



# 第十章 中间语言代码翻译

## 2024

赵伟



# 第十章 中间语言代码生成

- ▶ 中间语言
- ▶ 符号表的使用
- ▶ 面向下列语法单元的代码生成（也称为翻译）：
  - 表达式：算术表达式；布尔表达式
  - 语句：赋值；分支；循环；复合语句等
  - 数组引用
  - 函数调用



# 10.1 中间语言

- ▶ 中间语言是介于编译器前端和后端之间的接口
- ▶ 中间语言的作用及其特点
  - 容易把高级语言翻译为它；
  - 容易把它翻译为机器语言；
  - 容易进行代码优化。
- ▶ 中间语言的设计要点：
  - 抽象层次高低兼顾；
  - 有通用性；
  - 粒度大小有针对性。



## ➤ 线性IR

- 逆波兰表示：如将  $E \text{ op } T$  翻译为  $E' T' \text{ op}$
- 三地址代码
- 四元式、三元式

## ➤ 基于图的IR

- 树（抽象语法树）
- 有向无环图



➤  $x = y \text{ op } z$

- $x$ 和 $y$ 是左值引用，只能是变量
- $z$ 是右值引用，可以是变量或常量
- $\text{op}$ 是二元运算符

➤  $x = \text{uop } z$

- $\text{uop}$ 是一元运算符

➤  $x = z$

➤ 例：  $[x = -y]$                        $//$ 方括号表示一个代码片段

➤ 例：表达式  $2*a + (b-3)$  可翻译成  $[t1 = 2*a; t2 = b-3; t3 = t1+t2]$

➤ 或者换行为

$$\begin{aligned} &[t1 = 2*a \\ &\quad t2 = b-3 \\ &\quad t3 = t1+t2] \end{aligned}$$

➤ 其中 $t1$ ， $t2$ 和 $t3$ 为临时变量，调用 $\text{newvar}()$ 返回一个新的。



# 三地址指令

## 指令列表

$d = d \text{ op } r$

$d = \text{uop } r$

$d = r$

LABEL I

GOTO I

IF d rop r THEN I ELSE I'

$d = M[r]$

$M[r] = r$

PAR d

$d = \text{CALL } f, f@argc$

RETURN d

## 指令解释

取地址:  $\text{addr } d;$

r右值; d变量; I标号; f函数名

$\text{parlist} \rightarrow d \mid d, \text{parlist}$

## 特殊指令IF

[IF d rop r THEN I ELSE I'

LABEL I; ...

LABEL I';...]

访存  $M[r]$ 、 $M[\text{fp}+\text{offset}]$

函数调用  $f(\text{parlist})$  指令

PAR  $d_1$

..

PAR  $d_{\text{argc}}$

$d = \text{CALL } f, \text{argc}$



## 例：三地址代码

```
input x;  
if 0 < x then  
    fact = 1  
else fact=2;  
repeat  
    fact = fact * x;  
    x = x - 1  
until x == 0;  
print fact
```

```
INPUT x  
t1 = 0  
IF t1 < x THEN I1 ELSE I2  
LABEL I1  
fact = 1  
GOTO I3  
LABEL I2  
fact = 2  
LABEL I3  
fact = fact * x  
x = x - 1  
IF x == 0 THEN I4 ELSE I3  
LABEL I4  
PRINT fact
```



## 例：三地址代码

```
int x;  
int fact(int n; int a){  
    if (n==1) return a  
    else return fact (n-1, n*a,)  
};  
x=123+fact(5,1,);  
print x
```

注意：中间代码生成的目标是生成函数的代码，其中用到各个语法单元的代码生成功能。

```
@code=[  
    t4=123; t5=3; t6=1;  
    PAR t6; PAR t5;  
    t7=CALL fact, 2;  
    x=t4+t7;  
    PRINT x]
```

```
fact@code=[  
    IF n==1 THEN I1 ELSE I2;  
    LABEL I1; RETURN a;  
    GOTO I3;  
    LABEL I2;  
    t1=n-1; t2=n*a;  
    PAR t2; PAR t1;  
    t3=CALL fact, 2;  
    RETURN t3;  
    LABEL I3]
```





# 三地址代码实现为四元式

## 三地址代码

$$d = r \text{ op } r$$

## 四元式: $k(\text{op}, \text{arg1}, \text{arg2}, \text{result})$

- $k$ 是四元式的编号，实现为存储单元地址，连续编号；
- $\text{op}$ 是一个二元(也可一元或零元)运算符；
- $\text{arg1}, \text{arg2}$ 分别为运算对象(可以缺省)；
- 运算结果为 $\text{result}$ 。



## 例：从三地址代码到四元式

[INPUT x

**t1 = 0**

IF t1 < x THEN I1 ELSE I2

LABEL I1

fact = 1

GOTO I3

LABEL I2

fact = 2

LABEL I3

fact = fact \* x

x = x - 1

IF x == 0 THEN I4 ELSE I3

LABEL I4

PRINT fact]

100 (INPUT, x, \_, \_)

101 (J<, **0**, x, 103)

102 (J, \_, \_, 105)

103 (=, 1, \_, fact)

104 (J, \_, \_, 106)

105 (=, 2, \_, fact)

106 (\*, fact, x, fact)

107 (-, x, 1, x)

108 (J=, x, 0, 110)

109 (J, \_, \_, 106)

110 (PRINT, fact, \_, \_)

四元式适用于增量式代码生成模式，三地址代码除此还适合代码作为属性值的生成模式。

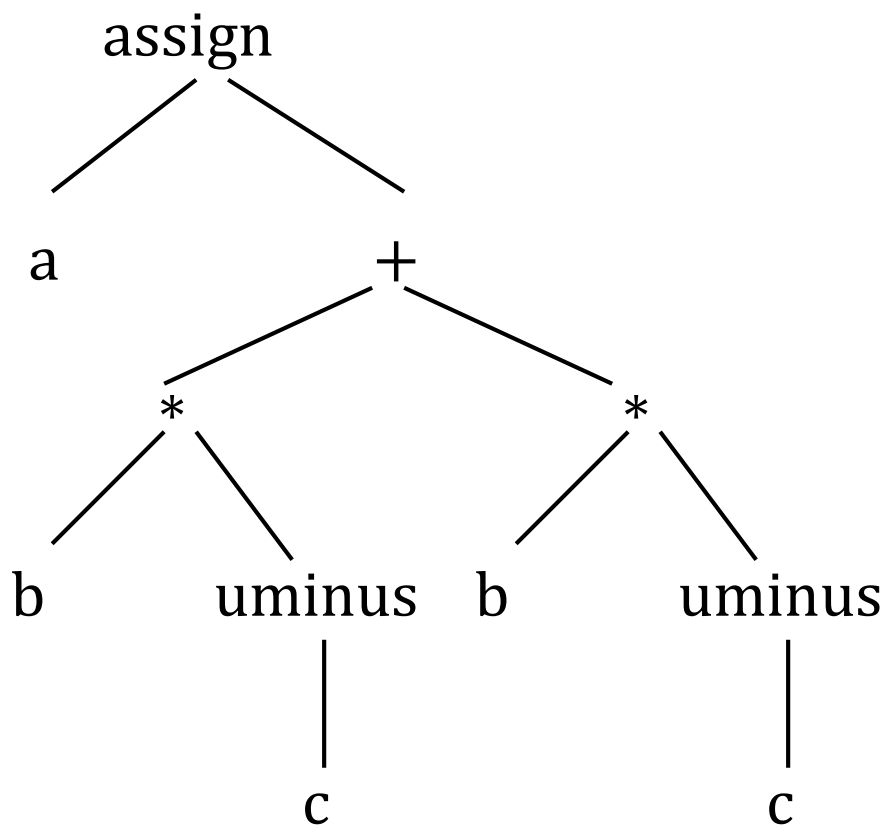


# 基于图的中间表示

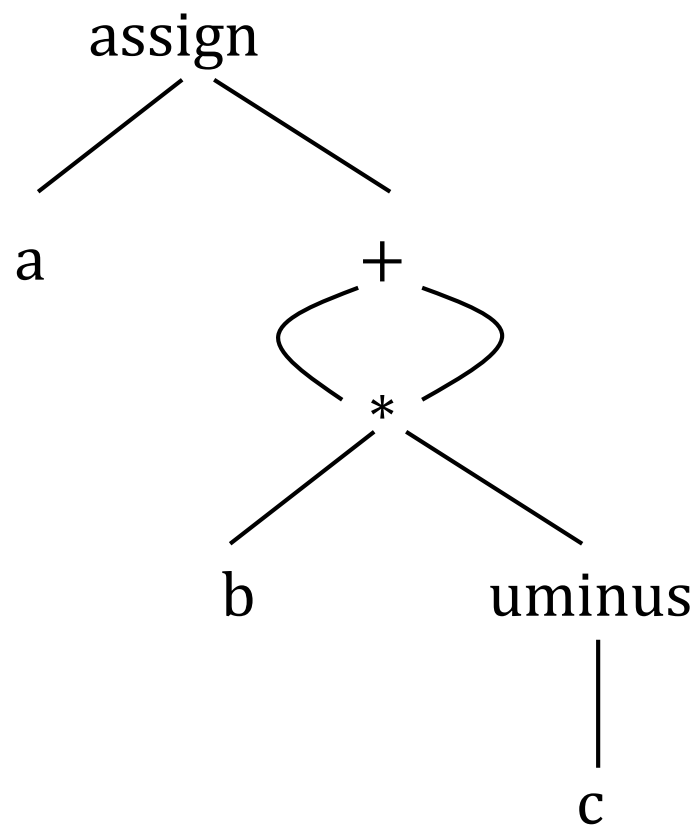
- ▶ 抽象语法树（不含声明语句）
  - 内结点均为操作符，叶子都是操作数
  - 不依赖于源语言文法（避免由修剪产生的干扰）
  - 不会表现出文法的全部细节（如括号）
  - 不包括声明语句
  
- ▶ 有向无循环图(Directed Acyclic Graph, 简称DAG)
  - 对表达式中的每个子表达式，DAG中都有一个结点
  - 一个内部结点代表一个操作符，它的孩子代表操作数
  - 在一个DAG中代表公共子表达式的结点具有多个父结点



