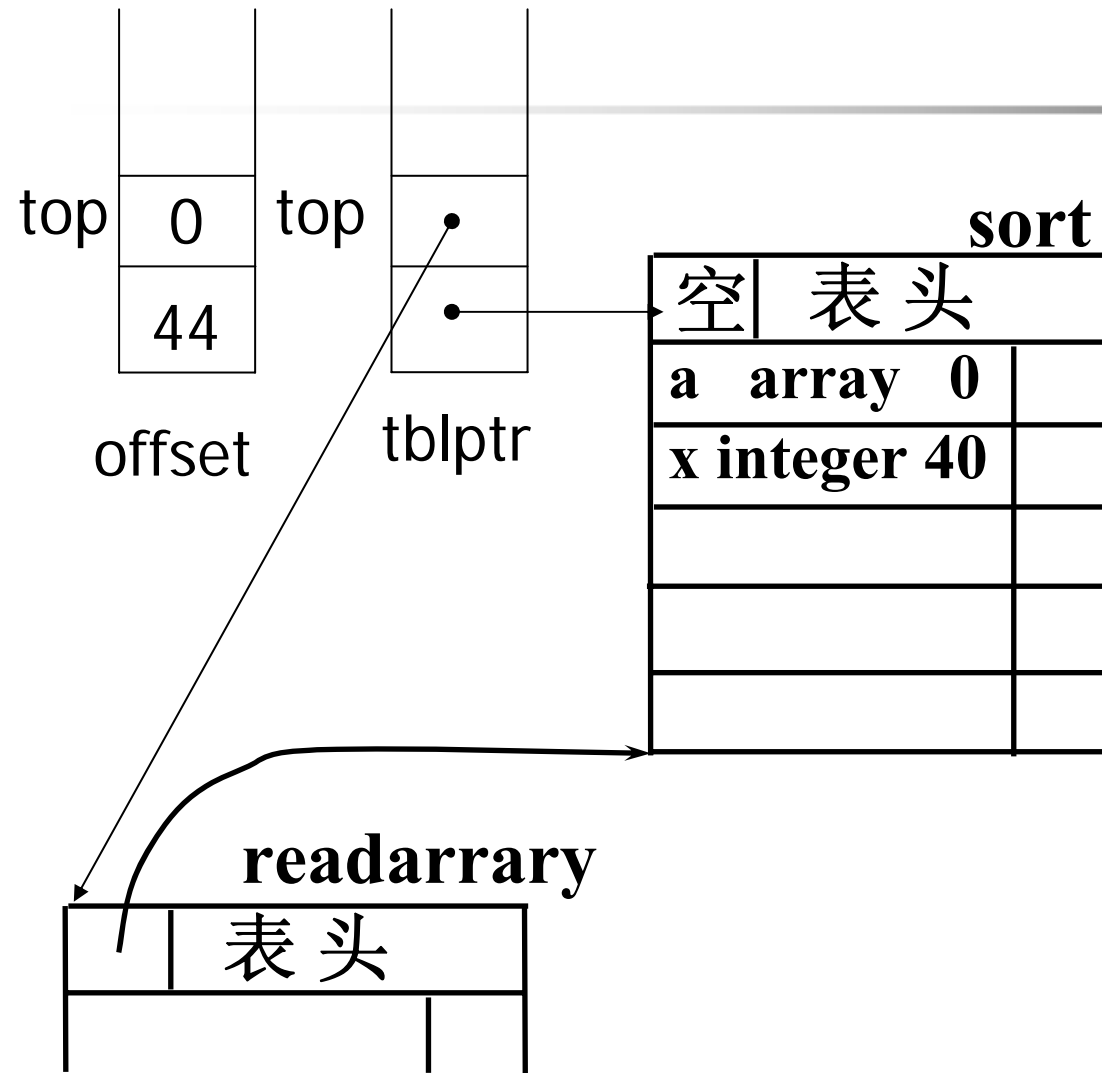
**例 一个带有嵌套的  
pascal程序(图7.11)**



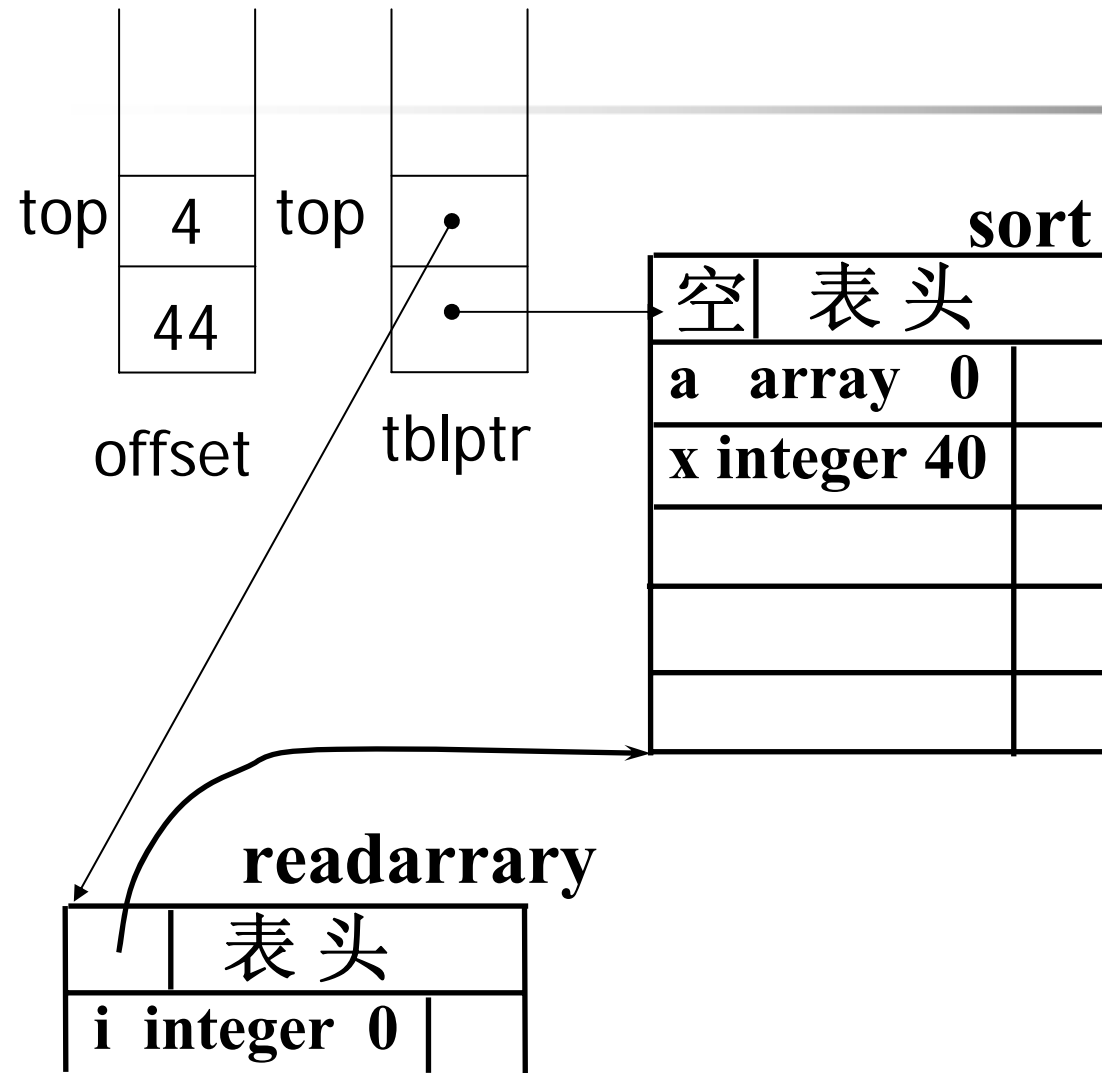


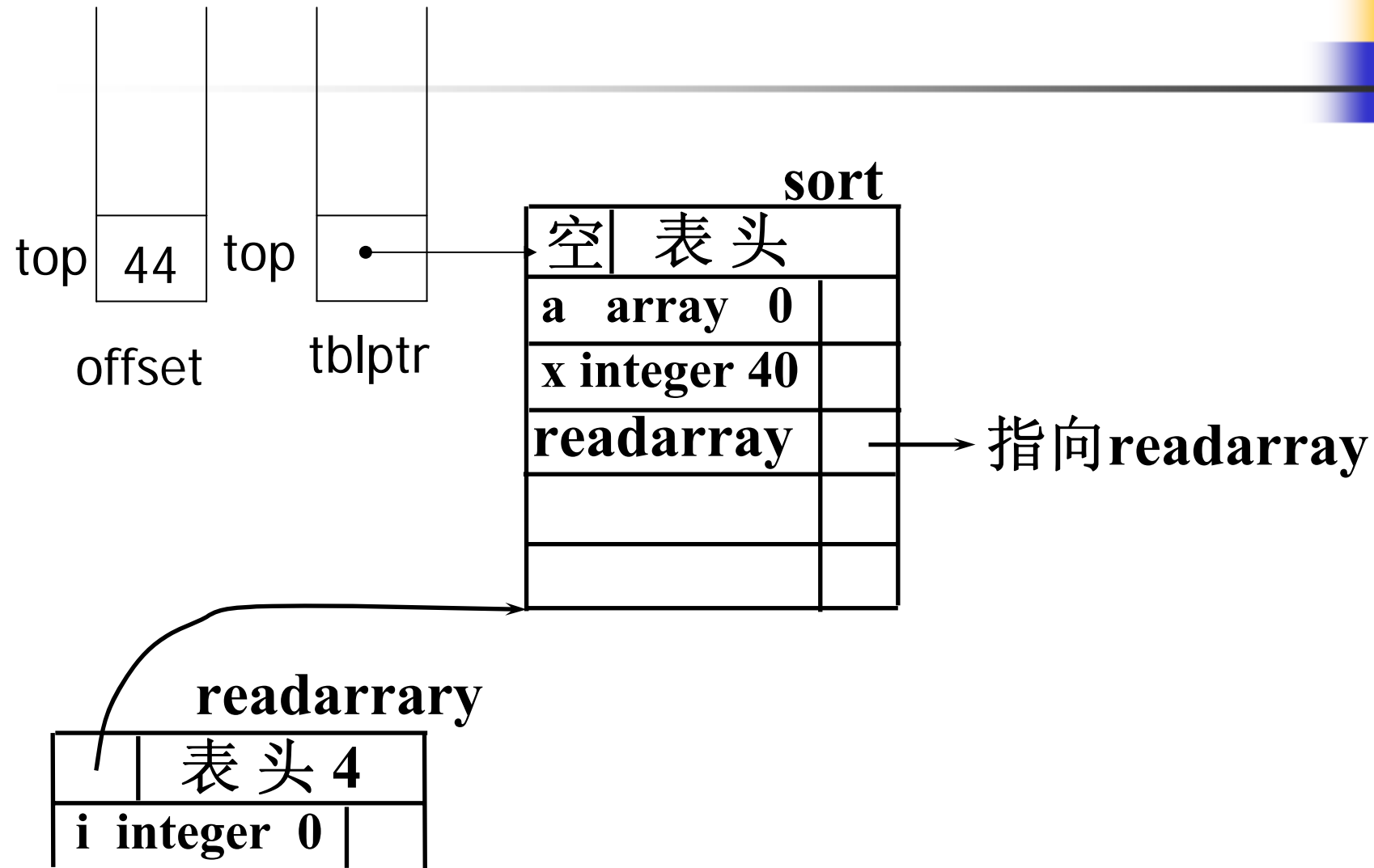


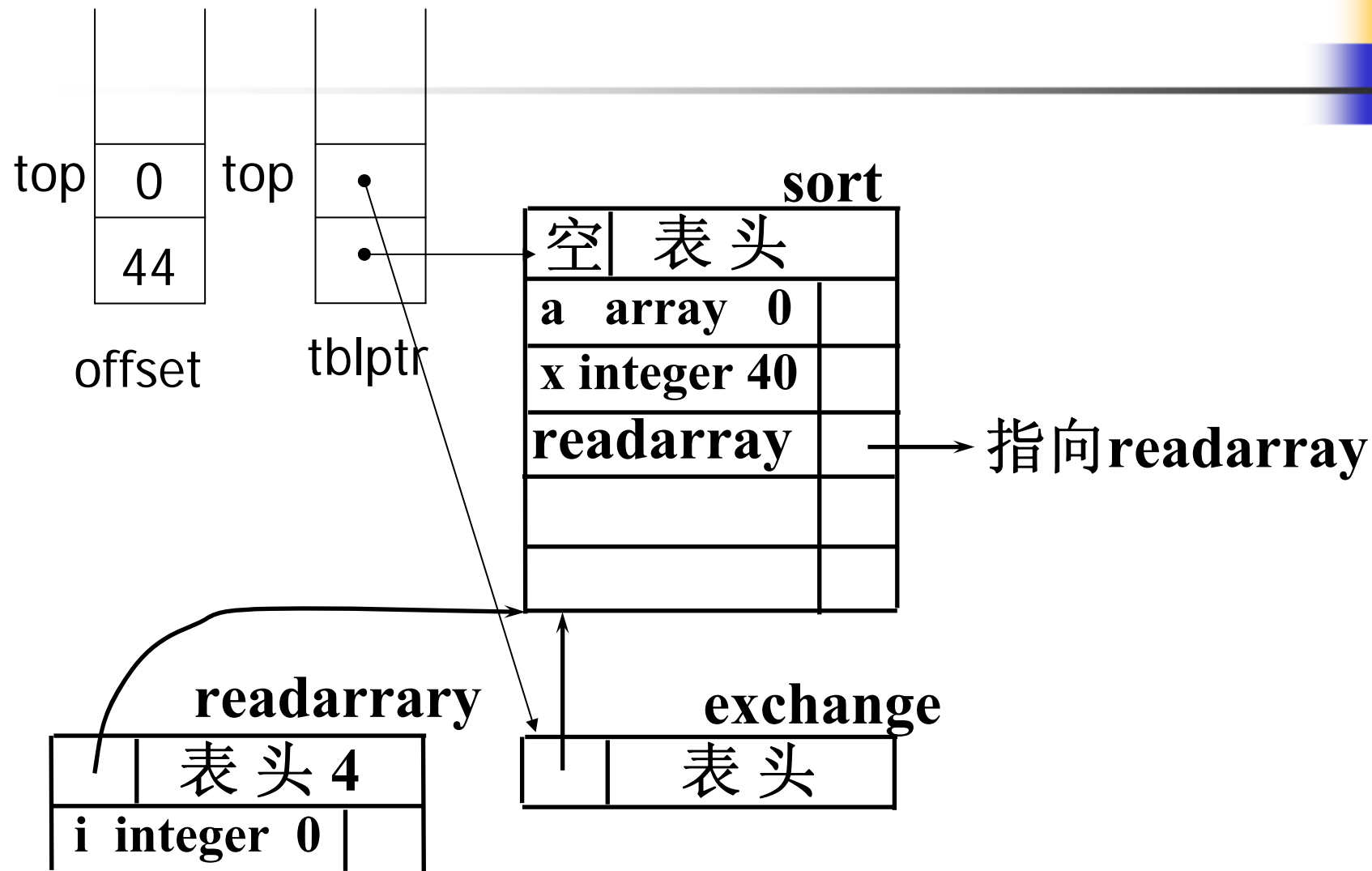


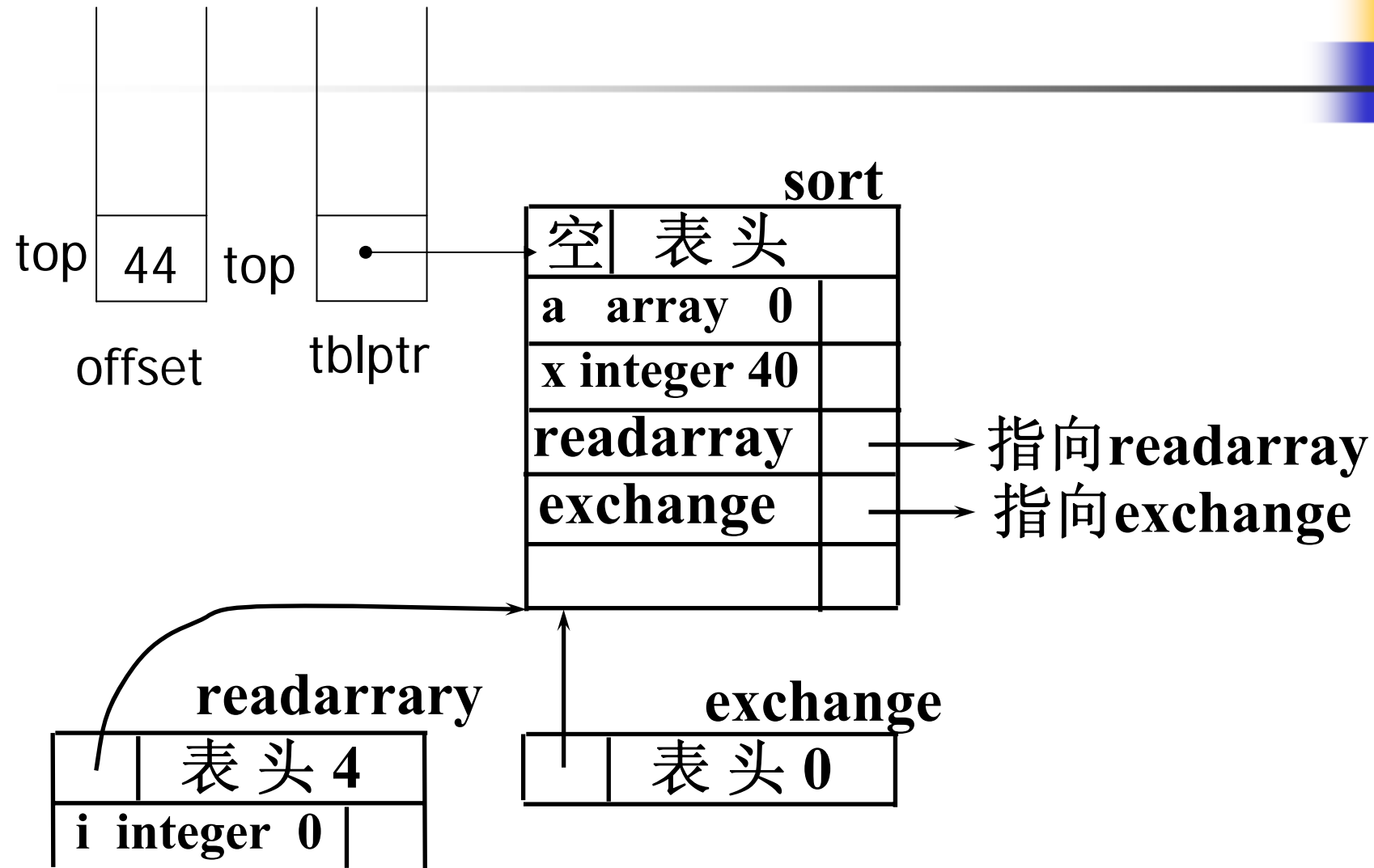


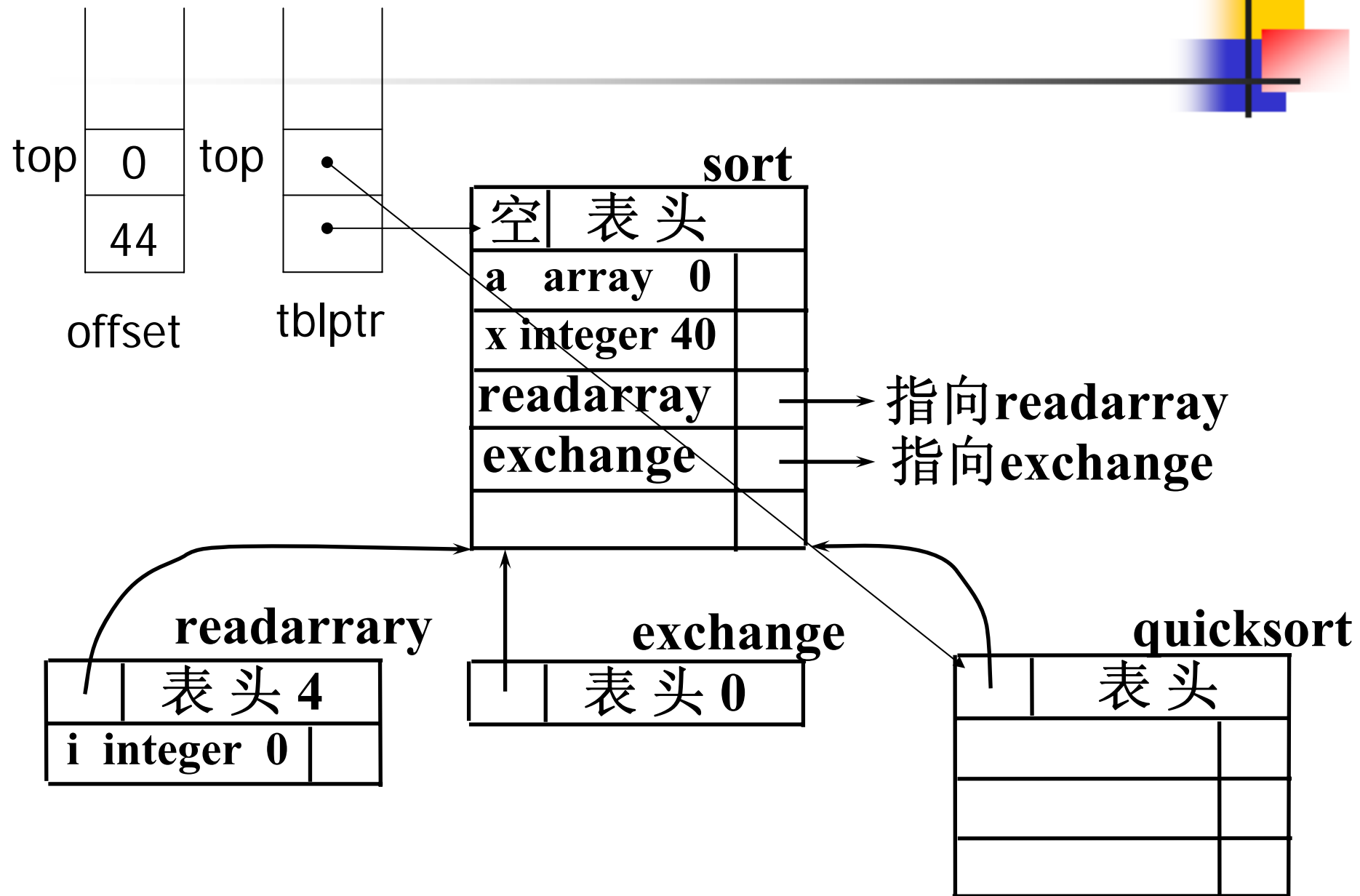


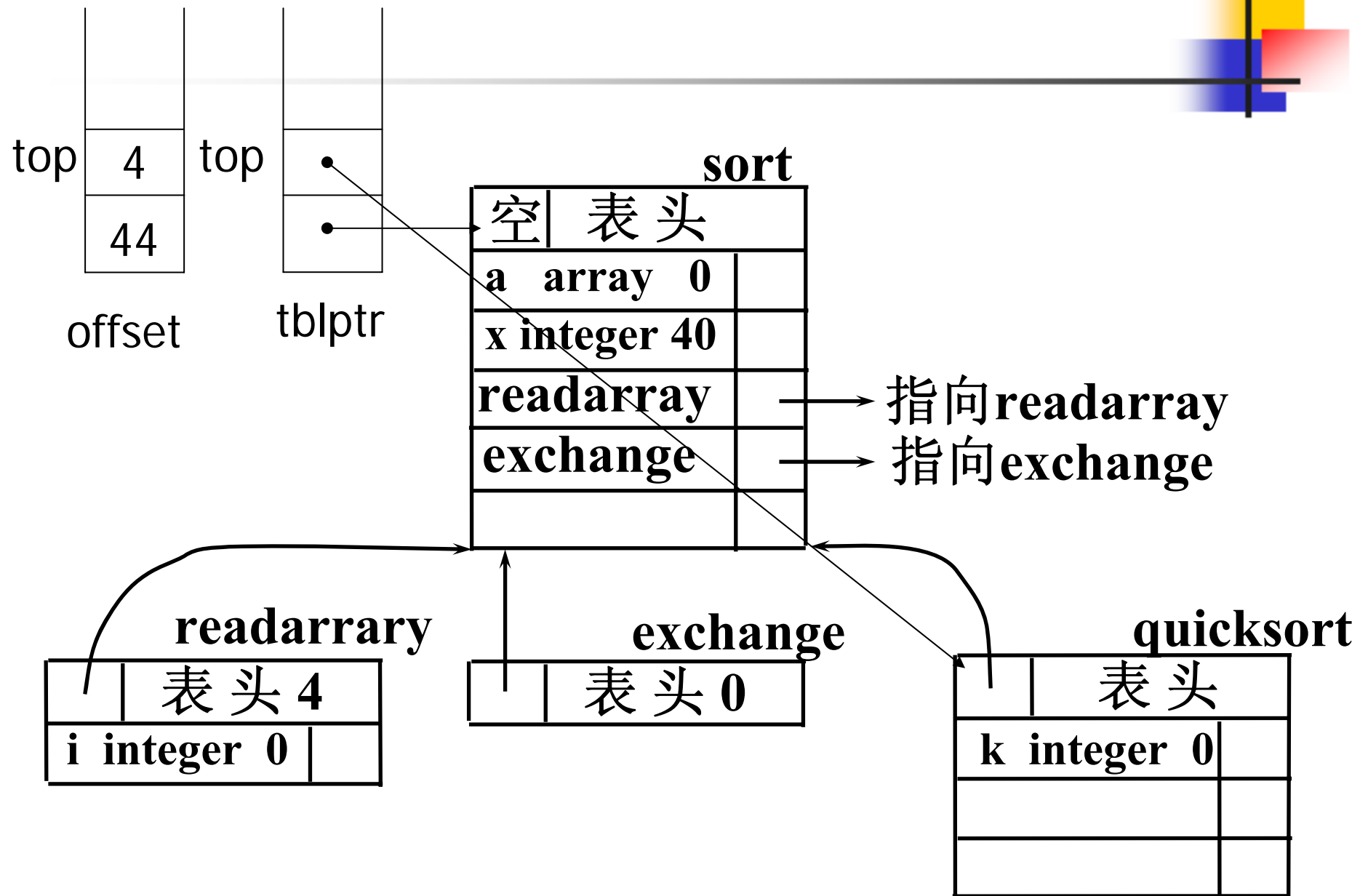




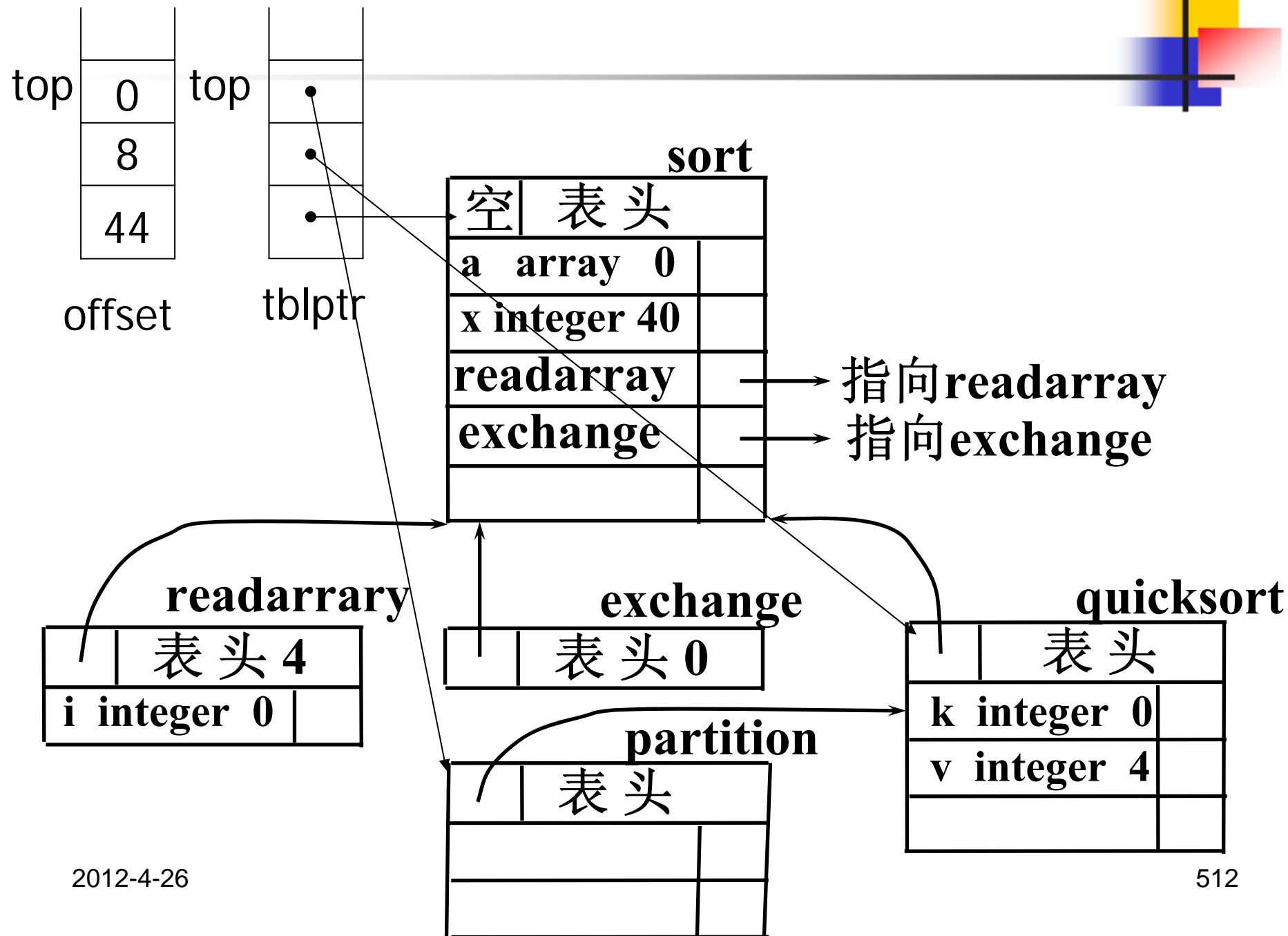




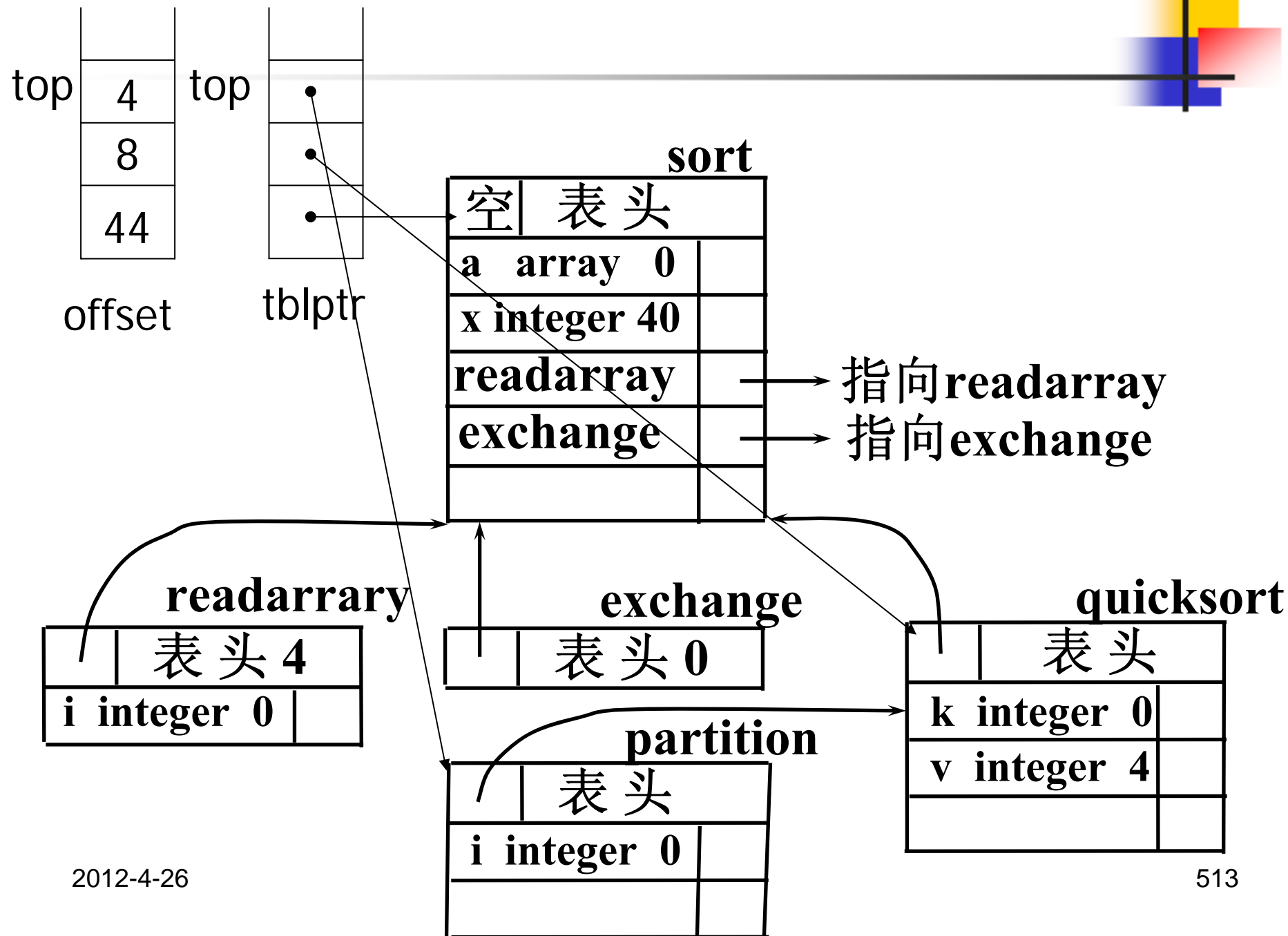


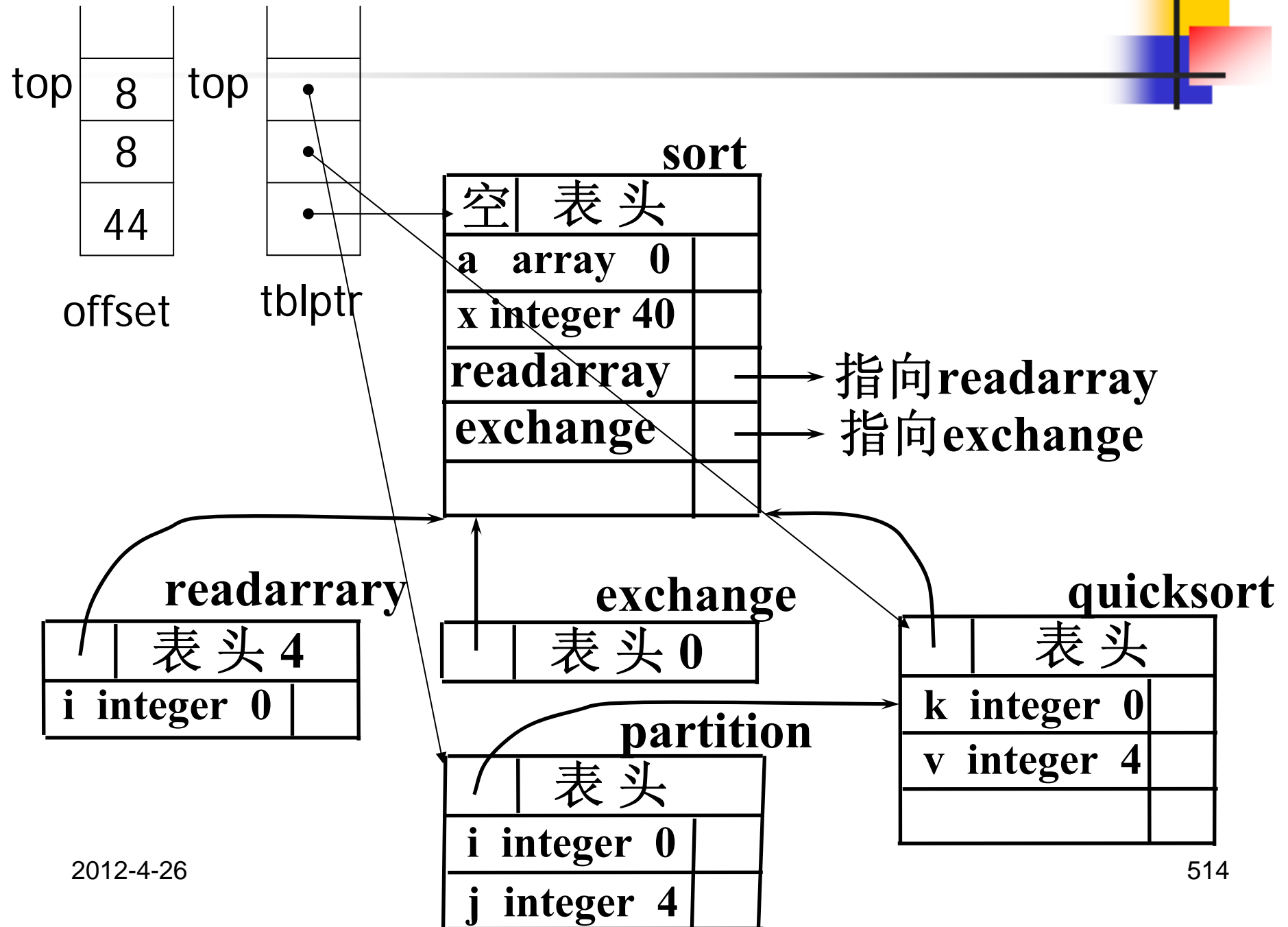


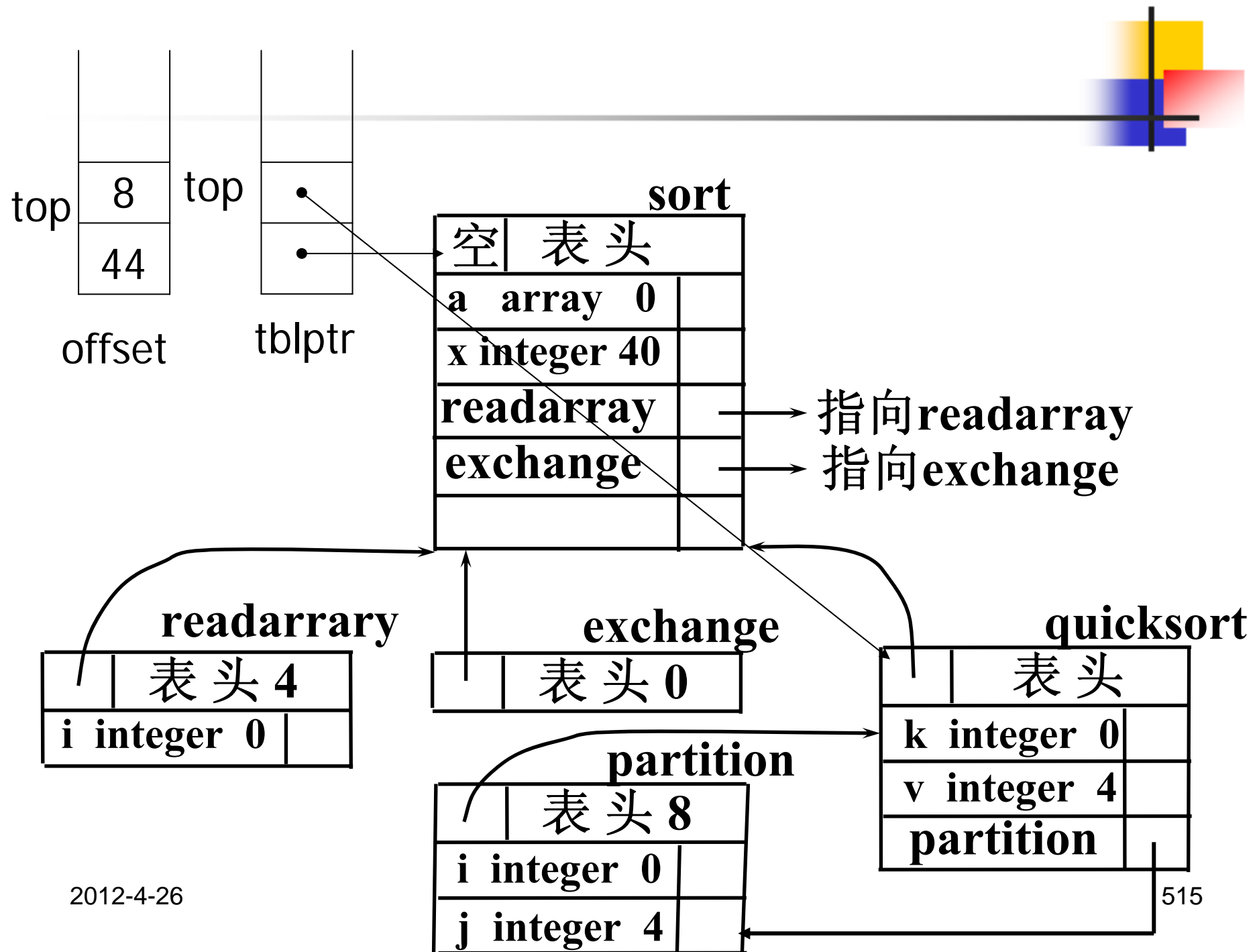


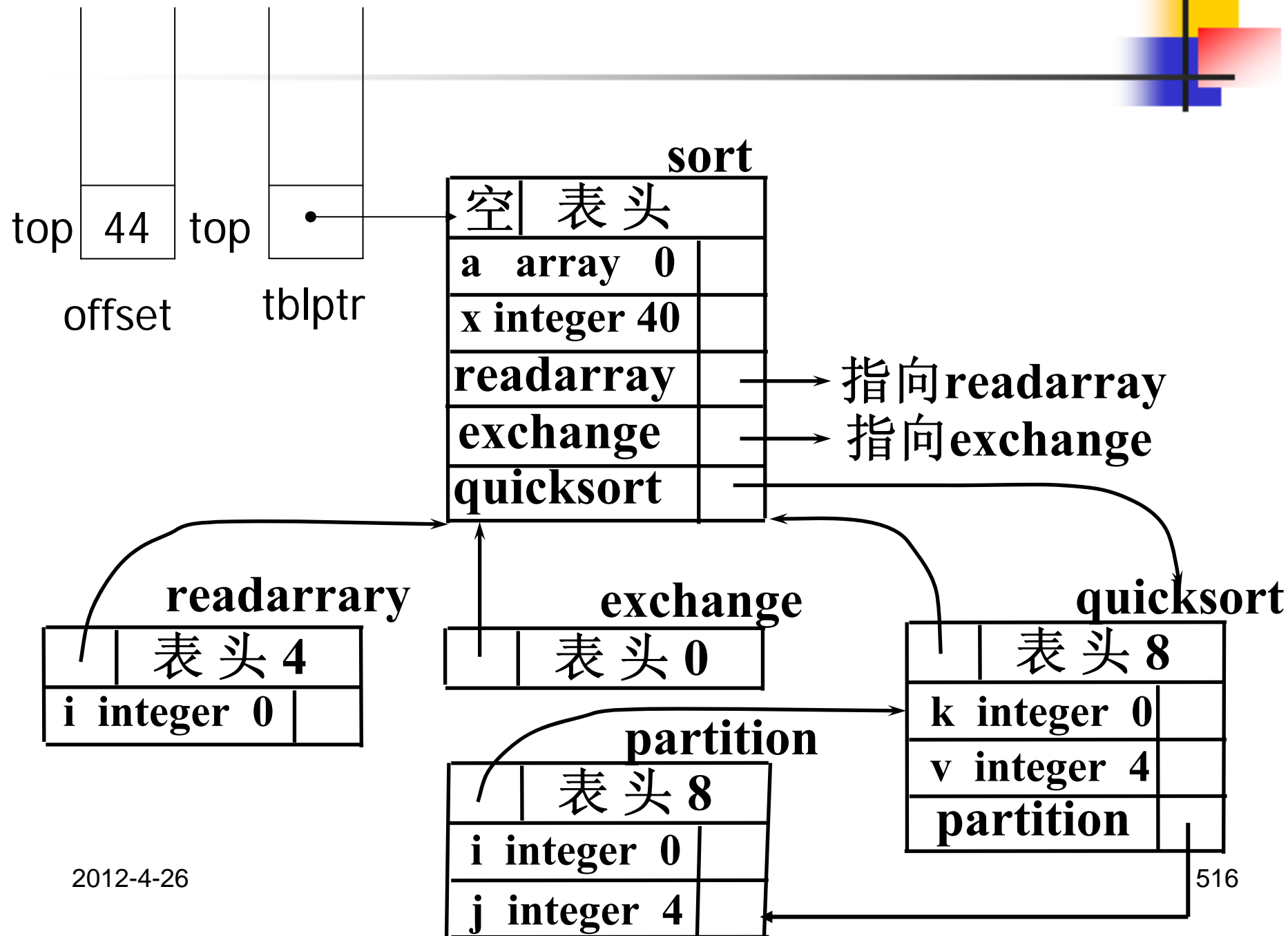


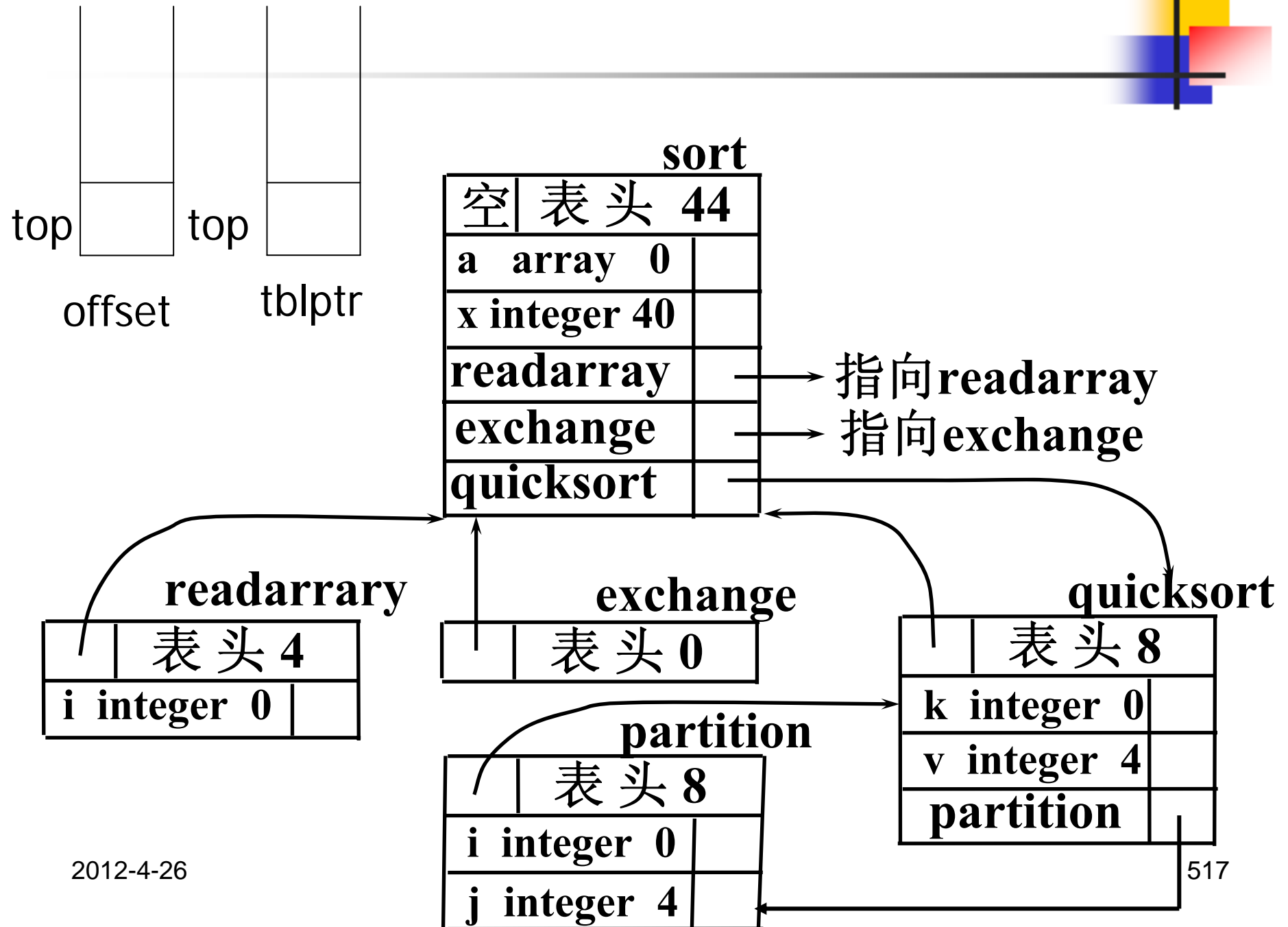












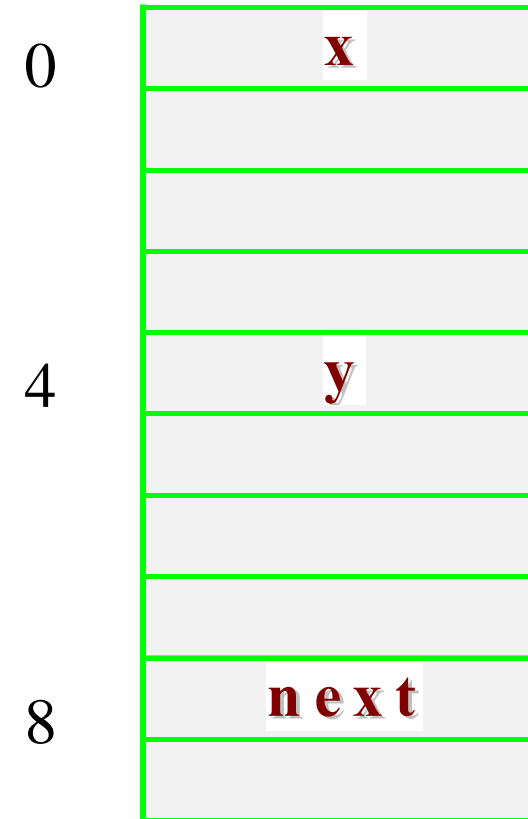
## 7.2.6 记录的翻译

- 空间分配

- 设置首地址和各元素的相对地址
- 大于所需的空间 (对齐)

- 例:

```
struct Node {  
    float x, y;  
    struct node *next;  
} node;
```





## 7.2.6 记录的翻译

---

### ■ 符号表及有关表格处理

- 为每个记录类型单独构造一张符号表
- 将域名id的信息(名字、类型、字节数)填入到该记录的符号表中
- 所有域都处理完后，offset将保存记录中所有数据对象的宽度总和
- T.type通过将类型构造器record应用于指向该记录符号表的指针获得



## 7.2.6 记录的翻译

---

**$T \rightarrow \text{record } D \text{ end}$**

**$T \rightarrow \text{record } L D \text{ end}$**

**$\{T.type := \text{record}(\text{top}(\text{tblptr}));$**

**$T.width := \text{top}(\text{offset});$**

**$\text{pop}(\text{tblptr}); \text{pop}(\text{offset}) \}$**

**$L \rightarrow \varepsilon$**

**$\{t := \text{mktable}(\text{nil});$**

**$\text{push}(t, \text{tblprt}); \text{push}(0, \text{offset}) \}$**





## 7.3 赋值语句的翻译

---

### 翻译的需求

- 充分了解各种语言现象的语义
  - 包括：控制结构、数据结构、单词
  - 充分了解它们的实现方法
- 目标语言的语义
  - 了解中间代码的语义
  - 了解运行环境



# 辅助子程序与语义属性设置

---

## ■ 辅助子程序

- **gencode(code)/emit(code):** 产生一条中间代码
- **newtemp:** 产生新的临时变量
- **lookup:** 检查符号表中是否出现某名字

## ■ 语义属性设置

- 中间代码序列: **code**
- 地址: **addr**
- 下一条四元式序号: **nextquad**



## 7.3.1 简单赋值语句的翻译

---

$S \rightarrow id := E$        $\{p := \text{lookup}(id.name);$   
                          if  $p \neq \text{nil}$  then  
                               $\text{gencode}(p, ':=', E.addr)$   
                          else error }

$E \rightarrow E_1 + E_2$        $\{E.addr := \text{newtemp};$   
                           $\text{gencode}(E.addr, ':=', E_1.addr, '+', E_2.addr)\}$

$E \rightarrow -E_1$   $\{E.addr := \text{newtemp};$   
                           $\text{gencode}(E.addr, ':=', 'uminus', E_1.addr)\}$

$E \rightarrow (E_1)$   $\{E.addr := E_1.addr \}$

$E \rightarrow id$      $\{p := \text{lookup}(id.name);$   
                  if  $p \neq \text{nil}$  then  $E.addr := p$  else error }



# 临时名字的重用

- 大量临时变量的使用对优化有利
- 大量临时变量会增加符号表管理的负担
- 也会增加运行时临时数据占的空间

$E \rightarrow E_1 + E_2$  的动作产生的代码的一般形式为

计算  $E_1$  到  $t_1$

计算  $E_2$  到  $t_2$

$t_3 := t_1 + t_2$

$(( ))(( ( ) ( ) ) ( ))$

临时变量的生存期像配对括号那样嵌套或并列

# 基于临时变量生存期特征的三地址代码

$x := a * b + c * d - e * f$

语 句	计数器c的值
	0
$\$0 := a * b$	1
$\$1 := c * d$	2
$\$0 := \$0 + \$1$	1
$\$1 := e * f$	2
$\$0 := \$0 - \$1$	1
$x := \$0$	0

