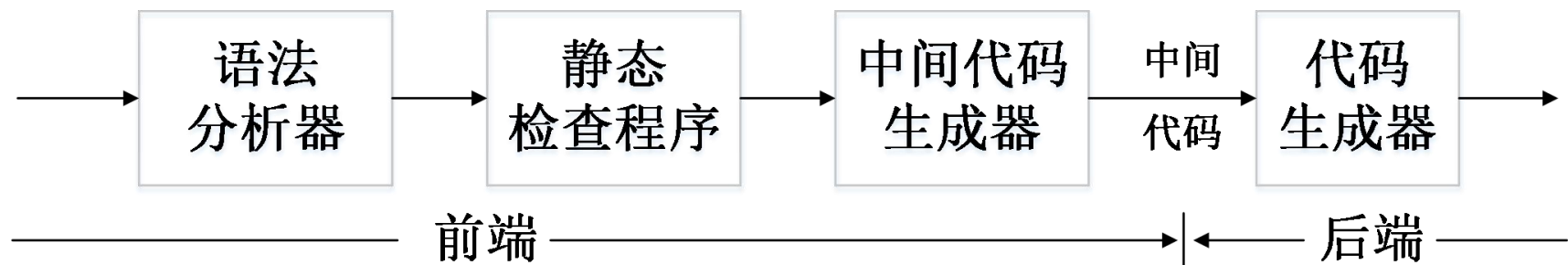


# 第六章 中间代码生成

---



# 中间代码生成



“中间代码生成”程序的**任务**是：把经过语法分析和语义分析而获得的源程序中间表示翻译为中间代码表示。

方法：语法制导翻译。

采用独立于机器的中间代码的好处：

1. 便于编译系统建立和编译系统的移植；
2. 便于进行独立于机器的代码优化工作。

# 学习内容

---

- 6.1 类型检查
- 6.2 中间表示
- 6.3 声明语句
- 6.4 赋值语句
- 6.5 控制流
- 6.6 回填
- 6.7 switch语句
- 6.8 过程的中间代码



# 学习内容

---

- 6.1 类型检查
- 6.2 中间表示
- 6.3 声明语句
- 6.4 赋值语句
- 6.5 控制流
- 6.6 回填
- 6.7 switch语句
- 6.8 过程的中间代码



# 类型

---

- 类型检查

- 利用一组逻辑规则来确定程序在运行时的行为
  - ✓ 保证运算分量的类型和运算符的预期类型匹配

- 翻译时的应用

- 确定一个名字需要的存储空间
- 计算一个数组元素引用的地址
- 插入显式的类型转换
- 选择算术运算符的正确版本



# 类型表达式

---

- 描述类型的结构
- 类型可以是基本类型，也可以是类型构造符（类型构造算子）作用于类型而得。
- 类型表达式：
  - 基本类型: *boolean, char, integer, float, void*
  - 类型名
  - 类型构造符



# 类型表达式（续）

---

- a) 数组：T是类型表达式，I为索引集合（整数范围），则`array(I, T)`是一个类型表达式，表示元素为类型T的数组类型

`int A[10];`——`array({0, ..., 9}, integer)` or  
——`array(10, integer)`

- b) 笛卡儿积：T<sub>1</sub>、T<sub>2</sub>为类型表达式，则T<sub>1</sub> × T<sub>2</sub>为类型表达式

- c) 记录：与笛卡儿积的不同之处仅在于记录的域有名字。<域名，域类型>元组

```
typedef struct {  
    int address;  
    char lexeme[15];  
} row;
```

`row:` —— `record ((address × integer) × (lexeme × array(15, char)))`



# 类型表达式（续）

---

- d) 指针：T为类型表达式，则`pointer(T)`为类型表达式，表示“指向类型为T的对象的指针”类型

`row *p; — pointer(row)`（声明变量p具有`pointer(row)`类型）

- e) 函数：数学上，一个集合“定义域”到另一个集合“值域”的映射。程序语言，定义域类型D到值域类型R的映射：

$D \rightarrow R$

`int *f(char a, char b); —  $(\text{char} \times \text{char}) \rightarrow \text{pointer}(\text{integer})$`

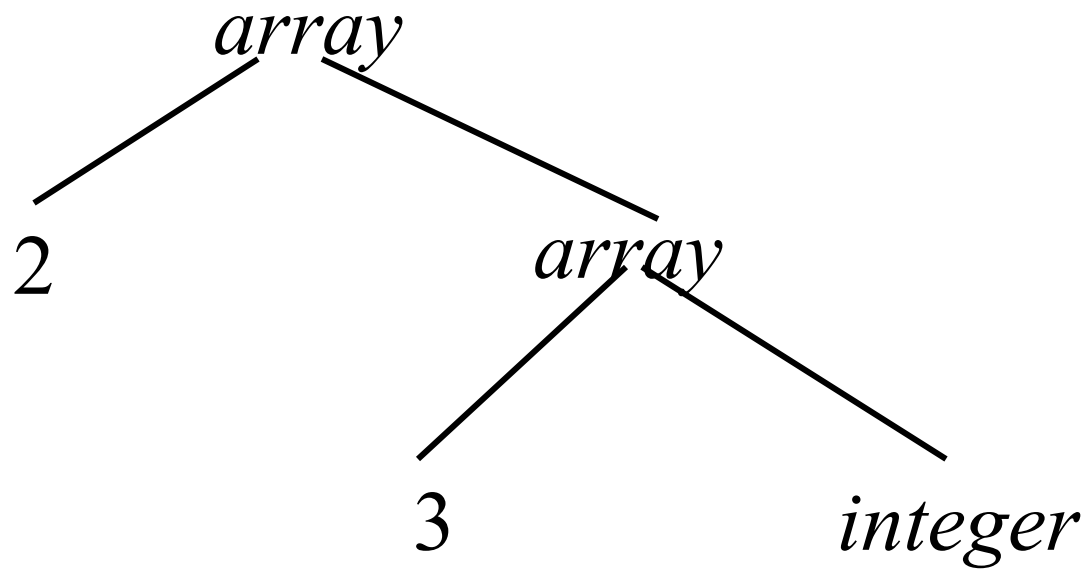


# 类型表达式（图表示法）

---

`array(2, array(3,integer) )`

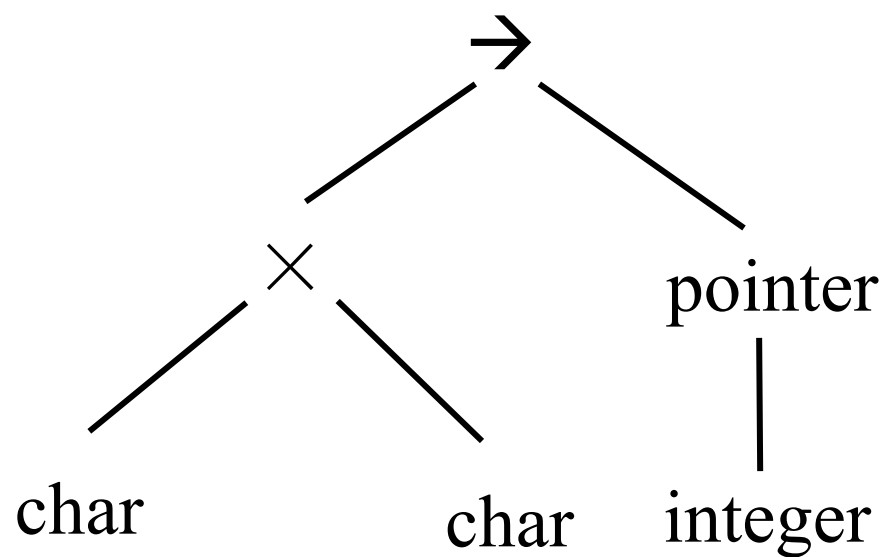
`int[2][3]`



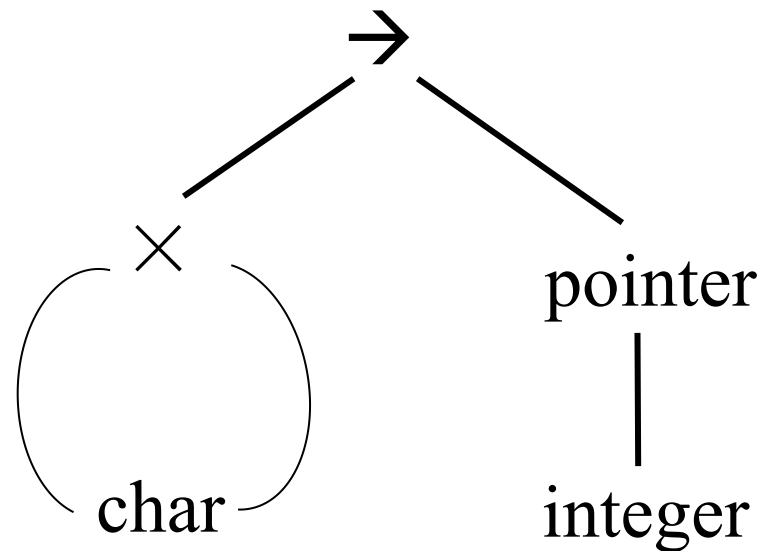
树形表示

# 类型表达式（例）

`int *f(char a, char b);`       $(\text{char} \times \text{char})$        $\rightarrow$    `pointer(int)`



语法树



DAG



# 类型表达式(例)

---

```
typedef struct person={  
    char name[8];  
    int sex;  
    int age;  
}
```

struct person table[50];

则person之类型表达式:

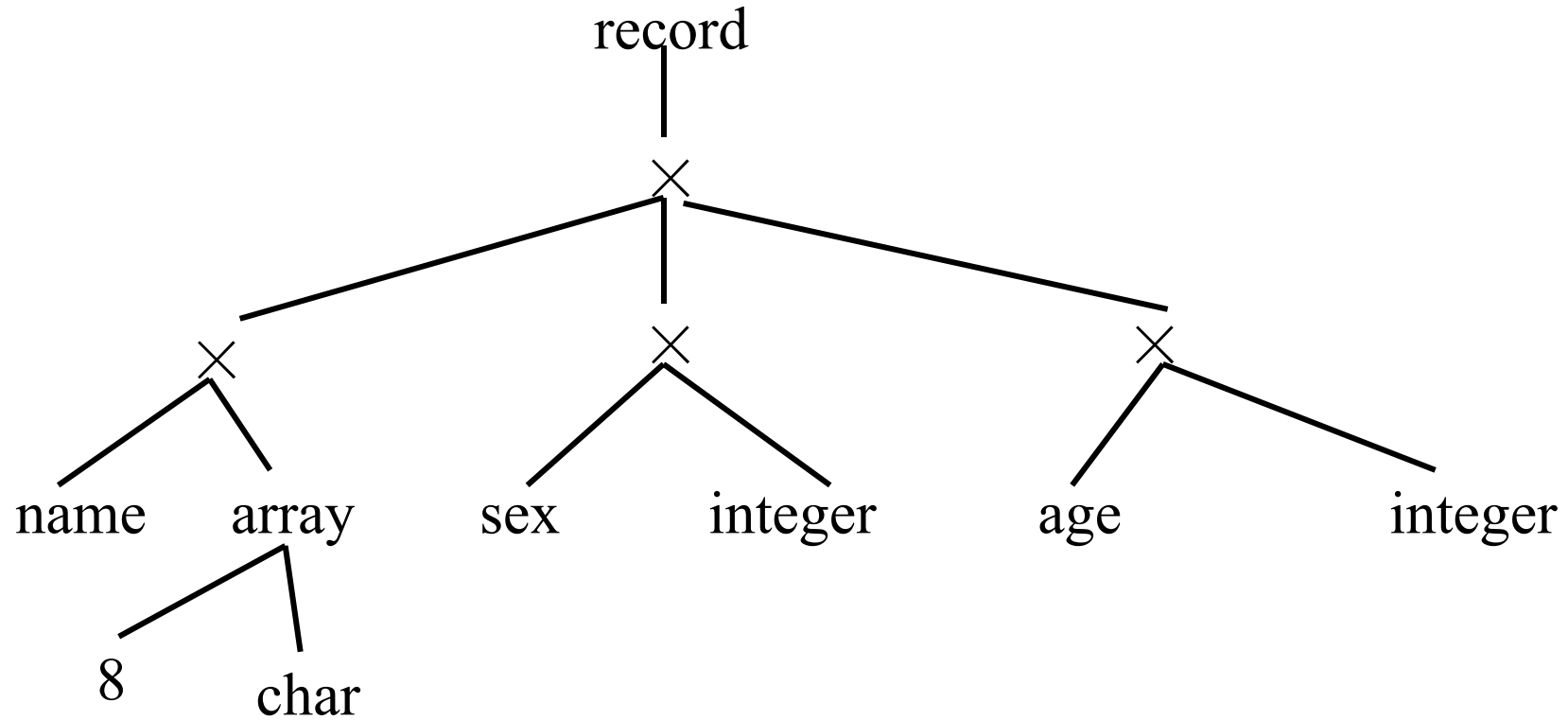
record( ( name  $\times$  array(8,char) )  $\times$  ( sex  $\times$  integer )  $\times$  ( age  $\times$  integer ) )

table之类型表达式:

array( 50, person )

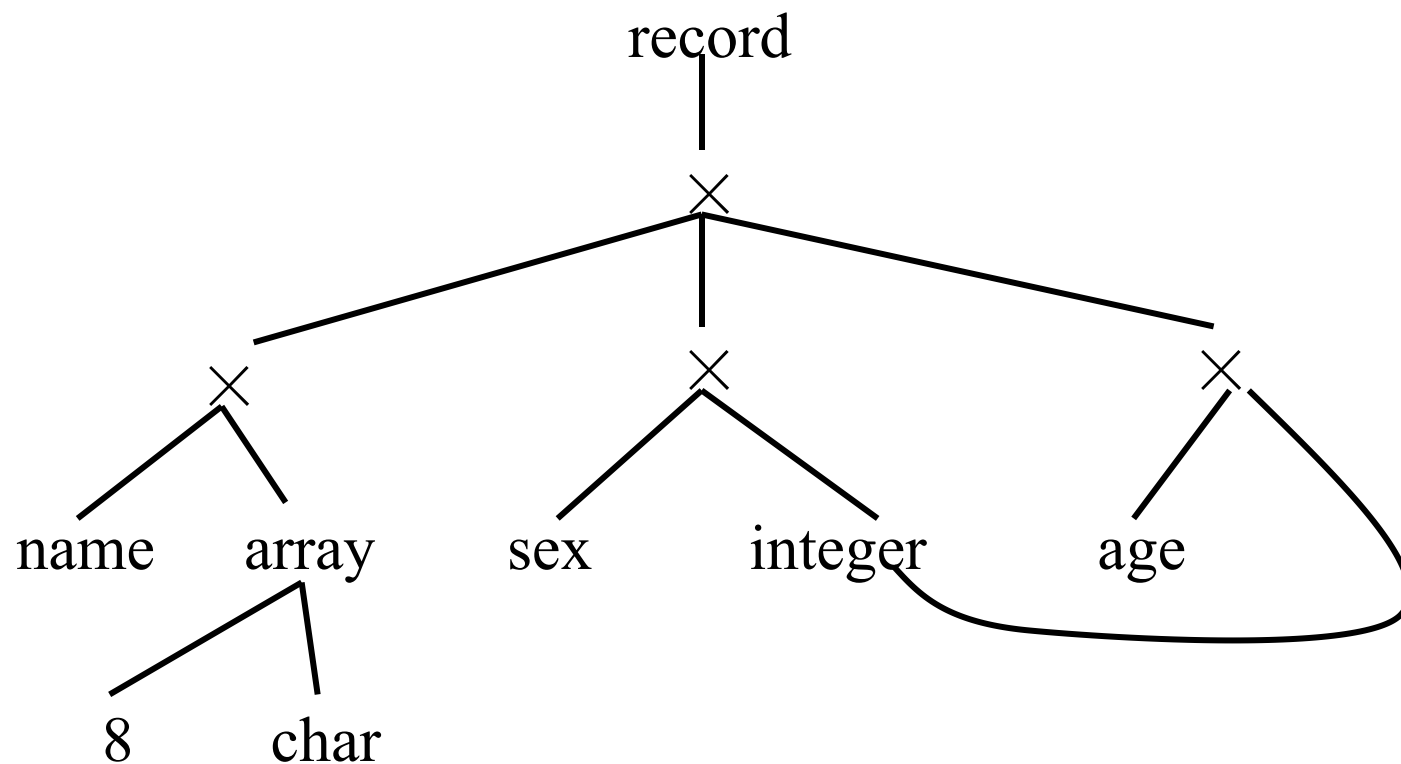


# 类型表达式(例)



`record((name × array(8,char)) × (sex × integer) × (age × integer))`

# 类型表达式(例)



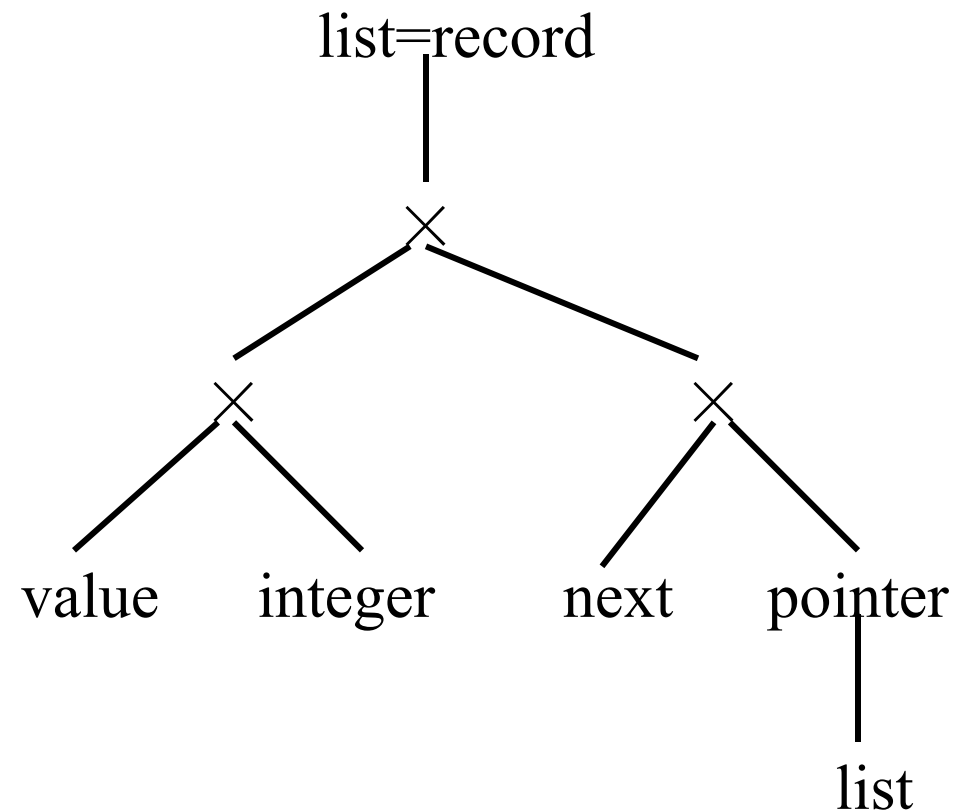
`record((name × array(8,char)) × (sex × integer) × (age × integer))`



# 类型表达式（例）

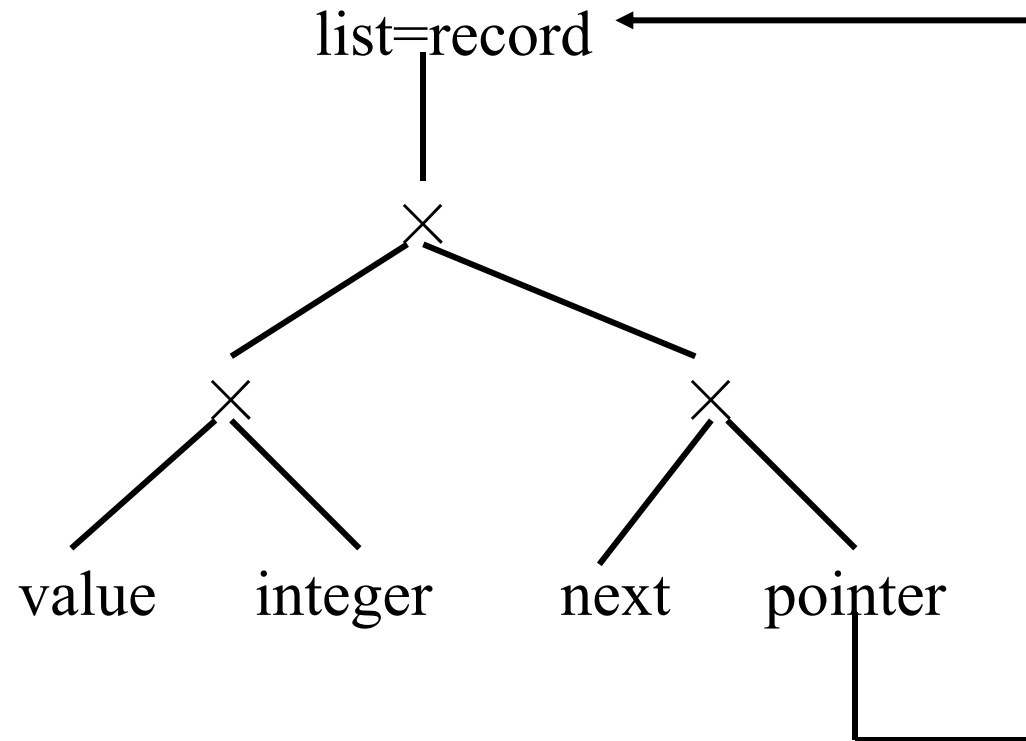
---

```
struct list{  
    int value;  
    struct list *next;  
}
```



# 类型表达式（例）

```
struct list{  
    int value;  
    struct list *next;  
}
```



# 类型等价

---

◦ **结构等价**：满足以下条件之一：

- 相同的基本类型
- 将相同类型构造算子应用于等价的类型而构建的
- 一个类型是另一个类型表达式的名字

◦ **名等价**：满足前两个条件

**结构等价**：类型名被类型表达式所代替，if 替换所有名字后，两个类型表达式结构上等价

**名等价**：将每个类型名看作是可区分的类型，名字完全相同





# 类型等价

---

例

type link =  $\uparrow$ cell;      p,q,r,s 类型相同 ?

var p,q : link;

var r,s :  $\uparrow$ cell;

名字等价:

p和q类型相同;   r和s类型相同;   p和r 类型不同

结构等价:

p,q,r,s 类型相同



# 类型的结构

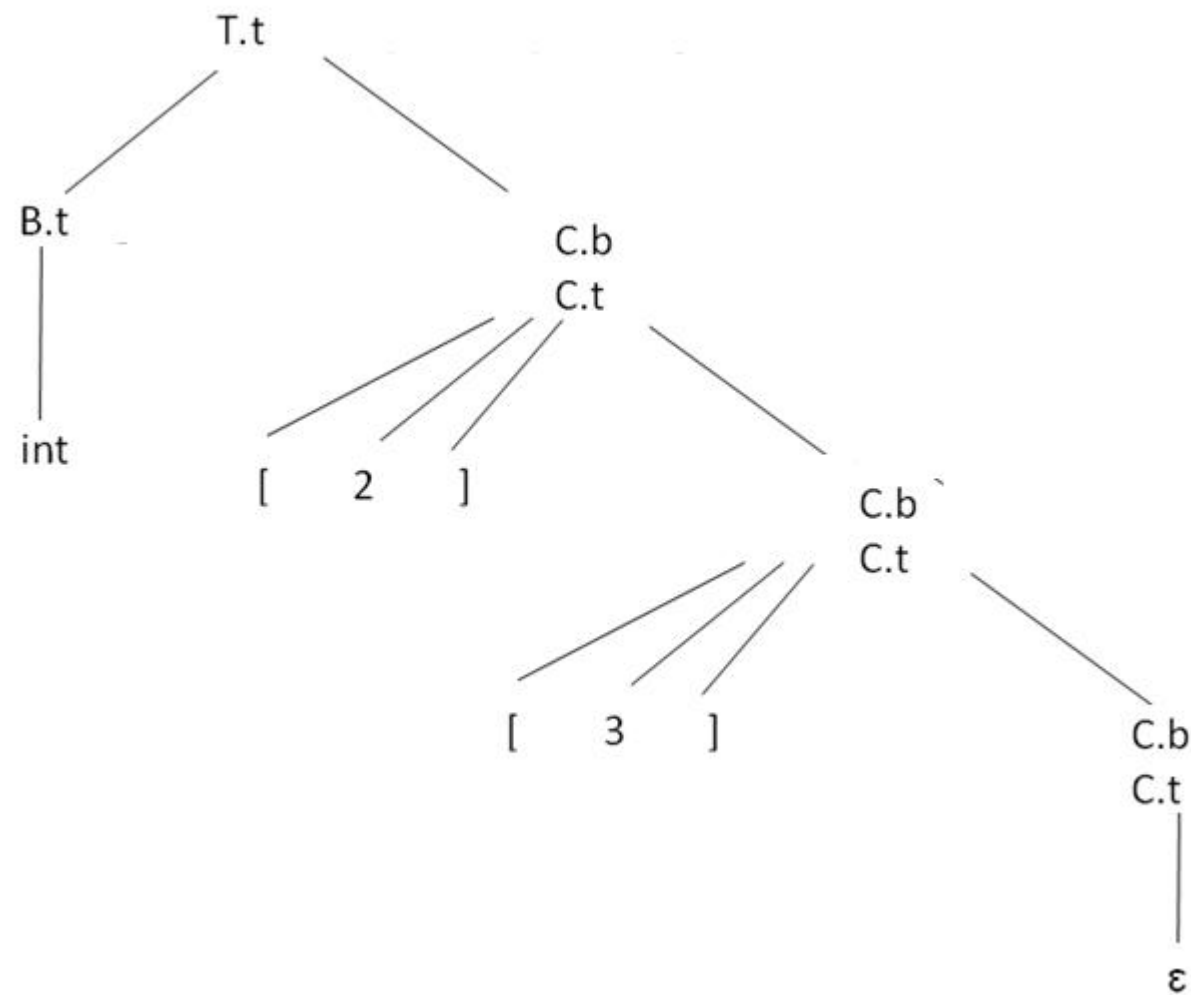
产生式	语义规则
$T \rightarrow BC$	$T.t = C.t$ $C.b = B.t$
$B \rightarrow \text{int}$	$B.t = \text{integer}$
$B \rightarrow \text{float}$	$B.t = \text{float}$
$C \rightarrow [\text{num}] C_1$	$C.t = \text{array}(\text{num.val}, C_1.t)$ $C_1.b = C.b$
$C \rightarrow \varepsilon$	$C.t = C.b$

T 生成一个基本类型或一个数组类型



# 类型的结构

数组类型的注释语法分析树



# 类型的结构

$E \rightarrow BC$

$T.t = C.t$

$C.b = B.t$

$B \rightarrow \text{int}$

$B.t = \text{integer}$

$B \rightarrow \text{float}$

$B.t = \text{float}$

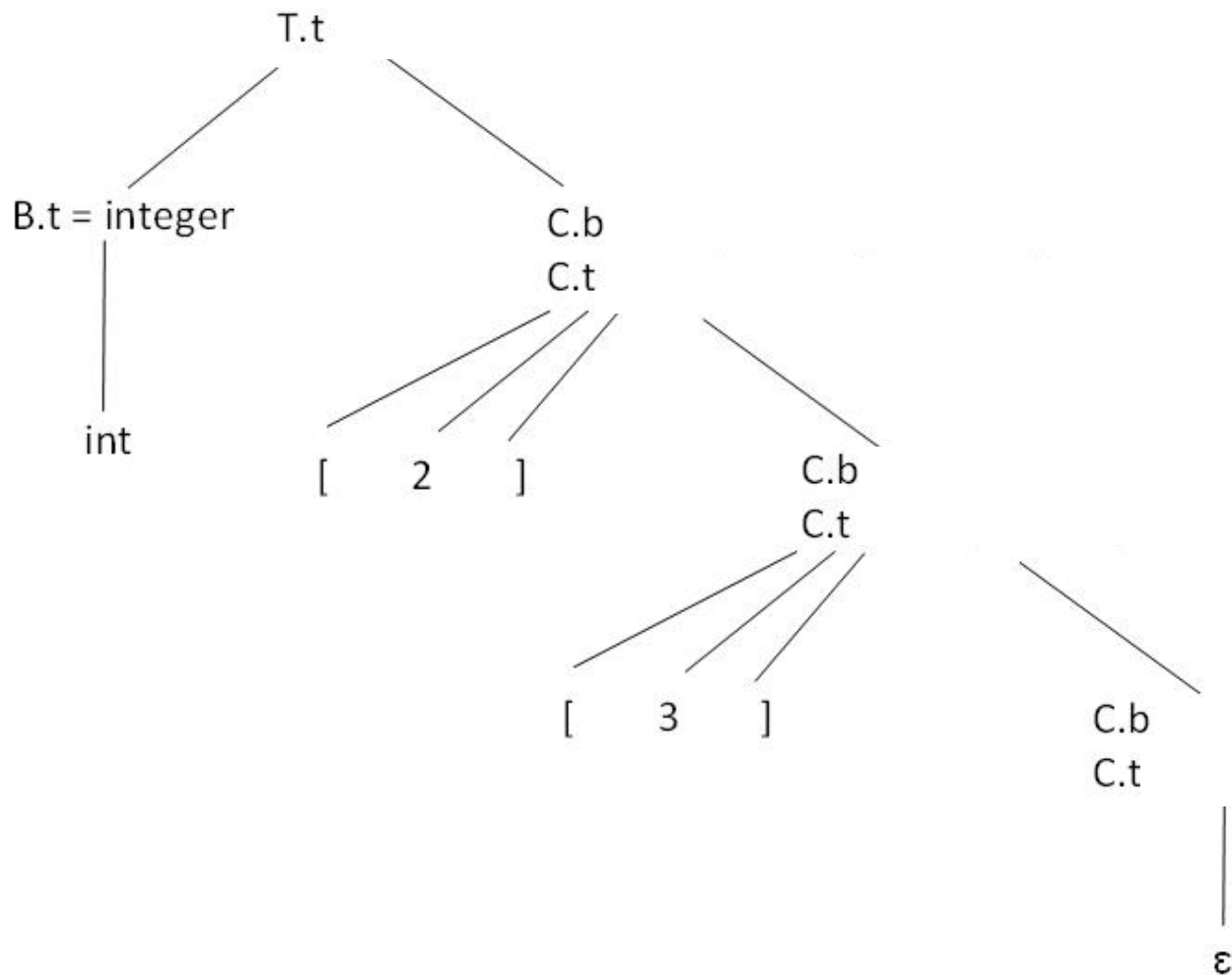
$C \rightarrow [\text{num}] C_1$

$C.t = \text{array}(\text{num.val}, C_1.t)$

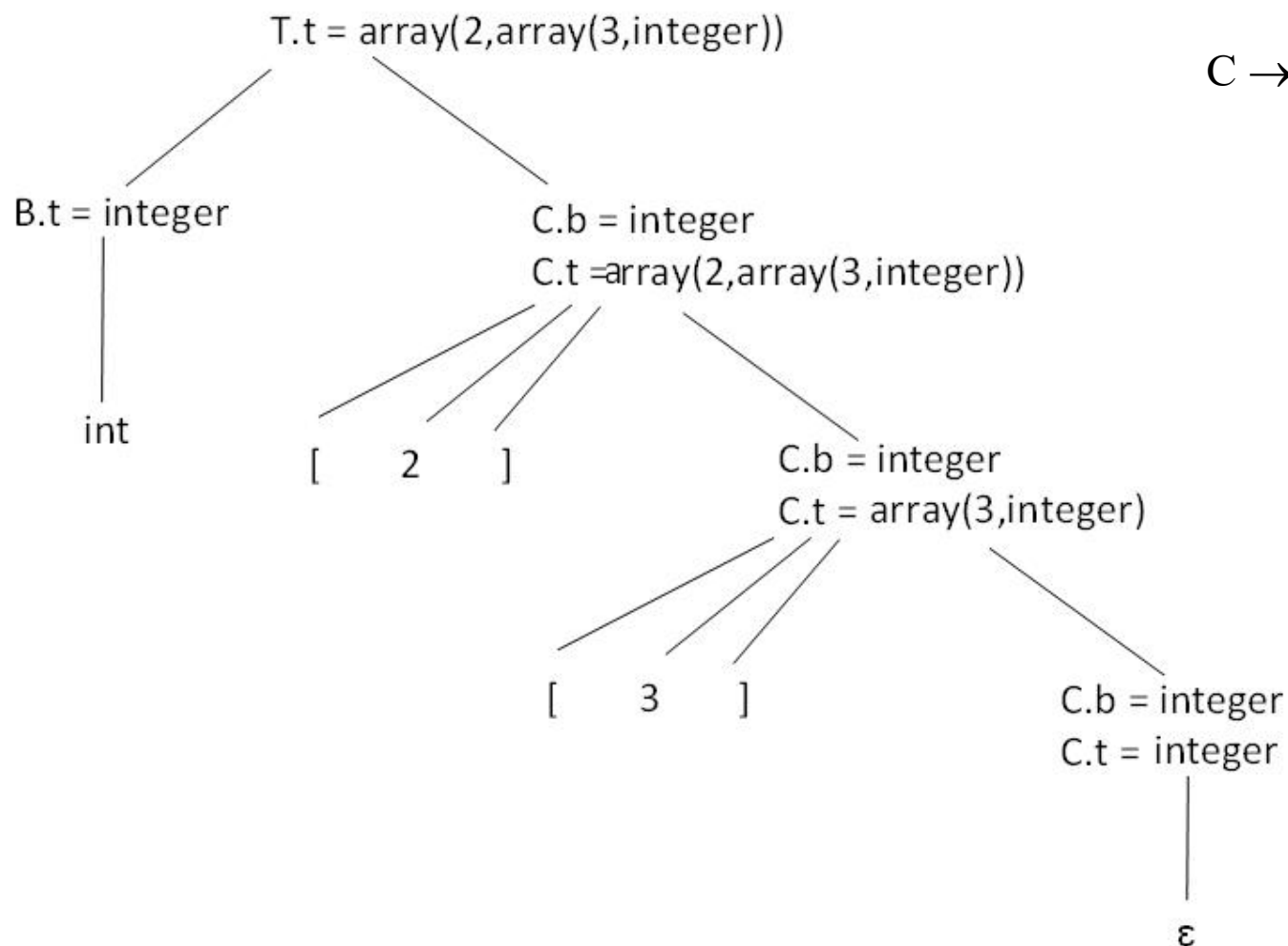
$C_1.b = C.b$

$C \rightarrow \epsilon$

$C.t = C.b$



# 类型的结构



$E \rightarrow BC$

$T.t = C.t$

$C.b = B.t$

$B \rightarrow \mathbf{int}$

$B.t = \mathbf{integer}$

$B \rightarrow \mathbf{float}$

$B.t = \mathbf{float}$

$C \rightarrow [\mathbf{num}] C_1$

$C.t = \mathbf{array}(\mathbf{num.val}, C_1.t)$

$C_1.b = C.b$

$C \rightarrow \varepsilon$

$C.t = C.b$



# 类型检查

---

每个程序设计语言都有自己的类型机制，包括类型说明和使用。

类型检查完成下面的主要任务：

- 进行类型转换
- 判定重载算符(函数)在程序中代表的是哪一个运算
- 对语言结构进行类型检查。如: **Pascal**语言中对数据类型的使用要进行同一、相容和赋值相容检查