# 例：AST与DAG



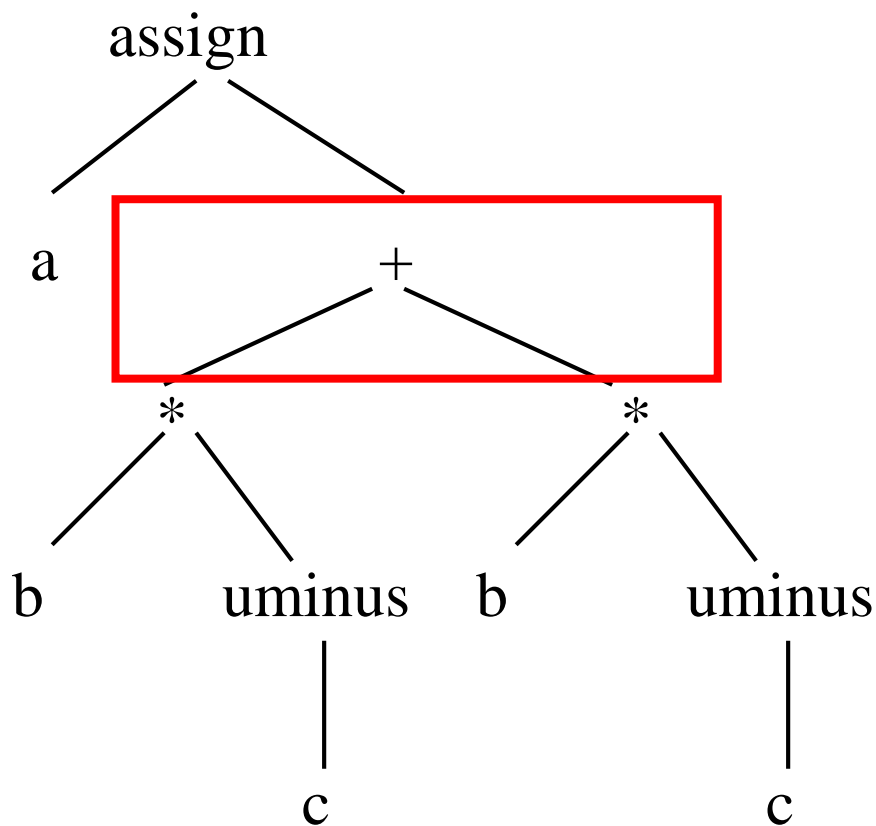
抽象语法树



DAG



# 例：AST与三地址代码



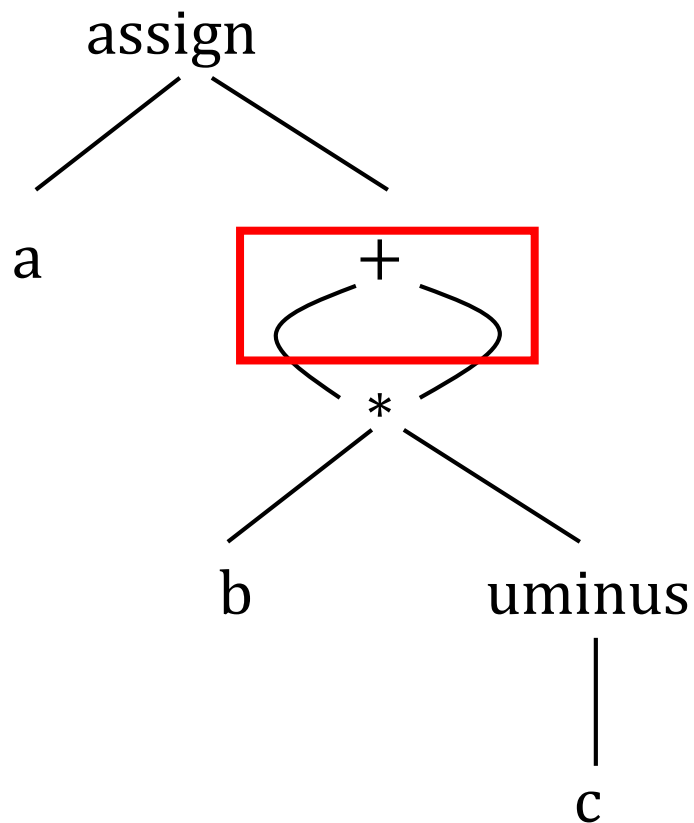
[t1=-c  
t2=b\*t1  
t3=-c  
t4=b\*t3  
t5=t2+t4  
a=t5]

抽象语法树



# 例：DAG与三地址代码

赵银亮



DAG

[t1=-c  
t2=b\*t1  
t3=t2+t2  
a=t3]



# 中间语言的选择

- ▶ 三地址代码作为中间表示
- ▶ 四元式代码作为中间表示
- ▶ 抽象语法树作为中间表示
- ▶ 使用语法制导的属性求值框架来完成翻译任务



- **习题9.1** 利用本章为主文法配套的属性文法，翻译下列声明为符号表表示（提示：共4个符号表），假定对于 $\check{S}$ 的代码生成都没有产生临时变量， $\check{S}$ 的代码用省略号表示。假定FLO类型宽度为8。然后写出h()函数声明语句的带注释语法树或带注释的规范归约，提示：画出语法树并将树中每个变元的属性逐一列出，考虑到树上标注不方便，可对相同变元的多次出现用上标区分如图8-1，然后在树外一一列出即可。

```
int x; int a[5];  
float b[3, 6];  
int g(int r(); int y; float b[];){  
    int c[10];  
     $\check{S}$ };  
int h(int f(); int y;){  
    int g(int c[];){ $\check{S}$ };  
     $\check{S}$ };  
 $\check{S}$ 
```



```
int x; int a[5]; float b[3, 6]; int g(int r(); int y; float b[];){int c[10];Š;
int h(int f(); int y;){ int g(int c[];){Š; Š; Š
```

```
@table:(outer:NULL width:184 rtype:INT argc:0 arglist:NIL level:0 code:...
entry:(name:x type:INT offset:4) entry:(name:a type:ARRAY base:24 ...)
entry:(name:b type:ARRAY base:168 etype:FLO dims:2 dim[0]:3 dim[1]:6)
entry:(name:g type:FUNC offset:176 mytab:g@table)
entry:(name:h type:FUNC offset:184 mytab: h@table))
```

```
g@table:(outer:@table width:56 argc:3 arglist:(r y b) rtype:INT level:1) c
ode:... entry:(name:r type:FUNPTT offset:8)entry:(name:y type:INT offset:
12)entry:(name:b type:ARRPTT offset: 16) entry:(name:c type:ARRAY bas
e:56 etype:INT dims:1 dim[0]:10))
```

```
h@table:(outer:@table width:20 argc:2 arglist:(f y) rtype:INT level:1 cod
e:... entry(name:f type:FUNPTT offset:8) entry:(name:y type:INT offset:12)
entry:(name:g type:FUNC offset:20 mytab:g@table))
```

```
g@table:(outer:h@table width:4 argc:1 arglist:(c) rtype:INT level:2 cod
e:...entry:(name:c type:ARRPTT offset:4))
```





## 10.2 简单算术表达式的翻译

- ▶ 不含数组引用和函数调用的算术表达式
  - $E \rightarrow i \mid d \mid E \text{ op } E \mid -E \mid (E)$
- ▶ 产生代码实现表达式的计算过程，这些代码在编译时生成，在运行时执行。当执行时得到表达式的值。
- ▶ 设计属性名
  - **code**用于存放由表达式翻译产生的中间代码，
  - **place**中存放一个变量，这个变量存放代码的执行结果
- ▶ 例：若  $E.\text{code}=[t1=24;t2=t1*36]$  那么  $E.\text{place}='t2'$



- $E \rightarrow i \mid d \mid E \text{ op } E$
- $E \rightarrow i$  {  
t=newvar();  
E.place=t;  
E.code=gen[?t = ?getv(i)]}
- $E \rightarrow d$  {  
x=getn(d);  
if(lookup(x)==UNBOUND)error();  
E.place=x;  
E.code=[]}
- $E \rightarrow E \text{ op } E$  {  
E[0].place=newvar();  
E[0].code=E[1].code++E[2].code++  
gen[?E[0].place = ?E[1].place ?getn(op) ?E[2].place]}



➤  $E \rightarrow (E) \mid -E$

➤  $E \rightarrow (E)$  {  
   $E[0].place = E[1].place;$   
   $E[0].code = E[1].code$  }

➤  $E \rightarrow -E$  {  
   $E[0].place = newvar();$   
   $E[0].code = E[1].code ++$   
   $gen[?E[0].place = - ?E[1].place]$  }



# 例：句子的带注释的规范归约

➤ 算术表达式  $a*(x-2)+y$  的LR制导的翻译

$a*(x-2)+y \Rightarrow E*(x-2)+y$	$E.place=a; E.code=[]$
$\Rightarrow E*(E-2)+y$	$E[2].place=x; E[2].code=[]$
$\Rightarrow E*(E-E)+y$	$E[3].place=t1; E[3].code=[t1=2]$
$\Rightarrow E*(E)+y$	$E[2].place=t2; E[2].code=[t1=2;t2=x-t1]$
$\Rightarrow E*E+y$	$E[2].place=t2; E[2].code=[t1=2;t2=x-t1]$
$\Rightarrow E+y$	$E.place=t3; E.code=[t1=2;t2=x-t1;t3=a*t2]$
$\Rightarrow E+E$	$E[2].place=y; E[2].code=[]$
$\Rightarrow E$	$E.place=t4; E.code=[t1=2;t2=x-t1;t3=a*t2;t4=t3+y]$

注：句子  $a*(x-2)+y$  带注释的规范归约如上所示。每一行的注释部分为这步归约时进行求值的各属性及求值结果。其中变元的索引值表示对应句型中该变元的出现，从1开始算起。如第三行句型  $E*(E-E)+y$  的标注中使用  $E[3]$  指第三个  $E$ 。对第一、第二个  $E$  没有标注是因为分别在第一、第二行标注过了，期间没有变化。其余类推。在标注时  $E.place$  属性标注为变量名是最终结果，没有带 ‘。



#	#	#
#a	#-	#-
#E	#a	#[]
#E*	#a-	#[]-
#E*(	#a--	#[]--
#E*(x	#a--	#[]---
#E*(E	#a--x	#[]--[]
#E*(E-	#a--x-	#[]--[]-
#E*(E-2	#a--x--	#[]--[]--
#E*(E-E	#a--x-t <sub>1</sub>	#[]--[]-[t <sub>1</sub> =2]
#E*(E	#a--t <sub>2</sub>	#[]--[t <sub>1</sub> =2; t <sub>2</sub> =x-t <sub>1</sub> ]
#E*(E)	#a--t <sub>2</sub> -	#[]--[t <sub>1</sub> =2; t <sub>2</sub> =x-t <sub>1</sub> ]-
#E*E	#a-t <sub>2</sub>	#[]-[t <sub>1</sub> =2; t <sub>2</sub> =x-t <sub>1</sub> ]
#E	#t <sub>3</sub>	#[t <sub>1</sub> =2; t <sub>2</sub> =x-t <sub>1</sub> ; t <sub>3</sub> =a*t <sub>2</sub> ]
#E+	#t <sub>3</sub> -	#[t <sub>1</sub> =2; t <sub>2</sub> =x-t <sub>1</sub> ; t <sub>3</sub> =a*t <sub>2</sub> ]-
#E+y	#t <sub>3</sub> --	#[t <sub>1</sub> =2; t <sub>2</sub> =x-t <sub>1</sub> ; t <sub>3</sub> =a*t <sub>2</sub> ]--
#E+E	#t <sub>3</sub> -y	#[t <sub>1</sub> =2; t <sub>2</sub> =x-t <sub>1</sub> ; t <sub>3</sub> =a*t <sub>2</sub> ]-[]
#E	#t <sub>4</sub>	#[t <sub>1</sub> =2; t <sub>2</sub> =x-t <sub>1</sub> ; t <sub>3</sub> =a*t <sub>2</sub> ; t <sub>4</sub> =t <sub>3</sub> +y]

**E→i** {t=newvar();E.place=t;  
E.code=gen[?t = ?getv(i)]}

**E→d** {x=getn(d);  
if(lookup(x)==UNBOUND)error();  
E.place=x; E.code=[]}

**E→E op E** {E[0].place=newvar();  
E[0].code=E[1].code++E[2].code++  
gen[?E[0].place = ?E[1].place  
?getn(op) ?E[2].place]}