引入一个计数器c，它的初值为0。每当临时变量仅作为运算对象使用时，c减去1；每当要求产生新的临时名字时，使用\$c并把c增加1。

## 7.3.2 数组元素的寻址

### 数组说明的翻译

- 符号表及有关表格(内情向量)的处理
- 维数、下标上界、下标下界、类型等
- 首地址、需用空间计算
- 按行存放、按列存放——影响具体元素地址的计算

### ■ 数组元素引用的翻译

- 完成上下界检查
- 生成完成相对地址计算的代码

### ■ 目标

- $x := y[i]$  和  $y[i] := x$
- $y$  为数组地址的固定部分——相当于第1个元素的地址， $i$  是相对地址，不是数组下标



## 数组元素地址计算——行优先

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

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

- 一维数组  $A[\text{low}_1:\text{up}_1]$  ( $n_k = \text{up}_k - \text{low}_k + 1$ )

$$\text{addr}(A[i]) = \text{base} + (i - \text{low}_1) * w$$

$$= (\text{base} - \text{low}_1 * w) + i * w = c + i * w$$

- 二维数组  $A[\text{low}_1:\text{up}_1, \text{low}_2:\text{up}_2]; A[i_1, i_2]$

$$\text{addr}(A[i_1, i_2]) = \text{base} + ((i_1 - \text{low}_1) * n_2 + (i_2 - \text{low}_2)) * w$$

$$= \text{base} + (i_1 - \text{low}_1) * n_2 * w + (i_2 - \text{low}_2) * w$$

$$= \text{base} - \text{low}_1 * n_2 * w - \text{low}_2 * w + i_1 * n_2 * w + i_2 * w$$

$$= \text{base} - (\text{low}_1 * n_2 - \text{low}_2) * w + (i_1 * n_2 + i_2) * w$$

$$= c + (i_1 * n_2 + i_2) * w$$



# 数组元素地址计算的翻译方案设计

---

下标变量访问的产生式

$L \rightarrow id[Elist] \mid id$

$Elist \rightarrow Elist, E \mid E$

为了使数组各维的长度 $n$ 在我们将下标表达式合并到 $Elist$ 时是可用的，需将产生式改写为：

$L \rightarrow Elist \mid id$

$Elist \rightarrow Elist, E \mid id[E]$





# 数组元素地址计算的翻译方案设计

---

$L \rightarrow Elist \mid id$

$Elist \rightarrow Elist, E \mid id[E]$

## ■ 目的

- 使我们在整个下标表达式列表Elist的翻译过程中随时都能知道符号表中相应于数组名id的表项，从而能够了解登记在符号表中的有关数组id的全部信息。
- 于是我们就可以为非终结符Elist引进一个综合属性Elist.array，用来记录指向符号表中相应数组名字表项的指针。



# 数组元素地址计算的翻译方案设计

---

## ■ 属性

- **Elist.array**, 用来记录指向符号表中相应数组名字表项的指针。
- **Elist.ndim**, 用来记录Elist中下标表达式的个数, 即数组的维数。
- **Elist.addr**, 用来临时存放Elist的下标表达式计算出来的值。

## ■ 函数

- **limit(array, j)**, 返回 $n_j$

### 7.3.3 带有数组引用的赋值语句的翻译

---

$S \rightarrow \text{Left} := E$

$E \rightarrow E + E$

$E \rightarrow (E)$

$E \rightarrow \text{Left}$

$\text{Left} \rightarrow \text{Elist}$

$\text{Left} \rightarrow \text{id}$

$\text{Elist} \rightarrow \text{Elist}, E$

$\text{Elist} \rightarrow \text{id}[E$

# 赋值语句的翻译模式

**Left** → **id** {Left.addr := id.addr; Left.offset := null}

**Elist** → **id**[**E** {Elist.addr := E.addr;  
Elist.ndim := 1;  
Elist.array := id.addr }

**Elist** → **Elist**<sub>1</sub>, **E** {t := newtemp; m := Elist<sub>1</sub>.ndim + 1;

**i**<sub>1</sub> \* **n**<sub>2</sub> encode(t, ':=' , Elist<sub>1</sub>.addr, '\*', limit(Elist<sub>1</sub>.array, m));  
gencode(t, ':=' , t, '+', E.addr);

**(i**<sub>1</sub> \* **n**<sub>2</sub>) + **i**<sub>2</sub> Elist.array := Elist<sub>1</sub>.array;  
Elist.addr := t; Elist.ndim := m}

**Left** → **Elist** { Left.addr := newtemp;  
Left.offset := newtemp  
base - (low<sub>1</sub> \* n<sub>2</sub> - low<sub>2</sub>) \* w  
gencode(Left.addr, ':=' , base(Elist.array), '-', invariant(Elist.array));  
gencode(Left.offset, ':=' , Elist.addr, '\*', w)}

**((i**<sub>1</sub> \* **n**<sub>2</sub>) + **i**<sub>2</sub>) \* **w**

# 赋值语句的翻译模式

**E** → **Left** {if Left.offset = null then /\* Left是简单变量 \*/

E.addr := Left.addr

else begin E.addr := newtemp;

gencode(E.addr, ':=', Left.addr, '[', Left.offset, ']')end}

**E** → **E<sub>1</sub> + E<sub>2</sub>** {E.addr := newtemp;

gencode(E.addr, ':=', E<sub>1</sub>.place, '+', E<sub>2</sub>.addr)}

**E** → **(E<sub>1</sub>)** {E.addr := E<sub>1</sub>.addr }

**S** → **Left := E** {if Left.offset=null then /\*Left是简单变量\*/

gencode(Left.addr, ':=' , E.addr)

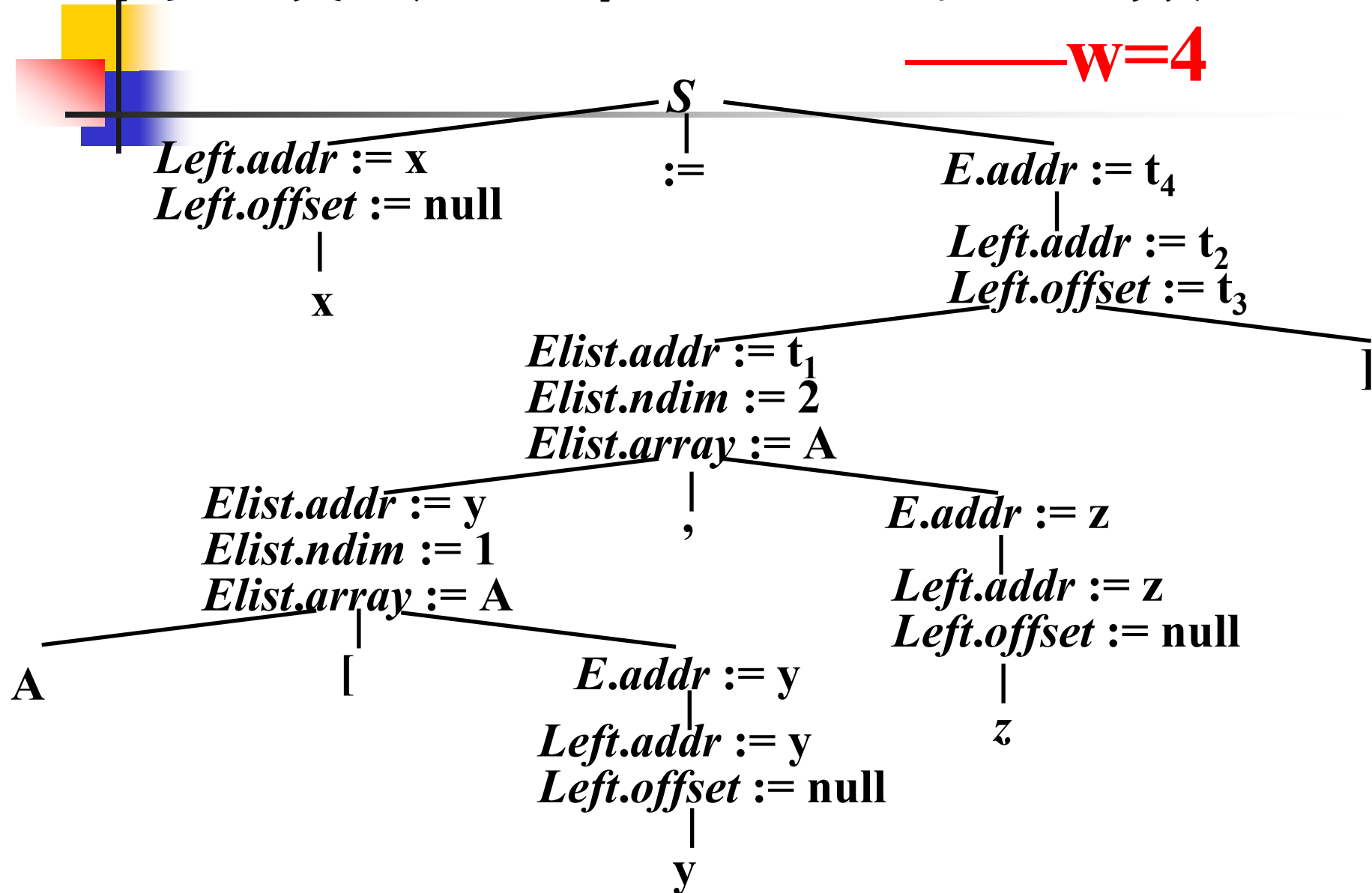
else

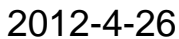
gencode(E.addr, ':=' , Left.addr, '[', Left.offset, ']')

((base-(low<sub>1</sub>\* n<sub>2</sub> -low<sub>2</sub>)\*w)

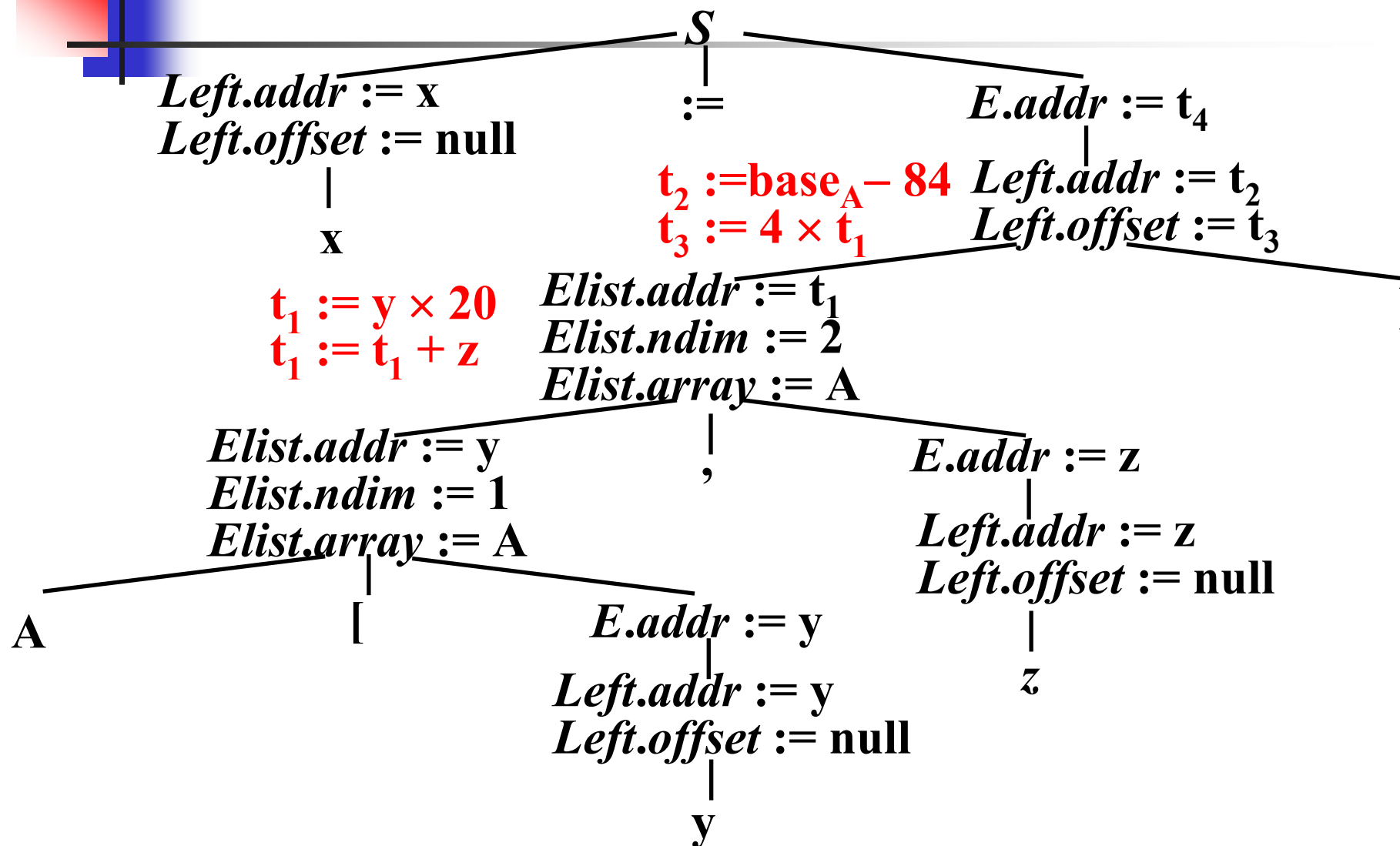
((i<sub>1</sub>\*n<sub>2</sub>)+i<sub>2</sub>)\*w+(base-(low<sub>1</sub>\* n<sub>2</sub> -low<sub>2</sub>)\*w)

例： 设A是一个 $10 \times 20$ 的整型数组





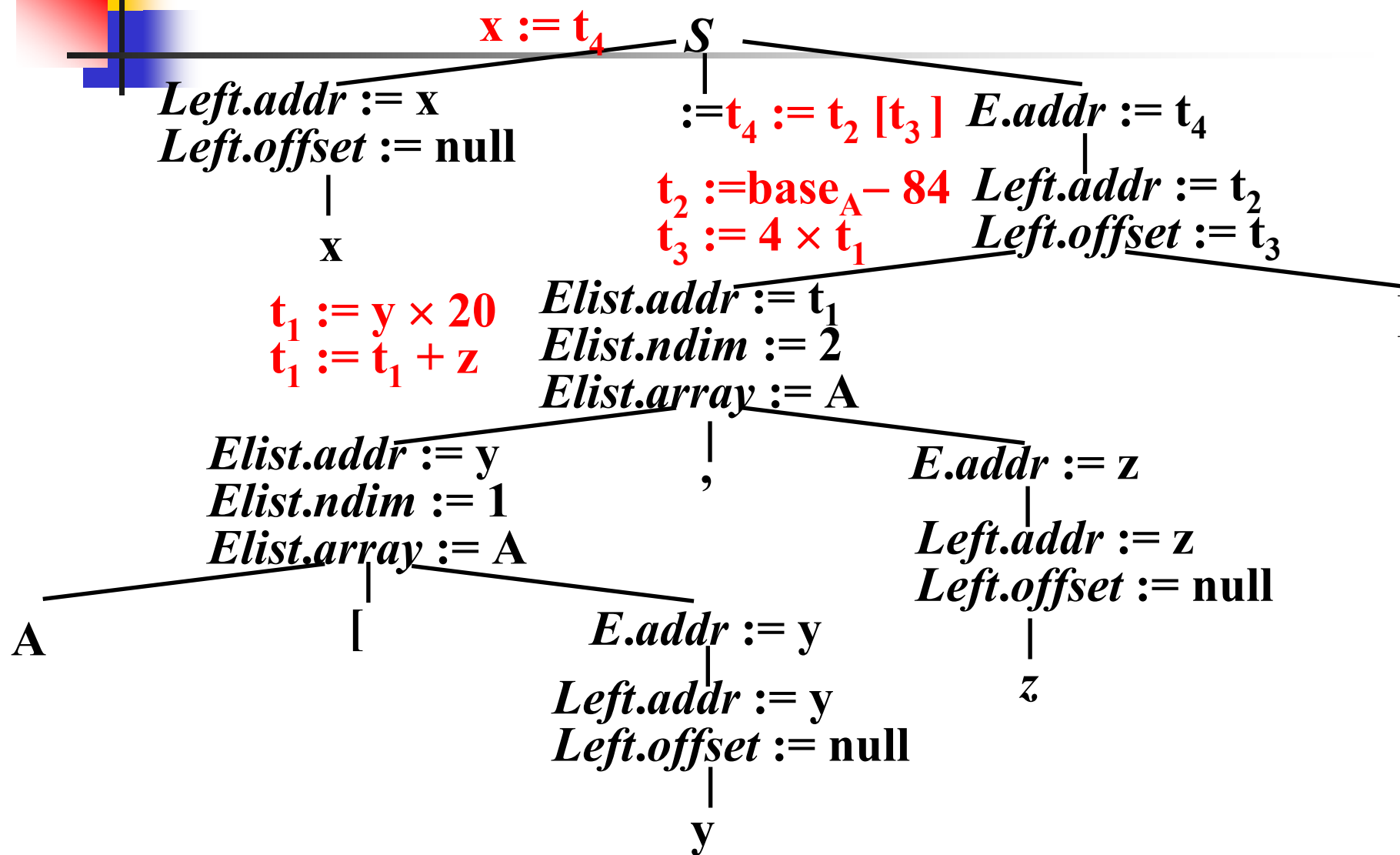
例







例





## 7.4 类型检查

- 类型检查具有发现程序错误的能力，为了进行类型检查，编译程序需要为源程序的每个语法成分赋予一个类型表达式，名字的类型表达式保存在符号表中，其他语法成分(如表达式 $E$ 或语句 $S$ )的类型表达式则作为文法符号的属性保存在语义栈中。
- 类型检查的任务就是确定这些类型表达式是否符合一定的规则，这些规则的集合通常称为源程序的类型系统



## 7.4.1 类型检查的规则

### ■ 类型综合

- 从子表达式的类型确定表达式的类型
- 要求名字在引用之前必须先进行声明
- if  $f$  的类型为  $s \rightarrow t$  and  $x$  的类型为  $s$  then 表达式  $f(x)$  的类型为  $t$

### ■ 类型推断

- 根据语言结构的使用方式来确定其类型
- 经常被用于从函数体推断函数类型
- if  $f(x)$  是一个表达式 then 对某两个类型变量  $\alpha$  和  $\beta$ ,  $f$  具有类型  $\alpha \rightarrow \beta$  and  $x$  具有类型  $\alpha$



## 7.4.2 类型转换

---

$x := y + i * j$

( $x$ 和 $y$ 的类型是real,  $i$ 和 $j$ 的类型是integer)

中间代码

$t_1 := i \text{ int} \times j$

$t_2 := \text{intto real } t_1$

$t_3 := y \text{ real} + t_2$

$x := t_3$



## 7.4.2 类型转换

---

**$E \rightarrow E_1 + E_2$  的语义子程序**

```
{E.addr := newtemp
if E1.type = integer and E2.type = integer then begin
    gencode(E.addr, ':=', E1.addr, 'int+', E2.addr);
    E.type = integer
end
else if E1.type = integer and E2.type = real then begin
    u := newtemp;
    gencode(u, ':=', inttoreal, E1.addr);
    gencode(E.addr, ':=', u, 'real+', E2.addr);
    E.type := real
end    ... }
```



## 7.5 控制结构的翻译

---

- 高级语言的控制结构
  - 顺序结构 `begin 语句; ... ; 语句 end`
  - 分支结构 `if_then_else`、`if_then`  
`switch`、`case`
  - 循环结构 `while_do`、`do_while`  
`for`、`repeat_until`
  - `goto`语句
- 三地址码
  - `goto n` (`j, _, _, n`)
  - `if x relop y goto n` (`jrelop, x, y, n`)



## 7.5.1 布尔表达式的翻译

---

### ■ 基本文法

$B \rightarrow B_1 \text{ or } B_2 \mid B_1 \text{ and } B_2 \mid \text{not } B_1 \mid (B_1) \mid E_1 \text{ relop } E_2 \mid \text{true} \mid \text{false}$

### ■ 处理方式

- 数值表示法(与算术表达式的处理类似)
  - 真:  $B.\text{addr} = 1$
  - 假:  $B.\text{addr} = 0$
- 真假出口表示法(作为其他语句的条件改变控制流程, 隐含着程序中的位置)
  - 真出口:  $B.\text{true}$
  - 假出口:  $B.\text{false}$





## 7.5.1 布尔表达式的翻译

---

- **a or b and not c**
  - $t_1 := \text{not } c$
  - $t_2 := b \text{ and } t_1$
  - $t_3 := a \text{ or } t_2$
- **a < b**
  - 100: if a < b goto 103
  - 101: t := 0
  - 102: goto 104
  - 103: t := 1
  - 104



## 用数值表示布尔值的翻译

- *nextquad*是下一条三地址码指令的序号，每生成一条三地址码指令*gencode*便会将*nextquad*加1

**$B \rightarrow B_1 \text{ or } B_2$**  {*B.addr* := newtemp;  
gencode(*B.addr* := '*B<sub>1</sub>.addr*' or '*B<sub>2</sub>.addr*')}

**$B \rightarrow B_1 \text{ and } B_2$**  {*B.addr* := newtemp;  
gencode(*B.addr* := '*B<sub>1</sub>.addr*' and '*B<sub>2</sub>.addr*')}

**$B \rightarrow \text{not } B_1$**  {*B.addr* := newtemp;  
gencode(*B.addr* := ' 'not'*B<sub>1</sub>.addr*')}



## 用数值表示布尔值的翻译

---

**$B \rightarrow (B_1)$**  {B.addr := B<sub>1</sub>.addr}

**$B \rightarrow E_1 \text{ relop } E_2$**  {B.addr := newtemp;

gencode('if'E<sub>1</sub>.addr relop.op E<sub>2</sub>.addr'goto'nextquad+3);

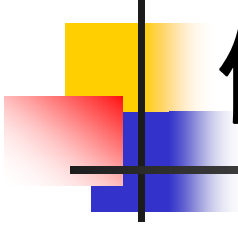
gencode(B.addr ':= '0') ;

gencode('goto'nextquad+2);

gencode(B.addr ':= '1'))}

**$B \rightarrow \text{true}$**  {B.addr := newtemp; gencode(B.addr ':= '1'))}

**$B \rightarrow \text{false}$**  {B.addr := newtemp; gencode(B.addr ':= '0'))}



## 例7.8 对 $a < b$ or $c < d$ and $e < f$ 的翻译

---

100: if  $a < b$  goto 103

101:  $t1 := 0$

102: goto 104

103:  $t1 := 1$

104: if  $c < d$  goto 107

105:  $t2 := 0$

106: goto 100

107:  $t2 := 1$

108: if  $e < f$  goto 111

109:  $t3 := 0$

110: goto 112

111:  $t3 := 1$

112:  $t4 := t2$  and  $t3$

113:  $t5 := t1$  or  $t4$



## 7.5.2 常见控制结构的翻译

---

### ■ 文法

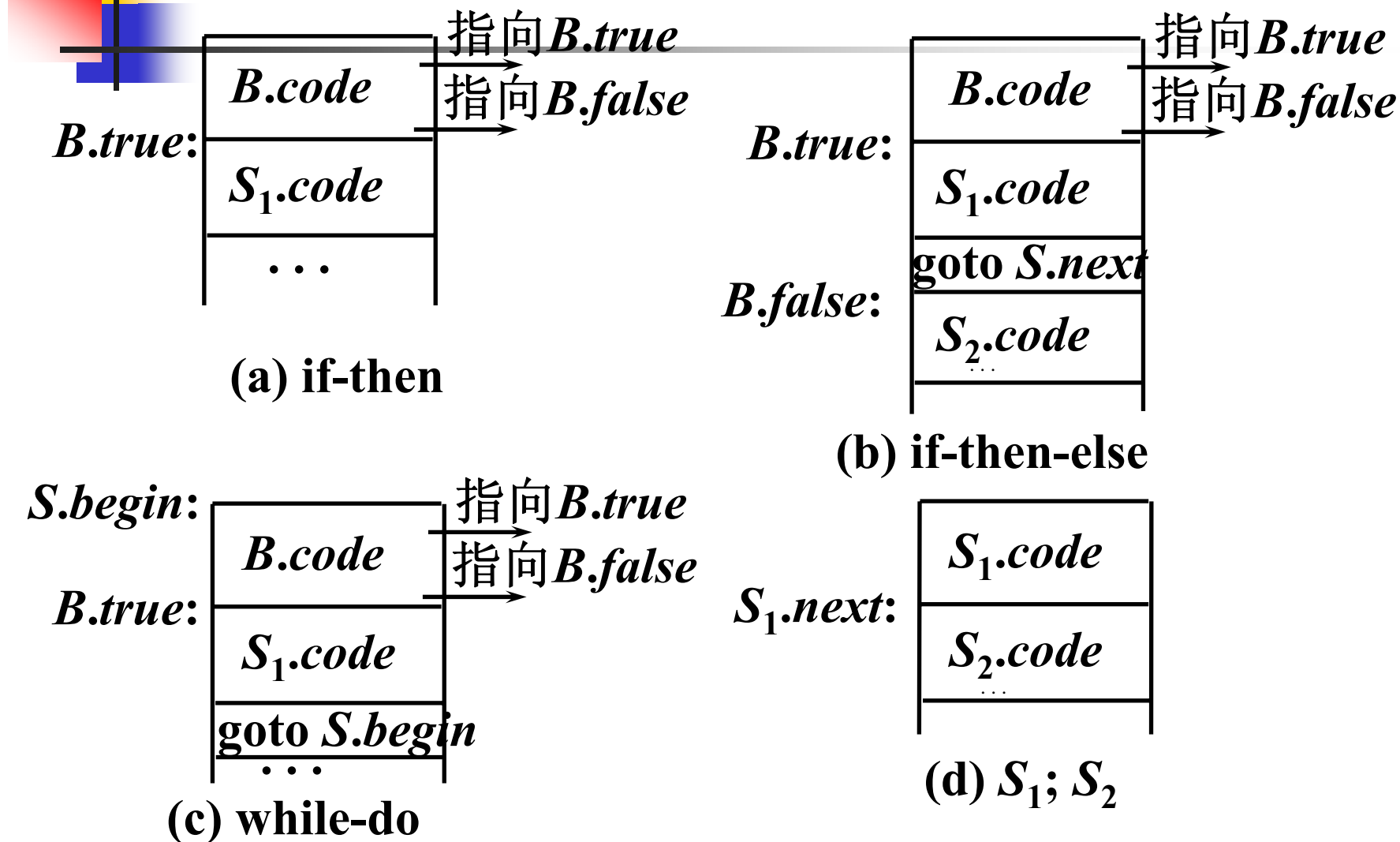
- $S \rightarrow \text{if } B \text{ then } S1$
- $S \rightarrow \text{if } B \text{ then } S1 \text{ else } S2$
- $S \rightarrow \text{while } B \text{ do } S1$
- $S \rightarrow \text{begin } Slist \text{ end}$
- $Slist \rightarrow Slist; S \mid S$
- **B**是控制结构中的布尔表达式
- 函数newlabel返回一个新的语句标号
- 属性**B.true**和**B.false**分别用于保存**B**为真和假时控制流要转向的语句标号



## 7.5.2 常见控制结构的翻译

- 翻译 $S$ 时允许控制从 $S.code$ 中跳转到紧接在 $S.code$ 之后的那条三地址码指令，但在某些情况下，紧跟 $S.code$ 的指令是跳转到某个标号 $L$ 的跳转指令，用继承属性 $S.next$ 可以避免从 $S.code$ 中跳转到一条跳转指令这样的连续跳转。 $S.next$ 的值是一个语句标号，它是 $S$ 的代码执行后应执行的第一个三地址码指令的标号。
- **while**语句中用 $S.begin$ 保存语句的开始位置

## 7.5.2 常见控制结构的翻译



## 7.5.2 常见控制结构的翻译

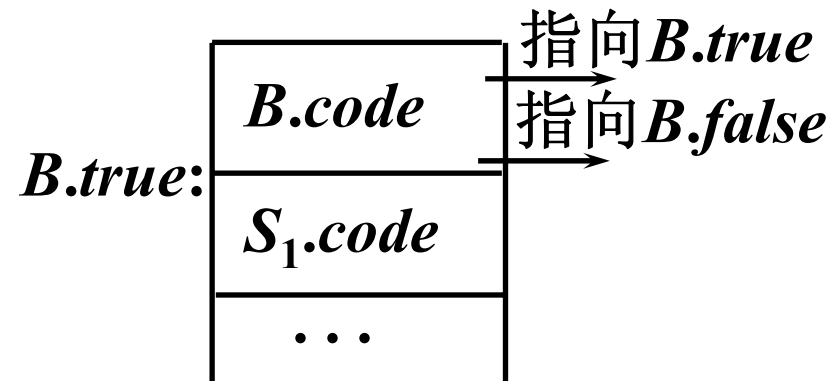
**$S \rightarrow \text{if } B \text{ then } S_1$**

**$\{B.true := \text{newlabel};$**

**$B.false := S.next;$**

**$S_1.next := S.next;$**

**$S.code := B.code || \text{gencode}(B.true, ':') || S_1.code \}$**



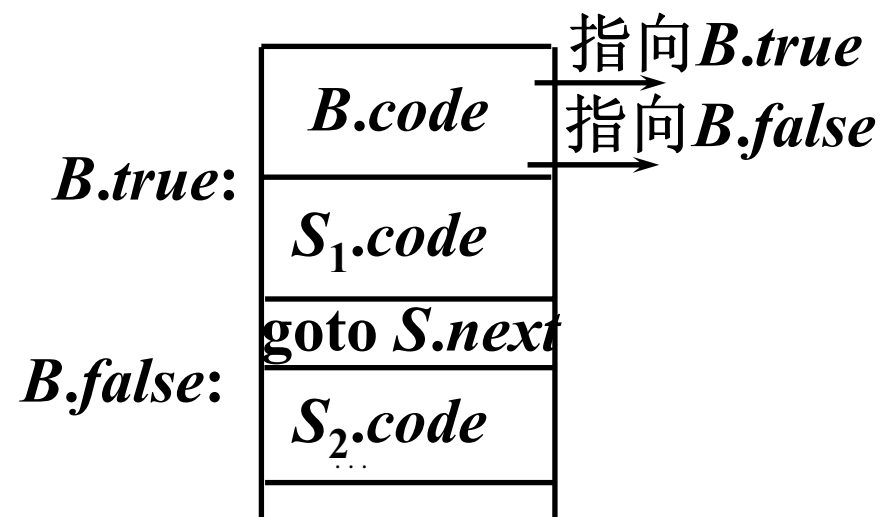
(a) if-then



## 7.5.2 常见控制结构的翻译

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

```
{B.true := newlabel;  
  B.false := newlabel;  
  S1.next := S.next;  
  S2.next := S.next;  
  S.code := B.code || gencode(B.true, ':') || S1.code ||  
  gencode('goto', S.next) || gen(B.false, ':') || S2.code}
```

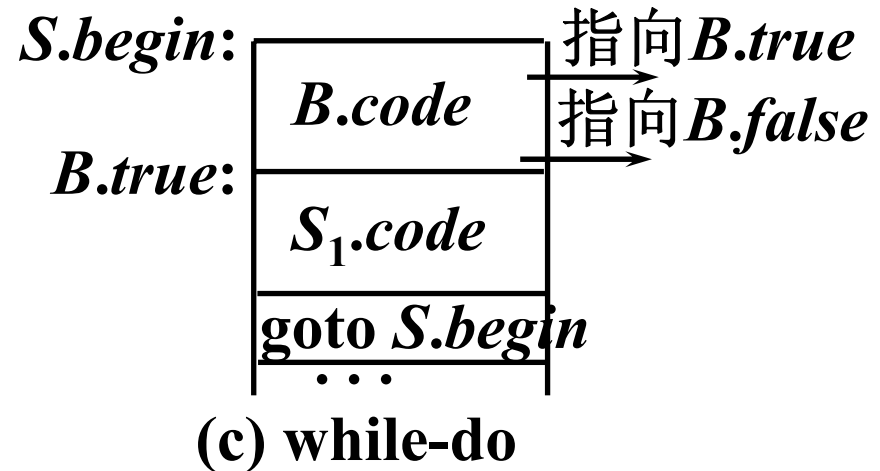


(b) if-then-else

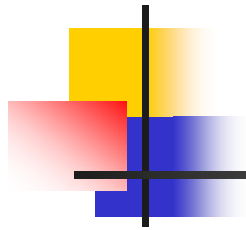
## 7.5.2 常见控制结构的翻译

**$S \rightarrow \text{while } B \text{ do } S_1$**

```
{S.begin:= newlabel;  
  B.true := newlabel;  
  B.false := S.next;  
  S1.next := S.begin;  
  S.code:=gencode(S.begin,':')||B.code||  
  gencode(B.true,':')||S1.code||  
  gencode('goto',S.begin)}
```



## 7.5.2 常见控制结构的翻译



$S \rightarrow S_1; S_2$

$\{S_1.next := newlabel; S_2.next := S.next;$   
 $S.code := S_1.code || gencode(S_1.next, ':') || S_2.code\}$

$S_1.next:$

$S_1.code$
$S_2.code$
...

(d)  $S_1; S_2$



## 7.5.3 布尔表达式的控制流翻译

- 属性 (继承属性)
  - $B.true$ ,  $B$ 为真时控制到达的位置;
  - $B.false$ ,  $B$ 为假时控制到达的位置。
- $a < b$ 
  - if  $a < b$  goto  $B.true$
  - goto  $B.false$
- $B \rightarrow B_1 \text{ or } B_2$ 
  - 如果 $B_1$ 为真, 则立即可知 $B$ 为真, 即 $B_1.true$ 与 $B.true$ 相同;
  - 如果 $B_1$ 为假, 则必须计算 $B_2$ 的值, 令 $B_1.false$ 为 $B_2$ 的开始
  - $B_2$ 的真假出口分别与 $B$ 的真假出口相同



# 简单布尔表达式的翻译示例

——例7.9  $a < b$  or  $c < d$  and  $e < f$

**if  $a < b$  goto Ltrue**

**goto  $L_1$**

**$L_1$ :if  $c < d$  goto  $L_2$**

**goto Lfalse**

**$L_2$ :if  $e < f$  goto Ltrue**

**goto Lfalse**



## 7.5.3 布尔表达式的控制流翻译

**$B \rightarrow B_1 \text{ or } B_2$**      $\{B_1.\text{true} := B.\text{true}; B_1.\text{false} := \text{newlabel};$

$B_2.\text{true} = B.\text{true}; B_2.\text{false} := B.\text{false};$

$B.\text{code} := B_1.\text{code} || \text{gencode}(B_1.\text{false}':') || B_2.\text{code}\}$

**$B \rightarrow B_1 \text{ and } B_2$**      $\{B_1.\text{true} := \text{newlabel}; B_1.\text{false} := B.\text{false};$

$B_2.\text{true} = B.\text{true}; B_2.\text{false} := B.\text{false};$

$B.\text{code} := B_1.\text{code} || \text{gencode}(B_1.\text{true}':') || B_2.\text{code}\}$

**$B \rightarrow \text{not } B_1$**      $\{B_1.\text{true} := B.\text{false}; B_1.\text{false} := B.\text{true};$

$B.\text{code} := B_1.\text{code}\}$



## 7.5.3 布尔表达式的控制流翻译

---

**$B \rightarrow (B_1)$**  { $B_1.\text{true} := B.\text{true}; B_1.\text{false} := B.\text{false};$   
 $B.\text{code} := B_1.\text{code}$ }

**$B \rightarrow E_1 \text{ relop } E_2$**

{ $B.\text{code} := \text{gencode}(\text{'if' } E_1.\text{addr relop } E_2.\text{addr 'goto' } B.\text{true});$   
 $\parallel \text{gencode('goto' } B.\text{false})$ }

**$B \rightarrow \text{true}$**  { $B.\text{code} := \text{gencode}(\text{'goto' } B.\text{true})$ }

**$B \rightarrow \text{false}$**  { $B.\text{code} := \text{gencode}(\text{'goto' } B.\text{false})$ }



## 例7.10: 翻译下列语句

---

**while a < b do**

**$B_1$**

**if c < d then**

**$B_2$**

**$S_3$**  {  **$S_1$**  **x := y+z**

**else**

**$S_2$**  **x := y-z**



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

生  
成  
的  
三  
地  
址  
代  
码  
序  
列

**L<sub>1</sub>: if a < b goto L<sub>2</sub>**

**goto L<sub>next</sub>**

**B<sub>1</sub>.code**

**L<sub>2</sub>: if c < d goto L<sub>3</sub>**

**goto L<sub>4</sub>**

**B<sub>2</sub>.code**

**L<sub>3</sub>: t<sub>1</sub> := y + z**

**x := t<sub>1</sub>**

**S<sub>1</sub>.code**

**goto L<sub>1</sub>**

**S<sub>3</sub>.code**

**L<sub>4</sub>: t<sub>2</sub> := y-z**

**x := t<sub>2</sub>**

**S<sub>2</sub>.code**

**goto L<sub>1</sub>**

**L<sub>next</sub>:**



## 7.5.4 混合模式的布尔表达式翻译

**$E \rightarrow E_1 \text{ relop } E_2 \mid E_1 + E_2 \mid E_1 \text{ and } E_2 \mid \text{id}$**

- $E_1 + E_2$ 、 $\text{id}$  产生算术结果
- $E_1 \text{ relop } E_2$ 、 $E_1 \text{ and } E_2$  产生布尔值
- $E_1 \text{ and } E_2$ :  $E_1$  和  $E_2$  必须都是布尔型的
- 引入语义属性
  - $E.\text{type}$ : arith 或者 bool
  - $E.\text{true}$ ,  $E.\text{false}$
  - $E.\text{addr}$



## 7.5.4 混合模式的布尔表达式翻译

---

**$E \rightarrow E_1 \text{ relop } E_2$**

**$\{E.\text{code} := E_1.\text{code} || E_2.\text{code} ||$**

**$\text{gencode( 'if' } E_1.\text{addr relop } E_2.\text{addr 'goto' } E.\text{true}) ||$**

**$\text{gencode('goto' } E.\text{false}) \}$**



## 7.5.4混合模式的布尔表达式翻译

**$E \rightarrow E_1 + E_2$**     {E.type:=arith;

if  $E_1.type = \text{arith}$  and  $E_2.type = \text{arith}$  then begin

**$E.addr := \text{newtemp}; E.code := E_1.code \parallel E_2.code$**

**gencode( $E.addr := E_1.addr + E_2.addr$ ) end**

else if  $E_1.type = \text{arith}$  and  $E_2.type = \text{bool}$  then begin

**$E.addr := \text{newtemp}; E_2.true := \text{newlabel}; E_2.false := \text{newlabel};$**

**$E.code := E_1.code \parallel E_2.code$**  gencode( $E_2.true := E.addr$   
     **$:= E_1.addr + 1$ )** || gencode('goto' nextstate+1) ||

**gencode( $E_2.false := E.addr := E_1.addr$ )**

**else if .....}**

# 混合模式布尔表达式的翻译示例

——例如:  $4+a>b-c$  and  $d$

$t_1=4+a$

$t_2=b-c$

if  $t_1>t_2$  goto  $L_1$

goto Lfalse

$L_1$ : if  $d$  goto Ltrue

goto Lfalse



## 7.6 回填

---

### ■ 两遍扫描

- 从给定的输入构造出一棵语法树;
- 对语法树按深度优先进行遍历,在遍历过程中执行语法制导定义中给出的翻译动作
- 效率较低



## 7.6 回填

---

### ■ 一遍扫描

#### ■ 问题：生成跳转语句时可能不知道要转向指令的标号

- 先产生暂时没有填写目标标号的转移指令
- 对于每一条这样的指令作适当的记录，建一个链表
- 一旦确定了目标标号,再将它“回填”到相应的指令中
  - E.truelist
  - E.falselist



## 7.6 回填

翻译模式用到如下三个函数：

1. **makelist(i)**: 创建一个只包含*i*的新表，*i*是四元式数组的一个索引(下标)，或者说*i*是四元式代码序列的一个标号。
2. **merge(p1, p2)**: 合并由指针*p1*和*p2*指向的两个表并且返回一个指向合并后的表的指针。
3. **backpatch(p, i)**: 把*i*作为目标标号回填到*p*所指向的表中的每一个转移指令中去。

此处的“表”都是为“回填”所拉的链



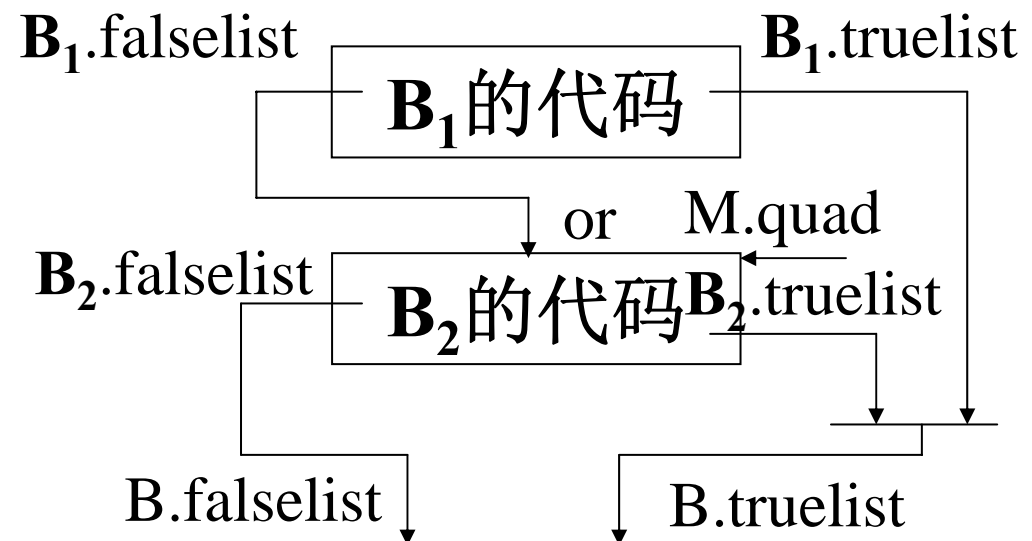
## 7.6.1 布尔表达式的回填式翻译

$B \rightarrow B_1 \text{ or } M B_2$

```
{backpatch(B1.falselist, M.quad);  
  B.truelist:=merge(B1.truelist, B2.truelist);  
  B.falselist := B2.falselist}
```

$M \rightarrow \varepsilon$

```
{M.quad:=nextquad}
```



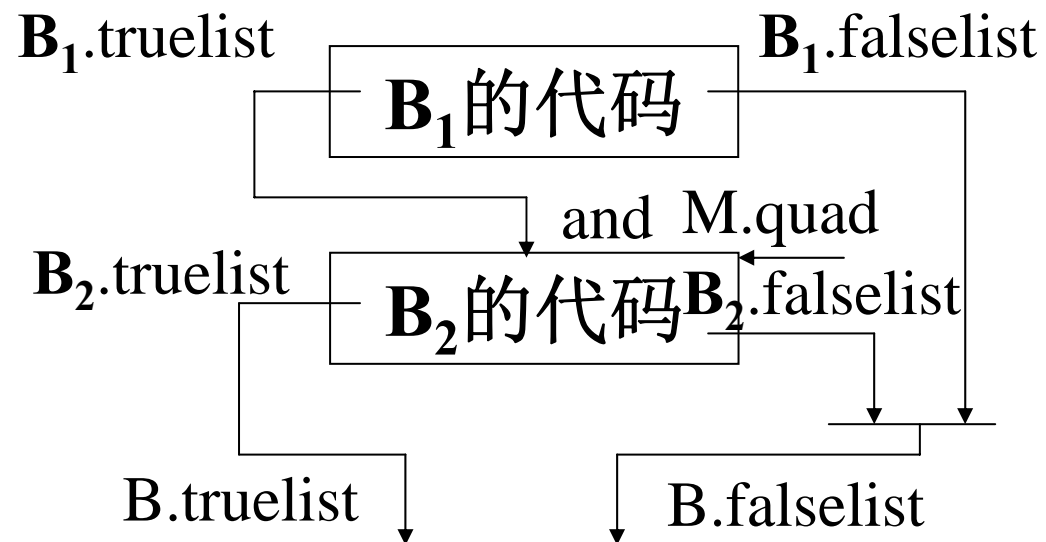
## 7.6.1 布尔表达式的回填式翻译

$B \rightarrow B_1 \text{ and } M B_2$

{backpatch( $B_1$ .truelist,  $M$ .quad);

$B$ .truelist :=  $B_2$ .truelist;

$B$ .falselist:=merge( $B_1$ .falselist,  $B_2$ .falselist);}





## 7.6.1 布尔表达式的回填式翻译

---

**$B \rightarrow \text{not } B_1$**

**$\{B.\text{truelist} := B_1.\text{falselist};$   
 $B.\text{falselist} := B_1.\text{truelist};\}$**

**$B \rightarrow (B_1)$**

**$\{B.\text{truelist} := B_1.\text{truelist};$   
 $B.\text{falselist} := B_1.\text{falselist};\}$**



## 7.6.1 布尔表达式的回填式翻译

---

**$B \rightarrow E_1 \text{ relop } E_2$**

```
{ B.truelist:=makelist(nextquad);  
  B.falselist:=makelist(nextquad+1);  
  gencode('if'E1.addr relop E2.addr 'goto-');  
  gencode('goto-') }
```



## 7.6.1 布尔表达式的回填式翻译

---

**$B \rightarrow \text{true}$**

```
{B.truelist := makelist(nextquad);  
  gencode('goto-')}
```

**$B \rightarrow \text{false}$**

```
{B.falselist := makelist(nextquad);  
  gencode('goto-')}
```

# 例7.11

100: if a<b goto -  
 101: goto 102  
 102: if c<d goto 104  
 103: goto -  
 104: if e<f goto -  
 105: goto -

B.t代表B.truelist  
 B.f代表B.falselist  
 M.q代表M.quad

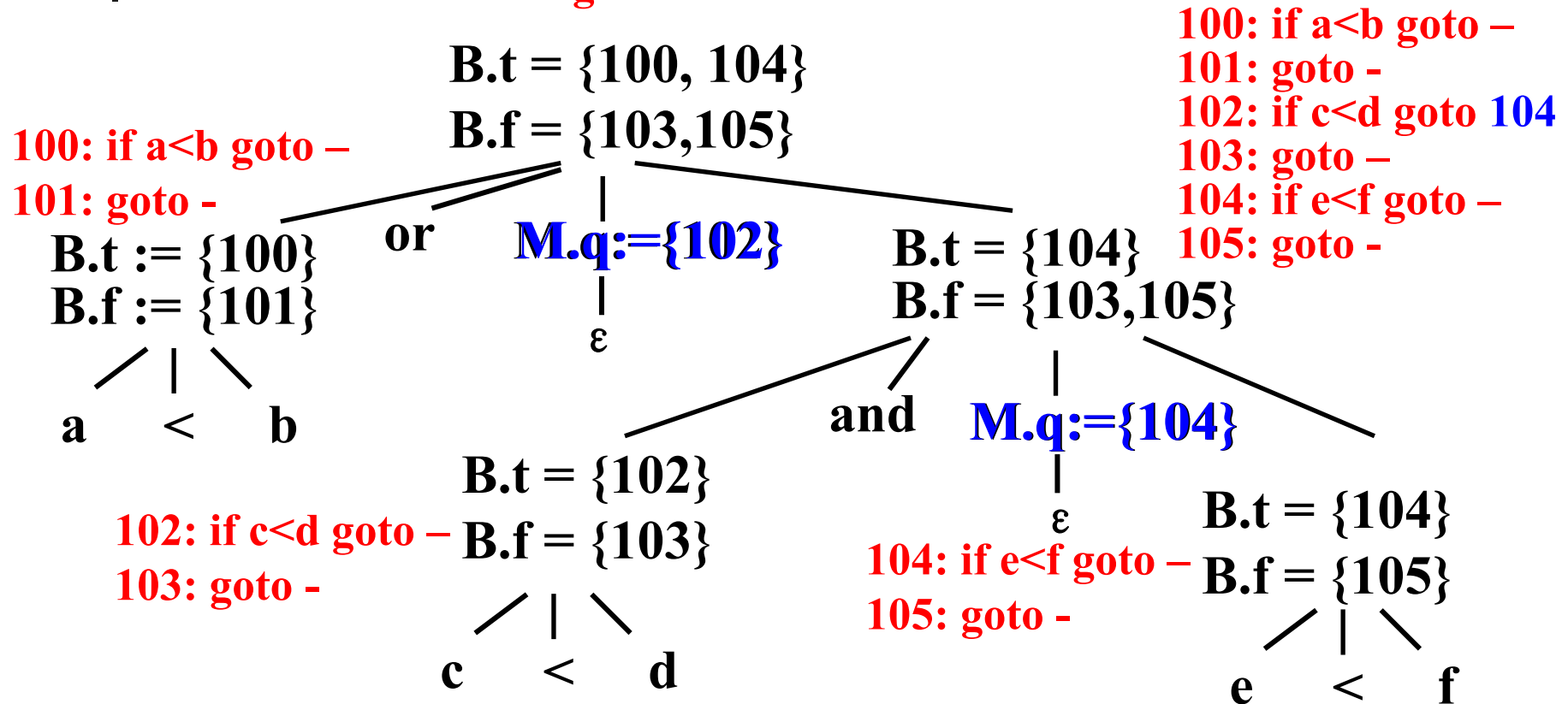


图7.27 a<b or c<d and e<f的注释分析树

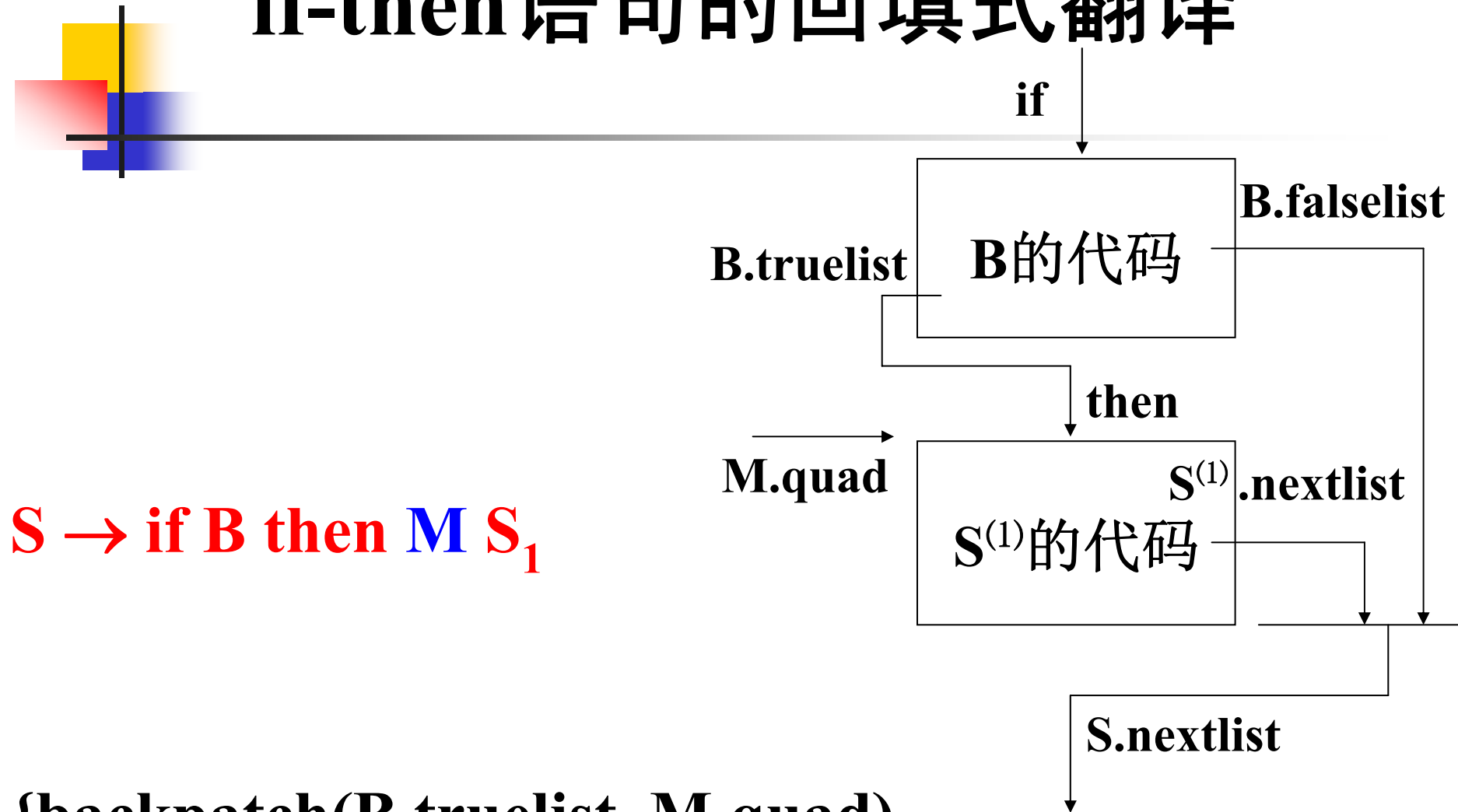
## 7.6.2 常见控制结构的回填式翻译

$S \rightarrow$  if B then  $M$   $S_1$   
| if B then  $M_1$   $S_1$   $N$  else  $M_2$   $S_2$   
| while  $M_1$  B do  $M_2$   $S_1$   
|  $S_1;M$   $S_2$

