



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 ₁	arg ₂	result
0	minus	b		t ₁
1	+	c	d	t ₂
2	*	t ₁	t ₂	t ₃
3	+	c	d	t ₄
4	-	t ₃	t ₄	t ₅
5	assign	t ₅		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 ， arg_1 和 arg_2 ， op ， arg_1 和 arg_2 的含义与四元式类似，区别只是 arg_1 和 arg_2 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 ₁	arg ₂
0	minus	b	
1	+	c	d
2	*	(0)	(1)
3	+	c	d
4	-	(2)	(3)
5	assign	a	(4)
	...		

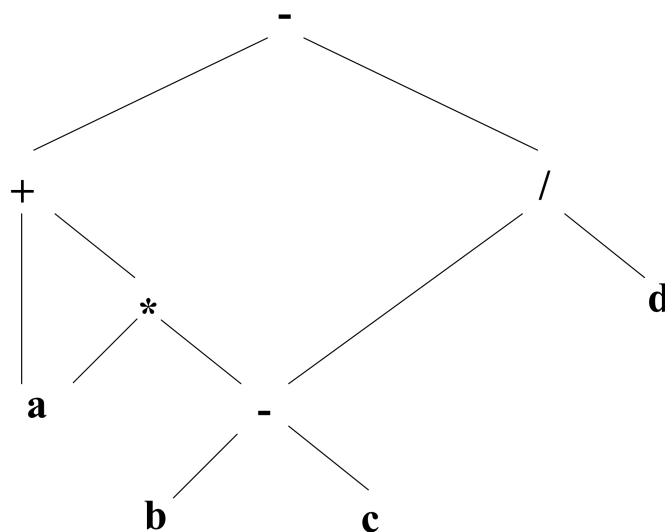
生成三地址码的语法制导定义

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 T_1 和 T_2 是类型表达式，则其笛卡尔乘积 $T_1 \times T_2$ 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): ↑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

- $P \rightarrow \text{prog id (input, output) } D ; S$
- $D \rightarrow D ; D \mid List : T \mid \text{proc id } D ; S$
- $List \rightarrow List_1, \text{id} \mid \text{id}$
- $T \rightarrow \text{integer} \mid \text{real} \mid \text{array } C \text{ of } T_1 \mid \uparrow T_1 \mid \text{record } D$
- $C \rightarrow [\text{num}] C \mid \varepsilon$
- 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

$P \rightarrow MD$

$M \rightarrow \varepsilon \quad \{\text{offset} := 0\}$

$P \rightarrow \{\text{offset} := 0\} D$

$D \rightarrow D ; D$

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

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

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

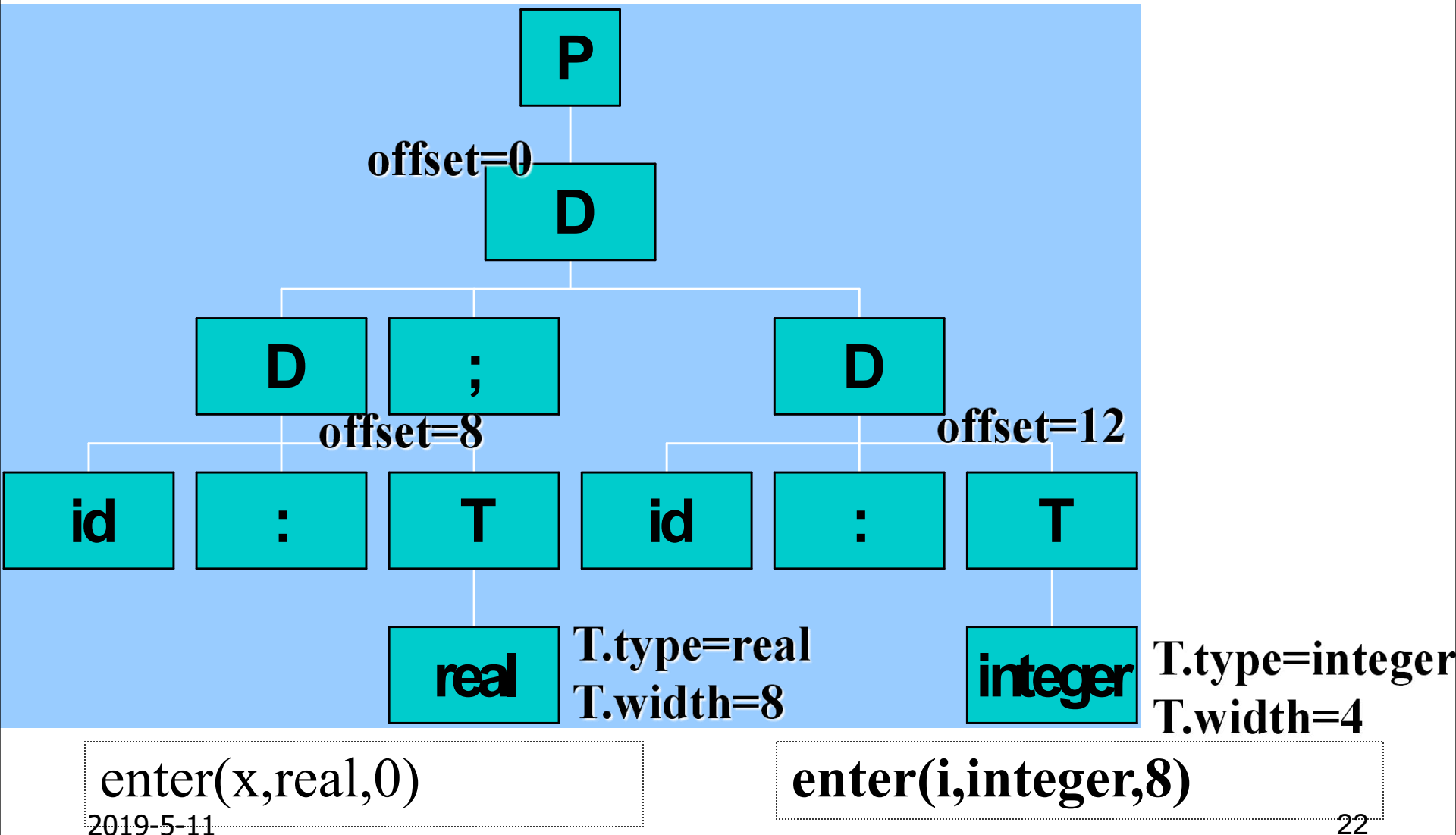
$T \rightarrow \text{array}[\text{num}] \text{ of } T_1$

$\quad \{T.\text{type} := \text{array}(\text{num.val}, T_1.\text{type});$
 $\quad T.\text{width} := \text{num.val} * T_1.\text{width}\}$

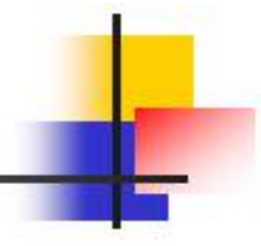
$T \rightarrow \uparrow T_1 \quad \{T.\text{type} := \text{pointer}(T_1.\text{type}); T.\text{width} := 4\}$

例 x:real; i:integer 的翻译

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



例 $x:\text{real}; i:\text{integer}$ 的翻译



$P \Rightarrow \{\text{offset}:=0\}D$
 $\Rightarrow \{\text{offset}:=0\}D;D$
 $\Rightarrow \{\text{offset}:=0\}x:T\{\text{enter}(x,T.\text{type},\text{offset});$
 $\qquad\qquad\qquad \text{offset}:=\text{offset}+T.\text{width}\};D$
 $\Rightarrow \{\text{offset}:=0\}x:\text{real}\{T.\text{type}:=\text{real};T.\text{width}:=8\}$
 $\qquad\qquad\qquad \{\text{enter}(x,T.\text{type},\text{offset});\text{offset}:=\text{offset}+T.\text{width}\};D$
 $\Rightarrow x:\text{real}\{(x,\text{real},0);\text{offset}:=8\};D$
 $\Rightarrow x:\text{real}\{(x,\text{real},0);\text{offset}:=8\};i:T$
 $\qquad\qquad\qquad \{\text{enter}(i.\text{name},T.\text{type},\text{offset}); \text{offset}:=\text{offset}+T.\text{width}\}$
 $\Rightarrow x:\text{real}\{(x,\text{real},0);\text{offset}:=8\};i:\text{integer}\{T.\text{type}:=\text{integer};$
 $\qquad\qquad\qquad T.\text{width}:=4\}\{\text{enter}(i,T.\text{type},\text{offset});\text{offset}:=\text{offset}+T.\text{width}\}$
 $\Rightarrow x:\text{real}\{(x,\text{real},0)\};i:\text{integer}\{(i,\text{integer},8);\text{offset}:=12\}$

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

$P \rightarrow \text{prog id (input,output) } \mathbf{M} \mathbf{D}; S \{ \text{addwidth}(\text{top}(\text{tblptr}), \text{top}(\text{offset})); \text{pop}(\text{tblptr}); \text{pop}(\text{offset}) \}$

$M \rightarrow \varepsilon \quad \{ t := \text{mktable}(\text{nil});$
 $\text{push}(t, \text{tblptr}); \text{push}(0, \text{offset}) \}$

$D \rightarrow D_1; D_2$

$D \rightarrow \text{proc id}; \mathbf{N} \mathbf{D}_1; S \{ t := \text{top}(\text{tblptr});$
 $\text{addwidth}(t, \text{top}(\text{offset})); \text{pop}(\text{tblptr}); \text{pop}(\text{offset});$
 $\text{enterproc}(\text{top}(\text{tblptr}), \text{id.name}, t) \}$

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

$N \rightarrow \varepsilon \quad \{ t := \text{mktable}(\text{top}(\text{tblptr}));$
 $\text{push}(t, \text{tblptr}); \text{push}(0, \text{offset}) \}$

```
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...
```