**E→(E)** {E[0].place=E[1].place;  
E[0].code=E[1].code }

a\*(x-2)+y# s

\*(x-2)+y# r

\*(x-2)+y# s

(x-2)+y# s

x-2)+y# s

-2)+y# r

-2)+y# s

2)+y# s

) +y# r

) +y# r

) +y# s

+y# r

+y# r

+y# s

y# s

# r

# r

# acc

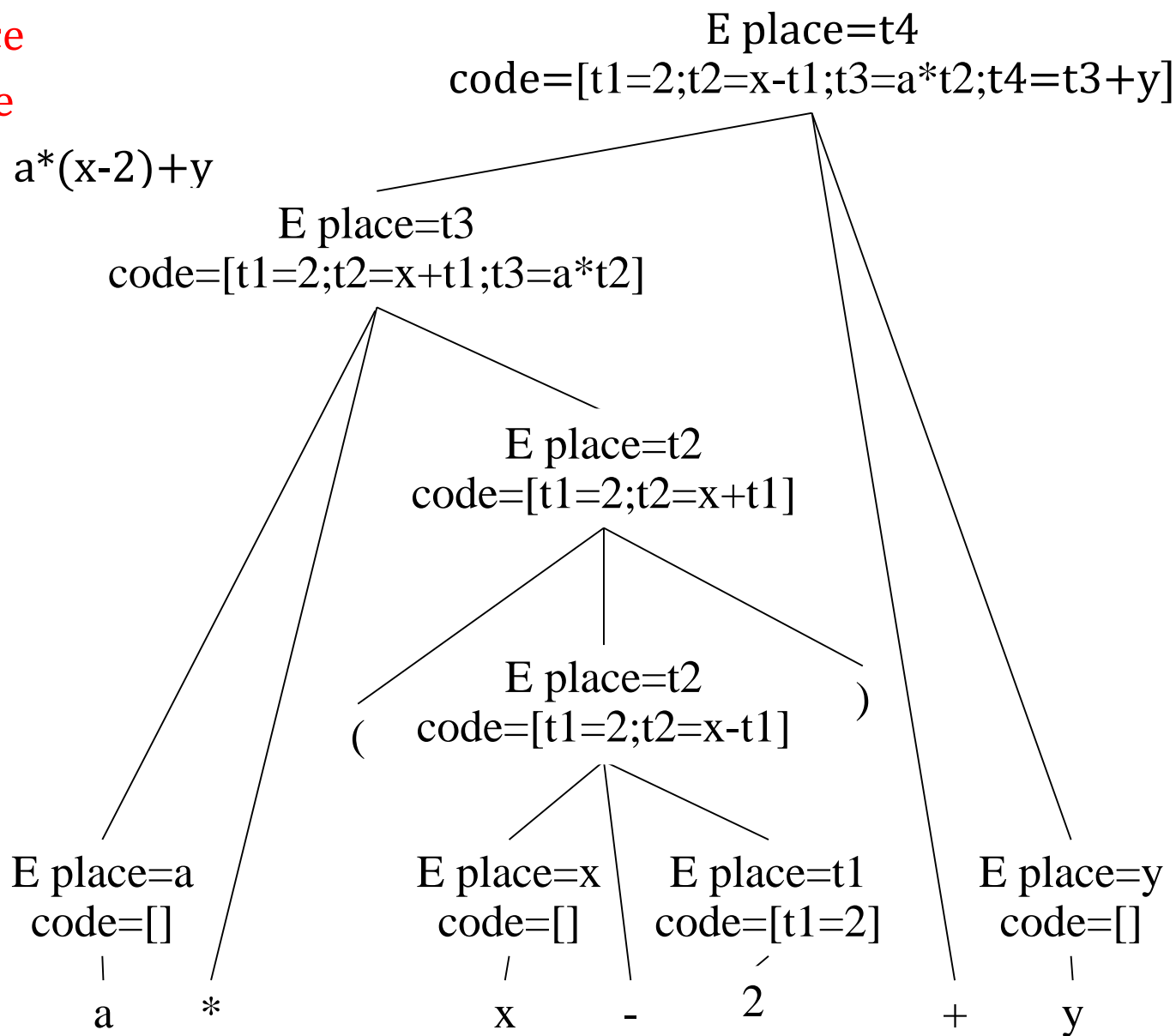


# 带注释语法树

➤ 属性名: **place**

➤ 属性名: **code**

➤ 算术表达式:  $a*(x-2)+y$



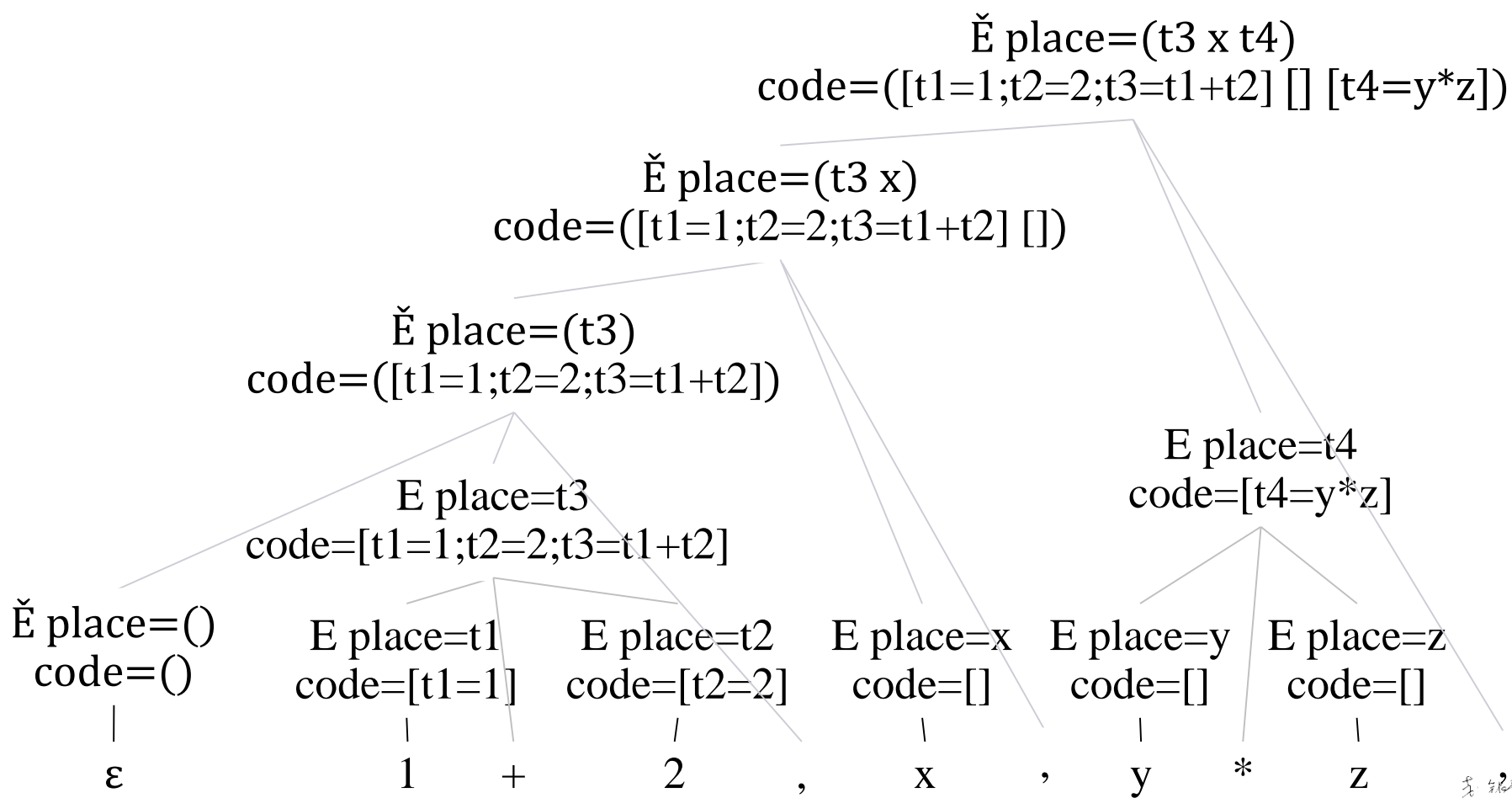




# 表达式列表

$\check{E} \rightarrow \varepsilon \quad \{\check{E}.place = NIL; \check{E}.code = NIL\}$

$\check{E} \rightarrow \check{E} E, \{ \quad \check{E}[0].place = endcons(\check{E}[1].place, E.place);$   
 $\check{E}[0].code = endcons(\check{E}[1].code, E.code) \quad \}$





# 常量折叠与整数列表

- ▶ 如果一个表达式里元素对象全部是整数，那么编译时计算出表达式的值并替代它。
- ▶  $I \rightarrow i \mid I \text{ op } I$
- ▶  $E \rightarrow d \mid E \text{ op } E \mid E \text{ op } I \mid I \text{ op } E \mid (E)$
- ▶ 类似地考虑实数、以及推广到常量
- ▶ 整数列表（出现在数组声明语句中）
- ▶  $\check{I} \rightarrow i \{ \check{I}.val = \text{list}(\text{getv}(i)) \}$
- ▶  $\check{I} \rightarrow \check{I}, i \{ \check{I}[0].val = \text{endcons}(\check{I}[1].val, \text{getv}(i)) \}$



- $S \rightarrow d = E$  {  
   $S.code = E.code + +gen[?getn(d) = ?E.place]$ }
- 类型转换
- 设置type属性，d和E类型不一致时先转换后赋值
- $S \rightarrow d = E$  {  
   $x = getn(d);$   
   $type = lookup(x, type:);$   
  if( $type == E.type$ )  $S.code = E.code + +gen[?x = ?E.place];$   
  else if( $type == INT$ )  
     $S.code = E.code + +gen[?x ftoi ?E.place];$   
  else if( $type == FLO$ )  
     $S.code = E.code + +gen[?x itof ?E.place];$   
  else error() }



- $E \rightarrow i$       {t=newvar(); E.place=t;  
emit(t '=' getv(i));}
- $E \rightarrow d$       {if(lookup(x)==UNBOUND)error();  
E.place=getn(d); }
- $E \rightarrow E \text{ op } E$     {E[0].place=newvar();  
emit(E[0].place '=' E[1].place getn(op) E[2].place);}
- $E \rightarrow (E)$       {  
E[0].place=E[1].place;}
- $E \rightarrow -E$       {E[0].place=newvar();  
emit(E[0].place '=' '-' E[1].place)];}

使用了函数**emit()**，它的功能是根据实参生成一条三地址指令并添加到代码区尾部，该函数对实参个数不限，实参之间用空格分隔，要对实参求值。



## 10.3 布尔表达式

- ▶  $B \rightarrow E \mid E \text{ r } E \mid B \wedge B \mid B \vee B \mid !B \mid (B)$
- ▶ 关系表达式是布尔表达式。
- ▶ 如果允许，算术表达式是布尔表达式
- ▶ 如果有布尔类型的话，布尔类型的常数、变量都是布尔表达式。
- ▶ 如果没有显式提供布尔类型，也可以像C语言那样将算术表达式作为布尔表达式。
- ▶ 布尔表达式通过布尔运算连接构成布尔表达式。
- ▶ 布尔类型常数只有真和假两个；
- ▶ 算术表达式被解释为布尔真（值为非0）和布尔假（值为0）
- ▶ 对布尔表达式求值采用短路算法。



➤ 关于运算符 $r$ 及优先级

➤ C语言:

- $\&\& \ || \ ! \ < \ == \ > \ <= \ >= \ !=$
- $\text{if}(1 < i \ \&\& \ i < 10) \dots$

➤ Pascal:

- ⊕  $\text{AND OR NOT } < \ = \ > \ <= \ >=$
- ⊕  $\text{if } (x = 0) \text{ AND } (a = 2) \text{ then} \dots$

➤ Fortran逻辑表达式:

- ⊕  $\text{.AND. .OR. .NOT. .LT. .EQ. .GT. .LE. .GE. .NE.}$
- ⊕  $b = a \text{ .AND. } 3 \text{ .LT. } 5/2$



# 布尔表达式的求值

▶ 短路算法:

- $A \vee B$  *if A then true else B*
- $A \wedge B$  *if A then B else false*
- $\neg A$  *if A then false else true*

```
if ( (p != NULL) && (p->val == 0) ) ...
```

- ▶ 作为条件控制用;
- ▶ 作为赋值语句的表达式用。



# 作为条件的布尔表达式

- ▶ 布尔表达式出现在分支、循环等控制语句中作为条件，
- ▶ 为此设计两类属性，名为`tc`和`fc`。
- ▶ `tc`和`fc`属性实例仅取值为标号，可以是多个标号，表示相应布尔表达式代码中为真转移去向，和为假时控制流流向。
- ▶ 对布尔表达式进行翻译将产生代码，用`code`属性来承载。





►  $B \rightarrow E$

```
{l1=newlabel();  l2=newlabel();  
B.tc=list(l1);  
B.fc=list(l2);  
B.code=E.code++gen[IF ?E.place != 0 THEN ?l1 ELSE ?l2]}
```

►  $B \rightarrow E \text{ r } E$

```
{l1=newlabel();  l2=newlabel();  
B.tc=list(l1);  
B.fc=list(l2);  
B.code=E[1].code++E[2].code++  
gen[IF ?E[1].place ?getn(r) ?E[2].place THEN ?l1  
      ELSE ?l2];}
```



- $B \rightarrow B \wedge B$  {  
   $B[0].tc = B[2].tc$ ;  
   $B[0].fc = \text{append}(B[1].fc, B[2].fc)$ ;  
   $B[0].code = B[1].code ++$   
     $\text{gen\_l}[LABEL ? B[1].tc] ++ B[2].code \}$
- $B \rightarrow B \vee B$  {  
   $B[0].tc = \text{append}(B[1].tc, B[2].tc)$ ;  
   $B[0].fc = B[2].fc$ ;  
   $B[0].code = B[1].code ++$   
     $\text{gen\_l}[LABEL ? B[1].fc] ++ B[2].code \}$
- $\text{append}(\text{list1}, \text{list2})$  将表list1和表list2合并为一个表，list1的表元素在前，list2的表元素在后，返回合并后的表。
- $\text{gen\_l}[LABEL ? llist]$  对llist中每个标号l都产生[LABEL ?l]



- $B \rightarrow ! B$  {  
     $B[0].tc = B[1].fc$ ;  
     $B[0].fc = B[1].tc$ ;  
     $B[0].code = B[1].code$ }
  
- $B \rightarrow ( B )$  {  
     $B[0].tc = B[1].tc$ ;  
     $B[0].fc = B[1].fc$ ;  
     $B[0].code = B[1].code$ }



# 举例

$(123 \vee 1 < x) \wedge y \Rightarrow (E \vee 1 < x) \wedge y$  E.place=t1;E.code=[t1=123]  
 $\Rightarrow (B \vee 1 < x) \wedge y$  B.tc=(l1);B.fc=(l2);B.code=[t1=123;if t1!=0 then l1 else l2] ①  
 $\Rightarrow (B \vee E < E) \wedge y$  E[1].place=t2;E[1].code=[t2=1]; E[2].place=x;E[2].code=[]  
 $\Rightarrow (B \vee B) \wedge y$  B[2].tc=(l3);B[2].fc=(l4);B[2].code=[t2=1;if t2<x then l3 else l4] ②  
 $\Rightarrow (B) \wedge y$  B.tc=(l1,l3);B.fc=(l4);B.code=[①;label l2;②]  
 $\Rightarrow * B \wedge E$  E.place=y;E.code=[];  
 $\Rightarrow B \wedge B$  B[2].tc=(l5);B[2].fc=(l6);B[2].code=[if y!=0 then l5 else l6] ③  
 $\Rightarrow B$  B.tc=(l5);B.fc=(l4 l6);B.code=[①;label l2;②;label l1; label l3; ③]

注：句子 $(123 \vee 1 < x) \wedge y$ 带注释的规范归约如上所示。每一行的注释部分为这步归约时进行求值的各属性及求值结果。其中变元的索引值表示对应句型中该变元的出现，从1开始算起。如第四行句型 $(B \vee B) \wedge y$ 的标注中使用B[2]指第二个B。对第一个B没有标注是因为在第二行标注过了，期间没有变化。其余类推。代码段后面带的序号，如第二行B.code的代码后附①，仅用于引用，如第5和8行引用该代码段来形成新的代码段。



# 例：对布尔表达式 $(123 \vee 1 < x) \wedge y$ 语义分析结果

李银亮

- B.code=[  
t1=123;  
IF t1!=0 THEN l1 ELSE l2;  
LABEL l2;  
t2=1;  
IF t2<x THEN l3 ELSE l4;  
LABEL l1;  
LABEL l3;  
IF y!=0 THEN l5 ELSE l6]
- B.tc=(l5)
- B.fc=(l4 l6)



# 增量式代码生成

- $B \rightarrow E$        $\{B.tc=newlabel(); B.fc=newlabel();$   
                   $emit[IF ?E.place \neq 0 THEN ?B.tc ELSE ?B.fc];\}$
- $\dot{B} \rightarrow B \wedge$        $\{emit[LABEL ?B.tc]; \dot{B}.fc=B.fc;\}$
- $\dot{B} \rightarrow B \vee$        $\{emit[LABEL ?B.fc]; \dot{B}.tc=B.tc;\}$
- $B \rightarrow \dot{B} B$        $\{B[0].tc=B[1].tc; B[0].fc=append(\dot{B}.fc, B[1].fc); \}$
- $E \rightarrow \dot{B} B$        $\{B[0].fc=B[1].fc; B[0].tc=append(\dot{B}.tc, B[1].tc); \}$
- $B \rightarrow E r E$      $\{B.tc=newlabel(); B.fc=newlabel();$   
                   $emit[IF ?E[1].place ?getn(r) ?E[2].place THEN ?B.tc ELSE$   
                   $?B.fc];\}$
- $B \rightarrow (B)$        $\{B[0].tc=B[1].tc; B[0].fc=B[1].fc;\}$
- $B \rightarrow \neg B$        $\{B[0].tc=B[1].fc; B[0].fc=B[1].tc;\}$



使用了宏**emit[]**，它的功能是根据实参生成一条三地址指令并添加到代码区尾部，该函数对实参个数不限，实参之间用空格分隔，不要对实参求值，除非参数带前边有？号。

对照函数**emit()**，它的功能是根据实参生成一条三地址指令并添加到代码区尾部，该函数对实参个数不限，实参之间用空格分隔，要对实参求值。



## 10.4 语句翻译

- ▶ 典型控制语句有顺序（复合语句）、分支和循环。
- ▶ 设计属性**code**承载语句翻译后产生的中间语言代码。
- ▶ 主要问题是对语句后继的访问如何进行处理。
- ▶ 采用综合属性替代继承属性完成语法树上的继承信息的处理。具体生成标号作为本语句在中间代码里所要访问的后继。这个标号将来会作为后继代码首地址（由祖先完成部署）。
- ▶ 概念：**结构化代码块**。只有一个入口（在最前）、只有一个出口（在最后）
- ▶ 由语句关心自己的后继，交给语境（上下文）去关心。
- ▶  $S \rightarrow \text{if } (B) S \text{ else } S$
- ▶  $S \rightarrow \text{while } (B) S$
- ▶  $S \rightarrow \{\check{S}\}$





- ▶  $S \rightarrow d = E$  { //前面已介绍过  
x=getn(d);  
if(lookup(x)==UNBOUND)error();  
S.code=E.code++gen[?x = ?E.place]}
- ▶  $S \rightarrow d[\check{E}] = E$  {?} //见数组声明与下标变量处理节



# 分支语句翻译

➤  $S \rightarrow \text{if } (B) S \{$   
     $S[0].\text{code} = B.\text{code}++$   
         $\text{gen\_l}[\text{LABEL ?}B.\text{tc}]++$   
     $S[1].\text{code}++$   
         $\text{gen\_l}[\text{LABEL ?}B.\text{fc}]\}$

$\text{if}(x < y) x = y$

$\Rightarrow \text{if}(B) x = y \quad B.\text{tc} = (I1) \quad B.\text{fc} = (I2) \quad B.\text{code}$   
 $= [\text{IF } x < y \text{ THEN } I1 \text{ ELSE } I2]$

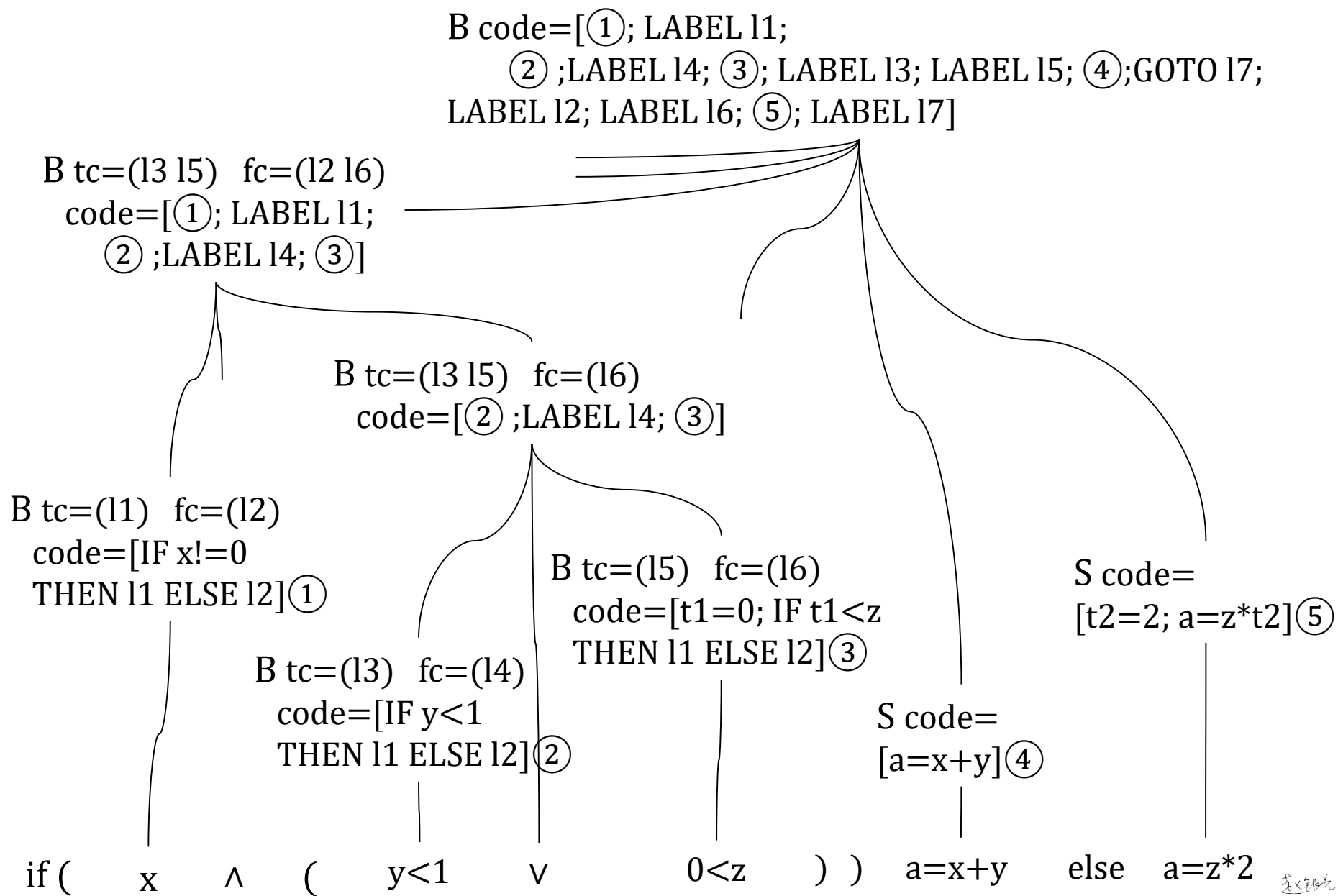
$\Rightarrow \text{if}(B) S \quad S.\text{code} = [x = y]$