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

$N \rightarrow \epsilon$  { $N.\text{nextlist} := \text{makelist}(\text{nextquad});$   
 $\text{gencode}(\text{'goto -})$ }

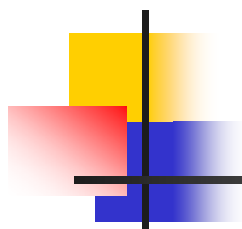
# if-then语句的回填式翻译



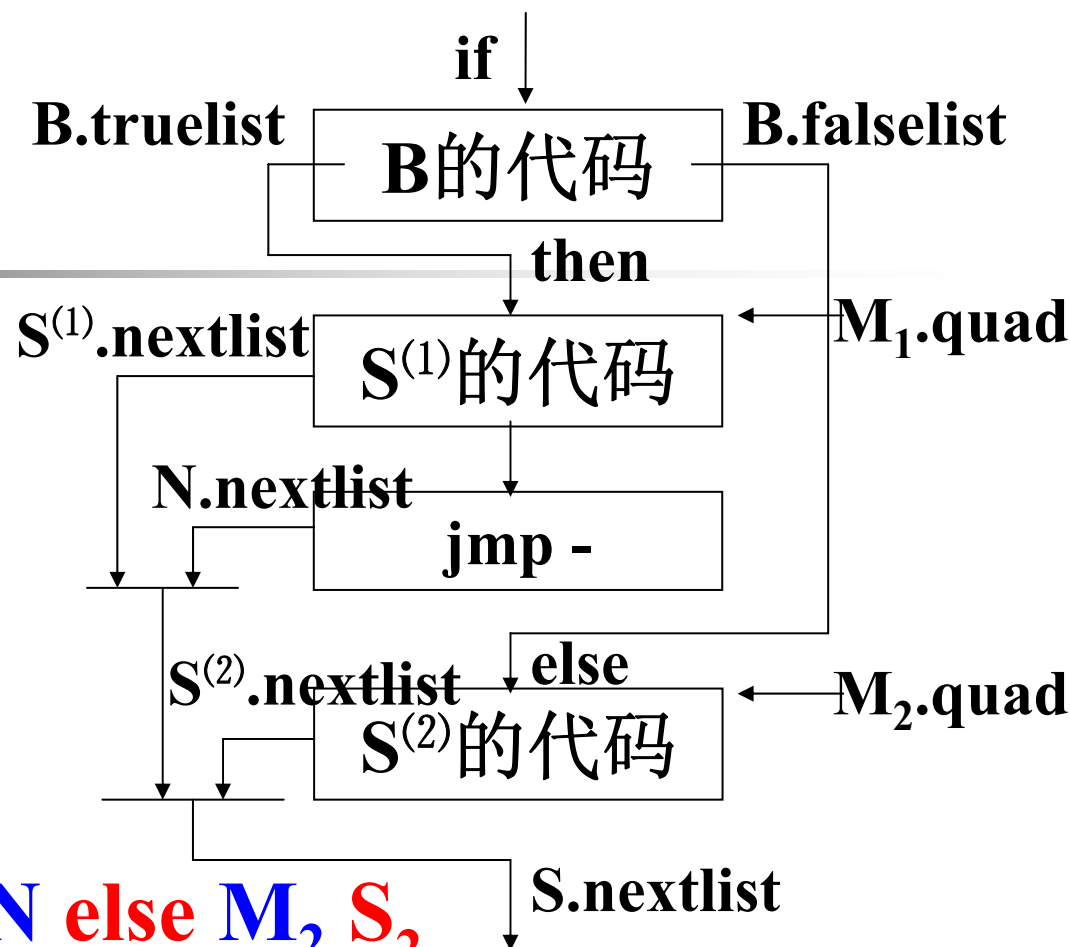
**S**  $\rightarrow$  **if** **B** **then** **M** **S<sub>1</sub>**

**{backpatch(B.truelist, M.quad)**  
**S.nextlist:=merge(B.falselist, S<sub>1</sub>.nextlist)}**





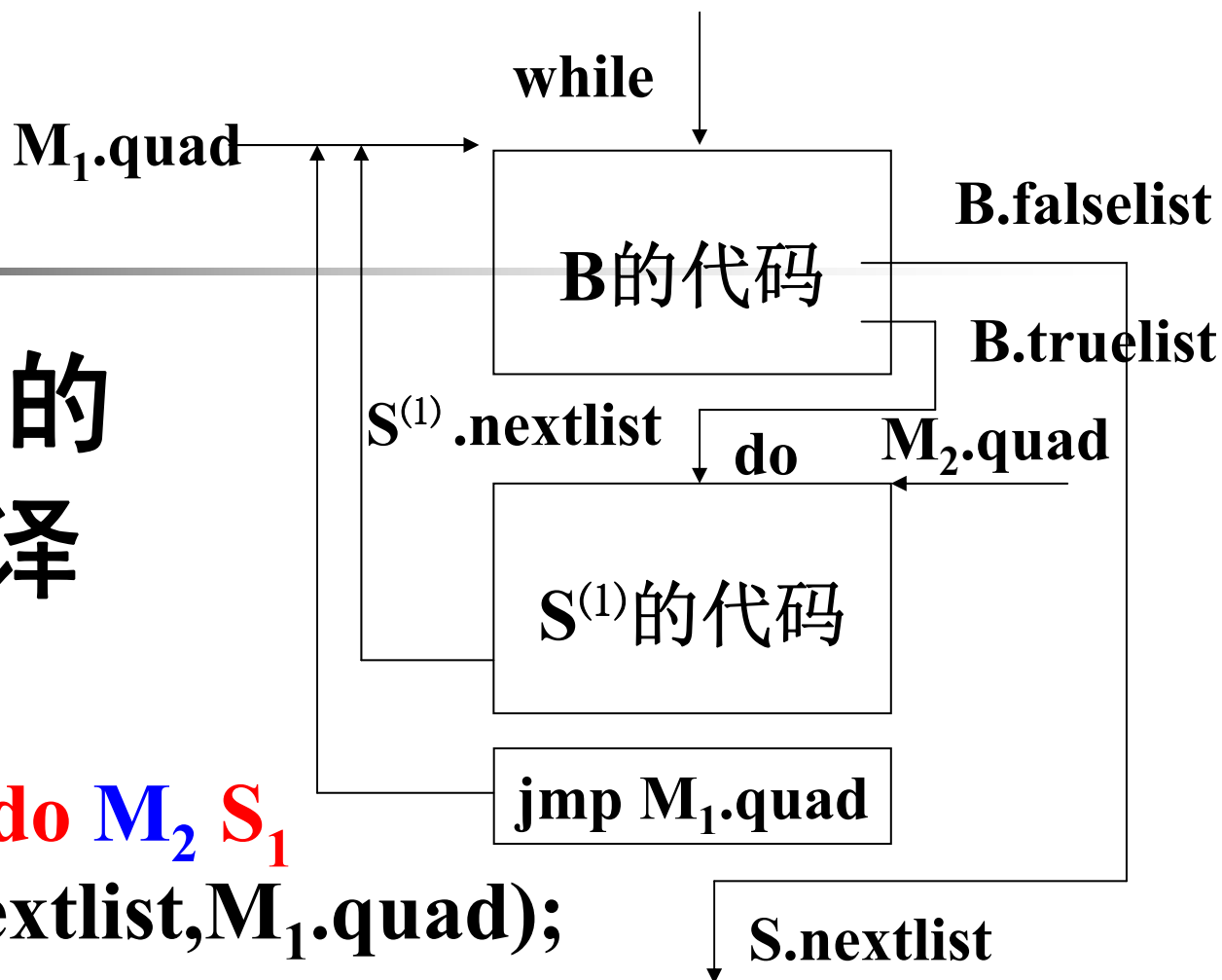
# if-then-else语句 的回填式翻译



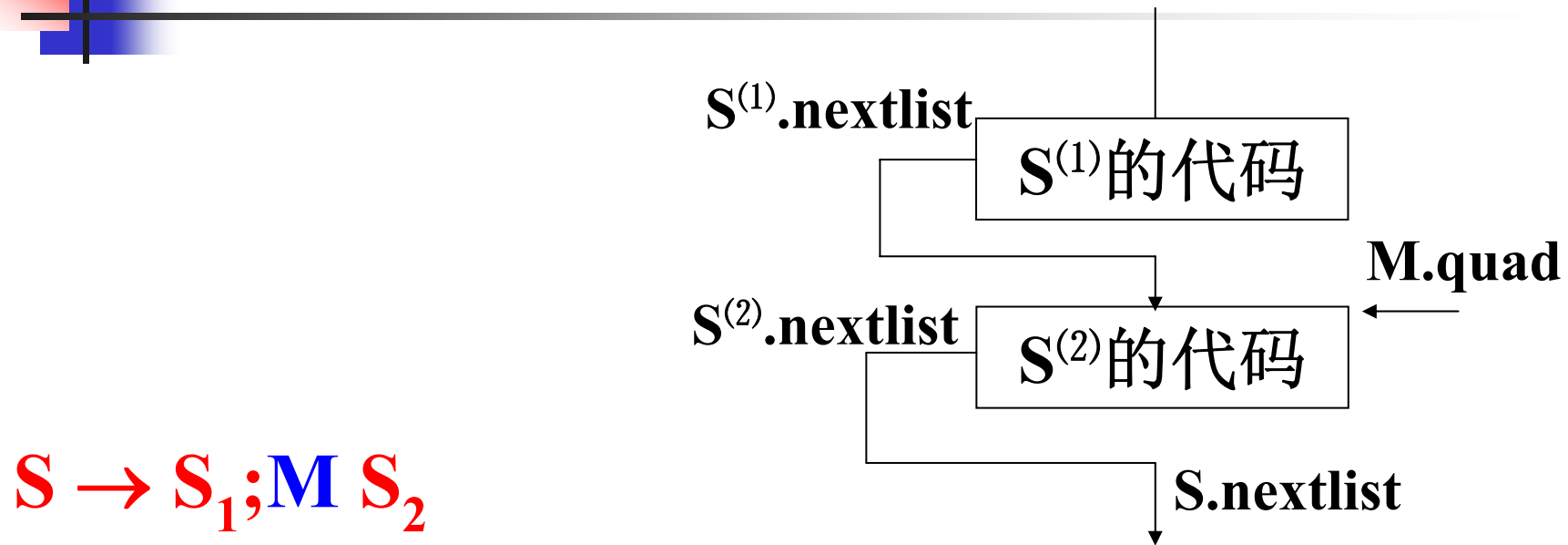
$S \rightarrow \text{if } B \text{ then } M_1 S_1 N \text{ else } M_2 S_2$   
 {backpatch(B.truelist,  $M_1.\text{quad}$ );  
 backpatch(B.falselist,  $M_2.\text{quad}$ );  
 S.nextlist:=  
 merge( $S_2.\text{nextlist}$ , merge( $N.\text{nextlist}$ ,  $S_1.\text{nextlist}$ ))}

# while语句的回填式翻译

$S \rightarrow \text{while } M_1 \text{ } B \text{ do } M_2 \text{ } S_1$   
 {backpatch( $S_1$ .nextlist,  $M_1$ .quad);  
 backpatch( $B$ .truelist,  $M_2$ .quad);  
 $S$ .nextlist :=  $B$ .falselist;  
 gencode('goto'  $M_1$ .quad)}



# 语句序列的回填式翻译



$S \rightarrow S_1; M S_2$

```
{backpatch( $S_1.nextlist$ ,  $M.quad$ );  
  $S.nextlist := S_2.nextlist$ }
```



## 例7.12 翻译下列语句

---

**while  $a < b$  do**

**$B_1$**

**if  $c < 5$  then**

**$B_2$**

**$S_3$**  {

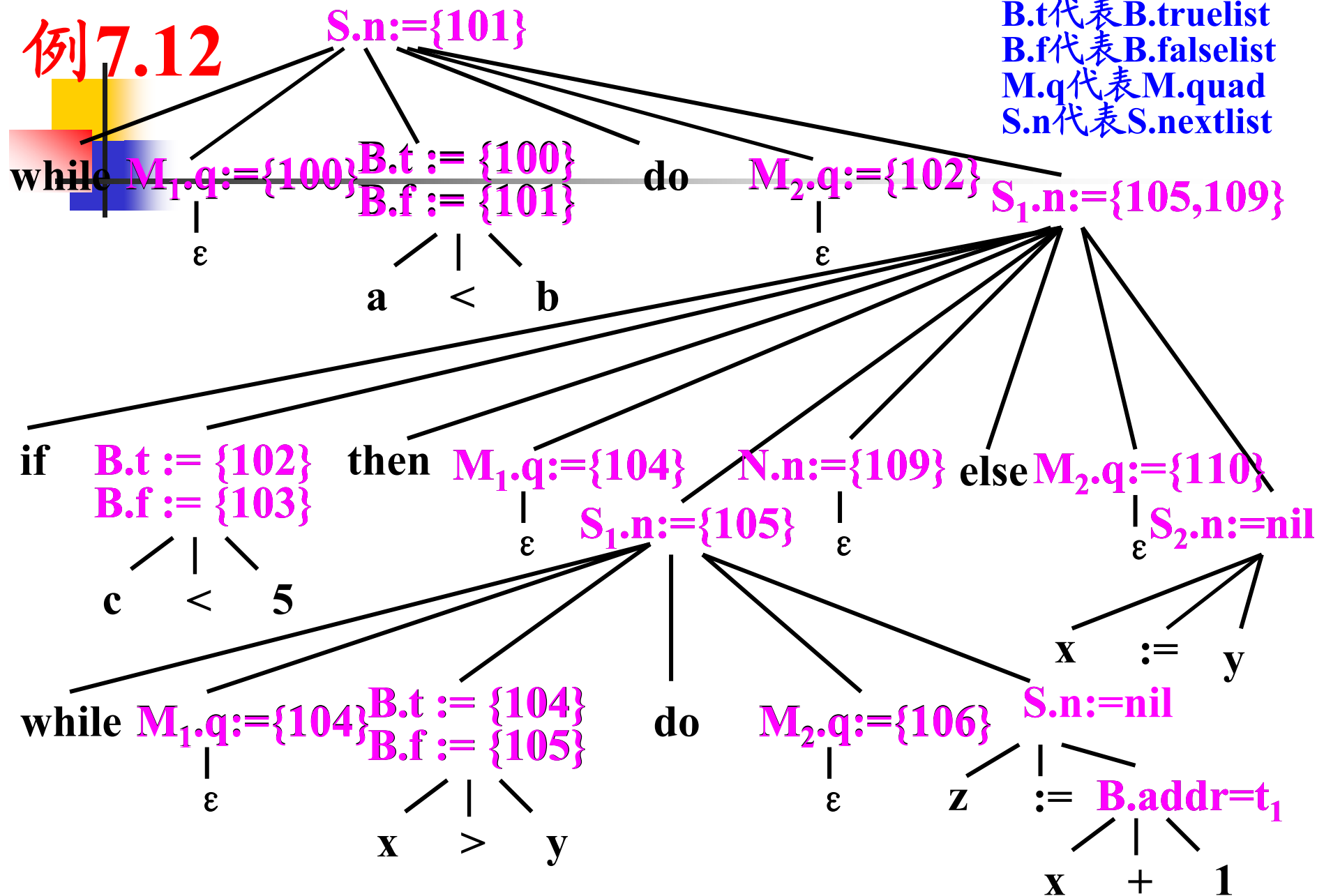
**$S_1$  while  $x > y$  do  $z := x + 1$ ;**

**else**

**$S_2$   $x := y$**

# 例7.12

B.t代表B.truelist  
B.f代表B.falselist  
M.q代表M.quad  
S.n代表S.nextlist



2012-4-26

**while a<b do if c<5 then while x>y do z:=x+1 else x:=y**的注释分析树<sup>581</sup>

**while a<b do if c<5 then while x>y do z:=x+1 else x:=y**

生  
成  
的  
四  
元  
式  
序  
列

100: ( j<,a,b,102 )

101: ( j,-,-,112 )

102: ( j<,c,5,104 )

103: ( j,-,-,110 )

104: ( j>,x,y,106 )

105: ( j,-,-,100 )

106: ( +,x,1,t<sub>1</sub> )

107: ( :=, t<sub>1</sub>,-,z )

108: ( j,-,-,104 )

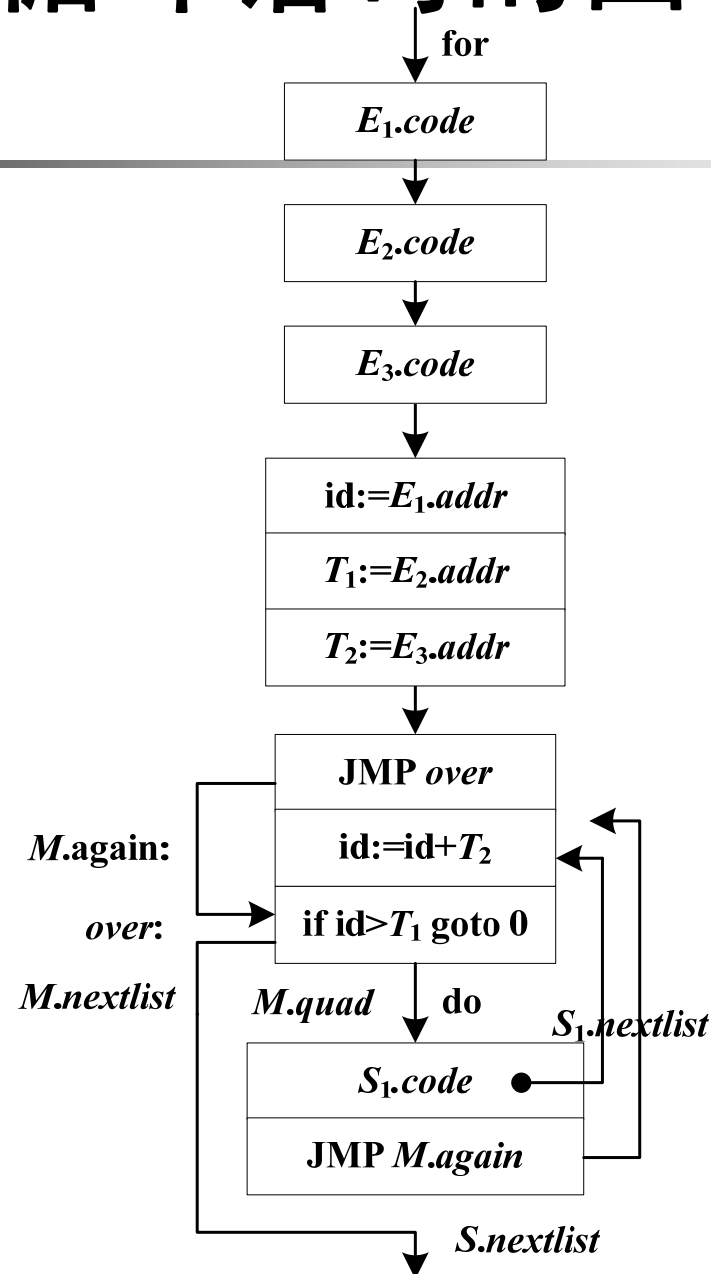
109: ( j,-,-,100 )

110: ( :=,y,-,x )

111: ( j,-,-,100 )

112:

## 7.6.3 for循环语句的回填式翻译



for 循环语句  
的目标结构



## 7.6.3 for循环语句的回填式翻译

---

$S \rightarrow \text{for id} := E_1 \text{ to } E_2 \text{ step } E_3 \text{ do } M S_1$   
 $\{ \text{backpatch}(S_1.\text{nextlist}, M.\text{again},);$   
 $\text{encode}(\text{'goto'}, -, -, M.\text{again});$   
 $S.\text{nextlist} := M.\text{nextlist}; \}$





## 7.6.3 for循环语句的回填式翻译

---

$M \rightarrow \varepsilon$

$\{M.addr := entry(id); gencode(':=', E_1.addr, -, M.addr);$

$T_1 := newtemp; gencode(':=', E_2.addr, -, T_1);$

$T_2 := newtemp; gencode(':=', E_3.addr, -, T_2);$

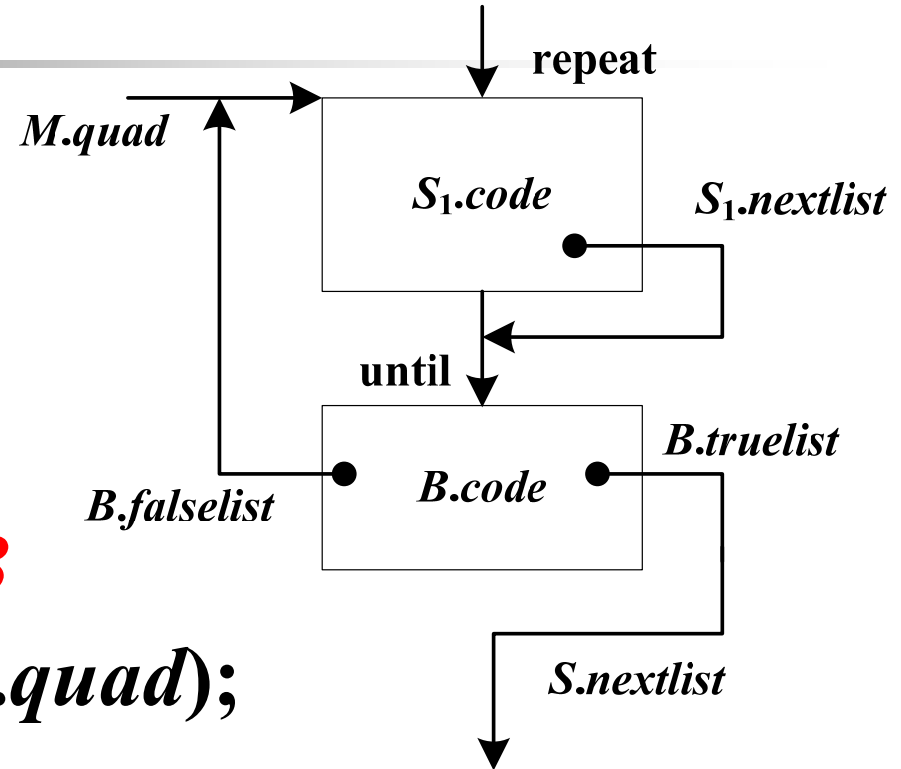
$q := nextquad; gencode('goto', -, -, q+2);$

$M.again := q+1; gencode('+', M.addr, T_2, M.addr);$

$M.nextlist := nextquad;$

$gencode('if' M.addr '>' T_1 'goto -'); \}$

## 7.6.4 repeat语句的回填式翻译



**$S \rightarrow \text{repeat } M \ S_1 \text{ until } N \ B$**

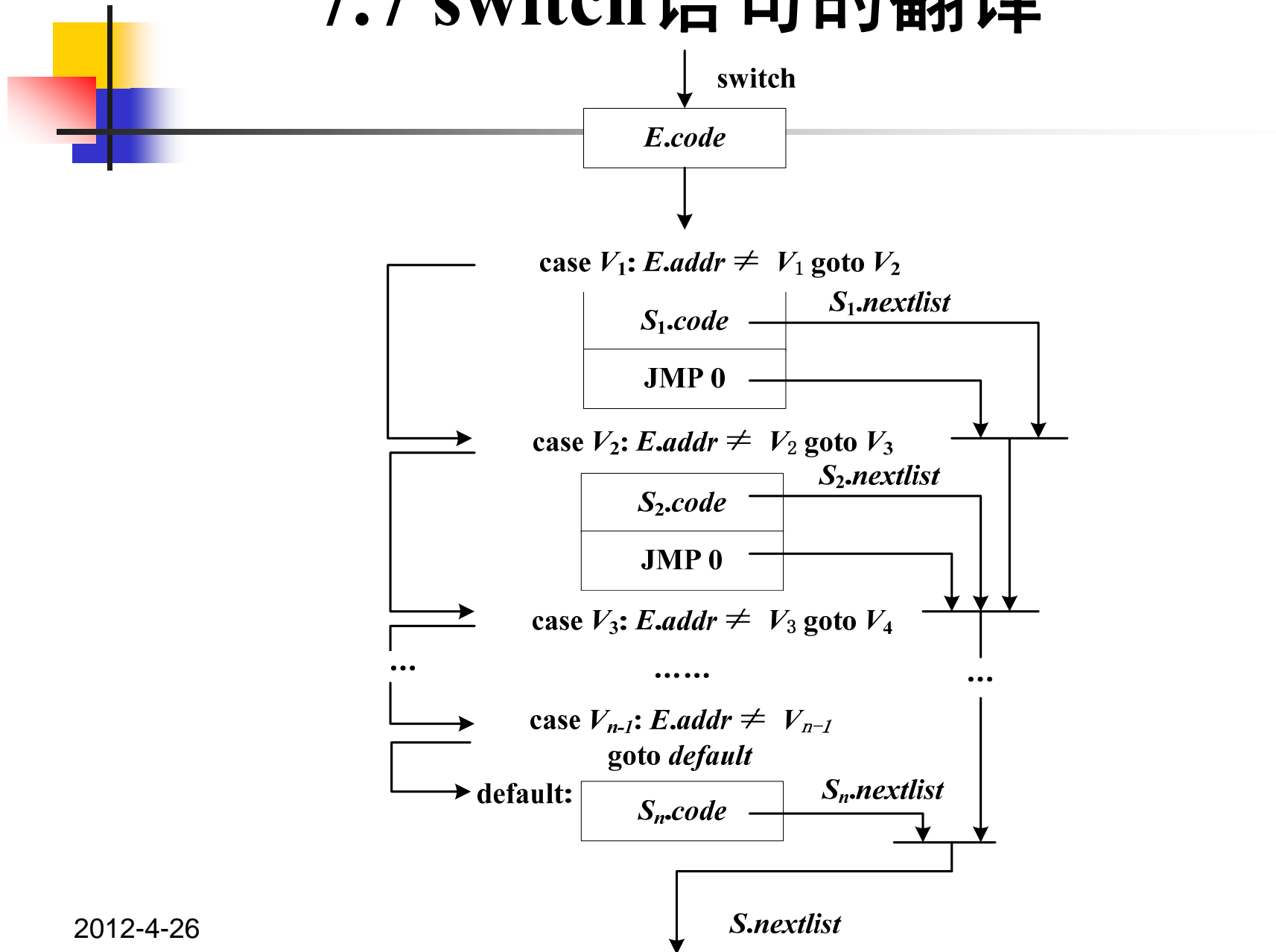
**$\{ \text{backpatch}(B.\text{falselist}, M.\text{quad});$**

**$S.\text{nextlist} := B.\text{truelist} \}$**

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

**$N \rightarrow \epsilon \ \{ \text{backpatch}(S_1.\text{nextlist}, \text{nextquad}) \}$**

## 7.7 switch语句的翻译





## 7.7.2 switch语句的语法制导翻译

---

$S \rightarrow \text{switch } (E) \{ i := 0; S_i.\text{nextlist} := 0; \text{push}$   
     $S_i.\text{nextlist}; \text{push } E.\text{addr};$   
     $\text{push } i; q := 0; \text{push } q \}$   
 $\text{Clist} \{ \text{pop } q; \text{pop } i; \text{pop } E.\text{addr}; \text{pop}$   
     $S_i.\text{nextlist}; S.\text{nextlist} := \text{merge}(S_i.\text{nextlist}, q);$   
     $\text{push } S.\text{nextlist} \}$

## 7.7.2 switch语句的语法制导翻译

*Clist* → **case** *V* : { pop *q*; pop *i*; *i* := *i* + 1; pop *E.addr*;  
if *nextquad* ≠ 0 then *backpatch*(*q*, *nextquad*);  
*q* := *nextquad*;  
*gencode*('if' *E.addr* '≠' *V<sub>i</sub>* 'goto' *Li*);  
push *E.addr*; push *i*;  
push *q*} **S** { pop *q*; pop *i*; pop *E.addr*; pop *S<sub>i-1</sub>.nextlist*;  
*p* := *nextquad*; *gencode*('goto -'); *gencode*(*L<sub>i</sub>* ':');  
*S<sub>i</sub>.nextlist* := *merge*(*S<sub>i</sub>.nextlist*, *p*);  
*S<sub>i</sub>.nextlist* := *merge*(*S<sub>i</sub>.nextlist*, *S<sub>i-1</sub>.nextlist*);  
push *S<sub>i</sub>.nextlist*; push *E.addr*; push *i*; push *q*} *Clist*

## 7.7.2 switch语句的语法制导翻译

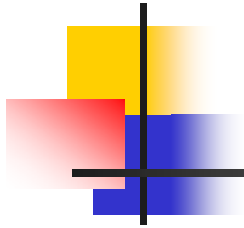
*Cl*ist  $\rightarrow$  default : { pop  $q$ ; pop  $i$ ;  $i := i + 1$ ; pop  $E.addr$ ;  
if  $nextquad \neq 0$  then  $backpatch(q, nextquad)$ ;  
 $q := nextquad$ ;  
 $gencode('if' E.addr \neq V_i 'goto' V_{i+1})$ ;  
push  $E.addr$ ; push  $i$ ;  
push  $q$  }  $S$  { pop  $q$ ; pop  $i$ ; pop  $E.addr$ ; pop  $S_{i-1}.nextlist$ ;  
 $p := nextquad$ ;  
 $gencode('goto -')$ ;  $gencode(L_i ':')$ ;  
 $S_i.nextlist := merge(S_i.nextlist, p)$ ;  
 $S_i.nextlist := merge(S_i.nextlist, S_{i-1}.nextlist)$ ;  
push  $S_i.nextlist$ ; push  $E.addr$ ; push  $i$ ; push  $q$  }



## 例7.14 翻译如下的switch语句

---

```
switch  $E$ 
begin
  case  $V_1: S_1$ 
  case  $V_2: S_2$ 
  ...
  case  $V_{n-1}: S_{n-1}$ 
  default:  $S_n$ 
end
```



E的代码

if E.addr  $\neq$   $V_1$  goto  $L_1$

$S_1$ 的代码

goto S.nextlist

$L_1$ :

if E.addr  $\neq$   $V_2$  goto  $L_2$

$S_2$ 的代码

goto S.nextlist

$L_2$ :

...

...

$L_{n-2}$ :

if E.addr  $\neq$   $V_{n-1}$  goto  $L_{n-1}$

$S_{n-1}$ 的代码

goto S.nextlist

$L_{n-1}$ :

$S_n$ 的代码

S.nextlist :





## 7.8 过程调用和返回语句的翻译

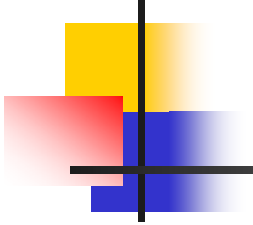
---

**$S \rightarrow \text{call id (Elist)}$**

**$\text{Elist} \rightarrow \text{Elist}, E$**

**$\text{Elist} \rightarrow E$**

**$S \rightarrow \text{return } E$**



# 过程调用 $\text{id}(E_1, E_2, \dots, E_n)$ 的中间代码结构

---

$E_1.\text{addr} := E_1$ 的代码

$E_2.\text{addr} := E_2$ 的代码

...

$E_n.\text{addr} := E_n$ 的代码

param  $E_1.\text{addr}$

param  $E_2.\text{addr}$

...

param  $E_n.\text{addr}$

call id.addr, n



**S → call id (Elist)**

{ n := 0;  
repeat

n := n + 1; 从queue的队首取出一个实参地址p;

gencode('param', -, -, p);

until queue为空;

gencode('call', id.addr, n, -)}

**Elist → Elist, E**

{将E.addr添加到queue的队尾 }

**Elist → E**

{初始化queue, 然后将E.addr加入到queue的队尾 }

**S → return E**{if 需要返回结果 then

gencode(':=', E.addr, -, F);

gencode('ret', -, -, -)}



## 7.9 输入输出语句的翻译

---

$P \rightarrow \text{prog id } (Parlist) M D ; S$

$Parlist \rightarrow \text{input}(\varepsilon \mid, \text{output})$

$S \rightarrow (\text{read} \mid \text{readln}) (N List); \{n:=0;$

repeat

$\text{move}(\text{Queue}, i_n);$

$\text{gencode}(\text{'par'}, \text{'in'}, -, -);$

$n:=n+1;$

until Queue为空;

$\text{gencode}(\text{'call'}, \text{'SYSIN'}, n, -); \}$

$List \rightarrow \text{id}, L(\varepsilon \mid List)$

## 7.9 输入输出语句的翻译

$S \rightarrow (write | writeln) (Elist); \{ n:=0;$

*repeat*

*move*(Queue,  $i_n$ );

*gencode*('par', 'out', -, -);

$n:=n+1$ ;

*until* Queue为空;

*gencode*('call', 'SYSOUT',  $n$ , 'w')}

/\* $n$ 为输出参数个数,  $w$ 是输出操作类型\*/

$EList \rightarrow E, K ( \varepsilon | EList)$

$M \rightarrow \varepsilon \{ gencode('prog', id, y, -) \}$

/\* $y$ 的值表示input,output或两者皆有\*/

$N \rightarrow \varepsilon \{ \text{设置一个语义队列Queue} \}$

$L \rightarrow \varepsilon \{ T:=entry(id); add(Queue, T) \}$

$K \rightarrow \varepsilon \{ T:=E.addr; add(Queue, T) \}$



# 本章小结

- 典型的中间代码有逆波兰表示、三地址码、图表示;
- 声明语句的主要作用是为程序中用到的变量或常量名指定类型。类型可以具有一定的层次结构, 可以用类型表达式来表示;
- 声明语句的翻译工作就是将名字的类型及相对地址等有关信息填加到符号表中;
- 赋值语句的语义是将赋值号右边算术表达式的值保存到左边的变量中, 其翻译主要是算术表达式的翻译。要实现数组元素的寻址, 需要计算该元素在数组中的相对位移;



# 本章小结

---

- 类型检查在于解决运算中类型的匹配问题，根据一定的规则来实现类型的转换；
- 控制语句的翻译中既可以通过根据条件表达式改变控制流程，也可以通过计算条件表达式的逻辑值的方式实现条件转移的控制；
- 回填技术是解决单遍扫描的语义分析中转移目标并不总是有效的问题；



# 本章小结

---

- **switch**语句的翻译通过对分支的实现来完成;
- 过程调用与返回语句的翻译主要在于实现参数的有效传递和相应的存储管理;
- **I/O**语句要求输出参数都是有效的。





*School of Computer Science & Technology*  
*Harbin Institute of Technology*



# 第8章 符号表管理

**重点：** 符号表的作用, 符号表的组织结构, 符号表与作用域。

**难点：** 符号表的组织结构及其性能评价。





# 第8章 符号表管理

---

8.1 符号表的作用

8.2 符号表中存放的信息

8.3 符号表的组织结构

8.4 符号表与作用域

8.5 本章小结



## 8.1 符号表的作用

---

- 编译的各个阶段都有可能会用到符号表中登记的信息
  - 协助进行语义检查(如检查一个名字的引用和之前的声明是否相符)和中间代码生成
  - 在目标代码生成阶段, 当需要为名字分配地址时, 符号表中的信息将是地址分配的主要依据
  - 编译器用符号表来记录、收集和查找出现在源程序中的各种名字及其语义信息。



## 8.1 符号表的作用

---

- 符号表是以名字为关键字来记录其信息的数据结构，其上支持的两个最基本操作应该是填加表项和查找表项，这两个操作必须是高效的

## 8.2 符号表中存放的信息

记录源程序中出现的各种名字及其属性信息是符号表的首要任务。

- 显然同一个名字在一段程序中应该表示同一个对象，即同一个符号表中不能出现相同的名字，因此名字可以作为符号表的关键字。于是，每一个符号表表项中需要存放的基本信息就是符号的名字及其属性。

	名字	属性
符号表表项1	abc	...
符号表表项2	i	...
⋮	...	...
符号表表项 $n$	...	...

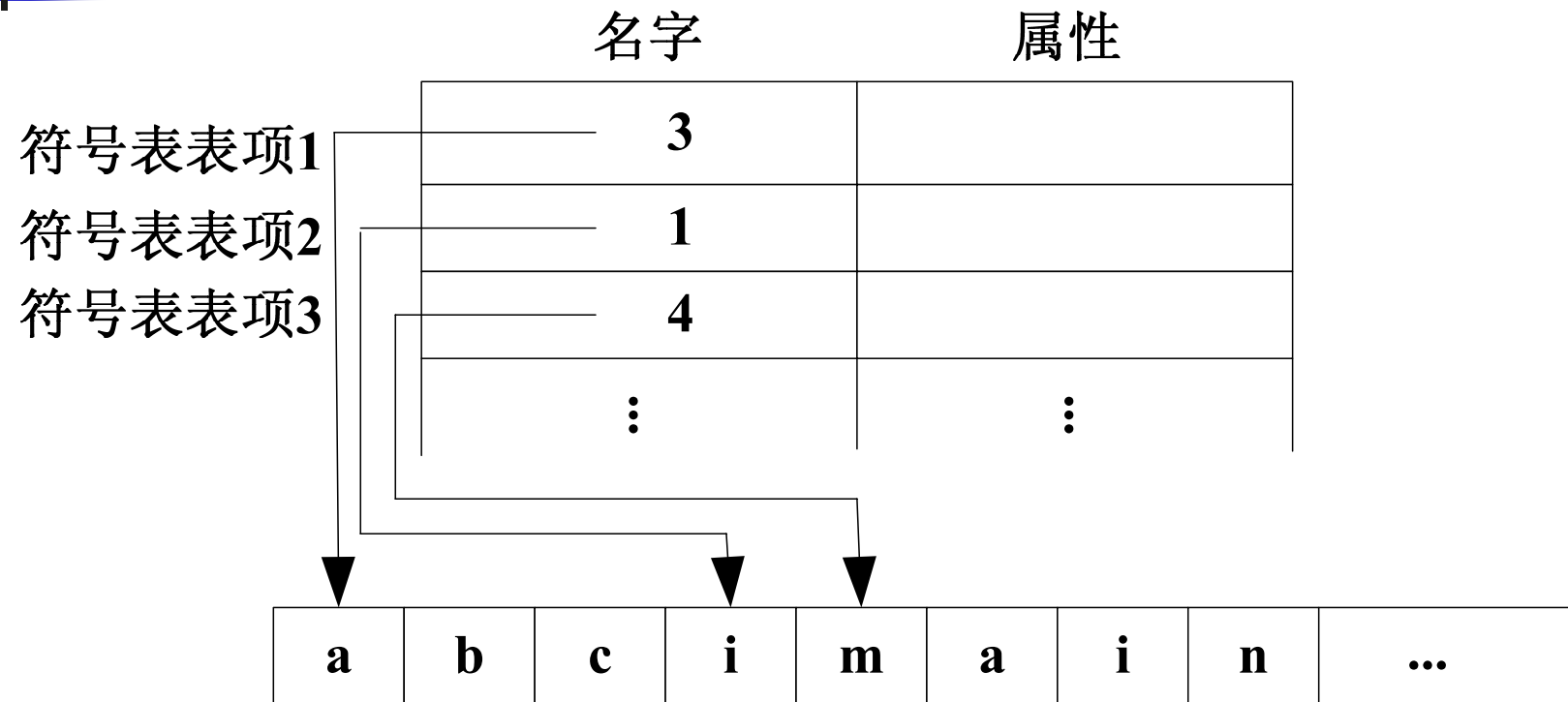


## 8.1.1 符号表中的名字

---

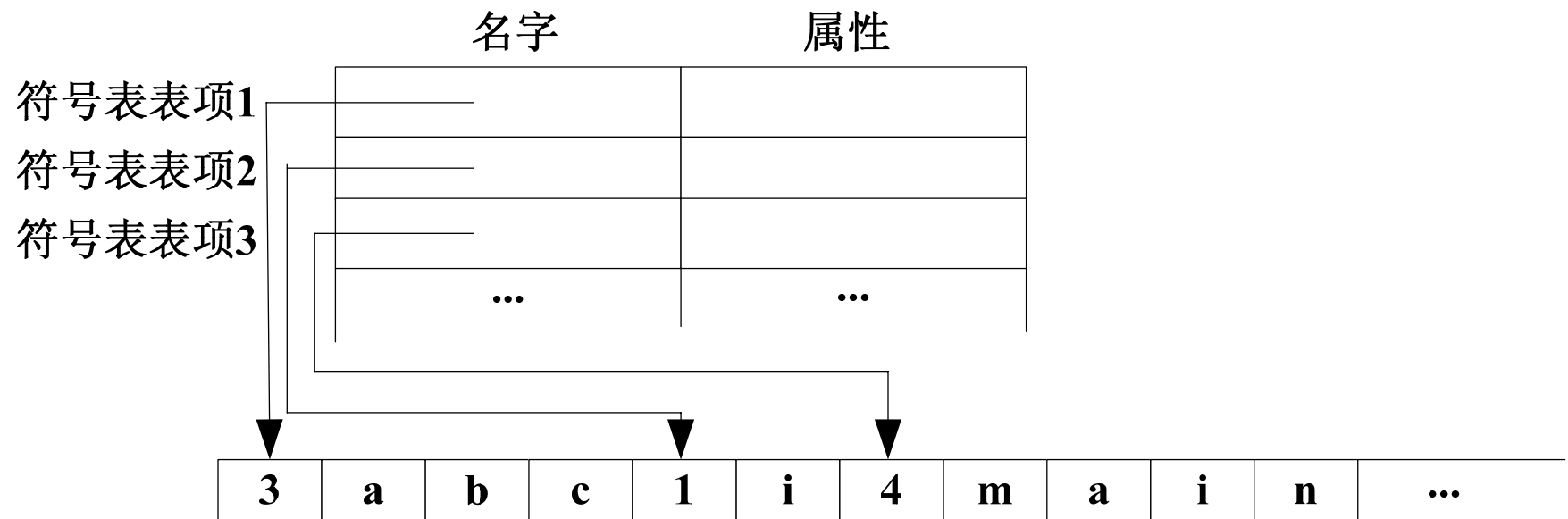
- 名字字段长度固定
  - 名字项的长度大小取决于标识符允许的最大长度
  - 不适于标识符长度变化范围较大的语言
  - 空间浪费
- 名字字段长度可变
  - 标识符的长度没有限制
  - 符号表上的操作复杂而低效
- 引入一个单独的字符串表，将符号表中的全部标识符集中放在这个字符串表中，而在符号表表项的名字部分只要给出相应标识符的首字符在字符串表中的位置即可

## 8.1.1 符号表中的名字



(a) 标识符长度放在符号表中

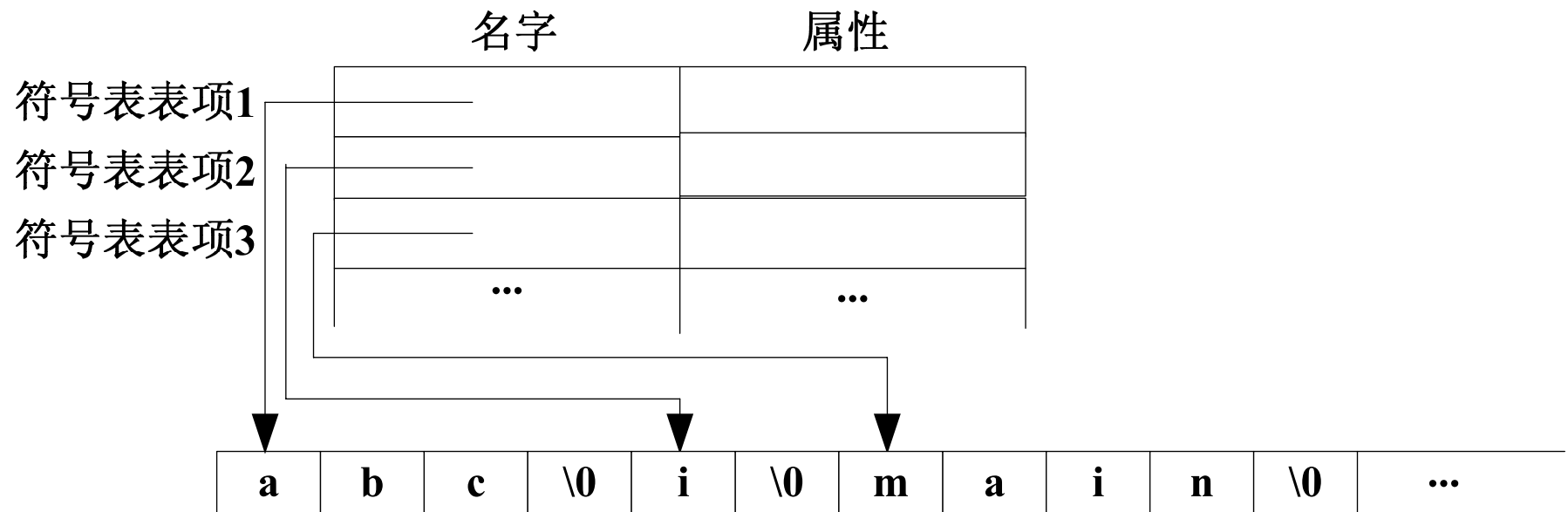
## 8.1.1 符号表中的名字



(b) 标识符长度放在字符串中



## 8.1.1 符号表中的名字



(c) 用'\0'表示标识符的结束



## 8.1.2 符号表中的属性

---

- 符号所表达的含义不同，符号表中需要存放的属性也就不同
  - 数组名字需要存放的属性信息应该包括数组的维数、各维的维长等
  - 函数(或过程)的名字应该存放其参数个数、各参数的类型、返回值的类型等



## 8.1.2 符号表中的属性

---

- 建立多个符号表来管理源程序中出现的各种符号，如常数表、变量表、函数表、数组表等
  - 可能出现不同种类符号出现重名的问题
- 建立一张共用的大表来管理各种符号，此时需要在符号表中增设一个标志来表明符号的种属
  - 不同种类符号所需存放属性信息在数量上的差异将会造成符号表的空间浪费

## 8.1.2 符号表中的属性

	名字	基本属性			扩展属性
	符号种类	类型	地址	扩展属性指针	
符号表表项1	abc	变量	int	0	NULL
符号表表项2	i	变量	int	4	NULL
符号表表项3	myarray	数组	int	8	<div><div>维数</div><div>各维维长</div><div><div>2</div><div>3</div><div>4</div></div></div>
	...	...			

图8.3 多种符号共用符号表的一种实现结构

## 8.1.2 符号表中的属性

	名字	基本属性			扩展属性指针
		符号种类	类型	地址	
符号表表项1	swap	函数	int		
符号表表项2	a	形参	int *		
符号表表项3	b	形参	int *		NULL
	...		...		