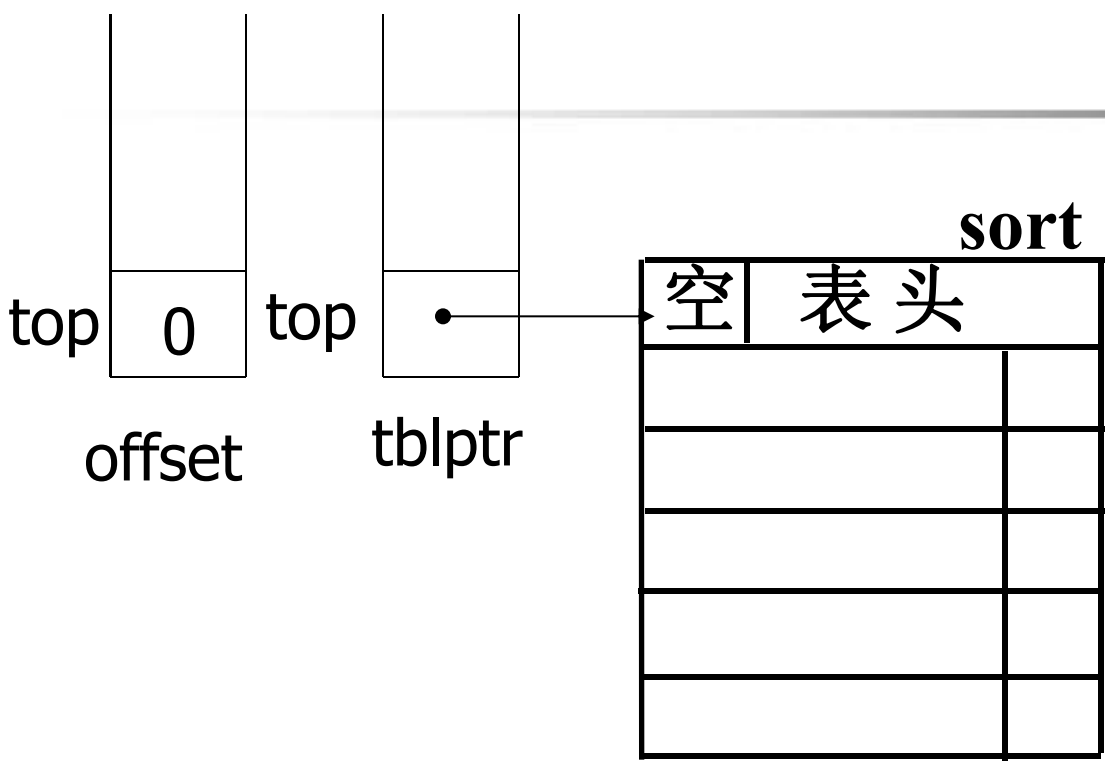
```
      ...exchange(i,j)...end;
```

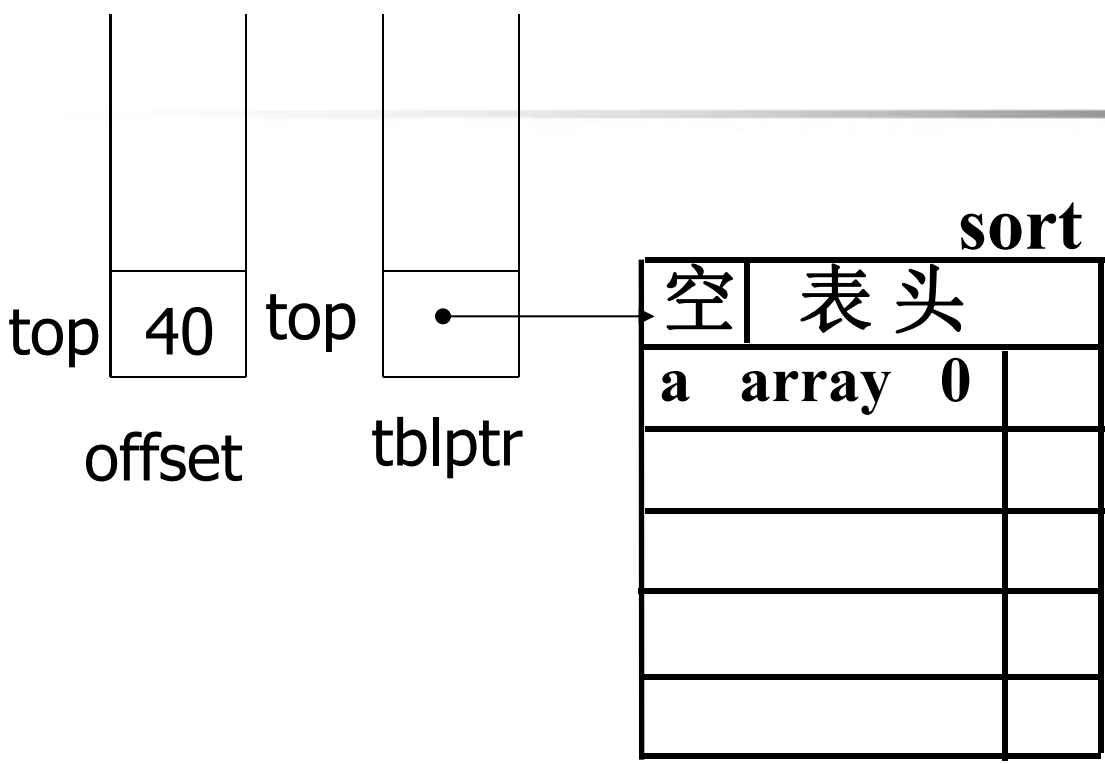
```
  begin ... end;
```

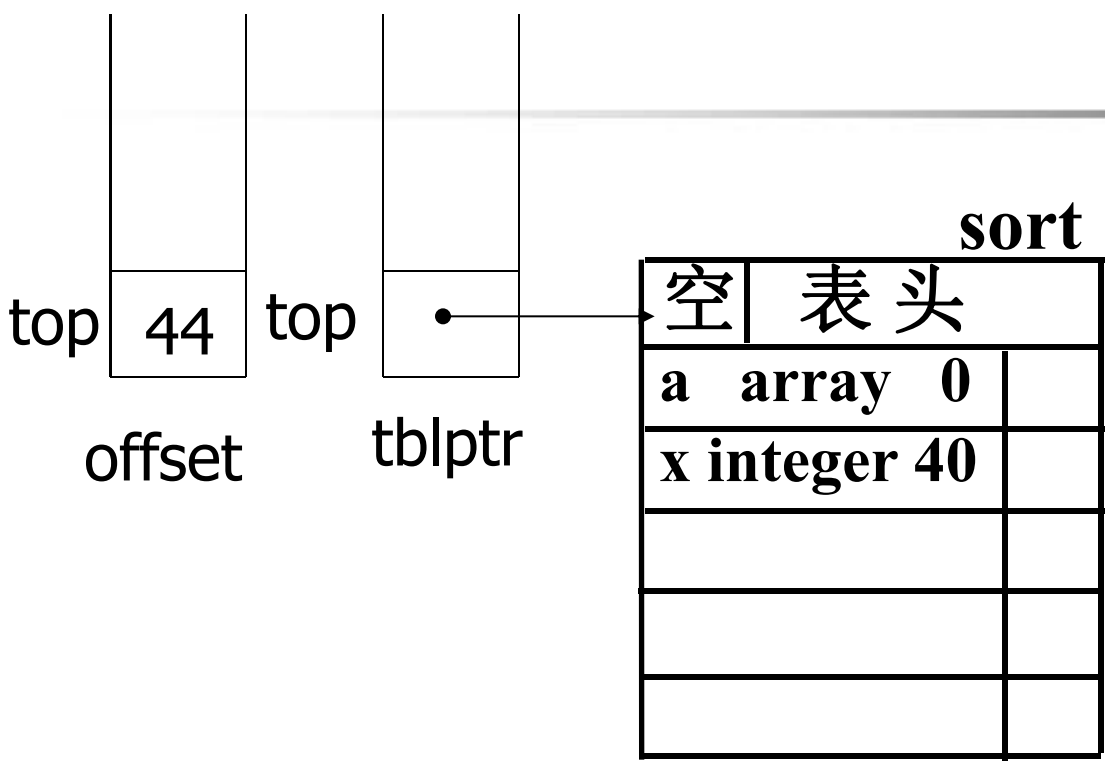
```
begin ... end;
```

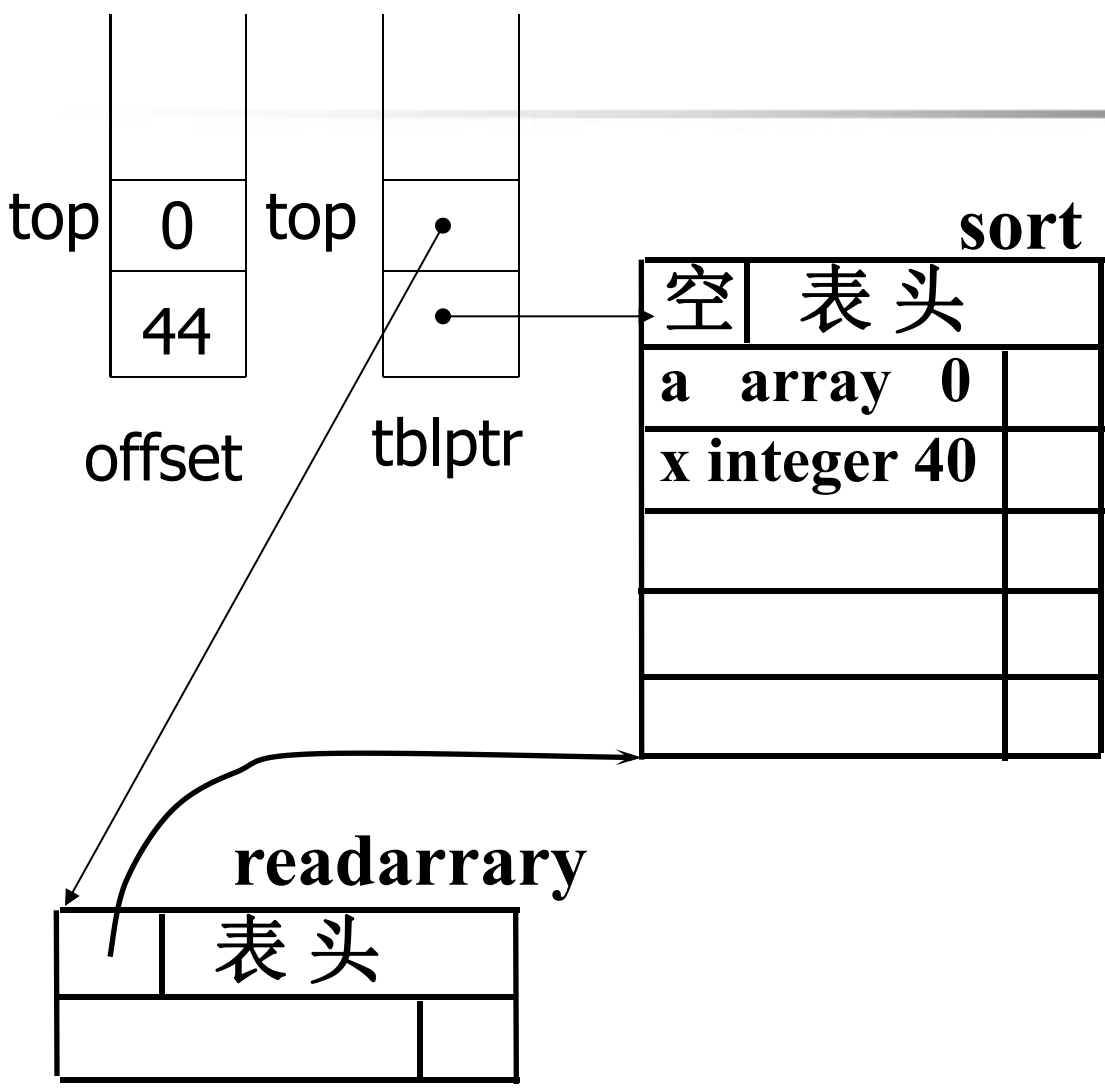
**例 一个带有嵌套的
pascal程序(图7.11)**

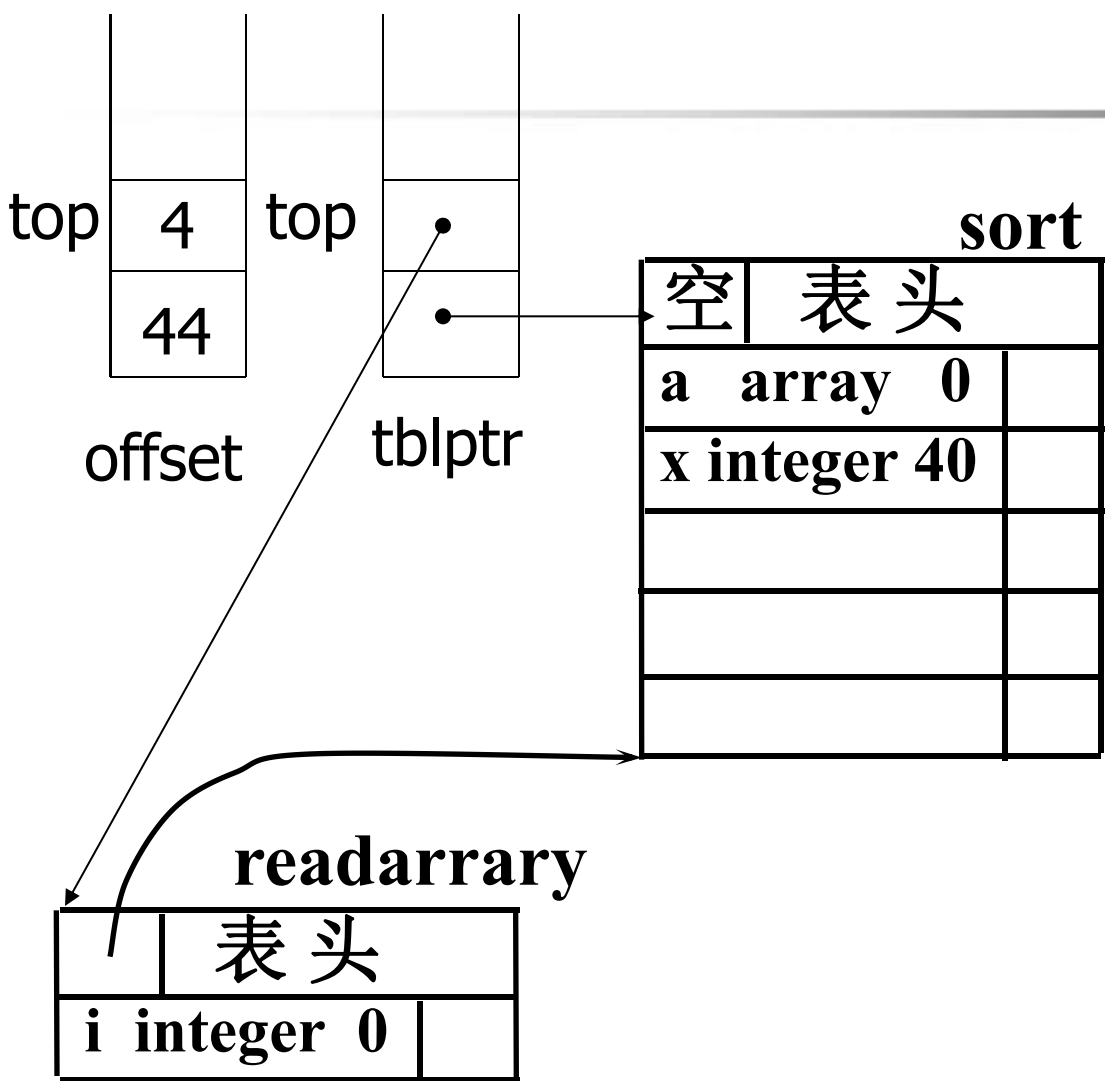


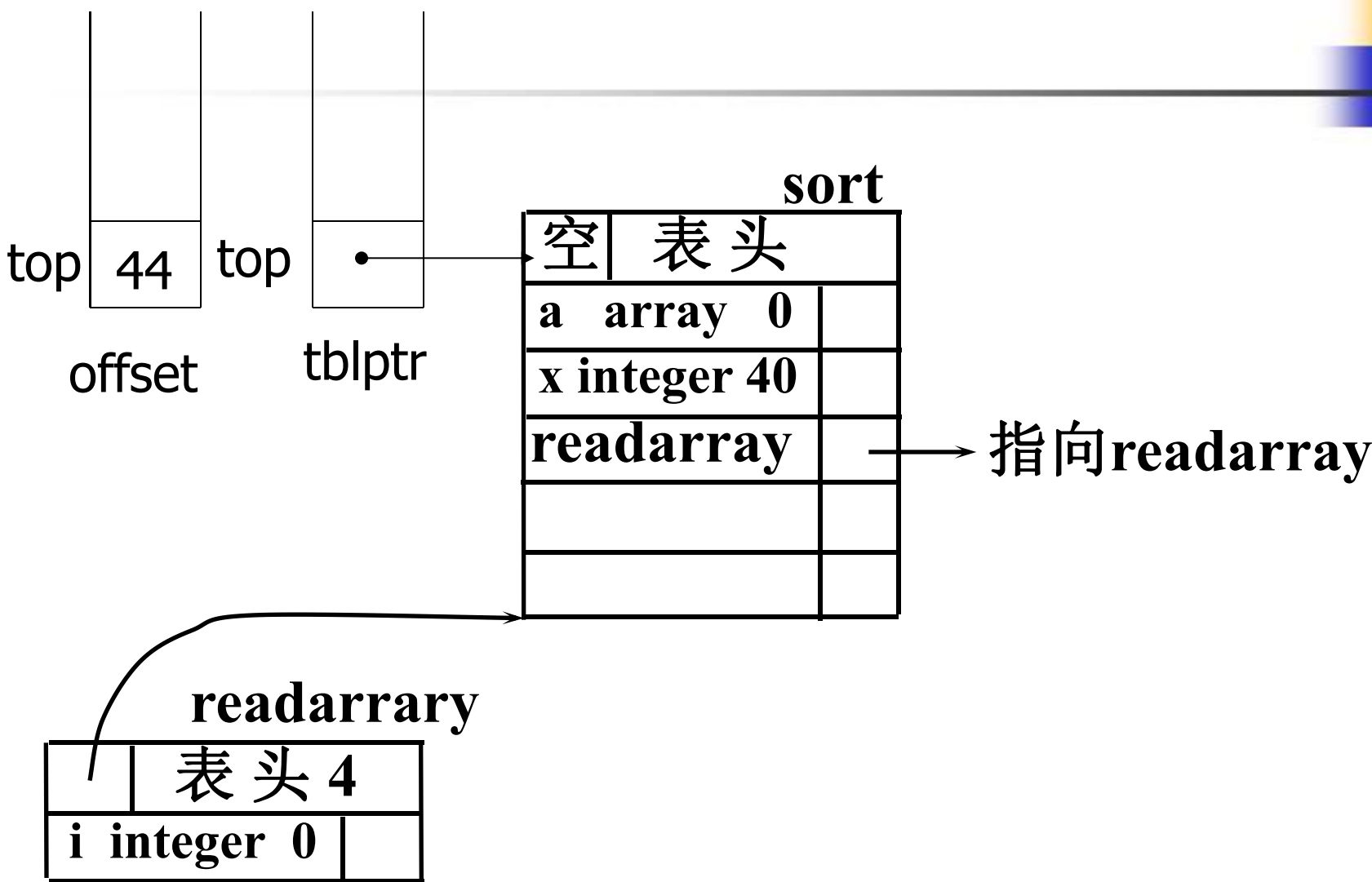


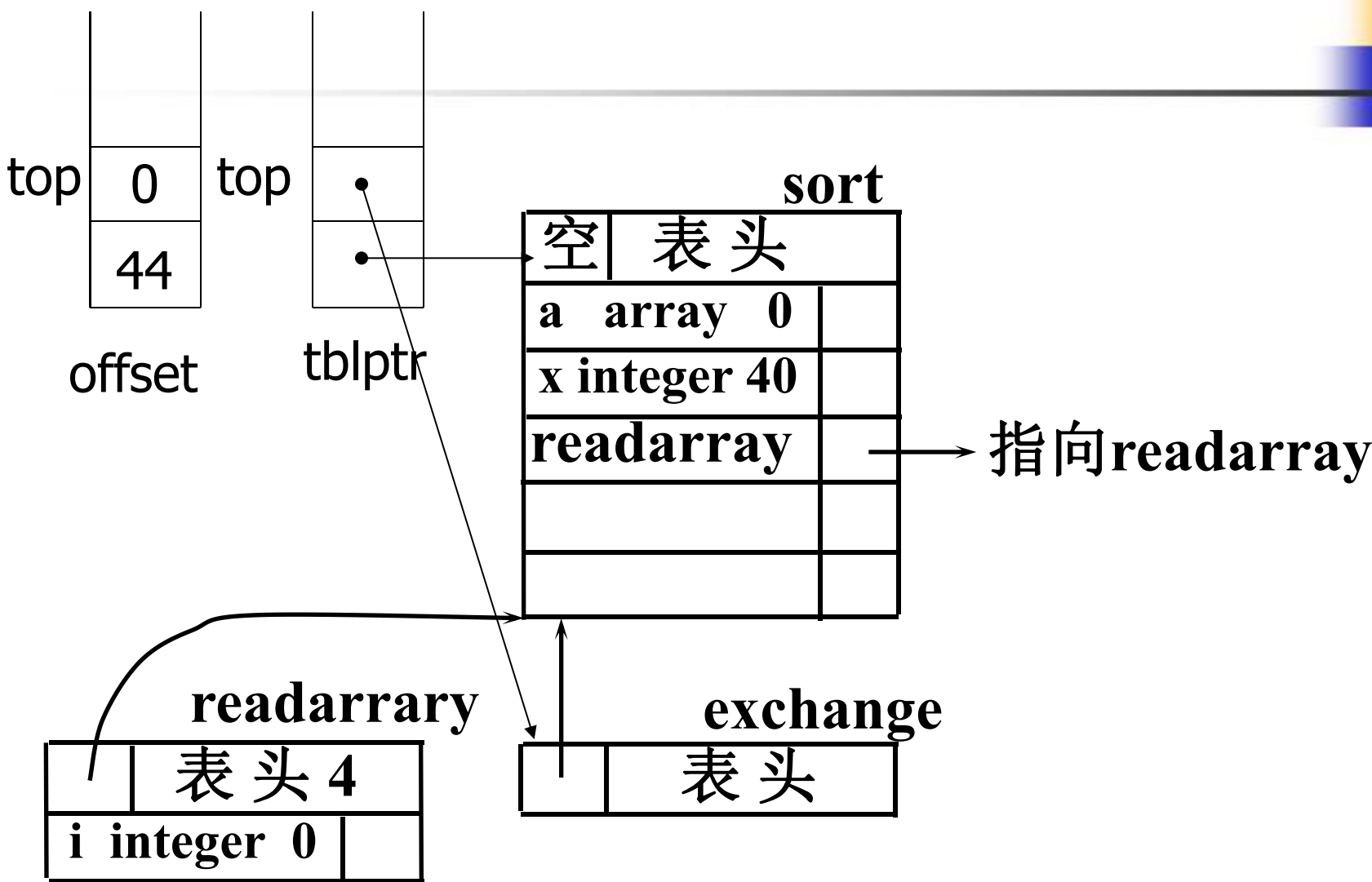


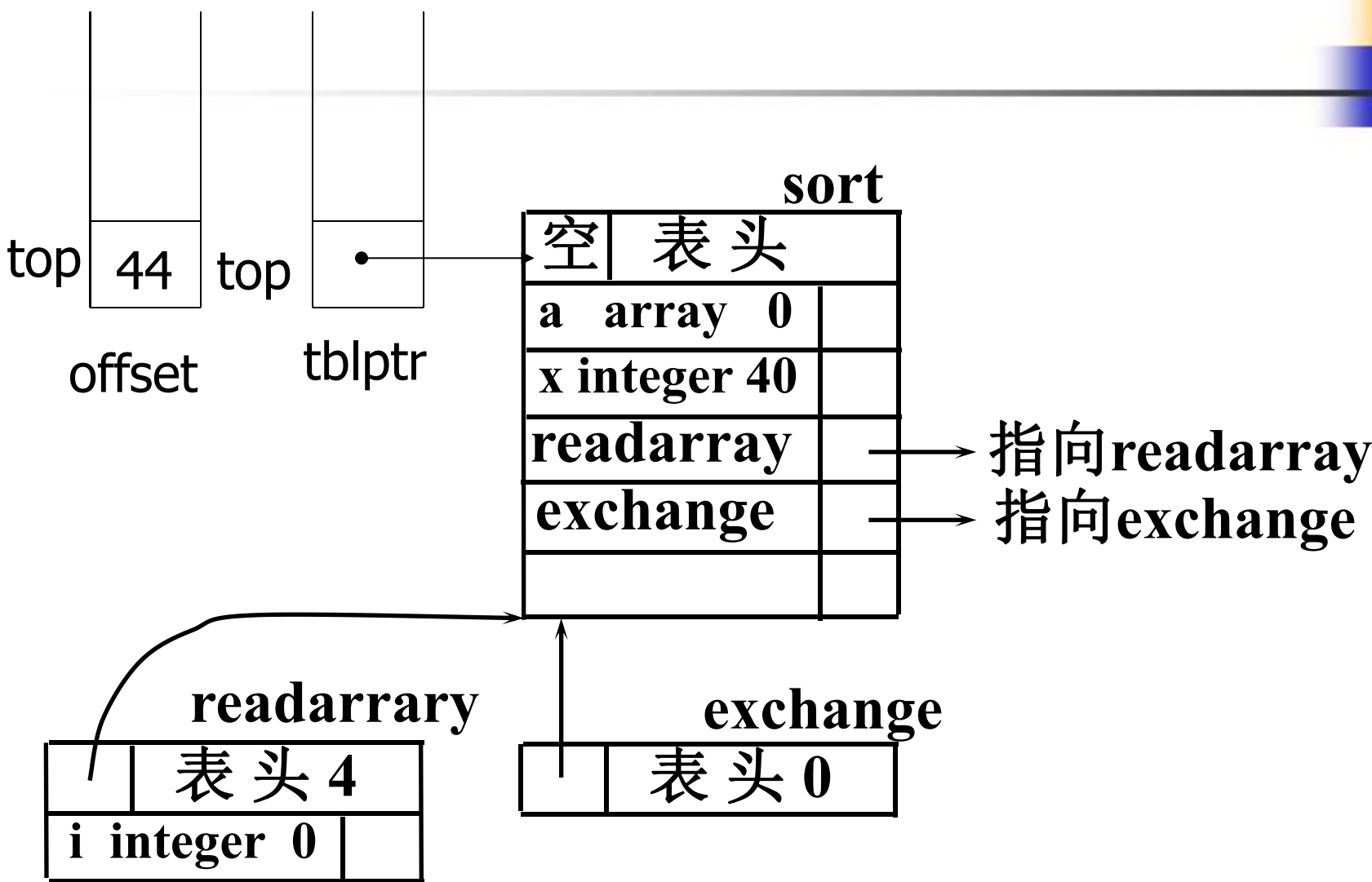


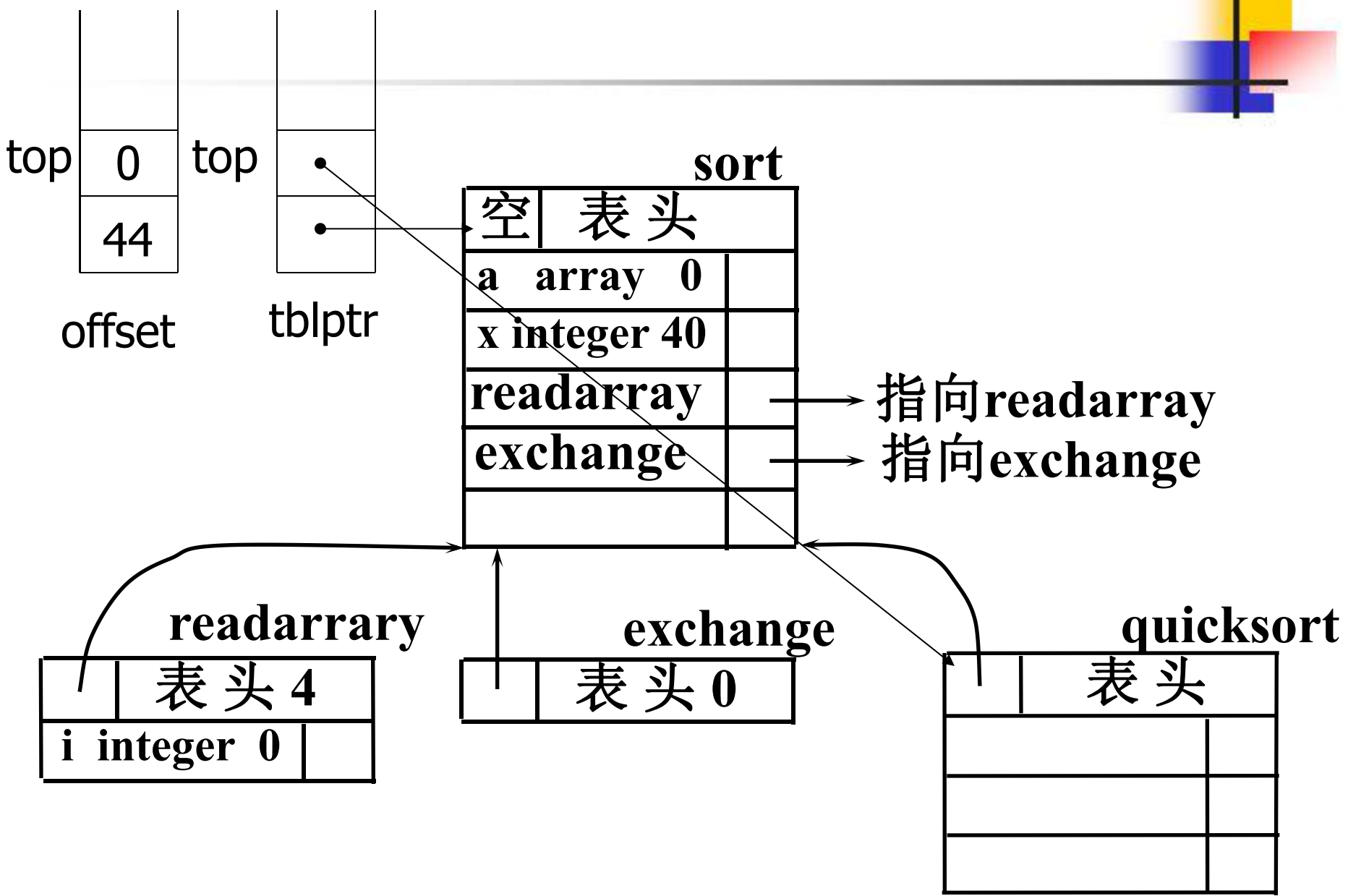


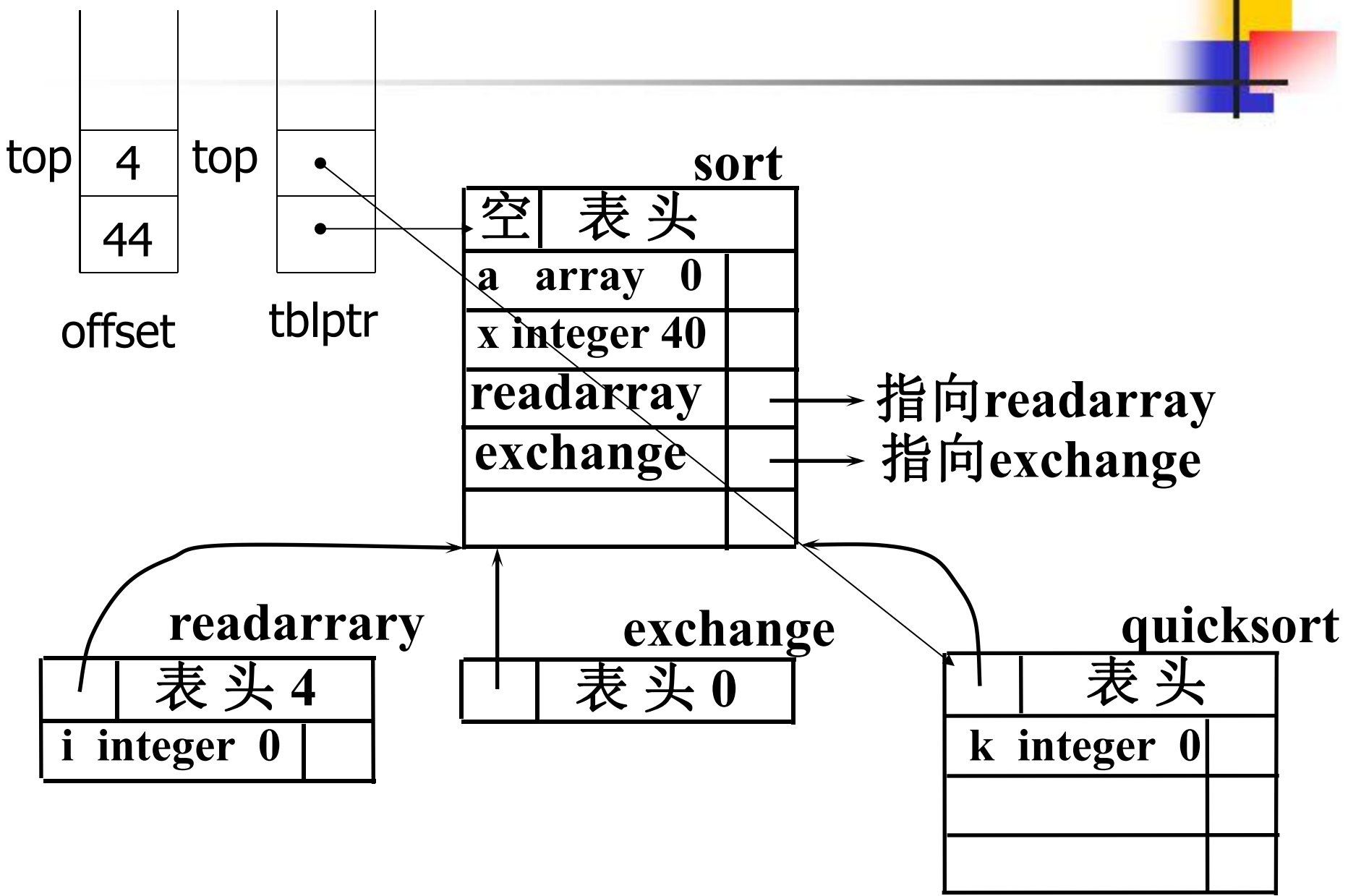


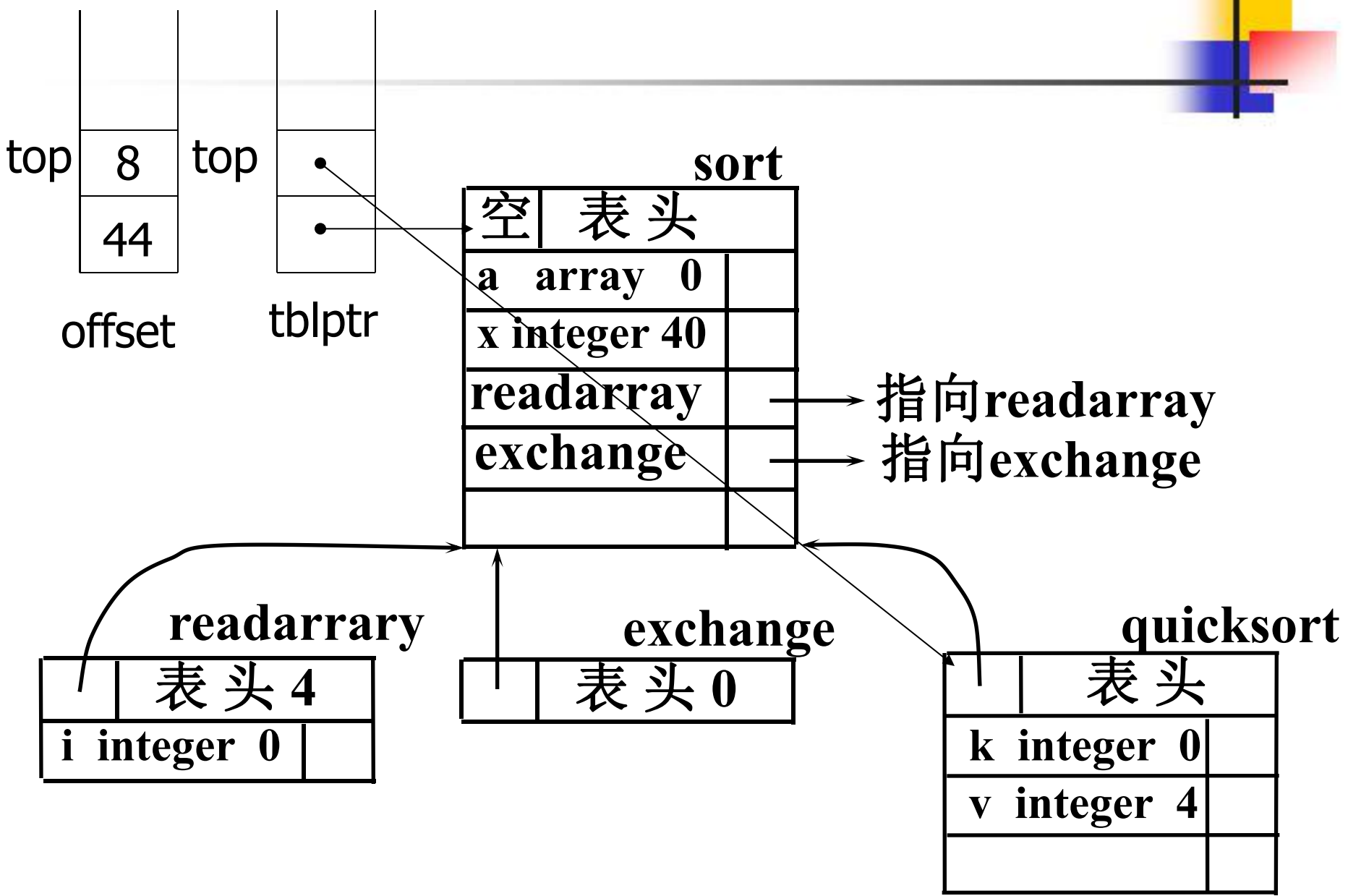


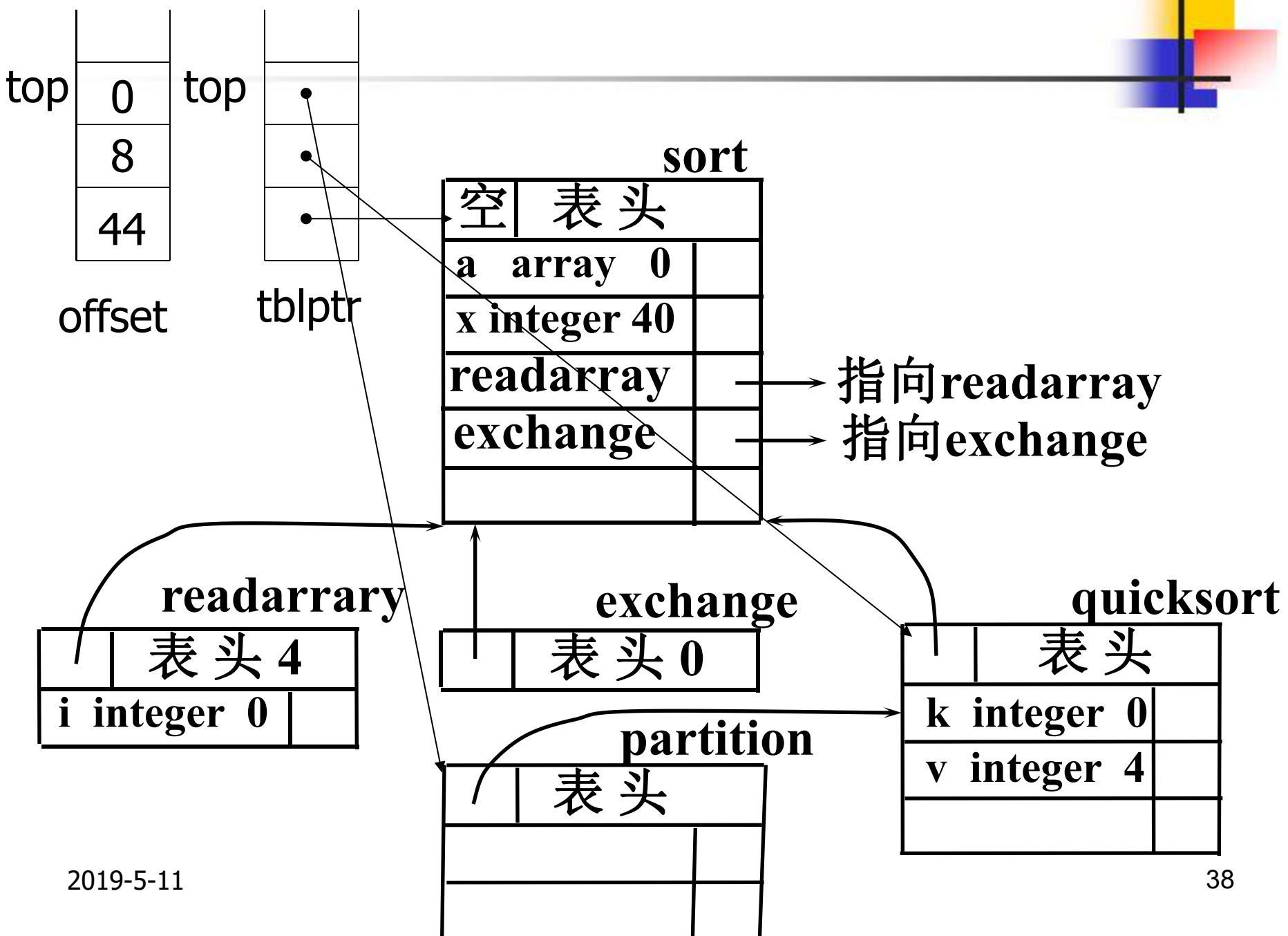


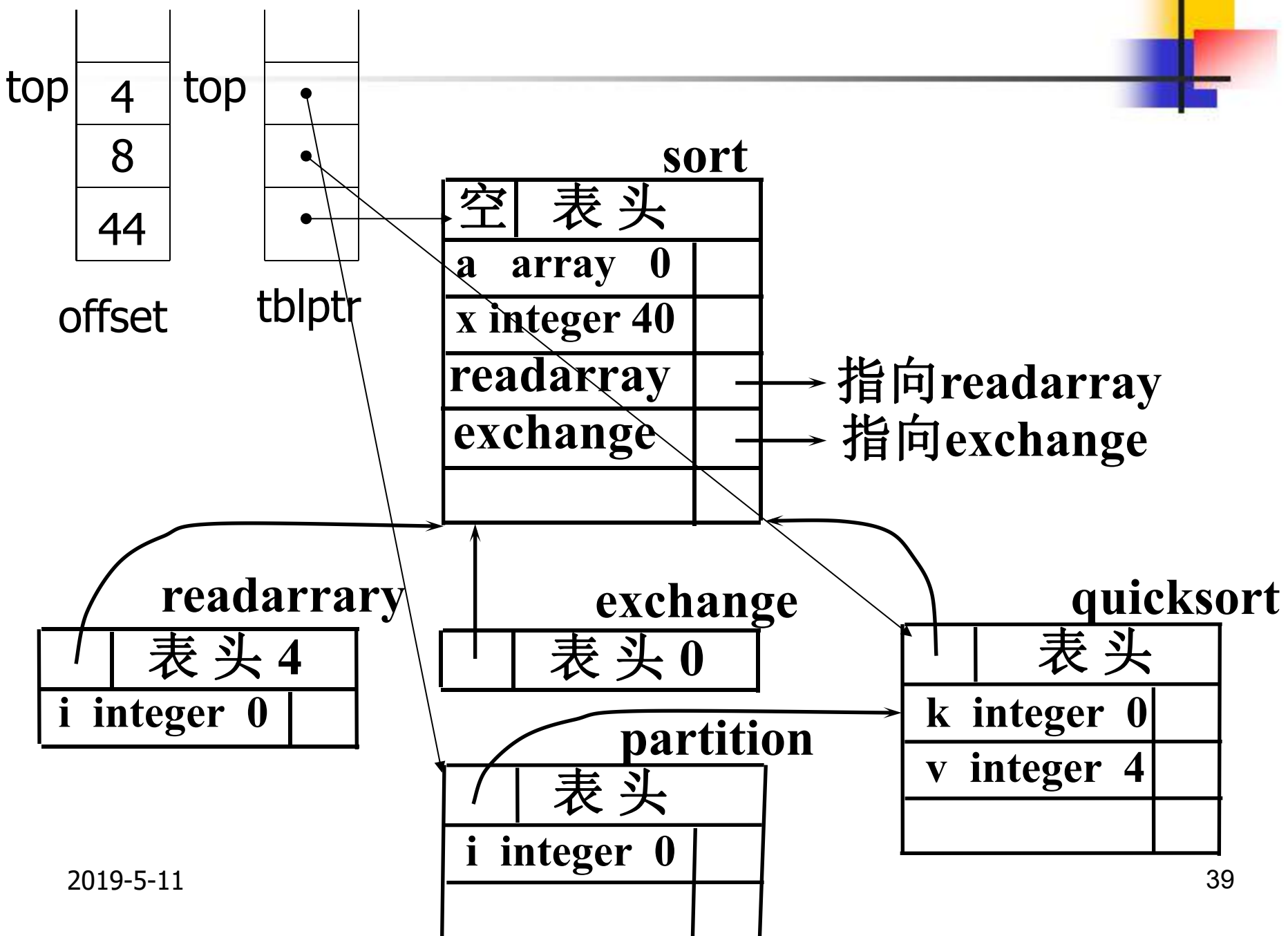


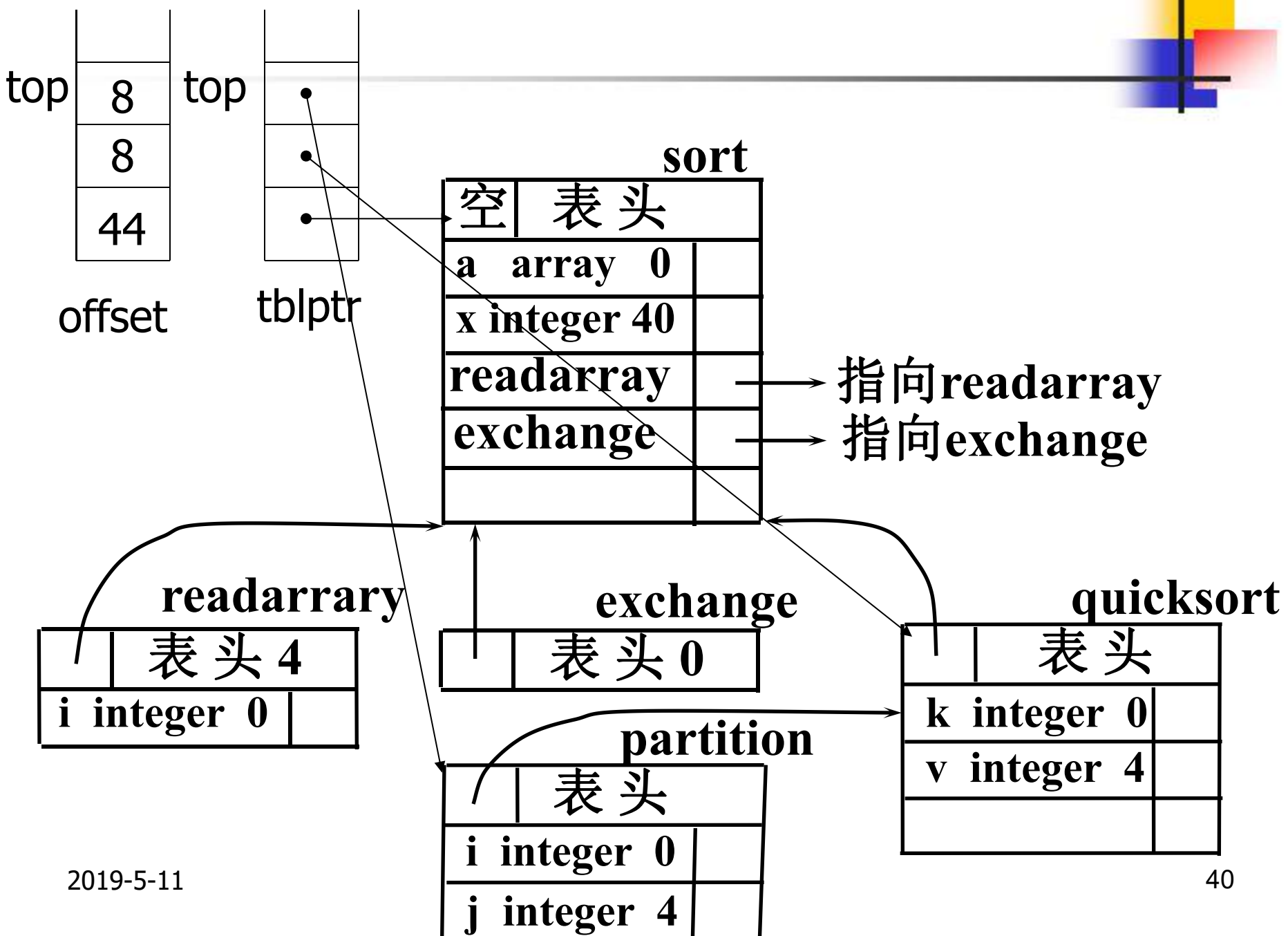


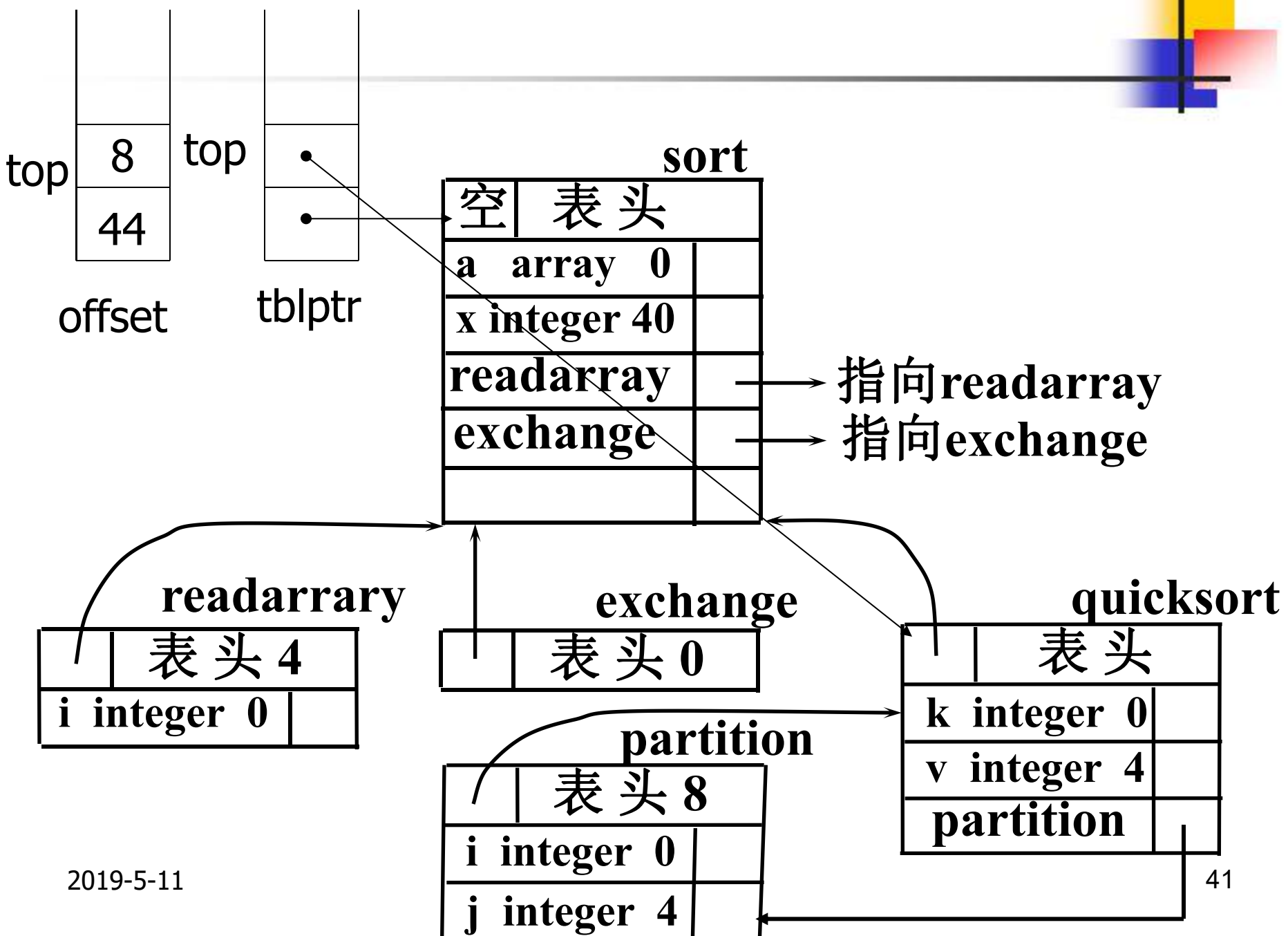


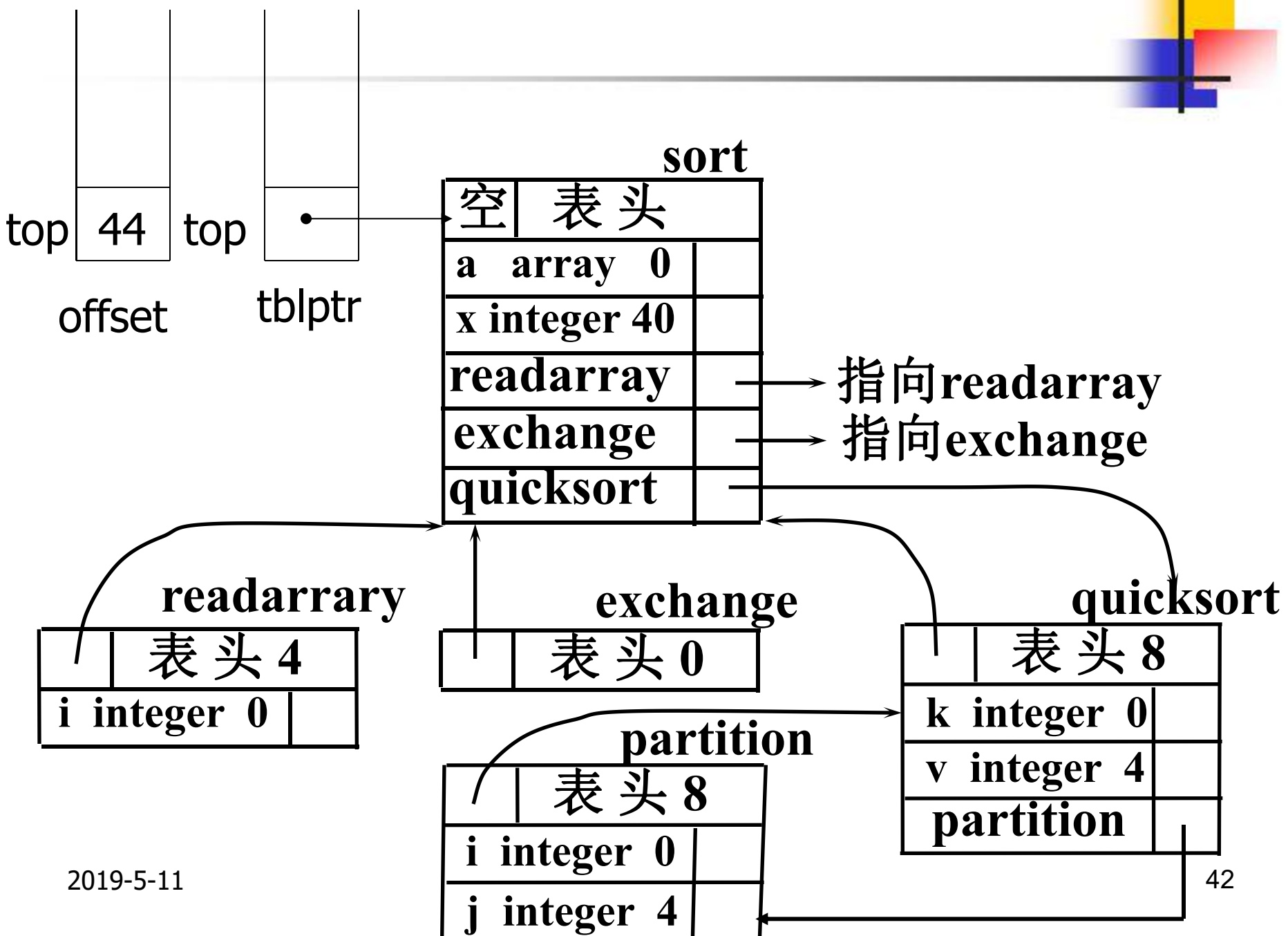