$\Rightarrow S \quad S.\text{code} = [\text{IF } x < y \text{ THEN } I1 \text{ ELSE } I2;$   
 $\text{LABEL } I1; x = y; \text{LABEL } I2]$

➤  $S \rightarrow \text{if } (B) S \text{ else } S \{$   
     $l = \text{newlabel}();$   
     $S[0].\text{code} = B.\text{code}++$   
         $\text{gen\_l}[\text{LABEL ?}B.\text{tc}]++$   
     $S[1].\text{code}++$   
         $\text{gen}[\text{GOTO ?}l]++ \text{gen\_l}[\text{LABEL ?}B.\text{fc}]++$   
     $S[2].\text{code}++$   
         $\text{gen}[\text{LABEL ?}l]\}$



# 举例：布尔表达式作为if语句的条件





►  $S \rightarrow \text{while } (B) S \{$

$l = \text{newlabel}();$

$S[0].\text{code} =$

$\text{gen}[\text{LABEL } ?l]++$

$B.\text{code}++$

$\text{gen}_l[\text{LABEL } ?B.\text{tc}]++$

$S[1].\text{code}++$

$\text{gen}[\text{GOTO } ?l]++$

$\text{gen}_l[\text{LABEL } ?B.\text{fc}] \}$

//单出口



- $\text{if } (x < y) \text{ while } (z) \{x=1; y=b\}$
- $x < y \Rightarrow B$   
 $B.tc=l1; B.fc=l2; B.code=[\text{IF } x < y \text{ THEN } l1 \text{ ELSE } l2]$
- $x=1 \Rightarrow S$        $S.code=[t1=1; x=t1]$
- $y=b \Rightarrow S$        $S.code=[y=b]$
- $\{S; S\} \Rightarrow S$        $S.code=[t1=1; x=t1; y=b] \textcircled{2}$
- $z \Rightarrow B$
- $B.tc=l3; B.fc=l4; B.code=[\text{IF } z \neq 0 \text{ THEN } l3 \text{ ELSE } l4] \textcircled{1}$
- $\text{while } (B) S \Rightarrow S$
- $S.code=[\text{LABEL } l5; \textcircled{1}; \text{LABEL } l3; \textcircled{2}; \text{GOTO } l5; \text{LABEL } l4] \textcircled{3}$
- $\text{if}(B) S \Rightarrow S$
- $S.code=[\text{IF } x < y \text{ THEN } l1 \text{ ELSE } l2; \text{ LABEL } l1; \textcircled{3}; \text{LABEL } l2]$



- ▶  $\text{if } (x < y) \text{ while } (z) \{x=1; y=b\} \Rightarrow^* S$
- ▶ IF  $x < y$  THEN l1 ELSE l2; LABEL l1; ③; LABEL l2, 其中③为, LABEL l5; ①; LABEL l3; ②; GOTO l5; LABEL l4, 所以,
- ▶ S.code=[  
IF  $x < y$  THEN l1 ELSE l2;  
LABEL l1;  
LABEL l5;  
IF  $z \neq 0$  THEN l3 ELSE l4;  
LABEL l3;  
t1=1;  
x=t1;  
y=b;  
GOTO l5;  
LABEL l4;  
LABEL l2]



- ▶  $S \rightarrow \{\check{S}\} \{$   
     $S.code = merge\_code(\check{S}.code); \}$
- ▶  $\check{S} \rightarrow S \{ \check{S}.code = list(S.code); \}$
- ▶  $\check{S} \rightarrow \check{S}; S \{ \check{S}.code = endcons(S.code, \check{S}[1].code); \}$



# 增量式代码生成

- ▶  $C \rightarrow \text{if}(B) \{ \text{emit}[\text{LABEL } ?B.tc]; C.next := E.fc; \}$
- ▶  $T^p \rightarrow C \text{ S else } \{ T^p.next = \text{newlabel}();$   
 $\text{emit}[\text{GOTO } ? T^p.next]; \text{emit}[\text{LABEL } C.next]; \}$
- ▶  $S \rightarrow T^p S \{ \text{emit}[\text{LABEL } ?T^p.next]; \}$
- ▶  $W \rightarrow \text{while} \{ W.addr = \text{newlabel}(); \text{emit}[\text{LABEL } W.addr]; \}$
- ▶  $W^d \rightarrow W(B) \{ \text{emit}[\text{LABEL } ?B.next]; W^d.addr = W.addr;$   
 $W^d.next = B.fc; \}$
- ▶  $S \rightarrow W^d S \{ \text{emit}[\text{GOTO } ?W^d.addr]; \text{emit}[\text{LABEL } ?W^d.next]; \}$

注：emit[LABEL ?ll]的参数ll为一个表，该表的每个元素是一个标号，那么对于ll中每个标号，依次连续生成LABEL指令。比如ll = '(l1, l2, l3)，那么生成3条LABEL指令，即为[LABEL l1; LABEL l2; LABEL l3]。





- $S \rightarrow \text{goto } d$
- $S \rightarrow d:S$
- 标号先定义后引用
- $l1:\text{if}(x<1)\dots$
- $\dots$
- $\text{goto } l1$
- 标号先引用后定义
- $\text{goto } l2$
- $\dots$
- $\text{goto } l2$
- $\dots$
- $l2:\dots$

- $S \rightarrow d:S$ 
  - d标号不存在：  
建立登记项并生成标号l1置于place域；  
生成[LABEL l1; ?S[1].code]
  - d标号存在；
    - 定义否？未定义：  
值定义否为已定义；取place域值l2  
生成[LABEL l2; ?S[1].code]
    - 定义否？已定义：标号重定义错
- $S \rightarrow \text{goto } d$ 
  - d标号不存在：  
建立登记项并生成标号l1置于place域；  
生成[GOTO l1]
  - d标号存在（定义否？未/已定义）：  
取place域值l2生成[GOTO l2]



```
➤ S → goto d {  
  l1 = newlabel();  
  x = getn(d);  
  if (lookup(x) == UNBOUND) { // 标号的引用先于定义  
    bind(x, LAB);  
    lookup(x, place, l1);  
    lookup(x, def, NO);  
    gen[GOTO ?l1]}  
  else { // 标号已经不是第一次被引用  
    l2 = lookup(x, place); // 标号肯定存在  
    S.code = gen[GOTO ?l2]}}
```



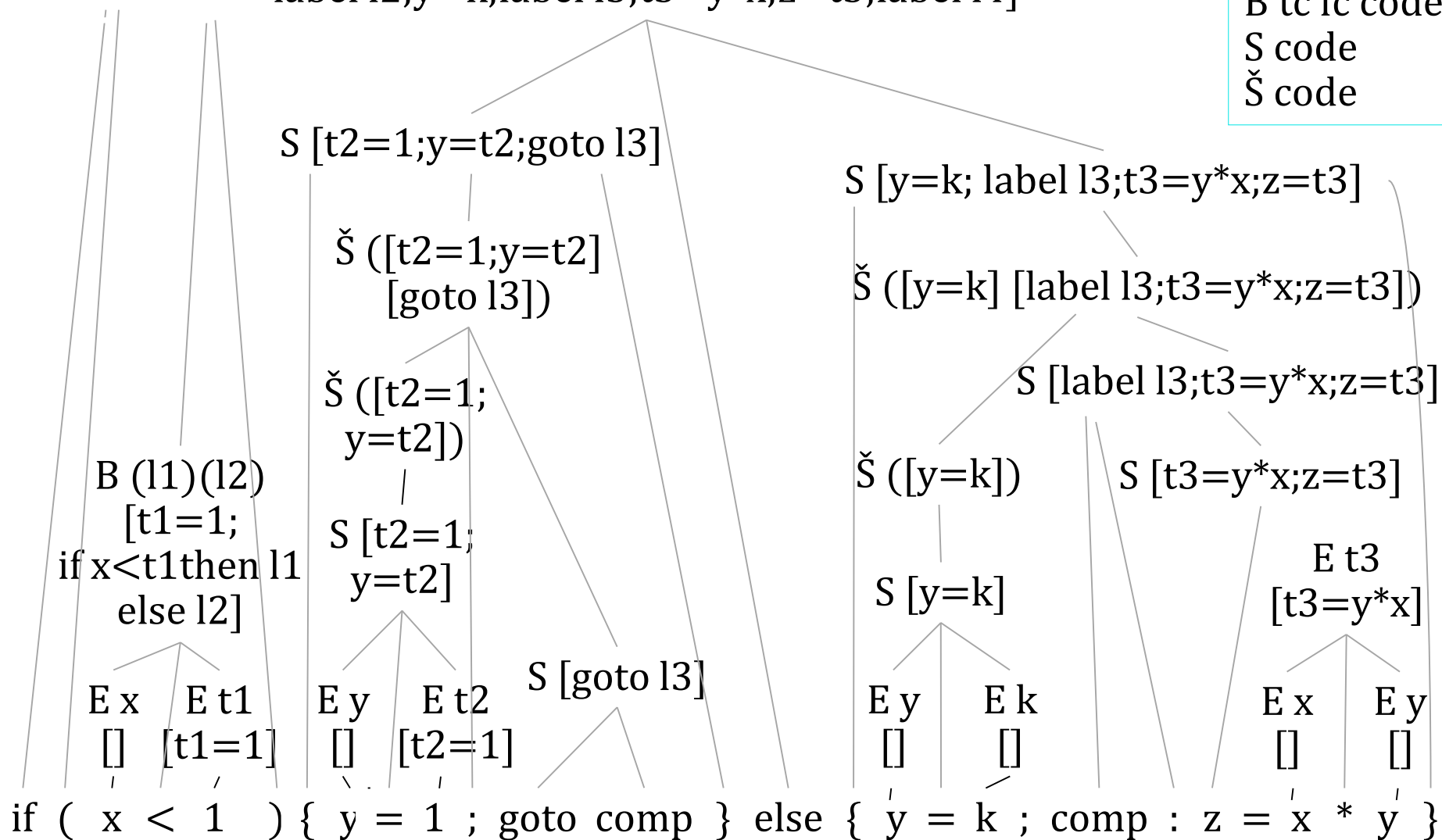
```
➤ S→d:S      {  
  l1=newlabel();  
  x=getn(d);  
  if(lookup(x)==UNBOUND){//标号定义在先  
    bind(x,LAB);  
    lookup(x,place,l1);  
    lookup(x,def,YES);  
    S[0].code=gen[LABEL ?l1]++S[1].code}  
  else if(lookup(x,type)==LAB&&lookup(x,def)!=YES){  
    l2=lookup(x,place);  
    lookup(x,def,YES);  
    S[0].code=gen[LABEL ?l2]++S[1].code}  
  else error()}
```