图8.4 用扩展属性链组织函数形参的符号表



## 8.1.3 符号的地址属性

- 如果采用静态存储分配策略，则符号 $x$ 绑定的地址等于静态分配的基址 $base$ 加上符号 $x$ 的偏移量 $offset$
- 如果采用的是栈式存储分配或堆式存储分配等动态分配策略，则符号是在程序执行过程中和地址动态绑定的。
  - 如栈式存储分配时， $i$ 的地址是以栈指针 $sp$ 为基址加上 $i$ 相对于活动记录起始地址的偏移量 $offset_i$
- 符号表中各符号的地址属性就是该符号相对于第一个符号的偏移地址

## 8.3 符号表的组织结构

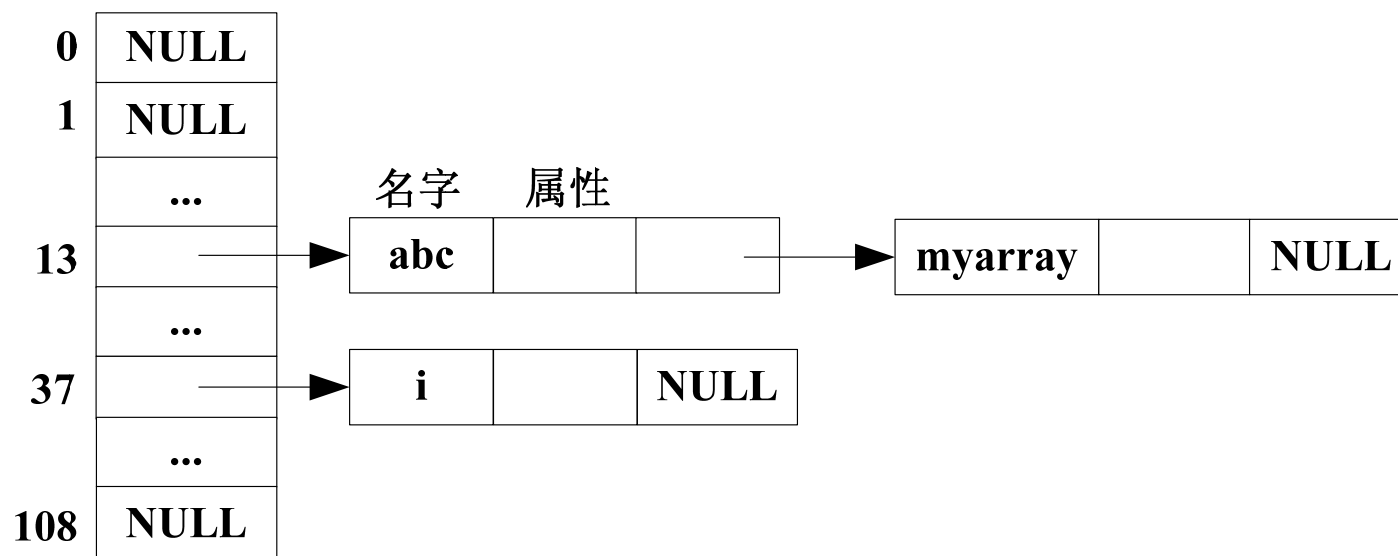
### 8.3.1 符号表的线性表实现

- 用线性表实现符号表较为直观
  - 数组实现：插入 $n$ 个符号、执行 $e$ 次查找操作的时间复杂度为 $T(n, e) = O(n(n+e))$
  - 有序数组实现：插入 $n$ 个符号、执行 $e$ 次查找操作的时间复杂度为 $T(n, e) = e \times \log n + \sum_{i=1}^n (\log i) + \sum_{i=1}^n i$   
 $\leq (n+e) \log n + O(n^2)$
  - 有序符号表结构只有在下面的情况下才能取得较好效果：和插入操作次数相比，符号表表项上的查找操作次数占绝对多数，即 $e \gg n$ 。

## 8.3.2 符号表的散列表实现

- 引入散列表不仅可以提高`lookup`操作的效率，同时也可以提高`insert`操作的效率，所以在许多实际编译器的符号表实现中均采用了散列技术

开散列表表头数组





## 8.3.2 符号表的散列表实现

- 引入散列表不仅可以提高 $lookup$ 操作的效率，同时也可以提高 $insert$ 操作的效率，所以在许多实际编译器的符号表实现中均采用了散列技术
  - 插入 $n$ 个符号，查找 $e$ 个符号的 $lookup$ 操作和 $insert$ 操作的时间复杂度 $T(n,e)$ 还将与 $m$ 有关，记为 $T(n,e,m)$ ， $T(n,e,m) \approx n(n+e)/m$
  - $S(n,m)=O(n)$
  - 散列函数应在满足 $\sum_{j=1}^m b_j = n$ 的前提下，使 $\sum_{j=1}^m b_j \times (b_j + 1)/2$ 达到最小

## 8.4 符号表与作用域

```
int main()
{
    int abc;
    abc = 1;
    {
        int abc;
        abc = 2;
        printf("abc is %d\n", abc);
    }
    printf("abc is %d\n", abc);
}
```

运行结果为:

abc is 2

abc is 1

说明abc在不同的范围内有效。

这个有效范围就是符号的作用域

## 8.4.1 程序块结构的符号表

变量的作用域满足最近嵌套原则

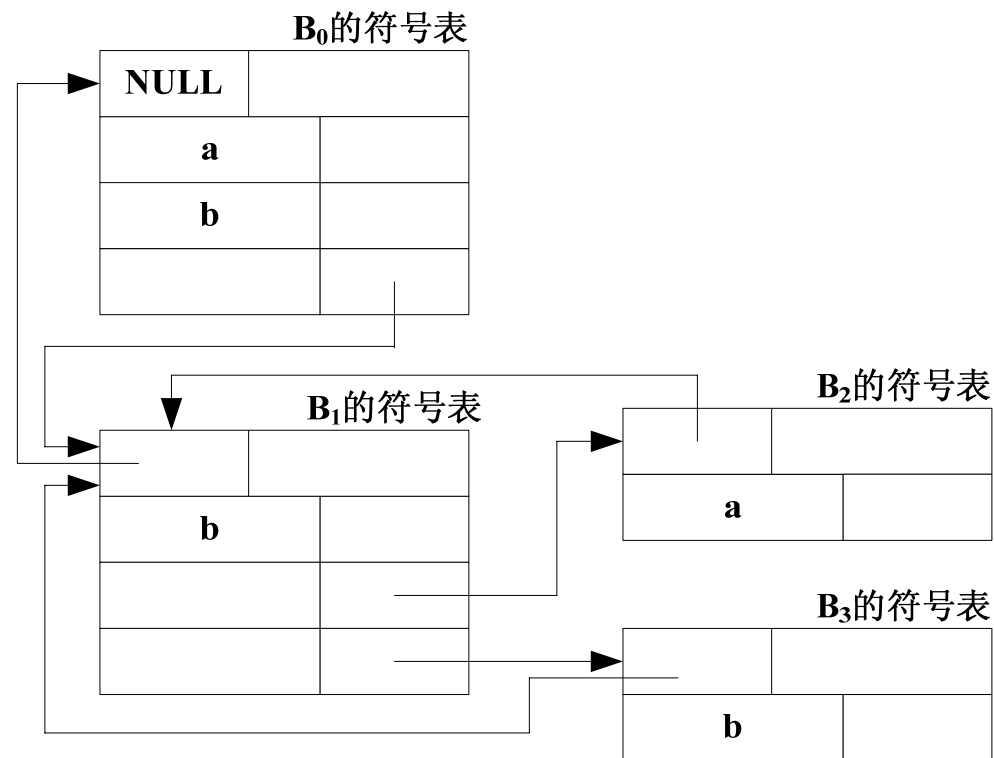
```
(1) int main()
(2) {
(3)     int a=0;
(4)     int b=0;
(5)     {
(6)         int b=1;
(7)         {
(8)             int a=2;
(9)             printf("%d %d\n", a, b );
(10)        }
(11)        {
(12)            int b=3;
(13)            printf("%d %d\n", a, b);
(14)        }
(15)        printf("%d %d\n", a, b);
(16)    }
(17)    printf("%d %d\n", a, b);
(18) }
```

2012-4-26

图8.10 一个C语言程序块实例

## 8.4.1 程序块结构的符号表

为每个程序块建立一个符号表，程序块内的符号记录在该程序块所对应的符号表中，还要建立起这些符号表之间的联系，以刻画出符号的嵌套作用域



## 8.4.2 程序块结构符号表的其他实现

- 将所有块的符号表放在一个大数组中，然后再引入一个程序块表来描述各程序块的符号表在大数组中的位置及其相互关系

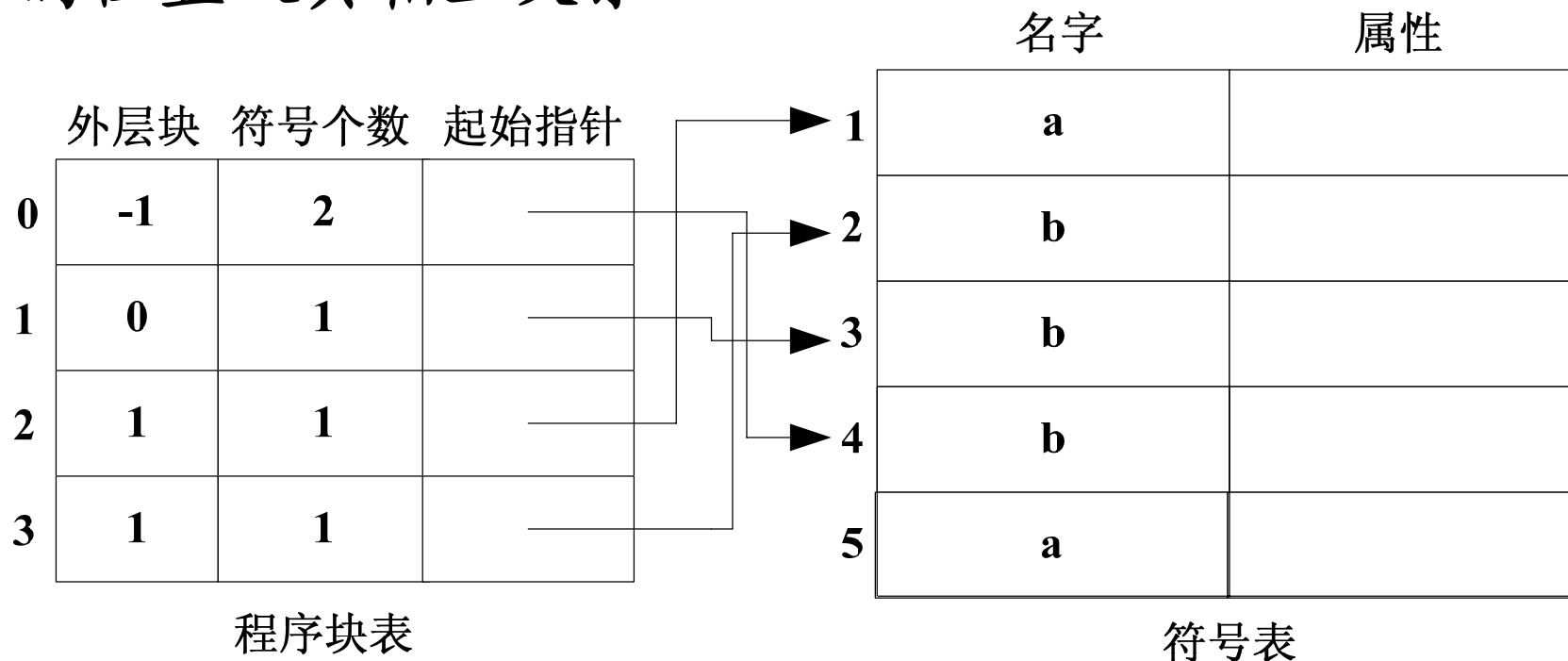


图8.12 图8.10的另一种符号表结构



## 8.4.2 程序块结构符号表的其他实现

- 将符号所属程序块的编号放在符号表表项中。查找某个符号的名字 *name* 时，只有当 *name* 和符号表中的名字字符串完全匹配，且符号表表项中的块编号和当前处理的块编号完全相同时才算查找成功，否则要新建一个名字为 *name* 且块号为当前块编号的符号表表项。
  - 程序块编号可以通过在语法制导定义中的块开始处和块结束处添加适当的语义规则计算得出。

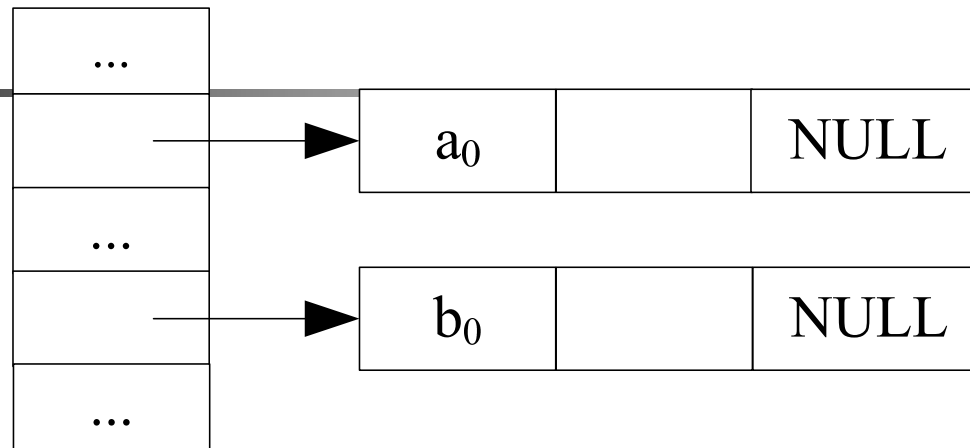


## 8.4.2 程序块结构符号表的其他实现

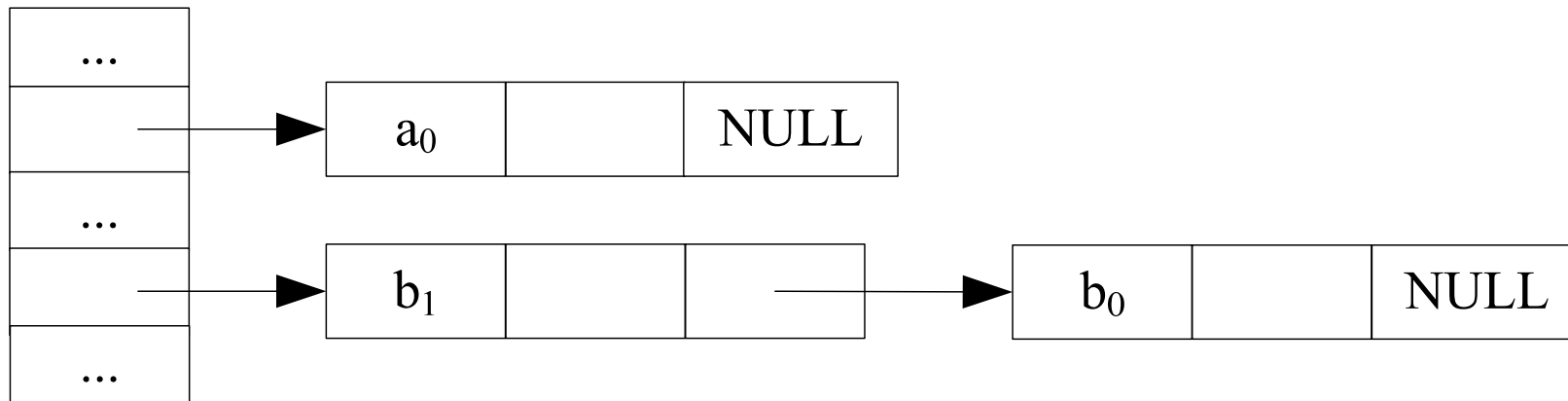
- 程序块满足最近嵌套原则
  - 内层程序块中的局部变量只有全部处理完成之后才进入外层块
  - 一旦进入外层程序块，内层块的局部变量就不会再使用了，可以从符号表中将这些符号删除
  - 符号表中最前面的符号一定是当前正在处理的块中的局部变量
  - 符号表表项中可以不用存放块编号，而是根据符号表表项在符号表中的位置来判断。

## 8.4.2 程序块结构符号表的其他实现

对图8.10  
中的程序



(a) 处理到语句(5)时的符号表

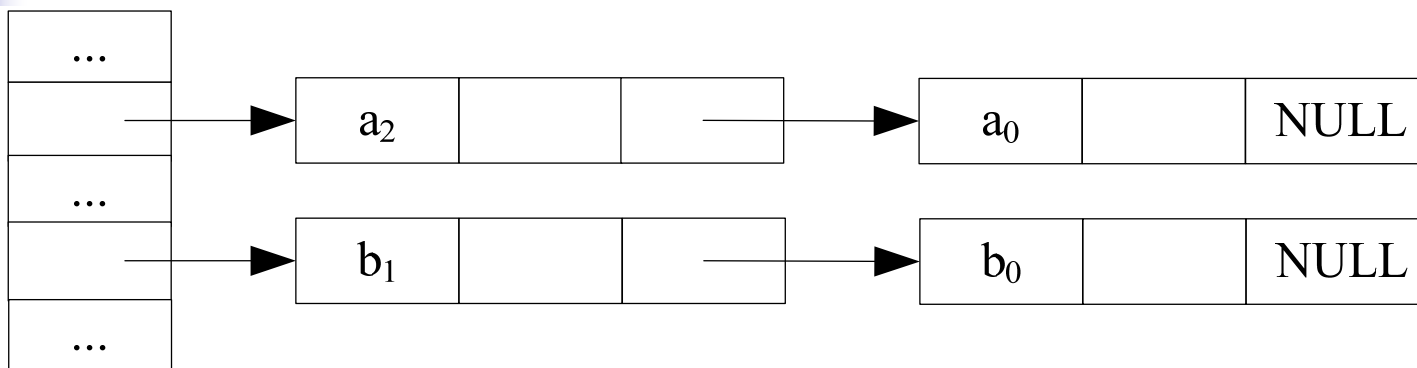


(b) 处理到语句(7)时的符号表

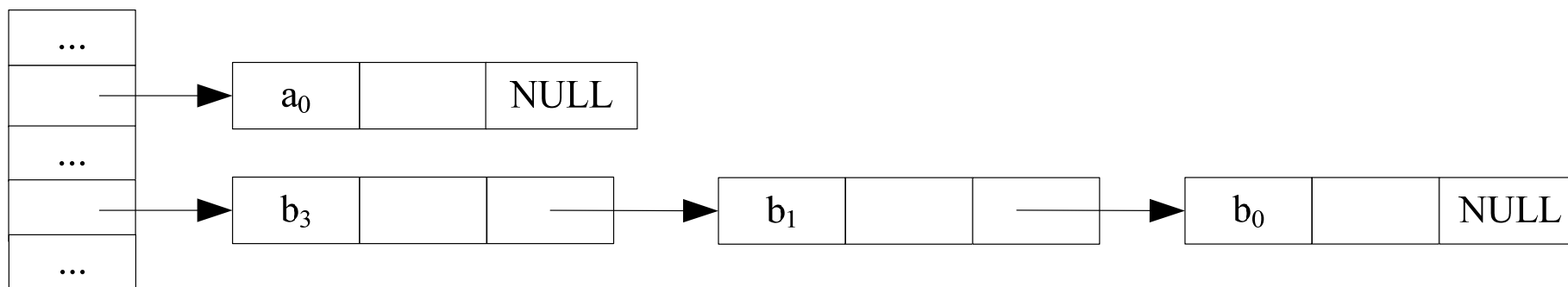


## 8.4.2 程序块结构符号表的其他实现

对图8.10中的程序



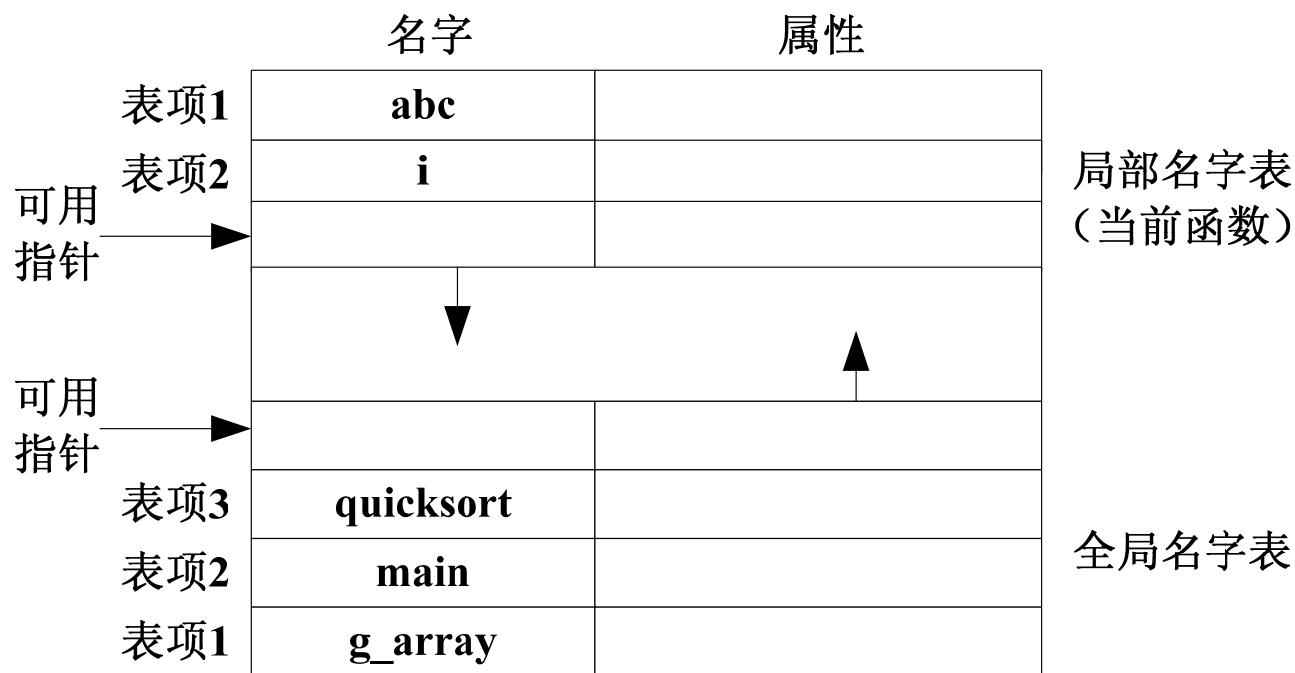
(c) 处理到语句(9)时的符号表



(d) 处理到语句(13)时的符号表

## 8.4.3 C语言的符号表

- 如果采取将每个函数分别编译成目标代码然后连接装配成一个可执行程序的处理方式，则每个函数中的符号经一遍处理即可，而且源程序中的多个函数是一个接一个处理的，不会出现交叉，此时可以按图8.16的方式组织符号表。





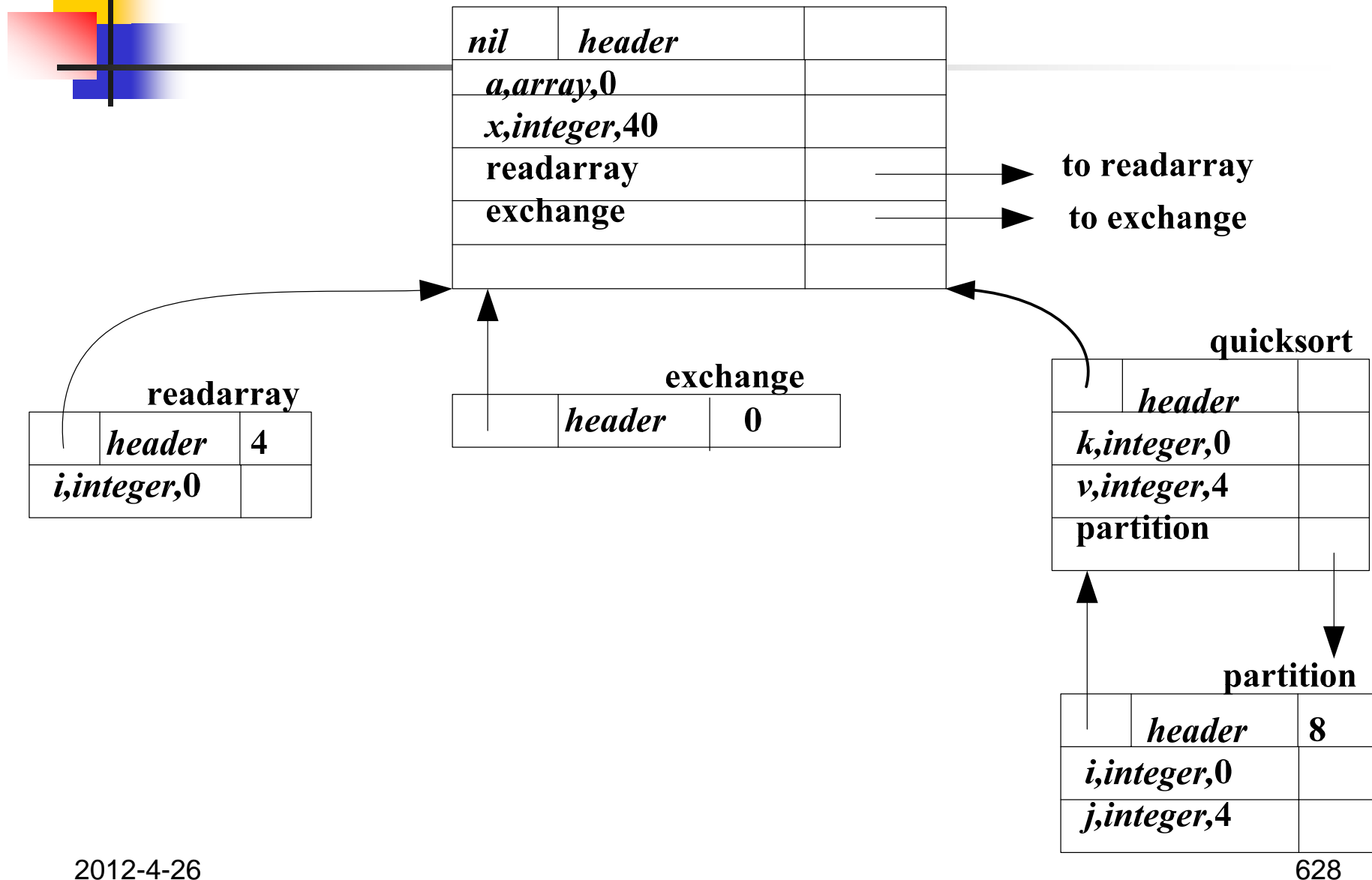
## 8.4.4 嵌套过程的符号表

---

**Pascal**等允许在过程中嵌套定义其它过程

```
program sort(input, output);  
  procedure readarray;  
    begin ... end{ readarray };  
  procedure exchange( i, j : integer );  
    begin ... end{ exchange };  
  procedure quicksort( m, n : integer );  
    function partition( x, y : integer );  
      begin ... end{ partition };  
    begin ... end{ quicksort };  
begin ... end{ sort };
```

## 8.4.4 嵌套过程的符号表





# 本章小结

- 符号表用来存放编译器各阶段收集来的各种名字的类型和特征等有关信息，并供编译程序用于语法检查、语义检查、生成中间代码及生成目标代码等；
- 源程序中会出现各种各样的名字，如函数名、函数参数名、函数中的局部变量名、全局变量名、数组名、结构名、文件名等，相应的属性可以是种属、类型、地址等。
- 根据符号所需的属性个数和类型的不同，可以组成不同的符号表，也可以组成统一的符号表，在组成统一符号表时，需要采用恰当的组织结构，以便可以对其进行高效处理。



## 本章小结

---

- 随着程序规模的扩大，符号名的数量会很大，因而必须关注符号表的组织和高效管理。无序线性符号表、有序线性符号表、散列表表示符号表具有不同性能的组织形式。
- 符号表管理中必须关注到语言所规定的符号的作用域，特别是在嵌套结构的程序中，符号的作用域是分层的。
- C语言符号表的管理。



# 第9章 运行时的存储组织

**重点：** 符号表的内容、组织，过程调用实现，  
静态存储分配、动态存储分配的基本方法。

**难点：** 参数传递，过程说明语句代码结构，  
过程调用语句的代码结构，  
过程调用语句的语法制导定义，  
栈式存储分配。





# 第9章 运行时的存储组织

---

- 9.1 与存储组织有关的源语言概念与特征
- 9.2 存储组织
- 9.3 静态存储分配
- 9.4 栈式存储分配
- 9.5 栈中非局部数据的访问
- 9.6 堆管理
- 9.7 本章小结



## 9.1 与存储组织有关的源语言概念与特征

- 编译程序必须准确地实现包含在源程序中的各种抽象概念，如名字、作用域、数据类型、操作符、过程、参数和控制流结构等，这些概念反映了源语言所具有的一些特征，对它们的支持往往会影响运行时的存储组织和分配策略
- 给定一个源程序，编译程序必须根据源语言的特征(规定)为源程序中的许多问题做出决策，包括何时、怎样为名字分配内存地址。
  - 静态策略：在编译时即可做出决定的策略
  - 动态策略：直到程序执行时才能做出决定的策略



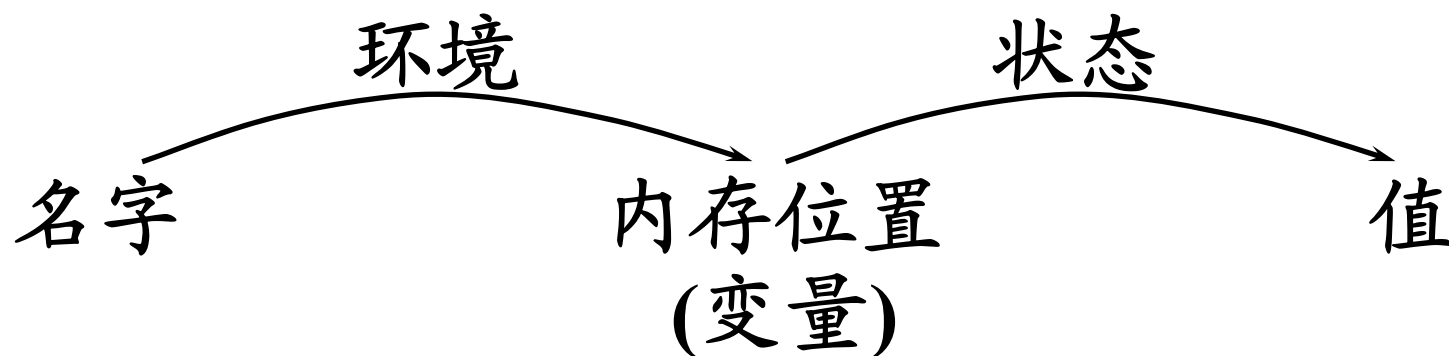
## 9.1.1 名字及其绑定

### ■ “名字”、“变量”和“标识符”的区别与联系

- 名字和变量分别表示编译时的名字和运行时该名字所代表的内存位置。
- 标识符则是一个字符串，用于指示数据对象、过程、类或对象的入口。
- 所有标识符都是名字，但并不是所有的名字都是标识符，名字还可以是表达式。例如，名字`x.y`可能表示`x`所代表结构的域`y`。
- 同一标识符可以被声明多次，但每个声明都引入一个新的变量。即使每个标识符只被声明一次，局部于某个递归过程的标识符在不同的运行时刻也将指向不同的内存位置。

# 名字的绑定

- 从名字到值的两步映射
  - 环境把名字映射到左值，而状态把左值映射到右值
  - 赋值改变状态，但不改变环境。
- 如果环境将名字 $x$ 映射到存储单元 $s$ ，我们就说 $x$ 被绑定到 $s$





## 9.1.2 声明的作用域

- $x$ 的声明的作用域是程序中的这样一段区域，在该区域中， $x$ 的引用均指向 $x$ 的这一声明。对于某种程序设计语言，如果只通过考察其程序就可以确定某个声明的作用域，则称该语言使用静态作用域；否则称该语言使用动态作用域。
- C++、Java和C#等还提供了对作用域的显式控制，其方法是使用public、private和protected这样的关键字。
- 声明的作用域是通过符号表进行管理的，详见8.4节的讨论。



# 1. 静态作用域

■ 在具有程序块结构的语言中，变量声明的静态作用域规则如下：

- 如果名字 $x$ 的声明 $D$ 属于程序块 $B$ ，则 $D$ 的作用域是 $B$ 的所有语句，只有满足如下条件的程序块 $B'$ 除外： $B'$ 嵌套在 $B$ 中(可以是任意的嵌套深度)，且 $B'$ 中具有同一名字 $x$ 的一个新的声明。
- 令 $B_1, B_2, \dots, B_k$ 是包围 $x$ 的本次引用的所有程序块， $B_{k-1}$ 是 $B_k$ 的直接外层程序块， $B_{k-2}$ 是 $B_{k-1}$ 的直接外层程序块，如此类推。找到使 $x$ 的某个声明属于 $B_i$ 的最大 $i$ ，则 $x$ 的本次引用指向 $B_i$ 中的这个声明。换句话说， $x$ 的本次引用处在 $B_i$ 中的这个声明的作用域中。



# 1. 静态作用域

声 明	作用域
<b>int a = 0;</b>	$B_0 - B_2$
<b>int b = 0;</b>	$B_0 - B_1$
<b>int b = 1;</b>	$B_1 - B_3$
<b>int a = 2;</b>	$B_2$
<b>int b = 3;</b>	$B_3$

表9.1 图8.10所示程序中声明的作用域



## 2. 显式访问控制

---

- 类和结构为其成员引入了一种新的作用域
  - 如果 $p$ 是某个带有域(成员) $x$ 的类的对象, 则 $p.x$ 中 $x$ 的引用将指向该类定义中的域 $x$ 。与程序块结构类似的是, 类 $D$ 中成员 $x$ 的声明的作用域将会扩展到 $D$ 的任何子类 $D'$ , 除非 $D'$ 中具有同一名字 $x$ 的一个局部声明。



## 2. 显式访问控制

---

- 通过使用像public、private和protected这样的关键字，C++或Java类的面向对象语言提供了一种对超类中成员名字的显式访问控制。这些关键字通过限制访问来支持封装。因此，私有名的作用域只包含与该类及其友类相关联的方法声明和定义，保护名只对其子类是可访问的，而公用名从类的外部也是可以访问的。





## 3. 动态作用域

---

- 动态作用域规则相对于时间而静态作用域规则相对于空间
  - 静态作用域规则要求我们找出某个引用所指向的声明，条件是该声明处在包围该引用的“空间上最近的”单元(程序块)中。
  - 动态作用域也是要求我们找出某个引用所指向的声明，但条件是该声明处在包围该引用的“时间上最近的”单元(过程活动)中。



## 3. 动态作用域

---

- “动态作用域”通常是指下面的策略
  - 名字 $x$ 的引用指向带有 $x$ 声明的最近被调用的过程中 $x$ 的这个声明。
  - 例如，过程被当做参数进行传递时



## 9.1.3 过程及其活动

- 将“过程、函数和方法”统称为“过程”
- **过程定义**是一个声明，它的最简单形式是把一个标识符和一个语句联系起来。该标识符是**过程名**，而这个语句是**过程体**。
- 当过程名出现在可执行语句中时，称相应的过程在该点被调用。**过程调用**就是执行被调用过程的过程体。注意，过程调用也可以出现在表达式中。



## 9.1.3 过程及其活动

- 出现在过程定义中的某些标识符具有特殊的意义，称为该过程的形式参数，简称为形参。调用过程时，表达式作为实在参数(或实参)传递给被调用的过程，以替换出现在过程体中的对应形式参数。9.1.4节将讨论实参和形参的结合方法。
- 过程体的每次执行叫做该过程的一个活动。过程p的一个活动的生存期是从过程体执行的第一步到最后一步，包括执行被p调用的过程的时间，以及再由这样的过程调用其它过程所花的时间，等等。



## 9.1.3 过程及其活动

- 如果 $a$ 和 $b$ 是过程的活动，那么它们的生存期或者不交迭，或者嵌套。也就是说，如果在 $a$ 结束之前 $b$ 就开始了，那么 $b$ 必须在 $a$ 结束之前结束。
- 如果同一个过程的一次新的活动可以在前一次活动结束前开始，则称这样的过程是递归的。递归过程 $p$ 也可以间接地调用自己。
- 如果某个过程是递归的，则在某一时刻可能有它的几个活动同时活跃，这时必须合理组织这些同时活跃着的活动的内存空间。



## 9.1.4 参数传递方式

---

- 当一个过程调用另一个过程时，它们之间交换信息的方法通常是通过非局部名字和被调用过程的参数来实现的。
  - 传值
  - 传地址
  - 传值结果
  - 传名
- 其主要区别在于实参所代表的究竟是左值、右值还是实参的正文本身

# 1. 传值

- 计算实参并将其右值传递给被调用过程
- 传值方式可以如下实现：
  - 被调用过程为每个形参开辟一个称为形式单元的存储单元，用于存放相应实参的值。
  - 调用过程计算实参，并把右值放入相应的形式单元中。

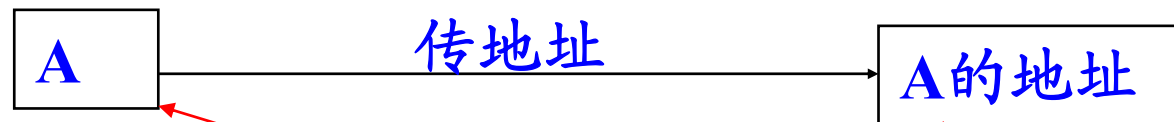


## 2. 传地址

- 调用过程将实参的地址传递给被调用过程
- 传地址方式可以如下实现：
  - 如果实参是一个具有左值的名字或表达式，则传递该左值本身
  - 如果实参是 $a+b$ 或 $2$ 这样的没有左值的表达式，则调用过程首先计算实参的值并将其存入一个新的存储单元，然后将这个新单元的地址传递给被调用过程

实际参数A

形式参数X



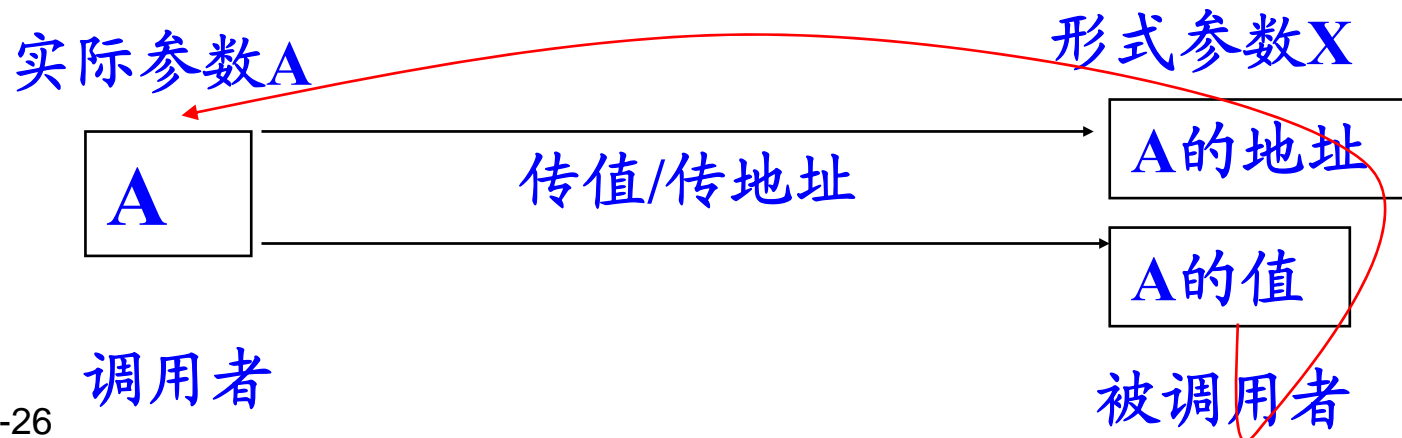
调用者

被调用者间址访问



### 3. 传值结果

- 传值结果就是将传值和传地址这两种方式结合起来
- 传值结果方式可以如下实现：
  - 实参的右值和左值同时传给被调用过程。
  - 在被调用过程中，像传值方式那样使用实参的右值。
  - 当控制返回调用过程时，根据传递来的实参的左值，将形参当前的值复制到实参存储单元。





## 4. 传名

- 用实参表达式对形参进行文字替换。
- 传名方式可以如下实现：
  - 在调用过程中设置计算实参左值或右值的形实转换子程序。
  - 被调用过程为每个形参开辟一个存储单元，用于存放该实参的形实转换子程序的入口地址。被调用过程执行时，每当要向形参赋值或取该形参的值时，就调用相应于该形参的形实转换子程序，以获得相应的实参地址或值。注意，形实转换子程序的运行环境是调用程序。形实转换子程序又称为换名子程序 **thunk**。



# 例

---

```
procedure swap(var x, y: integer);
```

```
  var temp: integer;
```

```
  begin
```

```
    temp := x;
```

```
    x := y;
```

```
    y := temp
```

```
  end
```

调用 swap(i, a[i])

temp := i;

i := a[i];

a[i] := temp

主程序

**A:=2; B:=3;**

**P(A+B, A, A);**

**print A**

子程序

**P(X,Y,Z);**

**{Y:=Y+1;**

**Z:=Z+X}**

传名

**A:=A+1=3**

**A:=A+A+B=3+3+3**

**9**

临时单元:

**T:A+B=5**

传值:


**2**


传值结果:

**X=T=5, Y=Z=A=2**

**Y:=Y+1=3**

**Z:=Z+X=5+2=7**

**Y  A=3**

**Z  A=7**

**7**

传地址:

**X=T=5, Y=Z=A=2**

**A:=A+1=2+1**

**A:=A+T=3+5**

**8**

# 编译程序组织存储空间时必须考虑的问题

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



## 9.2 存储组织

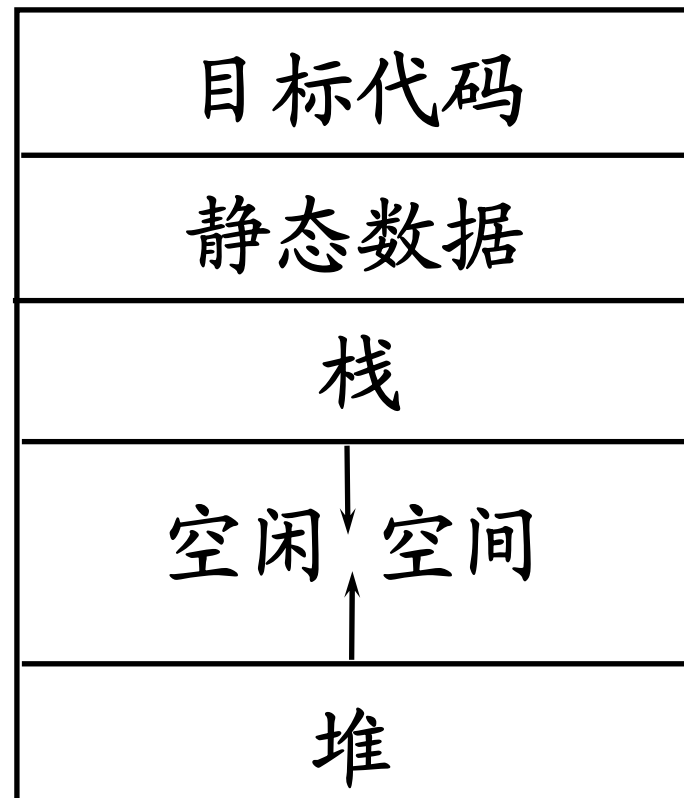
---

- 9.2.1 运行时内存的划分
- 9.2.2 活动记录
- 9.2.3 局部数据的组织
- 9.2.4 全局存储分配策略



## 7.2.1 运行时内存的划分

---





## 9.2.2 活动记录

---

- 过程的每个活动所需要的信息用一块连续的存储区来管理，这块存储区叫做**活动记录**
- 假定语言的特点为:允许过程递归调用、允许过程含有可变数组，过程定义允许/不允许嵌套。
- 采用栈式存储分配机制
- 活动记录：运行时，每当进入一个过程就有一个相应的活动记录压入栈顶。活动记录一般含有连接数据、形式单元、局部变量、局部数组的内情向量和临时工作单元等。



# 每个过程的活动记录内容

——非嵌套语言(如C)

TOP→	临时变量 数组内情向量 简单变量
	形式单元
2	参数个数
1	返回地址
SP→ 0	旧SP

↑  
对任何局部变量X的引用可表示为变址访问:

$dx[SP]$

$dx$ :变量X相对于活动记录起点的地址, 在编译时可确定。

# 每个过程的活动记录内容

——嵌套语言(如Pascal)



- 连接数据
  - 返回地址
  - 动态链：指向调用该过程前的最新活动记录地址的指针。
- 静态链：指向静态直接外层最新活动记录地址的指针，用来访问非局部数据。

# 每个过程的活动记录内容



- 形式单元：存放相应的实在参数的地址或值。
- 局部数据区：局部变量、内情向量、临时工作单元(如存放对表达式求值的结果)。



## 9.2.3 局部数据的组织

---

- 字节是可编址内存的最小单位。
- 变量所需的存储空间可以根据其类型而静态确定。
- 一个过程所声明的局部变量，按这些变量声明时出现的次序，在局部数据域中依次分配空间。
- 局部数据的地址可以用相对于某个位置的地址来表示。
- 数据对象的存储安排还有对齐的问题。
  - 整数必须放在内存中特定的位置，如被2、4、8整除的地址



## 9.2.3 局部数据的组织

---

在SPARC/Solaris工作站上下面两个结构的size  
分别是24和16，为什么不一样？

```
typedef struct _a{  
    char c1;  
    long i;  
    char c2;  
    double f;  
}a;
```

```
typedef struct _b{  
    char c1;  
    char c2;  
    long i;  
    double f;  
}b;
```



## 9.2.3 局部数据的组织

在SPARC/Solaris工作站上下面两个结构的size  
分别是24和16，为什么不一样？

```
typedef struct _a{  
    char c1; 0  
    long i;   4  
    char c2;  8  
    double f; 16  
}a;
```

```
typedef struct _b{  
    char c1; 0  
    char c2; 1  
    long i;  4  
    double f;8  
}b;
```



## 9.2.3 局部数据的组织

在**X86/Linux**机器的结果和SPARC/Solaris工作站不一样，是**20**和**16**。

```
typedef struct _a{  
    char c1; 0  
    long i;   4  
    char c2;  8  
    double f; 12  
}  
a;
```

```
typedef struct _b{  
    char c1; 0  
    char c2; 1  
    long i;  4  
    double f;8  
}  
b;
```



## 9.2.4 全局存储分配策略

### ■ 静态存储分配策略(FORTRAN)

如果在编译时能确定数据空间的大小，则可采用静态分配方法：在编译时刻为每个数据项目确定出在运行时刻的存储空间中的位置。

### ■ 动态存储分配策略(PASCAL)

如果在编译时不能确定运行时数据空间的大小，则必须采用动态分配方法。允许递归过程及动态申请、释放内存。

- 栈式动态存储分配
- 堆式动态存储分配

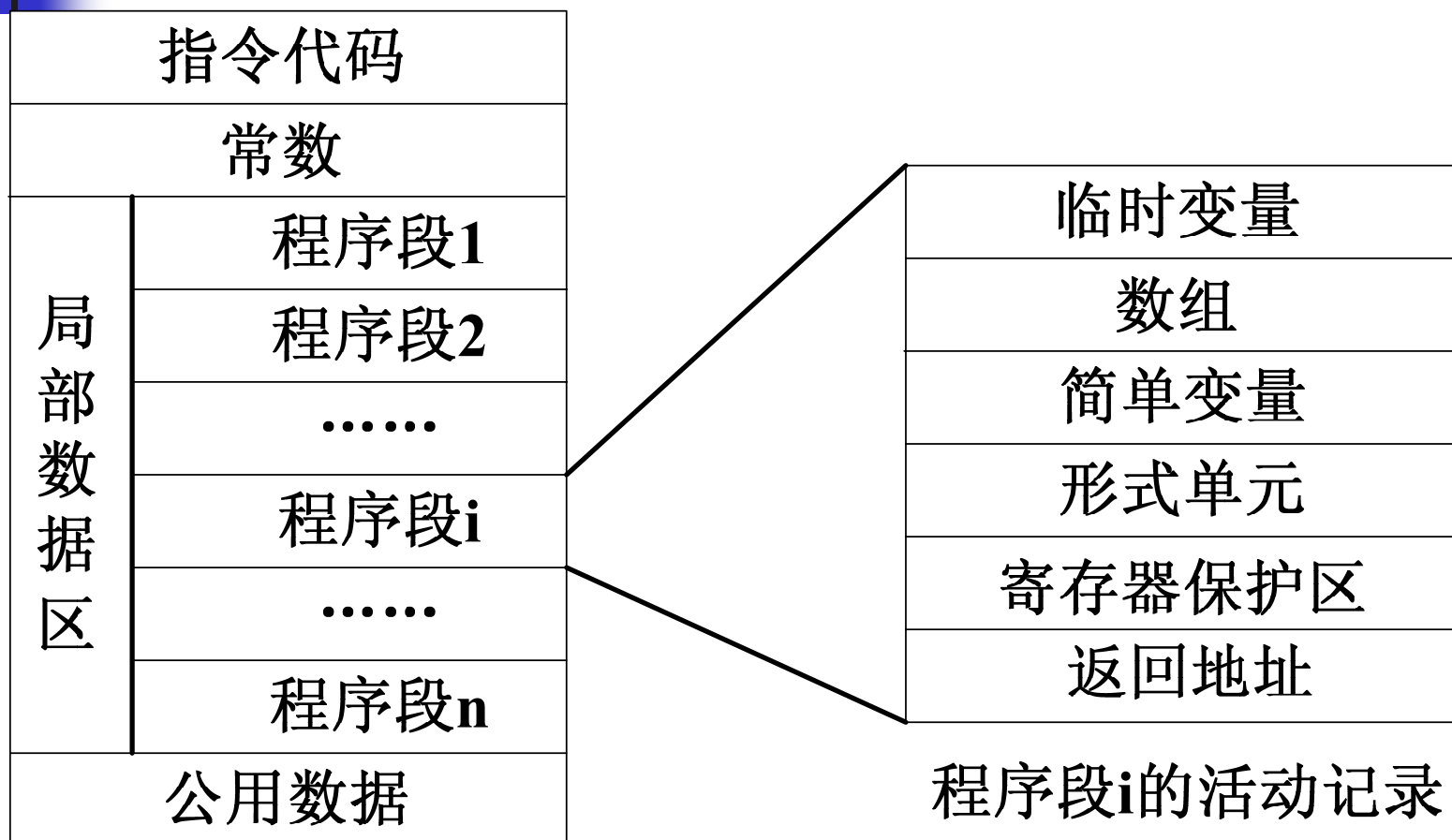




## 9.3 静态存储分配策略

- 如果在编译时就能确定一个程序在运行时所需要的存储空间的大小，则在编译时就能安排好目标程序运行时的全部数据空间，并能确定每个数据项的地址，存储空间的这种分配方法称为静态存储分配。必须满足如下条件：
  - 数据对象的长度和它在内存中的位置在编译时必须是已知的；
  - 不允许过程的递归调用，因为一个过程的所有活动都是用同样的局部名字绑定的；
  - 数据结构不能动态建立，因为没有运行时的存储分配机制。

# 某分段式程序运行时的内存划分



每个数据区有一个编号，地址分配时，在符号表中，对每个数据名登记其所属数据区编号及在该区中的相对位置。

考虑子程序段：

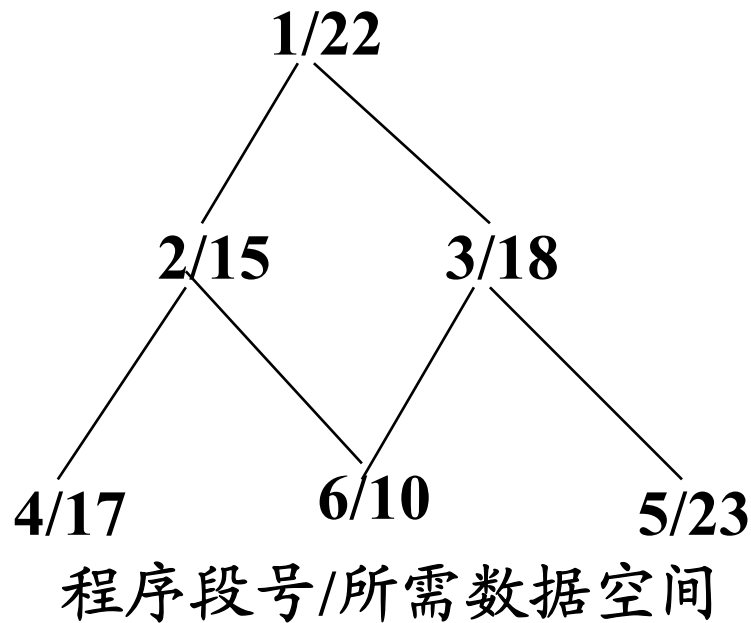
```
SUBROUTINE SWAP(A,B)
  T=A
  A=B
  B=T
  RETURN
END
```

名字	性质	地址	
NAME	ATTRIBUTE	DA	ADDR
SWAP	子程序，二目		
A	哑，实变量	k	a
B	哑，实变量	k	a+2
T	实变量	k	a+4



# 静态存储分配策略

- 顺序分配算法（基于调用图）
  - 按照程序段出现的先后顺序逐段分配



程序段 区域

1 0~21

2 22~36

3 37~54

4 55~71

5 72~94

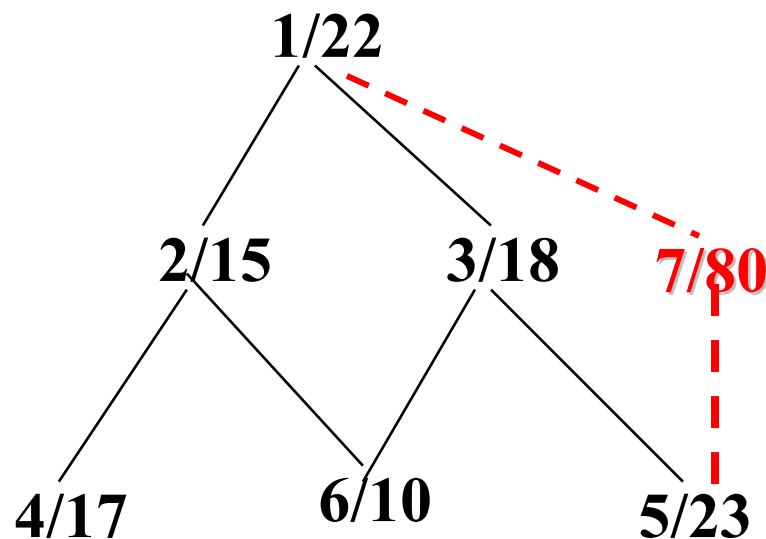
6 95~104

共需要105个存储单元

能用更少的空间么？

# 分层分配算法

- 允许程序段之间的覆盖（覆盖可能性分析）



程序段	区域
6	0~9
5	0~22
4	0~16
3	23~40
2	17~31
1	41~62

思考：如何设计分配算法？

共需要63个存储单元



## 9.4 栈式存储分配策略

### ■ 如果过程允许递归

- 某一时刻过程A可能已被自己调用了若干次，但只有最近一次处于执行状态。其余各次等待返回被中断的那次调用
- 必须保存每次调用相应的数据区内容（活动记录）

### ■ 引入一个运行栈

- 让过程的每次执行和过程的活动记录相对应。
- 每调用一次过程，就把该过程的活动记录压入栈中，返回时弹出。
- 假设寄存器top标记栈顶，局部名字x的相对地址为dx，则x的地址为 $\text{top} + \text{dx}$  或  $\text{dx}[\text{top}]$



## 9.4.1 调用序列和返回序列

- 过程调用和过程返回都需要执行一些代码来管理活动记录栈，保存或恢复机器状态等
- 过程调用和返回是通过在目标代码中生成调用序列和返回序列来实现的
  - 调用序列负责分配活动记录，并将相关信息填入活动记录中
  - 返回序列负责恢复机器状态，使调用过程能够继续执行
- 调用序列和返回序列常常都分成两部分，分处于调用过程和被调用过程中



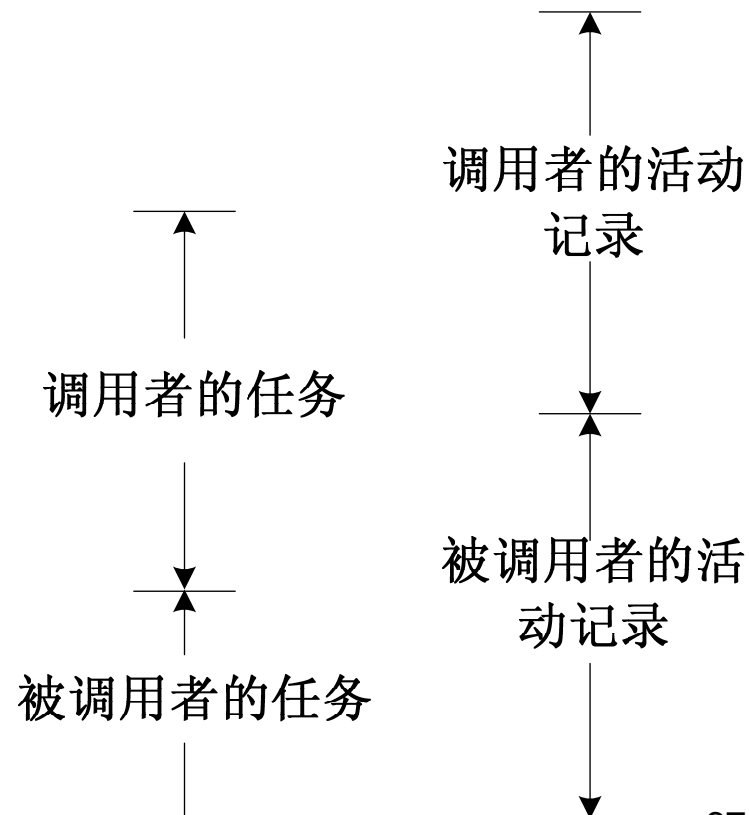
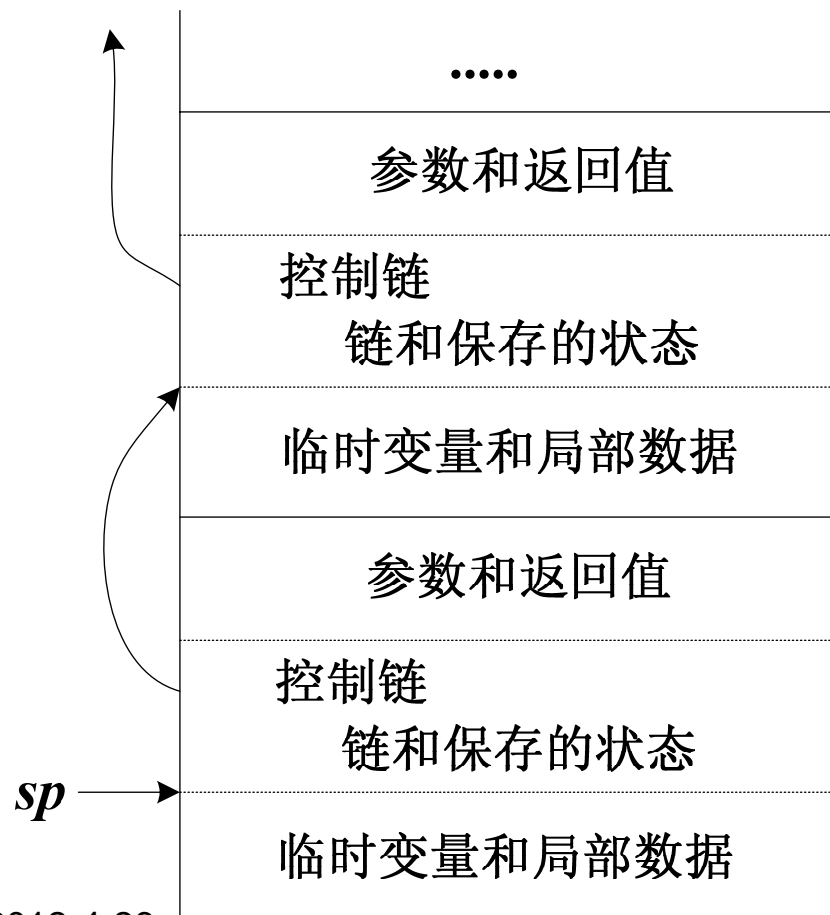
## 9.4.1 调用序列和返回序列