# 类型转换

---

$x+i$ ,  $x$ 为浮点型,  $i$ 为整型

- 不同保存格式, 不同加法指令
- 转换为相同类型进行运算

`float(i)`:将整数转换为浮点数

$2 * 3.14$

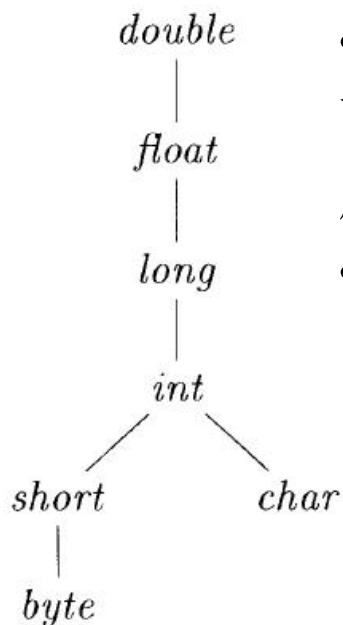
$t_1 = \text{float}(2)$

$t_2 = t_1 * 3.14$



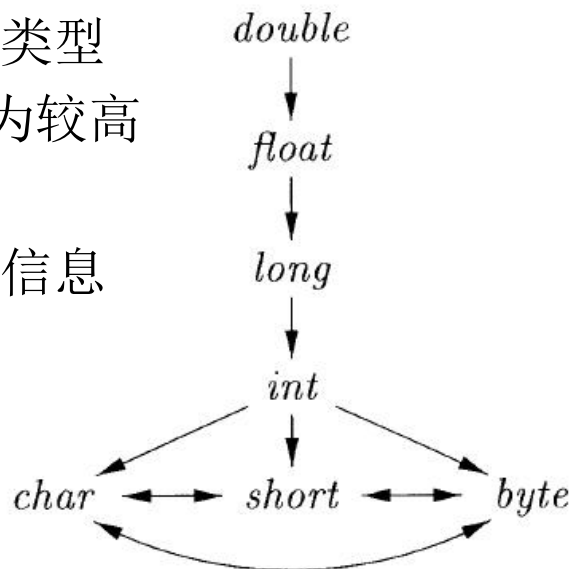
# 类型的widening和narrowing

Java类型转换规则区分了拓宽转换和窄化转换



a) 拓宽类型转换

- 较低层的类型可被拓宽为较高层的类型
- 保持原有信息



b) 窄化类型转换

- 如果存在一条从s到t的路径，则可以将类型s窄化为类型t
- 可能丢失信息

• 编译器自动完成的转换为**隐式转换**，程序员用代码指定的转换为**显式转换**。





# 处理类型转换的SDT

在表达式计算中引入类型转换

## widen函数的伪代码

```
Addr widen(Addr a, Type t, Type w)
    if ( t = w ) return a;
    else if ( t = integer and w = float ) {
        temp = new Temp();
        gen(temp '=' '(float)' a);
        return temp;
    }
    else error;
}
```

Max求的是两个类型在拓宽层次结构中的最大者

```

$$E \rightarrow E_1 + E_2 \quad \{ \begin{array}{l} E.type = \max(E_1.type, E_2.type); \\ a_1 = \text{widen}(E_1.addr, E_1.type, E.type); \\ a_2 = \text{widen}(E_2.addr, E_2.type, E.type); \\ E.addr = \text{new Temp}(); \\ \text{gen}(E.addr '=' a_1 '+' a_2); \end{array} \}$$

```



# 类型检查规则

---

## 类型综合

- 根据子表达式的类型构造出表达式的类型
- 要求先声明再使用
- if  $f$  的类型为  $s \rightarrow t$ , 且  $x$  的类型为  $s$   
then 表达式  $f(x)$  的类型为  $t$

## 类型推导

- 根据一个语言结构的使用方式来确定该结构的类型
- If  $f(x)$  是一个表达式  
Then 对某些  $\alpha$  和  $\beta$ ,  $f$  的类型为  $\alpha \rightarrow \beta$  且  $x$  的类型为  $\alpha$



# 函数和运算符重载

---

重载（**overloaded**）符号：根据上下文，具有不同的意义

- **+**：整型加法，浮点型加法
- **A(I)**：数组A的第I个元素，以参数I调用A，将I转换为类型A的显式类型转换

重载的解析（**resolved**）：在某个特定上下文，确定符号的唯一意义

- **1+2**：**+**为整型加法



# 子表达式可能类型集合

---

Ada允许对乘法运算符“\*”进行重载

- function “\*” (i, j : integer) return complex;
- function “\*” (x, y : complex) return complex;

\*具有三种可能类型

1.  $integer \times integer \rightarrow integer$
  2.  $integer \times integer \rightarrow complex$
  3.  $complex \times complex \rightarrow complex$
- $2*(3*5) \rightarrow 3*5$ ——类型1
  - $z*(3*5)$ ,  $z$ 为复数类型  $\rightarrow 3*5$ ——类型2



# 针对重载函数的类型综合规则

---

**if**  $f$  可能的类型为  $s_i \rightarrow t_i$  ( $1 \leq i \leq n$ )

**and**  $x$  的类型为  $s_k$  ( $1 \leq k \leq n$ )

**then** 表达式  $f(x)$  的类型为  $t_k$



# 类型推导和多态（polymorphic）函数

---

类型推导：常用于像ML这样的语言

普通函数：参数类型固定

多态函数：不同调用参数可为不同类型

某些内置操作符——多态操作符

- 数组索引符[], 函数调用符(), 指针操作符&
- &: “若操作对象类型为..., 则操作结果的类型为指向...的指针”



# 求任何类型列表长度的ML程序

---

**递归函数**

```
fun length(x) =  
  if null(x) then 0  
  else length(tl(x)) + 1;
```

**列表为空?**

**列表剩余部分**

可应用于任何类型的列表

- `length(["sun", "mon", "tue"]);`
- `length([10, 9, 8]);`



# 类型变量

---

a, b, ..., 表示未知类型

重要应用：不要求标识符先声明后使用的语言中，检查标识符使用的一致性

类型变量表示未声明标识符的类型

- 若类型变量发生变化，不一致！
- 若一直未变化，一致！同时得到标识符类型

类型推断，**type inference**

- 根据语言结构的使用方式判定其类型





# 例:

---

```
function deref(p);
```

```
begin
```

```
  return p^;
```

```
end;
```

扫描第一行，**p**的类型未知，用**b**表示

第三行，**^**应作用于指针，因此**p**为某未知基本类型**a**的指针类型，

**b=pointer(a)**

因此函数**deref**类型为：**pointer(a)→a**



# 置换，实例和合一

---

变量表示实际类型的形式化定义

类型变量  $\rightarrow$  类型表达式的映射，称为置换，substitution

```
subst(t : type_expression) : type_expression
{
    //利用置换（映射）S将类型表达式t中变量置换
    if (t为基本类型) return t;
    else if (t为类型变量) return S(t);
    else if (t为 $t_1 \rightarrow t_2$ ) return subst( $t_1$ )  $\rightarrow$  subst( $t_2$ );
}
```

$S(t)$ 表示置换t的类型表达式，称为t的实例，instance



# 置换，实例和合一

---

是实例的情况

- $pointer(integer) < pointer(a)$
- $pointer(real) < pointer(a)$
- $integer \rightarrow integer < a \rightarrow a$
- $pointer(a) < b$
- $a < b$

不是实例的情况

- $integer$  和  $real$
- $integer \rightarrow real$  和  $a \rightarrow a$
- $integer \rightarrow a$  和  $a \rightarrow a$



# 合一方法

---

类型表达式 $t_1$ 和 $t_2$ 能合一的条件→

存在置换 $S$ ,  $S(t_1) = S(t_2)$

最一般的合一置换, **most general unifier**——最少变量约束的置换

- 类型表达式 $t_1$ 和 $t_2$ 的最一般的合一置换是一个置换 $S$ , 它满足以下条件
  1.  $S(t_1) = S(t_2)$
  2. 对任何其他置换 $S'$ ,  $S'(t_1) = S'(t_2)$ ,  $S'$ 是 $S$ 的一个实例, 即, 对任意 $t$ ,  $S'(t)$ 是 $S(t)$ 的一个实例

# 例：多态函数的类型推导

---

**fun**  $\text{id}_0$  (  $\text{id}_1, \dots, \text{id}_k$  ) = E;

$\text{id}_0$ : 函数名,  $\text{id}_1, \dots, \text{id}_k$ : 参数, E遵从前面定义的用于多态函数类型检查的文法, 其中的标识符只可能是函数名、参数或内置函数

方法:

- 为函数名和参数创建新~~类型~~变量
- 多态函数的类型变量由 约束
- 检查 $\text{id}_0$  (  $\text{id}_1, \dots, \text{id}_k$  )和E类型是否匹配
- 若成功, 则推断出函数类型



例:

```
fun length(x) =  
  if null(x) then 0  
  else length(tl(x)) + 1;
```

表达式: 类型	合一
$x : \beta$	
$\text{length} : \beta \rightarrow \gamma$	
$x : \beta$	
$\text{null} : \text{list}(\alpha_n) \rightarrow \text{boolean}$	
$\text{null}(x) : \text{boolean}$	$\beta = \text{list}(\alpha_n)$
$0 : \text{integer}$	
$x : \text{list}(\alpha_n)$	
$\text{tl} : \text{list}(\alpha_t) \rightarrow \text{list}(\alpha_t)$	
$\text{tl}(x) : \text{list}(\alpha_n)$	$\alpha_t = \alpha_n$



# 例：（续）

```
fun length(x) =
  if null(x) then 0
  else length(tl(x)) + 1;
```

表达式: 类型	合一
$\text{length} : \text{list}(\alpha_n) \rightarrow \gamma$	
$\text{length}(\text{tl}(x)) : \gamma$	
$1 : \text{integer}$	
$+ : \text{integer} \times \text{integer} \rightarrow \text{integer}$	
$\text{length}(\text{tl}(x)) + 1 : \text{integer}$	$\gamma = \text{integer}$
$\text{if} : \text{boolean} \times \alpha_i \times \alpha_i \rightarrow \alpha_i$	
$\text{if} (...) : \text{integer}$	$\alpha_i = \text{integer}$

$\alpha_n$  最终未被替代, **length** 类型为  $\forall \alpha_n. \text{list}(\alpha_n) \rightarrow \text{integer}$



# 学习内容

---

- 6.1 类型检查
- 6.2 中间表示
- 6.3 声明语句
- 6.4 赋值语句
- 6.5 控制流
- 6.6 回填
- 6.7 switch语句
- 6.8 过程的中间代码





# 中间表示

---

- ❖ 优点: 容易为不同目标机器开发不同后端
- ❖ 缺点: 编译过程变慢 (因为中间步骤)
- ❖ 中间表示:
  - 语法树
  - 三地址代码表示



# 中间表示-抽象语法树

---

## 抽象语法树(AST)

- 抽象语法树（或者简称为语法树）：反映了抽象的语法结构，而分析树反映的是具体的语法结构。语法树是分析树的抽象形式或压缩形式



# 中间表示-抽象语法树

---

## 抽象语法(Abstract Syntax)

- 从具体语法中**抽象出**语言结构的本质性的东西，而不考虑语言的具体符号表示，从而可简化语义的形式描述。
- 在不同的语言中赋值语句有不同的写法

$x=y$ ;  $x := y$ ;  $y \rightarrow x$  等等

- 可以用抽象形式

assignment(variable, expression)

把前面各种具体形式统一起来。



# 中间表示-抽象语法树

---

语法规则中包含的某些符号可能起**标点符号作用**，也可能起**解释作用**。

回顾前述的赋值语句，其产生式规则是

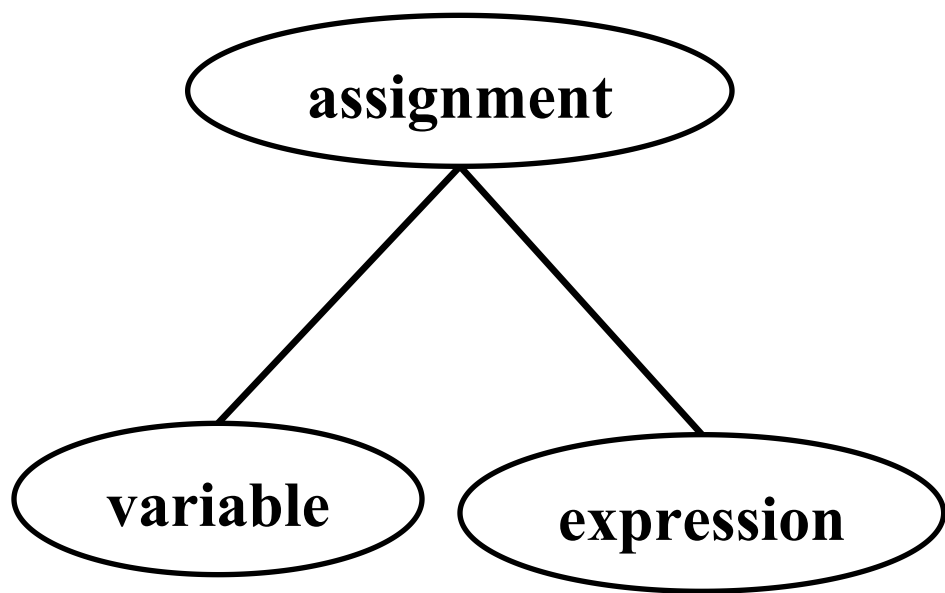
- $S \rightarrow V = e$
- 其中的赋值号“=”仅仅起标点符号作用，目的是把V和e分隔开

而条件语句的产生式规则：

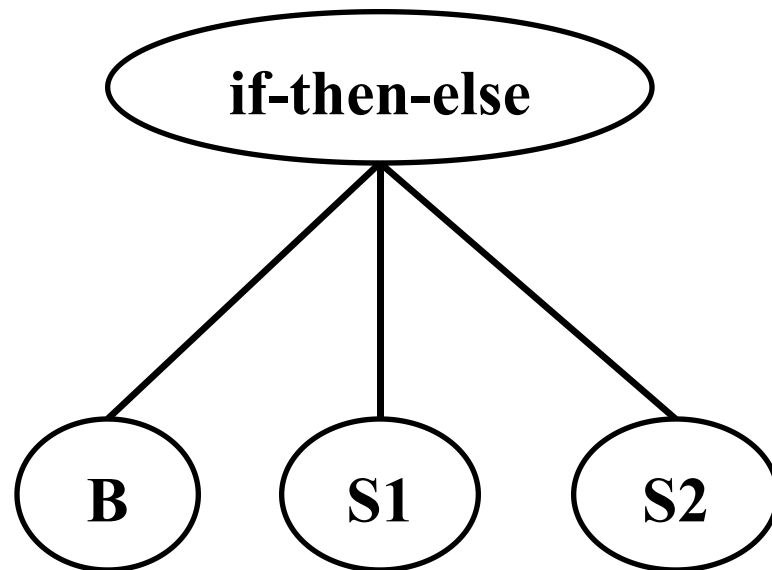
- $S \rightarrow \text{if } B \text{ then } S_1 \text{ else } S_2$
- 其中的关键字if、then、else起注释作用，说明当布尔表达式B为真时执行 $S_1$ 语句，否则执行 $S_2$



# 中间表示-抽象语法树



赋值语句语法树



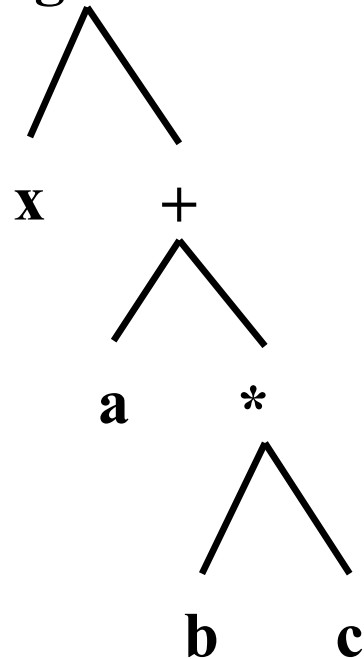
条件语句语法树

在语法树中，运算符和关键字都不在叶结点，而是在内部结点中出现。

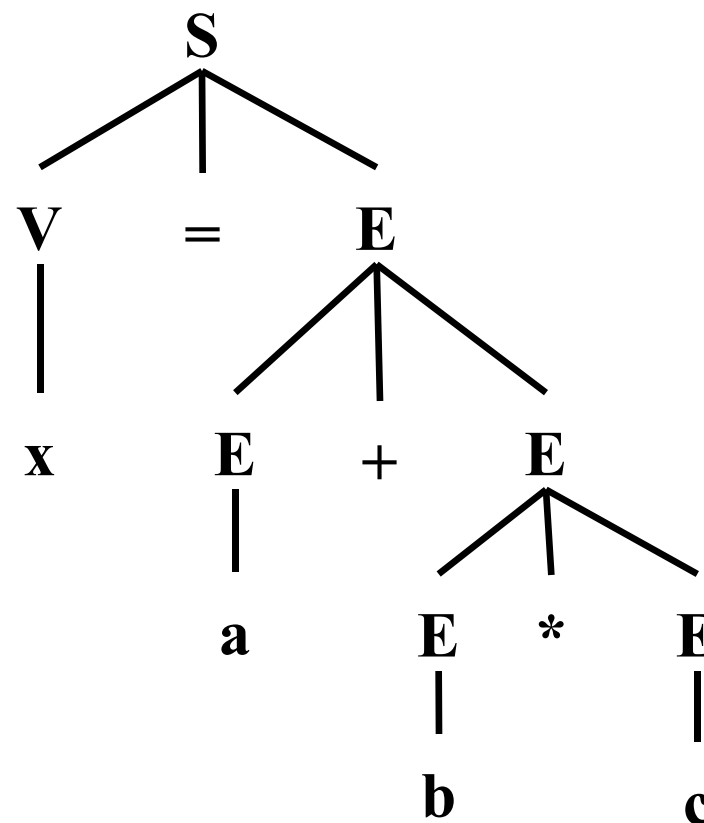
# 中间表示-抽象语法树

赋值语句 $x=a+b*c$ 的抽象语法树和分析树

**assignment**



抽象语法树



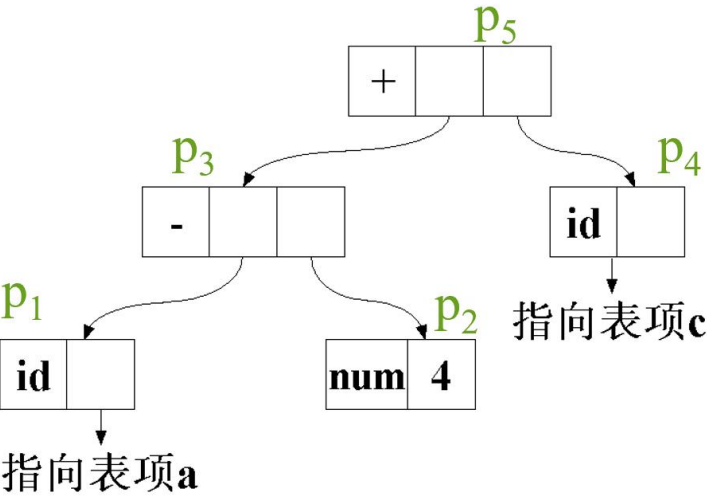
分析树



a-4+c的抽象语法树

产生式	语义规则
$E \rightarrow E_1 + T$	$E.nptr = mknode(+, E_1.nptr, T.nptr)$
$E \rightarrow E_1 - T$	$E.nptr = mknode(-, E_1.nptr, T.nptr)$
$E \rightarrow T$	$E.nptr = T.nptr$
$T \rightarrow (E)$	$T.nptr = E.nptr$
$T \rightarrow id$	$T.nptr = mkleaf(id, id.lexval)$
$T \rightarrow num$	$T.nptr = mkleaf(num, num.val)$

产生式	语义规则
$E \rightarrow T E'$	$E.node = E'.syn$ $E'.inh = T.node$
$E' \rightarrow + T E_1'$	$E_1'.inh = mknode(+, E'.inh, T.node)$ $E'.syn = E_1'.syn$
$E' \rightarrow - T E_1'$	$E_1'.inh = mknode(-, E'.inh, T.node)$ $E'.syn = E_1'.syn$
$E' \rightarrow \epsilon$	$E'.syn = E'.inh$
$T \rightarrow (E)$	$T.node \rightarrow E.node$
$T \rightarrow id$	$T.node = mkleaf(id, id.lexval)$
$T \rightarrow num$	$T.node = mkleaf(num, num.val)$



# 中间表示-有向无环图（DAG）

---

有向无环图(Directed Acyclic Graph, DAG): 抽象语法树的变体  
用途: 提取表达式中的公共子表达式, 以取得目标程序的局部优化。

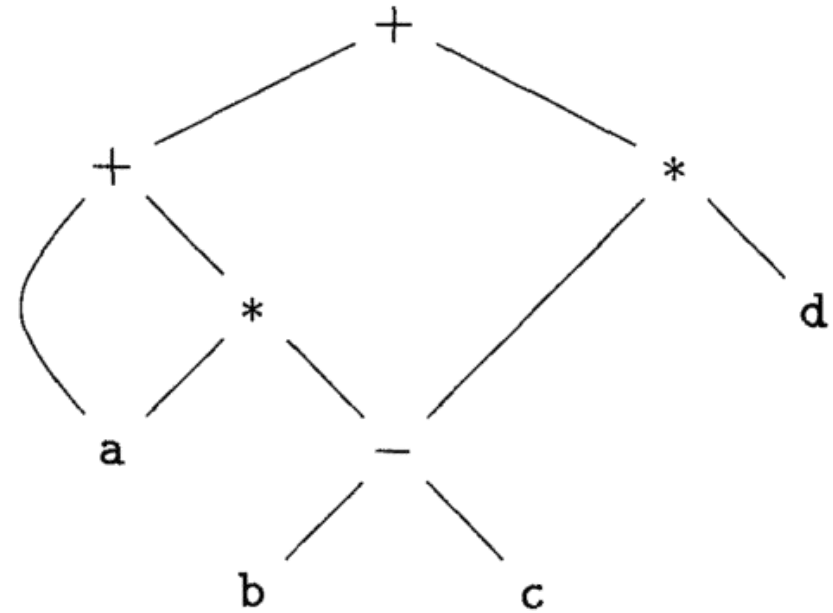
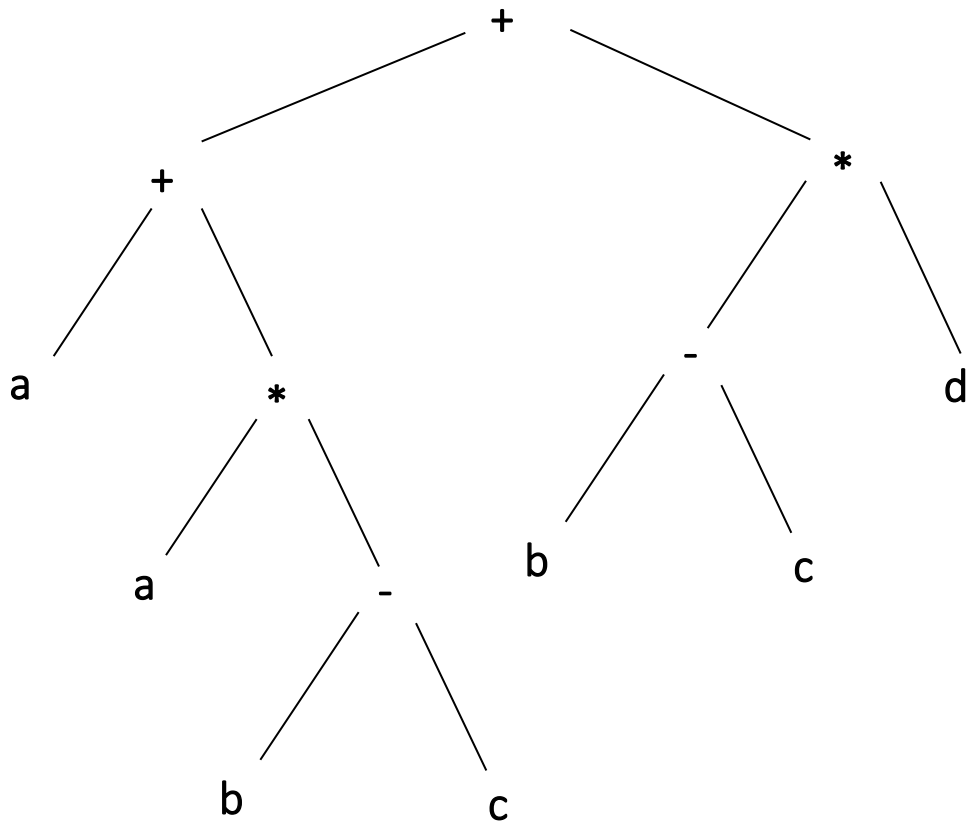
- 方法: 执行mknode和mkleaf时, 检查是否已有相同的结点, 若有, 则返回相应结点的指针。
- 与语法树的区别: 语法树中公共子表达式由重复的子树表示, 而 DAG 中只用一个子树表示, 因此代表公共子表达式的结点有多个父节点





# 中间表示-有向无环图 (DAG)

**$a + a * (b - c) + (b - c) * d$**



# 中间表示-有向无环图 (DAG)

生成语法树或DAG的语法制导定义

产生式	语义规则
$E \rightarrow E_1 + T$	$E.nptr = mknnode(+, E_1.nptr, T.nptr)$
$E \rightarrow E_1 - T$	$E.nptr = mknnode(-, E_1.nptr, T.nptr)$
$E \rightarrow T$	$E.nptr = T.nptr$
$T \rightarrow T_1 * F$	$T.nptr = mknnode(*, T_1.nptr, F.nptr)$
$T \rightarrow (E)$	$T.nptr = E.nptr$
$T \rightarrow id$	$T.nptr = mkleaf(id, id.entry)$
$T \rightarrow num$	$T.nptr = mkleaf(num, num.val)$

DAG

在构造结点前：  
检查现有结点，  
若存在相同结点  
则返回该结点的  
指针



# $a+a*(b-c)+(b-c)*d$ 的DAG的构造过程

```
p1 = mkleaf(id, entry-a);
```

**p2 = mkleaf(id, entry-a); = p1**

```
p3 = mkleaf(id, entry-b);
```

```
p4 = mkleaf(id, entry-c);
```

```
P5 = mknode('-', p3, p4);
```

```
P6 = mknode('*', p1, p5);
```

```
P7 = mknode('+', p1, p6);
```

**P8 = mkleaf(id, entry-b); = p3**

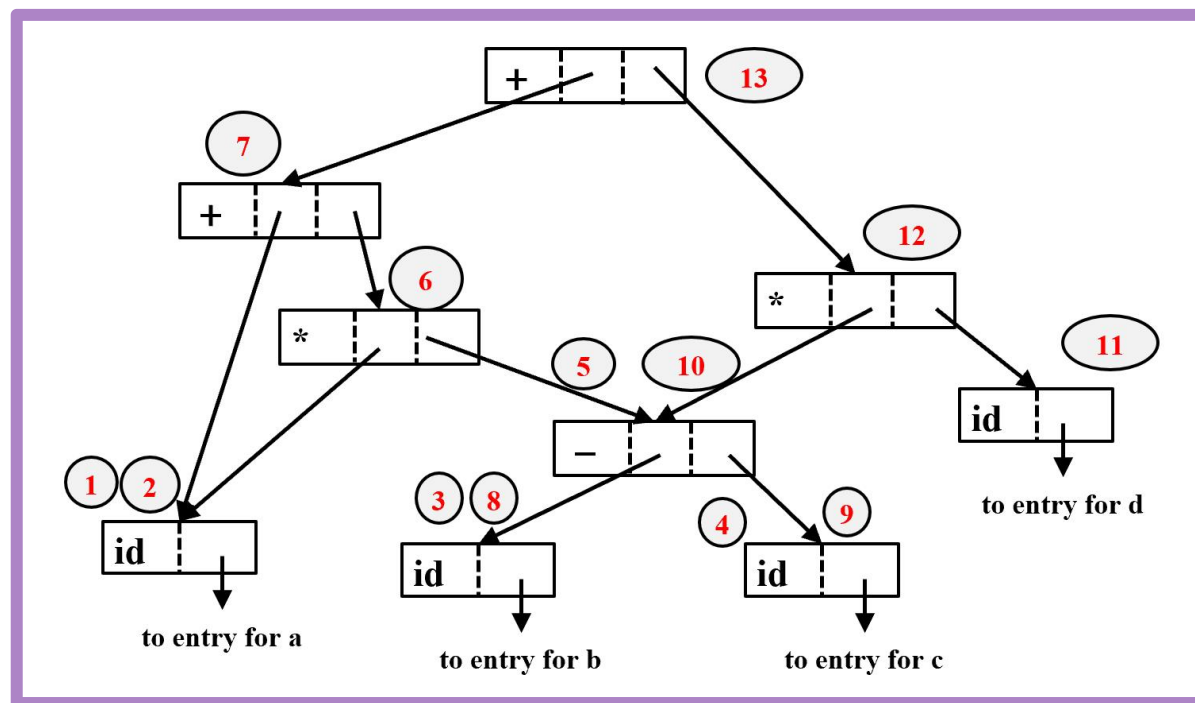
```
P9 = mkleaf(id, entry-c); = p4
```

```
P10 = mknode('-', p8, p9); = p5
```

```
p11 = mkleaf(id, entry-d);
```

```
P12 = mknode('*', p5, p11);
```

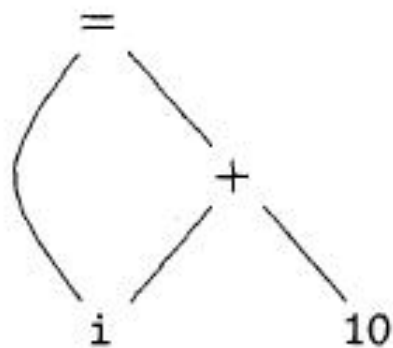
```
P13 = mknode('+', p7, p12)
```



# 构建DAG的值编码方法

□ 语法树或DAG的结点通常存放在一个记录数组中

- 数组的每一行表示一个记录



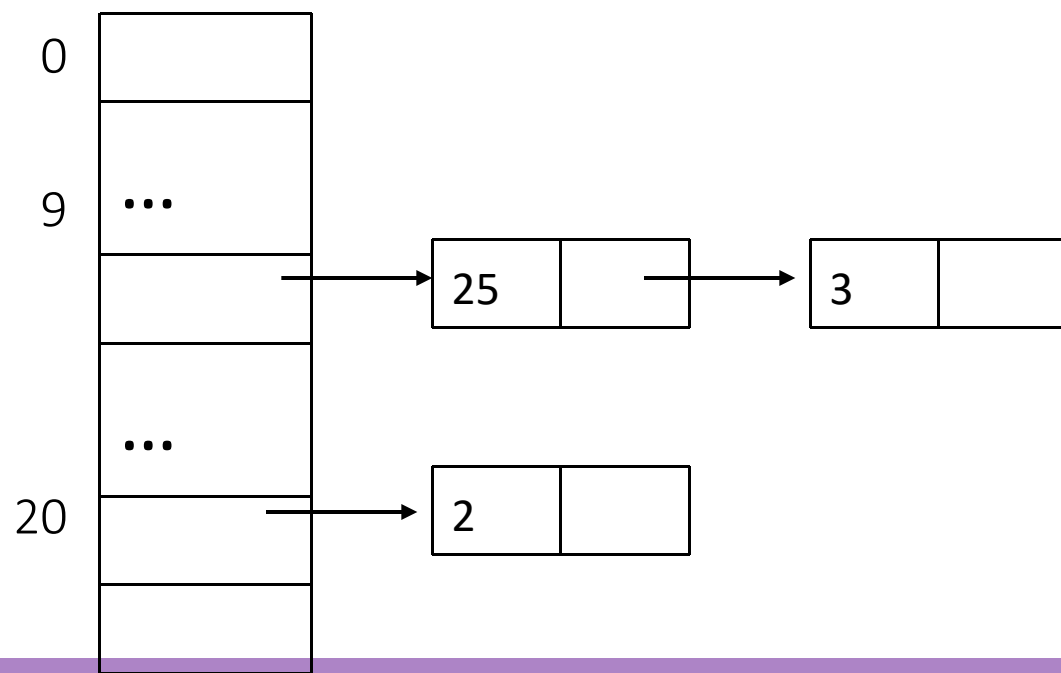
(a) DAG

1	id	to entry for i	
2	num	10	
3	+	1	2
4	=	1	3
5	...		