# 例：语句翻译

E place code  
B tc fc code  
S code  
Š code

S [t1=1;if x<t1 then l1 else l2;label l1;t2=1;y=t2;goto l3;goto l4;  
label l2;y=k;label l3;t3=y\*x;z=t3;label l4]



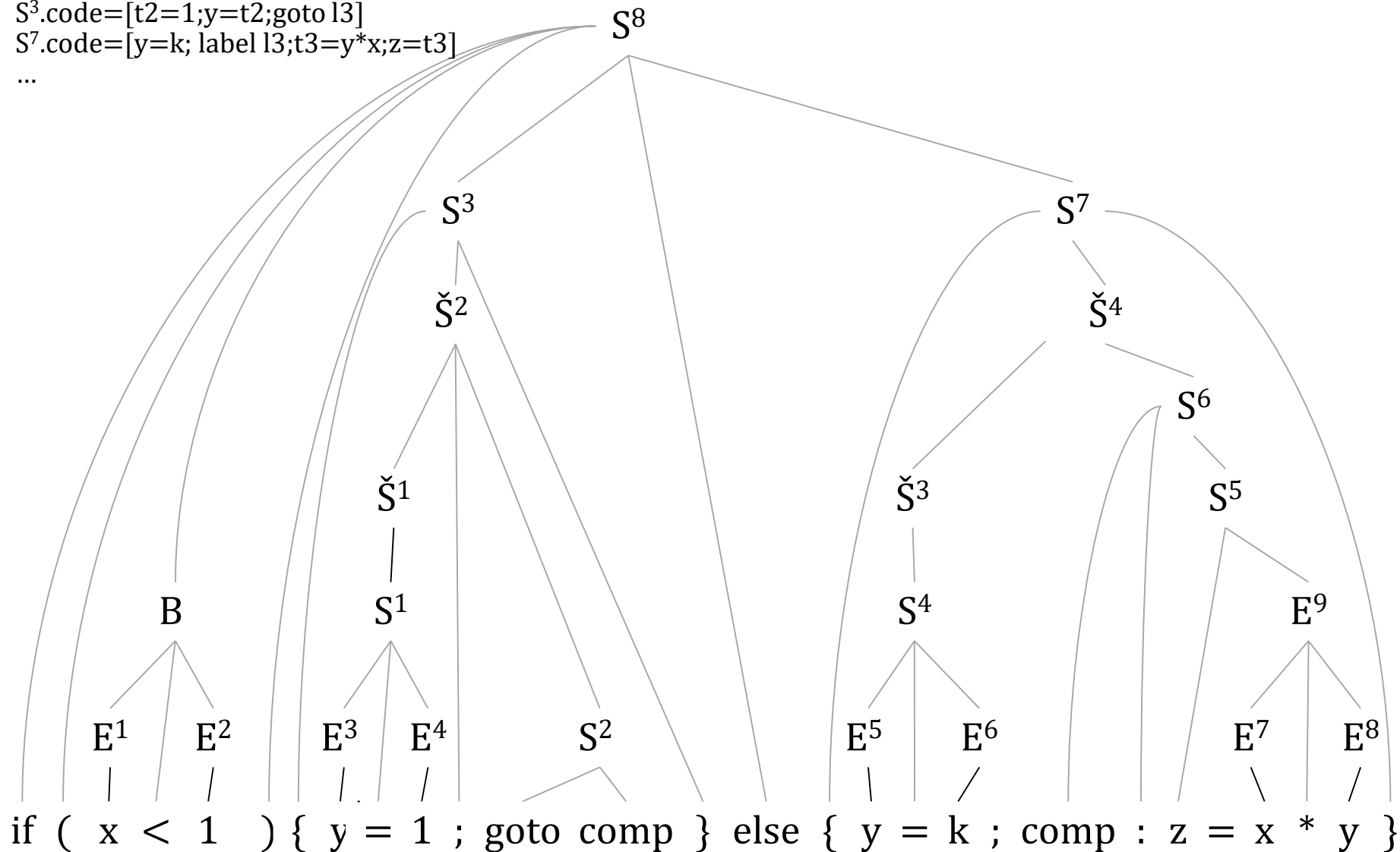
$S^8$  [t1=1;if x<t1 then l1 else l2;label l1;t2=1;y=t2;goto l3;goto l4;label l2;y=k;label l3;t3=y\*x;z=t3;label l4]

B.tc=(l1) fc=(l2) code=[t1=1;if x<t1 then l1 else l2]

$S^3$ .code=[t2=1;y=t2;goto l3]

$S^7$ .code=[y=k; label l3;t3=y\*x;z=t3]

...

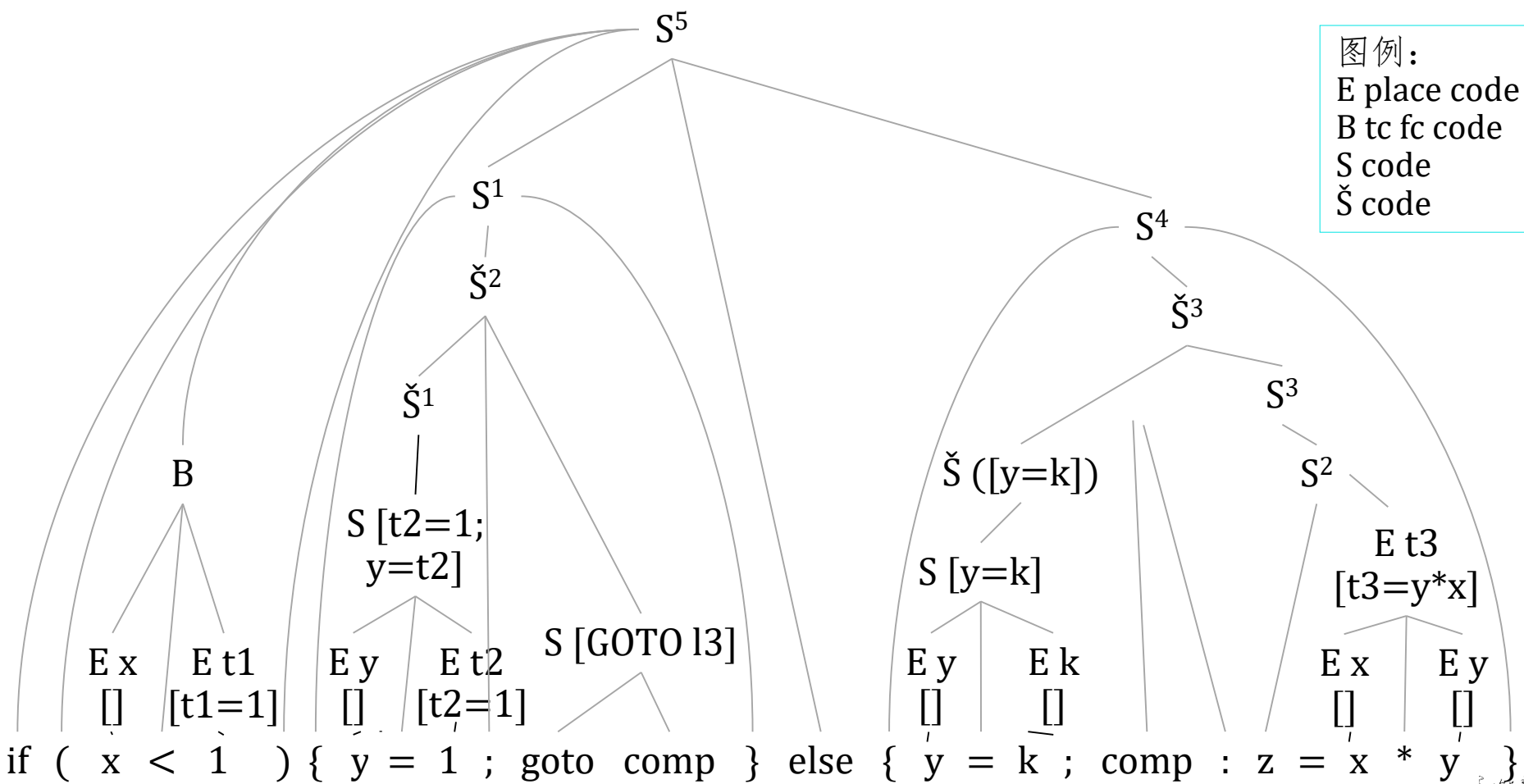




# 注释语法树

$S^5$  [t1=1;IF x<t1 THEN l1 ELSE l2;LABEL l1;t2=1;y=t2;GOTO l3;GOTO l4;LABEL l2;y=k;LABEL l3;t3=y\*x;z=t3;LABEL l4]    B.tc=(l1) fc=(l2) code=[t1=1;IF x<t1 THEN l1 ELSE l2]  
 $S^1$ .code=[t2=1;y=t2;GOTO l3]     $S^2$ .code=[t3=y\*x;z=t3]     $S^3$ .code=[LABEL l3;t3=y\*x;z=t3]  
 $S^4$ .code=[y=k; LABEL l3;t3=y\*x;z=t3]     $\check{S}^1$ .code= ([t2=1;y=t2])  
 $\check{S}^2$ .code=([t2=1;y=t2] [GOTO l3])     $\check{S}^3$ .code= ([y=k] [LABEL l3;t3=y\*x;z=t3])

图例：  
 E place code  
 B tc fc code  
 S code  
 $\check{S}$  code





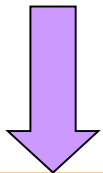
## 10.5 下标变量引用

- 例: `int a[10,5,20]`  $10 \times 5 \times 20 \times 4$
- `f@table: (width: 4000 ... entry: (name: a type: ARRAY base: 4000 etype: INT dims: 3 dim[0]: 10 dim[1]: 5 dim[2]: 20 )...)`
- 为左值引用  $S \rightarrow a[\check{E}] = E$  生成代码
- 为右值引用  $E \rightarrow a[\check{E}]$  生成代码



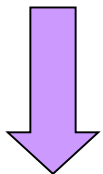
# 下标变量的语义

$a[i+1]$



$a + (i+1) * \text{sizeof}(\text{int})$

$a[t]$



$\text{base\_address}(a) + (t - \text{lower\_bound}(a)) * \text{element\_size}(a)$





# (1) 地址计算公式

- ▶ 对于一维数组  $a[i]$ :
  - 下标的变化范围:  $l \leq i < u$
  - 连续存储区的首址:  $\text{base}$ , 每个数组元素占用  $w$  个单元
  - 则  $a[i]$  地址为:  $\text{base} + (i - l) * w$
  
- ▶  $a[i]$  地址:  $\text{base} + (i - l) * w = (\text{base} - l * w) + i * w$
- ▶ 固定部分:  $\text{base} - l * w$       // 编译时确定
- ▶ 变化部分:  $i * w$       // 运行时确定
  