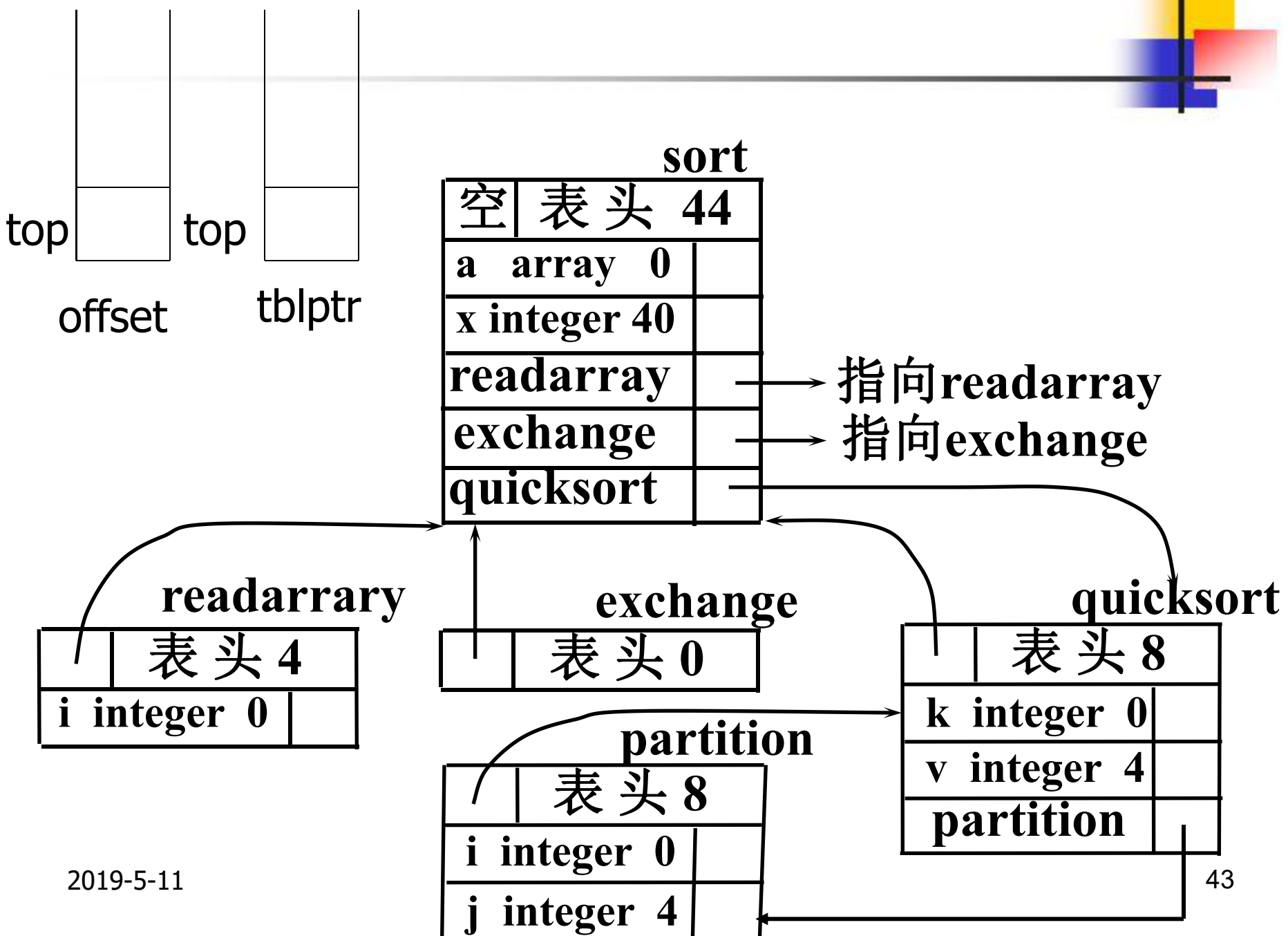












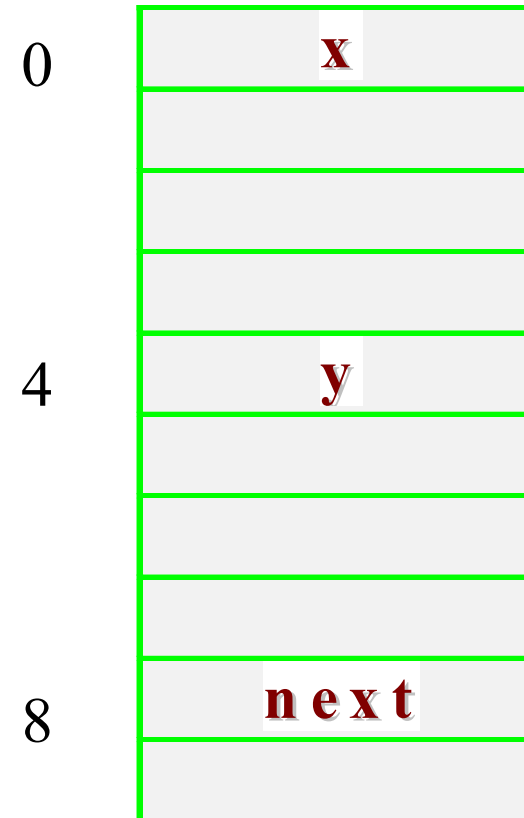
7.2.6 记录的翻译

■ 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

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

$T \rightarrow \text{record } L \text{ D end}$

$\{T.type := \text{record}(\text{top}(\text{tblptr}));$

$T.width := \text{top}(\text{offset});$

$\text{pop}(\text{tblptr}); \text{pop}(\text{offset}) \}$

$L \rightarrow \varepsilon$

$\{t := \text{mktable}(\text{nil});$

$\text{push}(t, \text{tblprt}); \text{push}(0, \text{offset}) \}$



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 }

$E \rightarrow E_1 + E_2$ { $E.addr := \text{newtemp};$
 $\text{emit}(E.addr, ':=', E_1.addr, '+', E_2.addr)$ }

$E \rightarrow -E_1$ { $E.addr := \text{newtemp};$