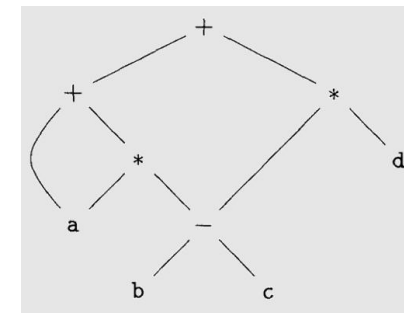
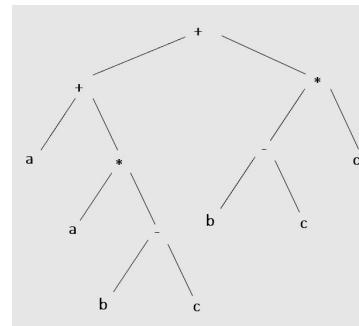
(b) Array.

# 构建DAG的值编码方法

- 高效定位结点：散列表（hash表）
- 结点 $\langle op, l, r \rangle$ 的结点的值编码： $h(op, l, r)$



# 中间表示-三地址代码



一般形式  $x = y \text{ op } z$ :

指令的右侧最多有一个运算符

表达式  $a + a * (b - c) + (b - c) * d$  的语法树和DAG对应的三地址代码

根据语法树

$$\begin{aligned} t_1 &= b - c \\ t_2 &= a * t_1 \\ t_3 &= a + t_2 \\ t_4 &= b - c \\ t_5 &= t_4 * d \\ t_6 &= t_3 + t_5 \end{aligned}$$

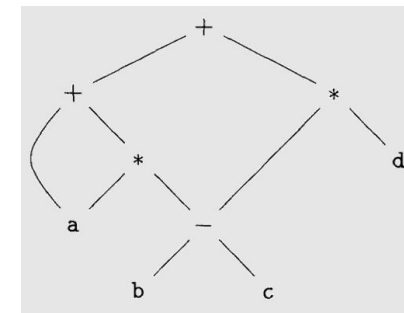
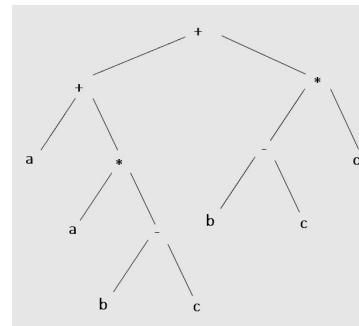
根据DAG

$$\begin{aligned} t_1 &= b - c \\ t_2 &= a * t_1 \\ t_3 &= a + t_2 \\ t_4 &= t_1 * d \\ t_5 &= t_3 + t_4 \end{aligned}$$

○ 三地址代码与语法树、DAG 的关系

三地址代码是语法树或DAG 的线性表示

# 中间表示-三地址代码



一般形式  $x = y \text{ op } z$ :

指令的右侧最多有一个运算符

表达式  $a + a * (b - c) + (b - c) * d$  的语法树和DAG对应的三地址代码

根据语法树

$$\begin{aligned} t_1 &= b - c \\ t_2 &= a * t_1 \\ t_3 &= a + t_2 \\ t_4 &= b - c \\ t_5 &= t_4 * d \\ t_6 &= t_3 + t_5 \end{aligned}$$

根据DAG

$$\begin{aligned} t_1 &= b - c \\ t_2 &= a * t_1 \\ t_3 &= a + t_2 \\ t_4 &= t_1 * d \\ t_5 &= t_3 + t_4 \end{aligned}$$

○ 三地址代码与语法树、DAG 的关系

三地址代码是语法树或DAG 的线性表示



# 三地址代码

---

两个基本概念：地址 和 指令

地址：

一般含三个地址(名字、常量、临时变量):两个操作分量和一个结果的抽象地址  
为方便起见,通常用变量名代替抽象地址

如, 源语言表达式 $x+y*z$ 可以被翻译为:

- $t_1 = y * z$
- $t_2 = x + t_1$
- 其中 $t_1$ 和 $t_2$ 是编译时产生的临时变量





# 三地址代码

---

两个基本概念：地址 和 指令

指令：

赋值指令，转移指令，过程调用和返回指令



# 三地址代码-----指令类型

---

(1)赋值语句  $x = y \text{ op } z$ ,  $\text{op}$ 为二目算术算符或逻辑算符

(2)赋值语句  $x = \text{op } y$ ,  $\text{op}$ 为一目算符, 如一目减 $\text{uminus}$ 、逻辑非 $\text{not}$ 、移位算符及转换算符

(3)地址和指针赋值语句

$x = \&y$

$x = *y$

$*x = y$

(4)无条件转移语句 $\text{goto } L$

(5)条件转移语句  $\text{if } x \text{ relop } y \text{ goto } L$ , 关系运算符 $\text{relop}(<, =, >= \text{等等})$

(6)复制语句  $x = y$

(7)带下标的复制语句:

$x = y[i]$

$x[i] = y$

(8)过程调用语句  $\text{param } x$  和  $\text{call } p, n$ 。过程调用语句 $p(x_1, x_2, \dots, x_n)$ 产生如下三地址代码:

$\text{param } x_1$

...

$\text{param } x_n$

$\text{call } p, n$



# 给三地址指令指定标号的两种方法

---

do  $i = i + 1$ ; while ( $a[i] < v$ );

符号标号

```
L: t1 = i + 1  
  i = t1  
  t2 = i * 8  
  t3 = a[t2]  
  if t3 < v goto L
```

位置号

```
100: t1 = i + 1  
101: i = t1  
102: t2 = i * 8  
103: t3 = a[t2]  
104: if t3 < v goto 100
```

# 三地址码的实现

---

- 三地址代码的具体实现

- 1. 四元式  $op, arg1, arg2, result$
- 2. 三元式  $op, arg1, arg2$
- 3. 间接三元式 间接码表+三元式表



# 四元式

四元式有4个字段：  $op$ ,  $arg_1$ ,  $arg_2$ , 和  $result$

- $arg_1$ ,  $arg_2$ , 和  $result$  中为指向符号表项目入口的指针。
- $op$  包含一个运算符的内部编码

$t_1 = \text{minus } c$

$t_2 = b * t_1$

$t_3 = \text{minus } c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

$a = t_5$

	op	arg1	arg2	result
0	minus	c		$t_1$
1	*	b	$t_1$	$t_2$
2	minus	c		$t_3$
3	*	b	$t_3$	$t_4$
4	+	$t_2$	$t_4$	$t_5$
5	=	$t_5$		a



# 三元式

三元式可以避免引入临时变量

- 使用获得变量值的**位置**来引用前面的运算结果

$t_1 = \text{minus } c$

$t_2 = b * t_1$

$t_3 = \text{minus } c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

$a = t_5$

	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)



# 间接三元式

包含一个指向三元式的**指针列表**，而不是列出三元式序列本身

instructio	
n	
35	
36	(0)
37	(1)
38	(2)
39	(3)
40	(4)
	(5)

	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)



# 三地址码的实现

---

## 三地址代码三种实现形式的比较

代码优化时，经常因调整计算次序而要移动三地址语句

- 四元式调整顺序方便，但引入的临时变量多，需存储空间大
- 三元式需存储空间最小，但调整顺序不便
- 间接三元式优化方便，在有公共子表达式时，需存储空间比四元式小
- 中间代码优化处理时，四元式比三元式方便的多，间接三元式与四元式同样方便





# 学习内容

---

- 6.1 类型检查
- 6.2 中间表示
- **6.3 声明语句**
- 6.4 赋值语句
- 6.5 控制流
- 6.6 回填
- 6.7 switch语句
- 6.8 过程的中间代码



# 声明语句

为每个声明的名，在符号表中会加入相应信息：**名**、**类型**、**位移**等等。

➤ **类型**:

$$D \rightarrow T \text{ id}; D \mid \varepsilon$$
$$T \rightarrow B \ C \mid \text{record } \{ D \}$$
$$B \rightarrow \text{int} \mid \text{float}$$
$$C \rightarrow \varepsilon \mid [ \text{num} ] \ C$$

$D$  生成一系列声明;

$T$  生成基本类型、数组类型或记录类型;

$B$  生成基本类型 `int` 或 `float` 之一;

$C$  产生零个或多个整数，多个整数用方括号括起来;

数组类型:  $B + C$

记录类型: { 各个字段的声明序列构成 }



# 声明语句

---

## ➤ 位移

位移指出相对地址:

- 全局数据的位移是指在静态数据区的位置
- 局部数据的位移是指在局部过程的活动记录的局部数据区的位置。

# 局部变量名的存储布局

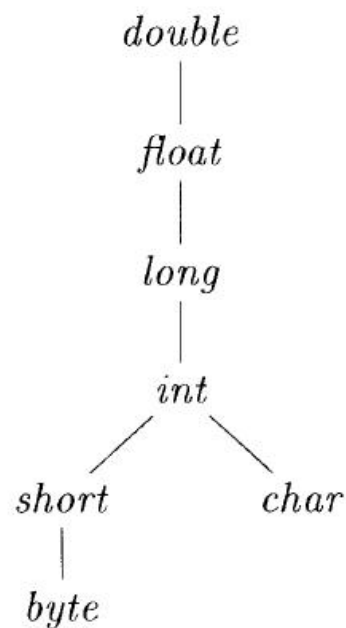
根据变量类型，可知变量在运行时刻需要的内存数量

符号表中：名字类型 + 相对地址（根据名字）

变长数据 or 动态数组：保留固定大小的存储区域

字节：最小的内存单位

类型的宽度(width)：类型的对象所需的存储单元的数量。



a) 拓宽类型转换



# 局部变量名的存储布局

---

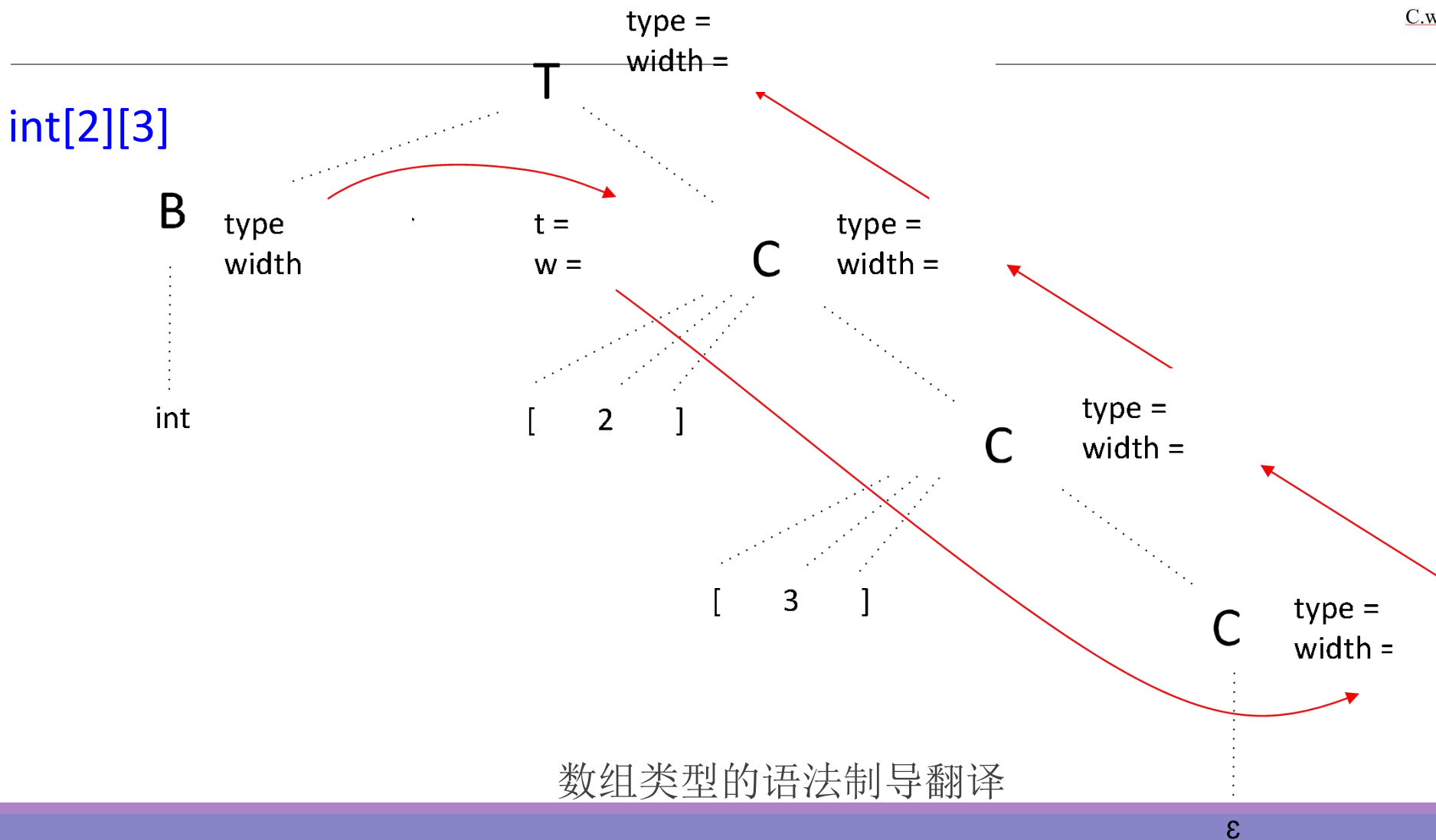
$T \rightarrow B$	$\{ t = B.type; w = B.width; \}$
$C$	$\{ T.type = C.type; T.width = C.width; \}$
$B \rightarrow \text{int}$	$\{ B.type = \text{integer}; B.width = 4; \}$
$B \rightarrow \text{float}$	$\{ B.type = \text{float}; B.width = 8; \}$
$C \rightarrow \varepsilon$	$\{ C.type = t; C.width = w; \}$
$C \rightarrow [\text{num}] C_1$	$\{ C.type = \text{array}(\text{num.value}, C_1.type);$ $C.width = \text{num.value} \times C_1.width; \}$

计算类型及其宽度



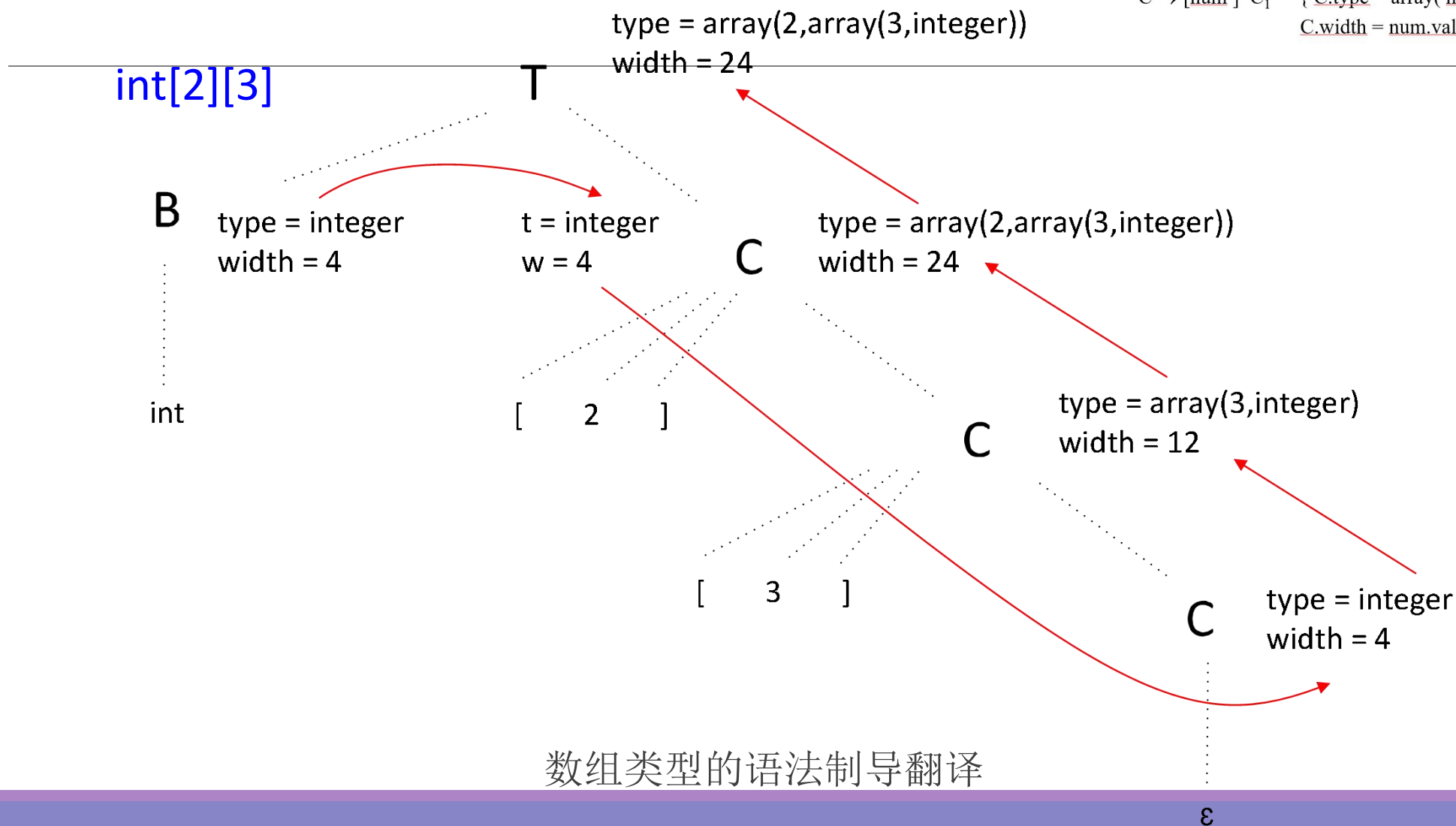
# 局部变量名的存储布局

$T \rightarrow B$	$\{ t = \underline{B.type}; w = \underline{B.width}; \}$
$C$	$\{ \underline{T.type} = \underline{C.type}; T.width = \underline{C.width}; \}$
$B \rightarrow \text{int}$	$\{ \underline{B.type} = \text{integer}; \underline{B.width} = 4; \}$
$B \rightarrow \text{float}$	$\{ \underline{B.type} = \text{float}; \underline{B.width} = 8; \}$
$C \rightarrow \varepsilon$	$\{ \underline{C.type} = t; \underline{C.width} = w; \}$
$C \rightarrow [\text{num}] C_1$	$\{ \underline{C.type} = \text{array}(\text{num.vale}, C_1.type);$ $\underline{C.width} = \text{num.value} \times C_1.width; \}$



# 局部变量名的存储布局

$T \rightarrow B$	{ $t = B.type$ ; $w = B.width$ ;} $C$
$B \rightarrow \text{int}$	{ $B.type = \text{integer}$ ; $B.width = 4$ ;} $B \rightarrow \text{float}$
$B \rightarrow \text{float}$	{ $B.type = \text{float}$ ; $B.width = 8$ ;} $C \rightarrow \varepsilon$
$C \rightarrow \varepsilon$	{ $C.type = t$ ; $C.width = w$ ;} $C \rightarrow [\text{num}] C_1$
$C \rightarrow [\text{num}] C_1$	{ $C.type = \text{array}(\text{num.vale}, C_1.type)$ ; $C.width = \text{num.value} \times C_1.width$ ;} $C_1$



# 声明的序列

---

全局变量`offset`: 跟踪下一个可用的相对地址  
计算被声明变量的相对地址

$P \rightarrow$	$\{ \textit{offset} = 0; \}$
$D$	
$D \rightarrow T \textit{id} ;$	$\{ \textit{top.put(id.lexeme, T.type, offset);}$ $\textit{offset} = \textit{offset} + T.width \}$
$D_1$	
$D \rightarrow \varepsilon$	





# 记录和类中的字段

---

记录内有字段名字  $x$ ，与记录外的对该名字的使用不冲突

```
float x;
```

```
record { float x; float y; } p
```

```
record{ int tag; float x; float y; } q
```

- 记录中各字段的名称不同
- 相对地址：相对于该记录的数据区字段而言



# 记录和类中的字段

---

- 记录类型的产生式:

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

- 这个记录类型中的字段由D生成的声明序列描述

- 处理记录中的字段名

$T \rightarrow \text{record } \{$	$\{ \text{Env.push ( top ); top = new Env();}$
	$\text{Stack.push ( offset ); offset = 0; } \}$
$D \}$	$\{ T.\text{type} = \text{record ( top ); T.width} = \text{offset};$
	$\text{top} = \text{Env.pop()}; \text{offset} = \text{Stack.pop()}; \}$



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

$D \rightarrow T \text{ id}; \{ \text{top.put}(\text{id.lexeme}, T.\text{type}, \text{offset});$

$\text{offset} = \text{offset} + T.\text{width}; \} D_1$

$D \rightarrow \varepsilon$

$T \rightarrow B \{ t = B.\text{type}; w = B.\text{width}; \}$

$C \{ T.\text{type} = C.\text{type}; T.\text{width} = C.\text{width}; \}$

$B \rightarrow \text{int} \{ B.\text{type} = \text{integer}; B.\text{width} = 4; \}$

$B \rightarrow \text{float} \{ B.\text{type} = \text{float}; B.\text{width} = 8; \}$

$C \rightarrow \varepsilon \{ C.\text{type} = t; C.\text{width} = w; \}$

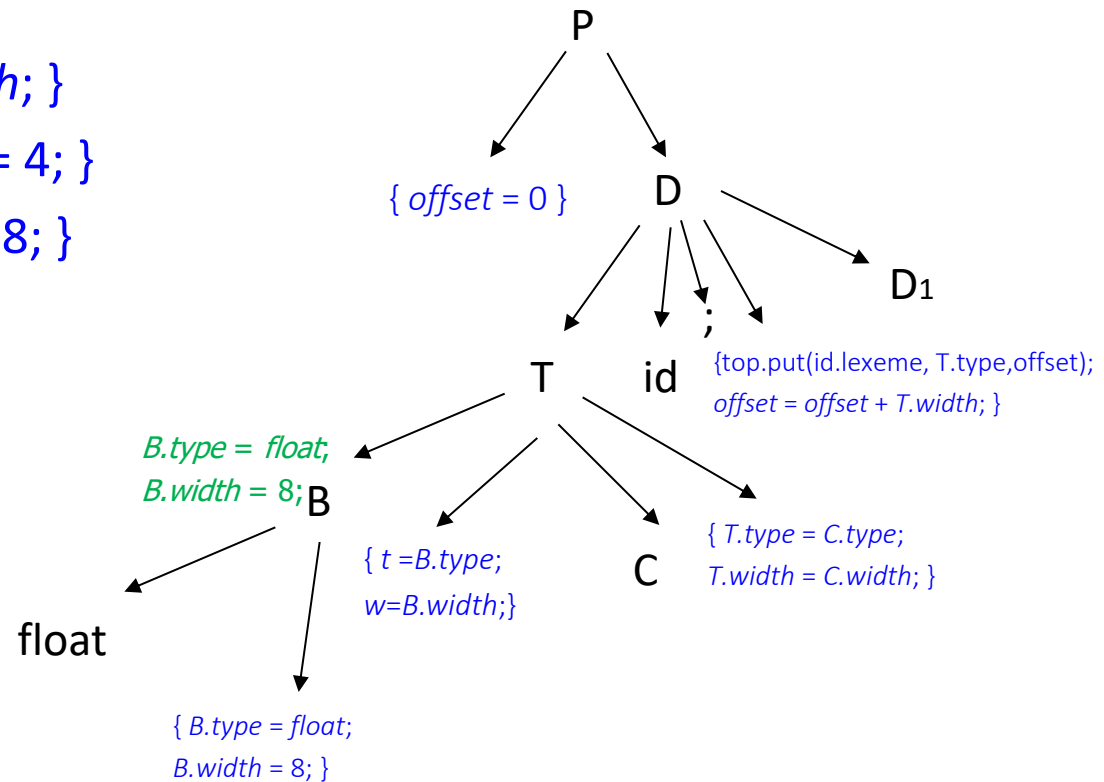
$C \rightarrow [\text{num}] C_1$

$\{ C.\text{type} = \text{array}(\text{num.val}, C_1.\text{type});$

$C.\text{width} = \text{num.val} \times C_1.\text{width}; \}$

$\text{offset} = 0$

float x; int i;



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

$D \rightarrow T \text{id}; \{ \text{top.put}(\text{id.lexeme}, T.\text{type}, \text{offset});$

$\text{offset} = \text{offset} + T.\text{width}; \} D_1$

$D \rightarrow \varepsilon$

$T \rightarrow B \{ t = B.\text{type}; w = B.\text{width}; \}$

$C \{ T.\text{type} = C.\text{type}; T.\text{width} = C.\text{width}; \}$

$B \rightarrow \text{int} \{ B.\text{type} = \text{integer}; B.\text{width} = 4; \}$

$B \rightarrow \text{float} \{ B.\text{type} = \text{float}; B.\text{width} = 8; \}$

$C \rightarrow \varepsilon \{ C.\text{type} = t; C.\text{width} = w; \}$

$C \rightarrow [\text{num}] C_1$

$\{ C.\text{type} = \text{array}(\text{num.val}, C_1.\text{type});$

$C.\text{width} = \text{num.val} \times C_1.\text{width}; \}$

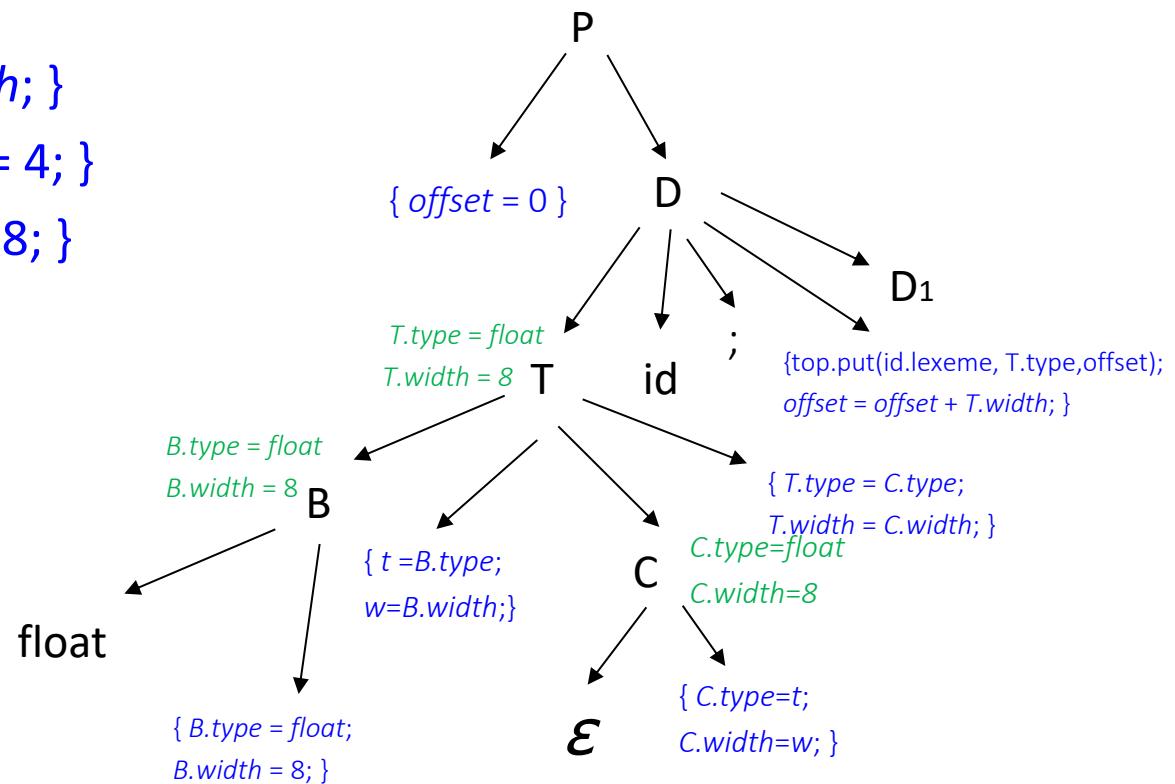
$\text{offset} = 0$

$t = \text{float}$

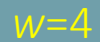
$w = 8$

float x; int i;