- ▶ 如果  $l$  恒为 0, 那么  $a[i]$  地址:  $\text{base} + i * w$



## (2) 数组声明 $a[l:u]$ 的信息

► 符号表登记项:

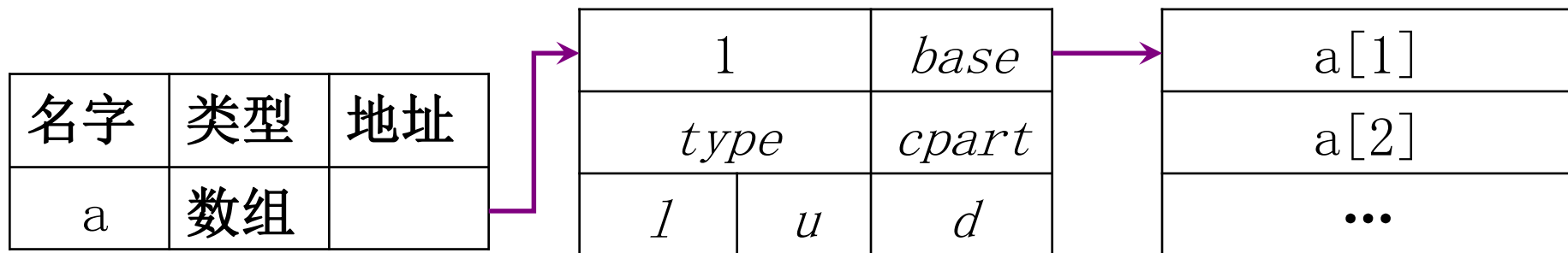
- name、type=ARRAY、etype、base、dims、dim[i]
- 允许下界不为0则需要更多类型特有域: cpart、lb[i]、ub[i]

---书的采用---

符号表项

内情向量表

元素存储区域



■ 对于一维数组元素引用形式 $a[i]$ , 该元素的存储位置为:

$$base + (i - l) * w = base - cpart + i * w$$



# 地址计算公式（二维）

► 对于二维数组  $a[i,j]$ :

- 下标的变化范围:  $l_1 \leq i < u_1; l_2 \leq j < u_2$
- 连续存储区的首址:  $base$ , 每个数组元素占用  $w$  个单元
- 则  $a[i,j]$  地址为:  $base + [(i-l_1)*(u_2-l_2) + j-l_2]*w$

$$base + [(i-l_1)*(u_2-l_2) + j-l_2]*w$$

$$= [base + (-l_1*(u_2-l_2) - l_2)*w] + [(i*(u_2-l_2) + j)*w]$$

固定部分:  $base$ ; 下界恒为0

变化部分:  $(i*(u_2-l_2) + j)*w$

用维长表示:  $(i*d_2 + j)*w$



# 地址计算公式（一般情形）

➤ 对于多维数组  $a[i_1, i_2, \dots, i_n]$ :

- 下标的变化范围:  $l_1 \leq i_1 < u_1; \dots; l_n \leq i_n < u_n$
- 连续存储区的首址:  $base$ , 每个数组元素占用单元  $w$  个

➤ 则  $a[i_1, i_2, \dots, i_n]$  地址为:

$$\begin{aligned} &\text{➤ } base + ((\dots((i_1 - l_1) * d_2 + (i_2 - l_2)) * d_3 + (i_3 - l_3) \dots) d_n + (i_n - l_n)) * w \\ &= base - ((\dots((l_1 * d_2 + l_2) * d_3 + l_3) \dots) d_n + l_n) * w + \\ &\quad i_1 * d_2 * \dots * d_n + i_2 * d_3 * \dots * d_n + \dots + i_n \\ &= base - cpart + \end{aligned}$$

$$(\dots((i_1) * d_2 + i_2) * d_3 + i_3) * d_4 + i_4 \dots) * d_n + i_n$$

➤ 如果下界恒为0, 则  $cpart$  为0, 只需为运行时产生代码对应于可变部分。这是本课程采用的情形。



- $E \rightarrow d[\check{E}]$
- $a[i+1, j*2, 4]$
- $\check{E}.code$ 是一个表，表元素同 $E.code$ ，顺序与列表一致。
  - $([t1=i+1] [t2=j*2] [t3=4])$
- $\check{E}.place$ 是一个表，表元素同 $E.place$ ，顺序与列表一致。
  - $(t1 \ t2 \ t3)$       //为各表达式的结果变量
- $E.code$ 用于存放由表达式翻译产生的中间代码。
  - 计算地址:  $(t1*d_2+t2)*d_3+t3$
  - $[t1=i+1; t2=j*2; t3=4; t100=t1*5; t101=t100+t2; t102=t101*20; t103=t102+t3; t104=t103*4; t105=a[t104]]$
- $E.place$ 中存放结果变量 $t105$ ，是上述计算下标变量地址并引用的代码的执行结果。



# 数组元素引用 (右值)

```

➤ E → d[Ĕ]    {
1  ilist=Ĕ.place;
2  t0=car(ilist); ilist=cdr(ilist);
3  code=[]; x=getn(d);
4  dims=lookup(x, dims:);
5  for(i=1;i<dims;i++){
6      t=newvar();
7      dim=lookup(x, dim[i]:);
8      code++=gen[?t=?t0 *?dim)];
9      t0=newvar();
10     code++=gen[?t0=?t+?car(ilist)];
11     ilist=cdr(ilist);
12 }
13 t=newvar();
14 code++=gen[?t=?t0*?sizeof(lookup(x,etype:))]
14 t0=newvar(); E.place=t0;
15 E.code=merge_code(Ĕ)+code++gen[?t0=?x [?t]]}
    
```

```

设Ĕ.place= '(t1 t2 t3)
t0='t1; code=[]; x='a; dims=3
t='t100; dim=5
code++=[t100=t1*5]
t0='t101
code++=[t101=t100+t2]
t='t102; dim=20
code++=[t102=t101*20]
t0='t103
code++=[t103=t102+t3]
t='t104
code++=[t104=t103*4]
t0='t105; E.place='t105
E.code=[t1...;t2...;t3...;t100...;t1
01...;t102...;t103...;t104...;
t105=a[t104]]
    
```



# 数组元素引用 (左值)

```
➤ S → d[Ě]=E      {  
1  ilist=Ě.place;  
2  t0=car(ilist); ilist=cdr(ilist);  
3  code=[]; x=getn(d);  
4  dims=lookup(x, dims:);  
5  for(i=1;i<dims;i++){  
6      t=newvar();  
7      dim=lookup(x, dim[i]:);  
8      code++=gen[?t=?t0 *?dim];  
9      t0=newvar();  
10     code++=gen[?t0=?t+?car(ilist)];  
11     ilist=cdr(ilist);  
12     }  
13 t=newvar();  
14 code++=gen[?t=?t0*?sizeof(lookup(x,etype:))]  
15 S.code=merge_code(Ě.code)++E.code++code++  
16     gen[?x [?t]=?E.place]}
```

```
Ě.place='(t5 t6 t7) E.place='t8  
t0='t5; code=[]; x='a; dims=3  
t='t100; dim=5  
code++=[t100=t5*5]  
t0='t101  
code++=[t101=t100+t6]  
t='t102; dim=20  
code++=[t102=t101*20]  
t0='t103  
code++=[t103=t102+t7]  
t='t104  
code++=[t104=t103*4]  
S.code=[t5...;t6...;t7...]++  
E.code++[t100...;t101...;t102...;  
t103...;t104...;a[t104]=t8]
```



- $S \rightarrow d[\check{E}] = E$
- $a[i+1, j*2, 4] = 66$
- $\check{E}.code$  是一个表，表元素同  $E.code$ ，顺序与列表一致。
  - $([t5=i+1] [t6=j*2] [t7=4])$
- $\check{E}.place$  是一个表，表元素同  $E.place$ ，顺序与列表一致。
  - $(t5 \ t6 \ t7)$
- $E.code$  用于存放由表达式翻译产生的中间代码。
- $E.place$  中存放一个变量  $t8$ ，这个变量存放代码的执行结果。
- $S.code$  语句  $S$  的代码
  - 计算地址：  $(t5*d_2+t6)*d_3+t7$
  - $[t5=i+1; t6=j+2; t7=4; t8=66; t100=t5*5; t101=t100+t6; t102=t101*20; t103=t102+t7; t104=t103*4; a[t104]=t8]$





- $a[i+1, j*2, 4,]=66$
- $\Rightarrow a[\check{E} i+1, j*2, 4,]=66 \quad \check{E}.place=NIL \quad \check{E}.code=NIL$
- $\Rightarrow a[\check{E} E, j*2, 4,]=66 \quad E.place='t5 \quad E.code=[t5=i+1]$
- $\Rightarrow a[\check{E} j*2, 4,]=66 \quad \check{E}.place='(t5) \quad \check{E}.code='([t5=i+1])$
- $\Rightarrow a[\check{E} E, 4,]=66 \quad E.place='t6 \quad E.code=[t6=j*2]$
- $\Rightarrow a[\check{E} 4,]=66 \quad \check{E}.place='(t5 \ t6) \quad \check{E}.code='([t5=i+1] \ [t6=j*2])$
- $\Rightarrow a[\check{E} E,]=66 \quad E.place='t7 \quad E.code=[t7=4]$
- $\Rightarrow a[\check{E}]=66 \quad \check{E}.place='(t5 \ t6 \ t7) \quad \check{E}.code='([t5=i+1] \ [t6=j*2] \ [t7=4])$
- $\Rightarrow a[\check{E}]=E \quad E.place='t8 \quad E.code=[t8=66]$
- $\Rightarrow S$   
 $S.code=[t5=i+1;t6=j*2;t7=4;t8=66;t100=t5*5;t101=t100+t6;t102=t101*20; t103=t102+t7;t104=t103*4; a[t104]=t8]$

例  $\text{int } a[5,20], b[10]; a[i,j+1]=k*a[b[i-1],k];$

S [t1=1;t2=j+t1; ③; t12=i\*20;t13=t12+t2;t14=t13\*4;a[t14]=t11]

E t11 [②;t11=k\*t10] ③

图例：S code  
E place code  
Ė place code

E t10  
[①;t7=t6\*20;t8=t7+k;t9=t8\*4; t10=a[t9]] ②

Ė (t6 k) (① [])

Ė (t6) (①)

E t6 [t3=1;t4=i-t3;t5=t4\*4;t6=b[t5]] ①

Ė (t4) ([t3=1;t4=i-t3])

E t4 [t3=1;t4=i-t3]

E i []

E t3 [t3=1]

E k []

E t2  
[t1=1;t2=j+t1]

Ė (i)  
([])

E i  
[]

E j  
[]

E t1  
[t1=1]

E k  
[]


a [ i , j + 1 ] = k \* a [ b [ i - 1 ] , k ]