- 即使是同一种语言，过程调用序列、返回序列和活动记录中各域的排放次序，也会因实现而异
- 设计这些序列和活动记录的一些原则：
  - 以活动记录中间的某个位置作为基地址；
  - 长度能较早确定的域放在活动记录的中间；
  - 一般把临时数据域放在局部数据域的后面，以便其长度的改变不会影响数据对象相对于中间那些域的位置；
  - 用同样的代码来执行各个活动的状态保存和恢复；
  - 把参数域和可能有的返回值域放在紧靠调用者活动记录的地方。



## 9.4.1 调用序列和返回序列

### 调用者和被调用者之间的任务划分





## 9.4.1 调用序列和返回序列

---

### 一种可能的调用序列:

- 调用者计算实参
- 调用者把返回地址和 $sp$ 的旧值存入被调用者的活动记录中，然后将 $sp$ 移过调用者的局部数据域和临时变量域，以及被调用者的参数域和状态域
- 被调用者保存寄存器值和其它机器状态信息
- 被调用者初始化其局部数据，并开始执行。



## 9.4.1 调用序列和返回序列

---

### 一种可能的返回序列:

- 被调用者将返回值放入临近调用者的活动记录的地方。
- 利用状态域中的信息，被调用者恢复 $sp$ 和其它寄存器，并且按调用者代码中的返回地址返回。
- 尽管 $sp$ 的值减小了，调用者仍然可以将返回值复制到自己的活动记录中，并用它来计算一个表达式。



## 9.4.1 调用序列和返回序列

---

### 过程的参数个数可变的情况

- 函数返回值改成用寄存器传递
- 编译器产生将这些参数逆序进栈的代码
- 被调用函数能准确地知道第一个参数的位置
- 被调用函数根据第一个参数到栈中取第二、第三个参数等等
- 如C语言的printf



## 9.4.2 C语言的过程调用和返回

---

- 过程调用的三地址码:

**param  $x_1$**

**param  $x_2$**

**...**

**param  $x_n$**

**call p, n**

# 对于par和call产生的目标代码

1) 每个param  $x_i (i=1,2,\dots,n)$ 可直接翻译成如下指令:

$(i+3)[top] := x_i$  (传值)

$(i+3)[top] := \text{addr}(x_i)$  (传地址)

2) call p, n 被翻译成如下指令:

$1[top] := sp$  (保护现行sp)

$3[top] := n$  (传递参数个数)

JSR p (转子指令)

临时单元  
内情向量  
局部变量

形式单元

参数个数

返回地址

老sp

top →

sp →

调用过程的活动记录

3) 进入过程p后, 首先应执行下述指令:

$sp := top + 1$  (定义新的sp)

$1[sp] := \text{返回地址}$  (保护返回地址)

$top := top + L$  (新top)

L: 过程P的活动记录所需单元数,  
在编译时可确定。

top →

sp →

临时单元  
内情向量  
局部变量

形式单元

参数个数

返回地址

老sp

调用过程的活动记录

4) 过程返回时, 应执行下列指令:

$top := sp - 1$  (恢复调用前  $top$ )

$sp := 0[sp]$  (恢复调用前  $SP$ )

$X := 2[top]$  (把返回地址取到  $X$  中)

$UJ\ 0[X]$  (按  $X$  返回)  $top \rightarrow$

$UJ$  为无条件转移指令,  
即按  $X$  中的返回地址实行变址转移

临时单元  
内情向量  
局部变量

形式单元

参数个数

返回地址

老  $sp$

调用过程的活动记录

$sp \rightarrow$   
 $top \rightarrow$   
 $sp \rightarrow$





## 9.4.3 栈中的可变长数据

---

活动记录的长度在编译时不能确定的情况

- 局部数组的大小要等到过程激活时才能确定
- 在活动记录中为这样的数组分别存放数组指针的单元
- 运行时，这些指针指向分配在栈顶的存储空间

## 9.4.3 栈中的可变长数据

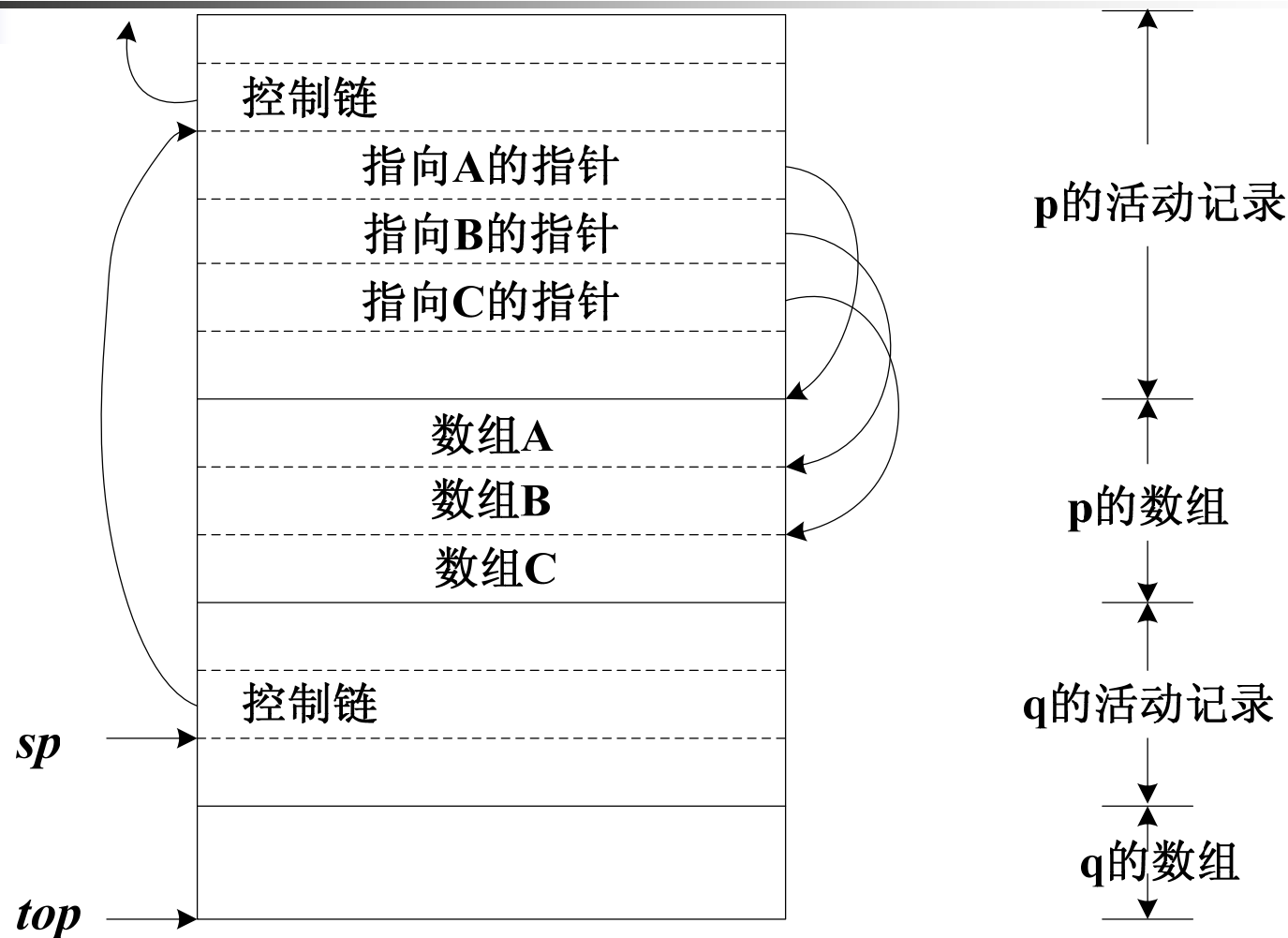


图9.13 访问动态分配的数组



# 栈式存储分配策略的局限

栈式分配策略在下列情况下行不通：

- 过程活动停止后，局部名字的值还必须维持
- 被调用者的活动比调用者的活动的生存期更长，此时活动树不能正确描绘程序的控制流
- 不遵守栈式规则的有Pascal语言和C语言的动态变量
- Java禁止程序员自己释放空间



## 9.5 栈中非局部数据的访问

---

### 本节内容

- 无嵌套过程的静态作用域的实现（C语言）
- 包含嵌套过程的静态作用域的实现（Pascal语言）
- 动态作用域的实现（Lisp语言）



## 9.5.1 无过程嵌套的静态作用域

- 假定:允许过程递归调用、允许过程含有可变数组, 但过程定义不允许嵌套, 如C语言。
- 过程体中的非局部引用可以直接使用静态确定的地址
- 局部变量在栈顶的活动记录中, 可以通过 $sp$ 指针来访问
- 无须深入栈中取数据, 无须访问链
- 过程可以作为参数来传递, 也可以作为结果来返回



## 9.5.1 无过程嵌套的静态作用域

---

C语言的函数声明不能嵌套，函数不论在什么情况下激活，要访问的数据分成两种情况：

- 非静态局部变量（包括形式参数），它们分配在活动记录栈顶的那个活动中
- 外部变量（包括定义在其它源文件中的外部变量）和静态的局部变量，它们都分配在静态数据区

## 全局数据说明

main()

{

### main中的数据说明

}

void R()

{

### R中的数据

}

...

void Q()

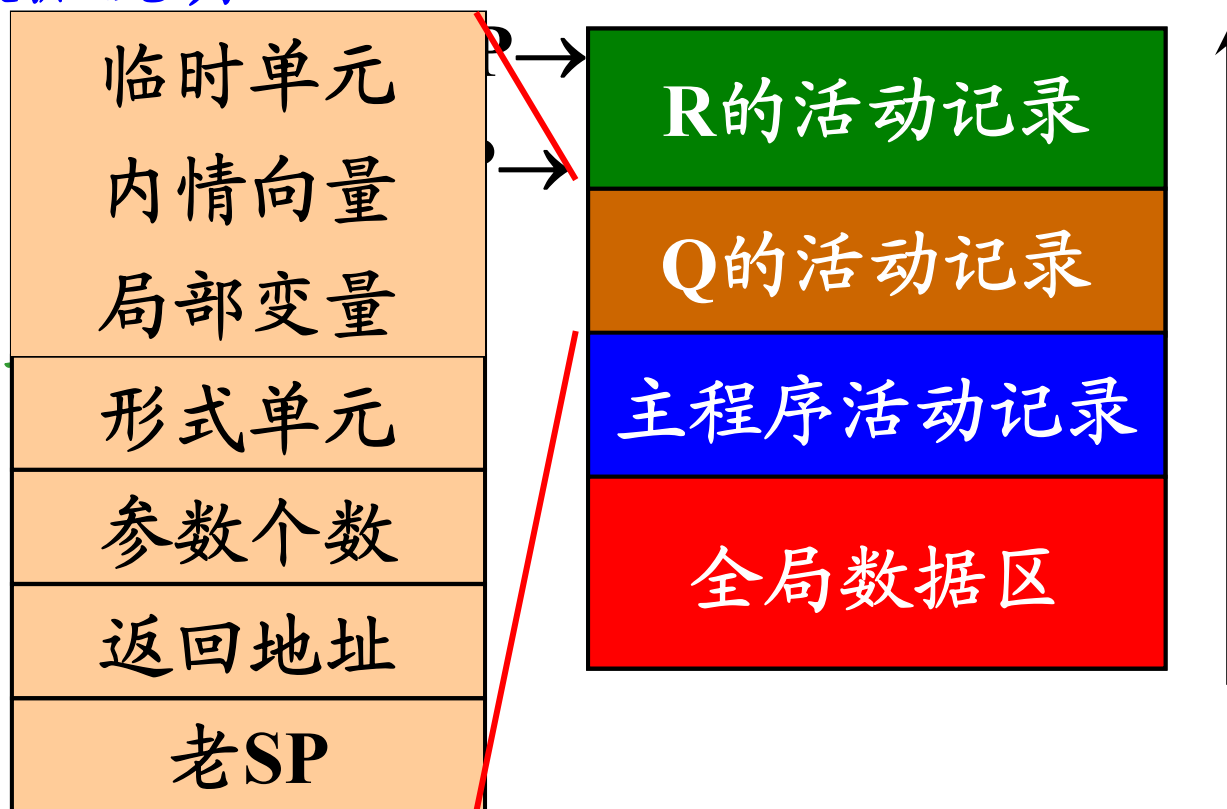
{

### Q中的数据说明

2012-4-26

}

■ 主程序 → 过程Q → 过程R





## 9.5.2 有过程嵌套的静态作用域

---

假定语言不仅允许过程的递归调用(和可变数组)，而且允许过程定义的嵌套，如**PASCAL**，**PL**语言。

**sort**

**readarray**

**exchange**

**quicksort**

**partition**



- (1) **program sort(input, output);**
- (2) **var a :array[0..10] of integer;**
- (3) **x :integer;**
- (4) **procedure readarray;**
- (5) **var i :integer;**
- (6) **begin ...a... end {readarray};**
- (7) **procedure exchange(i,j:integer);**
- (8) **begin**
- (9) **x:= a[i]; a[i]:=a[j]; a[j]:= x;**
- (10) **end {exchange };**

**program sort**

**procedure readarray**

**procedure exchange**

**procedure quicksort**

**function partition**

- (11) **procedure quicksort(m,n:integer);**
- (12) **var k,v:integer;**
- (13) **function partition(y,z:integer): integer;**
- (14) **var i,j:integer;**
- (15) **begin ...a...**
- (16) **...v...**
- (17) **...exchange(i,j);...**
- (18) **end {partition};**
- (19) **begin ...end {quicksort};**
- (20) **begin ...end {sort}.**

**program sort**

**procedure readarray**

**procedure exchange**

**procedure quicksort**

**function partition**



## 9.5.2 有过程嵌套的静态作用域

---

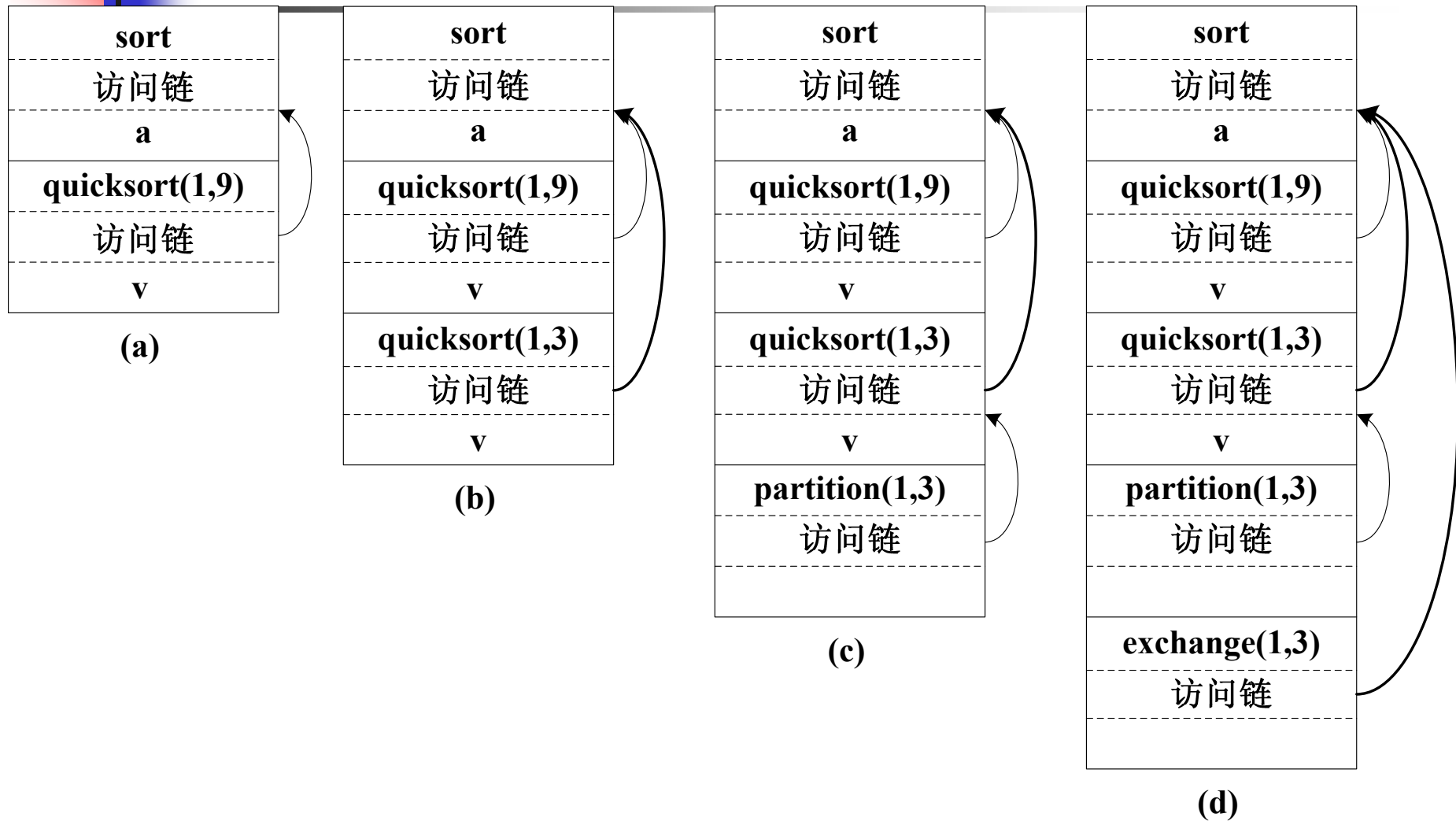
引入概念：过程嵌套深度

<b>sort</b>	<b>1</b>
<b>readarray</b>	<b>2</b>
<b>exchange</b>	<b>2</b>
<b>quicksort</b>	<b>2</b>
<b>partition</b>	<b>3</b>

- 变量的嵌套深度：它的声明所在过程的嵌套深度作为该名字的嵌套深度

## 9.5.2 有过程嵌套的静态作用域

### 1. 访问链



## 9.5.2 有过程嵌套的静态作用域

假定过程 $p$ 的嵌套深度为 $n_p$ ，它引用嵌套深度为 $n_a$ 的变量 $a$ ， $n_a \leq n_p$ 。如何访问 $a$ 的存储单元？

- 从栈顶的活动记录开始，追踪访问链 $n_p - n_a$ 次。
- 到达 $a$ 的声明所在过程的活动记录。
- 访问链的追踪用间接操作就可完成。



## 9.5.2 有过程嵌套的静态作用域

---

过程 $p$ 对变量 $a$ 访问时,  $a$ 的地址由下面的二元组表示:

( $n_p - n_a$ ,  $a$ 在活动记录中的偏移)



## 9.5.2 有过程嵌套的静态作用域

### 如何建立访问链

假定嵌套深度为 $n_p$ 的过程 $p$ 调用嵌套深度为 $n_x$ 的过程 $x$

- $n_p < n_x$  的情况

- $x$  必须在 $p$ 中进行定义，否则它对于 $p$ 来说就是不可访问的
- 被调用过程的访问链必须指向调用过程的活动记录的访问链



## 9.5.2 有过程嵌套的静态作用域

### 如何建立访问链

假定嵌套深度为 $n_p$ 的过程 $p$ 调用嵌套深度为 $n_x$ 的过程 $x$

#### ■ $n_p \geq n_x$ 的情况

- $p$ 和 $x$ 的嵌套深度分别为1, 2, ...,  $n_x - 1$ 的外围过程肯定相同
- 追踪访问链 $n_p - n_x + 1$ 次, 到达了静态包围 $x$ 和 $p$ 的且离它们最近的那个过程的最新活动记录
- 所到达的访问链就是 $x$ 的活动记录中的访问链应该指向的那个访问链



## 9.5.2 有过程嵌套的静态作用域

### 2. 过程型参数的访问链

```
program param(input, output);
```

```
  procedure b(function h(n: integer): integer);
```

```
    begin writeln(h(2)) end {b};
```

```
  procedure c;
```

```
    var m: integer;
```

```
    function f(n: integer): integer;
```

```
      begin f := m+n end {f};
```

```
    begin m := 0; b(f) end {c};
```

```
  begin
```

```
    c
```

```
  end.
```

过程作为参数传递时，怎样在该过程被激活时建立它的访问链  
从b的访问链难以建立f的访问链

激活f

c传递参数f时，像调用f一样为f建立一个访问链，该访问链同f一起传递给b。f被激活时用这个访问链建立f的活动记录的访问链

## 9.5.2 有过程嵌套的静态作用域

program param(input, output); ( 过程作为参数 )

procedure b(function h(...  
begin writeln(h(2)) end ;

procedure c;

var m: integer;

function f(n: integer)...  
begin f := m+n end {f};

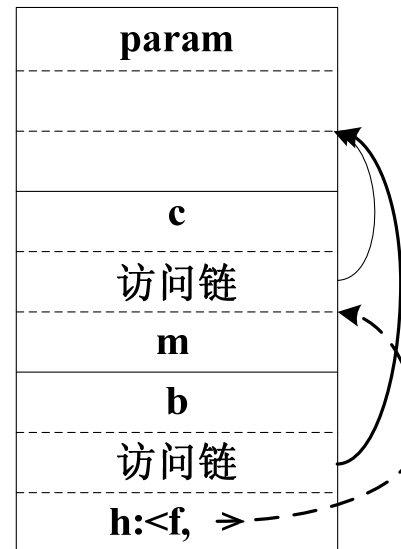
begin m := 0; b(f) end {c};

begin

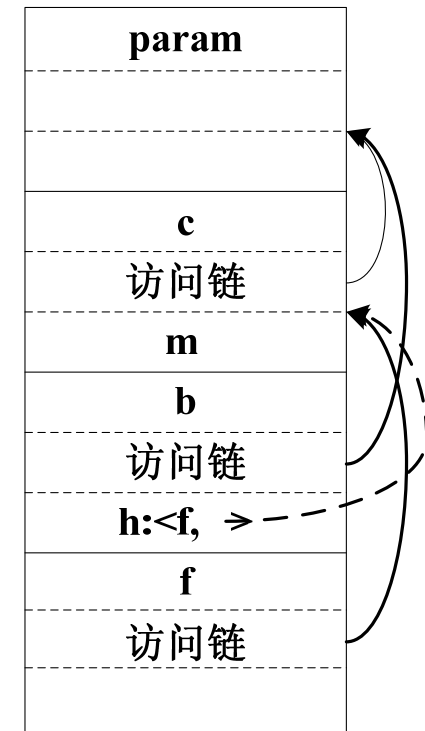
c

end.

2012-4-26



(a)



(b)

## 9.5.2 有过程嵌套的静态作用域

```
program ret (input, output); ( 过程作为返回值 )
```

```
  var f: function (integer): integer;
```

```
  function a: function (integer): integer;
```

```
    var m: integer;
```

```
    function addm (n: integer): integer;
```

```
      begin return m+n end;
```

```
    begin m:= 0; return addm end;
```

```
  procedure b (g: function (integer): integer);
```

```
    begin writeln ( g(2)) end;
```

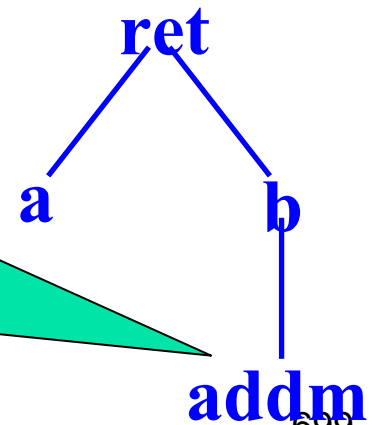
```
begin
```

```
  f := a; b(f)
```

```
end.
```

栈式存储分配  
策略失效!!

活动树



被调过程addm  
活动的生存期超  
出了调用过程a  
活动的生存期



## 9.5.2 有过程嵌套的静态作用域

### 3. Display表

Display表是一个指针数组 $d$ ，在运行过程中，若当前活动的过程的嵌套深度为 $i$ ，则 $d[i]$ 中存放当前的活动记录地址， $d[i-1], d[i-2], \dots, d[1]$ 中依次存放着当前活动的过程的直接外层， $\dots$ ，直至最外层（主程序）等每一层过程的最新活动地址。

这样，嵌套深度为 $j$ 的变量 $a$ 存放在由 $d[j]$ 所指出的活动记录中。

在运行过程中维持一个Display表实现非局部访问比存取链效率要高。



## 9.5.2 有过程嵌套的静态作用域

Display表的维护（过程不作为参数传递）

当嵌套深度为 $i$ 的过程的活动记录压在栈顶时：

- (1) 在新的活动记录中保存 $d[i]$ 的值；
- (2) 置 $d[i]$ 指向新的活动记录。

在一个活动结束前， $d[i]$ 置成保存的旧值。

用Display表如何访问非局部量？

1. Display表是一个数组，开始地址用通用寄存器指出；
2. Display表由一组寄存器实现。

```

program P;
var a, x : integer;
procedure Q(b: integer);
  var i: integer;
  procedure R(u: integer; var v:
integer);
    var c, d: integer;
  begin
    if u=1 then R(u+1, v)
    .....
    v:=(a+c)*(b-d);
    .....
  end {R}
begin
  .....
  R(1,x);
  .....
end {Q}

```

## 一个例子

主程序 P → 过程 S →  
 过程 Q → 过程 R →  
 过程 R

```

procedure S;
  var c, i: integer;
  begin
    a:=1;
    Q(c);
    .....
  end {S}
begin
  a:=0;
  S;
  .....
end. {P}

```

## 一、通过静态链访问非局部数据

- **静态链**：指向本过程的直接外层过程的活动记录的起始地址，也称访问链。
- **动态链**：指向本过程的调用过程的活动记录的起始地址，也称控制链。

TOP→

临时单元  
内情向量  
局部变量  
形式单元  
参数个数

2

**静态链**

1

返回地址

SP→0

**动态链(老SP)**



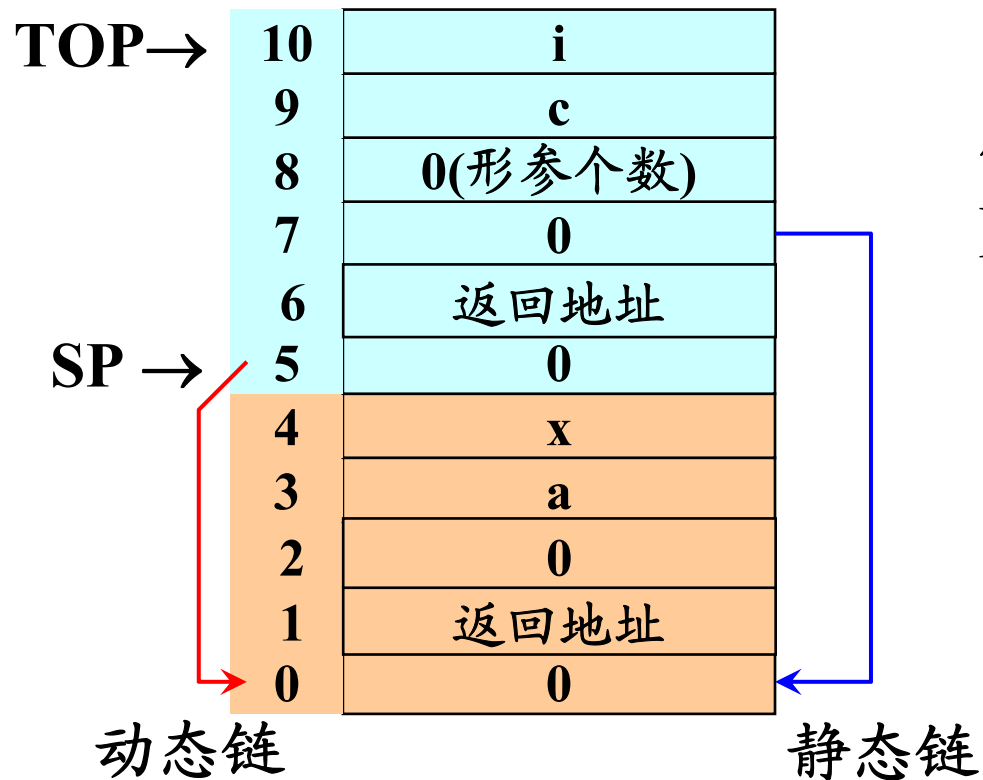
## ● 主程序 P

---

TOP→	4	x
	3	a
	2	0
	1	返回地址
SP →	0	0



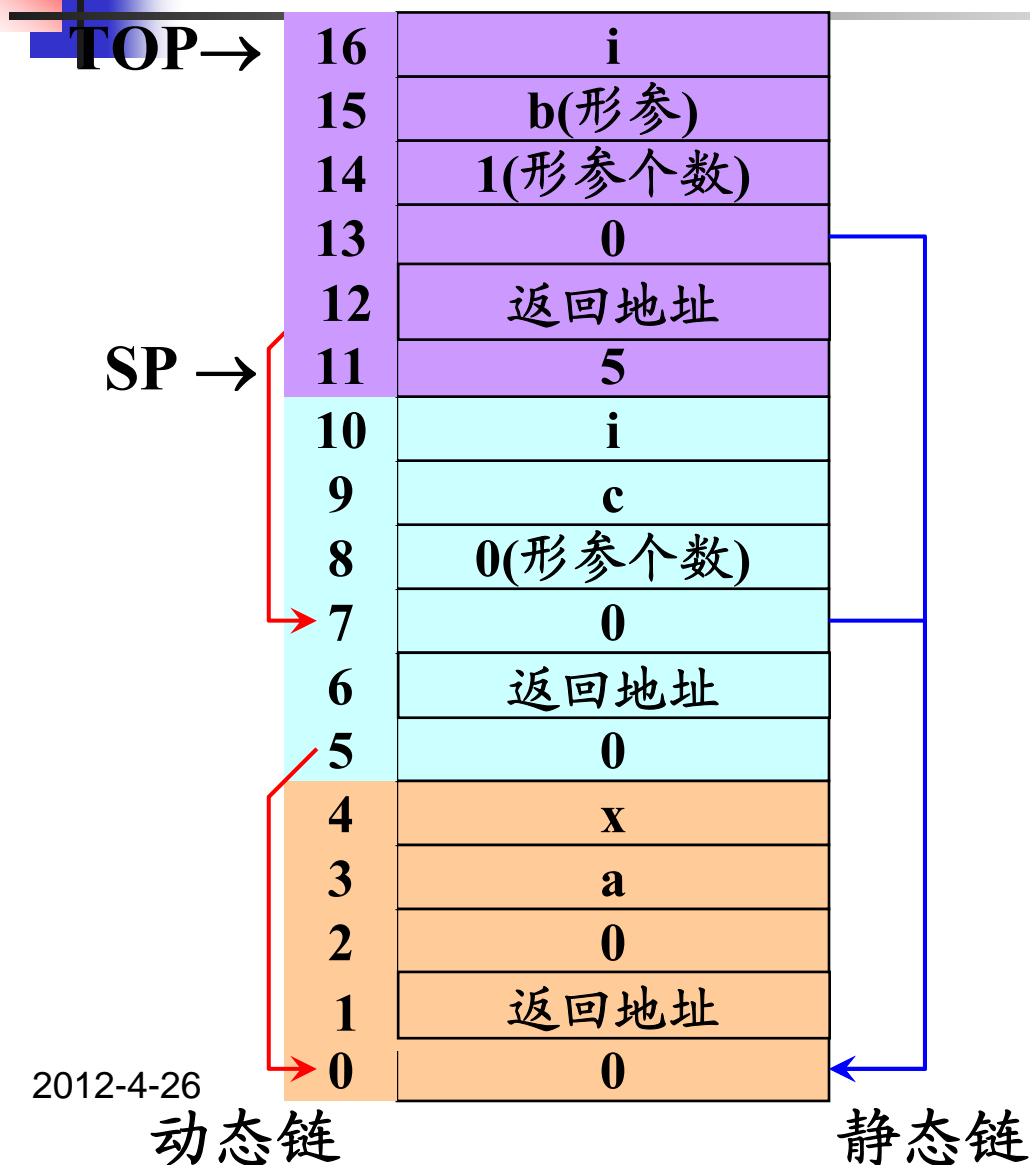
## ● 主程序 P → 过程 S



**问：** 第N层过程调用第 N+1层过程，如何确定被调用过程(第 N+1层) 的静态链？

**答：** 调用过程(第N层过程)的最新活动记录的起始地址。

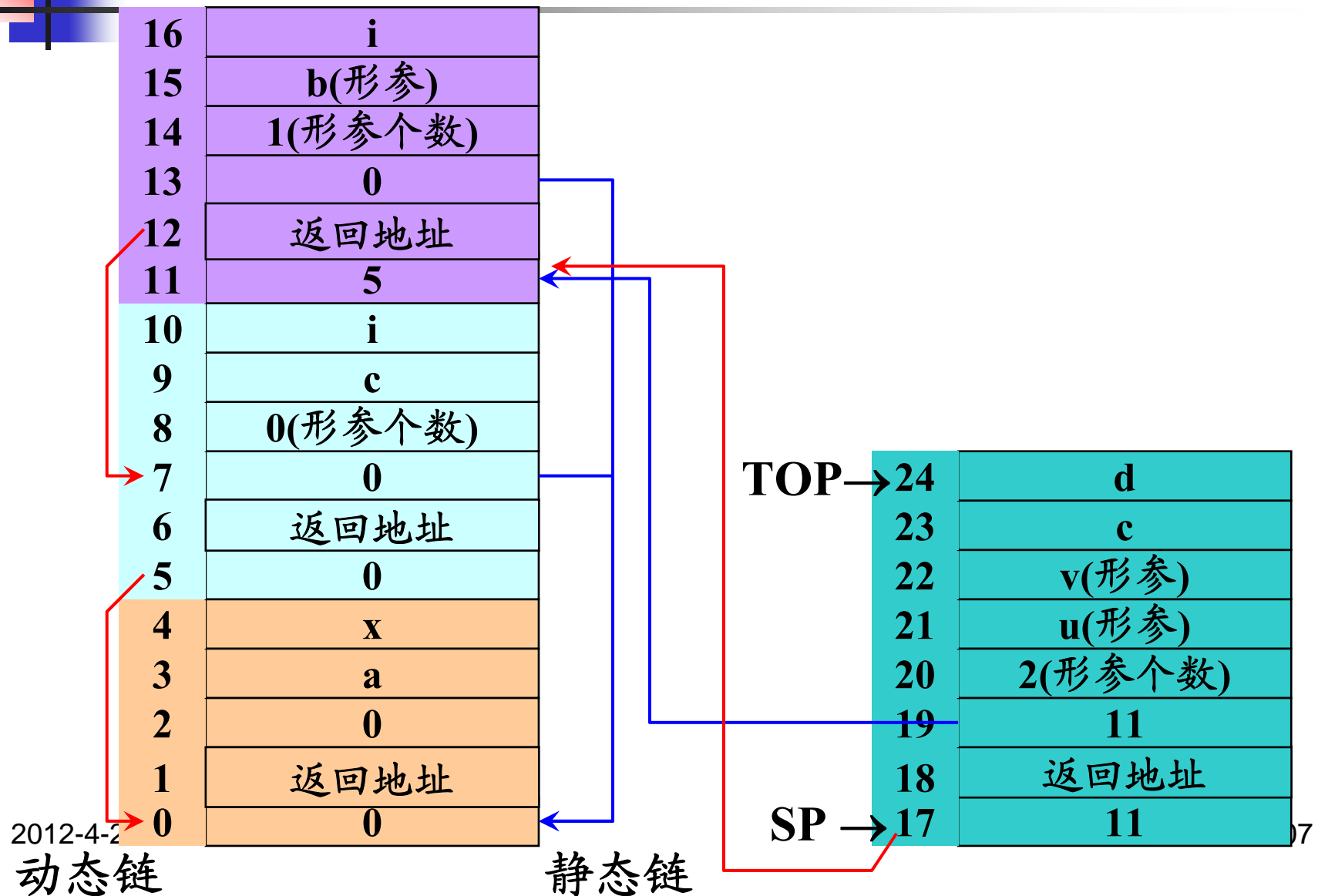
● 主程序 P → 过程 S → 过程 Q



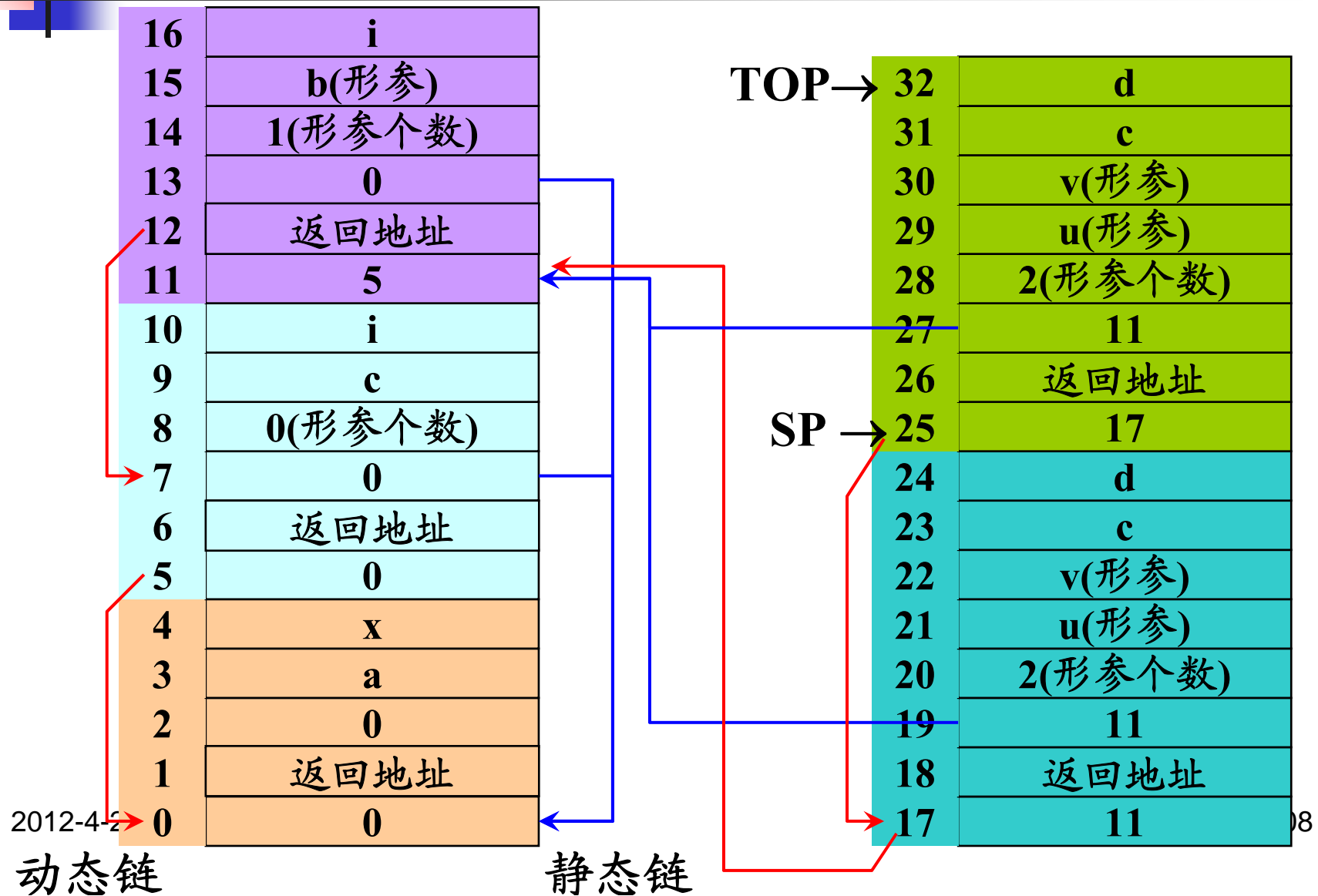
问：第N层过程调用第N层过程，如何确定被调用过程(第N层)的静态链？

答：调用过程(第N层过程)的静态链的值。

● 主程序 P → 过程 S → 过程 Q → 过程 R



● 主程序 P → 过程 S → 过程 Q → 过程 R  
→ 过程 R



● 主程序 P → 过程 S → 过程 Q → 过程 R  
→ 过程 Q

16	i
15	b(形参)
14	1(形参个数)
13	0
12	返回地址
11	5
10	i
9	c
8	0(形参个数)
7	0
6	返回地址
5	0
4	x
3	a
2	0
1	返回地址
0	0

2012-4-2

动态链

静态链

答: 沿着调用过程(第 N 层过程)的静态链向前走 x 步到达的活动记录的静态链的值。

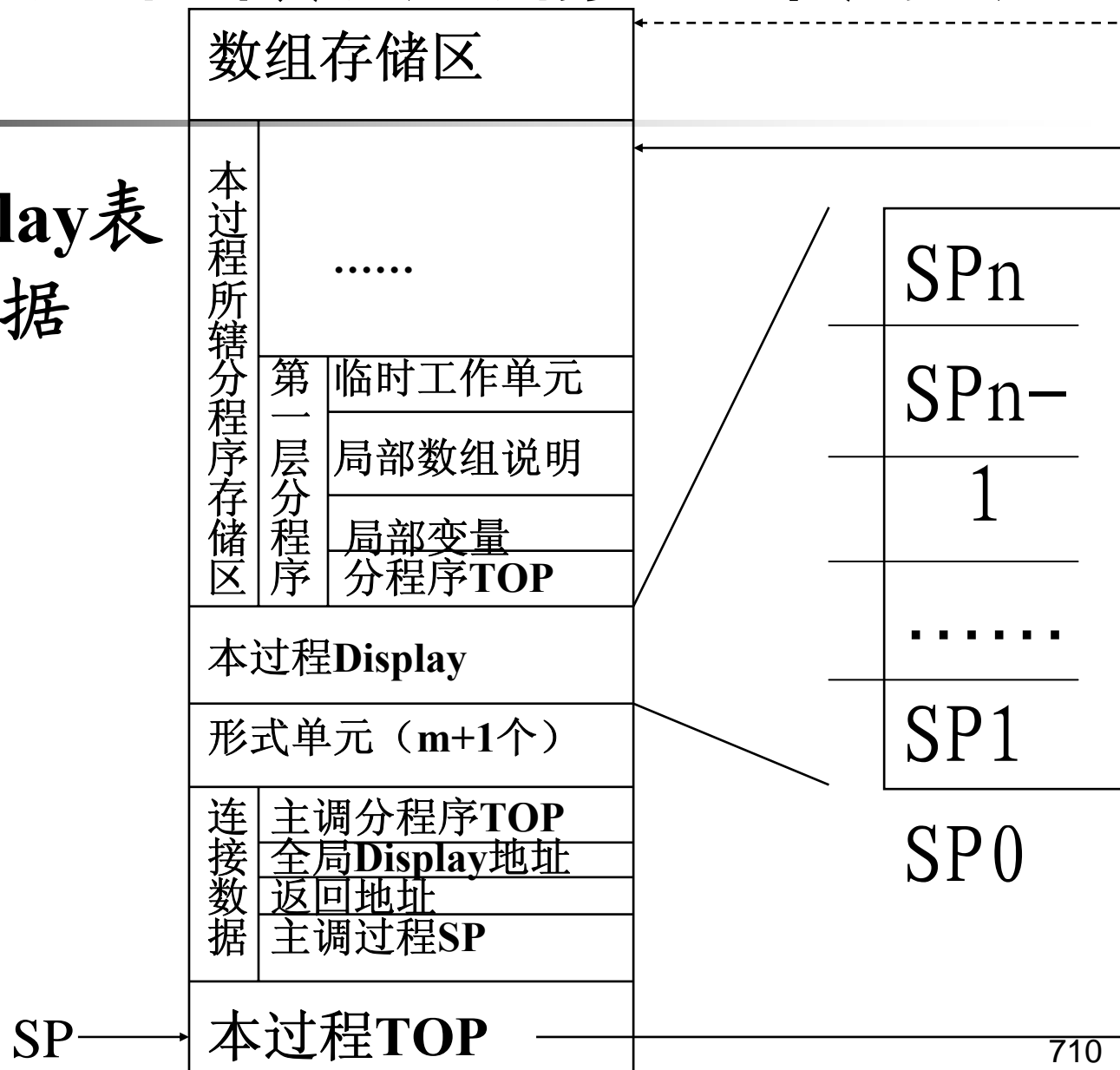
28	1(形参个数)
27	0
26	返回地址
25	17
24	d
23	c
22	v(形参)
21	u(形参)
20	2(形参个数)
19	11
18	返回地址
17	11


9

## 9.5.2 有过程嵌套的静态作用域

二、通过Display表访问非局部数据

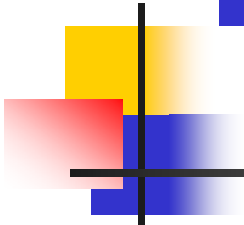
$SP_n$  为第  $n$  层过程数据区首址

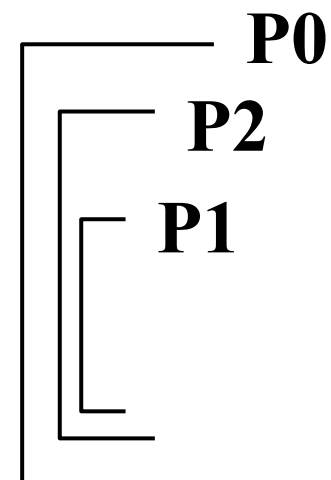
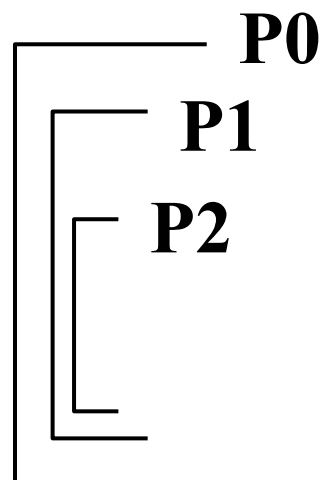
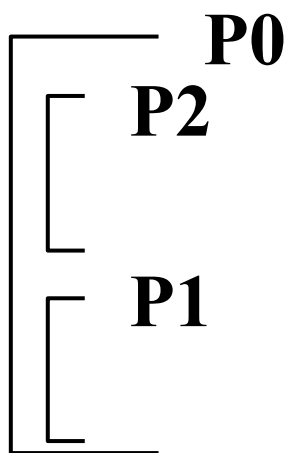




令过程R的外层为Q，Q的外层为主程序P，  
则过程R运行时的DISPLAY表内容为：

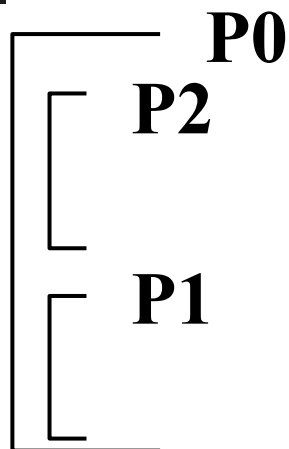
2	R 的现行活动记录的 地址(SP 的现值)
1	Q 的最新活动记录 的地址
0	P 的活动记录 的地址

- 
- 问题：当过程P1调用过程P2而进入P2后，P2应如何建立起自己的display表？

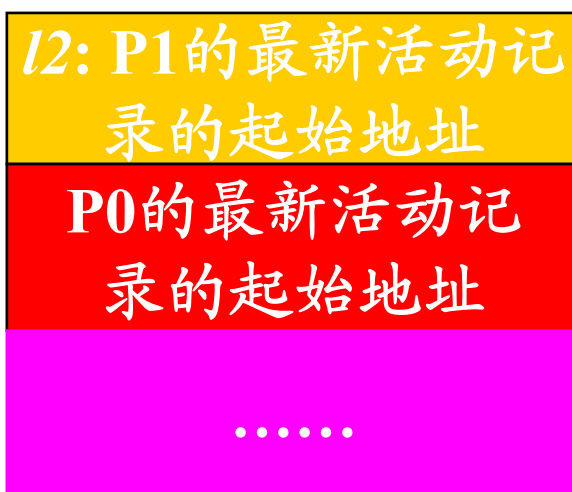




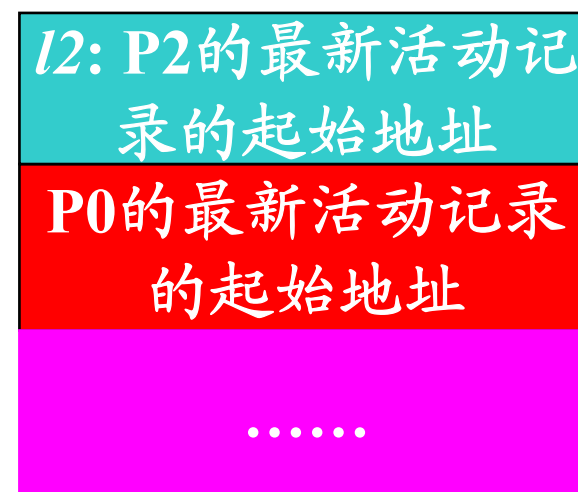
- 问题：当过程P1调用过程P2而进入P2后，P2应如何建立起自己的display表？



从P1的display表中自底而上地取过*l2*个单元（*l2*为P2的层数）再添上进入P2后新建的SP值就构成了P2的display表。

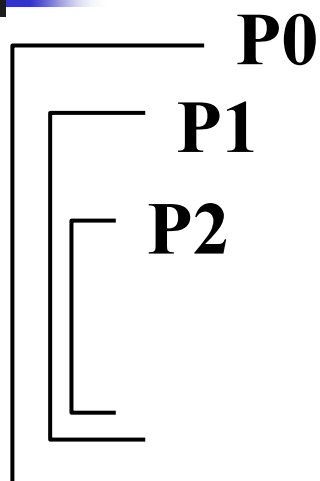


P1的display表

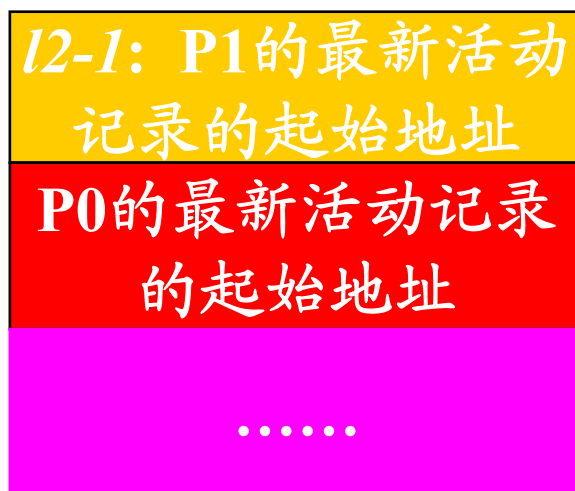


P2的display表

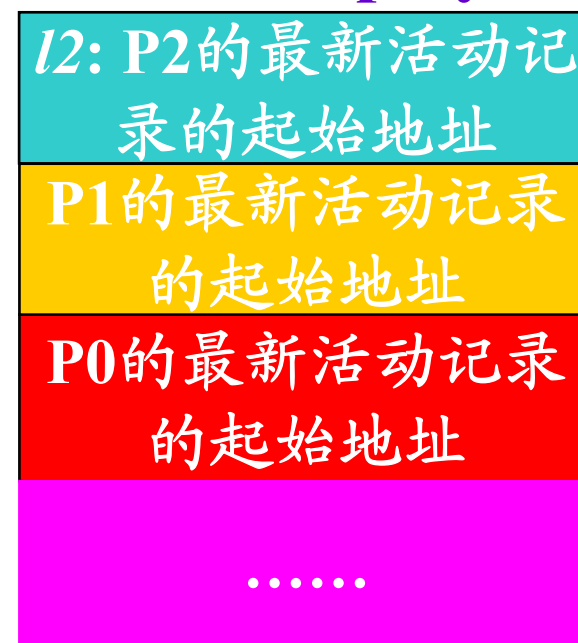
- 问题：当过程P1调用过程P2而进入P2后，P2应如何建立起自己的display表？



从P1的display表中自底而上地取过 $l2$ 个单元（ $l2$ 为P2的层数）再添上进入P2后新建的SP值就构成了P2的display表。

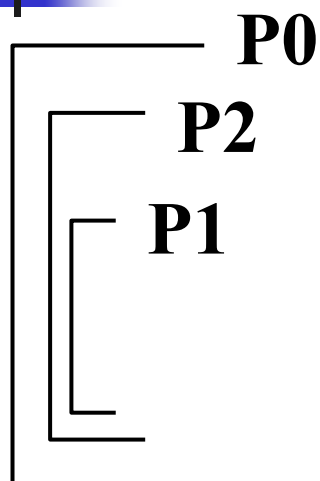


P1的display表



P2的display表

- 问题：当过程P1调用过程P2而进入P2后，P2应如何建立起自己的display表？



从P1的display表中自底而上地取过 $l2$ 个单元（ $l2$ 为P2的层数）再添上进入P2后新建的SP值就构成了P2的display表。

P1的最新活动记录的起始地址

$l2$ : P2的最新活动记录的起始地址

P0的最新活动记录的起始地址

.....