$\text{top.put}(x, \text{float}, 0);$



```
float x; int i;
```



# 学习内容

---

- 6.1 类型检查
- 6.2 中间表示
- 6.3 声明语句
- **6.4 赋值语句**
- 6.5 控制流
- 6.6 回填
- 6.7 switch语句
- 6.8 过程的中间代码



# 赋值语句

产生式	语义规则
$S \rightarrow id = E ;$	$S.code = E.code \parallel \text{gen}(\text{top.get}(id.lexeme) \text{ '=' } E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \text{new Temp}()$ $E.code = E_1.code \parallel E_2.code \parallel \text{gen}(E.addr \text{ '=' } E_1.addr + E_2.addr)$
$\quad   - E_1$	$E.addr = \text{new Temp}()$ $E.code = E_1.code \parallel \text{gen}(E.addr \text{ '=' 'minus' } E_1.addr)$
$\quad   (E_1)$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\quad   id$	$E.addr = \text{top.get}(id.lexeme)$ $E.code = \text{' '}$

表达式的三地址码

例：

$a = b + -c ;$



$t_1 = \text{minus } c$   
 $t_2 = b + t_1$   
 $a = t_2$



# 赋值语句

产生式	语义规则
$S \rightarrow id = E ;$	$S.code = E.code \parallel \text{gen}(\text{top.get}(id.lexeme) \text{ '=' } E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \text{new Temp}()$ $E.code = E_1.code \parallel E_2.code \parallel \text{gen}(E.addr \text{ '=' } E_1.addr + E_2.addr)$
$\quad   - E_1$	$E.addr = \text{new Temp}()$ $E.code = E_1.code \parallel \text{gen}(E.addr \text{ '=' } \text{'minus'} E_1.addr)$
$\quad   (E_1)$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\quad   id$	$E.addr = \text{top.get}(id.lexeme)$ $E.code = \text{' '}$

表达式的三地址码

例：

$a = b + -c ;$



$t_1 = \text{minus } c$   
 $t_2 = b + t_1$   
 $a = t_2$





# 增量翻译

---

`code`属性：很长的字符串。

去掉`code`属性，利用`gen`连续生成一个指令序列



# 增量翻译

增量生成表达式的三地址码

产生式	语义规则
$S \rightarrow id = E ;$	$S.code = E.code \parallel \text{gen}(\text{top.get}(id.lexeme) \text{ '=' } E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \text{new Temp}()$ $E.code = E_1.code \parallel E_2.code \parallel \text{gen}(E.addr \text{ '=' } E_1.addr + E_2.addr)$
$\mid - E_1$	$E.addr = \text{new Temp}()$ $E.code = E_1.code \parallel \text{gen}(E.addr \text{ '=' 'minus' } E_1.addr)$
$\mid (E_1)$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid id$	$E.addr = \text{top.get}(id.lexeme)$ $E.code = \text{' '}$

产生式	语义规则
$S \rightarrow id = E ;$	{ $\text{gen}(\text{top.get}(id.lexeme) \text{ '=' } E.addr)$ ; }
$E \rightarrow E_1 + E_2$	{ $E.addr = \text{new Temp}()$ $\text{gen}(E.addr \text{ '=' } E_1.addr + E_2.addr)$ ; }
$\mid - E_1$	{ $E.addr = \text{new Temp}()$ ; $\text{gen}(E.addr \text{ '=' 'minus' } E_1.addr)$ ; }
$\mid (E_1)$	{ $E.addr = E_1.addr$ ; }
$\mid id$	{ $E.addr = \text{top.get}(id.lexeme)$ ; }



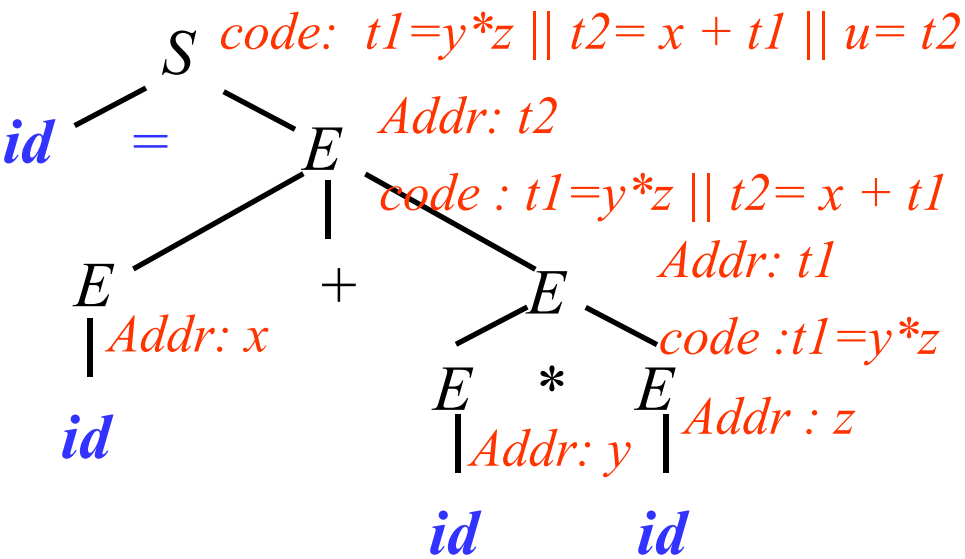
# 例：表达式 $u = x + y * z$ 翻译成三地址语句序列

$$t_1 = y * z$$

$$t_2 = x + t_1$$

$$u = t_2$$

产生式	语义规则
$S \rightarrow id = E ;$	$S.code = E.code \parallel \text{gen}(\text{top.get}(\text{id.lexeme}) \text{'=' } E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \text{new Temp}()$ $E.code = E_1.code \parallel E_2.code \parallel \text{gen}(E.addr \text{'=' } E_1.addr + E_2.addr)$
- $E_1$	$E.addr = \text{new Temp}()$ $E.code = E_1.code \parallel \text{gen}(E.addr \text{'=' 'minus' } E_1.addr)$
$(E_1)$	$E.addr = E_1.addr$ $E.code = E_1.code$
id	$E.addr = \text{top.get}(\text{id.lexeme})$ $E.code = \text{' '}$



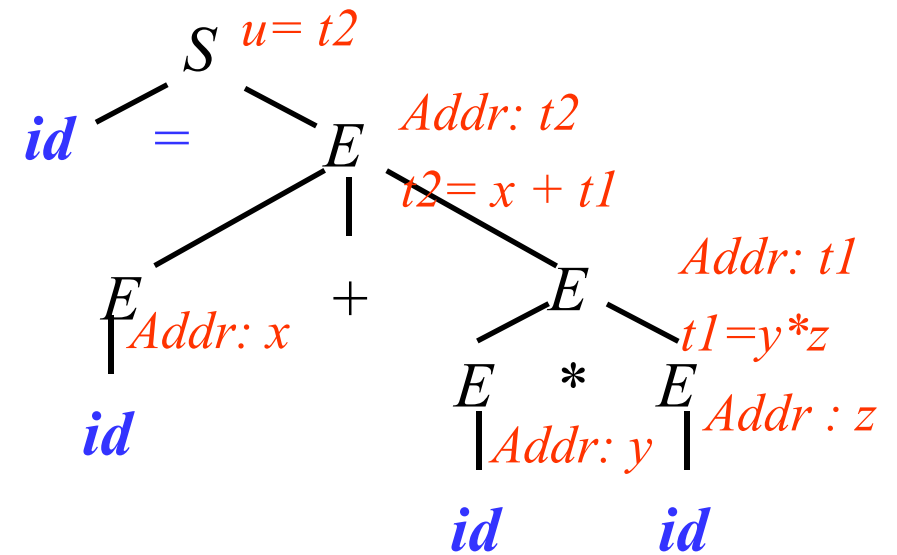
# 例：表达式 $u = x + y * z$ 翻译成三地址语句序列

$t_1 = y * z$

$t_2 = x + t_1$

$u = t_2$

产生式	语义规则
$S \rightarrow id = E ;$	{ <u>gen(top.get(id.lexeme) '=' E.addr);</u> }
$E \rightarrow E_1 + E_2$	{ <u>E.addr = new Temp()</u> <u>gen(E.addr '=' E<sub>1</sub>.addr + E<sub>2</sub>.addr);</u> }
- E <sub>1</sub>	{ <u>E.addr = new Temp();</u> <u>gen(E.addr '=' 'minus' E<sub>1</sub>.addr);</u> }
(E <sub>1</sub> )	{ <u>E.addr = E<sub>1</sub>.addr;</u> }
id	{ <u>E.addr = top.get(id.lexeme);</u> }



# 数组元素的寻址

1st row	A[1, 1]	1st column	A[1, 1]
	A[1, 2]		A[2, 1]
	A[1, 3]		A[1, 2]
2nd row	A[2, 1]	2nd column	A[2, 2]
	A[2, 2]		A[1, 3]
	A[2, 3]		A[2, 3]

数组存储在一块连续存储空间中。

一维数组A，若元素宽度为 $w$ ，则A中第 $i$ 个元素始于：

$$base + i \times w$$

二维数组可采用：行优先（*row-major*）或列优先（*column-major*）方式存储。设第 $j$ 维上的数组元素个数为 $n_j$ ，在行优先存储时，一行的宽度是 $w_1$ ，同一行中每个元素的宽度是 $w_2$ ， $A[i_1][i_2]$ 存于：

$$base + i_1 \times w_1 + i_2 \times w_2$$

k维数组元素  $A[i_1] \dots [i_k]$ ：

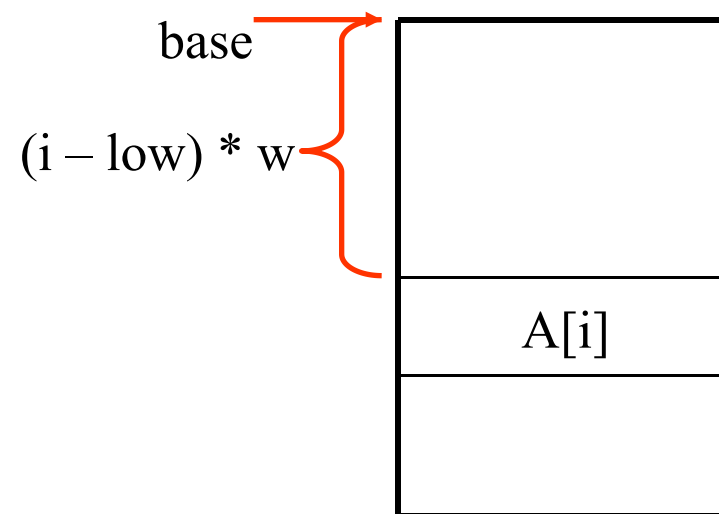
$$base + i_1 \times w_1 + \dots + i_k \times w_k$$



# 数组元素的寻址

## 一维数组

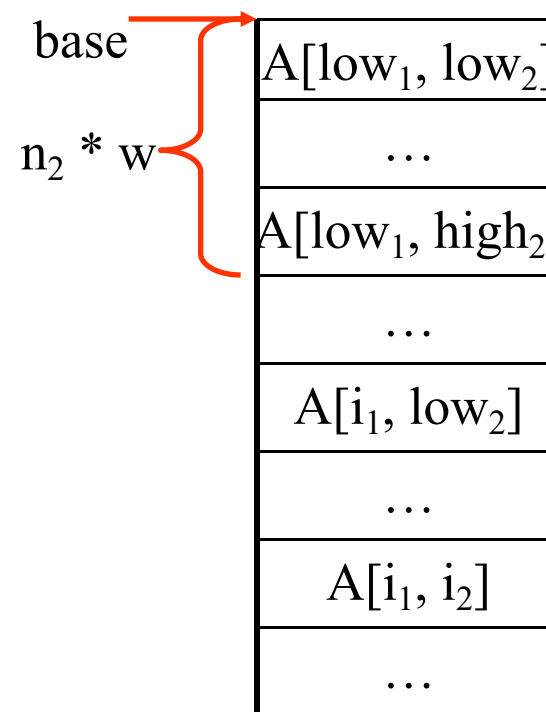
- `type A[low..high];`
- 计算 $A[i]$ 的地址
  - $A$ 的起始地址—— $\text{base}$
  - 数组元素大小—— $w$
  - $A[i]$ 与 $A$ 起始位置的“距离”—— $(i - \text{low}) * w$
  - 最终结果:  $\text{base} + (i - \text{low}) * w$   
    ➔  $i * w + (\text{base} - \text{low} * w)$
  - $(\text{base} - \text{low} * w)$ 为常量, 可在编译时计算



# 数组元素的寻址

## 二维数组:

- type  $A[\text{low}_1..\text{high}_1, \text{low}_2..\text{high}_2]$
- 计算  $A[i_1, i_2]$  地址
  - 数组的数组，两次利用一维数组计算方法
  - 行:  $n_2 = \text{high}_2 - \text{low}_2 + 1$  个元素的一维数组  $\rightarrow$  元素  
二维数组:  $n_1 = \text{high}_1 - \text{low}_1 + 1$  个“行”的一维数组
  - $i_1$  行的位置  $(i_1 - \text{low}_1) * n_2$
  - $A[i_1, i_2]$  距行  $i_1$  开始位置的距离:  $i_2 - \text{low}_2$
  - 最终结果  $\text{base} + ((i_1 - \text{low}_1) * n_2 + i_2 - \text{low}_2) * w$   
 $\rightarrow ((i_1 * n_2) + i_2) * w + (\text{base} - ((\text{low}_1 * n_2) + \text{low}_2) * w)$
  - $(\text{base} - ((\text{low}_1 * n_2) + \text{low}_2) * w)$  为常量



# 数组元素的寻址

---

## 扩展到多维情况

- type  $A[\text{low}_1..\text{high}_1, \dots, \text{low}_k..\text{high}_k]$
- 计算  $A[i_1, i_2, \dots, i_k]$  的地址
- 最终结果
  - $((\dots((i_1 n_2 + i_2) n_3 + i_3) \dots) n_k + i_k) * w +$   
 $\text{base} - ((\dots((\text{low}_1 n_2 + \text{low}_2) n_3 + \text{low}_3) \dots) n_k + \text{low}_k) * w$



# 数组引用的翻译

---

数组引用生成代码：地址计算 + 文法

令 $V_N$  L生成一个数组名 + 下标表达式的序列：

$L[E] \mid \mathbf{id}[E]$

L有三个综合属性：

*L.addr*: 临时变量，用于存放数组下标变量的偏移量

*L.array*: 数组的符号表入口地址，(*L.array.base*为数组的基地址)

*L.type*: L生成的子数组的类型

数组引用的地址：*L.array.base*[*L.addr*]

符号表入口地址(相当*base*)。



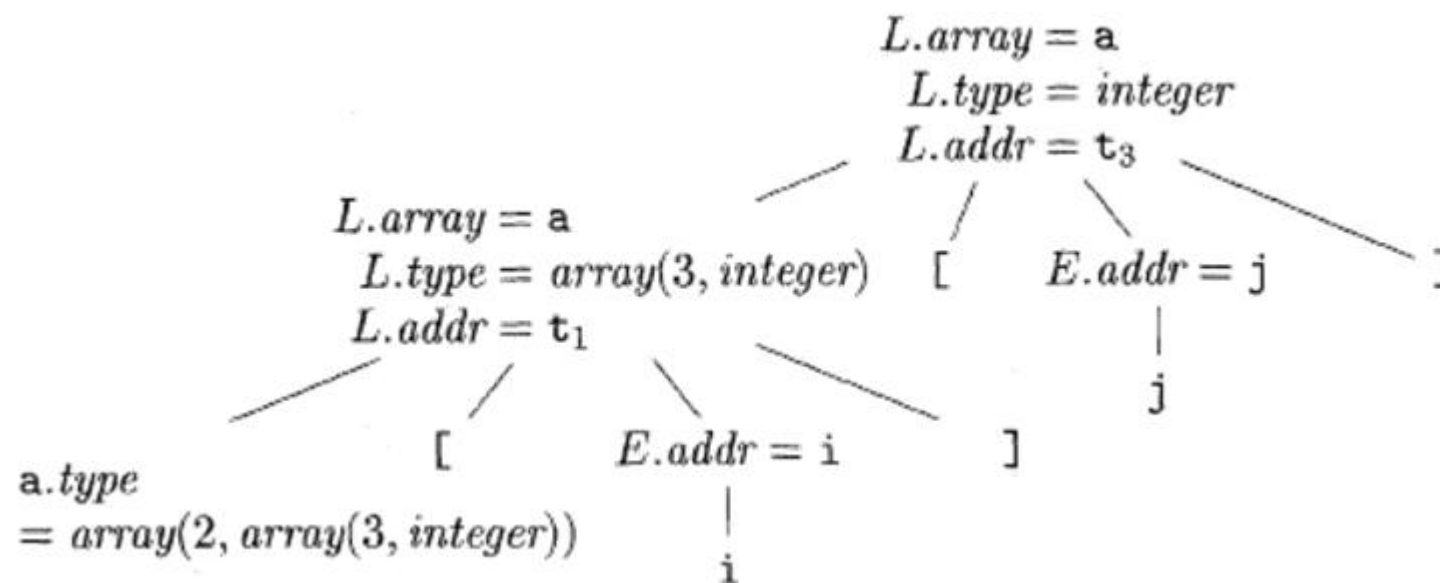
# 数组引用的翻译

$L[E] \mid \mathbf{id}[E]$  的语义动作

- $L \rightarrow \mathbf{id}[E]$   
 $L.array = \text{top.get}(\text{id.lexeme});$   
 $L.type = L.array.type.elem;$   
 $L.addr = \text{new Temp}();$   
 $\text{gen}(L.addr \text{ '=' } E.addr \text{ '*' } L.type.width) ;$
- $L \rightarrow L_1[E]$   
 $L.array = L_1.array;$   
 $L.type = L_1.type.elem;$   
 $t = \text{new Temp}();$   
 $L.addr = \text{new Temp}();$   
 $\text{gen}(t \text{ '=' } E.addr \text{ '*' } L.type.width);$   
 $\text{gen}(L.addr \text{ '=' } L_1.addr \text{ '+' } t);$



例:  $a[i][j]$



$$t_1 = i * 12$$

$$t_2 = j * 4$$

$$t_3 = t_1 + t_2$$

$$t_4 = a[t_3]$$

令  $a$  表示  $2 \times 3$  整数数组,  $i, j$  都是宽度为4的整数

以上给出了  $a[i][j]$  的注释分析树

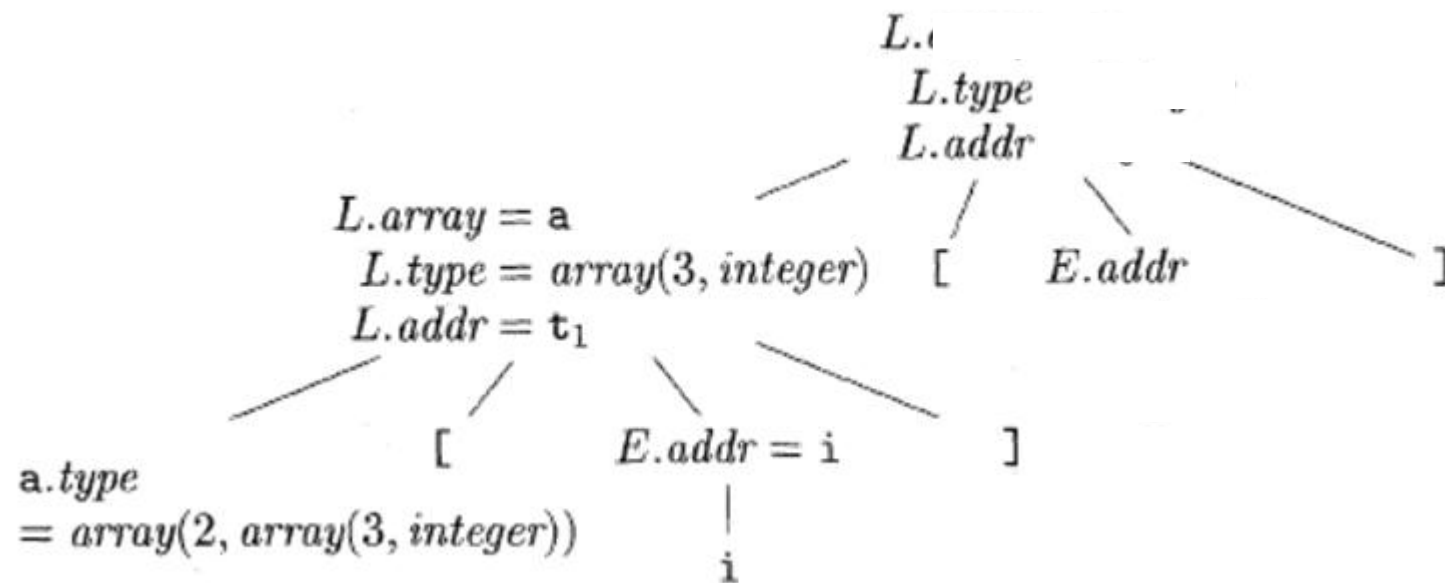


$L \rightarrow id[ E ]$

```
L.array = top.get(id.lexeme);  
L.type = L.array.type.elem;  
L.addr = new Temp();  
gen(L.addr '=' E.addr '*' L.type.width) ;
```

例:  $a[i][j]$

$$t_1 = i * 12$$



$L \rightarrow L_1[E]$

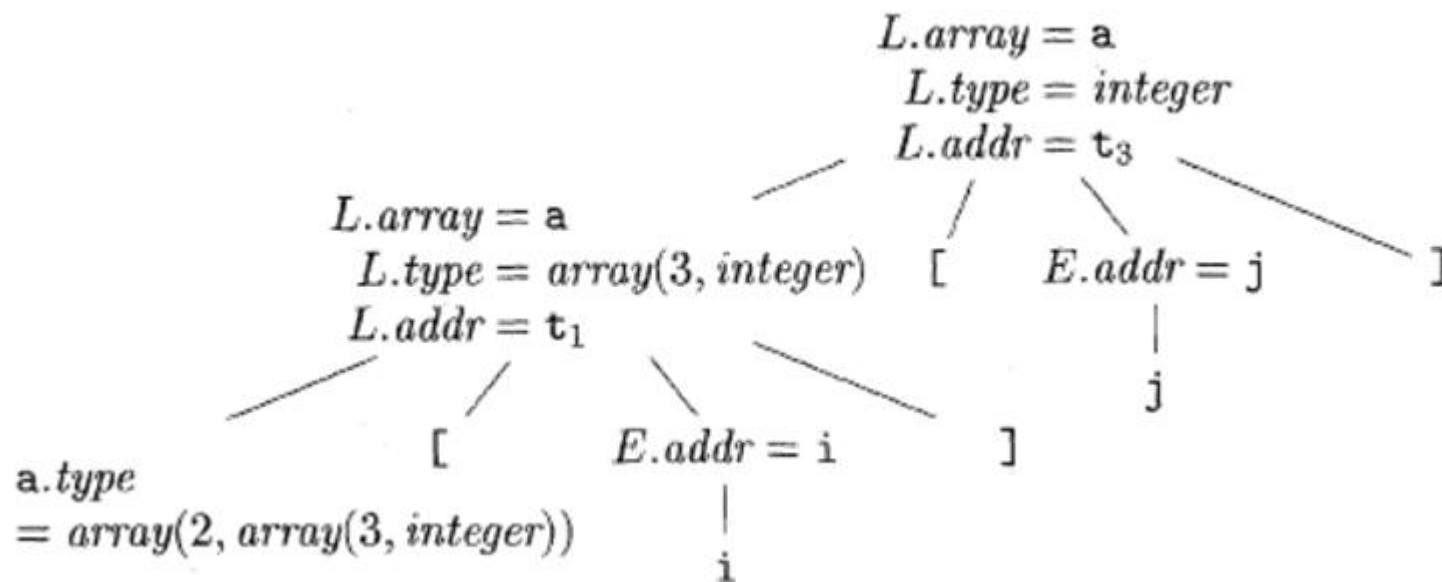
```
L.array = L1.array;  
L.type = L1.type.elem;  
t = new Temp();  
L.addr = new Temp();  
gen(t '=' E.addr '*' L.type.width);  
gen(L.addr '=' L1.addr '+' t);
```

例:  $a[i][j]$

$t_1 = i * 12$

$t_2 = j * 4$

$t_3 = t_1 + t_2$



# 学习内容

---

- 6.1 类型检查
- 6.2 中间表示
- 6.3 声明语句
- 6.4 赋值语句
- **6.5 控制流**
- 6.6 回填
- 6.7 switch语句
- 6.8 过程的中间代码



# 控制流

---

if-else， while语句的翻译常与布尔表达式的翻译结合在一起  
布尔表达式有两个基本目的

- 计算逻辑值
- 改变控制流， 在控制流语句中用作条件表达式

# 布尔表达式

---

## 布尔表达式文法

$$B \rightarrow B \parallel B \mid B \&\& B \mid !B \mid (B) \mid E \text{ rel } E \mid \text{true} \mid \text{false}$$

- 属性 **rel.op**代表: <, <=, =, !=, >, >=
- **||, &&** 是左结合的;
- 优先级 **|| < && < !**



# 数值表示

---

- $x < 100 \parallel x > 200 \ \&\& \ x \neq y$

$x < 100$

if  $x < 100$  then 1 else 0

---

if  $x < 100$  goto L1

t1=0

goto L2

L1: t1 = 1

L2: if  $x > 200$  goto L3

t2=0

Goto L4

L3: t2=1

L4: if  $x \neq y$  goto

t3=0

goto L6

L5: t3=1

L6: t4=t2 and t3

t5=t1 or t4

# 短路代码(跳转代码)

---

- `if (x<100 || x > 200 && x!=y) x =0;`

```
    if x < 100 goto L2
    ifFalse x > 200 goto L1
    ifFalse x != y goto L1
L2:  x = 0
L1:
```

运算符不出现在代码中，布尔表达式的值通过代码序列中的位置来表示的。

# 控制流语句

---

控制流语句文法:

$$\begin{aligned} S \rightarrow & \textbf{if } (B) S_1 \\ & | \textbf{if } (B) S_1 \textbf{ else } S_2 \\ & | \textbf{while } (B) S_1 \end{aligned}$$

属性:

$B.code, S.code$

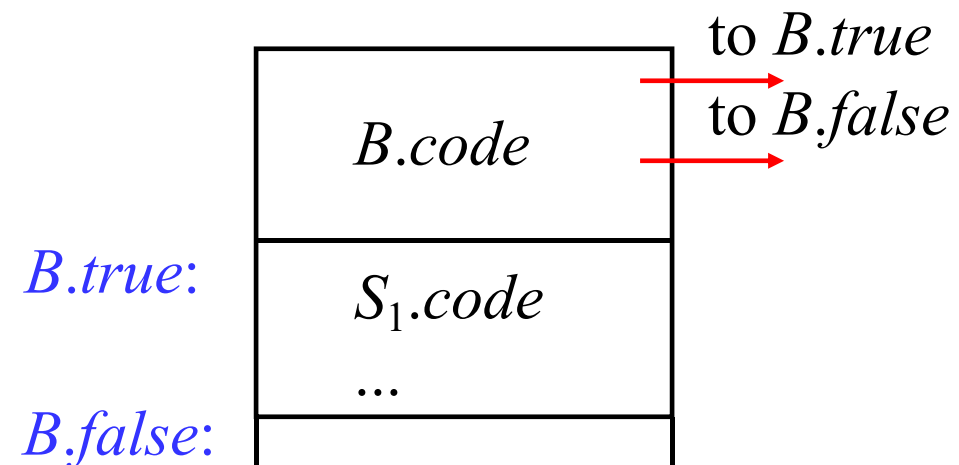
$B.true$ 和 $B.false$ : 表示 $B$ 为真和假时控制流转向的标号;

$S.next$ :紧跟在 $S$ 之后的指令的标号

# 控制流语句的代码结构

---

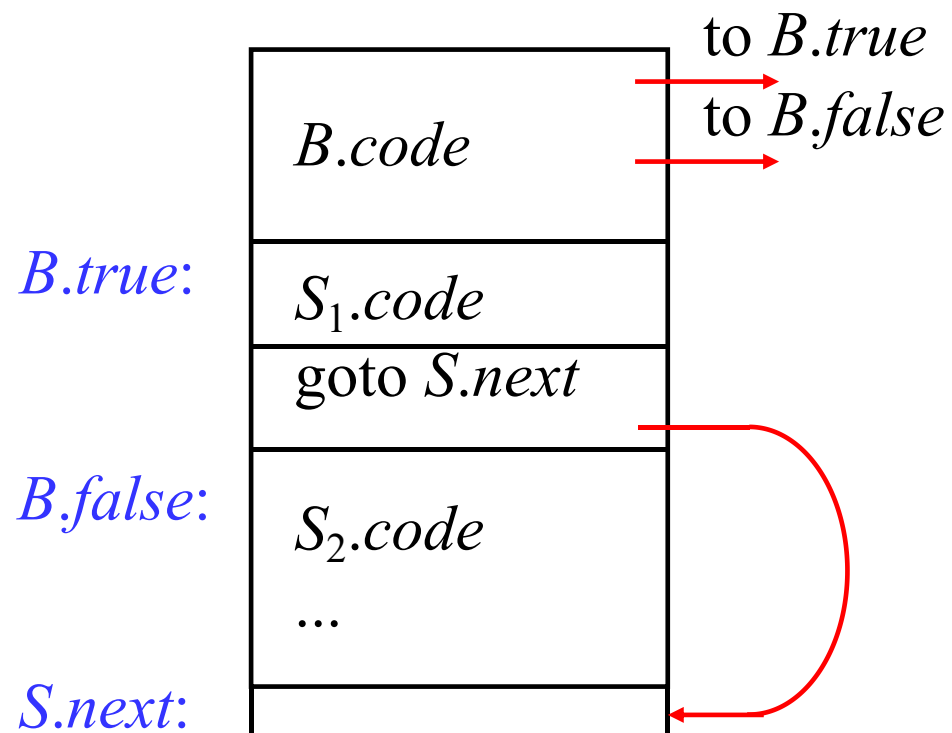
$S \rightarrow \mathbf{if} (B) S_1$



# 控制流语句的代码结构

---

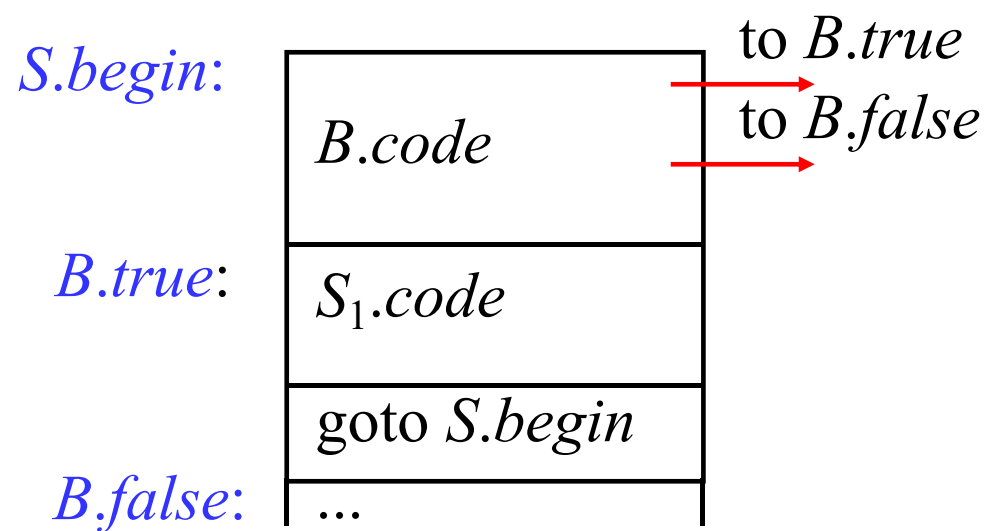
$S \rightarrow \text{if } (B) \ S_1 \ \text{else } S_2$



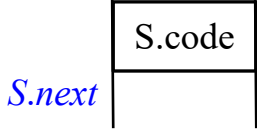
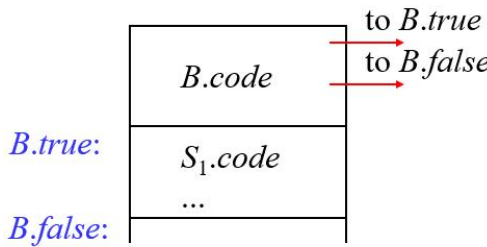
# 控制流语句的代码结构

---

$S \rightarrow \mathbf{while} (B) S_1$



# 控制流语句的语法制导定义

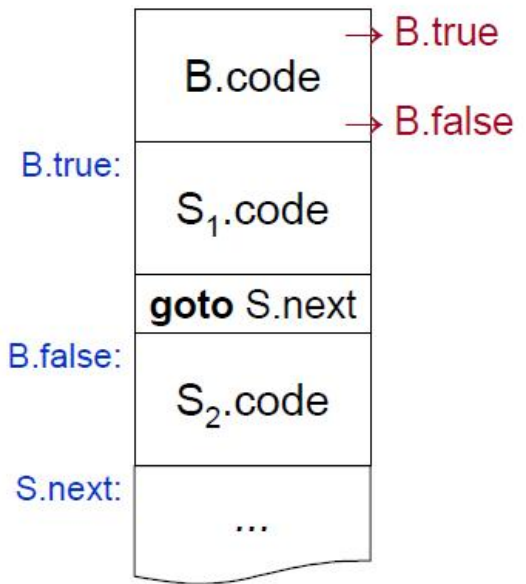
产生式	语义规则
$P \rightarrow S$ 	$S.next = newlabel()$ $P.code = S.code \parallel label(S.next)$
$S \rightarrow \text{assign}$	$S.code = \text{assign}.code$
$S \rightarrow \text{if } (B) S_1$ 	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \parallel label(B.true) \parallel S_1.code$

# 控制流语句的语法制导定义

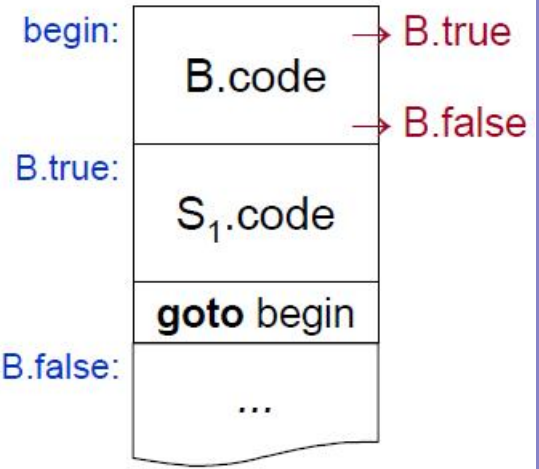
产生式	语义规则
$S \rightarrow \text{if } (B) S_1 \text{ else } S_2$ <pre>graph TD     B[B.code] -- B.true --&gt; S1[S1.code]     B -- B.false --&gt; Goto[goto S.next]     S1 --&gt; Goto     Goto --&gt; S2[S2.code]     S2 --&gt; Ellipsis[...]</pre>	



# 控制流语句的语法制导定义

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

# 控制流语句的语法制导定义

产生式	语义规则
$S \rightarrow \mathbf{while} (B) S_1$ 	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \parallel B.code$ $\parallel label(B.true) \parallel S_1.code$ $\parallel gen('goto' begin)$

# 控制流语句的语法制导定义

产生式	语义规则			
$S \rightarrow S_1;S_2$  <div><math>S_1.next:</math><table><tr><td><math>S_1.code</math></td></tr><tr><td><math>S_2.code</math></td></tr><tr><td>...</td></tr></table></div>	$S_1.code$	$S_2.code$	...	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code$ $\quad    label(S_1.next)    S_2.code$
$S_1.code$				
$S_2.code$				
...				