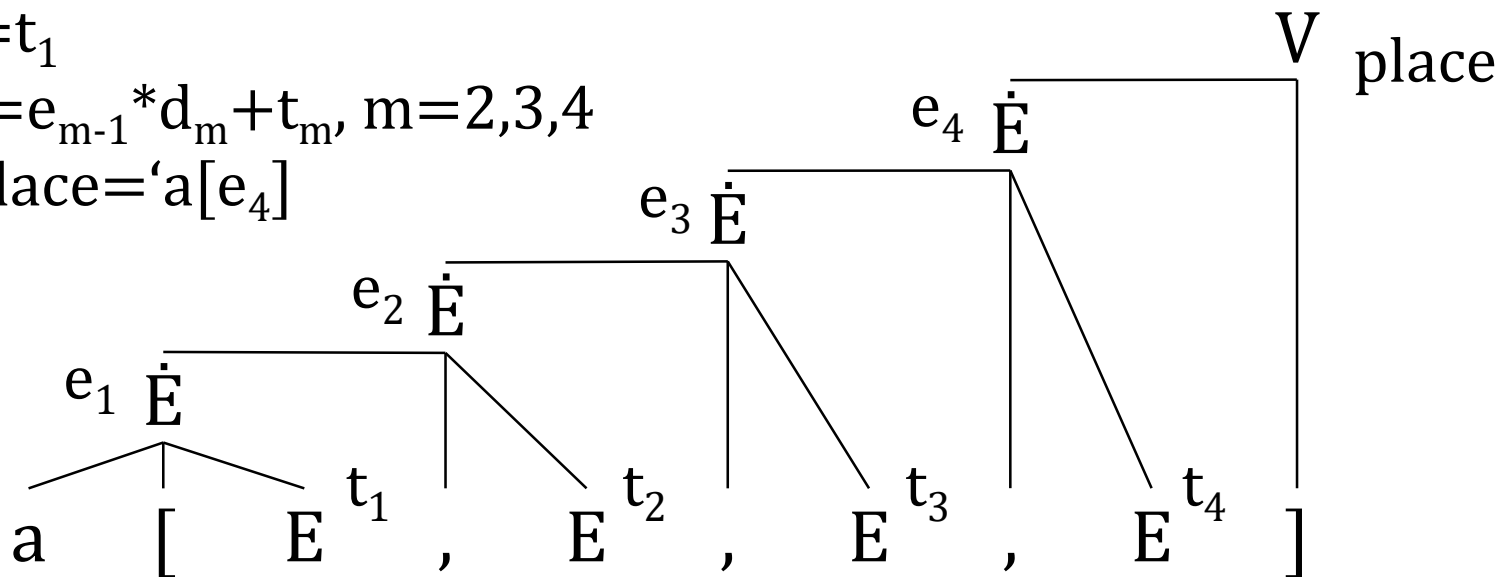
```
int a[5,20],b[10]; a[i,j+1]=k*a[b[i-1],k];
```

```
[t1=1  
t2=j+t1  
t3=1  
t4=i-t3  
t5=t4*4  
t6=b[t5]  
t7=t6*20  
t8=t7+k  
t9=t8*4  
t10=a[t9]  
t11=k*t10  
t12=i*20  
t13=t12+t2  
t14=t13*4  
a[t14]=t11]
```

## 对下标变量的处理

$$\begin{array}{l} V \rightarrow \dot{E}] \mid d \\ \dot{E} \rightarrow \dot{E}, E \mid d[E \end{array}$$
  $e_1 = t_1$ 
$$\mathbf{e}_m = \mathbf{e}_{m-1} * \mathbf{d}_m + \mathbf{t}_m$$
$$e_1 = t_1$$
$$e_m = e_{m-1} * d_m + t_m, m = 2, 3, 4$$

V.place='a[e<sub>4</sub>]





## 10.6 函数调用语句

- ▶ 函数有类型**FUNC**以及有返回值类型。
- ▶ 函数有代码，就是函数体被翻译得到的中间语言代码。
- ▶ 函数的参数可以是任意类型，参数可以多个。
  - 数组作为参数，声明为数组原型。
  - 函数作为参数，声明为函数原型。
- ▶  $D \rightarrow T d (\check{A}) \{\check{D} \check{S}\}$  其中 $\check{D}$ 中可以声明函数（类似PASCAL）或者不允许再声明函数（类似C语言）。
- ▶ 函数声明的有用信息都在语义分析时保存到符号表中。



# 访问符号表登记项

- **lookup()** 类似只是默认 **symtab** 栈顶符号表
- **lookup1(tab, name)** 查找 **name** 返回该登记项或 **UNBOUND**
- **lookup1(tab, name, nature:)** 返回 **name** 登记项的 **nature** 域值
- **lookup1(tab, name, nature:, value)** 置 **name** 登记项的 **nature** 域值为 **value**
- **lookup1(tab, name, nature[index]:, value)** 置 **name** 登记项的 **nature** 域的第 **index** 个元素的值为 **value**
- **bind()** 同 **bind1()** 只是默认 **symtab** 栈顶符号表
- **bind1(tab, name, type)** 将 **name** 加入 **tab** 作为一个登记项类型为 **type**, 若已存在则返回错误



```
➤ E → d (Ĕ) {  
  code=[];  
  parlist=Ĕ.place;  
  x=getn(d);  
  argc=length(parlist); //实参个数  
  while(parlist!=NIL){  
    code=gen[PAR ?car(parlist)]++code; //采用实参倒序  
    parlist=cdr(parlist);}  
  v=newvar();  
  code++=gen[?v=CALL ?x, ?argc];  
  E.place=v;//该变量存放返回结果  
  E.code=merge_code(Ĕ.code)++code;  
}
```



```
➤ S → d (Ė)  {  
  code=[];  
  parlist=Ė.place;  
  x=getn(d);  
  argc=length(parlist);  
  while(parlist!=NIL){  
    code=gen[PAR ?car(parlist)]++code;  
    parlist=cdr(parlist);}  
  v=newvar();  
  code++=gen[?v=CALL ?x, ?argc];  
  S.code=merge_code(Ė.code)++code;  
}
```





- $\check{R} \rightarrow \varepsilon \mid \check{R} R,$
- $R \rightarrow E \{R.place=E.place;\}$
- $R \rightarrow d[] \{R.place=getn(d);\}$
- $R \rightarrow d() \{R.place=getn(d);\}$
- $E \rightarrow d(\check{R})$
- $S \rightarrow d(\check{R})$

考虑到数组和函数作为实参，形式为 $d[]$ 和 $d()$ ，不能归约为 $E$ ，因为既不是数组元素引用也不是函数调用。所以在主文法中归约为 $R$ 。相应地采用这些文法来描述实参。

为了简单期间，对于实参 $d[]$ 和 $d()$ ，只要写为数组名和函数名，就达到同样效果。这种情况下也不需要变元 $R$ ，仍然用 $E$ 就够了。所以ppt上采用简化的实参形式。



# 函数返回语句

▶  $S \rightarrow \text{return } E$       {  
     $S.\text{code} = E.\text{code} ++ \text{gen}[\text{RETURN ?}E.\text{place}]$  }



# 采用全局名的翻译样例

```
int z;  
int a[10,20];  
int bar(int x){  
    return x++;  
}  
float foo(int x, int b[],  
          int boo()){  
    if (x > 0) z=sqrt(b[0,0])  
    else return boo(z);  
    print foo(0, a[],bar())
```

@table:

```
(outer:NULL width:820 argc:0 arglist:NIL rtype:INT  
code:[t5=0; PAR t5; PAR a; PAR bar;t6=CALL foo@label, 3]  
entry:(name:z type:INT offset:4)  
entry:(name:a type:ARRAY base:804 etype:INT dims:2 dim[0]:10  
dim[1]:20)  
entry:(name:bar type:FUNC offset:812 mytab:bar@table)  
entry:(name:foo type:FUNC offset:820 mytab:foo@table))
```

```
foo@table:(outer:@table width:16 argc:3  
arglist:(x b boo) rtype:FLO  
code:[IF x>0 THEN I1 ELSE I2;  
LABEL I1; t1=0; t2=b[t1];PAR t2; t3=CALL sqrt, 1;  
z=t3; GOTO I3;  
LABEL I2;PAR z; t4=CALL boo, 1;RETURN t4;LABEL I3]  
entry:(name:x type:INT offset:4)  
entry:(name:b type:ARRPTT offset:8 etype:INT)  
entry:(name:boo type:FUNPTT offset:16 rtype:INT))
```

```
bar@table:(outer:@table width:4  
argc:1 arglist:(x) rtype:INT  
code:[x=x+1; RETURN x]  
entry(name:x type:INT offset:4)
```



- ▶ 检查类型声明是否满足类型的语义性质。
- ▶ 设计时允许文法修剪与附加语义约束取得折中，
  - $D \rightarrow Td(\check{D})\{\check{D}\check{S}\}$  修剪为  $D \rightarrow Td(\check{A})\{\check{D}\check{S}\}$ ；或者给  $D \rightarrow Td(\check{D})\{\check{D}\check{S}\}$  附加语义约束，将  $\check{D}[1]$  约束为只能是  $\check{A}$  情况；
  - $D \rightarrow Td[\check{E}]$  修剪为  $D \rightarrow Td[\check{I}]$ ；或者给  $D \rightarrow Td[\check{E}]$  附加语义约束，将  $\check{E}$  约束为只能是整数列表否则报错；
  - 各种边界及类型范围的约束等等。
- ▶ 总之在简化文法与简化语义性质之间平衡。
- ▶ 当然有些语义性质是无法通过修剪文法来平衡掉的，因为这些语义性质是固有的。
- ▶ 此外还有类型推理问题如类型转换等（略）



- ▶ 几种中间表示，掌握三地址指令
- ▶ 各语法单元的翻译10.2-10.5
- ▶ 采用的属性名及各个属性含义
  - place; code; tc; fc
- ▶ 属性文法中用到的功能函数
  - newvar(); newlabel(); merge\_code(); lookup(); update[]
  - gen[]; emit[]; emit()
- ▶ 写属性文法与读属性文法
- ▶ 作业：（待给出）给定文法设计属性文法基于LR制导生成中间语言代码。
- ▶ 语义分析程序实现构想。



## 第九次作业

► 习题10.1 试参照ppt翻译下面的高级语言语句，写出翻译成的三地址代码段，并给出带注释的语法树或规范归约：

a) while ( $1 < x \&\& y > 1$ )

    if ( $-n + 1 < x * y$ )  $a = 1$

    else  $b = a$

b) if(x) while(y){

        if( $x > y$ ) goto next;

$x = x * y$ }

else {

$y = x * y$ ;

next: print y}



## ► 习题10.2

参照教材C的p218: 11, 得到C语言中的for语句的文法为

$$S \rightarrow \text{for}(S; B; S) S$$

其意义如下:

`S[1].code`

`while(B.code){`

`S[3].code`

`S[2].code`

`}`

试设计属性文法把C语言的for语句翻译为三地址代码, 并对语句

`for(i=0; i<100; i=i+1) print i`

给出翻译成的三地址代码段, 并给出带注释语法树或规范归约作为对翻译过程和结果的解释。



### ► 习题10.3

参照教材下的p218: 12完成如下各小题:

- a) 使用课程主文法符号（见附录A），写出PASCAL语言中的for语句的文法；
- b) 回答原题第(1)问；
- c) 试设计属性文法把PASCAL语言的for语句翻译为三地址代码；
- d) 对该题第(1)问中代码倒数第3和第2行，写出翻译后的三地址代码段，并给出带注释语法树或规范归约作为对翻译过程和结果的解释。





# 例：数组、函数作形参

## ► 更正习题10.4

对于如下程序：

```
int x; float z;
```

```
int a[10,20]; //初始化值为a[i,j]=i+j
```

```
float bar(int y,){
```

```
    float x;
```

```
    x=y*PI;
```

```
    return x};
```

```
float foo(int x, float boo(), int arr[10],){
```

```
    if (x==0)z=boo(arr[1])
```

```
    else return boo(arr[15*x])};
```

```
print foo(2, bar(), a[])
```

试写出对程序声明部分和语句部分进行语义分析的结果。