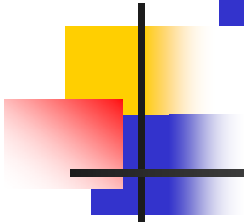
P1的display表

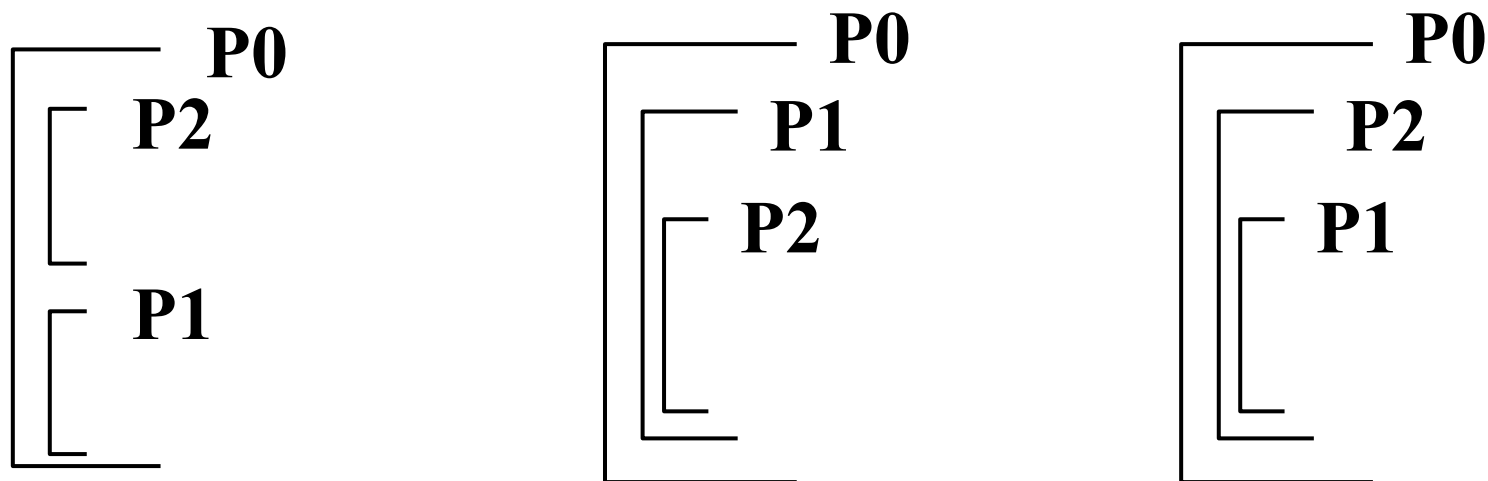
$l2$ : P2的最新活动记录的起始地址

P0的最新活动记录的起始地址

.....

P2的display表

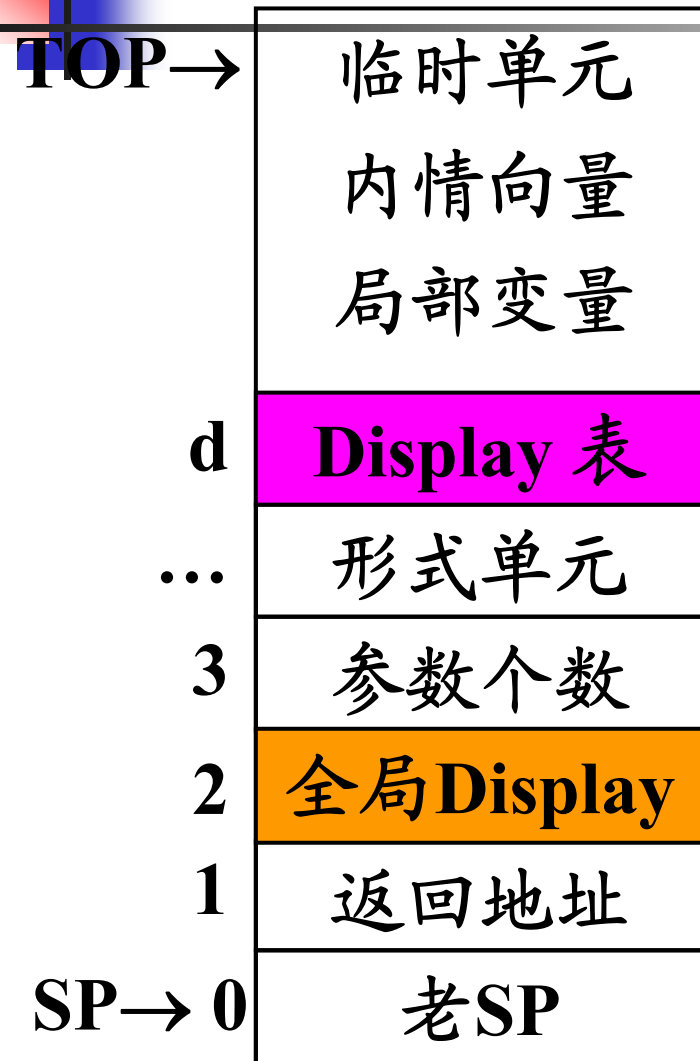
- 
- 问题：当过程P1调用过程P2而进入P2后，P2应如何建立起自己的display表？



**答案：**从P1的display表中自底而上地取过*l2*个单元（*l2*为P2的层数）再添上进入P2后新建立的SP值就构成了P2的display表。

👉 把P1的display表地址作为连接数据之一(称为**全局display地址**)传送给P2就能够建立P2的display表。

## 嵌套过程语言活动记录



- diplay表在活动记录中的相对地址d在编译时能完全确定。
- 假定在现行过程中引用了某层过程(令其层次为k)的X变量, 那么, 可用下面两条指令获得X的地址:  
LD R<sub>1</sub> (d+k)[SP]  
LD R<sub>2</sub> dx[R<sub>1</sub>]

```

program P;
var a, x : integer;
procedure Q(b: integer);
  var i: integer;
  procedure R(u: integer; var v:
integer);
    var c, d: integer;
  begin
    if u=1 then R(u+1, v)
    .....
    v:=(a+c)*(b-d);
    .....
  end {R}
begin
  .....
  R(1,x);
  .....
end {Q}

```

## 一个例子

主程序 P → 过程 S →  
 过程 Q → 过程 R →  
 过程 R

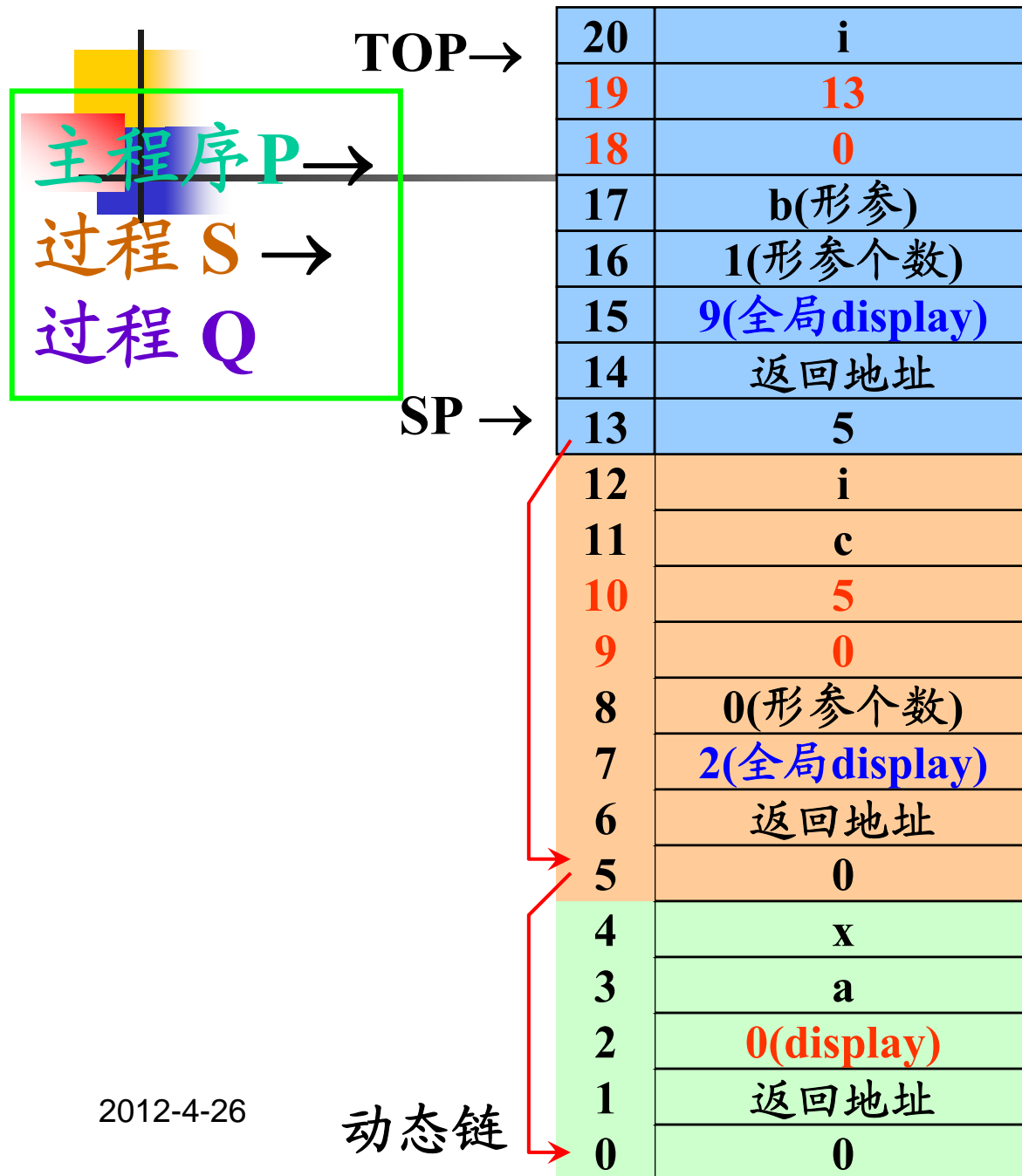
```

procedure S;
  var c, i:integer;
  begin
    a:=1;
    Q(c);
    .....
  end {S}
begin
  a:=0;
  S;
  .....
end. {P}

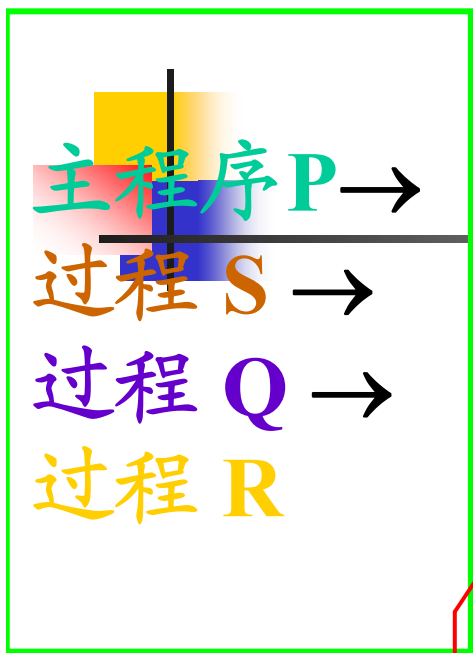
```

主程序 →  
过程 S

TOP →	12	i
	11	c
	10	5
	9	0
	8	0(形参个数)
	7	2(全局display)
SP →	6	返回地址
	5	0
	4	x
	3	a
	2	0(display)
	1	返回地址
动态链	0	0



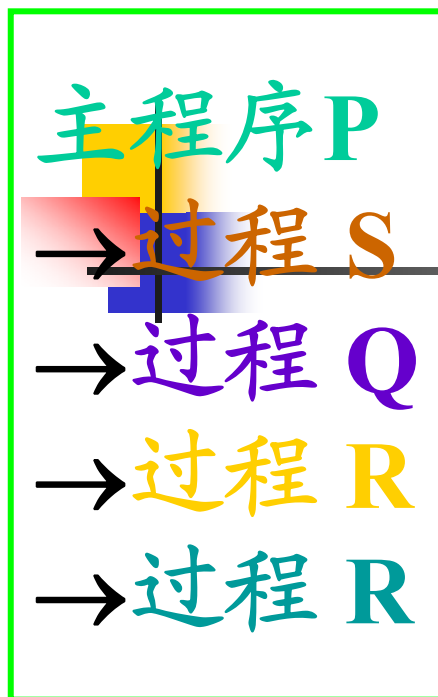




20	i
19	13
18	0
17	b(形参)
16	1(形参个数)
15	9(全局display)
14	返回地址
13	5
12	i
11	c
10	5
9	0
8	0(形参个数)
7	2(全局display)
6	返回地址
5	0
4	x
3	a
2	0(display)
1	返回地址
0	0

TOP →	31	d
	30	c
	29	21
	28	13
	27	0
	26	v(形参)
	25	u(形参)
	24	2(形参个数)
	23	18(全局display)
	22	返回地址
SP →	21	13

2012-4-26 动态链



20	i
19	13
18	0
17	b(形参)
16	1(形参个数)
15	9(全局display)
14	返回地址
13	5
12	i
11	c
10	5
9	0
8	0(形参个数)
7	2(全局display)
6	返回地址
5	0
4	x
3	a
2	0(display)
1	返回地址
0	0

TOP → 42	d
41	c
40	32
39	13
38	0
37	v(形参)
36	u(形参)
35	2(形参个数)
34	27(全局display)
33	返回地址
SP → 32	21
31	d
30	c
29	21
28	13
27	0
26	v(形参)
25	u(形参)
24	2(形参个数)
23	18(全局display)
22	返回地址
21	13

# 过程调用、过程进入、过程返回

1. 每个par  $T_i (i=1,2,\dots,n)$ 可直接翻译成如下指令:

$(i+4)[TOP] := T_i$  (传值)

$(i+4)[TOP] := \text{addr}(T_i)$  (传地址)

TOP →

SP →

临时单元  
内情向量  
局部变量

Display 表

形式单元

参数个数

全局Display

返回地址

老SP

调用过程的  
活动记录

.....

# 过程调用、过程进入、过程返回

2. call P, n 被翻译成:

1[TOP]:=SP (保护现行SP)

3[TOP]:=SP+d (传送现行display地址)

4[TOP]:=n (传递参数个数)

JSR (转子指令)

TOP→

SP →

临时单元  
内情向量  
局部变量

Display 表

形式单元

参数个数

全局Display

返回地址

老SP

调用过程的  
活动记录

.....

# 过程调用、过程进入、过程返回

3. 转进过程P后，首先定义新的SP和TOP，保存返回地址。
4. 根据"全局display"建立现行过程的display：从全局display表中自底向上地取1个单元，在添上进入P后新建立的SP值就构成了P的display。

TOP →

临时单元  
内情向量  
局部变量

**Display 表**

形式单元

参数个数

全局Display

**返回地址**

老SP

SP →

调用过程的  
活动记录

.....

# 过程调用、过程进入、过程返回

5. 过程返回时，执行下述指令：

**TOP:=SP-1**

**SP:=0[SP]**

**X:=2[TOP]**

**UJ 0[X]**

TOP→

临时单元  
内情向量  
局部变量

Display 表

形式单元

参数个数

全局Display

返回地址

老SP

SP →

TOP→

调用过程的  
活动记录

SP →

.....



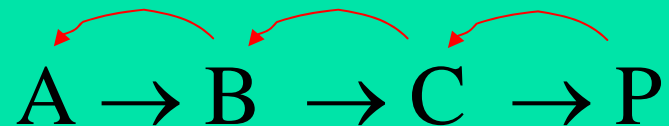
## 9.5.3 动态作用域的实现

- 从技术上讲，如果作用域策略需要基于程序运行时才能知道的因素，则任何作用域策略都将是动态的。但“动态作用域”这个术语通常是指下面的策略：名字 $x$ 的引用指向带有 $x$ 声明的最近被调用的过程中 $x$ 的这个声明。
- 在某种意义上说，动态作用域规则相对于时间，而静态作用域规则相对于空间。

## 9.5.3 动态作用域

程序运行时，一个名字 $a$ 实施其影响，直到含 $a$ 的声明的一个过程开始执行时暂停，此过程停止时，该影响恢复。

设有下面的调用序列：



过程P中有对 $x$ 的非局部引用，沿动态链（红链）查找，最先找到的便是。



## 9.5.3 动态作用域

program dynamic(input, output);

var r: real;

procedure show;

begin write(r: 5: 3) end;

procedure small;

var r: real;

begin r := 0.125; show end;

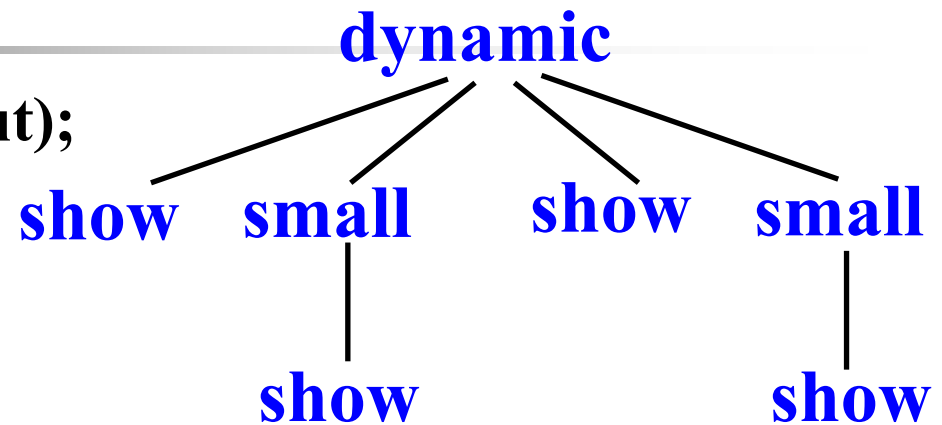
begin

r := 0.25;

show; small; writeln;

show; small; writeln

end.



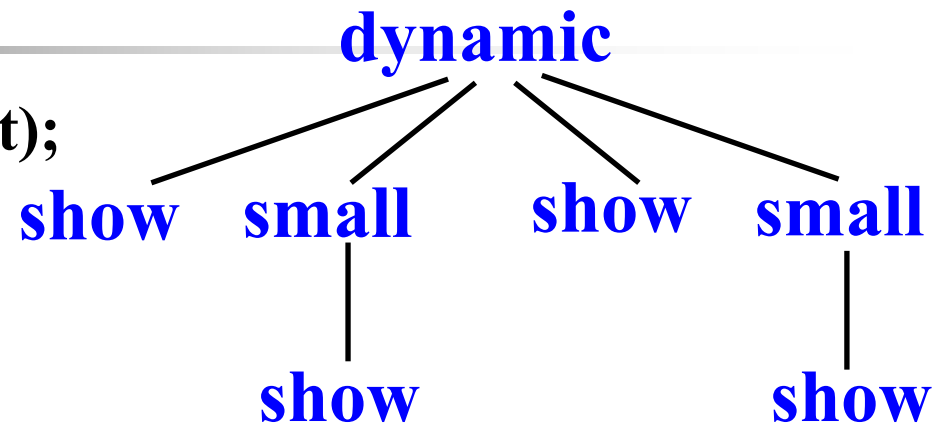
静态作用域

0.250 0.250

0.250 0.250

## 9.5.3 动态作用域

```
program dynamic(input, output);  
  var r: real;  
  procedure show;  
    begin write(r: 5: 3) end;  
  procedure small;  
    var r: real;  
    begin r := 0.125; show end;  
  
begin  
  r := 0.25;  
  show; small; writeln;  
  show; small; writeln  
end.
```



动态作用域

0.250	0.125
0.250	0.125



## 9.5.3 动态作用域

### 实现动态作用域的方法

- 深访问(访问非局部名字时的开销大,易于实现过程类参数)  
用控制链搜索运行栈, 寻找包含该非局部名字的  
第一个活动记录
- 浅访问(活动开始和结束时的开销大)
  - 将每个名字的当前值保存在静态分配的内存空间中
  - 当过程 $p$ 开始一个新的活动时,  $p$ 的局部名字 $n$ 使用  
在静态数据区分配给 $n$ 的内存单元。 $n$ 的先前值必须  
保存在 $p$ 的活动记录中, 当 $p$ 的活动结束时再恢复



## 9.6 堆管理

- 对于允许程序为变量在运行时**动态申请和释放存储空间**的语言,采用堆式分配是最有效的解决方案.
  - 活动结束后必须保持局部名字的值。
  - 被调用者的活动比调用者的活动的生存期长。
- 堆式分配的**基本思想是**,为运行的程序划出适当大的空间(称为堆**Heap**),每当程序申请空间时,就从堆的空闲区找出一块空间分配给程序,每当释放时则回收之。
- 内存管理器
  - 分配和回收堆区空间的子系统
  - 是应用程序和操作系统之间的一个接口



## 9.6.1 内存管理器

---

### ■ 内存管理器的基本任务

- 空间分配。每当程序为某个变量或对象申请一块内存空间时，内存管理器就产生一块连续的具有被请求大小的堆空间。
- 空间回收。内存管理器把回收的空间返还到空闲空间的缓冲池中，用于满足其它的分配请求。



## 9.6.1 内存管理器

---

- 如果下面两个条件成立的话，内存管理将会变得相对简单一些
  - 所有的分配请求都要求相同大小的块；
  - 存储空间按照某种可以预见的方式来释放，如先分配者先释放。



## 9.6.1 内存管理器

### ■ 内存管理器的设计目标

- **空间效率**。内存管理器应该使程序所需堆区空间的总量达到最小，以便在某个固定大小的虚拟地址空间中运行更大的程序。方法：**减少内存碎片**的数量。
- **程序效率**。内存管理器应该充分利用内存子系统，以便使程序运行得更快。方法：**利用程序的“局部性”**，即程序在访问内存时具有非随机性聚集的特性。
- **低开销**。由于存储分配和回收在很多程序中都是常用的操作，所以内存管理器应该尽可能地提高这些操作的执行效率，也就是说要尽可能地降低它们的开销。

## 9.6.2 内存体系

- 要想做好内存管理和编译器优化，首先要充分了解内存的工作机理。
- 内存访问时间的巨大差异来源于硬件技术的局限。

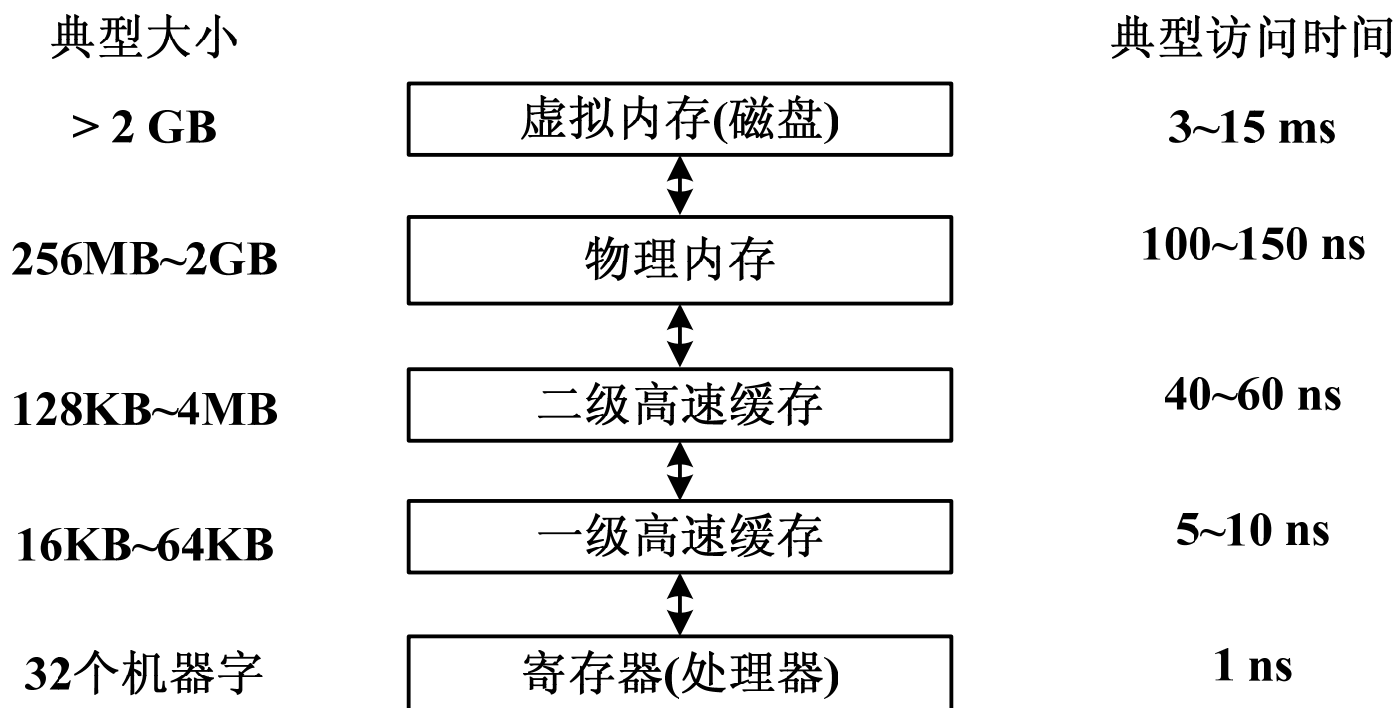


图9.21 典型的内存体系





## 9.6.3 程序中的局部性

---

### ■ 程序的局部性

- 程序中的大部分运行时间都花费在较少的一部分代码中，而且只是涉及到一小部分数据。
- 时间局部性：如果某个程序访问的内存位置有可能在很短的时间内被再次访问
- 空间局部性：如果被访问过的内存位置的邻近位置有可能在很短的时间内被再次访问



## 9.6.3 程序中的局部性

---

- 程序的局部性使得我们可以充分利用图9.21所示的内存层次结构，即将最常用的指令和数据放在快而小的内存中，而将其余部分放在慢而大的内存中，这将显著降低程序的平均内存访问时间
- 很多程序在对指令和数据的访问方式上既表现出时间局部性，又表现出空间局部性



## 9.6.3 程序中的局部性

---

- 虽然将最近使用过的数据放在最快的内存层中在普通程序中可以发挥很好的作用，但是在某些数据密集型的程序中的作用却并不明显，如循环遍历大数组的程序。将最近使用过的指令放在高速缓存中的策略一般都很有效。



## 9.6.3 程序中的局部性

---

- 空间局部性：让编译器将可能会连续执行的指令连续存放
  - 执行一条新指令时，其下一条指令也很有可能被执行
  - 属于同一个循环或同一个函数的指令也极可能被一起执行



## 9.6.3 程序中的局部性

---

- 通过改变数据布局或计算顺序也可以改进程序中数据访问的时间局部性和空间局部性
  - 例如，当某些程序反复访问大量数据而每次访问只完成少量计算时，它们的性能就不会很好。我们可以每次将一部分数据从内存层次中的较慢层加载到较快层，并趁它们处于较快层时执行所有针对这些数据的运算，这必将大大提高程序的性能。

## 9.6.4 降低碎片量的堆区空间管理策略

- 空闲块又被称为孔洞
- 如果对孔洞的使用不加管理的话，空闲的内存空间最终就会变成若干碎片，即大量不连续且很小的孔洞。此时就会出现这样的情况：尽管总的空闲空间足够大，却找不到一个足够大的孔洞来满足某个即将到来的分配请求。
- 1. 堆区空间分配策略
- 2. 空闲空间管理策略



# 堆区空间分配策略

---

## ■ 最佳适应策略

- 总是将请求的内存分配在满足要求的最小可用孔洞中
- 倾向于将大孔洞预留起来以便用来满足后续的更大请求，这是令程序产生最少碎片的一种很好的堆分配策略



# 堆区空间分配策略

---

## ■ 首次适应策略

- 将请求的内存分配在第一个满足要求的孔洞中
- 这种策略在分配空间时所花费的时间较少，但在总体性能上要低于最佳适应策略





# 堆区空间分配策略

- 如果将空闲块按大小不同放入不同的桶中，则可以更有效地实现最佳适应策略
- 桶机制更容易按最佳适应策略找到所需的空闲块：
  - 如果被请求的空闲块尺寸对应有一个专用桶，，则从该桶中任意取出一个空闲块即可
  - 如果被请求的空闲块尺寸没有对应的专用桶，则找一个允许存放所需尺寸空闲块的桶。在桶中使用首次适应策略或者最佳适应策略寻找满足要求的空闲块
  - 如果目标桶是空的，或者桶中没有满足要求的空闲块，则需要在稍大的桶中进行搜索



# 空闲空间管理策略

- 如果通过手工方式释放某个对象所占用的内存块，则内存管理器必须将其设置为空闲的，以便它可以被再次分配。为了减少碎片的产生，如果回收的内存块在堆中的相邻块也是空闲的，则需要将它们合并成更大的空闲块。
- 可以使用下面的两种数据结构来接合相邻的空闲块
  - 边界标签
  - 双向链接的空闲块列表



# 空闲空间管理策略

---

## ■ 边界标签

- 在每个内存块(已用/空闲)的高低两端均设置一个 **free/used** 标签位，用来标识该块是已用(**used**)还是空闲(**free**)，在与 **free/used** 位相邻的位置上则存放该块的字节总数。



# 空闲空间管理策略

## ■ 双向链接的空闲块列表

- 各个空闲块还由一个双向链表链接起来。链表的指针就保存在这些空闲块中，如紧挨某一端边界标签的位置上。于是我们不需要额外的空间来存放该空闲块链表，当然这会给空闲块的大小设置一个下界，即空闲块必须保存两个边界标签和两个指针，即使要保存的对象只有一个字节也得这样。空闲块链表中块的顺序留待用户确定，譬如，可以按块的大小来排序，这样可以支持最佳适应策略。



## 9.6.5 人工回收请求

### 1. 人工回收面临的问题

- ①一直未能删除不再被引用的数据，又称**内存泄露**；  
自动垃圾回收通过回收所有的垃圾来消除内存泄露问题。
- ②引用已被删除的数据，又称**悬空引用**。  
把沿着某个指针试图使用它所指向的对象的各种操作(如读、写、回收等)都称为对该指针的去引用。
- ③还有一种可能的错误形式就是**访问非法地址**。  
常见的例子包括对空指针的去引用和访问一个超出下标界限的数组元素。存在安全隐患，可以让编译器在每次访问中插入检查代码，以便保证此次访问不会越界。



## 2. 编程规范和工具

---

- 编程规范和工具可以协助程序员应对存储管理的复杂性

(1)Rational的Purify可以帮助程序员寻找程序中的内存访问错误和内存泄露。



## 2. 编程规范和工具

---

(2)当某个对象的生命周期可以被静态地推导出来时，对象所有者的概念将会非常有用。

其基本思想是在任何时候都给每个对象关联上一个所有者。该所有者是指向该对象的一个指针，通常属于某个函数调用。所有者(即该函数)负责删除该对象或者把该对象传递给另一个所有者。该规范可以消除内存泄露，同时也避免将同一对象删除两次。但它对解决悬空引用没有什么帮助，因为沿着某个不代表拥有关系的指针可以访问某个已经被删除的对象。



## 2. 编程规范和工具

(3)当某个对象的生命周期需要动态确定时，引用计数将会很有帮助。

其基本思想是给每个动态分配的对象附加一个计数。创建指向该对象的引用时就将该对象的引用计数加一，删除其某个引用时则将其引用计数减一。当某个对象的引用计数变成0时表明该对象将不会再被引用，可以将其删除。然而，该技术不能发现无用的循环数据结构，即使其中的某组对象不会再被引用，但由于它们之间互相引用的原因，其引用计数也不会变成0。因为不存在指向已删除对象的引用，所以引用计数技术可以根除所有的悬空引用。不过，由于引用计数在保存指针的每次运算上增加了额外的开销，所以其运行时代价很大。





## 2. 编程规范和工具

---

(4)对于其生命周期局限于计算过程中某个特定阶段的对象，可以使用基于区域的分配方法。当被创建的对象只在某计算过程的某个步骤中使用时，我们可以把这些对象分配在同一个区域中。一旦该计算步骤完成后，我们就删除整个区域。基于区域的分配方法具有一定的局限性，但当其可用时又非常高效，这是因为该技术可以成批地一次性删除区域中的所有对象。



# 本章小结

---

- 存储组织与管理是编译系统的重要组成部分，用来实现目标程序的存储组织与分配。根据程序设计语言的要求，有静态管理策略和动态管理策略
- 名字的绑定要体现名字的声明和作用域约定
- 过程调用中参数的传递方式分为传值、传地址、传值结果和传名4种
- 典型的运行时内存空间包括目标代码、静态数据区和动态数据区



# 本章小结

---

- 对编译时就能确定其运行时所需要的存储空间的对象实行静态存储分配，对编译时无法确定过程何时运行、运行时所需要存储空间的对象实行动态存储分配
- 静态存储管理方式支持较高的运行效率，利用适当的策略可以提高内存的利用率，但是无法支持动态数据和“过程”的递归调用



# 本章小结

---

- 栈式动态存储管理策略针对过程的每一次调用在栈顶创建一个栈单元用来存放这次过程调用产生的活动的数据区，访问链和display表可以用来实现相关数据的访问。这种方式支持过程的递归调用
- 堆管理解决活动结束后数据仍需有效的为题，但需要使用内存管理器实现对孔洞的管理
- 对内存的层次体系的有效利用有助于提高目标代码的运行效率



*School of Computer Science & Technology*  
*Harbin Institute of Technology*



# 第十章 代码优化

**重点：** 代码优化的任务，局部优化、循环优化、全局优化的基本方法。

**难点：** 控制流分析，数据流分析。





# 第10章 代码优化

---

**10.1 优化的种类**

**10.2 控制流分析**

**10.3 数据流分析**

**10.4 局部优化**

**10.5 循环优化**

**10.6 全局优化**

**10.6 本章小结**



# 第10章 代码优化

- 代码优化就是为了提高目标程序的效率，对程序进行等价变换，亦即在保持功能不变的前提下，对源程序进行合理的变换，使得目标代码具有更高的时间效率和/或空间效率。
- 空间效率和时间效率有时是一对矛盾，有时不能兼顾。
- 优化的要求：
  - 必须是等价变换(保持功能)
  - 为优化的努力必须是值得的。
  - 有时优化后的代码的效率反而会下降。



# 代码优化程序的结构

控制流分析



数据流分析



代码变换

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





# 10.1 优化的种类

---

## ■ 机器相关性

- 机器相关优化：寄存器优化，多处理器优化，特殊指令优化，无用指令消除等。
- 机器无关优化：

## ■ 优化范围

- 局部优化：单个基本块范围内的优化，常量合并优化，公共子表达式删除，计算强度削弱和无用代码删除。
- 全局优化：主要是基于循环的优化：循环不变优化，归纳变量删除，计算强度削减。

## ■ 优化语言级

- 优化语言级：针对中间代码，针对机器语言。



# 程序例子

## 本节所用的例子

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

```
(1) i := m - 1  
(2) j := n  
(3) t1 := 4 * n  
(4) v := a[t1]  
(5) i := i + 1  
(6) t2 := 4 * i  
(7) t3 := a[t2]  
(8) if t3 < v goto(5)
```



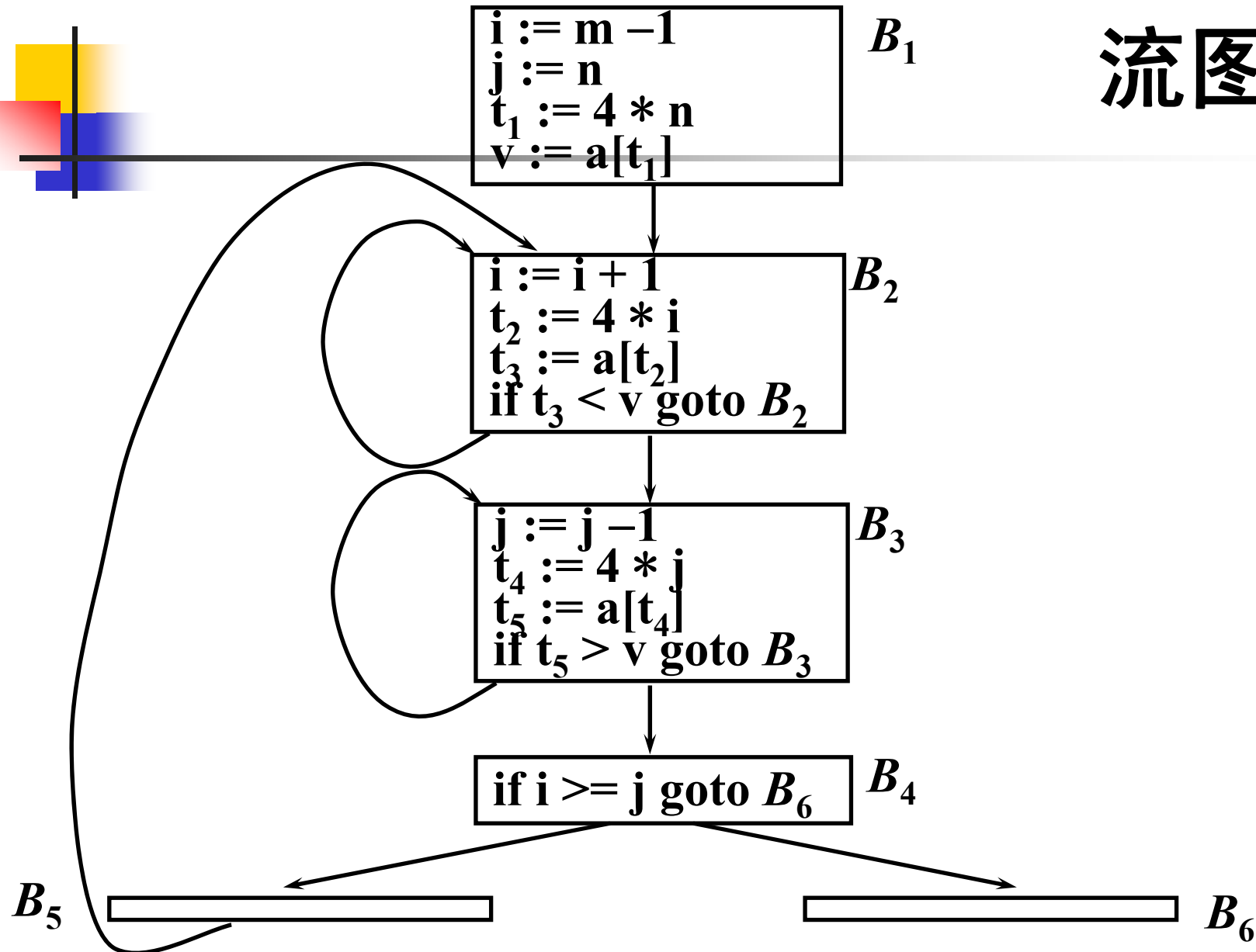
# 程序例子

## 本节所用的例子

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

```
(9) j := j - 1  
(10) t4 := 4 * j  
(11) t5 := a[t4]  
(12) if t5 > v goto(9)  
(13) if i >= j goto(23)  
(14) t6 := 4 * i  
(15) x := a[t6]  
. . .
```

# 流图





## 10.1.1 公共子表达式删除

### 局部公共子表达式

$B_5$   $x=a[i]; a[i]=a[j]; a[j]=x;$

```
t6 := 4 * i  
x := a[t6]  
t7 := 4 * i  
t8 := 4 * j  
t9 := a[t8]  
a[t7] := t9  
t10 := 4 * j  
a[t10] := x  
goto B2
```



## 10.1.1 公共子表达式删除

### 局部公共子表达式

$B_5$   $x=a[i]; a[i]=a[j]; a[j]=x;$

```
t6 := 4 * i  
x := a[t6]  
t7 := 4 * i  
t8 := 4 * j  
t9 := a[t8]  
a[t7] := t9  
t10 := 4 * j  
a[t10] := x  
goto B2
```



## 10.1.1 公共子表达式删除

### 局部公共子表达式

$B_5$   $x = a[i]; a[i] = a[j]; a[j] = x;$

```
t6 := 4 * i
x := a[t6]
t7 := 4 * i
t8 := 4 * j
t9 := a[t8]
a[t7] := t9
t10 := 4 * j
a[t10] := x
goto B2
```

```
t6 := 4 * i
x := a[t6]
t8 := 4 * j
t9 := a[t8]
a[t6] := t9
a[t8] := x
goto B2
```



## 10.1.1 公共子表达式删除

### 全局公共子表达式

$B_5$   $x = a[i]; a[i] = a[j]; a[j] = x;$

```
t6 := 4 * i
x := a[t6]
t7 := 4 * i
t8 := 4 * j
t9 := a[t8]
a[t7] := t9
t10 := 4 * j
a[t10] := x
goto B2
```

```
t6 := 4 * i
x := a[t6]
t8 := 4 * j
t9 := a[t8]
a[t6] := t9
a[t8] := x
goto B2
```



## 10.1.1 公共子表达式删除

### 全局公共子表达式

$B_5$   $x = a[i]; a[i] = a[j]; a[j] = x;$

```
t6 := 4 * i  
x := a[t6]  
t7 := 4 * i  
t8 := 4 * j  
t9 := a[t8]  
a[t7] := t9  
t10 := 4 * j  
a[t10] := x  
goto B2
```

```
t6 := 4 * i  
x := a[t6]  
t8 := 4 * j  
t9 := a[t8]  
a[t6] := t9  
a[t8] := x  
goto B2
```

```
x := a[t2]  
t9 := a[t4]  
a[t2] := t9  
a[t4] := x  
goto B2
```



## 10.1.1 公共子表达式删除

$B_5$   $x = a[i]; a[i] = a[j]; a[j] = x;$

```
t6 := 4 * i  
x := a[t6]  
t7 := 4 * i  
t8 := 4 * j  
t9 := a[t8]  
a[t7] := t9  
t10 := 4 * j  
a[t10] := x  
goto B2
```

```
t6 := 4 * i  
x := a[t6]  
t8 := 4 * j  
t9 := a[t8]  
a[t6] := t9  
a[t8] := x  
goto B2
```

```
x := a[t2]  
t9 := a[t4]  
a[t2] := t9  
a[t4] := x  
goto B2
```



## 10.1.1 公共子表达式删除

$B_5$   $x = a[i]; a[i] = a[j]; a[j] = x;$

```
t6 := 4 * i
x := a[t6]
t7 := 4 * i
t8 := 4 * j
t9 := a[t8]
a[t7] := t9
t10 := 4 * j
a[t10] := x
goto B2
```

```
t6 := 4 * i
x := a[t6]
t8 := 4 * j
t9 := a[t8]
a[t6] := t9
a[t8] := x
goto B2
```

```
x := a[t2]
t9 := a[t4]
a[t2] := t9
a[t4] := x
goto B2
```

```
x := t3
a[t2] := t5
a[t4] := x
goto B2
```



## 10.1.1 公共子表达式删除

---

$B_6$   $x = a[i]; a[i] = a[n]; a[n] = x;$

$t_{11} := 4 * i$   
 $x := a[t_{11}]$   
 $t_{12} := 4 * i$   
 $t_{13} := 4 * n$   
 $t_{14} := a[t_{13}]$   
 $a[t_{12}] := t_{14}$   
 $t_{15} := 4 * n$   
 $a[t_{15}] := x$

$x := t_3$   
 $t_{14} := a[t_1]$   
 $a[t_2] := t_{14}$   
 $a[t_1] := x$



## 10.1.1 公共子表达式删除

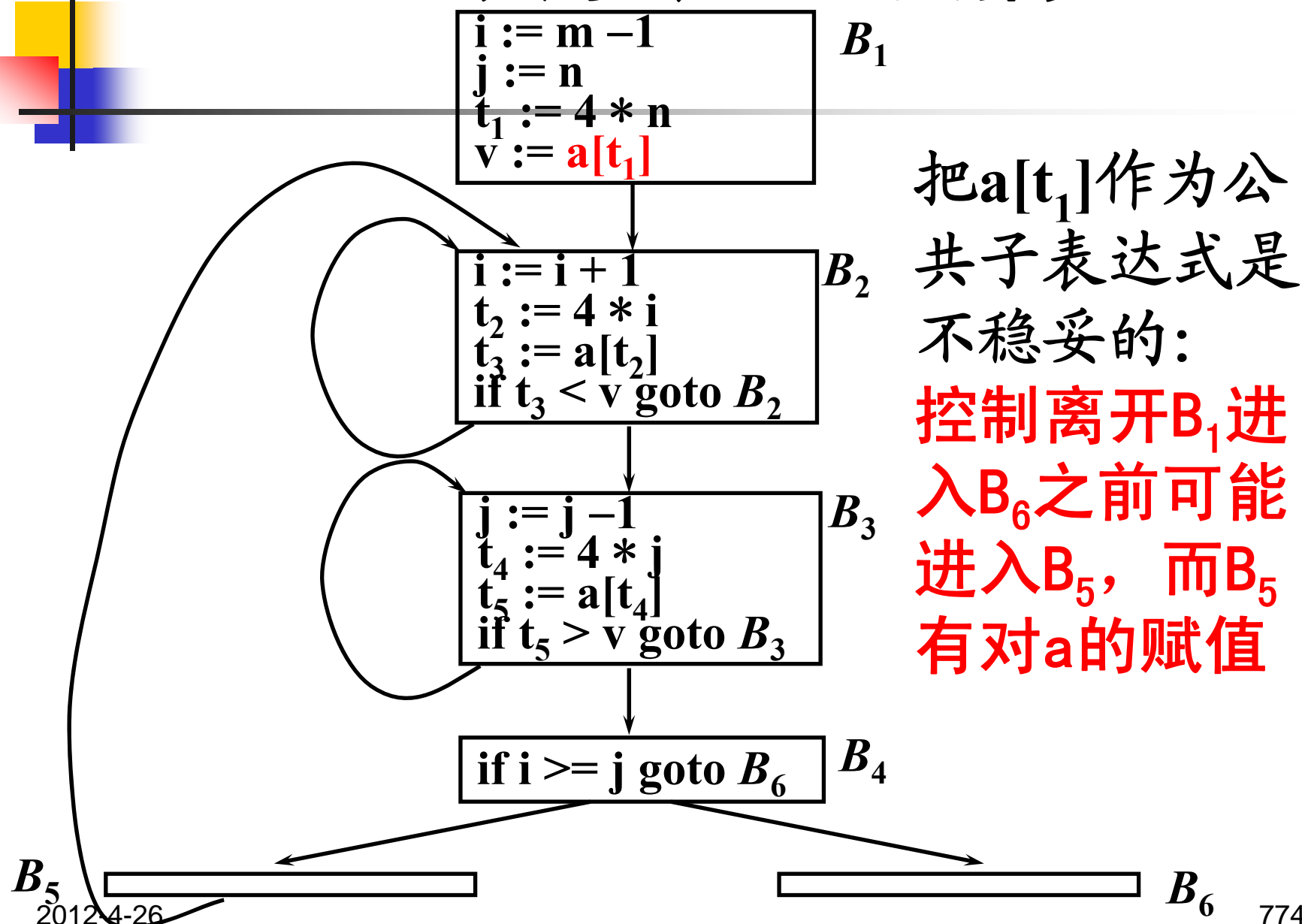
$B_6$   $x = a[i]; a[i] = a[n]; a[n] = x;$

$a[t_1]$ 能否作为公共子表达式?

```
t11 := 4 * i
x := a[t11]
t12 := 4 * i
t13 := 4 * n
t14 := a[t13]
a[t12] := t14
t15 := 4 * n
a[t15] := x
```

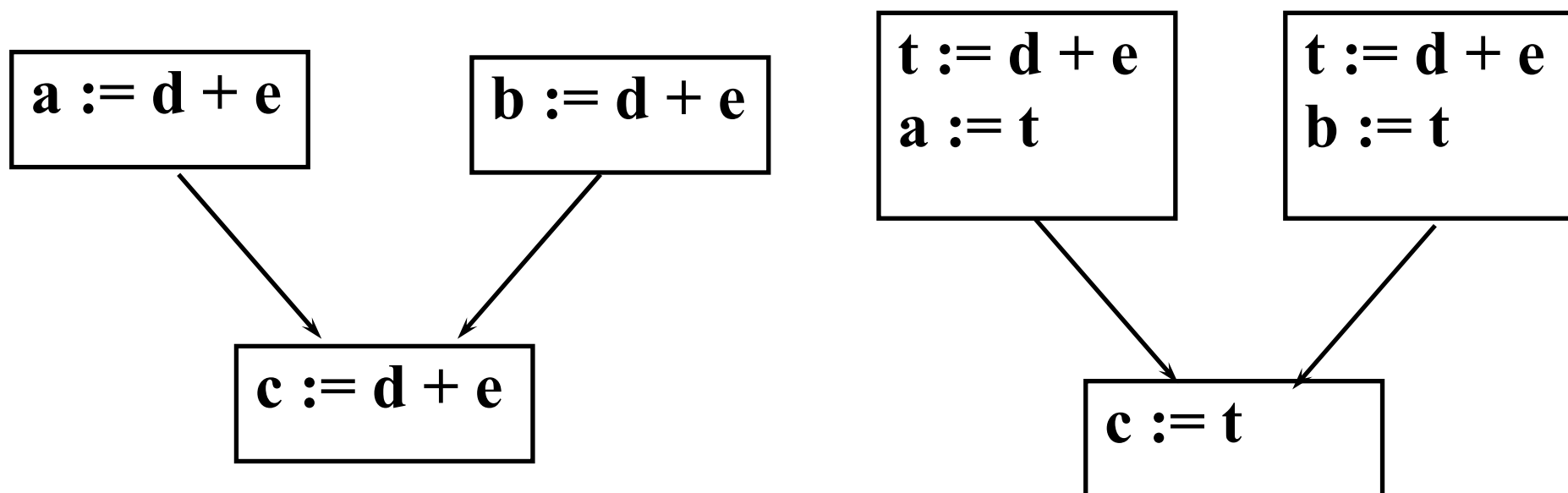
```
x := t3
t14 := a[t1]
a[t2] := t14
a[t1] := x
```

## 10.1.1 公共子表达式删除



## 10.1.2 复制传播

- 形如  $f := g$  的赋值语句叫做复制语句
- 优化过程中会大量引入复制



删除局部公共子表达式期间引进复制



## 10.1.2 复制传播

- 复制传播变换的思想是在复制语句  $f := g$  之后尽可能用  $g$  代替  $f$
- 复制传播变换本身并不是优化，但它给其它优化带来机会
  - 无用代码删除

```
x := t3  
a[t2] := t5  
a[t4] := x  
goto B2
```

```
x := t3  
a[t2] := t5  
a[t4] := t3  
goto B2
```





## 10.1.3 无用代码删除

---

- 无用代码是指计算结果以后不被引用的语句
- 一些优化变换可能会引入无用代码
- 例:

**debug := true;**

**. . .**          测试后改成

**if(debug)print ...**

**debug := false;**

**. . .**

**if(debug)print ...**



## 10.1.3 无用代码删除

- 无用代码是指计算结果以后不被引用的语句
- 一些优化变换可能会引入无用代码

例：复制传播可能会引入无用代码

```
x := t3  
a[t2] := t5  
a[t4] := t3  
goto B2
```

```
a[t2] := t5  
a[t4] := t3  
goto B2
```



## 10.1.4 代码外提

- 结果独立于循环执行次数的表达式称为循环不变计算。如果将循环不变计算从循环中移出到循环的前面，将会减少循环执行的代码总数，大大提高代码的执行效率。这种与循环有关的优化方法称为代码外提。
- 例如，下面的while语句中，`limit-2`就是循环不变计算。
  - `while(i <= limit-2 ) { /*假设循环体中的语句不改变limit的值 */ }`
  - 代码外提将生成如下的等价语句：
  - `t := limit-2;`
  - `while (i <= t) { /*假设循环体中的语句不改变limit或t*/ }`