# 布尔表达式的控制流翻译

---

对于象Pascal这样的语言，在控制流语句中的布尔表达式的值仅仅用来控制语句的转向，并不产生副作用，因此没有必要计算出布尔表达式的值。

形如 $a < b$  翻译为:

**if**  $a < b$  **goto**  $B.true$

**goto**  $B.false$

# 布尔表达式的语法制导定义

---

产生式	语义规则
$B \rightarrow B_1 \parallel B_2$	$\begin{aligned} B_1.true &= B.true \\ B_1.false &= newlabel() \\ B_2.true &= B.true \\ B_2.false &= B.false \\ B.code &= B_1.code \parallel label(B_1.false) \parallel B_2.code \end{aligned}$

---

产生式	语义规则
$B \rightarrow B_1 \ \&\& \ B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \parallel label(B_1.true) \parallel B_2.code$

---

产生式	语义规则
$B \rightarrow ! B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow (B_1)$	$B_1.true = B.true$ $B_1.false = B.false$ $B.code = B_1.code$

---

产生式	语义规则
$B \rightarrow E_1 \textbf{rel} E_2$	$\begin{aligned} B.code &= E_1.code \parallel E_2.code \\ &\parallel \textit{gen}(\text{'if' } E_1.addr \textbf{rel.op} E_2.addr \text{'goto' } B.true) \\ &\parallel \textit{gen}(\text{'goto' } B.false) \end{aligned}$
$B \rightarrow \textbf{true}$	$B.code = \textit{gen}(\text{'goto' } B.true)$
$B \rightarrow \textbf{false}$	$B.code = \textit{gen}(\text{'goto' } B.false)$



# 例:控制流语句中的翻译。

---

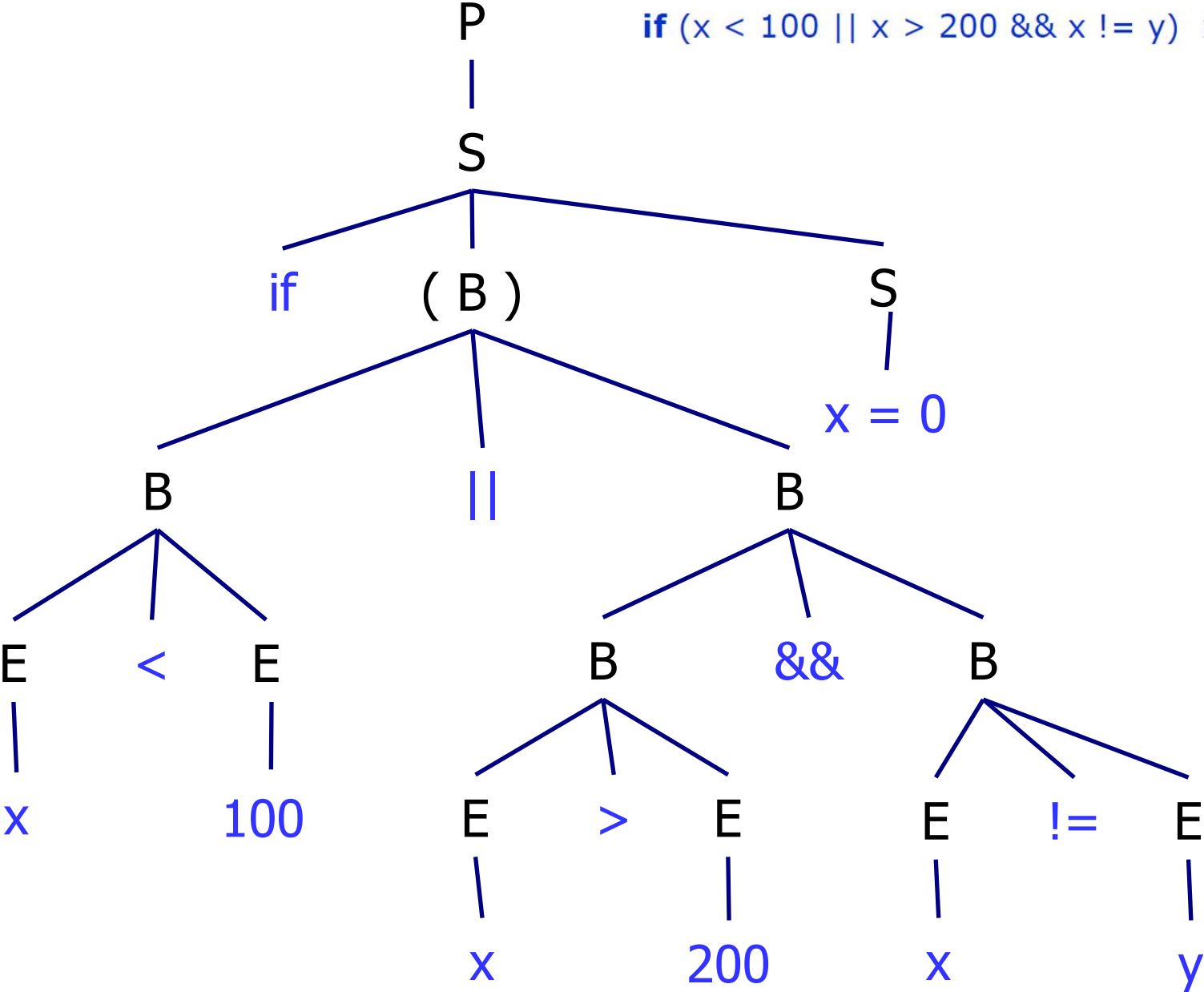
Source code

```
if (x < 100 || x > 200 && x != y) x = 0
```

Intermediate code

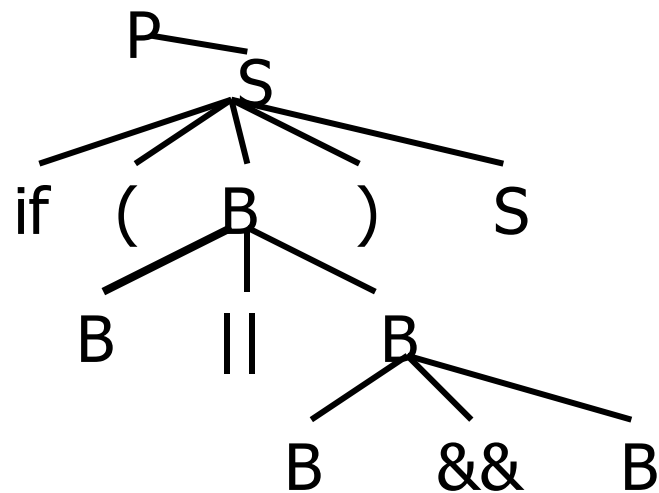
```
    if x < 100 goto L2  
    goto L3  
L3: if x > 200 goto L4  
    goto L1  
L4: if x != y goto L2  
    goto L1  
L2: x = 0  
L1: ...
```

```
if (x < 100 || x > 200 && x != y) x = 0
```



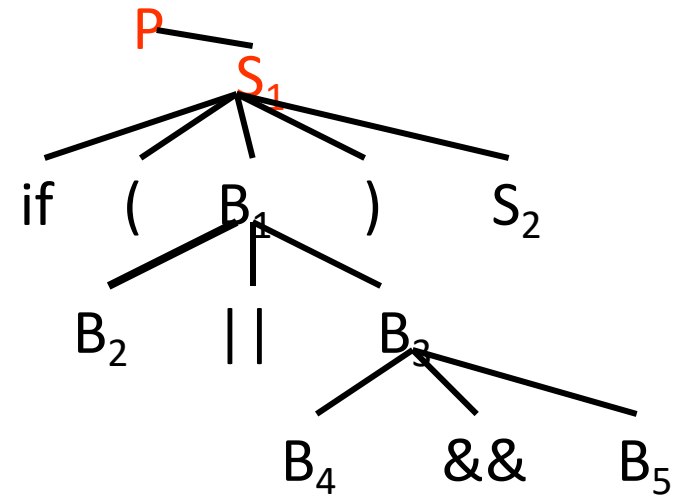
例:

**if(x<100 || x>200 && x!=y) x=0;**



if(x<100 || x>200 && x!=y) x=0;

$S_1.next = L_1;$



$P \rightarrow S$

$S.next = newlabel( )$

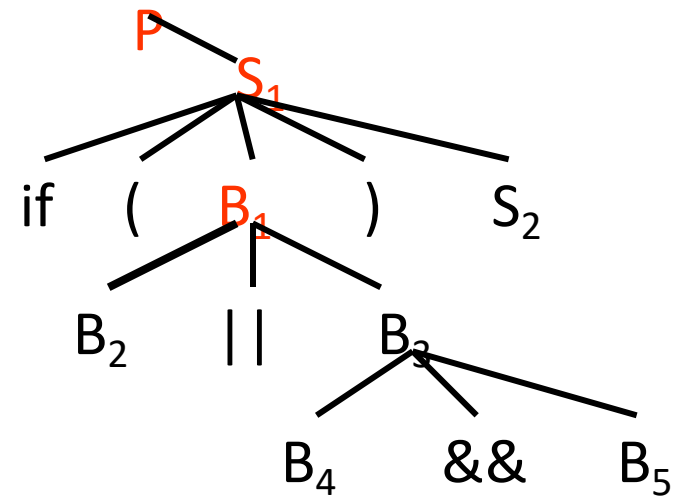
$P.code = S.code \parallel label(S.next)$

**if(x<100 || x>200 && x!=y) x=0;**

$S_1.next = L_1;$

$B_1.true = L_2;$

$B_1.false = L_1;$



$S \rightarrow \text{if } (B) S_1$

$B.true = newlabel()$

$B.false = S_1.next = S.next$

$S.code = B.code \parallel label(B.true) \parallel S_1.code$

if(x<100 || x>200 && x!=y) x=0;

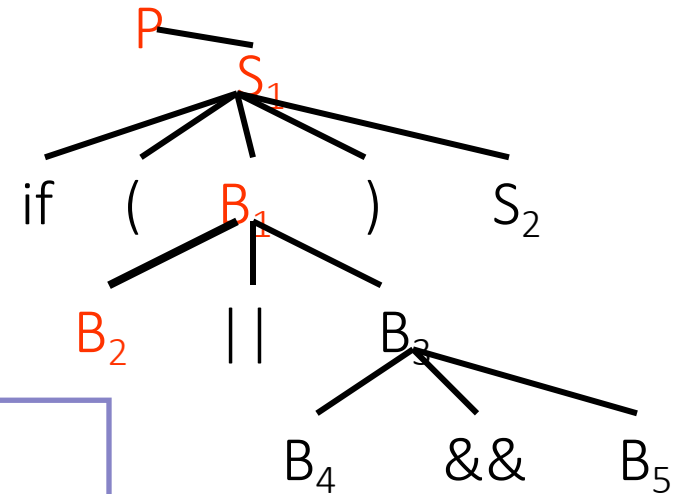
$S_1.next = L_1;$

$B_1.true = L_2;$

$B_1.false = L_1;$

$B_2.true = L_2;$

$B_2.false = L_3;$



$B \rightarrow B_1 || B_2$

$B_1.true = B.true$

$B_1.false = newlabel()$

$B_2.true = B.true$

$B_2.false = B.false$

$B.code = B_1.code || label(B_1.false) || B_2.code$

**if(x<100 || x>200 && x!=y) x=0;**

$S_1.next = L_1;$

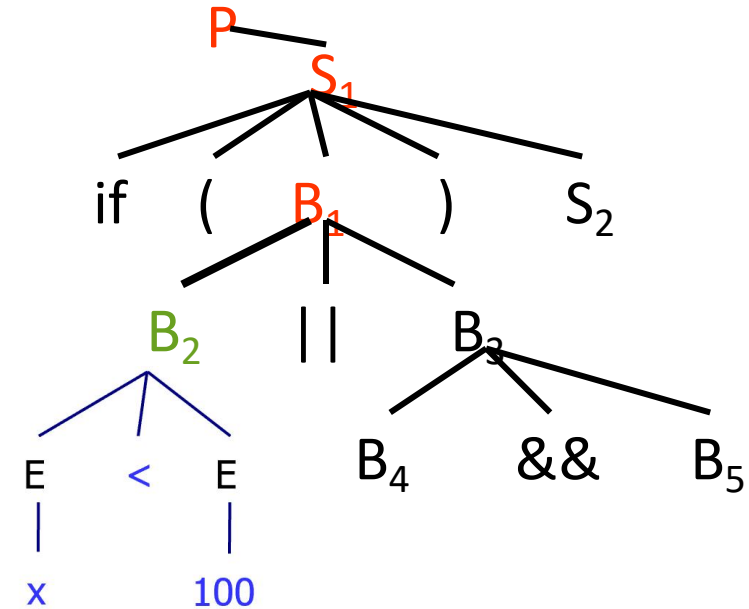
$B_1.true = L_2;$

$B_1.false = L_1;$

$B_2.true = L_2;$

$B_2.false = L_3;$

$B_2.code = \text{if}(x < 100) \text{ goto } L_2$   
 $\text{goto } L_3$



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

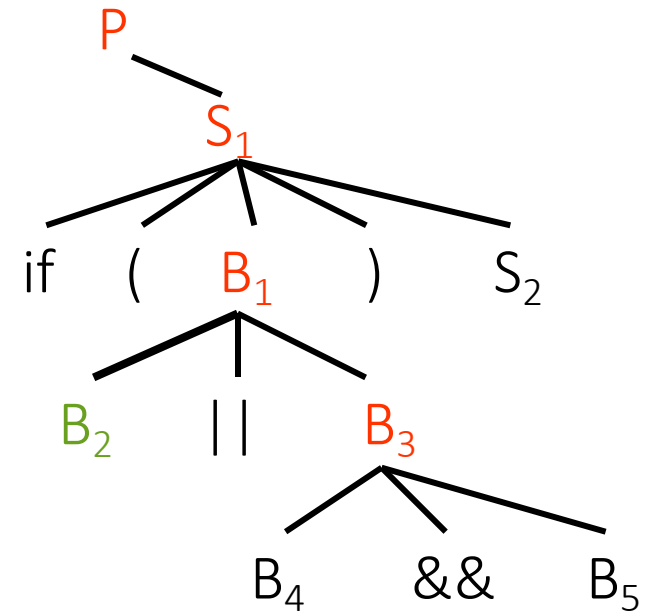
$B.code = E_1.code \ || \ E_2.code$

$\ || \ gen('if' \ E_1.addr \ rel.op \ E_2.addr \ 'goto' \ B.true) \ || \ gen('goto' \ B.false)$

**if(x<100 || x>200 && x!=y) x=0;**

$S_1.next = L_1;$   
 $B_1.true = L_2;$   
 $B_1.false = L_1;$   
 $B_2.true = L_2;$   
 $B_2.false = L_3;$   
 $B_3.true = L_2;$   
 $B_3.false = L_1;$

$B_2.code = \text{if}(x < 100) \text{ goto } L_2$   
 $\text{goto } L_3$



$B \rightarrow B_1 || B_2$

$B_1.true = B.true$

$B_1.false = \text{newlabel}()$

$B_2.true = B.true$

$B_2.false = B.false$

$B.code = B_1.code || \text{label}(B_1.false) || B_2.code$



if(x<100 || x>200 && x!=y) x=0;

$S_1.next = L_1;$

$B_1.true = L_2;$

$B_1.false = L_1;$

$B_2.true = L_2;$

$B_2.false = L_3;$

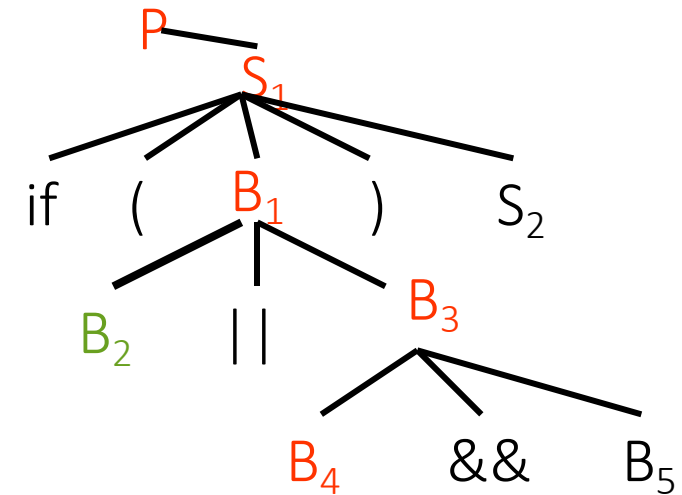
$B_3.true = L_2;$

$B_3.false = L_1;$

$B_4.true = L_4;$

$B_4.false = L_1;$

$B_2.code = \text{if}(x < 100) \text{ goto } L_2$   
goto L3



$B \rightarrow B_1 \&\& B_2$

$B_1.true = \text{newlabel}()$      $B_1.false = B.false$

$B_2.true = B.true$      $B_2.false = B.false$

$B.code = B_1.code \ || \ \text{label}(B_1.true) \ || \ B_2.code$

**if(x<100 || x>200 && x!=y) x=0;**

$S_1.next = L_1;$

$B_1.true = L_2; B_1.false = L_1;$

$B_2.true = L_2; B_2.false = L_3;$

$B_3.true = L_2; B_3.false = L_1;$

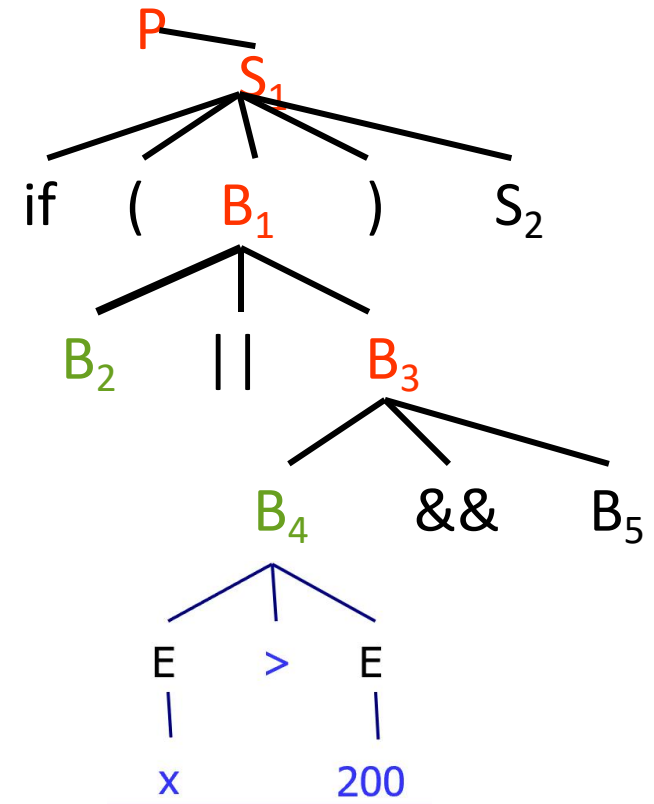
$B_4.true = L_4; B_4.false = L_1;$

$B_2.code = \text{if}(x < 100) \text{ goto } L_2$

$\text{goto } L_3$

$B_4.code = \text{if}(x > 200) \text{ goto } L_4$

$\text{goto } L_1$



$B \rightarrow E_1 \text{ rel } E_2 \quad B.code = E_1.code \ || \ E_2.code \ || \ gen('if' \ E_1.addr \ rel.op \ E_2.addr \ 'goto' \ B.true) \ || \ gen('goto' \ B.false)$

if(x<100 || x>200 && x!=y) x=0;

$S_1.next = L_1;$

$B_1.true = L_2;$

$B_1.false = L_1;$

$B_2.true = L_2;$

$B_2.false = L_3;$

$B_3.true = L_2;$

$B_3.false = L_1;$

$B_4.true = L_4;$

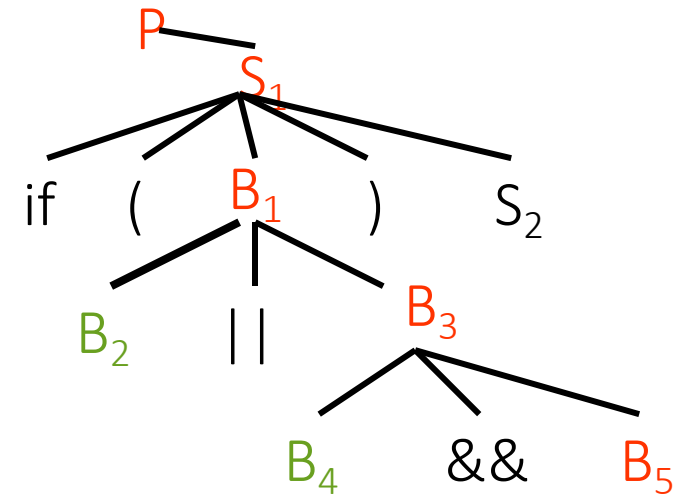
$B_4.false = L_1;$

$B_5.true = L_2;$

$B_5.false = L_1;$

$B_2.code = \text{if}(x < 100) \text{ goto } L_2$   
                   goto L3

$B_4.code = \text{if}(x > 200) \text{ goto } L_4$   
                   goto L1



$B \rightarrow B_1 \&\& B_2$

$B_1.true = \text{newlabel}() \quad B_1.false = B.false$

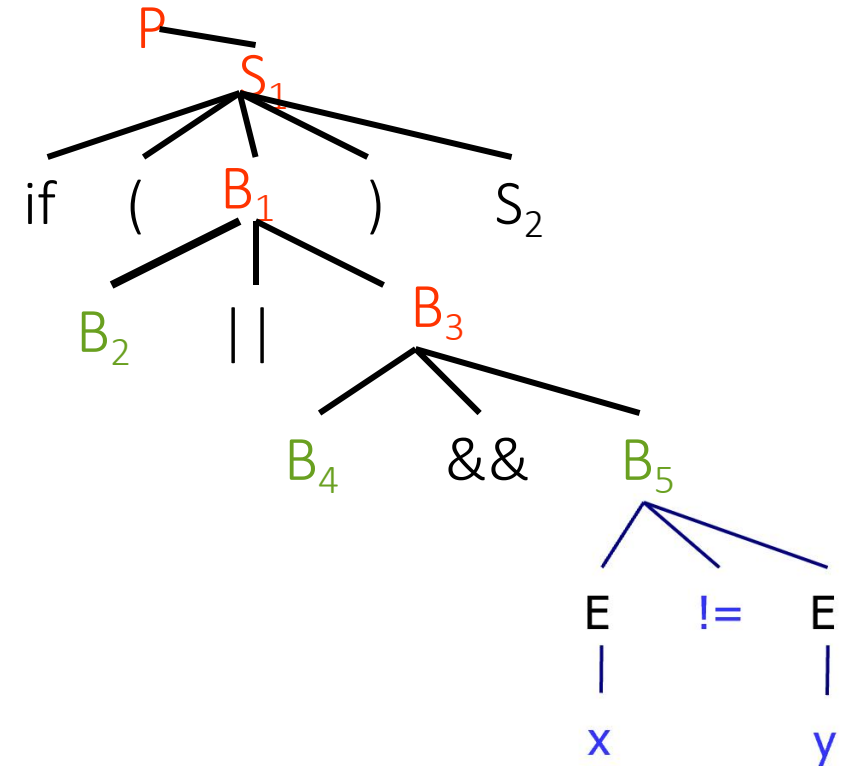
$B_2.true = B.true \quad B_2.false = B.false$

$B.code = B_1.code \ || \ \text{label}(B_1.true) \ || \ B_2.code$

**if(x<100 || x>200 && x!=y) x=0;**

$S_1.next = L_1;$   
 $B_1.true = L_2;$   
 $B_1.false = L_1;$   
 $B_2.true = L_2;$   
 $B_2.false = L_3;$   
 $B_3.true = L_2;$   
 $B_3.false = L_1;$   
 $B_4.true = L_4;$   
 $B_4.false = L_1;$   
 $B_5.true = L_2;$   
 $B_5.false = L_1;$

$B_2.code = \text{if}(x < 100) \text{ goto } L_2$   
           goto L3  
 $B_4.code = \text{if}(x > 200) \text{ goto } L_4$   
           goto L1  
 $B_5.code = \text{if}(x \neq y) \text{ goto } L_2$   
           goto L1



$B \rightarrow E_1 \text{ rel } E_2 \quad B.code = E_1.code \parallel E_2.code \parallel \text{gen}(\text{'if' } E_1.addr \text{ rel.op } E_2.addr \text{ 'goto' } B.true) \parallel \text{gen}(\text{'goto' } B.false)$

**if(x<100 || x>200 && x!=y) x=0;**

$S_1.next = L_1;$

$B_1.true = L_2; B_1.false = L_1;$

$B_2.true = L_2; B_2.false = L_3;$

$B_3.true = L_2; B_3.false = L_1;$

$B_4.true = L_4; B_4.false = L_1;$

$B_5.true = L_2; B_5.false = L_1;$

$B_2.code = \text{if}(x < 100) \text{ goto } L_2$

$\text{goto } L_3$

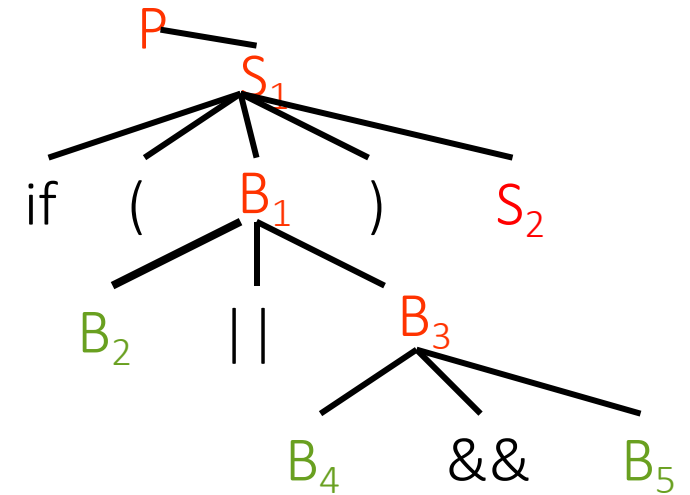
$B_4.code = \text{if}(x > 200) \text{ goto } L_4$