 $\text{emit}(E.addr, ':=', \text{'uminus'}, E_1.addr)$ }

$E \rightarrow (E_1)$ { $E.addr := E_1.addr$ }

$E \rightarrow id$ $\{p := \text{lookup}(id.name);$
 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

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

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

数组元素地址计算——行优先

- 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

$L \rightarrow \text{Elist} \mid \text{id}$

$\text{Elist} \rightarrow \text{Elist}, \text{E} \mid \text{id}[\text{E}]$

■ 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 → **Elist₁**, **E** {t := newtemp; m := Elist₁.ndim + 1;

i₁ * n₂ gencode(t, ':=' , Elist₁.addr, '*', limit(Elist₁.array, m));

gencode(t, ':=' , t, '+', E.addr);

(i₁ * n₂) + i₂ Elist.array := Elist₁.array;

Elist.addr := t; Elist.ndim := m}

Left → **Elist** { Left.addr := newtemp;

Left.offset := newtemp; **base - (low₁ * n₂ - low₂) * w**

gencode(Left.addr, ':=' , base(Elist.array), '-', invariant(Elist.array));

((i₁ * n₂) + i₂) * w gencode(Left.offset, ':=' , Elist.addr, '*', w)}

赋值语句的翻译模式

E \rightarrow **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 \rightarrow **E**₁ + **E**₂ {E.addr := newtemp;
gencode(E.addr, ':=', E₁.place, '+', E₂.addr)}

E \rightarrow (**E**₁) {E.addr := E₁.addr }

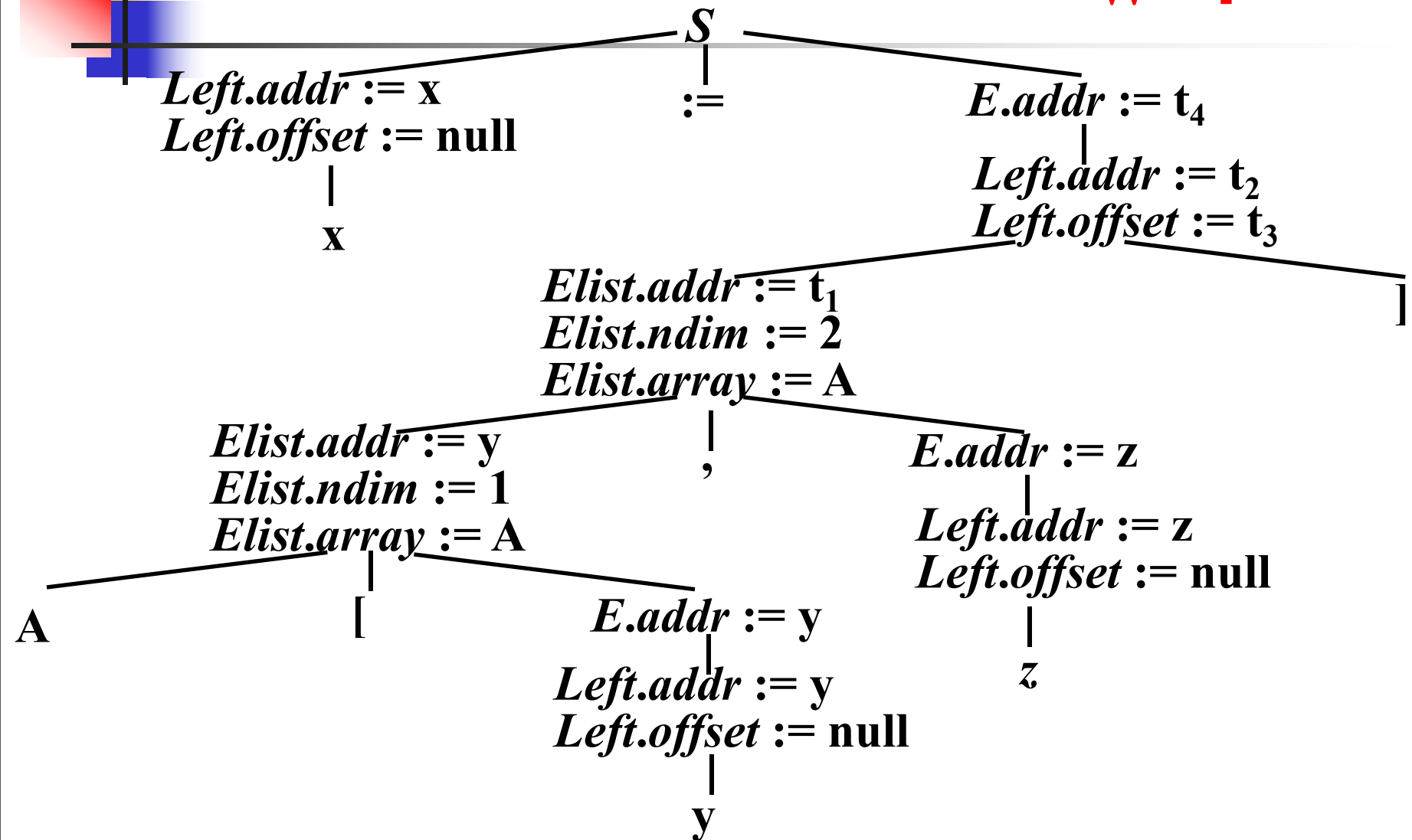
S \rightarrow **Left** := **E** {if Left.offset=null then /*Left是简单变量*/
gencode(Left.addr, ':=' , E.addr)

else
gencode(E.addr, '[', Left.offset, ']', ':=', E.addr)}

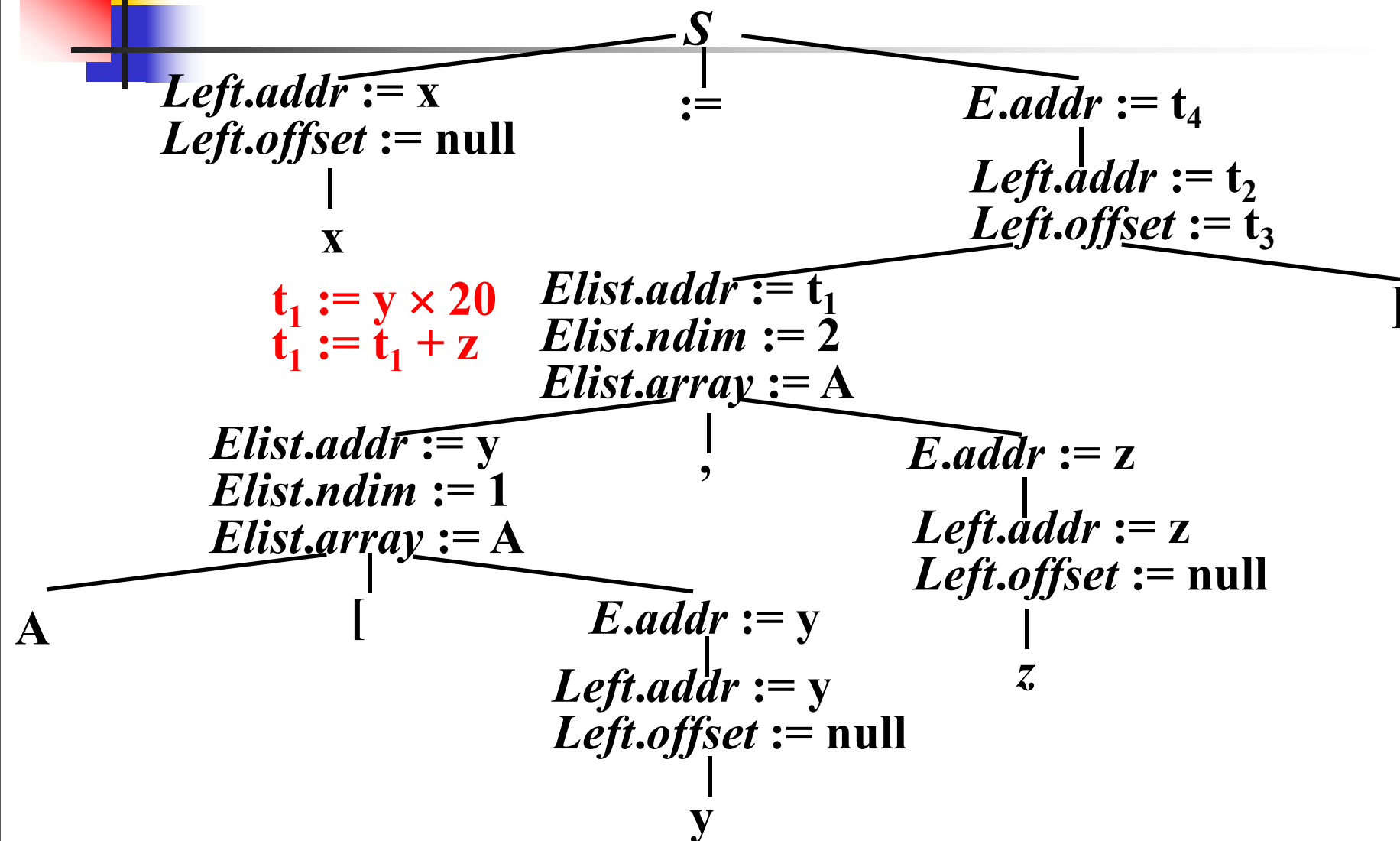
$$((i_1 * n_2) + i_2) * w + (base - (low_1 * n_2 - low_2) * w)$$