## 10.1.5 强度削弱

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

## 10.1.5 强度削弱

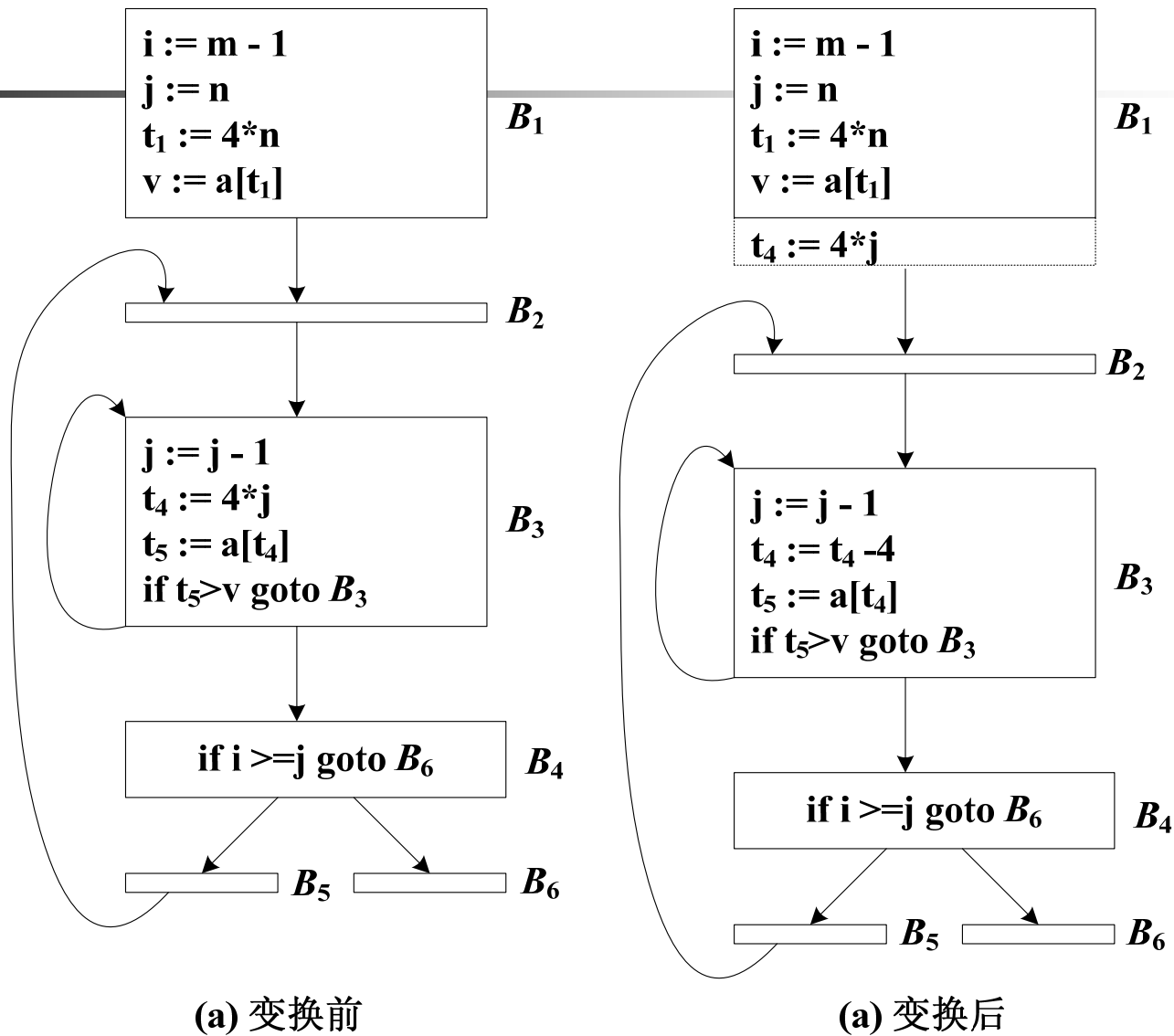


图10.6 将强度削弱应用到块 $B_3$ 中的 $4*j$

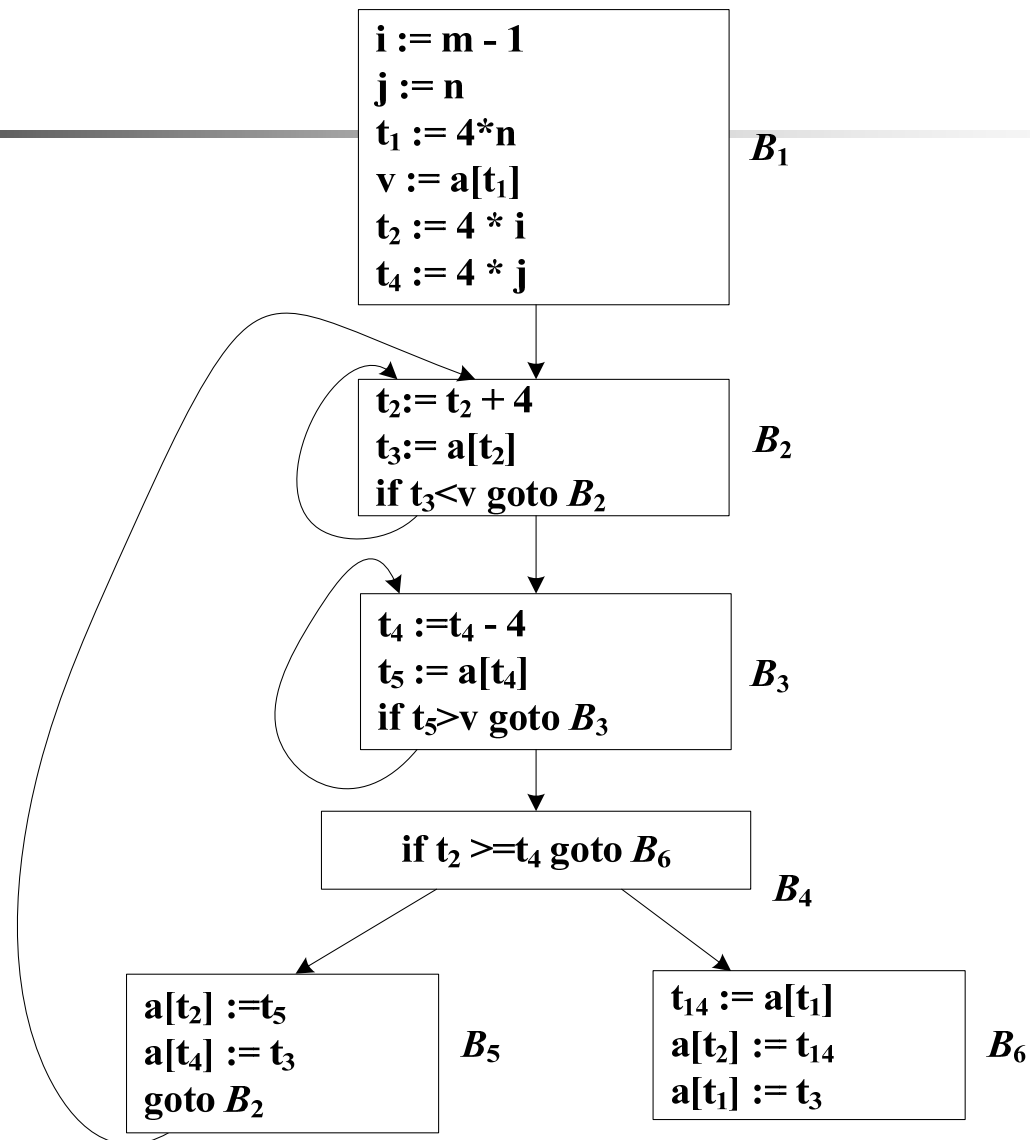


## 10.1.5 归纳变量删除

---

- 在围绕 $B3$ 的循环的每次迭代中， $j$ 和 $t4$ 的值总是同步变化，每次 $j$ 的值减1， $t4$ 的值就减4，这是因为 $4*j$ 赋给了 $t4$ ，我们将这样的变量称为归纳变量
- 如果循环中存在两个或更多的归纳变量，也许可以只保留一个，而去掉其余的，以便提高程序的运行效率。

## 10.1.5 归纳变量删除





## 10.2 控制流分析

---

- 为了对程序进行优化，尤其是对循环进行优化，必须首先分析程序中的控制流程，以便找出程序中的循环结构，这是控制流分析的主要任务。
- 为此，首先需要将程序划分为基本块集合并转换成流图，然后再从流图中找出循环。





## 10.2.1 基本块

---

- 基本块(basic block)是一个连续的语句序列，控制流从它的开始进入，并从它的末尾离开，中间不存在中断或分支(末尾除外)。下面的三地址码序列就形成了一个基本块：

- $t1 := a * a$
- $t2 := a * b$
- $t3 := 2 * t2$
- $t4 := t1 + t3$
- $t5 := b * b$
- $t6 := t4 + t5$



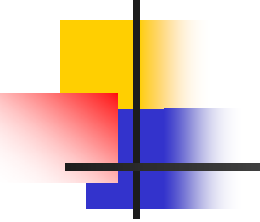
# 基本块的划分算法

- 算法10.1 基本块的划分。
- 输入：一个三地址码序列；
- 输出：一个基本块列表，其中每个三地址码仅在一个块中；
- 步骤：
  - 1. 首先确定所有的入口(leader)语句，即基本块的第一个语句。确定入口的规则如下：
    - a) 第一个语句是入口语句；
    - b) 任何能由条件转移语句或无条件转移语句转移到的语句都是入口语句；
    - c) 紧跟在转移语句或条件转移后面的语句是入口语句。
  - 2. 对于每个入口语句，其基本块由它和直到下一个入口语句(但不含该入口语句)或程序结束为止的所有语句组成。



## 10.2.2 流图

- 程序的控制流信息可以用流图表示，流图是一个节点为基本块的有向图。
- 以程序的第一个语句作为入口语句的节点称为**初始节点**。
- 如果在某个执行序列中 $B_2$ 跟随在 $B_1$ 之后，则从 $B_1$ 到 $B_2$ 有一条有向边。
- 如果从 $B_1$ 的最后一条语句有条件或无条件转移到 $B_2$ 的第一个语句；或者按程序正文的次序 $B_2$ 紧跟在 $B_1$ 之后，并且 $B_1$ 不是结束于无条件转移，则称 $B_1$ 是 $B_2$ 的**前驱**，而 $B_2$ 是 $B_1$ 的**后继**。



## 10.2.3 循环

- 流图中的**循环**就是具有唯一入口的强连通子图，而且从循环外进入循环内，必须首先经过循环的入口节点。
- 如果从流图的初始节点到节点 $n$ 的每条路径都要经过节点 $d$ ，则说节点 $d$ **支配**(dominate)节点 $n$ ，记作 $d \text{ dom } n$ ， $d$ 又称为 $n$ 的**必经节点**。
- 根据该定义，每个节点都支配它自身，而循环的入口节点则支配循环中的所有节点。



# 支配节点集计算

算法10.2 支配节点集计算。

输入：流图  $G$ ，其节点集为  $N$ ，边集为  $E$ ，初始节点为  $n_0$

输出：关系  $dom$ ;

步骤：

1.  $D(n_0) := \{n_0\};$
2. for  $N - \{n_0\}$  中的  $n$  do  $D(n) := N;$   
    /\* 初始化完毕 \*/
3. while  $D(n)$  发生变化 do
4.   for  $N - \{n_0\}$  中的  $n$  do
5.      $D(n) := \{n\} \cup \bigcap_{n \text{ 的前驱 } p} D(p);$

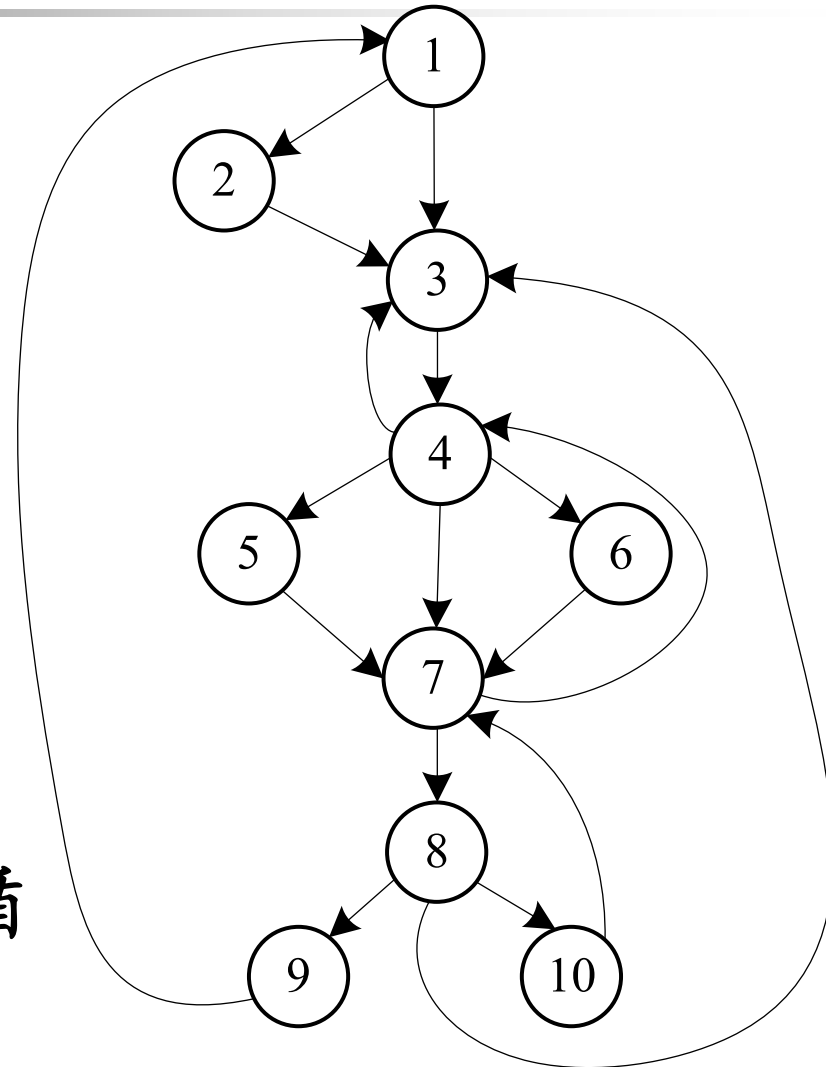


# 循环

- 循环的性质：
  - 循环必须有唯一的入口点，称为首节点(header)。首节点支配循环中的所有节点。
  - 循环至少迭代一次，亦即至少有一条返回首节点的路径。
- 为了寻找流图中的循环，必须寻找可以返回到循环入口节点的有向边，这种边叫做回边(back edge)，如果 $b \in \text{dom } a$ ，则边 $a \rightarrow b$ 称为回边。利用支配节点集可以求出流图中的所有回边。
- 给定一条回边 $n \rightarrow d$ ，定义该边的自然循环(natural loop)为 $d$ 加上所有不经过 $d$ 而能到达 $n$ 的节点集合。 $d$ 是该循环的首节点。

## 例10.6

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





# 自然循环的构造

算法10.3 构造回边的自然循环。

输入：流图  $G$  和回边  $n \rightarrow d$ ;

输出：由  $n \rightarrow d$  的自然循环中所有节点构成的集合  $loop$ ;

**procedure** *insert*( $m$ );

**if**  $m$  不在  $loop$  中 **then begin**     $loop := loop \cup \{m\}$ ;

  将  $m$  压入栈  $stack$     **end**;

/\* 下面是主程序 \*/

$stack :=$  空;     $loop := \{d\}$ ;    *insert*( $n$ );

**while**  $stack$  非空 **do begin**

  从  $stack$  弹出第一个元素  $m$ ;

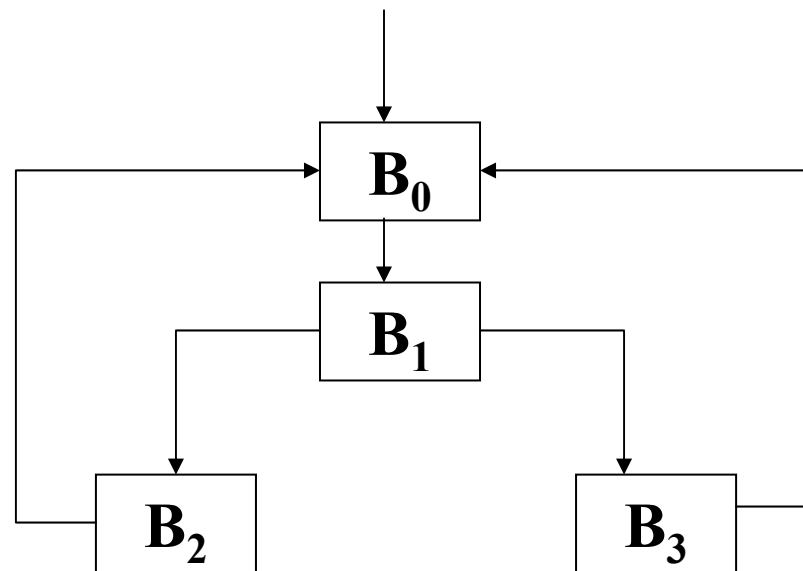
**for**  $m$  的每个前驱  $p$  **do** *insert*( $p$ )

**end**



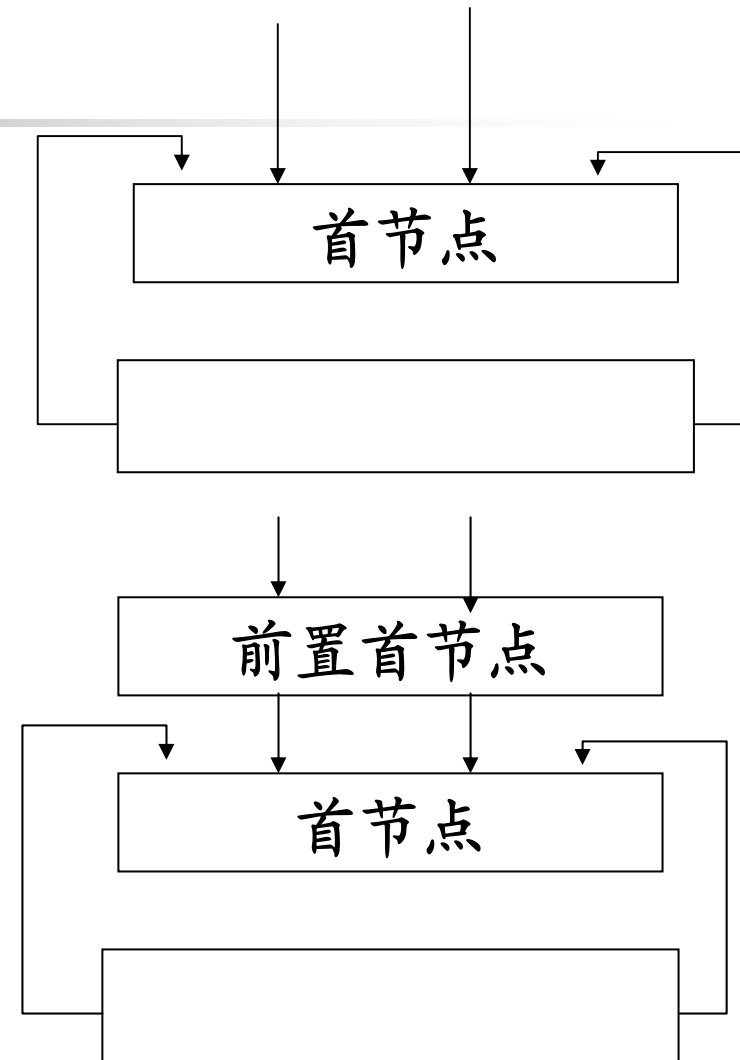
# 循环

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



# 循环

- 某些变换要求我们将循环中的某些语句移到首节点的前面。因此，在开始处理循环 $L$ 之前，往往需要先创建一个称为前置首节点的新块。前置首节点的唯一后继是 $L$ 的首节点，原来从 $L$ 外到达 $L$ 首节点的边都将改成进入前置首节点，从循环 $L$ 里面到达首节点的边不改变。
- 初始时前置首节点为空，但对 $L$ 的变换可能会将一些语句放到该节点中。



# 可归约流图

- **可归约流图**：一个流图 $G$ 是可约的，当且仅当可以把它的边分成两个不相交的组，其中一组仅含回边，另一组的边都不是回边，这些边被称为前向边，所有前向边形成一个无环有向图，在该图中，每个节点都可以从 $G$ 的初始节点到达。

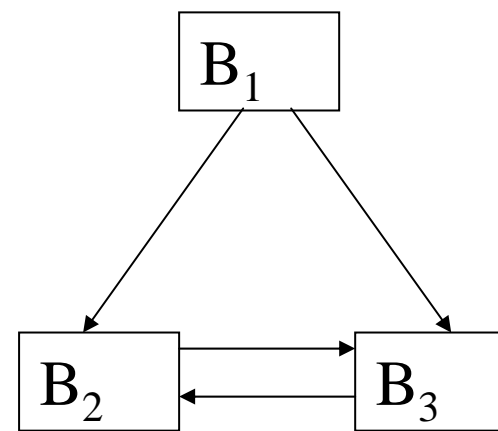


图10.10 一个不可约流图

- 循环的可约流图具有一个重要的性质，就是当流图中的一些节点被看作是循环的节点集合时，这些节点之间一定包含一条回边。于是，为了找出流图可约程序中的所有循环，只要检查回边的自然循环即可。



## 10.3 数据流分析

---

- 为了进行代码优化，编译器必须掌握程序中变量的定义和引用情况，有了这些信息才能对源程序进行合适的等价变换。
  - 例如，了解每个基本块的出口处哪些变量是活跃的可以改进寄存器的利用率，执行常量合并和无用代码删除则需要利用变量的“到达-定义”信息。
- 对程序中变量的定义和引用关系的分析称为数据流分析



# 数据流分析的种类

---

- 到达-定义分析
- 活跃变量分析
- 可用表达式分析



## 10.3.1 数据流方程的一般形式

- 下面的方程叫做数据流方程：

- $out[S] = gen[S] \cup (in[S] - kill[S])$  (10.3)

该方程的含义是，当控制流通过一个语句 $S$ 时，在 $S$ 末尾得到的信息( $out[S]$ )或者是在 $S$ 中产生的信息( $gen[S]$ )，或者是进入 $S$ 开始点时携带的、并且没有被 $S$ 注销的那些信息( $in[S]$ 表示进入 $S$ 开始点时携带的信息， $kill[S]$ 表示被 $S$ 注销的信息)。

- 也可以根据 $out[S]$ 来定义 $in[S]$

$$in[S] = (out[S] - kill[S]) \cup gen[S] \quad (10.4)$$



## 10.3.1 数据流方程的一般形式

- 不同的问题方程的意义可能有所不同，主要可由以下两点来区别：
  - 信息流向问题。根据信息流向可以将数据流分析问题分为正向和反向两类，正向的含义是根据 $in$ 集合来计算 $out$ 集合，反向则是从 $out$ 集合来计算 $in$ 集合。
  - 聚合操作问题。所谓聚合操作，是指当有多条边进入某一基本块 $B$ 时，由 $B$ 的前驱节点的 $out$ 集计算 $in[B]$ 时采用的集合操作(并或交)。到达-定义等方程采用并操作，全局可用表达式采用的则是交操作。



## 10.3.1 数据流方程的一般形式

- 在基本块中，将相邻语句间的位置称为点，第一个语句前和最后一个语句后的位置也称为点。从点 $p_1$ 到点 $p_n$ 的路径是这样的点序列 $p_1, p_2, \dots, p_n$ ，对于 $\forall i(1 \leq i \leq n-1)$ 下列条件之一成立：
  - $p_i$ 是紧接在一个语句前面的点， $p_{i+1}$ 是同一块中紧跟在该语句后面的点；
  - $p_i$ 是某基本块的结束点，而 $p_{i+1}$ 是后继块的开始点。
- 为简单起见，假设控制只能从一个开始点进入基本块，而当基本块结束时控制只能从一个结束点离开。





## 10.3.2 到达-定义分析

- 变量x的定义是一条赋值或可能赋值给x的语句。最普通的定义是对x的赋值或从I/O设备读一个值并赋给x的语句，这些语句比较明确地为x定义了一个值，称为x的明确定义。也有一些语句只是可能为x定义一个值，称为x的含糊定义，其常见形式有：
  - 1. 以x为参数的过程调用(传值方式除外)或者可能访问x的过程；
  - 2. 通过可能指向x的指针对x赋值。



## 10.3.2 到达-定义分析

- 对于定义 $d$ ，如果存在一条从紧跟 $d$ 的点到 $p$ 的路径，并且在这条路径上 $d$ 没有被“注销”，则称定义 $d$ **到达**(reach)点 $p$ 。
- 如果沿着这条路径的某两点间存在 $a$ 的其它定义，则将**注销**(kill)变量 $a$ 的那个定义。注意，只有 $a$ 的明确定义才能注销 $a$ 的其它定义。
- 到达定义信息可以用引用-定义链(即ud-链)来保存，它是一个链表，对于变量的每次引用，到达该引用的所有定义都保存在该链表中。



## 10.3.2 到达-定义分析

- 如果块 $B$ 中在变量 $a$ 的引用之前没有任何 $a$ 的明确定义，那么 $a$ 的这次引用的ud-链为 $in[B]$ 中 $a$ 的定义的集合。如果 $B$ 中在 $a$ 的这次引用之前存在 $a$ 的明确定义，那么只有 $a$ 的最后一次定义会在ud-链中，而 $in[B]$ 不能放在ud-链中
- 另外，如果存在 $a$ 的含糊定义，那么所有那些在该定义和 $a$ 的这次引用之间没有 $a$ 的明确定义的定义都将被放在 $a$ 的这次引用的ud-链中
- 利用ud-链可以求出循环中的所有循环不变计算，常量传播也需要用到ud-链信息



# 到达定义数据流方程(记号)

- $\text{in}[B]$ : 表示基本块 $B$ 的入口点处各个变量的定义集合。
  - 如果 $B$ 中点 $p$ 之前有 $x$ 的定义 $d$ , 且这个定义能够到达 $p$ , 则点 $p$ 处 $x$ 的ud链是 $\{d\}$ 。
  - 否则, 点 $p$ 处 $x$ 的ud链就是 $\text{in}[B]$ 中 $x$ 的定义集合。
- $P[B]$ :  $B$ 的所有前驱基本块的集合。



# 到达定义数据流方程(记号)

---

- **gen[B]**: 各个变量在B内定义, 并能够到达B的出口点的所有定义的集合。
- **out[B]**: 各个变量的能够到达基本块B的出口点的所有定义的集合。
- **kill[B]**: 各个变量在基本块B中重新定义, 即在此块内部被注销的定义点的集合。



# 到达定义数据流方程

---

- $\text{in}[B] = \bigcup \text{out}[p]$  where  $p$  is in  $P[B]$
- $\text{out}[B] = \text{gen}[B] \cup (\text{in}[B] - \text{kill}[B])$
- 其中:
  - $\text{gen}[B]$  可以从基本块中求出: 使用 **DAG** 图就可以得到。
  - $\text{kill}[B]$  中, 对于整个流图中的所有  $x$  的定义点, 如果  $B$  中有对  $x$  的定义, 那么该定义点在  $\text{kill}[B]$  中。



# 方程求解算法

---

## ■ 使用迭代方法。

初始值设置为:  $\text{in}[B_i] = \text{空}$ ;  $\text{out}[B] = \text{gen}[B_i]$ ;

**change = true;**

**while(change)**

**{ change = false;**

**for each B do**

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

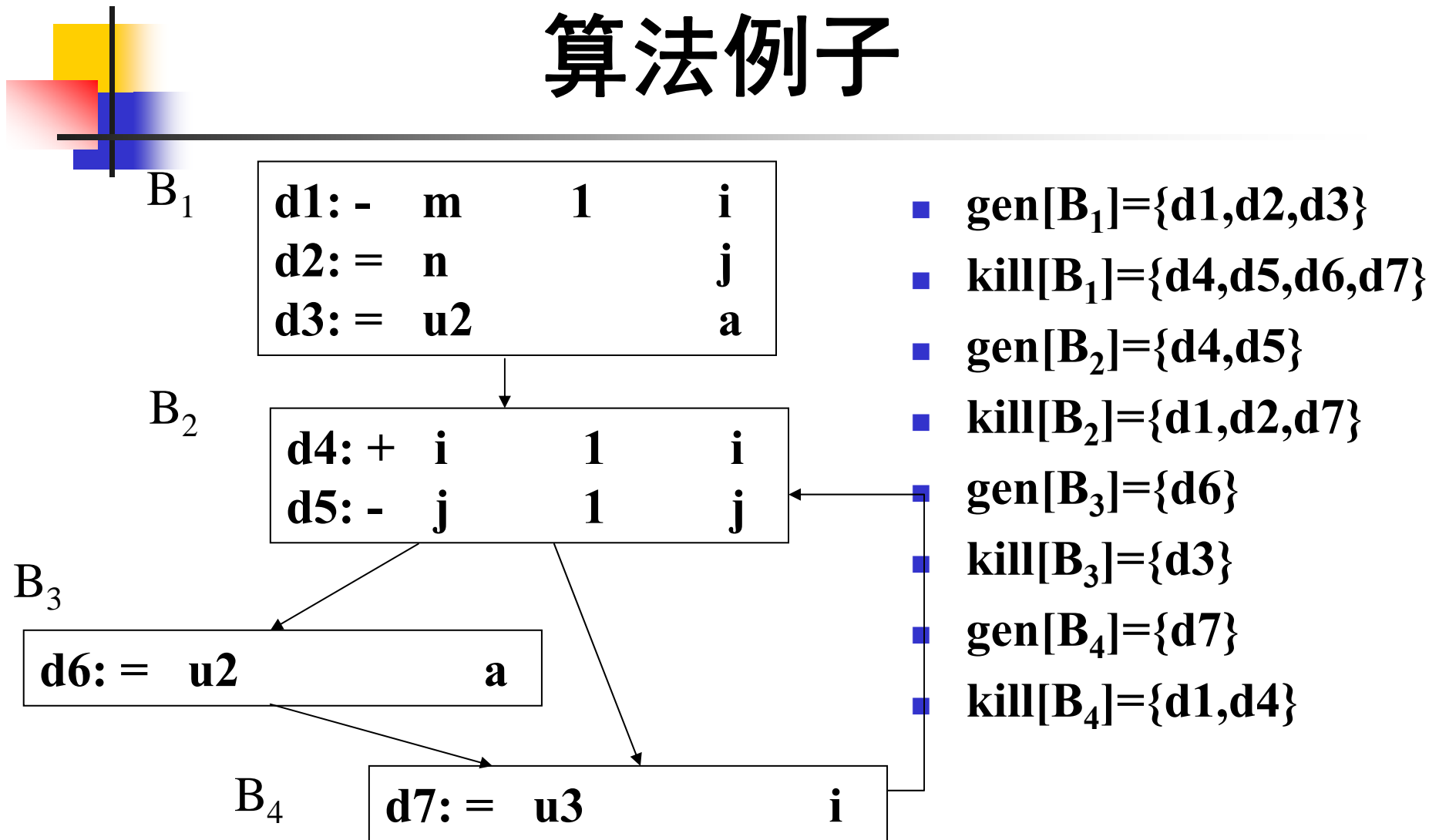
**$\text{out}[B] = \text{gen}[B] \bigcup (\text{in}[B] - \text{kill}[B]);$**

**oldout = out[B];**

**if(out[B] != oldout) change = true;}**

**}**

# 算法例子







# 计算过程

初始化:

- $\text{in}[B_1] = \text{in}[B_2] = \text{in}[B_3] = \text{in}[B_4] = \text{空}$
- $\text{out}[B_1] = \{d1, d2, d3\}, \text{out}[B_2] = \{d4, d5\}$
- $\text{out}[B_3] = \{d6\}, \text{out}[B_4] = \{d7\}.$

■ 第一次循环:

- $\text{in}[B_1] = \text{空}; \text{in}[B_2] = \{d1, d2, d3, d7\}; \text{in}[B_3] = \{d4, d5\};$
- $\text{in}[B_4] = \{d4, d5, d6\}; \text{out}[B_1] = \{d1, d2, d3\};$
- $\text{out}[B_2] = \{d3, d4, d5\} \dots$

■ 结果:

- $\text{in}[B_1] = \text{空}; \text{out}[B_1] = \{d1, d2, d3\};$
- $\text{in}[B_2] = \{d1, d2, d3, d5, d6, d7\}; \text{out}[B_2] = \{d3, d4, d5, d6\};$
- $\text{in}[B_3] = \{d3, d4, d5, d6\}; \text{out}[B_3] = \{d4, d5, d6\};$
- $\text{in}[B_4] = \{d3, d4, d5, d6\}; \text{out}[B_4] = \{d3, d5, d6, d7\};$



## 10.3.3 活跃变量分析

- 对于变量 $x$ 和点 $p$ ，在流图中沿从 $p$ 开始的某条路径，是否可以引用 $x$ 在 $p$ 点的值。如果可以则称 $x$ 在 $p$ 点是活跃的，否则， $x$ 在 $p$ 点就是无用的。
- 活跃变量信息在目标代码生成时具有重要的作用。当我们在寄存器中计算一个值之后，通常假设在某个块中还要引用它，如果它在该块的末尾是无用的，则不需要存储该值。
- 消除复制四元式的依据也是对活跃变量的分析。如果某个变量的值在以后不被引用，那么该复制四元式可以被消除。



# 记号

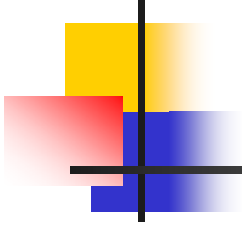
---

- **in[B]**: 基本块**B**的入口点的活跃变量集合。
- **out[B]**: 是在基本块**B**的出口点的活跃变量集。
- **def[B]**: 是在基本块**b**内的定义，但是定义前在**B**中没有被引用的变量的集合。
- **use[B]**: 表示在基本块中引用，但是引用前在**B**中没有被定义的变量集合。
- 其中，**def[B]**和**use[B]**是可以从基本块**B**中直接求得的量，因此在方程中作为已知量。



# 活跃变量数据流方程

- $\text{in}[B] = \text{use}[B] \cup (\text{out}[B] - \text{def}[B])$
- $\text{out}[B] = \bigcup \text{in}[S]$ , 其中S为B的所有后继。
- 变量在某点活跃, 表示变量在该点的值在以后会被使用。
- 第一个方程表示:
  - 一个变量在进入块B时是活跃的, 如果它在该块中于定义前被引用
  - 一个变量在离开块B时是活跃的, 而且在该块中没有被重新定义。
- 第二个方程表示:
  - 一个变量在离开块B时是活跃的, 当且仅当它在进入该块的某个后继时是活跃的。

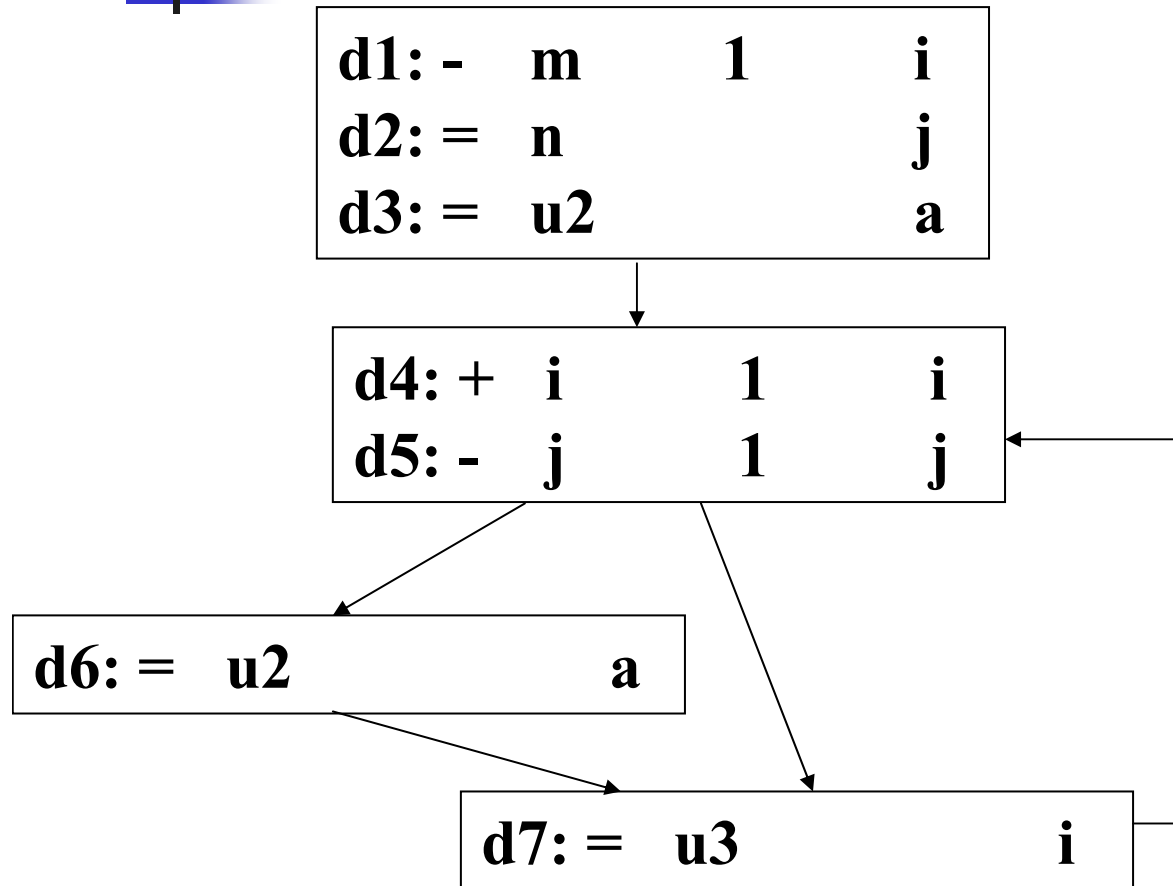


# 活跃变量数据流方程求解

---

- 设置初值:  $\text{in}[B_i] = \text{空}$ ;
- 重复执行以下步骤直到  $\text{in}[B_i]$  不再改变:  
    **for**( $i=1$ ;  $i < n$ ;  $i++$ )  
    {  
         $\text{out}[B_i] = \bigcup \text{in}[s]$ ;  $s$  是  $B_i$  的后继;  
         $\text{in}[B_i] = \text{use}[B_i] \bigcup (\text{out}[B_i] - \text{def}[B_i])$ ;  
    }

# 活跃变量数据流方程例子



- $\text{def}[B_1] = \{i, j, a\}$
- $\text{use}[B_1] = \{m, n, u1\}$
- $\text{def}[B_2] = \text{空}$
- $\text{use}[B_2] = \{i, j\}$
- $\text{def}[B_3] = \{a\}$
- $\text{use}[B_3] = \{u2\}$
- $\text{def}[B_4] = \{i\}$
- $\text{use}[B_4] = \{u3\}$



# 迭代过程

- 第一次循环:

- $\text{out}[B_1]=\text{空}$        $\text{in}[B_1]=\{m,n,u1\}$
- $\text{out}[B_2]=\text{空}$        $\text{in}[B_2]=\{i,j\}$
- $\text{out}[B_3]=\{i,j\}$      $\text{in}[B_3]=\{i,j,u2\}$
- $\text{out}[B_4]=\{i,j,u2\}$   $\text{in}[B_4]=\{j,u2,u3\}$

- 第二次循环:

- $\text{out}[B_1]=\{i,j,u2,u3\}$      $\text{in}[B_1]=\{m,n,u1,u2,u3\}$
- $\text{out}[B_2]=\{i,j,u2,u3\}$      $\text{in}[B_2]=\{i,j,u2,u3\}$
- $\text{out}[B_3]=\{i,j,u2,u3\}$      $\text{in}[B_3]=\{i,j,u2,u3\}$
- $\text{out}[B_4]=\{i,j,u2,u3\}$      $\text{in}[B_4]=\{j,u2,u3\}$

- 第三次循环各个值不再改变, 完成求解。



## 10.3.3 活跃变量分析

### ■ 定义-引用链

- 定义-引用链(简称du-链)是一种和活跃变量分析方式相同的数据流信息。定义-引用链的计算与引用-定义链的计算正好相反，它是从变量 $x$ 的某个定义点 $p$ 出发，计算该定义可以到达的所有引用 $s$ 的集合，所谓可以到达是指从 $p$ 到 $s$ 有一条没有重新定义 $x$ 的路径。

- 在代码优化过程中，无用代码删除、强度削弱、循环中的代码外提以及目标代码生成过程中的寄存器分配都要用到du-链信息。





## 10.3.4 可用表达式分析

- 如果从初始节点到 $p$ 的每一条路径(不必是无环路的)都要计算 $x \text{ op } y$ , 而且在到达 $p$ 的这些路径上没有对 $x$ 或 $y$ 的赋值, 则称**表达式 $x \text{ op } y$ 在 $p$ 点是可用的**。
- 对表达式的注销: 如果基本块 $B$ 中含有对 $x$ 或 $y$ 的赋值(或可能赋值), 而且后来没有重新计算 $x \text{ op } y$ , 则称 $B$ **注销**了表达式 $x \text{ op } y$ 。
- 表达式的生成: 如果基本块 $B$ 明确地计算了 $x \text{ op } y$ , 并且后来没有重新定义 $x$ 或 $y$ , 则称 $B$ **生成**了表达式 $x \text{ op } y$ 。
- 可用表达式信息的主要用途是检测公共子表达式。



# 记号

---

- **out[B]**: 在基本块出口处的可用表达式集合。
- **in[B]**: 在基本块入口处的可用表达式集合。
- **e\_gen[B]**: 基本块B生成的可用表达式的集合。
- **e\_kill[B]**: 基本块B注销的可用表达式的集合。
- **e\_gen[B]**和**e\_kill[B]**的值可以直接从流图计算出来，因此在数据流方程中，可以将**e\_gen[B]**和**e\_kill[B]**当作已知量看待。



# e\_gen[B]的计算

---

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

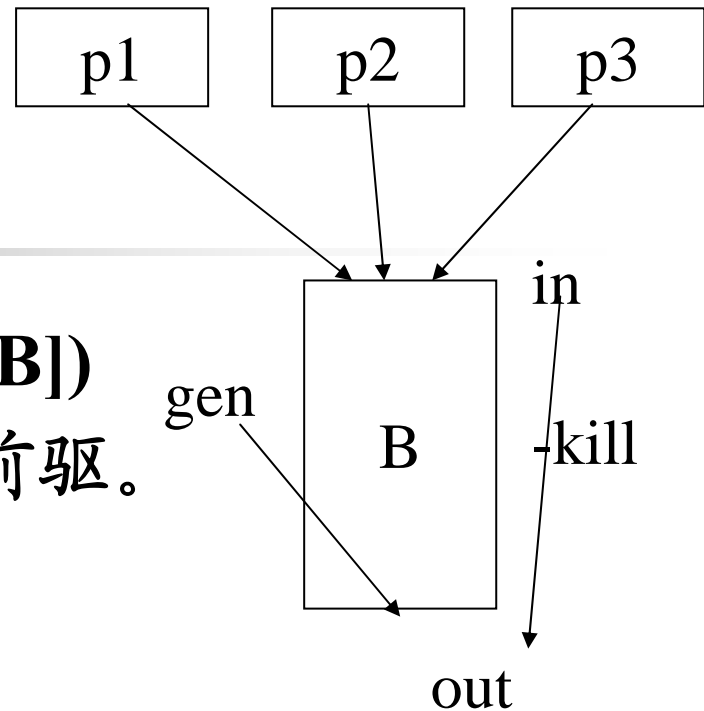


# e\_kill[B]的计算

---

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

# 数据流方程



1.  $\text{out}[B] = \text{e\_gen}[B] \cup (\text{in}[B] - \text{e\_kill}[B])$
2.  $\text{in}[B] = \bigcap \text{out}[p] \mid B \neq B_1, p \text{ 是 } B \text{ 的前驱。}$
3.  $\text{in}[B_1] = \text{空集}$

■ 说明:

- 在程序开始的时候, 无可用表达式。(3)
- 一个表达式在某个基本块的入口点可用, 必须要求它在所有前驱的出口点也可用。(2)
- 一个表达式在某个基本块的出口点可用, 或者该表达式是由它产生的; 或者该表达式在入口点可用, 且没有被注销掉。(1)



# 方程求解算法

## ■ 迭代算法

- 初始化:  $\text{in}[B_1] = \text{空};$   
 $\text{out}[B_1] = \text{e\_gen}[B_1]; \text{out}[B_i] = U - \text{e\_kill}[B_i] (i \geq 2)$
- 重复执行下列算法直到out稳定:  

```
for ( i=2; i<=n ;i++) {  
     $\text{in}[B_i] = \bigcap \text{out}[p], p \text{ 是 } B_i \text{ 的前驱};$   
     $\text{out}[B_i] = \text{e\_gen}[B_i] \cup (\text{in}[B_i] - \text{e\_kill}[B_i]);$   
}
```



# 算法说明

---

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



## 10.4 局部优化

---

- 基本块的功能实际上就是计算一组表达式，这些表达式是在基本块出口活跃的变量的值。如果两个基本块计算一组同样的表达式，则称它们是等价的。
- 可以对基本块应用很多变换而不改变它所计算的表达式集合，许多这样的变换对改进最终由某基本块生成的代码的质量很有用。
- 利用基本块的dag表示可以实现一些常用的对基本块的变换。





## 10.4.1 基本块的dag表示

### ■ dag的构造方法

- (1) 基本块中出现的每个变量都有一个dag节点表示其初始值。
- (2) 基本块中的每个语句 $s$ 都有一个dag节点 $n$ 与之相关。  $n$ 的子节点是那些在 $s$ 之前、最后一次对 $s$ 中用到的运算对象进行定义的语句所对应的节点。
- (3) 节点 $n$ 由 $s$ 中用到的运算符来标记，节点 $n$ 还附加了一组变量，这些变量在基本块中都是由 $s$ 最后定义的。
- (4) 如果有的话，还要记下那些其值在块的出口是活跃的节点，它们是输出节点。流图的另一个基本块以后可能会用到这些变量的值。



# 利用dag进行的基本块变换

---

- (1) 局部公共子表达式删除。
- (2) 无用代码删除。
- (3) 交换两个独立的相邻语句的次序，以便减少某个临时值需要保存在寄存器中的时间。
- (4) 使用代数规则重新排列三地址码的运算对象的顺序，以便简化计算过程。

## 10.4.2 局部公共子表达式删除

### ■ 例10.12

$a := b+c$

$b := a-d$

$c := b+c$

$d := a-d$  (10.8)

### ■ 如果b在出口处不是活跃的:

$a := b+c$

$d := a-d$

$c := d+c$

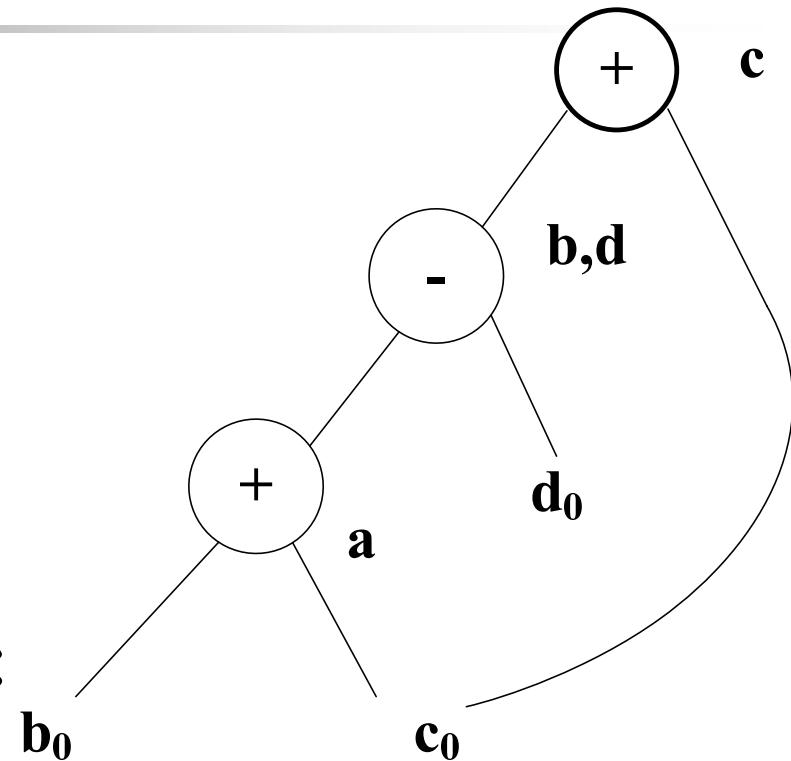


图10.13 基本块(10.8)的dag

## 10.4.3 无用代码删除

- 在dag上删除无用代码的方法很简单：只要从dag上删除所有没有附加活跃变量的根节点(即没有父节点的节点)即可。重复进行这样的处理即可从dag中删除所有与无用代码相对应的节点。

**a := b+c**

**b := b-d**

**c := c+d**

**e := b+c**

图10.14中的a和b是活跃变量，而c和e不是，我们可以立即删除标记为e的根节点。随后，标记为c的节点变成了根节点，下一步也可以被删除。

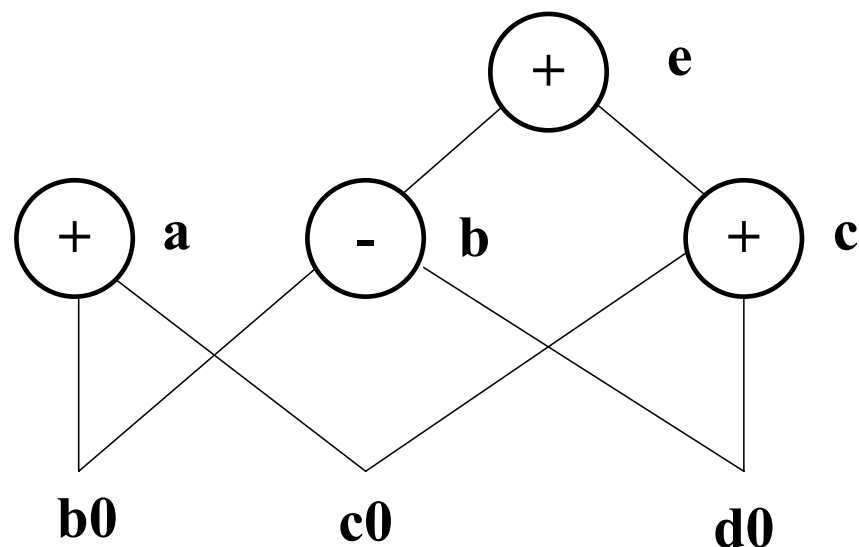


图10.14基本块(10.9)的dag



## 10.4.4代数恒等式的使用

- 代数恒等式代表基本块上另一类重要的优化方法，下面是一些常见的代数恒等式：
- $x+0 = 0+x = x$       $x-0 = x$       $x*1 = 1*x = x$   
 $x/1 = x$
- 另外一类代数优化是局部强度削弱，即用较快的运算符取代较慢的运算符，例如：
- $x**2 = x*x$       $2.0*x = x+x$       $x/2 = x*0.5$



## 10.4.4代数恒等式的使用

---

- 第三类相关的优化技术是常量合并。即在编译时对常量表达式进行计算，并利用它们的值取代常量表达式。例如，表达式 $2*3.14$ 可以替换为6.28。
- dag构造过程可以帮助我们应用上述和更多其它的通用代数变换，比如交换律和结合律。



## 10.4.5 数组引用的dag表示

- 在dag中表示数组访问的正确方法为：
  - 将数组元素赋给其他变量的运算(如 $x = a[i]$ )用一个新创建的运算符为 $=[]$ 的节点表示。该节点的左右子节点分别代表数组初始值(本例中为 $a_0$ )和下标 $i$ 。变量 $x$ 则是该节点的附加标记之一。
  - 对数组元素的赋值(如 $a[j]=y$ )则用一个新创建的运算符为 $[]=$ 的节点来表示。该节点的三个子节点分别表示 $a_0$ 、 $j$ 和 $y$ 。该节点不带任何附加标记。这是因为该节点的创建注销了所有当前已经创建的、其值依赖于 $a_0$ 的节点，而一个被注销的节点不可能再获得任何标记。也就是说，它不可能成为一个公共子表达式。

## 10.4.5 数组引用的dag表示

### ■ 例10.14 基本块

$x = a[i]$

$a[j] = y$

$z = a[i]$

的dag如图10.15所示。

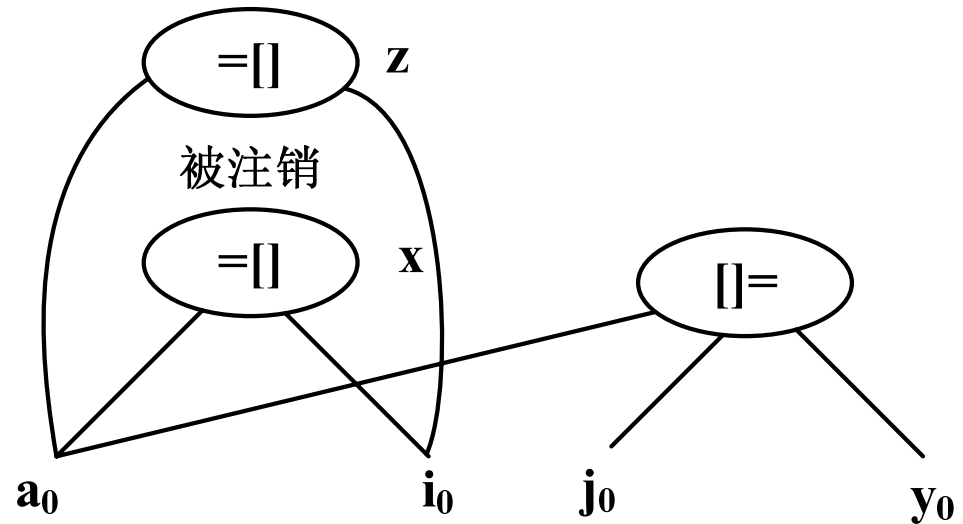


图10.15 将数组元素赋值给其他变量的语句序列的dag



## 10.4.6 指针赋值和过程调用的dag表示

- 当像语句  $x=*p$  或  $*q=y$  那样通过指针进行间接赋值时，并不知道  $p$  和  $q$  指向哪里。
- $x=*p$  可能是对任意某个变量的引用，而  $*q=y$  则可能是对任意某个变量的赋值。因而，运算符  $=*$  必须把当前所有带有附加标识符的节点当作其参数。但这么做将会影响无用代码的删除。更为严重的是， $=*$  运算符将把所有迄今为止构造出来的 dag 中的其他节点全部注销。
- 可以进行一些全局指针分析，以便把一个指针在代码中某个位置上可能指向的变量限制在一个较小的子集内。



## 10.4.7 从dag到基本块的重组

- 对每个具有一个或多个附加变量的节点，构造一个三地址码来计算其中某个变量的值。尽量把计算得到的结果赋给一个在基本块出口处活跃的变量，如果没有全局活跃变量的信息，则假设程序的所有变量在基本块的出口处都是活跃的(临时变量除外)。
- 如果节点具有多个附加的活跃变量，则必须引入复制语句，以便为每一个变量都赋予正确的值。当然，通过全局优化技术可以消除这些复制语句。



## 10.4.7 从dag到基本块的重组

- 当从dag重构基本块时，不仅要关心用哪些变量来存放dag中节点的值，还要关心计算不同节点值的语句顺序。下面是应遵循的规则：
  - (1) 语句的顺序必须遵守dag中节点的顺序。也就是说，只有在计算出某个节点的各个子节点的值之后，才能计算该节点的值。
  - (2) 对数组的赋值必须跟在所有(按照原基本块中的语句顺序)在它之前的对同一数组的赋值或引用之后