$\text{goto } L_1$

$B_5.code = \text{if}(x \neq y) \text{ goto } L_2$

$\text{goto } L_1$

$B_3.code = B_4.code \ || \ \text{label}(L_4) \ || \ B_5.code$



$B \rightarrow B_1 \ \&\& \ B_2$

$B_1.true = \text{newlabel}() \quad B_1.false = B.false$

$B_2.true = B.true \quad B_2.false = B.false$

$B.code = B_1.code \ || \ \text{label}(B_1.true) \ || \ B_2.code$

**if(x<100 || x>200 && x!=y) x=0;**

$S_1.next = L_1;$

$B_1.true = L_2; B_1.false = L_1;$

$B_2.true = L_2; B_2.false = L_3;$

$B_3.true = L_2; B_3.false = L_1;$

$B_4.true = L_4; B_4.false = L_1;$

$B_5.true = L_2; B_5.false = L_1;$

$B_2.code = \text{if}(x < 100) \text{ goto } L_2$

$\text{goto } L_3$

$B_4.code = \text{if}(x > 200) \text{ goto } L_4$

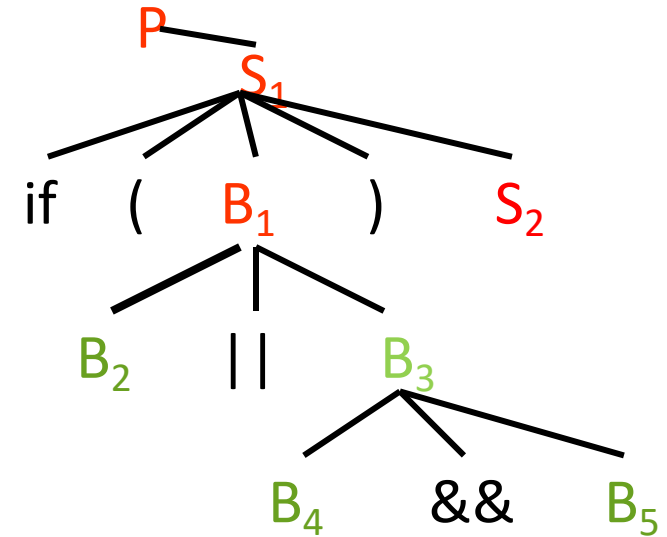
$\text{goto } L_1$

$B_5.code = \text{if}(x \neq y) \text{ goto } L_2$

$\text{goto } L_1$

$B_3.code = B_4.code \ || \ \text{label}(L_4) \ || \ B_5.code$

$B_1.code = B_2.code \ || \ \text{label}(L_3) \ || \ B_3.code$



$B \rightarrow B_1 \ || \ B_2$

$B_1.true = B.true$

$B_1.false = \text{newlabel}()$

$B_2.true = B.true$

$B_2.false = B.false$

$B.code = B_1.code \ || \ \text{label}(B_1.false) \ || \ B_2.code$

**if(x<100 || x>200 && x!=y) x=0;**

$S_1.next = L_1;$

$B_1.true = L_2; B_1.false = L_1;$

$B_2.true = L_2; B_2.false = L_3;$

$B_3.true = L_2; B_3.false = L_1;$

$B_4.true = L_4; B_4.false = L_1;$

$B_5.true = L_2; B_5.false = L_1;$

$B_2.code = \text{if}(x < 100) \text{ goto } L_2$

$\text{goto } L_3$

$B_4.code = \text{if}(x > 200) \text{ goto } L_4$

$\text{goto } L_1$

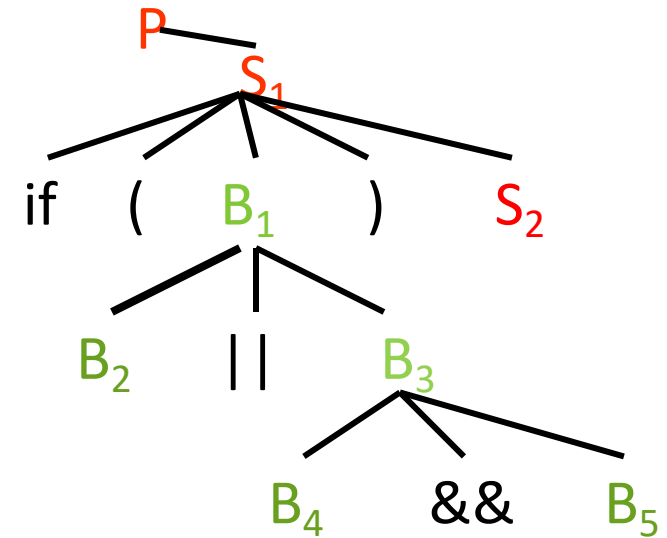
$B_5.code = \text{if}(x \neq y) \text{ goto } L_2$

$\text{goto } L_1$

$B_3.code = B_4.code \ || \ \text{label}(L_4) \ || \ B_5.code$

$B_1.code = B_2.code \ || \ \text{label}(L_3) \ || \ B_3.code$

$S_2.code = x = 0$



$S \rightarrow \text{if } (B) S_1$        $B.true = \text{newlabel}()$   
                                   $B.false = S_1.next = S.next$   
                                   $S.code = B.code \ || \ \text{label}(B.true) \ || \ S_1.code$

**if(x<100 || x>200 && x!=y) x=0;**

$S_1.next = L_1;$

$B_1.true = L_2;$

$B_1.false = L_1;$

$B_2.true = L_2;$

$B_2.false = L_3;$

$B_3.true = L_2;$

$B_3.false = L_1;$

$B_4.true = L_4;$

$B_4.false = L_1;$

$B_5.true = L_2;$

$B_5.false = L_1;$

$B_2.code = \text{if}(x < 100) \text{ goto } L_2$

$\text{goto } L_3$

$B_4.code = \text{if}(x > 200) \text{ goto } L_4$

$\text{goto } L_1$

$B_5.code = \text{if}(x \neq y) \text{ goto } L_2$

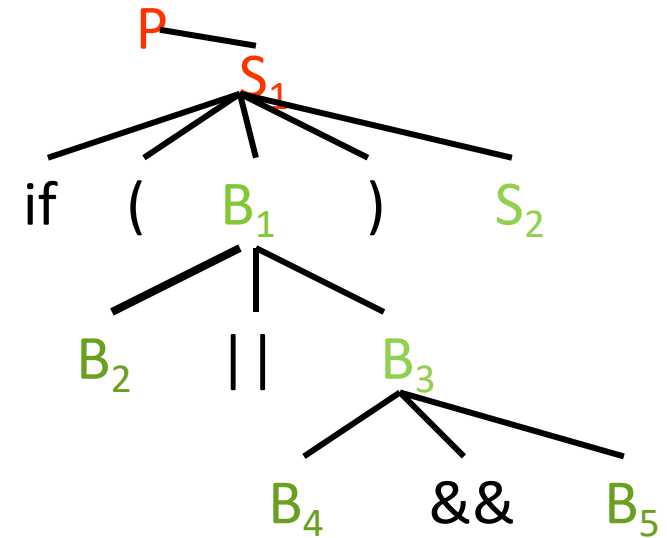
$\text{goto } L_1$

$B_3.code = B_4.code \ || \ \text{label}(L_4) \ || \ B_5.code$

$B_1.code = B_2.code \ || \ \text{label}(L_3) \ || \ B_3.code$

$S_2.code = x = 0$

$S_1.code = B_1.code \ || \ \text{label}(L_2) \ || \ S_2.code$



$S \rightarrow \text{if } (B) S_1$

$B.true = \text{newlabel}()$

$B.false = S_1.next = S.next$

$S.code = B.code \ || \ \text{label}(B.true) \ || \ S_1.code$



if(x<100 || x>200 && x!=y) x=0;

S<sub>1</sub>.next=L<sub>1</sub>;

B<sub>1</sub>.true=L<sub>2</sub>;

B<sub>1</sub>.false=L<sub>1</sub>;

B<sub>2</sub>.true=L<sub>2</sub>;

B<sub>2</sub>.false=L<sub>3</sub>;

B<sub>3</sub>.true=L<sub>2</sub>;

B<sub>3</sub>.false=L<sub>1</sub>;

B<sub>4</sub>.true=L<sub>4</sub>;

B<sub>4</sub>.false=L<sub>1</sub>;

B<sub>5</sub>.true=L<sub>2</sub>;

B<sub>5</sub>.false=L<sub>1</sub>;

B<sub>2</sub>.code = if(x<100) goto L2

goto L3

B<sub>4</sub>.code = if(x>200) goto L4

goto L1

B<sub>5</sub>.code = if(x!=y) goto L2

goto L1

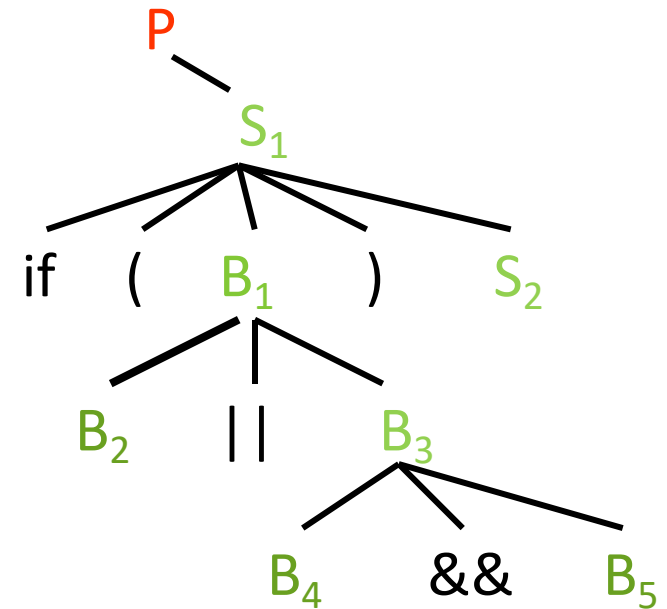
B<sub>3</sub>.code = B<sub>4</sub>.code || label(L<sub>4</sub>) || B<sub>5</sub>.code

B<sub>1</sub>.code = B<sub>2</sub>.code || label(L<sub>3</sub>) || B<sub>3</sub>.code

S<sub>2</sub>.code = x=0

S<sub>1</sub>.code = B<sub>1</sub>.code || label(L<sub>2</sub>) || S<sub>2</sub>.code

P.code = S<sub>1</sub>.code || label(L<sub>1</sub>)



$P \rightarrow S$

$S.next = newlabel( )$

$P.code = S.code || label(S.next)$

**if(x<100 || x>200 && x!=y) x=0;**

S<sub>1</sub>.next=L<sub>1</sub>;  
B<sub>1</sub>.true=L<sub>2</sub>;  
B<sub>1</sub>.false=L<sub>1</sub>;  
B<sub>2</sub>.true=L<sub>2</sub>;  
B<sub>2</sub>.false=L<sub>3</sub>;  
B<sub>3</sub>.true=L<sub>2</sub>;  
B<sub>3</sub>.false=L<sub>1</sub>;  
B<sub>4</sub>.true=L<sub>4</sub>;  
B<sub>4</sub>.false=L<sub>1</sub>;  
B<sub>5</sub>.true=L<sub>2</sub>;  
B<sub>5</sub>.false=L<sub>1</sub>;

B<sub>2</sub>.code = if(x<100) goto L2  
          goto L3  
B<sub>4</sub>.code = if(x>200) goto L4  
          goto L1  
B<sub>5</sub>.code = if(x!=y) goto L2  
          goto L1  
B<sub>3</sub>.code = B<sub>4</sub>.code || label(L<sub>4</sub>) || B<sub>5</sub>.code  
B<sub>1</sub>.code = B<sub>2</sub>.code || label(L<sub>3</sub>) || B<sub>3</sub>.code  
S<sub>2</sub>.code = x=0  
S<sub>1</sub>.code = B<sub>1</sub>.code || label(L<sub>2</sub>) || S<sub>2</sub>.code  
P.code = S<sub>1</sub>.code || label(L<sub>1</sub>)

**if(x<100) goto L2  
goto L3  
L3: if(x>200) goto L4  
goto L1  
L4: if(x!=y) goto L2  
goto L1  
x=0  
L2:  
L1:**

# 避免生成冗余的goto指令

---

$x > 200$  翻译为:

```
if  $x > 200$  goto  $L_4$   
goto  $L_1$ 
```

$L_4$ :

替换为:

```
ifFalse  $x > 200$  goto  $L1$ 
```

```
 $L4$ :...
```

# 避免生成冗余的goto指令

---

特殊标号“fall”：不生成任何跳转指令

```
B.true = newlabel()  
B.false = S1.next = S.next  
S.code = B.code || label(B.true) || S1.code
```

```
B.true = fall  
B.false = S1.next = S.next  
S.code = B.code || S1.code
```

# 避免生成冗余的goto指令

B特殊标号“fall”：不生成任何跳转指令

$B \rightarrow E_1 \text{ rel } E_2 \quad B.code = E_1.code \parallel E_2.code \parallel \text{gen}(\text{'if' } E_1.addr \text{ rel.op } E_2.addr \text{ 'goto' } B.true) \parallel \text{gen}(\text{'goto' } B.false)$

$B \rightarrow E_1 \text{ rel } E_2$  的语义规则

$test = E_1.addr \text{ rel.op } E_2.addr$   
 $s = \text{if } B.true \neq fall \text{ and } B.false \neq fall \text{ then}$   
 $\text{gen}(\text{'if' } test \text{ 'goto' } B.true) \parallel \text{gen}(\text{'goto' } B.false)$   
 $\text{else if } B.true \neq fall \text{ then } \text{gen}(\text{'if' } test \text{ 'goto' } B.true)$   
 $\text{else if } B.false \neq fall \text{ then } \text{gen}(\text{'ifFalse' } test \text{ 'goto' } B.false)$   
 $\text{else ''}$   
 $B.code = E_1.code \parallel E_2.code \parallel s$

# 学习内容

---

- 6.1 类型检查
- 6.2 中间表示
- 6.3 声明语句
- 6.4 赋值语句
- 6.5 控制流
- **6.6 回填**
- 6.7 switch语句
- 6.8 过程的中间代码



# 回填 (Backpatching)

---

语法制导定义如何实现?

在之前的语法制导定义中, *S.next*, *E.true*, *E.false*均为继承属性。可以通过两边扫描, 首先构造分析树, 然后再深度优先遍历分析树进行翻译。

为了避免构造分析树, 实现一遍扫描, 可以采用“**回填**”技术。

这种技术针对一个未知地址A设置一个列表, 表中记录需要用地址A填充的指令的位置(用语句序号表示), 一旦地址A的标号被确定, 将标号回填到表中的指令中。

# 翻译模式中使用的全局变量、函数和属性

---

***B***的综合属性 $truelist, falselist$ : 存放回填表指针，分别记录真值出口和假值出口的待填指令的序号。

全局变量 $nextinstr$ : 保存紧跟着的下一条指令的序号。使用四元式时不必生成专门的标号，四元式的序号就作为四元式的地址。由指令生成函数自动增值。

***M.instr***: 记录布尔表达式代码段第一条指令的序号，该属性值即为回填的地址。



# 基本的回填操作

---

- 1、*makelist(i)*: 建立一个含语句序号*i*的新回填表, *makelist()*表示建立一个空表。
- 2、*merge(p<sub>1</sub>, p<sub>2</sub>, ..., p<sub>n</sub>)*: 将回填表 *p<sub>1</sub>, p<sub>2</sub>, ..., p<sub>n</sub>* 合并为一个回填表, 参数和返回值均为指向回填表的指针。
- 3、*backpatch(p, i)*: 回填, 即将语句地址*i*填入回填表*p*中各语句序号所指示的四元式中的待填地址域中。

# 使用回填翻译布尔表达式

---

构造翻译模式，用于自底向上语法分析过程中生成布尔表达式的四元式。

基础文法：

$$\begin{aligned} B \rightarrow & B_1 \parallel M B_2 \\ & | B_1 \&\& M B_2 \\ & | ! B_1 \\ & | ( B_1 ) \\ & | E_1 \text{ rel } E_2 \\ & | \text{true} \\ & | \text{false} \end{aligned}$$
$$M \rightarrow \varepsilon$$

# 语义动作分析

$B \rightarrow B_1 \ \&\& \ M \ B_2$

- 如果 $B_1$ 为假，那么 $B$ 一定为假
- 如果 $B_1$ 为真，那么下一步计算 $B_2$ ， $B_1$ 的 $truelist$ 就是 $B_2$ 的第一条语句。

if (a>0 && b>0) c=0;  
else c=1;

$B \rightarrow B_1 \ \&\& \ M \ B_2$

{ *backpatch*( $B_1.truelist$ ,  $M.instr$ );

$B.truelist = B_2.truelist$ ;

$B.falselist = merge(B_1.falselist, B_2.falselist);$  }

$M.instr$

**B1.truelist** 100: if a>0 goto 102

**B1.falselist** 101: goto 105

**B2.truelist** 102: if b>0 goto 104

**B2.falselist** 103: goto 105

**B.truelist** 104: c = 0

**B.falselist** 105: c = 1

# 布尔表达式的翻译模式

---

$B \rightarrow B_1 \parallel \textcolor{red}{M} B_2$

$\{ \text{backpatch}(B_1.\text{falselist}, M.\text{instr});$   
 $B.\text{truelist} = \text{merge}(B_1.\text{truelist}, B_2.\text{truelist});$   
 $B.\text{falselist} = B_2.\text{falselist}; \}$

$B \rightarrow B_1 \&\& \textcolor{red}{M} B_2$

$\{ \text{backpatch}(B_1.\text{truelist}, M.\text{instr});$   
 $B.\text{trulist} = B_2.\text{truelist};$   
 $B.\text{falselist} = \text{merge}(B_1.\text{falselist}, B_2.\text{falselist}); \}$

# 布尔表达式的翻译模式

---

$$B \rightarrow ! B_1 \{ B.truelist = B_1.falselist; \\ B.falselist = B_1.truelist; \}$$

$$B \rightarrow (B_1) \{ B.truelist = B_1.truelist; \\ B.falselist = B_1.falselist; \}$$

# 布尔表达式的翻译模式

---

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

```
{  $B.trulist = makelist(nextinstr);$   
   $B.falselist = makelist(nextinstr+1);$   
   $gen('if' E_1.addr \text{ relop.op } E_2.addr 'goto \_');$   
   $gen('goto \_');$ }
```

$B \rightarrow \text{true}$

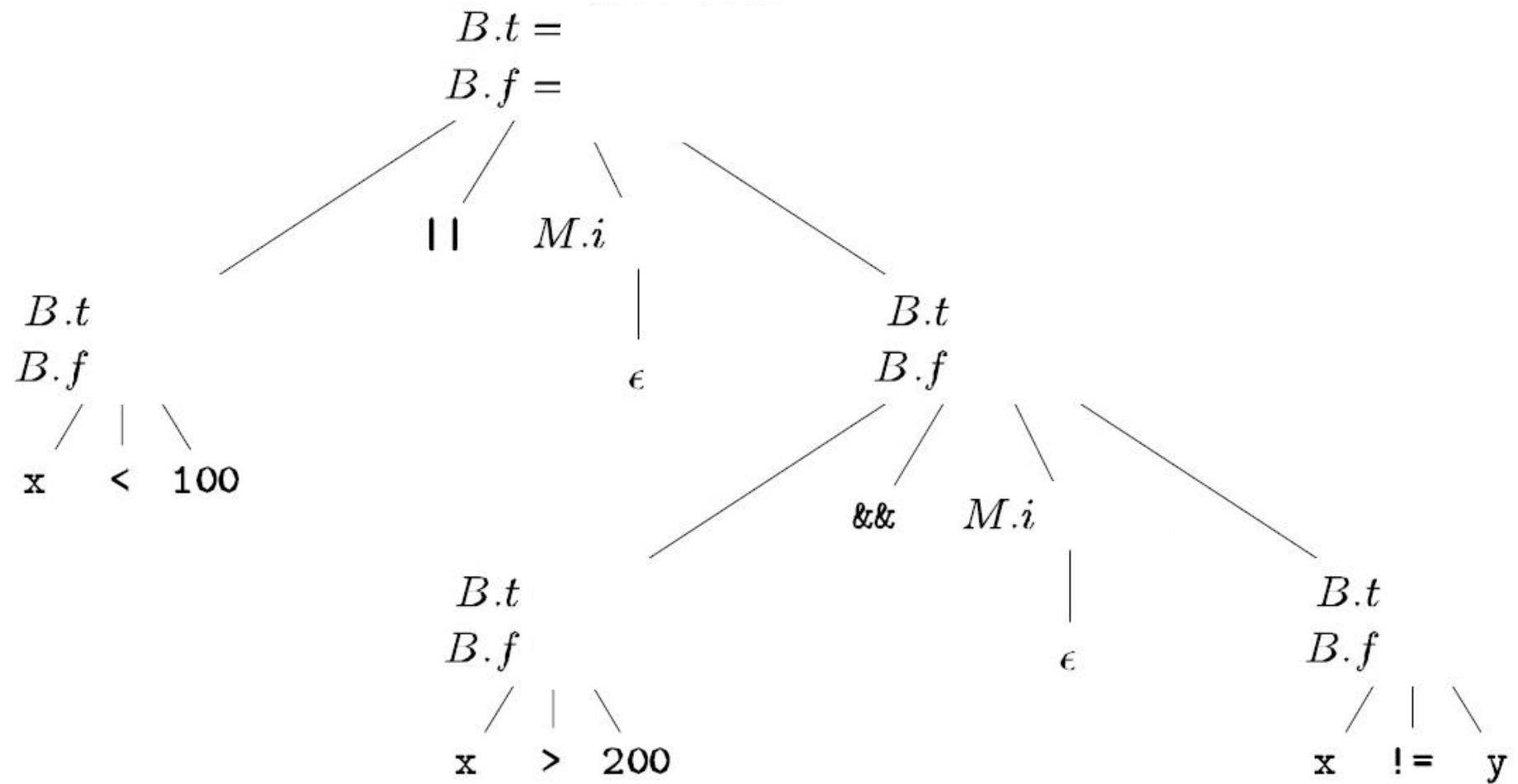
```
{  $B.trulist = makelist(nextinstr);$   
   $gen('goto \_');$ }
```

$B \rightarrow \text{false}$

```
{  $B.falselist = makelist(nextinstr);$   
   $gen('goto \_');$ }
```

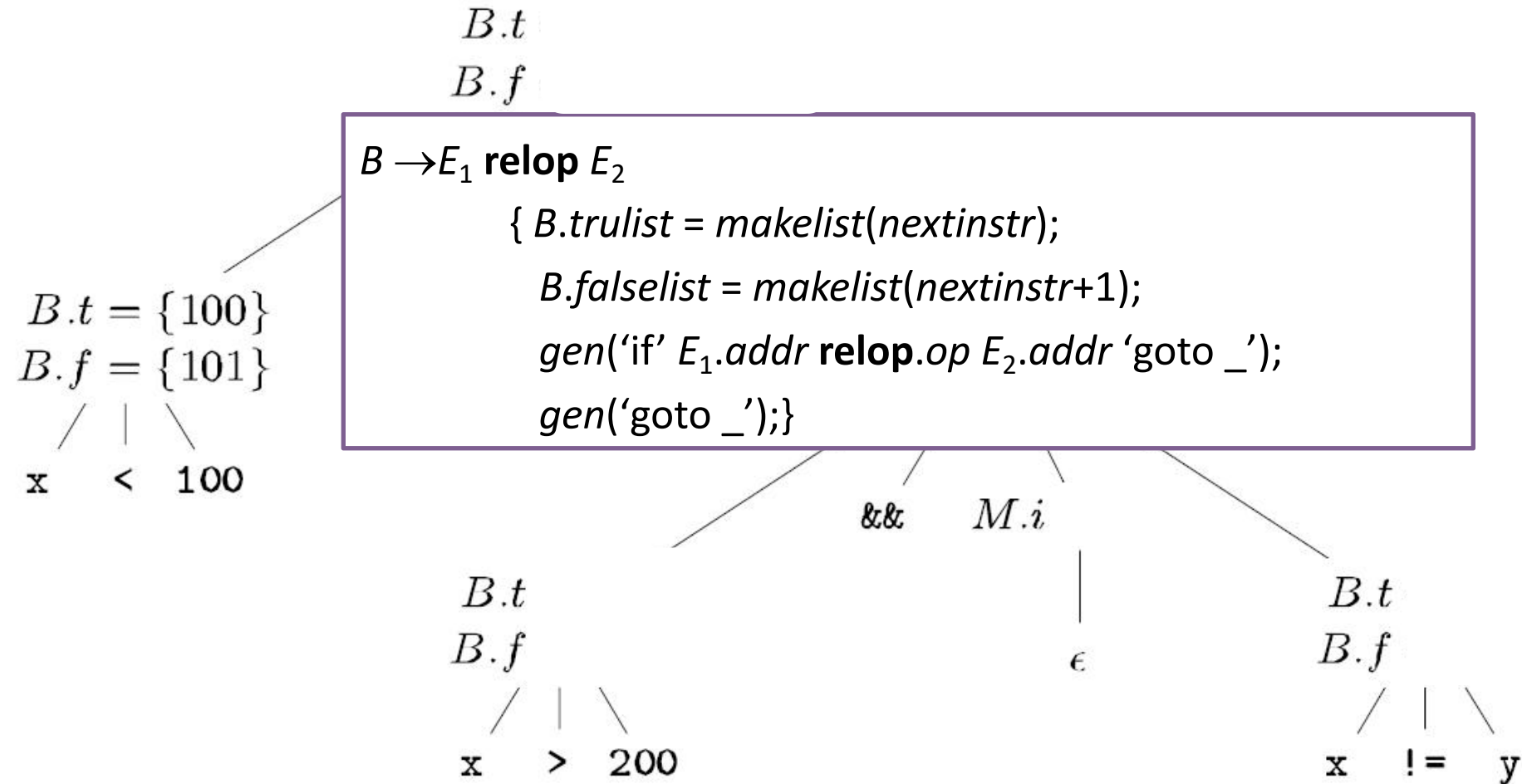
$M \rightarrow \varepsilon \{ M.instr = nextinstr; \}$

$x < 100 \ || \ x > 200 \ \&\& \ x \neq y$



$x < 100 \ || \ x > 200 \ \&\& \ x \neq y$

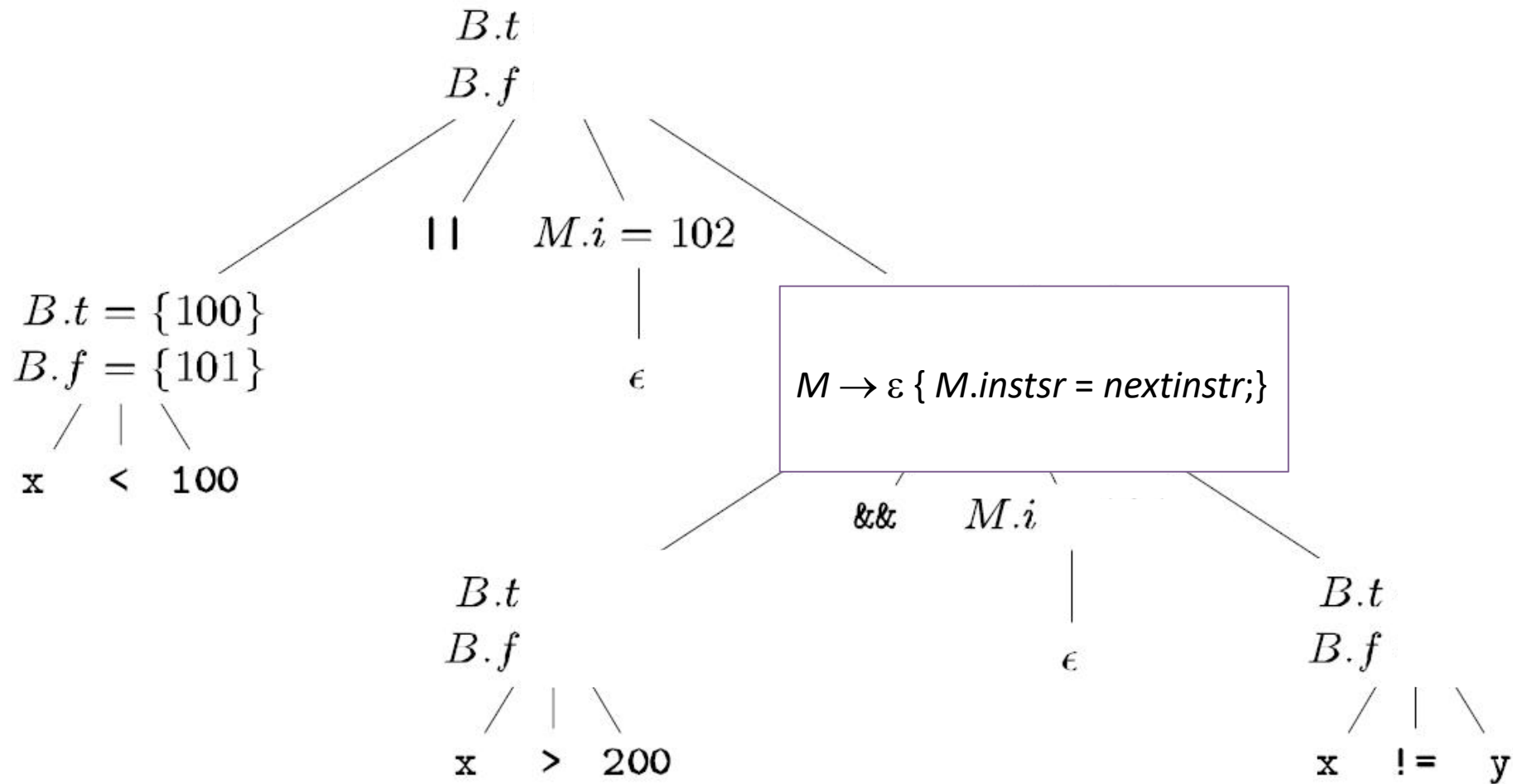
100: if  $x < 100$  goto \_  
101: goto \_





$x < 100 \ || \ x > 200 \ \&\& \ x \neq y$

100: if  $x < 100$  goto \_  
101: goto \_



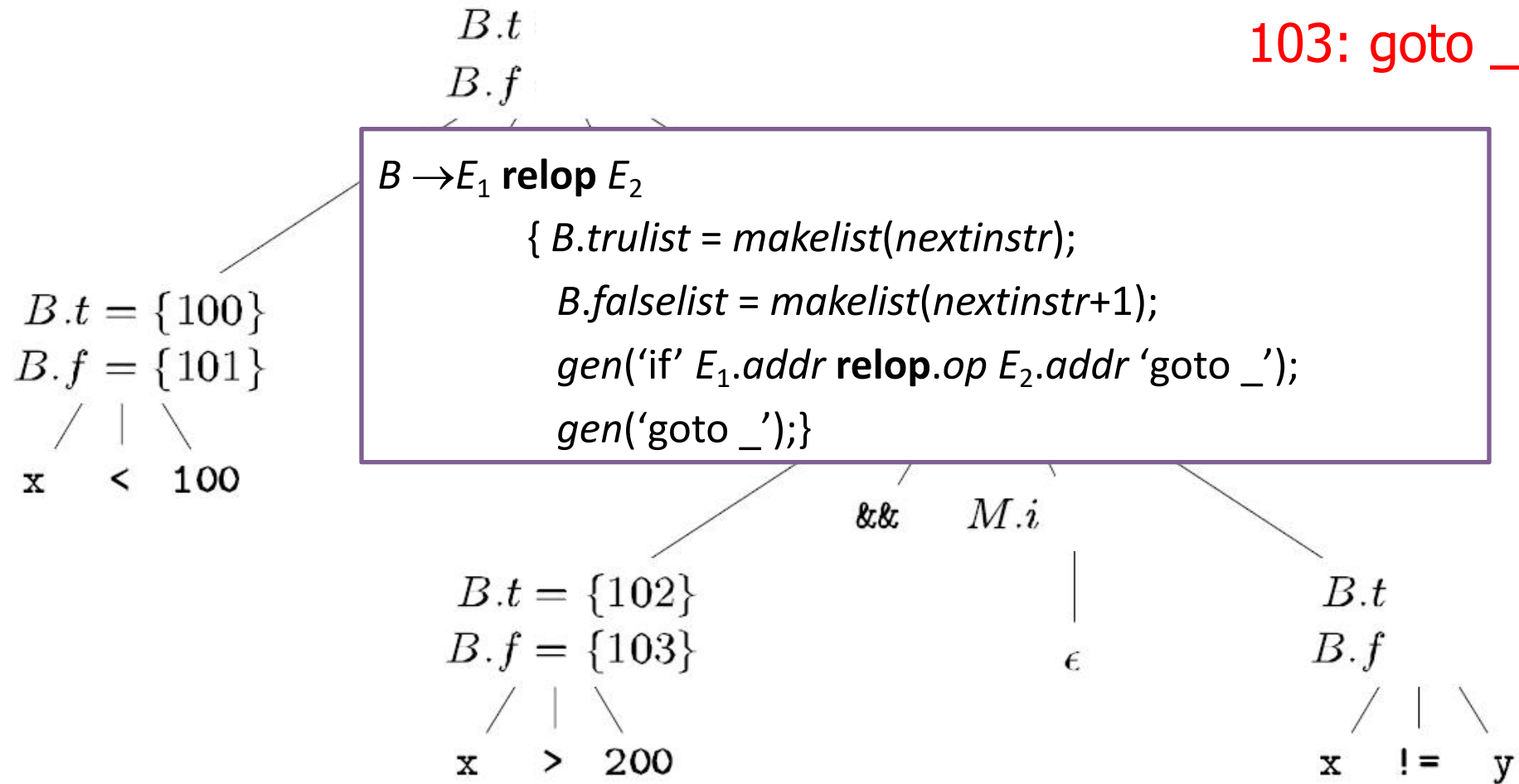
$x < 100 \ || \ x > 200 \ \&\& \ x \neq y$