例：设A是一个 10×20 的整型数组

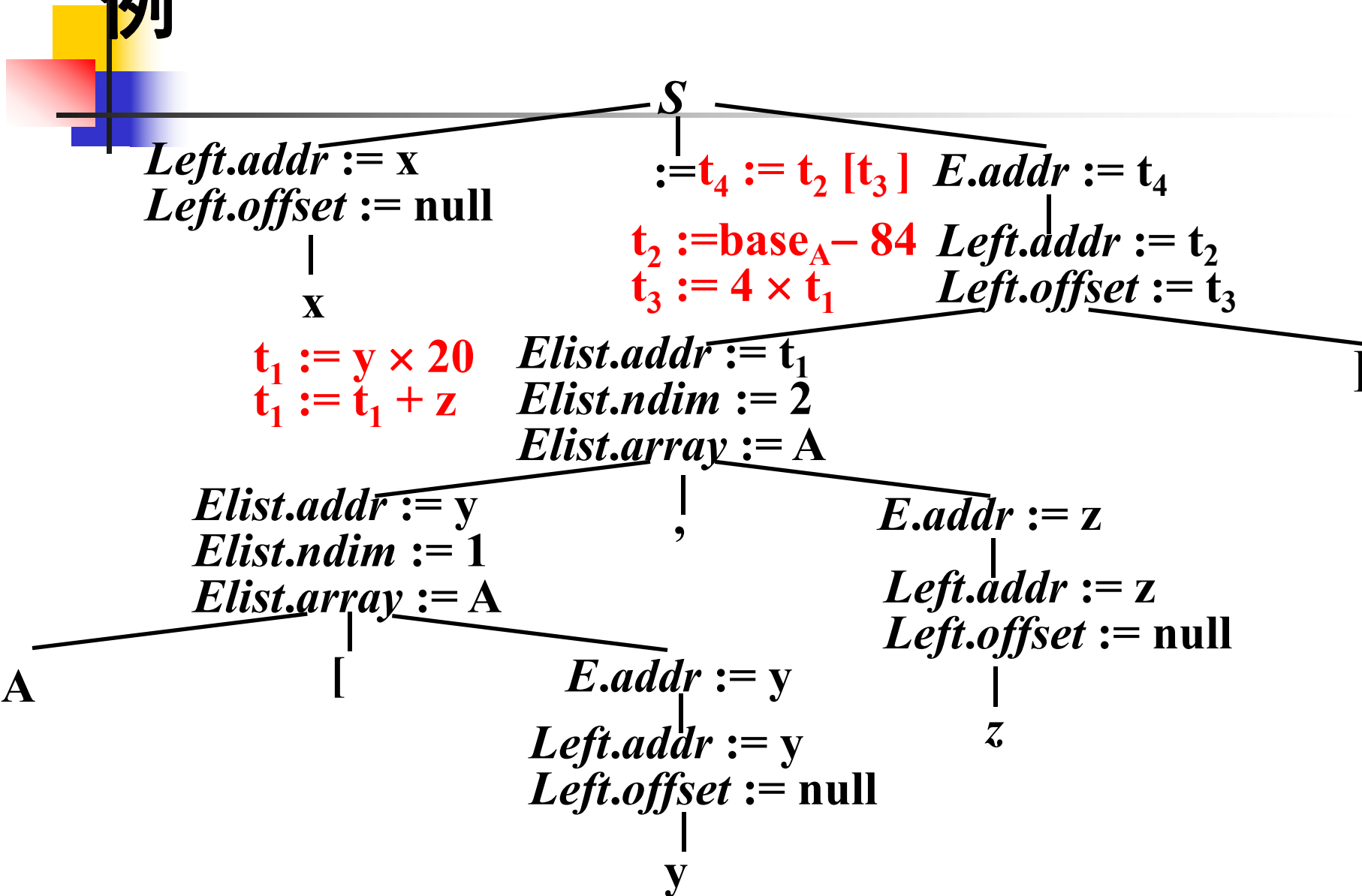
——w=4



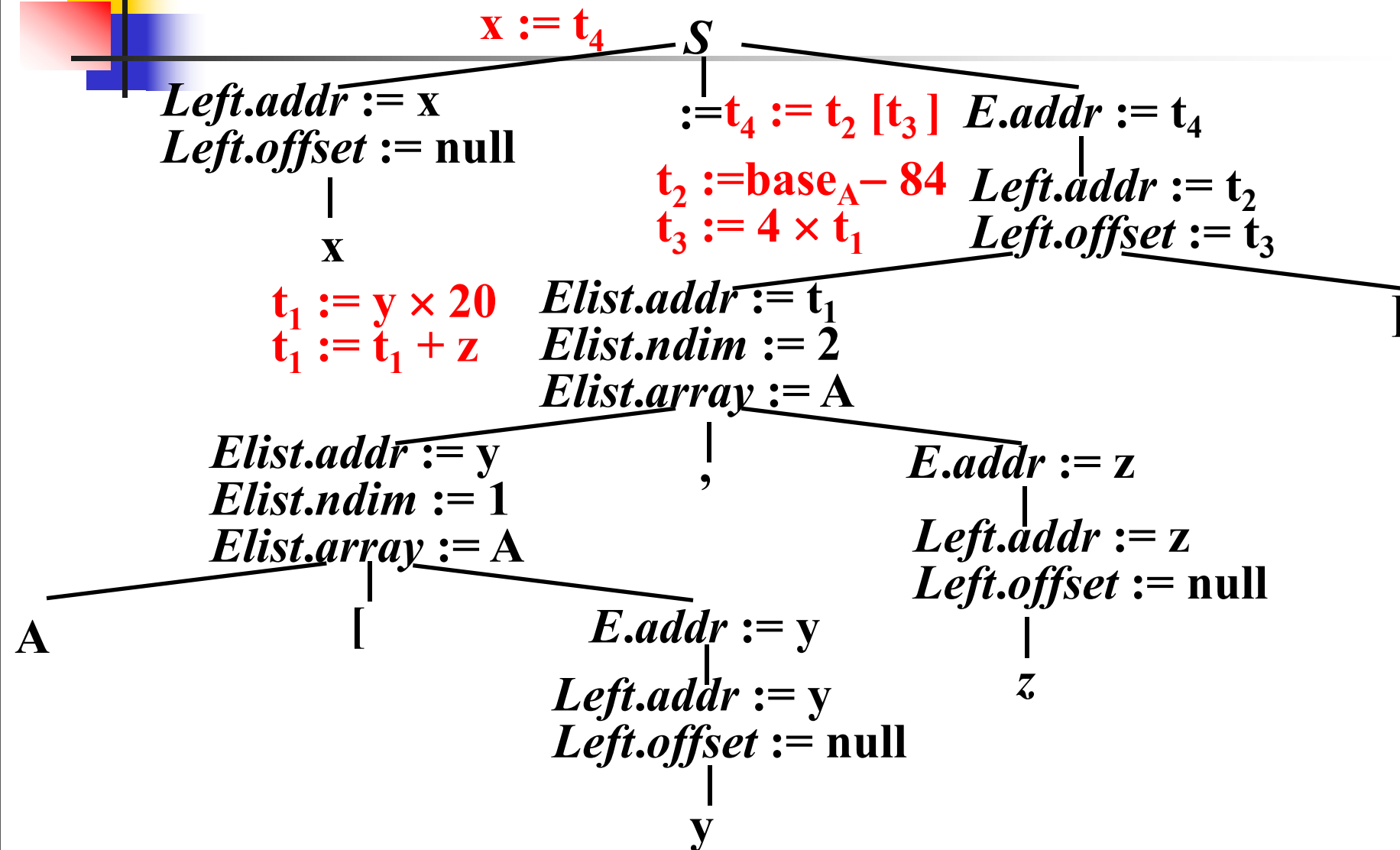
例







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 α 和 β , f 具有类型 $\alpha \rightarrow \beta$ and x 具有类型 α



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 $E_1.type = \text{integer}$ and $E_2.type = \text{integer}$ then begin

gencode(E.addr, ':=', $E_1.addr$, 'int+', $E_2.addr$);

E.type = integer

end

else if $E_1.type = \text{integer}$ and $E_2.type = \text{real}$ then begin

u := newtemp;

gencode(u, ':=', 'inttoreal', $E_1.addr$);

gencode(E.addr, ':=', u, 'real+', $E_2.addr$);

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* := '*B₁.addr* or '*B₂.addr*')}

$B \rightarrow B_1 \text{ and } B_2$ {*B.addr* := newtemp;
gencode(*B.addr* := '*B₁.addr* and '*B₂.addr*')}

$B \rightarrow \text{not } B_1$ {*B.addr* := newtemp;
gencode(*B.addr* := ' 'not'*B₁.addr*')}



用数值表示布尔值的翻译

$B \rightarrow (B_1)$ {B.addr:= B₁.addr}

$B \rightarrow E_1 \text{ relop } E_2$ {B.addr := newtemp;

gencode('if'E₁.addr relop.op E₂.addr'goto'nextquad+3);

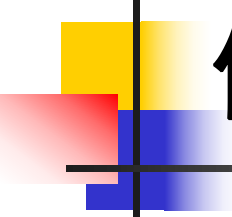
gencode(B.addr ':= '0') ;

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: $t_1 := 0$

102: goto 104

103: $t_1 := 1$

104: if $c < d$ goto 107

105: $t_2 := 0$

106: goto 108

107: $t_2 := 1$

108: if $e < f$ goto 111

109: $t_3 := 0$

110: goto 112

111: $t_3 := 1$

112: $t_4 := t_2$ and t_3

113: $t_5 := t_1$ or t_4



7.5.2 常见控制结构的翻译

■ 文法

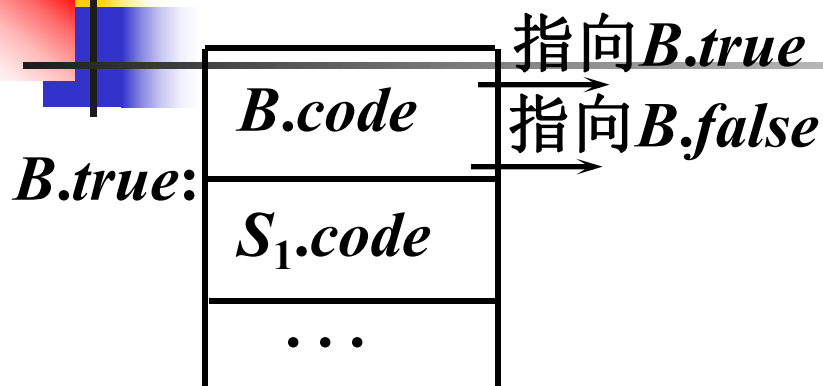
- $S \rightarrow \text{if } B \text{ then } S_1$
- $S \rightarrow \text{if } B \text{ then } S_1 \text{ else } S_2$
- $S \rightarrow \text{while } B \text{ do } S_1$
- $S \rightarrow \text{begin } Slist \text{ end}$
- $Slist \rightarrow Slist; S \mid S$
- B 是控制结构中的布尔表达式
- 函数newlabel返回一个新的语句标号
- 属性 $B.true$ 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



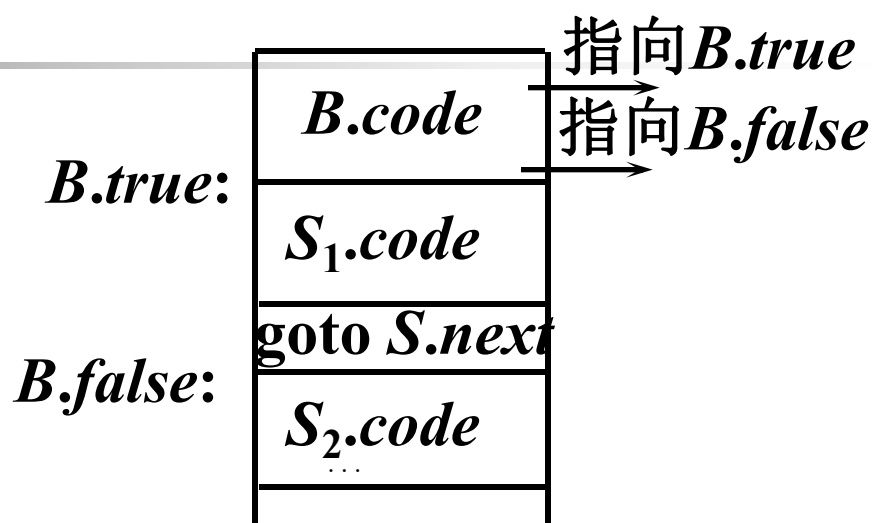
7.5.2 常见控制结构的翻译

- 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

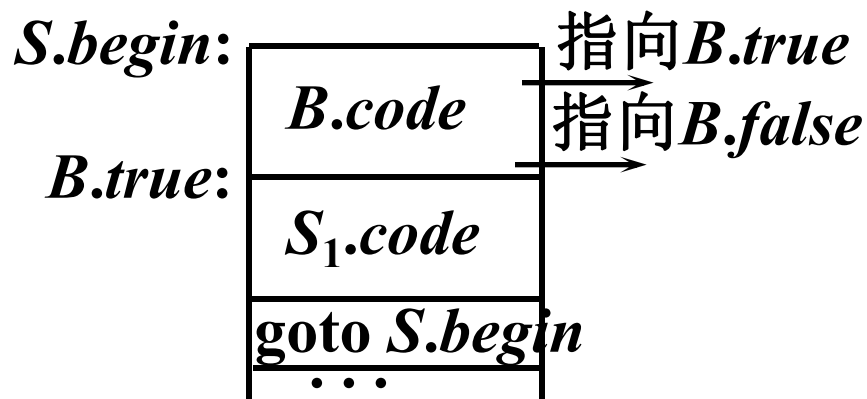
7.5.2 常见控制结构的翻译



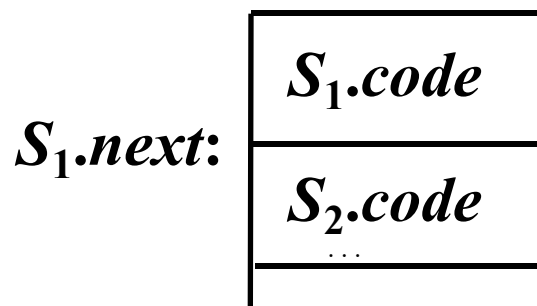
(a) if-then



(b) if-then-else



(c) while-do



(d) *S₁*; *S₂*

7.5.2 常见控制结构的翻译

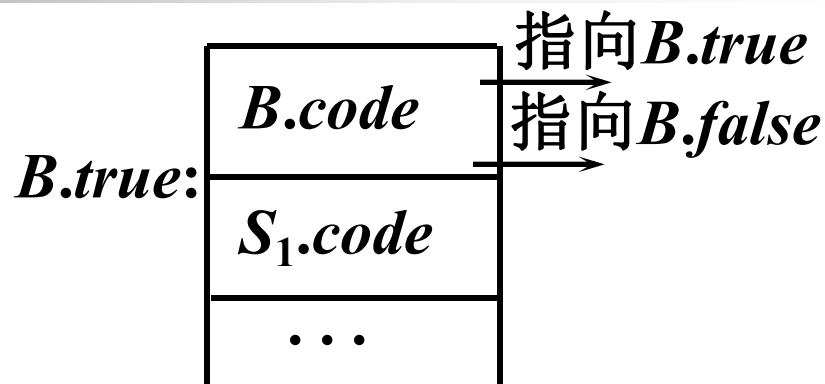
$S \rightarrow \text{if } B \text{ then } S_1$

$\{B.\text{true} := \text{newlabel};$

$B.\text{false} := S.\text{next};$

$S_1.\text{next} := S.\text{next};$

$S.\text{code} := B.\text{code} || \text{gencode}(B.\text{true}, ':') || S_1.\text{code} \}$

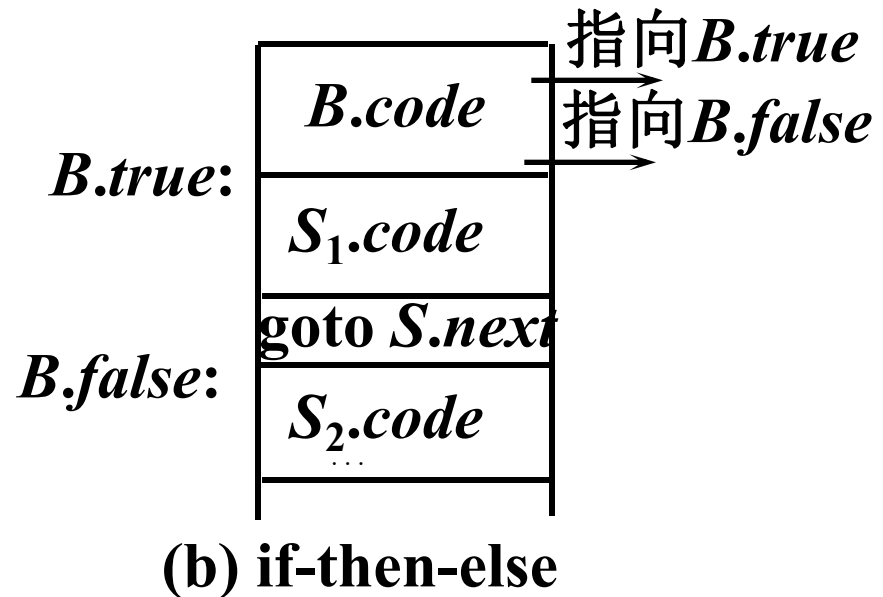


(a) if-then

7.5.2 常见控制结构的翻译

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

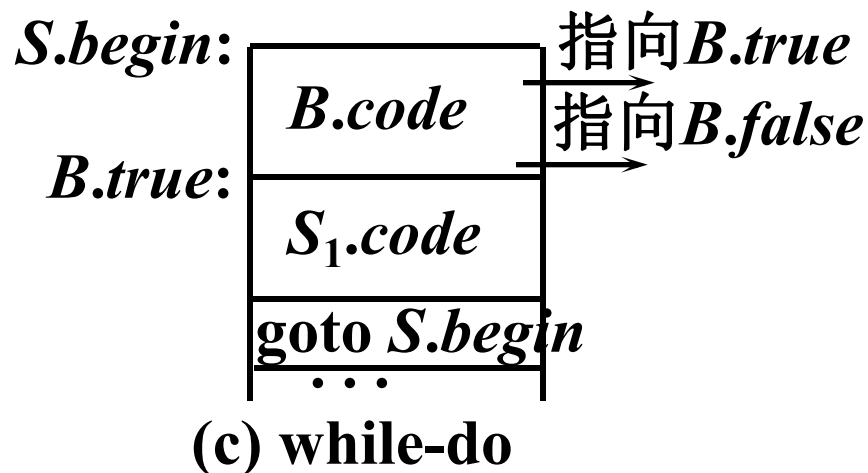
```
{B.true := newlabel;  
  B.false := newlabel;  
  S1.next := S.next;  
  S2.next := S.next;  
  S.code := B.code || gencode(B.true, ':') || S1.code ||  
  gencode('goto', S.next) || gen(B.false, ':') || S2.code}
```



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)}
```

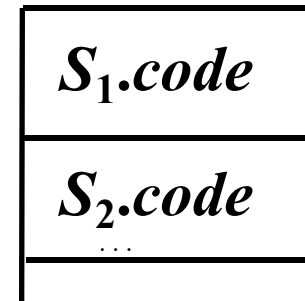


7.5.2 常见控制结构的翻译

$S \rightarrow S_1; S_2$

$\{S_1.next := newlabel; S_2.next := S.next;$
 $S.code := S_1.code || gencode(S_1.next, ':') || S_2.code\}$

$S_1.next:$



(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 $e < f$

假定整个表达式的真假出口已分别置为Ltrue和Lfalse

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

$B \rightarrow (B_1)$ { $B_1.\text{true} := B.\text{true}; B_1.\text{false} := B.\text{false};$
 $B.\text{code} := B_1.\text{code}$ }

$B \rightarrow E_1 \text{ relop } E_2$

{ $B.\text{code} := \text{gencode}(\text{'if' } E_1.\text{addr relop } E_2.\text{addr 'goto' } B.\text{true});$
 || $\text{gencode}(\text{'goto' } B.\text{false})$ }

$B \rightarrow \text{true}$ { $B.\text{code} := \text{gencode}(\text{'goto' } B.\text{true})$ }

$B \rightarrow \text{false}$ { $B.\text{code} := \text{gencode}(\text{'goto' } B.\text{false})$ }



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

L₁: if a < b goto L₂

goto L_{next}

B₁.code

L₂: if c < d goto L₃

goto L₄

B₂.code

L₃: t₁ := y + z

x := t₁

S₁.code

goto L₁

S₃.code

L₄: t₂ := y-z

x := t₂

S₂.code

goto L₁

L_{next}:

7.5.4 混合模式的布尔表达式翻译

$E \rightarrow E_1 \text{ relop } E_2 \mid E_1 + E_2 \mid E_1 \text{ and } E_2 \mid \text{id}$

- $E_1 + E_2$ 、 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:=newtemp; E.code:= $E_1.code \parallel E_2.code$

 gencode(E.addr:='E₁.addr '+'E₂.addr) end

else if $E_1.type = \text{arith}$ and $E_2.type = \text{bool}$ then begin

 E.addr:=newtemp; E₂.true:=newlabel; E₂.false:=newlabel;

 E.code:= $E_1.code \parallel E_2.code$ gencode(E₂.true:='E.addr ':='E₁.addr+1)

\parallel gencode('goto' nextstate+2) \parallel gencode(E₂.false:='E.addr ':='E₁.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 L_{false}

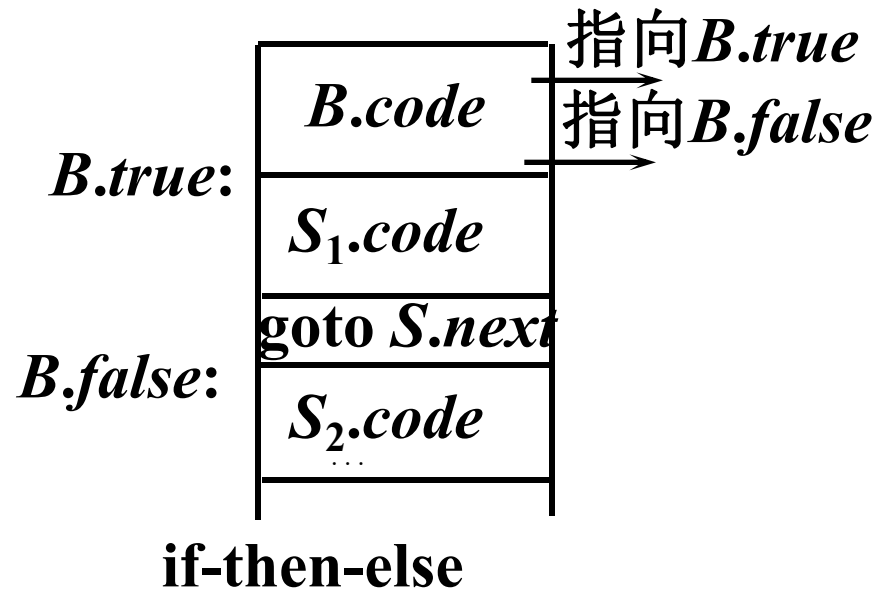
L_1 : if d goto L_{true}

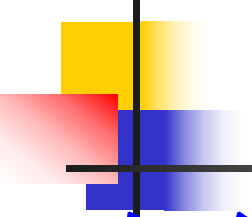
goto L_{false}

回顾： 7.5 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}
```