## 10.4.7 从dag到基本块的重组

---

- (3) 对数组元素的引用必须跟在所有(在原基本块中)在它之前的对同一数组的赋值语句之后。对同一数组的两次引用可以交换顺序，前提是交换时它们都没有越过某个对同一数组的赋值运算。
- (4) 对某个变量的引用必须跟在所有(在原基本块中)在它之前的过程调用或指针赋值运算之后。
- (5) 任何过程调用或指针赋值都必须跟在所有(在原基本块中)在它之前的对任何变量的引用之后。



## 10.5 循环优化

---

- 循环不变计算的检测
- 代码外提
- 归纳变量删除和强度削弱



## 10.5.1 循环不变计算的检测

算法10.7 循环不变计算检测。

输入：由一组基本块构成的循环 $L$ ，每个基本块包括一系列的三地址码，且每个三地址码的ud-链均可用；

输出：从控制进入循环 $L$ 一直到离开 $L$ ，每次都计算同样值的三地址码；

步骤：

1. 将如下语句标记为“不变”：它们的运算对象或者是常数，或者它们的所有到达-定义都在循环 $L$ 的外面。
2. 重复步骤(3),直到某次重复没有新的语句可标记为“不变”为止
3. 将如下语句标记为“不变”：它们先前没有被标记，而且它们的所有运算对象或者是常数，或者其到达定义都在循环 $L$ 之外，或者只有一个到达定义，这个定义是循环 $L$ 中已标记为“不变”的语句。



## 10.5.2 代码外提

- 将语句  $s: x := y + z$  外提的条件:
  1. 含有语句  $s$  的块是循环中所有出口节点的支配节点，出口节点指的是其后继节点不在循环中的节点
  2. 循环中没有其它语句对  $x$  赋值。如果  $x$  是只赋值一次的临时变量，该条件肯定满足，因此不必检查。
  3. 循环中  $x$  的引用仅由  $s$  到达，如果  $x$  是临时变量，该条件一般也可以满足。



## 10.5.2 代码外提

### ■ 算法10.8 代码外提

- 输入：带有ud-链和支配节点信息的循环 $L$ ；
- 输出：循环的修正版本，增加了前置首节点，且(可能)有一些语句外提到前置首节点中；
- 步骤：
  1. 应用算法10.7寻找循环不变语句。
  2. 对(1)中找到的每个定义 $x$ 的语句 $s$ ，检查是否满足下列条件：
    - a)  $s$ 所在的块支配 $L$ 的所有出口；
    - b)  $x$ 在 $L$ 的其它地方没有被定义，而且
    - c)  $L$ 中所有 $x$ 的引用只能由 $s$ 中 $x$ 的定义到达。
  3. 按算法10.7找出的次序，把(1)中找出的满足(2)中3个条件的每个语句移到新的前置首节点。但是，若 $s$ 的运算对象在 $L$ 中被定义(由算法10.7的(3)找出这种 $s$ )，那么只有这些对象的定义语句外提到前置首节点后，才能外提 $s$ 。





## 10.5.3 归纳变量删除和强度削弱

### ■ 算法10.9 归纳变量检测

输入：带有到达定义信息和循环不变计算信息(由算法10.7得到)的循环 $L$ ;

输出：一组归纳变量，以及与每个归纳变量 $j$ 相关联的三元组 $(i, c, d)$ ，其中 $i$ 是基本归纳变量， $c$ 和 $d$ 是常量，在 $j$ 的定义点， $j$ 的值由 $c*i+d$ 给出。称 $j$ 属于 $i$ 族，基本归纳变量 $i$ 也属于它自己的族；

步骤：

1. 在循环 $L$ 中找出所有的基本归纳变量，此处需要用到循环不变计算的信息。与每个基本归纳变量 $i$ 相关联的三元组为 $(i, 1, 0)$ 。

## 10.5.3 归纳变量删除和强度削弱

2. 在 $L$ 中寻找具有下列形式之一且只被赋值一次的变量 $k$ :

$$k := j * b, \quad k := b * j, \quad k := j / b, \quad k := j \pm b, \quad k := b \pm j$$

其中,  $b$ 是常数,  $j$ 是基本的或非基本的归纳变量。

(1) 如果 $j$ 是基本归纳变量, 则 $k$ 在 $j$ 族中。 $k$ 的三元组依赖于定义它的语句。例如, 如果 $k$ 是由 $k := j * b$ 定义的, 则 $k$ 的三元组为 $(j, b, 0)$ 。类似地可以定义其他情况的三元组。

(2) 如果 $j$ 不是基本归纳变量, 假设 $j$ 属于 $i$ 族, 则 $j$ 还要满足如下要求:

- a) 在循环 $L$ 中对 $j$ 的唯一赋值和对 $k$ 的赋值之间没有对 $i$ 的赋值。
- b) 循环 $L$ 外没有 $j$ 的定义可到达 $k$ 。

此时利用 $j$ 的三元组 $(i, c, d)$ 和定义 $k$ 的语句即可计算 $k$ 的三元组。例如, 定义 $k := b * j$ 导致 $k$ 的三元组为 $(i, b * c, b * d)$ 。注意,  $b * c$ 和 $b * d$ 可以在分析过程中完成计算, 因为 $b$ ,  $c$ 和 $d$ 都是常数。



## 10.5.3 归纳变量删除和强度削弱

算法10.10 用于归纳变量的强度削弱。

输入：循环 $L$ ，附带有到达定义信息和由算法10.9算出的归纳变量族；

输出：修正后的循环；

步骤：依次考察每个基本归纳变量 $i$ 。对每个三元组为 $(i, c, d)$ 的 $i$ 族归纳变量 $j$ ，执行如下步骤：

1. 建立新变量 $s$ ，但如果变量 $j_1$ 和 $j_2$ 具有同样的三元组，则只为它们建立一个新变量。
2. 用 $j := s$ 代替对 $j$ 的赋值。



## 10.5.3 归纳变量删除和强度削弱

3. 在 $L$ 中紧跟在每个赋值语句 $i := i + n$ 之后( $n$ 是常数), 添加如下语句:

$$s := s + c * n$$

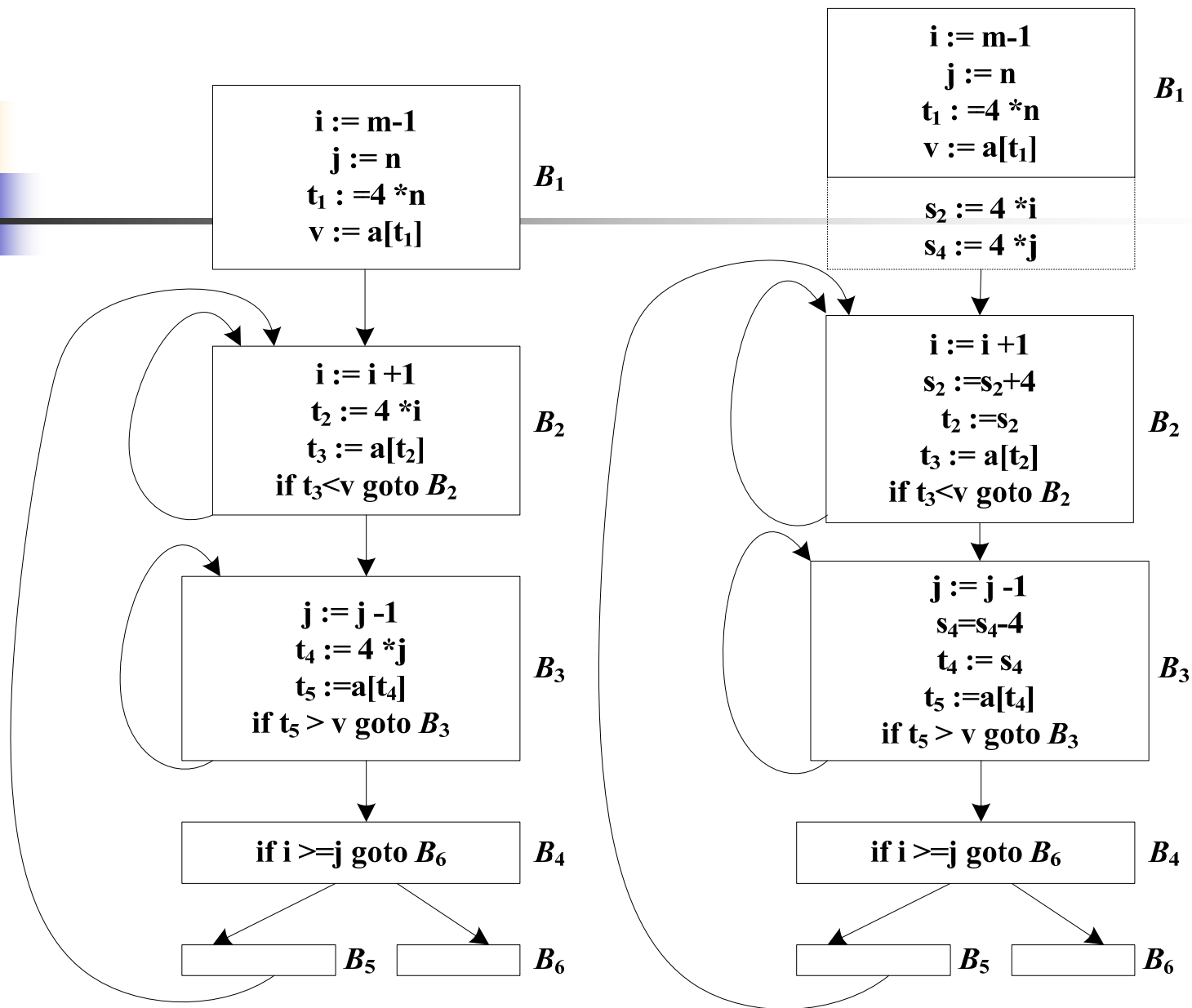
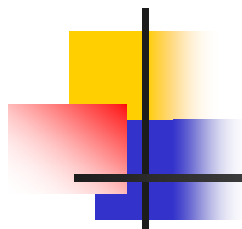
其中, 表达式 $c * n$ 的计算结果为一个常数, 因为 $c$ 和 $n$ 都是常数。将 $s$ 放入 $i$ 族, 其三元组为 $(i, c, d)$ 。

4. 必须保证在循环的入口处 $s$ 的初值为 $c * i + d$ , 该初始化可以放在前置首节点的末尾, 由下面两个语句组成:

$$s := c * i \quad /* \text{如果 } c \text{ 为 } 1, \text{ 则为 } s := i * /$$

$$s := s + d \quad /* \text{如果 } d \text{ 为 } 0 \text{ 则省略 } * /$$

注意,  $s$ 是 $i$ 族的归纳变量。





## 10.5.3 归纳变量删除和强度削弱

---

### 算法10.11 归纳变量删除

输入：带有到达-定义信息、循环不变计算信息  
(利用算法10.7求得)和活跃变量信息的循环 $L$ ;

输出：修正后的循环;

步骤:

## 10.5.3 归纳变量删除和强度削弱

1. 考虑每个仅用于计算同族中其它归纳变量和条件分支的基本归纳变量*i*。取*i*族的某个*j*，优先取其三元组(*i*, *c*, *d*)中的*c*和*d*尽可能简单的*j*(即优先考虑*c* = 1和*d* = 0的情况)，把每个包含*i*的测试语句改成用*j*代替*i*。假定下面的*c*是正的。用下面的语句来代替形如if *i* relop *x* goto *B*的测试语句，其中*x*不是归纳变量。

$r := c * x$       /\* 如果*c*等于1，则为  $r := x$  \*/

$r := r + d$       /\* 如果*d*为0则省略 \*/

if *i* relop *x* goto *B*

最后，因为被删除的归纳变量已经没有什么用处，所以从循环中删除所有对它们的赋值。



## 10.5.3 归纳变量删除和强度削弱

2. 再考虑由算法10.10为其引入语句 $j:=s$ 的每个归纳变量 $j$ 。首先检查在引入的 $j:=s$ 和任何 $j$ 的引用之间有没有对 $s$ 的赋值，应该没有。一般情况下， $j$ 在定义它的块中被引用，这可以简化该检查；否则，需要用到达定义信息，并加上一些对图的分析来实现这种检查。然后用对 $s$ 的引用代替所有对 $j$ 的引用，并删除语句 $j:=s$ 。





## 10.6 全局优化

---

- 全局公共子表达式的删除
- 复制传播



## 10.6.1 全局公共子表达式删除

算法10.12 全局公共子表达式删除。

输入：带有可用表达式信息的流图；

输出：修正后的流图；

步骤：对每个形如 $x:=y+z$ 的语句 $s$ ，如果 $y+z$ 在 $s$ 所在块的开始点是可用的，且该块中在语句 $s$ 之前没有对 $y$ 或 $z$ 的定义，则执行下面的步骤：

1. 为寻找到达 $s$ 所在块的 $y+z$ 的计算，只需沿流图的边从该块开始反向搜索，但不穿过任何计算 $y+z$ 的块。在遇到的每个块中，对 $y+z$ 的最后一次计算将是到达 $s$ 的 $y+z$ 的计算。
2. 建立新变量 $u$ 。
3. 把(1)中找到的每个语句 $w:=y+z$ 用如下语句代替：  
     $u := y+z$   
     $w := u$
4. 用 $x:=u$ 代替语句 $s$ 。



## 10.6.2 复制传播

- 算法10.12、归纳变量删除算法和其它一些算法都会引入形如 $x := y$ 的代码。
- 如果能找出复制代码 $s$ ： $x := y$ 中定义 $x$ 的所有引用点，并用 $y$ 代替 $x$ ，则可以删除该复制语句。前提是 $x$ 的每个引用 $u$ 必须满足下列条件：
  1.  $s$ 必须是到达 $u$ 的 $x$ 的唯一定义(即引用 $u$ 的ud-链只包含 $s$ )。
  2. 在从 $s$ 到 $u$ 的每条路径，包括穿过 $u$ 若干次(但没有第二次穿过 $s$ )的路径上，没有对 $y$ 的赋值。



## 10.6.2 复制传播

算法10.13复制传播。

输入：流图 $G$ ；到达每个块 $B$ 的定义的ud-链； $c\_in[B]$ ，即沿着每条路径到达块 $B$ 的复制语句 $x:=y$ 的集合，在这些路径上 $x:=y$ 的最后一次出现之后没有再对 $x$ 或 $y$ 进行赋值；每个定义的引用的du-链；

输出：修正后的流图；

步骤：对每个复制 $s: x:=y$ 执行下列步骤：

1. 确定由该 $x$ 的定义所能到达的那些 $x$ 的引用。
2. 对(1)中找到的每个 $x$ 的引用，确定 $s$ 是否在 $c\_in[B]$ 中，其中块 $B$ 是含有 $x$ 的本次引用的基本块，而且块 $B$ 中该引用的前面没有 $x$ 或 $y$ 的定义。回想一下，如果 $s$ 在 $c\_in[B]$ 中，那么 $s$ 是唯一的到达块 $B$ 的 $x$ 的定义。
3. 如果 $s$ 满足(2)中的条件，则删掉 $s$ ，且把(1)中找出的所有 $x$ 的引用用 $y$ 来代替。



# 本章小结

---

- 代码优化就是对程序进行等价变换，以提高目标程序的效率，通常只对中间代码进行优化。通常包括控制流分析、数据流分析和变换三部分。
- 以程序的基本块为基础，基本块内的优化叫局部优化，跨基本块的优化为全局优化，循环优化是针对循环进行的优化，是全局优化的一部分。
- 公共子表达式的删除、复制传播、无用代码删除、代码外提、强度削弱和归纳变量删除等都是常用的针对局部或者全局的代码优化方法。



# 本章小结

---

- 划分基本块、构造并分析表示程序控制流信息的流图是进行控制流分析的基础工作。在对循环的分析中，用到支配节点、回边、自然循环、前置首节点、流图的可约等重要概念。
- 对程序中变量的定义和引用关系的分析称为数据流分析，用数据流方程表达变量的定义、引用等，具体进行到达-定义分析、定义-引用分析、可用表达式分析。



*School of Computer Science & Technology*  
*Harbin Institute of Technology*



# 第11章 代码生成

**重点:** 代码生成器设计中的问题, 目标语言,  
一个简单的代码生成器, 寄存器的分配和指派

**难点:** 寄存器的分配和指派





# 第11章 代码生成

---

11.1 代码生成器设计中的问题

11.2 目标语言

11.3 一个简单的代码生成器

11.4 窥孔优化

11.5 寄存器分配与指派

11.6 本章小结





# 第11章 代码生成

---

- 代码生成是编译的最后一个阶段，由代码生成器完成。其任务是把中间代码转换为等价的、具有较高质量的目标代码，以充分利用目标机器的资源。当然，代码生成器本身也必须具有较高的运行效率。
- 目标代码可以是绝对地址的机器代码，或相对地址的机器代码，也可以是汇编代码。
- 本章用微型机的汇编指令来表示目标代码。



## 11.1 代码生成器设计中的问题

---

- 虽然代码生成器的具体实现依赖于目标机器的体系结构、指令系统和操作系统，但存储管理、指令选择、寄存器分配和计算顺序等问题却是设计各种代码生成器都要考虑的问题，本节讨论这类共性问题。



## 11.1.1 代码生成器的输入

- 代码生成器的输入包括中间代码和符号表信息，符号表信息主要用来确定中间代码中的变量所代表的数据对象的运行时地址。
- 假设在代码生成前，编译器的前端已经将源程序扫描、分析和翻译成为足够详细的中间代码，其中变量的值已经可以表示为目标机器能够直接操作的量(位、整数、实数、指针等)；
- 已经完成了必要的类型检查；
- 在需要的地方已经插入了类型转换符；明显的语义错误(如试图把浮点数作为数组下标)也都被检测出来了。



## 11.1.2 目标代码的形式

- 代码生成器的输出是目标代码。目标代码的形式主要有如下3种：
  - **绝对机器语言代码**。所有地址均已定位，可以立即被执行。适于小程序的编译，因为它们可以迅速地被执行。
  - **可重定位的机器语言代码**。允许分别将子程序编译成一组可重定位模块，再由连接装配器将它们和某些运行程序连接起来，转换成能执行的机器语言程序。好处是比较灵活，并能利用已有的程序资源，代价是增加了连接和装配的开销。
  - **汇编语言代码**。生成汇编语言代码后还需要经过汇编程序汇编成可执行的机器语言代码，但其好处是简化了代码生成过程并增加了可读性。



## 11.1.3 指令选择

---

- 所谓**指令选择**是指寻找一个合适的机器指令序列来实现给定的中间代码。
- 目标机器指令系统的性质决定了指令选择的难易程度
  - 指令系统的一致性和完备性是两个重要的因素
  - 特殊机器指令的使用和指令速度是另一些重要的因素



## 11.1.3 指令选择

- 若不考虑目标程序的效率，指令的选择将非常简单：
  - 如：三地址语句  $x := y + z$  翻译成如下代码序列：  
( $x$ ,  $y$  和  $z$  都是静态分配)
    - **MOV**  $y$ ,  $R0$  /\* 把  $y$  装入寄存器  $R0$  \*/
    - **ADD**  $z$ ,  $R0$  /\*  $z$  加到  $R0$  上 \*/
    - **MOV**  $R0$ ,  $x$  /\* 把  $R0$  存入  $x$  中 \*/
- 逐个语句地产生代码，常常得到低质量的代码



## 11.1.3 指令选择

语句序列  $a := b + c$   
 $d := a + e$

的代码如下

**MOV**        **b,    R0**

**ADD**        **c,    R0**

**MOV**        **R0, a**    -- 若a不再使用, 第三条也多余

**MOV**        **a,    R0** -- 多余的指令

**ADD**        **e,    R0**

**MOV**        **R0, d**



## 11.1.3 指令选择

---

- 如果目标机器有加1指令(INC), 则  $a := a+1$  的最有效实现是:

**INC a**

而不是

**MOV a, R0**

**ADD #1, R0**

**MOV R0, a**





## 11.1.4 寄存器分配

---

- 将运算对象放在寄存器中的指令通常要比将运算对象放在内存中的指令快且短，因此，要想生成高质量的目标代码，必须充分使用目标机器的寄存器，寄存器的使用包括：
  - 寄存器分配：为程序的某一点选择驻留在寄存器的一组变量
  - 寄存器指派：确定变量将要驻留的具体寄存器



## 11.1.4 寄存器分配

---

- 选择最优的寄存器指派方案是一个NP完全问题，如果考虑到目标机器的硬件和(或)操作系统对寄存器的使用约束，该问题还会进一步复杂。有关寄存器分配和指派的策略将在11.5节再进行详细讨论。



## 11.1.5 计算顺序选择

---

- 计算执行的顺序同样会影响目标代码的效率。后面将会看到，某些计算顺序比其它顺序需要较少的寄存器来保存中间结果，因而其目标代码的效率也要高。
- 选择最佳计算顺序也是一个NP完全问题。为简单起见，只讨论如何按给定的三地址码的顺序生成目标代码。



# 11.2 目标语言

## 11.2.1 目标机模型

选择可作为几种微机代表的寄存器机器

- 字节寻址，四字节组成一个字，具有 $n$ 个通用寄存器  $R0, R1, \dots, R_{n-1}$ 。

- 二地址指令： **op** 源，目的

**MOV**                    {将源移到目的中}

**ADD**                    {将源加到目的中}

**SUB**                    {在目的中减去源}



## 11.2 目标语言

寻址模式和它们的汇编语言形式及相关开销

寻址模式	汇编形式	地址	增加的开销
------	------	----	-------

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>
-----	----------	----------	----------

下标	$c(R)$	$c + contents(R)$	<b>1</b>
----	--------	-------------------	----------

间址寄存器	$*R$	$contents(R)$	<b>0</b>
-------	------	---------------	----------

间址下标	$*c(R)$	$contents(c + contents(R))$	<b>1</b>
------	---------	-----------------------------	----------

直接量	$\#c$	$c$	<b>1</b>
-----	-------	-----	----------



## 11.2 目标语言

---

### 指令实例

**MOV R0, M**

**MOV 4(R0), M**

***contents(4 + contents(R0))***

**MOV \*4(R0), M**

***contents(contents(4+contents(R0)))***

**MOV #1, R0**



## 11.2 目标语言

---

### 11.2.2 程序和指令的开销

指令开销:= 1+源和目的寻址模式的附加开销

指令	开销
<b>MOV R0, R1</b>	<b>1</b>
<b>MOV R5, M</b>	<b>2</b>
<b>ADD #1, R3</b>	<b>2</b>
<b>SUB 4(R0), *12(R1)</b>	<b>3</b>



# 程序和指令的开销

---

**$a := b + c,$**       **a、b和c都静态分配内存单元**

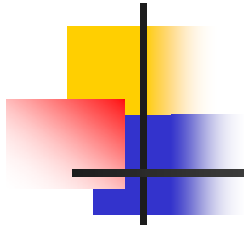
**MOV b, R0**

**ADD c, R0**

**MOV R0, a**

**开销= 6**





# 程序和指令的开销

**$a := b + c,$**       **a、b和c都静态分配内存单元**

**MOV b, R0**

**ADD c, R0**

**MOV R0, a**

**开销= 6**

**MOV b, a**

**ADD c, a**

**开销= 6**



# 程序和指令的开销

---

$a := b + c$ ,       $a$ 、 $b$ 和 $c$ 都静态分配内存单元  
若 $R0$ ,  $R1$ 和 $R2$ 分别含 $a$ ,  $b$ 和 $c$ 的地址, 则

**MOV \*R1, \*R0**

**ADD \*R2, \*R0**

开销= 2



# 程序和指令的开销

---

$a := b + c$ ,       $a$ 、 $b$ 和 $c$ 都静态分配内存单元

若 $R0$ ,  $R1$ 和 $R2$ 分别含 $a$ ,  $b$ 和 $c$ 的地址, 则

**MOV  $*R1, *R0$**

**ADD  $*R2, *R0$**

开销= 2

若 $R1$ 和 $R2$ 分别含 $b$ 和 $c$ 的值, 并且 $b$ 的值在这个赋值后不再需要, 则

**ADD  $R2, R1$**

**MOV  $R1, a$**

开销= 3



## 11.2.3 变量的运行时刻地址

- 存储分配策略以及过程的活动记录中局部数据的布局决定了如何访问变量所对应的内存位置
- 前面假设三地址码中的变量实际上是一个指向符号表表项的指针，在代码生成阶段，变量必须被替换为运行时的内存地址
- 例11.1 考虑三地址码 $x:=0$ ，假设处理完过程的声明部分之后， $x$ 在符号表中的相对地址为12



## 11.2.3 变量的运行时刻地址

- 如果x被分配在一个地址从static开始的静态内存区域中，则x的运行时刻地址为static+12。如果静态区从地址100开始， $x:=0$ 的目标代码为：

**MOV #0, 112。**

- 如果采用栈式存储分配策略，则只有等到运行时刻才能知道一个过程的活动记录位置。此时， $x:=0$ 的目标代码为：

**MOV #0, 12(SP)。**



## 11.3 一个简单的代码生成器

---

- 依次考虑基本块的每个语句，为其产生代码
- 假定三地址语句的每种算符都有对应的目标机器算符
- 假定计算结果留在寄存器中尽可能长的时间，除非：
  - 该寄存器要用于其它计算，或者
  - 到达基本块末尾
- 后续的目标代码也要尽可能地引用保存在寄存器中的变量值



## 11.3.1 后续引用信息

- 为了在代码生成过程中能充分合理地使用寄存器，应把基本块中还会再被引用的变量的值尽量保留在寄存器中，而把基本块内不会再被引用的变量所占用的寄存器及早释放。为此，对于每个形如  $a := b \text{ op } c$  的三地址码，需要知道变量  $a$ 、 $b$  和  $c$  在基本块内是否还会再被引用以及会在哪里被引用，这些信息称为**后续引用信息**。



## 11.3.1 后续引用信息

---

- 如果在一个基本块中，语句 $i$ 定义了 $x$ ，语句 $j$ 要引用 $x$ 的值，且从 $i$ 到 $j$ 之间没有 $x$ 的其它定义，则称 $i$ 中 $x$ 的定义能够**到达** $j$ 。从 $j$ 所能到达的每一个 $x$ 的引用点都称为 $i$ 点定义的变量 $x$ 的**后续引用信息**，所有这样的 $j$ 所组成的引用链则称为变量 $x$ 的**后续引用信息链**。





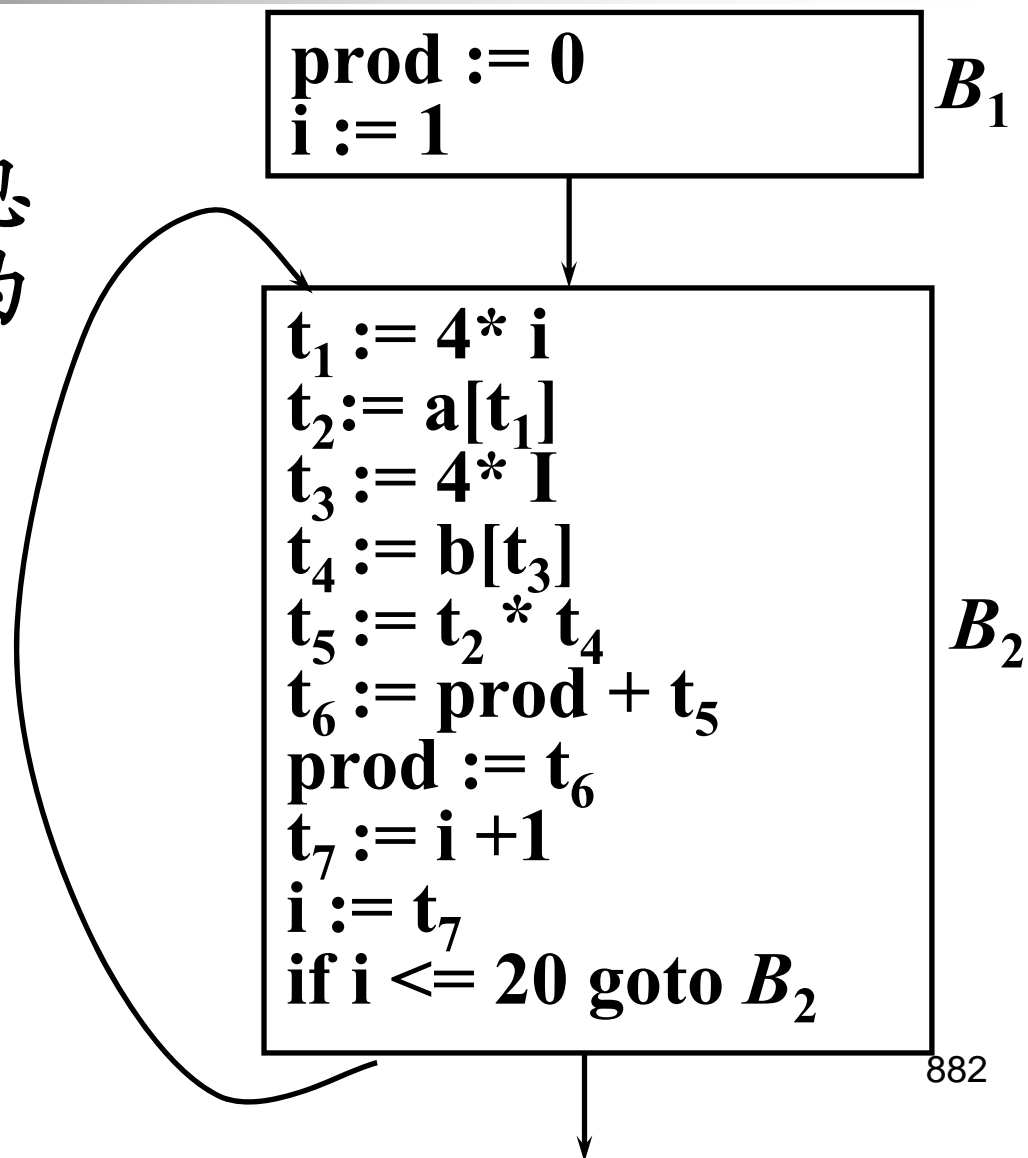
# 后续引用信息的计算

- (1) 初始时，将基本块中各变量的符号表表项的后续引用信息域置为“无后续引用”，并根据该变量在基本块的出口是否活跃，将其活跃信息域置为“活跃”或“不活跃”；
- (2) 从基本块出口向入口反向扫描，并对每个形如 $i:a:=b \text{ op } c$ 的三地址码依次执行如下操作：
  - ① 将符号表中 $a$ 的后续引用信息和活跃信息附加到 $i$ 上
  - ② 将符号表中 $a$ 的后续引用信息和活跃信息分别置为“无后续引用”和“不活跃”
  - ③ 将符号表中 $b$ 和 $c$ 的后续引用信息和活跃信息附加到 $i$ 上
  - ④ 将符号表中变量 $b$ 和 $c$ 的后续引用信息均置为 $i$ ，活跃信息均置为“活跃”

注意，上述次序不能颠倒，因为 $b$ 和 $c$ 也可能就是 $a$ 。此外，因为过程调用可能带来副作用，所以在划分基本块时将过程调用也作为基本块的入口。

## 11.3 一个简单的代码生成器

在没有收集全局信息前，暂且以基本块为单位来生成代码



## 11.3.2 寄存器描述符与地址描述符

例：对  $a := b + c$

- 如果寄存器  $R_i$  含  $b$ ， $R_j$  含  $c$ ，且  $b$  此后不再活跃
  - 产生  $\text{ADD } R_j, R_i$ ，结果  $a$  在  $R_i$  中
- 如果  $R_i$  含  $b$ ，但  $c$  在内存单元， $b$  仍然不再活跃
  - 产生  $\text{ADD } c, R_i$ ，或者
  - $\text{MOV } c, R_j$   
 $\text{ADD } R_j, R_i$

若  $c$  的值以后还要用，第二种代码比较有吸引力



## 11.3.2 寄存器描述符与地址描述符

在代码生成过程中，需要跟踪寄存器的内容和名字的地址

- **寄存器描述符**记录每个寄存器当前存的是什么
  - 在任何一点，每个寄存器保存若干个(包括零个)名字的值
- **名字的地址描述符**记录运行时每个名字的当前值存放的一个或多个位置
  - 该位置可能是寄存器、栈单元、内存地址或者是它们的某个集合
  - 这些信息可以存放在符号表中
- 这两个描述符在代码生成过程中是变化的。



## 11.3.3 代码生成算法

对每个三地址语句  $i: x := y \text{ op } z$

- 调用函数  $getreg(i: x := y \text{ op } z)$  确定可用于保存  $y \text{ op } z$  的计算结果的位置  $L$
- 查看  $y$  的地址描述符以确定  $y$  值当前的一个位置  $y'$ 。如果  $y$  值当前既在内存单元中又在寄存器中，则选择寄存器作为  $y'$ 。如果  $y$  的值还不在于  $L$  中，则生成指令  $MOV\ y',\ L$
- 生成指令  $op\ z',\ L$ ，其中  $z'$  是  $z$  的当前位置之一
- 如果  $y$  和/或  $z$  的当前值没有后续引用，在块的出口也不活跃，并且还在寄存器中，则修改寄存器描述符以表示在执行了  $x := y \text{ op } z$  之后，这些寄存器分别不再包含  $y$  和(或)  $z$  的值。



## 11.3.3 代码生成算法

寄存器选择函数 *getreg*

函数 *getreg* 返回保存  $x := y \text{ op } z$  的  $x$  值的位置  $L$

- 如果变量  $y$  在  $R$  中, 且  $R$  不含其它变量的值, 并且在执行  $x := y \text{ op } z$  后  $y$  不会再被引用, 则返回  $R$  作为  $L$ 。
- 否则, 返回一个空闲寄存器, 如果有的话
- 否则, 如果  $x$  在块中还会再被引用, 或者  $op$  是必须使用寄存器的算符, 则找一个已被占用的寄存器  $R$  (可能产生  $\text{MOV } R, M$  指令, 并修改  $M$  的地址描述符)
- 否则, 如果  $x$  在基本块中不会再被引用, 或找不到适当的被占用寄存器, 则选择  $x$  的内存单元作为  $L$ 。



## 11.3.3 代码生成算法

---

赋值语句  $d := (a - b) + (a - c) + (a - c)$

编译产生的三地址码序列为：

$$t_1 := a - b$$

$$t_2 := a - c$$

$$t_3 := t_1 + t_2$$

$$d := t_3 + t_2$$

$d$ 在基本块的出口是活跃的。

## 11.3.3 代码生成算法

语 句	生成的代码	寄存器描述符	名字地址描述符
		寄存器空	
$t_1 := a - b$	MOV a, R0 SUB b, R0	R0含 $t_1$	$t_1$ 在R0中
$t_2 := a - c$	MOV a, R1 SUB c, R1	R0含 $t_1$ R1含 $t_2$	$t_1$ 在R0中 $t_2$ 在R1中
$t_3 := t_1 + t_2$	ADD R1, R0	R0含 $t_3$ R1含 $t_2$	$t_3$ 在R0中 $t_2$ 在R1中
$d := t_3 + t_2$	ADD R1, R0  MOV R0, d	R0含d	d在R0中  d在R0和内存中





## 11.3.3 代码生成算法

---

前三条指令可以修改，使执行代价降低

**MOV a, R0**

**SUB b, R0**

**MOV a, R1**

**SUB c, R1**

• • •

**MOV a, R0**

**MOV R0, R1**

**SUB b, R0**

**SUB c, R1**

• • •



## 11.3.4 常用三地址码的代码生成

(1)复制:  $a:=b$

- 如果**b**的当前值在寄存器**R**中, 则不必生成代码, 只要将**a**添加到**R**的寄存器描述符中, 并把**a**的地址描述符置为**R**即可。
- 如果**b**在基本块中不会再被引用且在基本块的出口也不活跃, 则还要从**R**的寄存器描述符中删除**b**, 并从**b**的地址描述符中删除**R**。
- 但若**b**的当前值只在内存单元中, 如果只是简单地将**a**的地址描述符置为**b**的内存地址, 那么, 若不对**a**的值采取保护措施, **a**的值将会为**b**的再次定义所影响。此时, 生成一条形如**MOV b, R**的指令会较为稳妥。

## 11.3.4 常用三地址码的代码生成

(2) 一元运算:  $a := op\ b$

- 与二元运算的处理类似。

(3) 数组元素引用:  $a := b[i]$ 。

- 假设 $a$ 在基本块中还会再被引用, 而且寄存器 $R$ 是可用的, 则将 $a$ 保留在寄存器 $R$ 中。于是, 如果 $i$ 的当前值不在寄存器中, 则生成如下指令序列:

**MOV  $i, R$**

**MOV  $b(R), R$**       开销=4

- 如果 $i$ 的当前值在寄存器 $R_i$ 中, 则生成如下指令:

**MOV  $b(R_i), R$**       开销=2

## 11.3.4 常用三地址码的代码生成

(4) 数组元素赋值:  $a[i] := b$

- 假设  $a$  是静态分配的。如果  $i$  的当前值不在寄存器中，则生成如下指令序列：

**MOV  $i, R$**

**MOV  $b, a(R)$       开销=5**

- 如果  $i$  的当前值在寄存器  $R_i$  中，则生成如下指令：

**MOV  $b, a(R_i)$       开销=3**

## 11.3.4 常用三地址码的代码生成

(5) 指针引用:  $a := *p$

- 同样假设 $a$ 在基本块中还会再被引用, 而且寄存器 $R$ 是可用的, 则将 $a$ 保留在寄存器 $R$ 中。  
于是, 如果 $p$ 的当前值不在寄存器中, 则生成如下指令:

**MOV  $p, R$**

**MOV  $*R, R$       开销=3**

- 如果 $p$ 的当前值在寄存器 $R_i$ 中, 则可生成如下指令:

**MOV  $*R_i, a$       开销=2**

## 11.3.4 常用三地址码的代码生成

(6) 指针赋值:  $*p := a$

- 假设  $a$  是静态分配的。如果  $p$  的当前值不在寄存器中，则生成如下指令：

**MOV  $p, R$**

**MOV  $a, *R$**       开销=4

- 如果  $p$  的当前值在寄存器  $R_i$  中，则可生成如下指令：

**MOV  $a, *R$**       开销=2



## 11.3.4 常用三地址码的代码生成

---

(7) 无条件转移: `goto L`

- 假设L为三地址语句的序号，则生成指令**JMP L'**。
- 其中，L'为序号为L的三地址语句的目标代码首址。

## 11.3.4 常用三地址码的代码生成

(8) 条件转移: if a rop b goto L

- 同样假设L为三地址语句的序号, 则生成如下的指令序列:

**CMP a, b**

**CJrop L'**

- 其中, L'的含义与(7)中相同, **CMP**为比较指令, **Cjrop**为条件码跳转指令, **CMP**根据rop取>、<或=而将条件码分别置为正、负或零。如果a和/或b的当前值在寄存器中, 则在生成目标代码时应尽量使用寄存器寻址模式。





## 11.4 窥孔优化

- 窥孔优化是一种简单有效的局部优化方法，它通过检查目标指令中称为窥孔的短序列，用更小更短的指令序列进行等价代替，以此来提高目标代码的质量。
- 窥孔是放在目标程序上的一个移动的小窗口。孔中的代码不需要是连续的。
- 该技术的特点是每次优化后的结果可能又为进一步的优化带来了机会。所以有时会对目标代码重复进行多遍优化。下面介绍几种典型的窥孔优化技术。



## 11.4 窥孔优化

---

- 11.4.1 冗余指令消除

- 如果遇到如下的指令序列:

(1) **MOV R0, a**

(2) **MOV a, R0** (11.1)

则可以删除指令(2)。但是，如果(2)带有标号，通常是不能删除的。



## 11.4 窥孔优化

---

- 11.4.2 不可达代码消除
- 删除紧跟在无条件跳转指令后的无标号指令称为不可达代码删除。



## 11.4 窥孔优化

---

### ■ 11.4.3 强度削弱

- 强度削弱是指在目标机器上用时间开销小的等价操作代替时间开销大的操作。例如用  $x * x$  实现  $x^2$  要比调用一个指数过程快很多。用移位操作实现乘以2或除以2的定点运算要更快一些。用乘法实现(近似)浮点除法也可能会更快一些。



## 11.4 窥孔优化

---

### ■ 11.4.4 特殊机器指令的使用

- 为了提高效率，目标机器有时会使用一些硬件指令来实现某些特定的操作。例如，有一些机器具有自动加1和自动减1的寻址模式。这些模式的运用可以大大提高参数传递过程中压栈和出栈的代码质量，它们还可以用在形如 $i:=i+1$ 的语句的代码中。



## 11.4 窥孔优化

---

### ■ 11.4.5 其他处理

- 也可以利用其他一些途经进行窥孔优化。例如，通过删除那些不必要的连续转移；利用代数恒等性质删除形如  $x := x + 0$  或  $x := x * 1$  的代码的代数化简。



## 11.5 寄存器分配与指派

---

- 由于只涉及寄存器运算对象的指令要比那些涉及内存运算对象的指令短且快，因此有效地利用寄存器非常重要。
- 寄存器分配的任务是为程序的某一点选择应该驻留在寄存器中的一组变量
- 寄存器指派则负责挑出变量将要驻留的具体寄存器。



## 11.5.1 全局寄存器分配

- 由于程序的大多数时间都花在内层循环上，因此一种比较自然的全局寄存器分配方法是在整个循环中将经常引用的值保存在固定的寄存器中。
- 假设已经利用第10章的技术找出了流图中的循环结构，而且知道基本块中计算出的哪些值要在该块外被引用，则有一种简单的全局寄存器分配策略，那就是分配固定数目的寄存器来保存每个内部循环中最活跃的值。
- 不同循环选中的值也会有所不同。
- 未被分配的寄存器可用于存放11.4节讨论的基本块的局部值。
- 该方法的缺点是：固定的寄存器数目对全局寄存器分配来说可能不够用，但实现简单。





## 11.5.2 引用计数

- 如果 $x$ 在寄存器中，则对 $x$ 的每次引用都将节省一个单元的开销。于是可以采用一种简单的方法来确定执行循环 $L$ 时将变量 $x$ 保存在寄存器中所节省的开销。
- 计算循环 $L$ 中为 $x$ 分配寄存器所节省开销的近似公式：

$$\sum_{L \text{ 中的块 } B} (use(x, B) + 2 * live(x, B)) \quad (11.5)$$

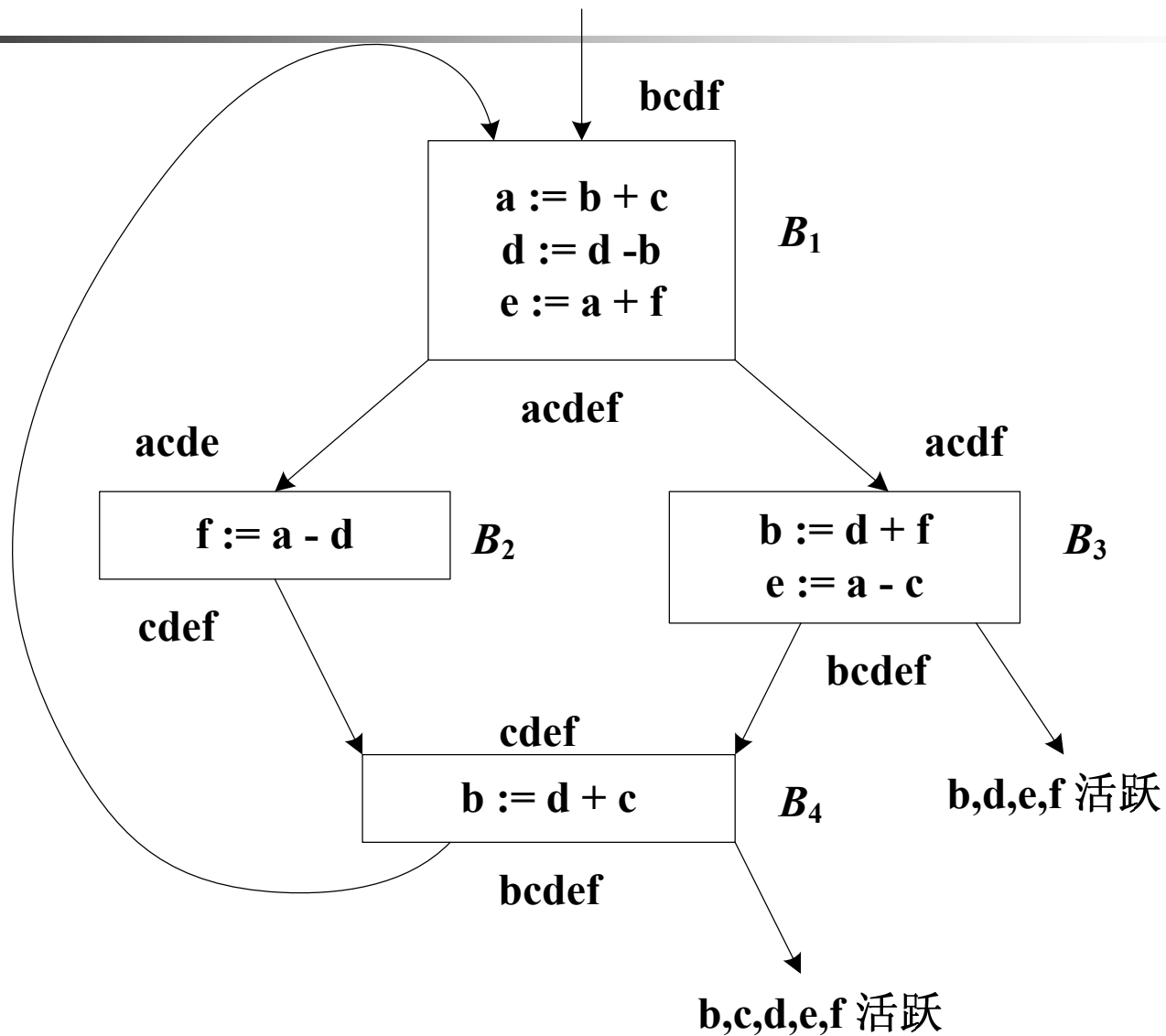
- 其中， $use(x, B)$ 是定义 $x$ 之前，块 $B$ 中对 $x$ 的引用次数。如果 $x$ 在 $B$ 的出口处是活跃的，而且 $B$ 中含有对 $x$ 的赋值，则 $live(x, B)$ 为1，否则 $live(x, B)$ 为0。注意，(11.5)是个近似公式。这是因为循环中的块的迭代次数可能是不一样的，而且我们还假设循环会迭代许多次。



## 11.5.2 引用计数

- 例11.3 考虑图11.2中内层循环中的基本块，块中的跳转语句均已删除。假设用寄存器R0、R1和R2来保存循环中的值。为方便起见，图中还列出了在每个块入口和出口活跃的变量。e和f在 $B_1$ 的末尾都是活跃的，但只有e在 $B_2$ 的入口是活跃的，只有f在 $B_3$ 的入口是活跃的。一般地，块末尾活跃的变量是其后继块入口活跃变量的并集。

# 例11.3



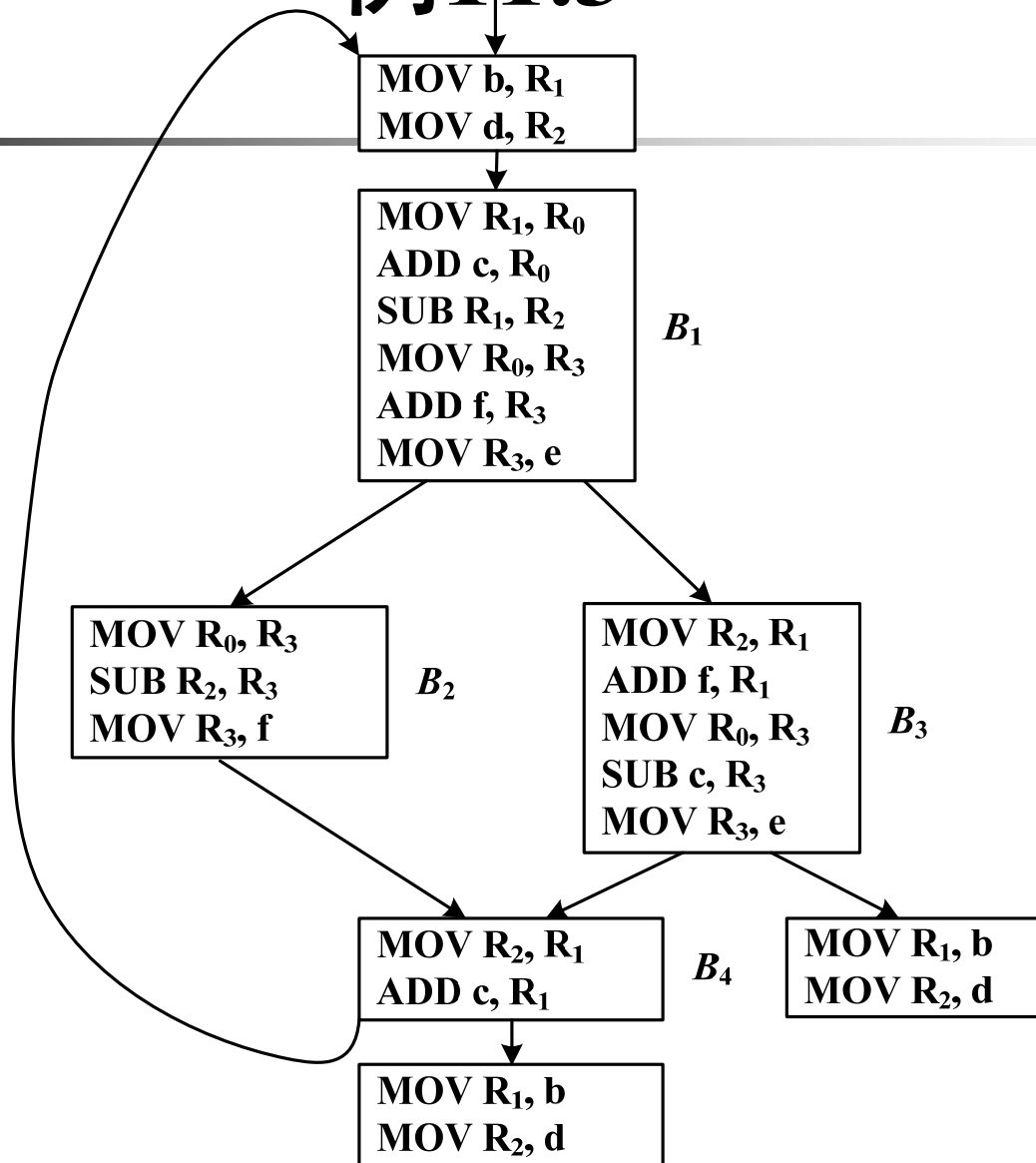
## 例11.3

- 首先计算 $x=a$ 时(11.5)的值，由于 $a$ 在 $B_1$ 的出口是活跃的， $B_1$ 中还含有对 $a$ 的赋值，而且 $a$ 在 $B_2$ 和 $B_3$ 的出口不活跃，因此

因为 $a$ 是在任何引用之前于 $B_1$ 中定义的，所以 $use(a, B_1)=0$ 。同理， $use(a, B_2)=use(a, B_3)=1$ ，而且， $use(a, B_4)=0$ ，于是，

综上， $x=a$ 时(11.5)式的值为4。亦即，将 $a$ 存入某个全局寄存器可以节省4个单元的开销。由于 $x=b$ 、 $c$ 、 $d$ 、 $e$ 和 $f$ 时(11.5)式的值分别为6、3、6、4和4，因此可以将 $a$ 、 $b$ 和 $d$ 放入寄存器R0、R1和R2。使用R0存放 $e$ 或 $f$ 而不是 $a$ 可以节省相同的开销。

## 例11.3





## 11.5.3 外层循环的寄存器指派

- 为内层循环分配了寄存器并生成代码之后，可以将同样的方法应用到外层循环上。
- 如果外层循环 $L_1$ 包含内层循环 $L_2$ ，则在 $L_2$ 中分配了寄存器的变量不必再在 $L_2-L_1$ 中分配寄存器。
- 如果变量 $x$ 是在循环 $L_1$ 中而不是在 $L_2$ 中分配了寄存器，则必须在 $L_2$ 的入口处保存 $x$ 并在离开 $L_2$ 进入块 $L_1-L_2$ 之前装载 $x$ 。



## 11.5.3 外层循环的寄存器指派

- 如果在 $L_2$ 而不是 $L_1$ 中为 $x$ 分配了寄存器，则必须在 $L_2$ 的入口装载 $x$ ，并在 $L_2$ 的出口保存 $x$ 。
- 如果计算时需要寄存器但所有可用的寄存器均被占用，则必须将某个正被使用的寄存器中的内容存放(溢出)到内存中以释放一个寄存器。
- 图染色法是一种简单的用于寄存器分配和寄存器溢出管理的系统技术。



# 本章小结

1. 目标代码的生成需要尽力开发利用机器提供的资源，特别是根据开销选用恰当的指令和寄存器，以提高其执行效率；
2. 稀缺资源寄存器的有效利用涉及到后续引用问题，寄存器描述符用来记录每个寄存器当前的内容；地址描述符记录运行时存放变量当前值的一个或多个位置，用来确定对变量的存取方式；
3. 使用引用计数能够良好地实现寄存器的分配和指派；
4. 不同形式的三地址码对应不同的目标代码，且具有不同的执行代价；
5. 不可达和冗余指令删除、控制流优化、强度削弱、代数化简、特殊指令使用等都是有效的窥孔优化方法；