100: if  $x < 100$  goto \_

101: goto \_

102: if  $x > 200$  goto \_

103: goto \_



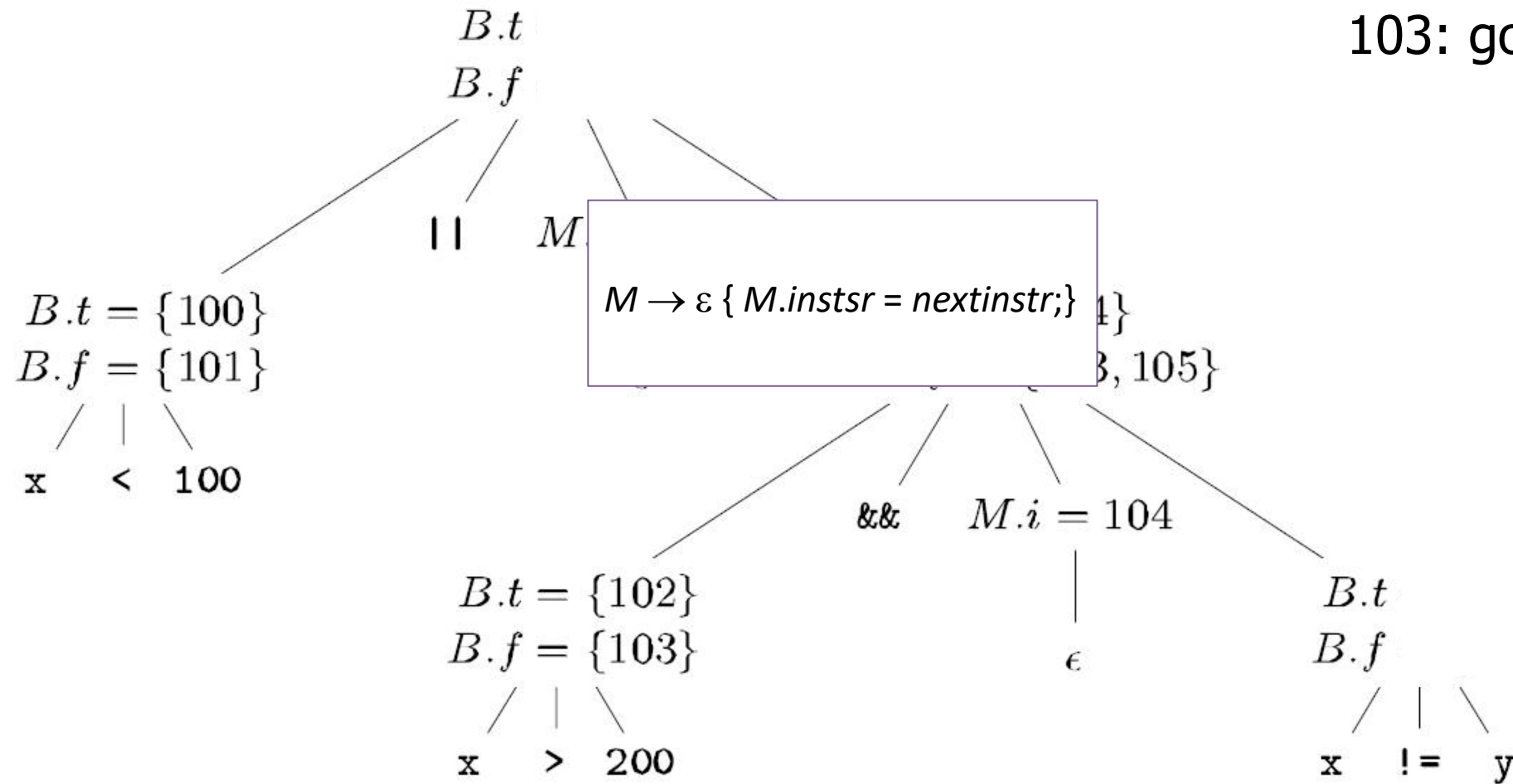
$x < 100 \ || \ x > 200 \ \&\& \ x \neq y$

100: if  $x < 100$  goto \_

101: goto \_

102: if  $x > 200$  goto \_

103: goto \_



$x < 100 \ || \ x > 200 \ \&\& \ x \neq y$

100: if  $x < 100$  goto \_

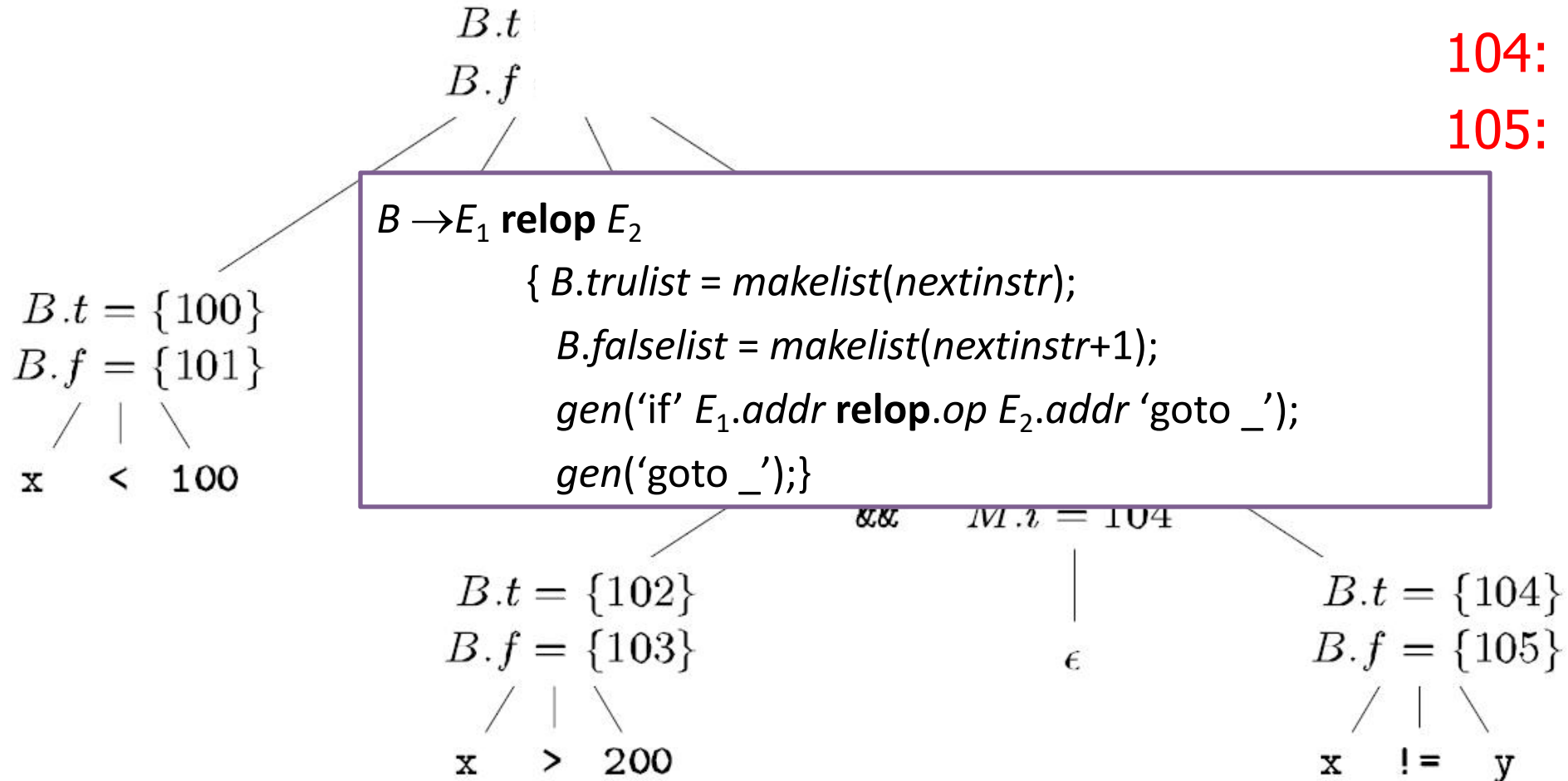
101: goto \_

102: if  $x > 200$  goto \_

103: goto \_

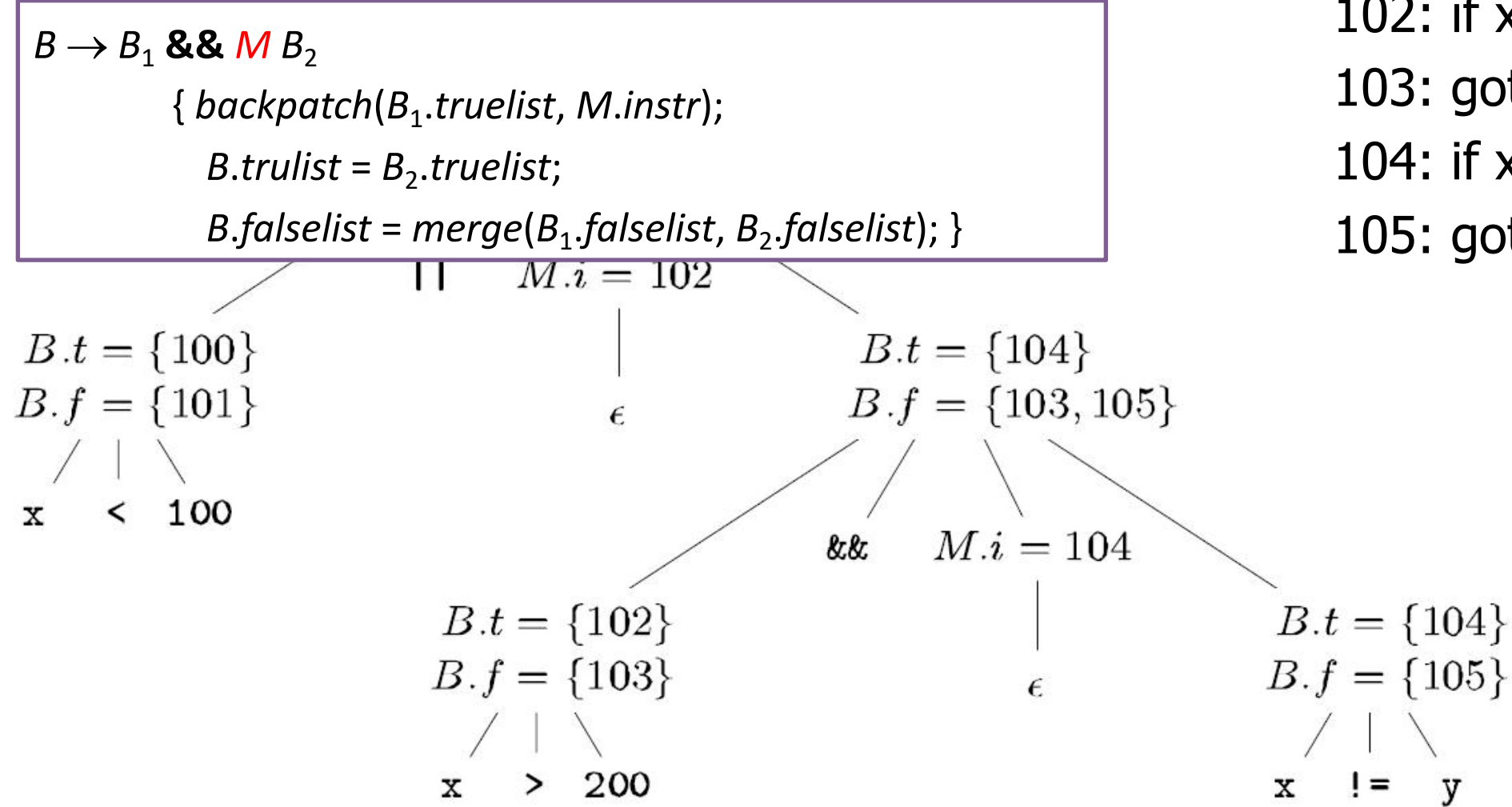
104: if  $x \neq y$  goto \_

105: goto \_



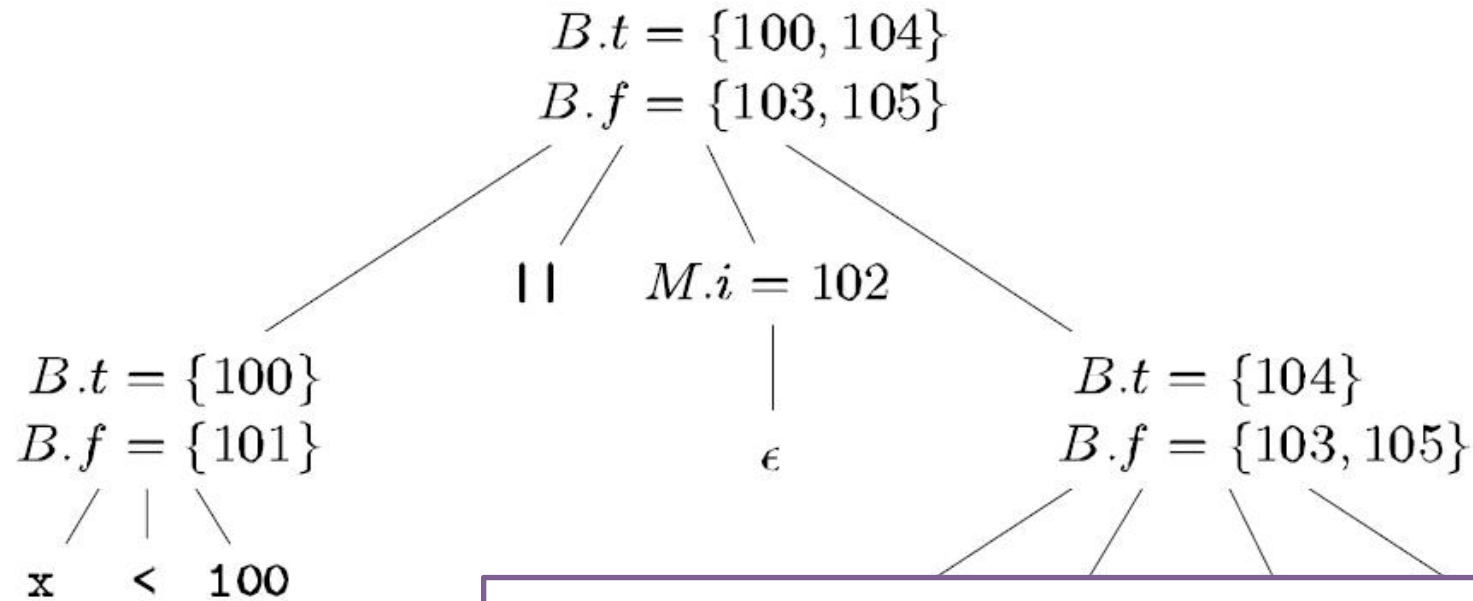
$x < 100 \ || \ x > 200 \ \&\& \ x \neq y$

100: if  $x < 100$  goto \_  
101: goto \_  
102: if  $x > 200$  goto 104  
103: goto \_  
104: if  $x \neq y$  goto \_  
105: goto \_



$x < 100 \ || \ x > 200 \ \&\& \ x \neq y$

100: if  $x < 100$  goto \_  
 101: goto **102**  
 102: if  $x > 200$  goto 104  
 103: goto \_  
 104: if  $x \neq y$  goto \_  
 105: goto \_



$B \rightarrow B_1 \ || \ M \ B_2$

{ *backpatch*( $B_1$ .*false*list,  $M$ .*instr*);

$B$ .*true*list = *merge*( $B_1$ .*true*list,  $B_2$ .*true*list);

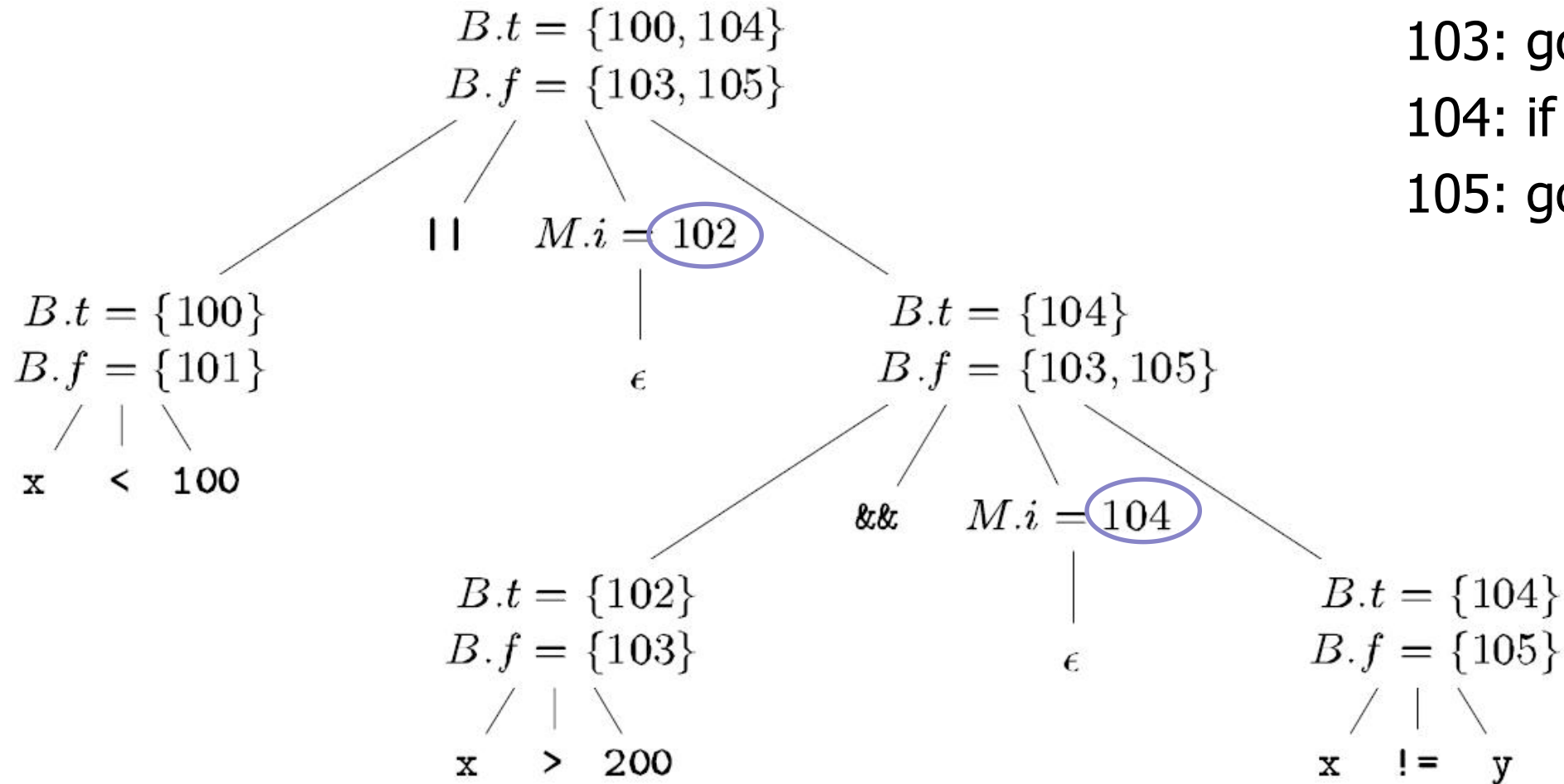
$B$ .*false*list =  $B_2$ .*false*list;}

$x > 200$

$x \neq y$

$x < 100 \ || \ x > 200 \ \&\& \ x \neq y$

100: if  $x < 100$  goto \_  
101: goto 102  
102: if  $x > 200$  goto 104  
103: goto \_  
104: if  $x \neq y$  goto \_  
105: goto \_



# 使用回填翻译控制流语句

---

## 1、语法

(1)  $S \rightarrow \text{if } ( B ) S$

(2)     |  $\text{if } ( B ) S \text{ else } S$

(3)     |  $\text{while } ( B ) \text{ do } S$

(4)     |  $\{ L \}$

(5)     |  $A;$

(6)  $L \rightarrow L S$

(7)     |  $S$

上述语法描述了比较完整的程序设计语言的语句



# 使用回填翻译控制流语句

---

当确定目标时再回填跳转语句的转向

- $B$ 的真转向
- $B$ 的假转向
- 语句的 $next$ 转向

属性

- $L.nextlist, S.nextlist, N.nextlist$ : 存放回填表指针, 其中记录了将来要用后继语句的序号回填的待填指令序号。

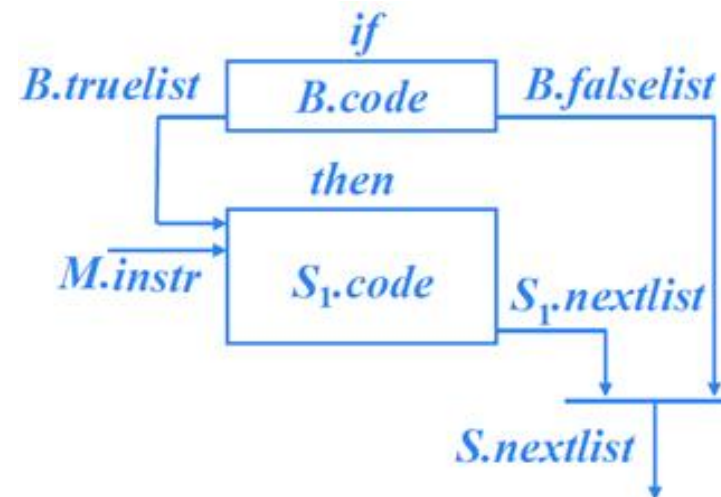
# 翻译模式

---

(1)  $S \rightarrow \text{if } (B) \text{ } M S_1$

$\{ \text{backpatch}(B.\text{truelist}, M.\text{instr});$   
 $S.\text{nextlist} = \text{merge}(B.\text{falselist}, S_1.\text{nextlist}); \}$

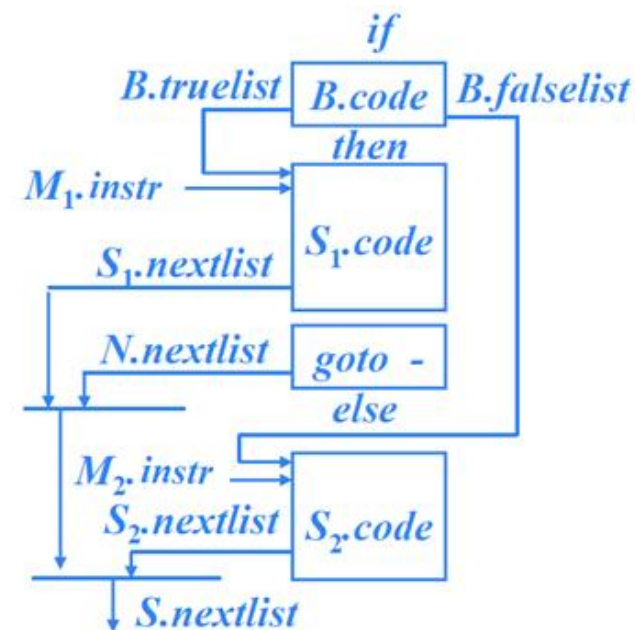
(2)  $M \rightarrow \varepsilon \quad \{ M.\text{instr} = \text{nextinstr}; \}$



# 翻译模式

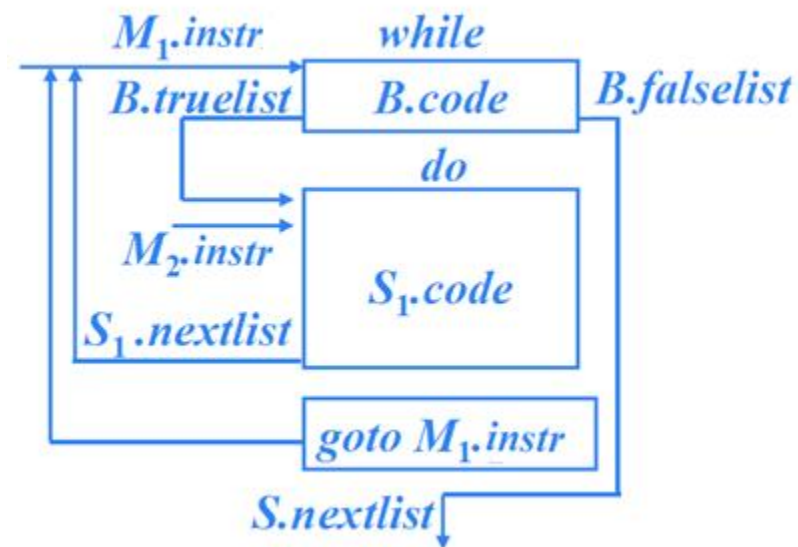
(3)  $S \rightarrow \text{if } (B) \text{ } M_1 \text{ } S_1 \text{ } N \text{ else } M_2 \text{ } S_2$   
 $\{ \text{backpatch}(B.\text{truelist}, M_1.\text{instr});$   
 $\text{backpatch}(B.\text{falselist}, M_2.\text{instr});$   
 $S.\text{nextlist} = \text{merge}(S_1.\text{nextlist}, N.\text{nextlist},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

(4)  $N \rightarrow \varepsilon$   
 $\{ N.\text{nextlist} = \text{makelist}(\text{nextinstr});$   
 $\text{gen}(\text{'goto _'}) \}$



# 翻译模式

(5)  $S \rightarrow \text{while } M_1 (B) \text{ do } M_2 S_1$   
 $\{ \text{backpatch}(B.\text{truelist}, M_2.\text{instr});$   
 $\text{backpatch}(S_1.\text{nextlist}, M_1.\text{instr});$   
 $S.\text{nextlist} = B.\text{falselist};$   
 $\text{gen}(\text{'goto'}, M_1.\text{instr}) \}$



# 翻译模式

---

(6)  $S \rightarrow \{ L \}$   
 $\{ S.nextlist = L.nextlist \}$

(7)  $S \rightarrow A; \quad \{ S.nextlist = \mathbf{null}; \}$

(8)  $L \rightarrow L_1 \textcolor{red}{M} S$   
 $\{ \textit{backpatch}(L_1.nextlist, M.instr);$   
 $\quad L.nextlist = S.nextlist; \}$

(9)  $L \rightarrow S \quad \{ L.nextlist = S.nextlist \}$

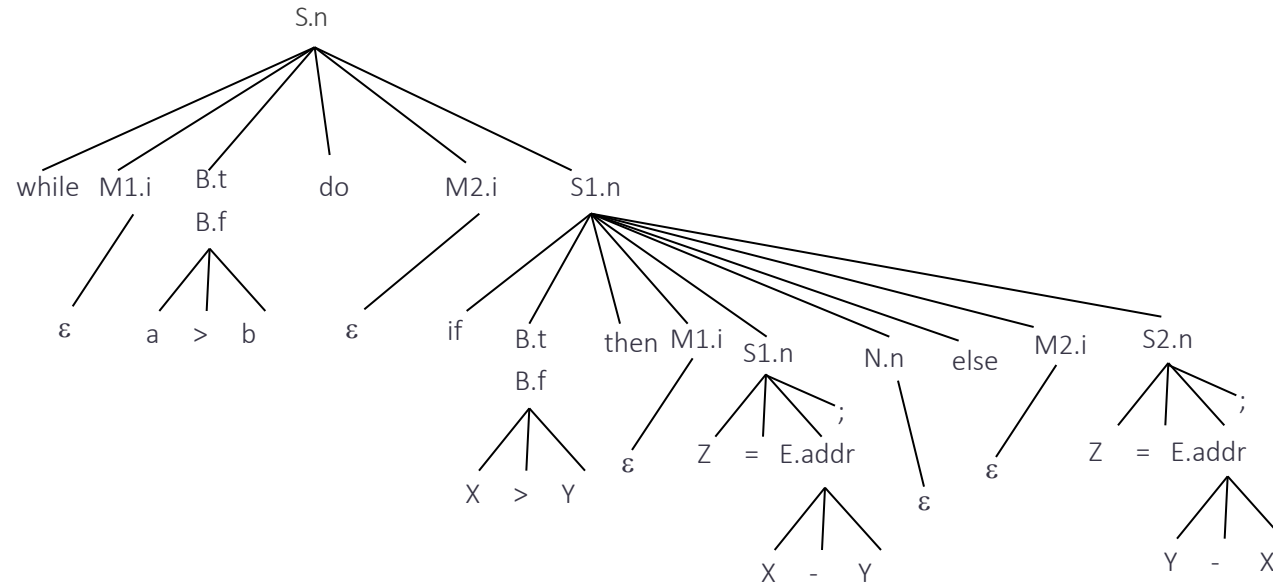
# 注意

---

除了规则(4)、(5)之外都没有生成新的四元式；  
其它四元式都是由赋值语句和表达式的语义动作产生

控制流仅仅是进行适当的回填，以便赋值语句和表达式的产生四元式能正确地连接

while a>b do if X>Y then Z=X-Y; else Z=Y-X;

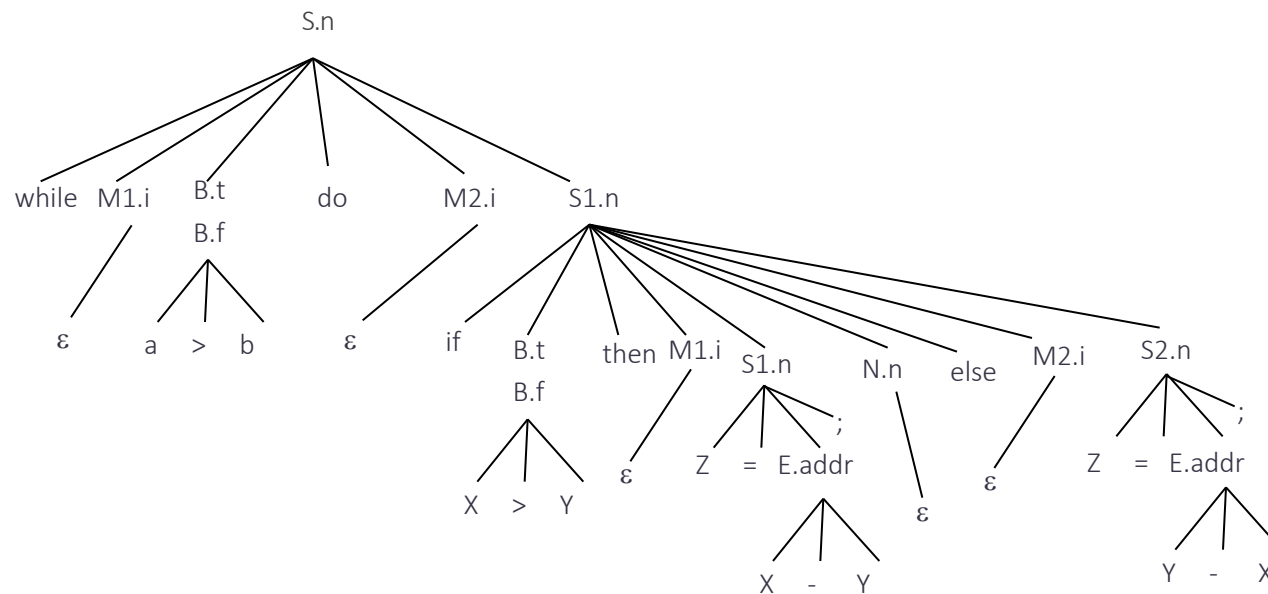




while a>b do if X>Y then Z=X-Y; else Z=Y-X;

$S \rightarrow \text{while } M_1(B) \text{ do } M_2 S_1$   
 $\{ \text{backpatch}(B.\text{truelist}, M_2.\text{instr});$   
 $\text{backpatch}(S_1.\text{nextlist}, M_1.\text{instr});$   
 $S.\text{nextlist} = B.\text{falselist};$   
 $\text{gen}(\text{'goto'}, M_1.\text{instr}) \}$