7.6 回填

- 如何实现7.5节的控制流语句的翻译？
- 一、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
- 二、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



7.6 回填

■ 方法：采用回填技术

- 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**”到链表中记录的相应指令中
- **关键**：引入合适的属性、设计合适的语义动作
 - 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所指向的链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

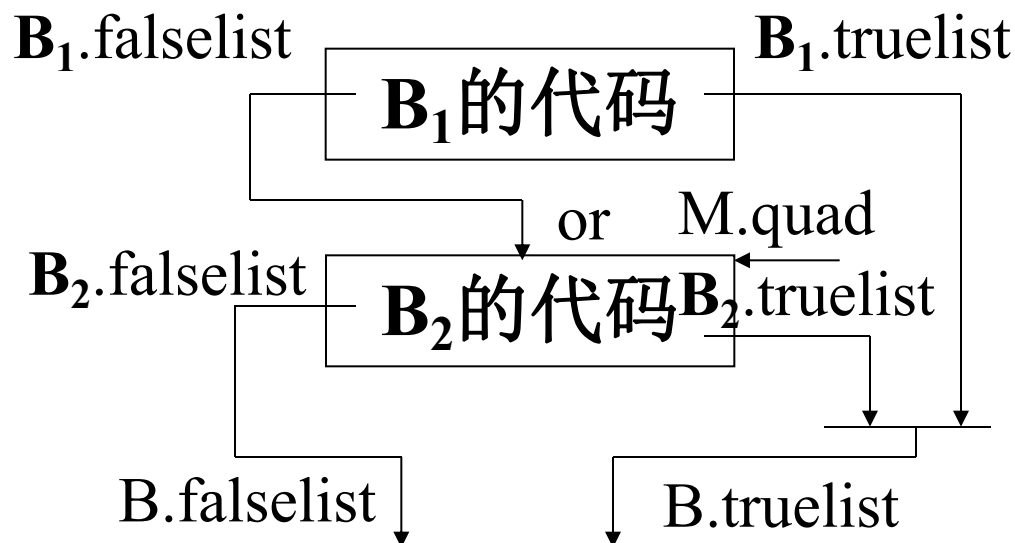
7.6.1 布尔表达式的回填式翻译

B \rightarrow **B₁** or **M B₂**

```
{backpatch(B1.falselist, M.quad);  
  B.truelist:=merge(B1.truelist, B2.truelist);  
  B.falselist := B2.falselist}
```

M \rightarrow ϵ

```
{M.quad:=nextquad}
```



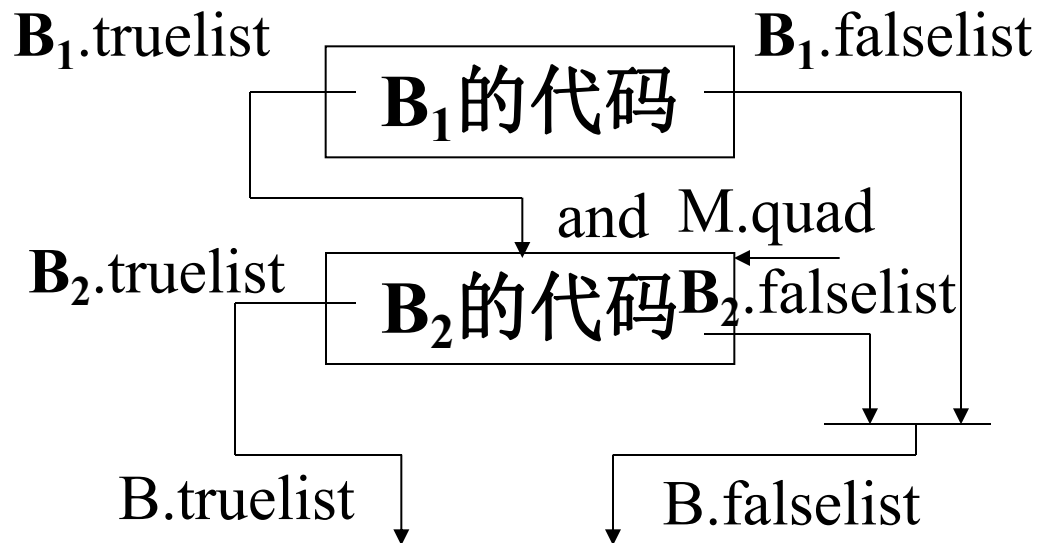
7.6.1 布尔表达式的回填式翻译

$B \rightarrow B_1 \text{ and } M B_2$

{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 true

```
{B.truelist := makelist(nextquad);  
  gencode('goto-')}
```

B \rightarrow 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

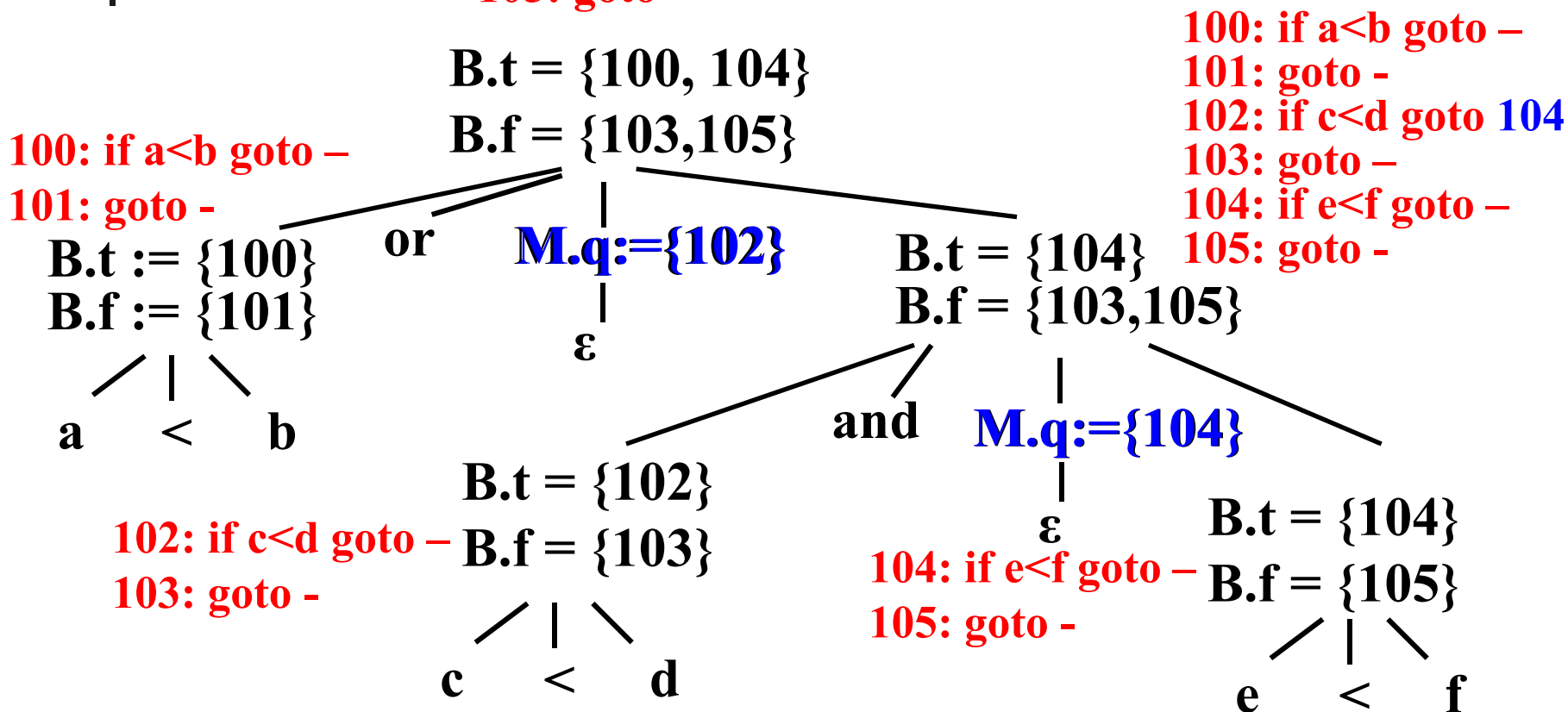


图7.27 a<b or c<d and e<f的注释分析树

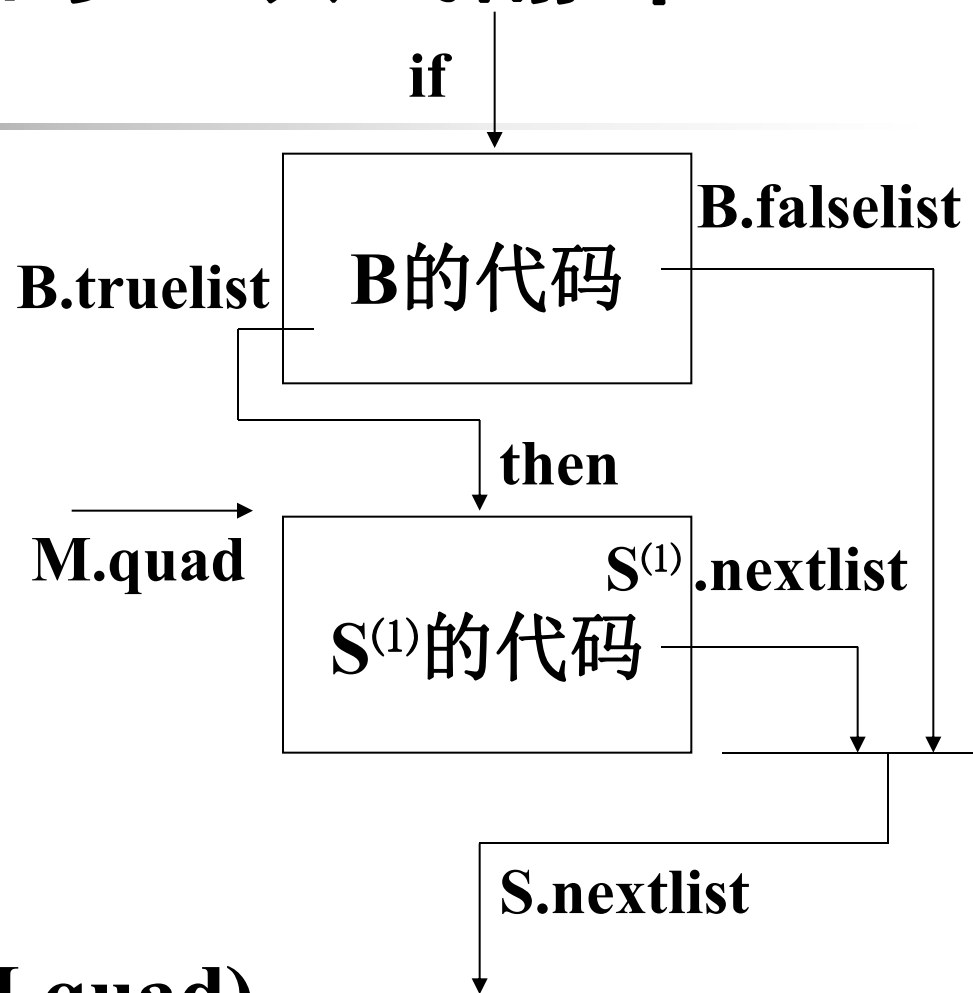
7.6.2 常见控制结构的回填式翻译

$S \rightarrow$ 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

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

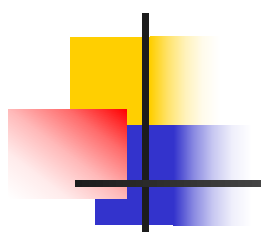
$N \rightarrow \epsilon$ { $N.\text{nextlist} := \text{makelist}(\text{nextquad});$
 $\text{gencode}(\text{'goto -})$ }

if-then语句的回填式翻译

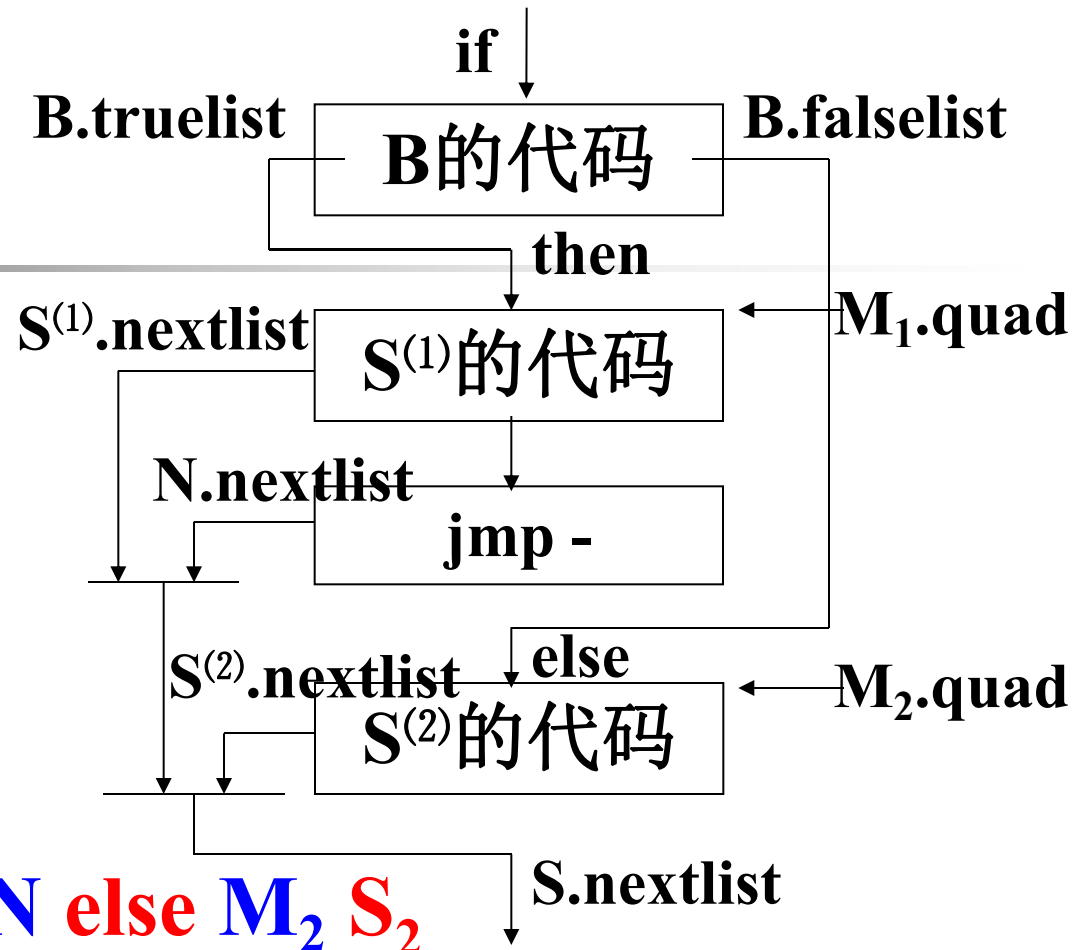


$S \rightarrow \text{if } B \text{ then } M \ S_1$

$\{\text{backpatch}(\text{B.truelist}, \text{M.quad})$
 $\text{S.nextlist} := \text{merge}(\text{B.falselist}, \text{S}_1.\text{nextlist})\}$

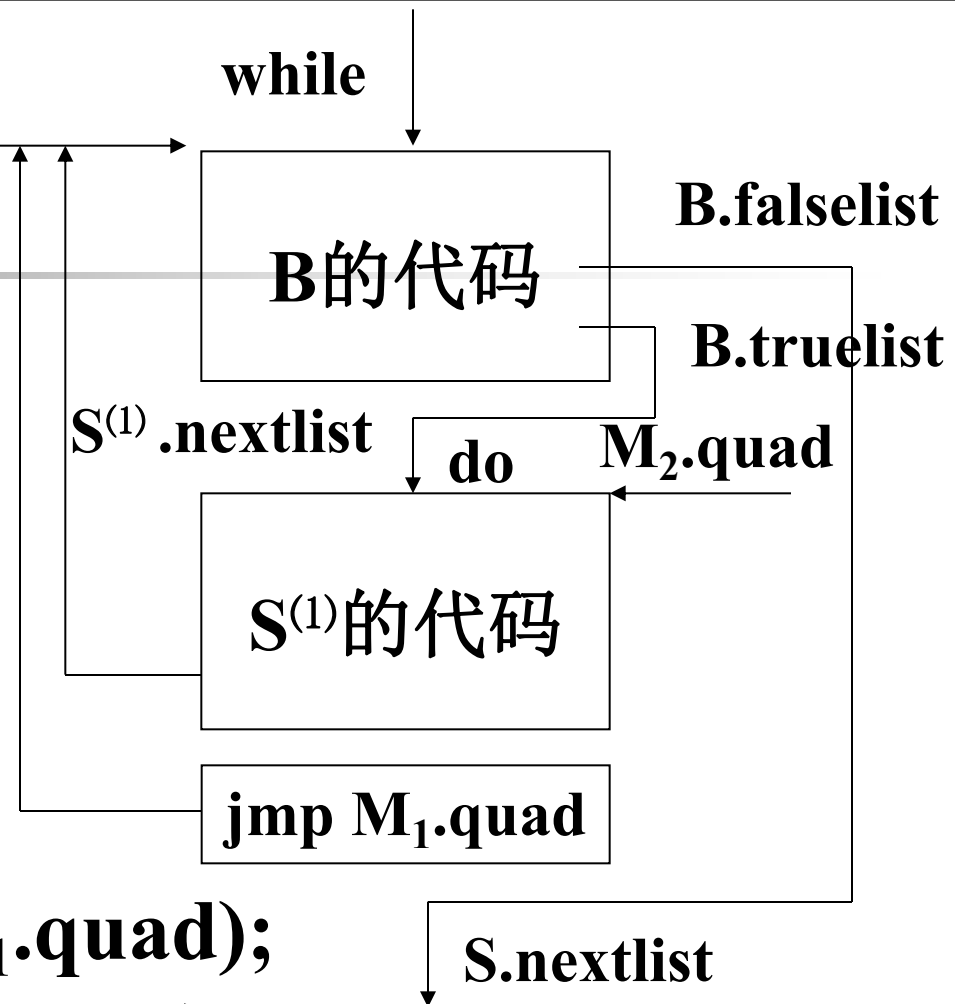


if-then-else语句的回填式翻译



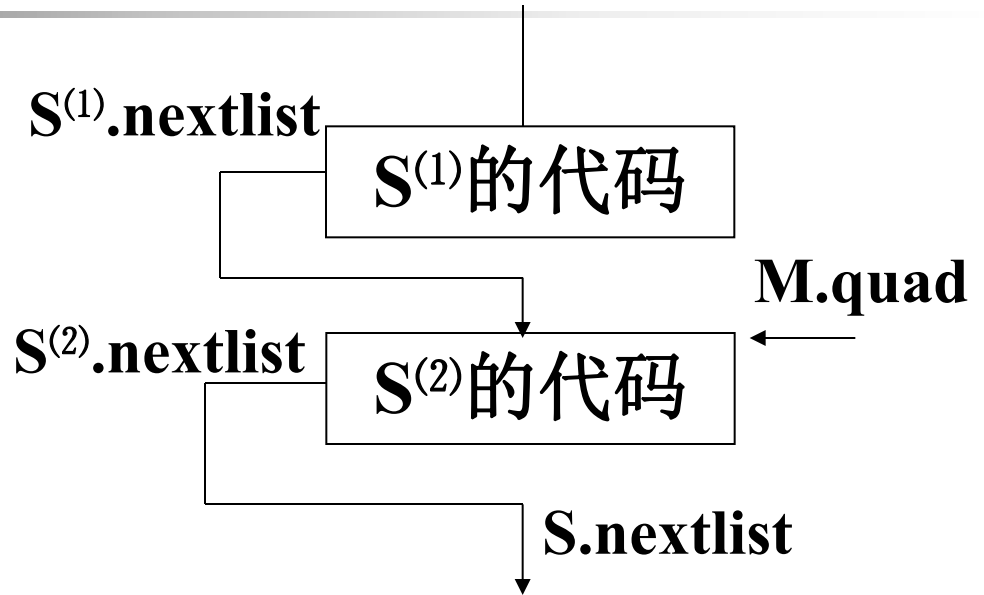
$S \rightarrow \text{if } B \text{ then } M_1 S_1 N \text{ else } M_2 S_2$
{backpatch(B.truelist, $M_1.\text{quad}$);
backpatch(B.falselist, $M_2.\text{quad}$);
 $S.\text{nextlist} : =$
merge($S_2.\text{nextlist}$, merge($N.\text{nextlist}$, $S_1.\text{nextlist}$))}

while语句的 回填式翻译



$S \rightarrow \text{while } M_1 \text{ B do } M_2 S_1$
{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

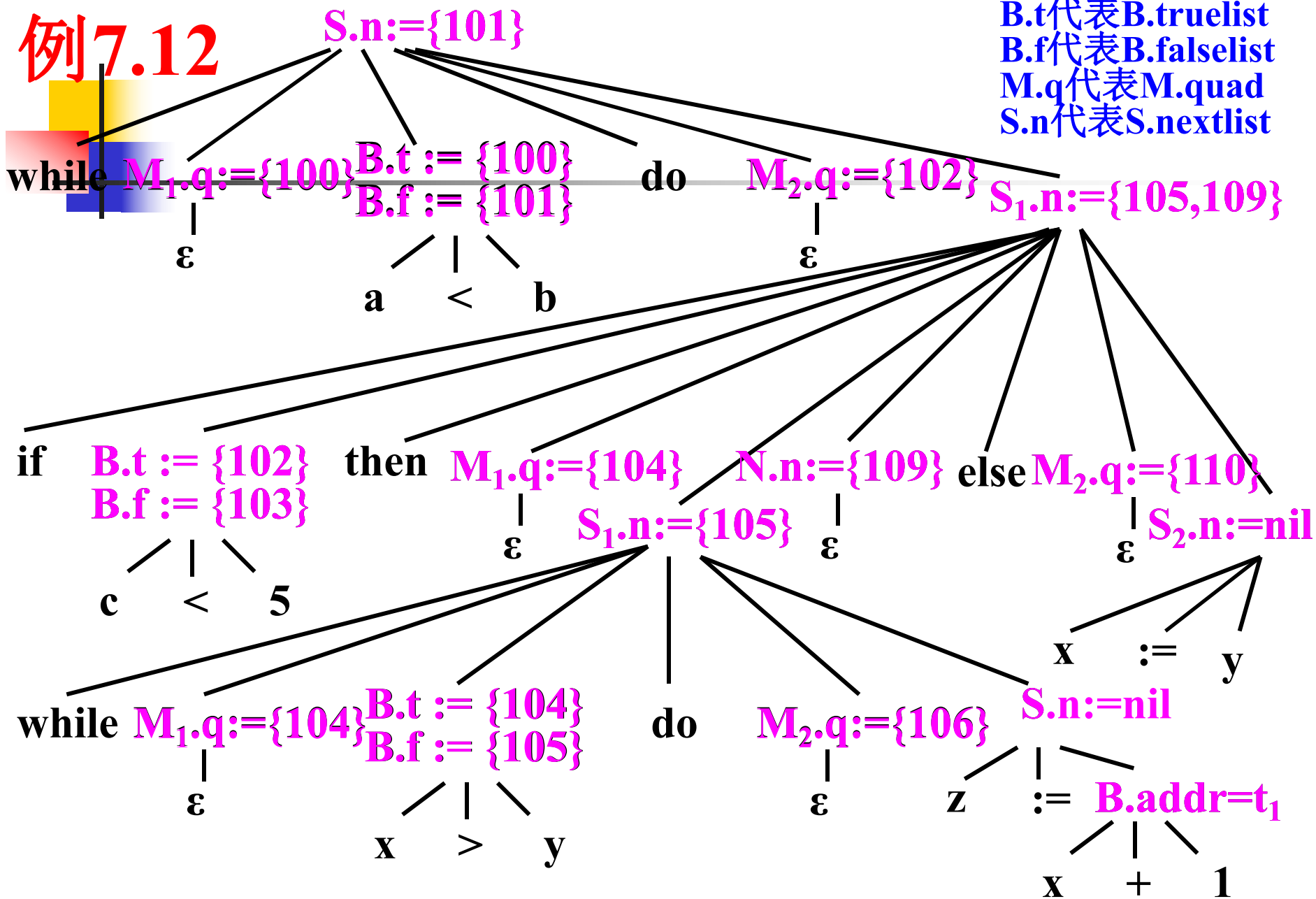
if $c < 5$ then

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



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,t₁)

107: (: =, t₁,-,z)

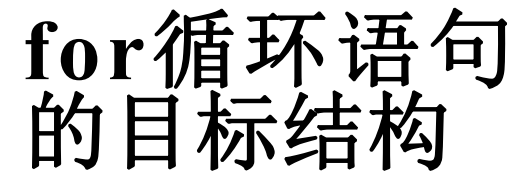
108: (j,-,-,104)

109: (j,-,-,100)

110: (: =,y,-,x)

111: (j,-,-,100)

112:





7.6.3 for循环语句的回填式翻译

$S \rightarrow \text{for id} := E_1 \text{ to } E_2 \text{ step } E_3 \text{ do } M S_1$
 $\{ \text{backpatch}(S_1.\text{nextlist}, M.\text{again},);$
 $\text{gencode}(\text{'goto'}, -, -, M.\text{again});$
 $S.\text{nextlist} := M.\text{nextlist}; \}$

7.6.3 for循环语句的回填式翻译

$M \rightarrow \varepsilon$

```
{M.addr := entry(id); gencode(':', E1.addr, -, M.addr);  
T1 := newtemp; gencode(':', E2.addr, -, T1);  
T2 := newtemp; gencode(':', E3.addr, -, T2);  
q := nextquad; gencode('goto', -, -, q+2);  
M.again := q+1; gencode('+', M.addr, T2, M.addr);  
M.nextlist := nextquad;  
gencode('if' M.addr '>' T1 'goto -');}
```


7.6.4 repeat语句的回填式翻译

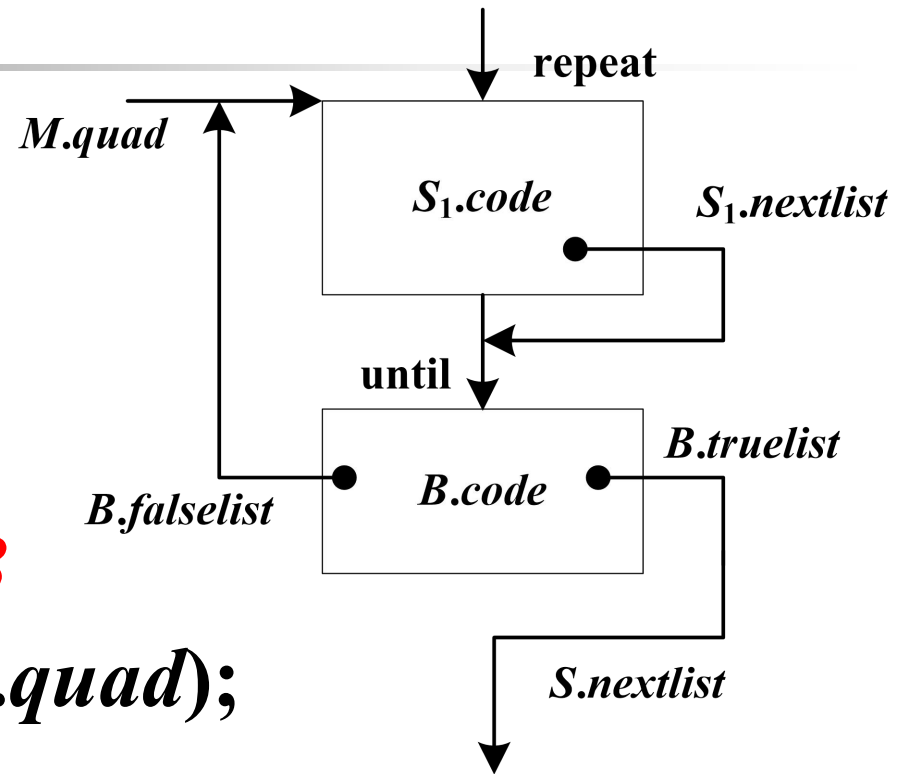
$S \rightarrow \text{repeat } M \text{ } S_1 \text{ until } N \text{ } B$

$\{ \text{backpatch}(B.\text{falselist}, M.\text{quad});$

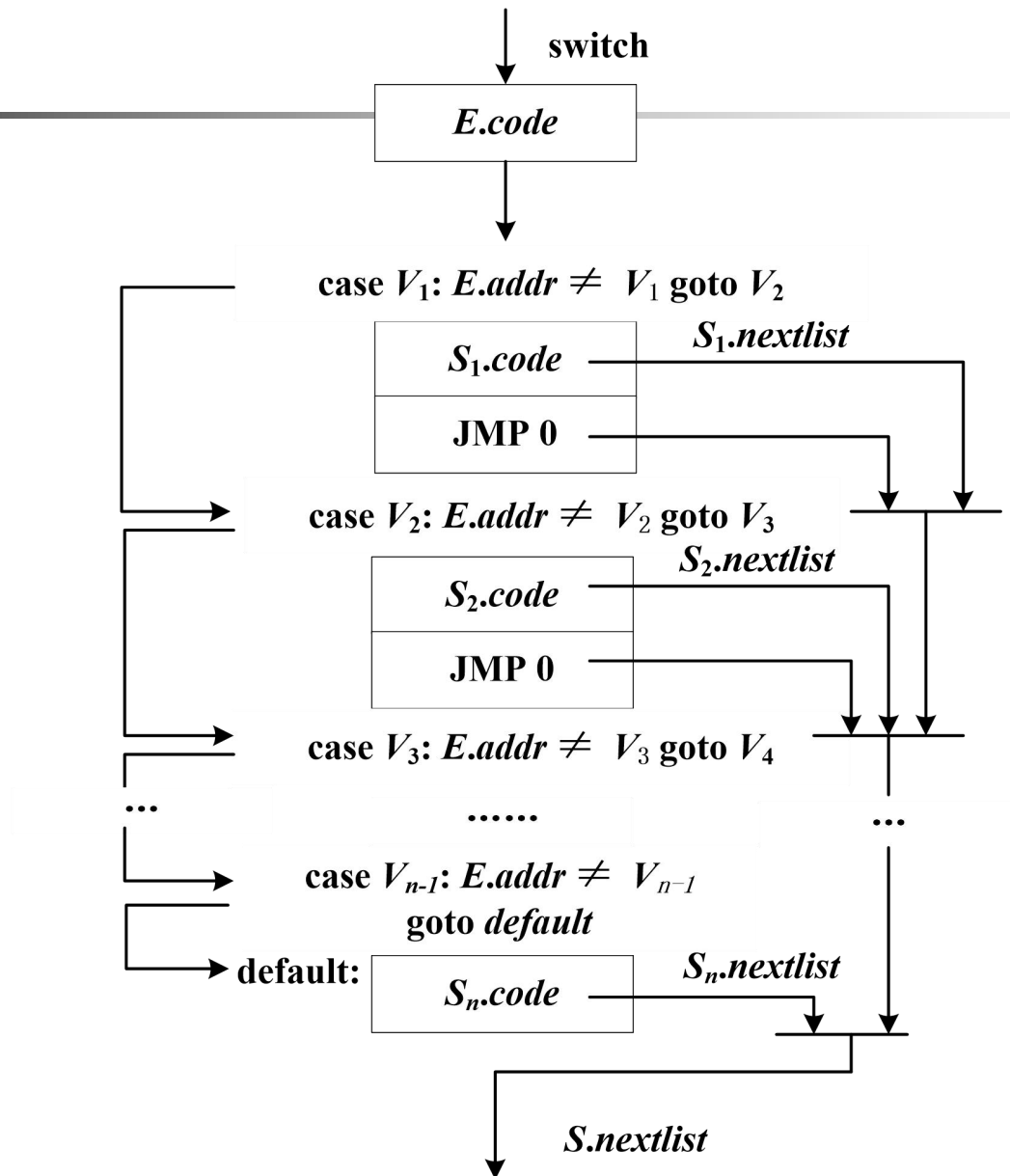
$S.\text{nextlist} := B.\text{truelist} \}$

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

$N \rightarrow \epsilon \{ \text{backpatch}(S_1.\text{nextlist}, \text{nextquad}) \}$



7.7 switch语句的翻译





7.7.2 switch语句的语法制导翻译

$S \rightarrow \text{switch } (E)$ { $i := 0$; $S_i.\text{nextlist} := 0$; push
 $S_i.\text{nextlist}$; push $E.\text{addr}$;

push i ; $q := 0$; push q }

$Clist$ { pop q ; pop i ; pop $E.\text{addr}$; pop

$S_i.\text{nextlist}$; $S.\text{nextlist} := \text{merge}(S_i.\text{nextlist}, q)$;
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_i* 'goto' *Li*);
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*} **Clist**

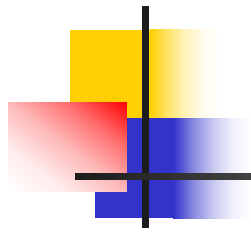
7.7.2 switch语句的语法制导翻译

Clis~~t~~→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 :



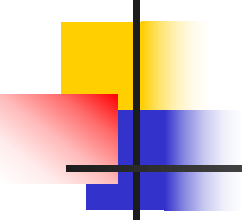
7.8 过程调用和返回语句的翻译

$S \rightarrow \text{call id (Elist)}$

$\text{Elist} \rightarrow \text{Elist}, E$

$\text{Elist} \rightarrow E$

$S \rightarrow \text{return } E$



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 $\text{id}.\text{addr}, n$



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*/

$EList \rightarrow E K (\epsilon|, EList)$

$M \rightarrow \epsilon \{gencode('prog', id, y, -)\}$

/* y 的值表示input,output或两者皆有*/

$N \rightarrow \epsilon \{设置一个语义队列Queue\}$

$L \rightarrow \epsilon \{T:=entry(id); add(Queue, T)\}$

$K \rightarrow \epsilon \{T:= E.addr; add(Queue, T)\}$



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添加到符号表中；
- 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。