$M \rightarrow \epsilon \{ \underline{M.\text{instr}} = \underline{\text{nextinstr}}; \}$



100:



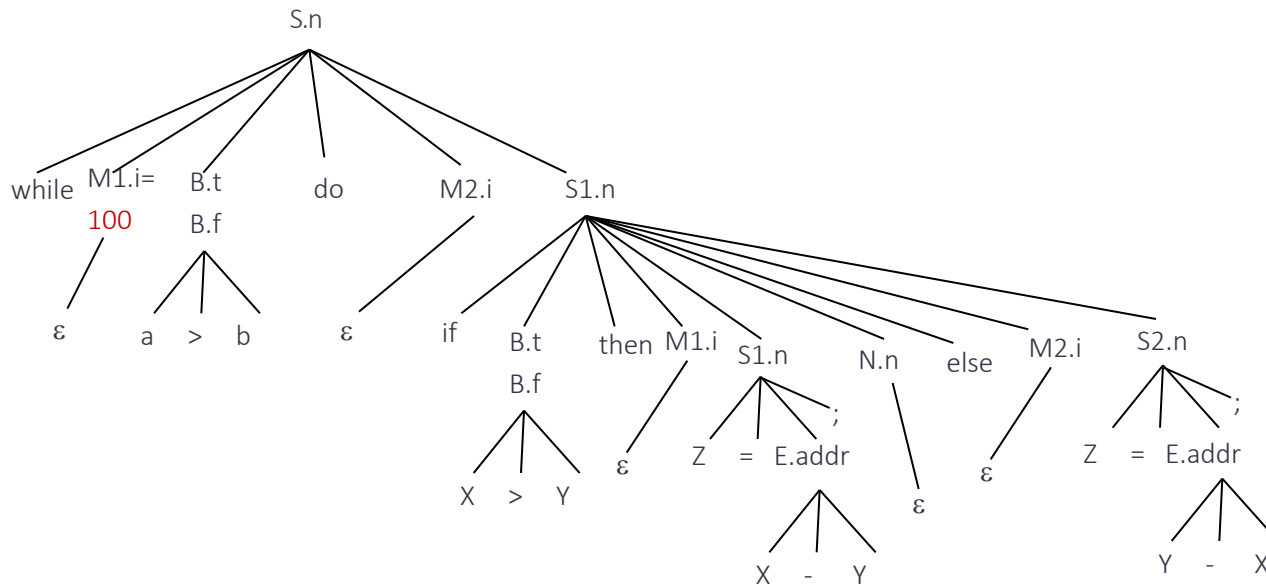


while a>b do if X>Y then Z=X-Y; else Z=Y-X;

$S \rightarrow \text{while } M_1(B) \text{ do } M_2 S_1$   
 $\{ \text{backpatch}(B.\text{truelist}, M_2.\text{instr});$   
 $\text{backpatch}(S_1.\text{nextlist}, M_1.\text{instr});$   
 $S.\text{nextlist} = B.\text{falselist};$   
 $\text{gen}(\text{'goto'}, M_1.\text{instr}) \}$

$M \rightarrow \epsilon \{ \underline{M.\text{instr}} = \underline{\text{nextinstr}}; \}$

100:





while a>b do if X>Y then Z=X-Y; else Z=Y-X;

$S \rightarrow \text{while } M_1 (B) \text{ do}$   
 $\{ \text{backpatch}(B.\text{trulist}, M_1.i, S.n);$   
 $\text{backpatch}(S_1.\text{nextlist}, B.f, S.n);$   
 $S.\text{nextlist} = B.f; \text{gen}(\text{'goto'}, M_1.i, S.n); \}$

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

$\{ B.\text{trulist} = \text{makelist}(\text{nextinstr});$

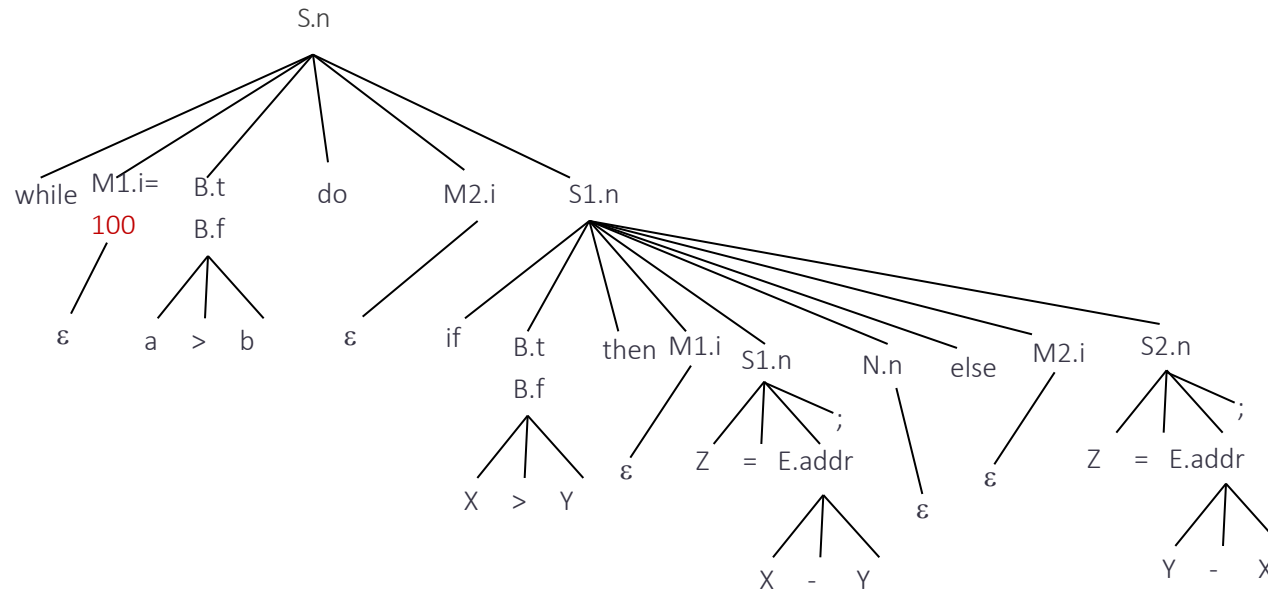
$B.\text{falselist} = \text{makelist}(\text{nextinstr}+1);$

$\text{gen}(\text{'if' } E_1.\text{addr relop.op } E_2.\text{addr 'goto _'});$

$\text{gen}(\text{'goto _'}); \}$

$M \rightarrow \epsilon \{ \underline{M.\text{instr}} = \underline{\text{nextinstr}}; \}$

100:





while a>b do if X>Y then Z=X-Y; else Z=Y-X;

$S \rightarrow \text{while } E_1 \text{ relop } E_2$   
{ backp  
backp  
 $S.nextl$   
gen('g

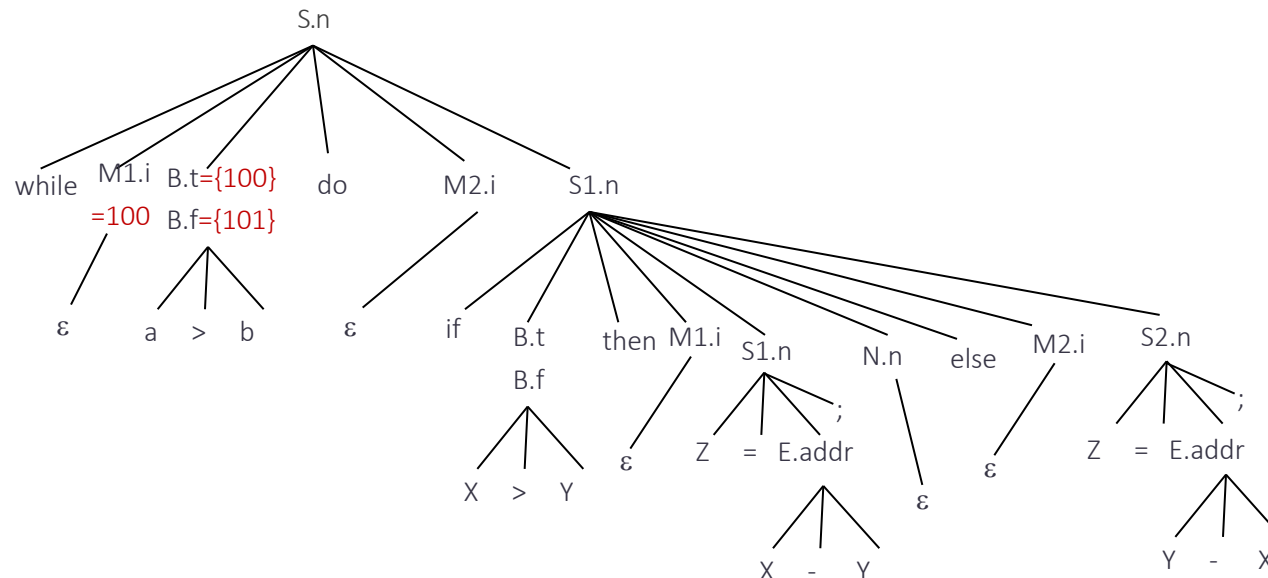
$B \rightarrow E_1 \text{ relop } E_2$

{  $B.trulist = makelist(nextinstr);$   
 $B.falselist = makelist(nextinstr+1);$   
 $gen('if' E_1.addr \text{ relop.op } E_2.addr 'goto \_');$   
 $gen('goto \_');$  }

$nstr; \}$

100: if a>b goto \_

101: goto\_





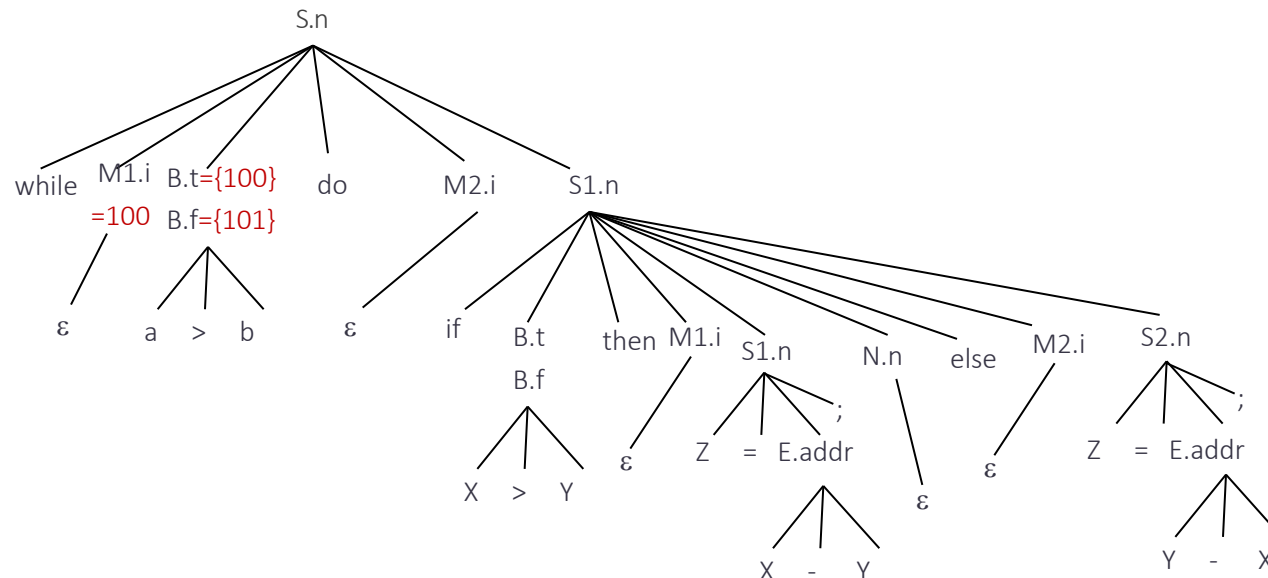
while a>b do if X>Y then Z=X-Y; else Z=Y-X;

$S \rightarrow \text{while } M_1 (B) \text{ do } M_2 S_1$   
 $\{ \text{backpatch}(B.\text{truelist}, M_2.\text{instr});$   
 $\text{backpatch}(S_1.\text{nextlist}, M_1.\text{instr});$   
 $S.\text{nextlist} = B.\text{falselist};$   
 $\text{gen}(\text{'goto'}, M_1.\text{instr}) \}$

$M \rightarrow \epsilon \{ \underline{M.\text{instr}} = \underline{\text{nextinstr}}; \}$

100: if a>b goto \_

101: goto\_





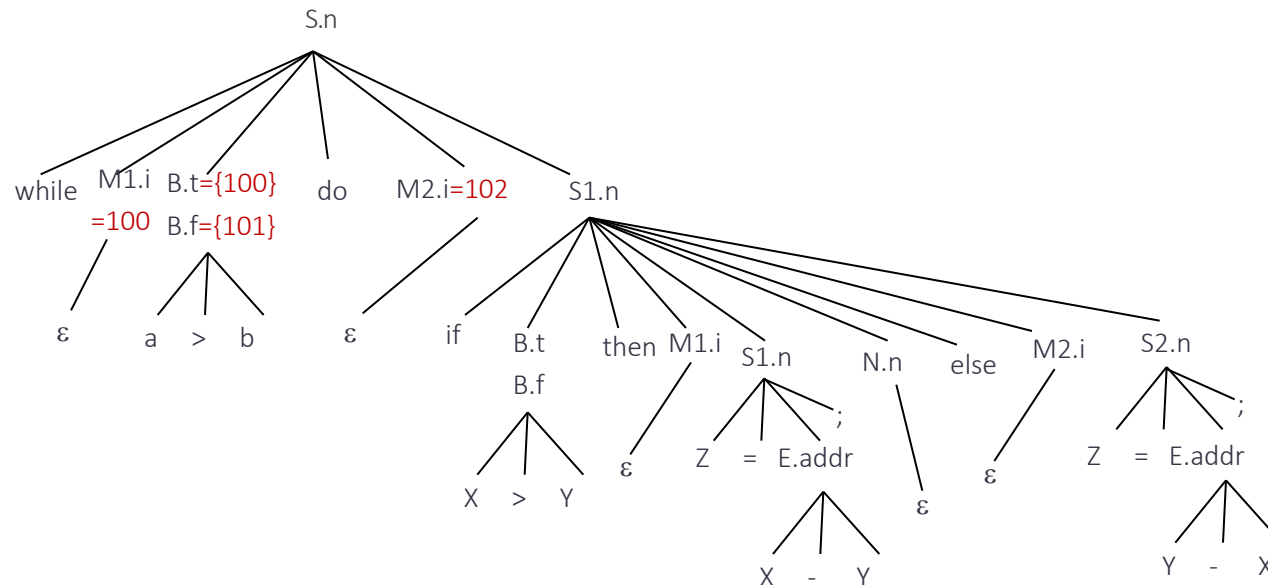
while a>b do if X>Y then Z=X-Y; else Z=Y-X;

$S \rightarrow \text{while } M_1(B) \text{ do } M_2 S_1$   
 $\{ \text{backpatch}(B.\text{truelist}, M_2.\text{instr});$   
 $\text{backpatch}(S_1.\text{nextlist}, M_1.\text{instr});$   
 $S.\text{nextlist} = B.\text{falselist};$   
 $\text{gen}(\text{'goto'}, M_1.\text{instr}) \}$

$M \rightarrow \epsilon \{ \underline{M.\text{instr}} = \underline{\text{nextinstr}}; \}$

100: if a>b goto \_

101: goto\_

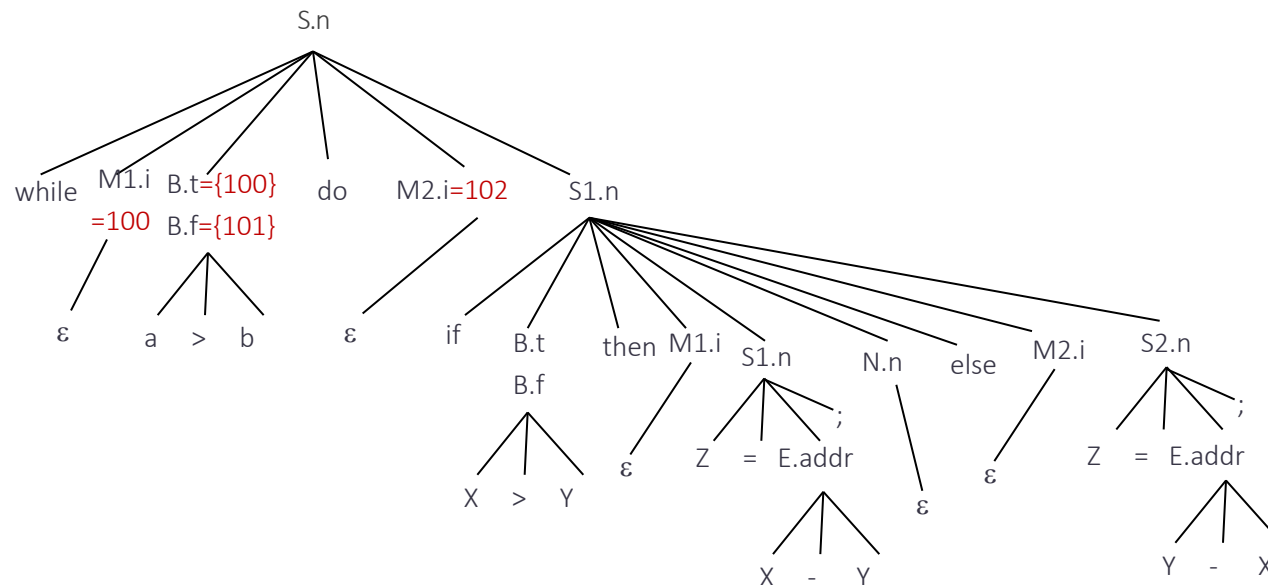




while a>b do if X>Y then Z=X-Y; else Z=Y-X;

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

$M \rightarrow \epsilon \text{ } \{ \text{ } \underline{M.instr} = \underline{nextinstr}; \}$



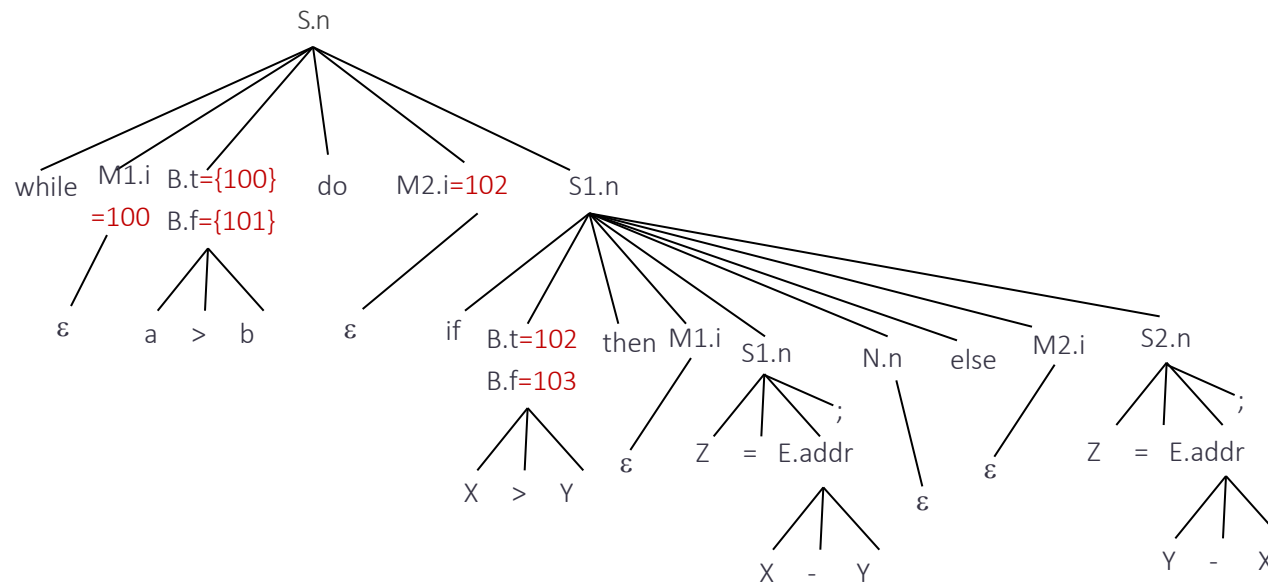
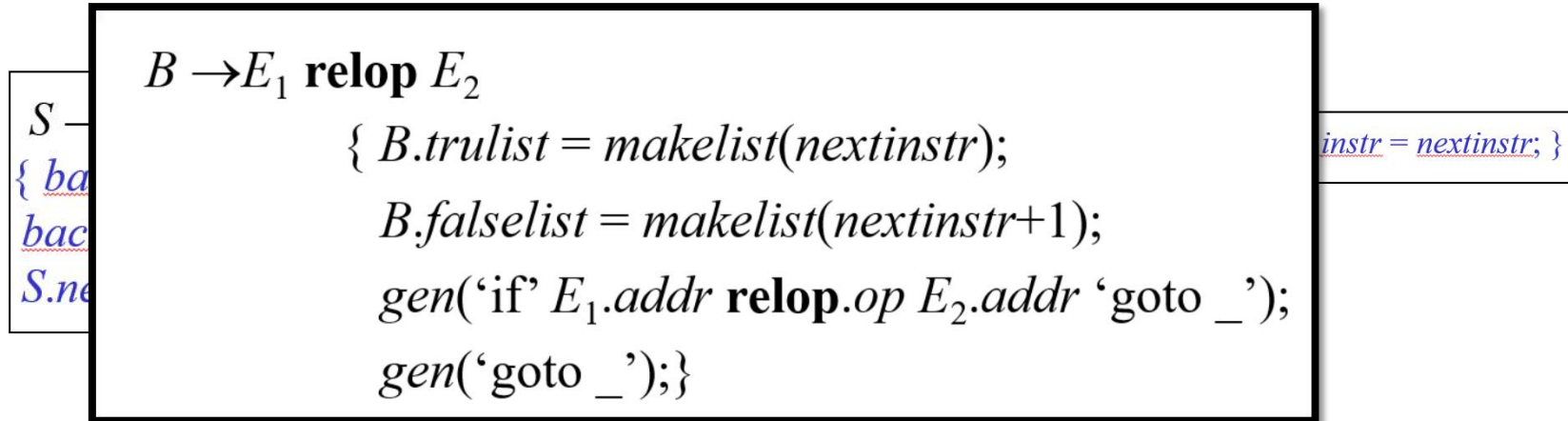
100: if a>b goto \_

101: goto\_

102



while a>b do if X>Y then Z=X-Y; else Z=Y-X;



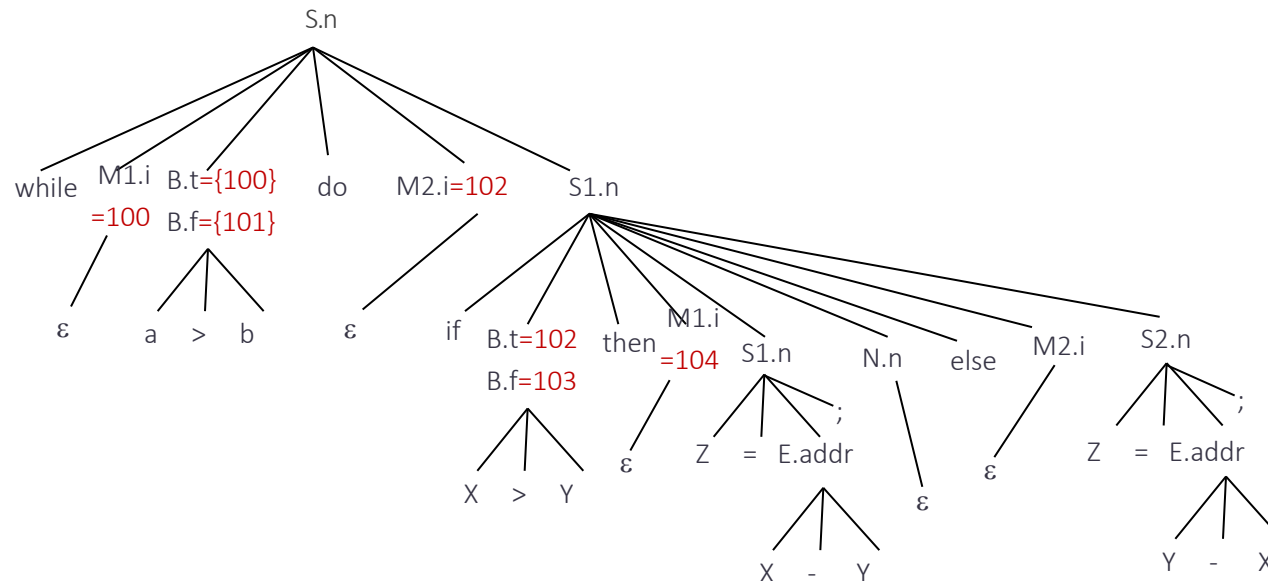
100: if a>b goto \_  
101: goto\_  
102: if X>Y goto \_  
103: goto\_

↓

while a>b do if X>Y then Z=X-Y; else Z=Y-X;

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

$M \rightarrow \epsilon \quad \{ \text{M.instr} = \text{nextinstr}; \}$



100: if a>b goto \_

101: goto \_

102: if X>Y goto \_

103: goto \_

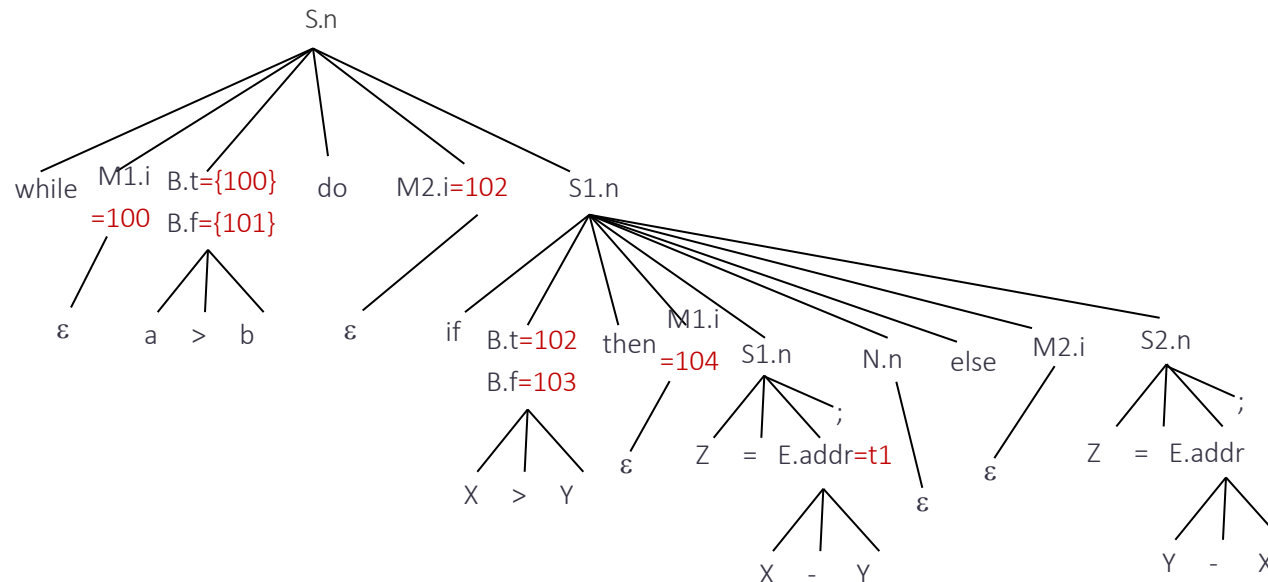
104:



↓

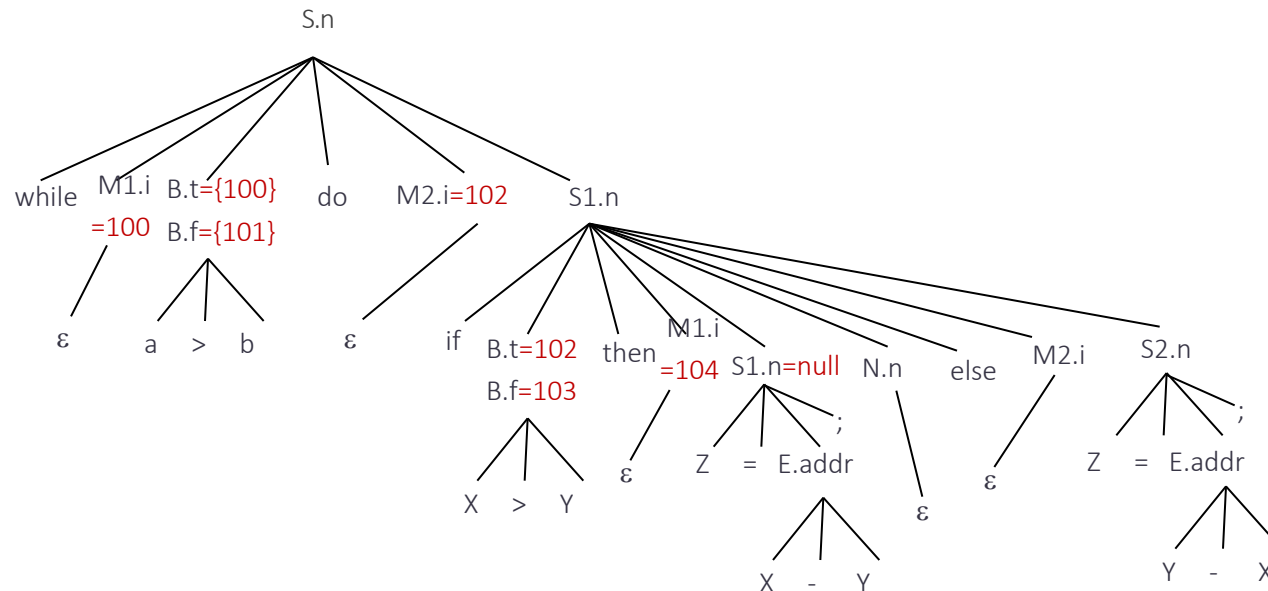
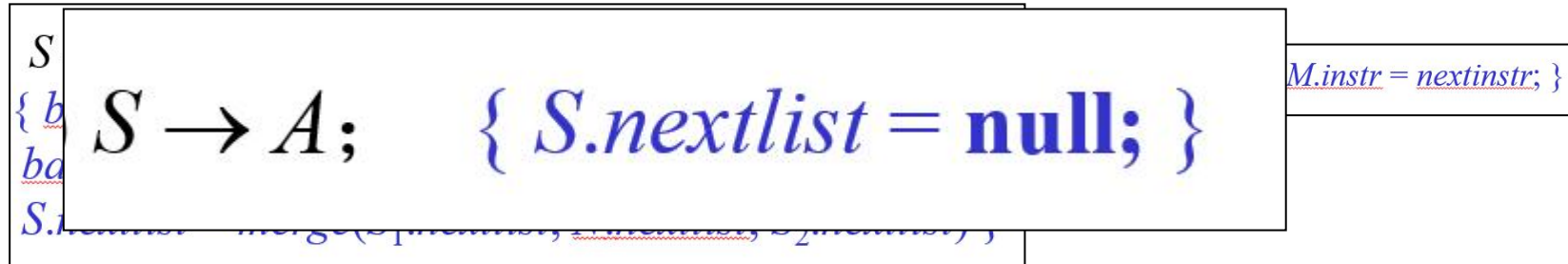
while a>b do if X>Y then Z=X-Y; else Z=Y-X;

$S \rightarrow id = E ;$ $E \rightarrow E_1 - E_2$	<pre>{ gen(top.get(id.lexeme) '=' E.addr); } { E.addr = new Temp()   gen(E.addr '=' E<sub>1</sub>.addr - E<sub>2</sub>.addr); }</pre>
---	---



101: goto\_  
102: if X>Y goto \_  
103: goto\_  
104: t1 = X-Y  
105: Z = t1

while a>b do if X>Y then Z=X-Y; else Z=Y-X;

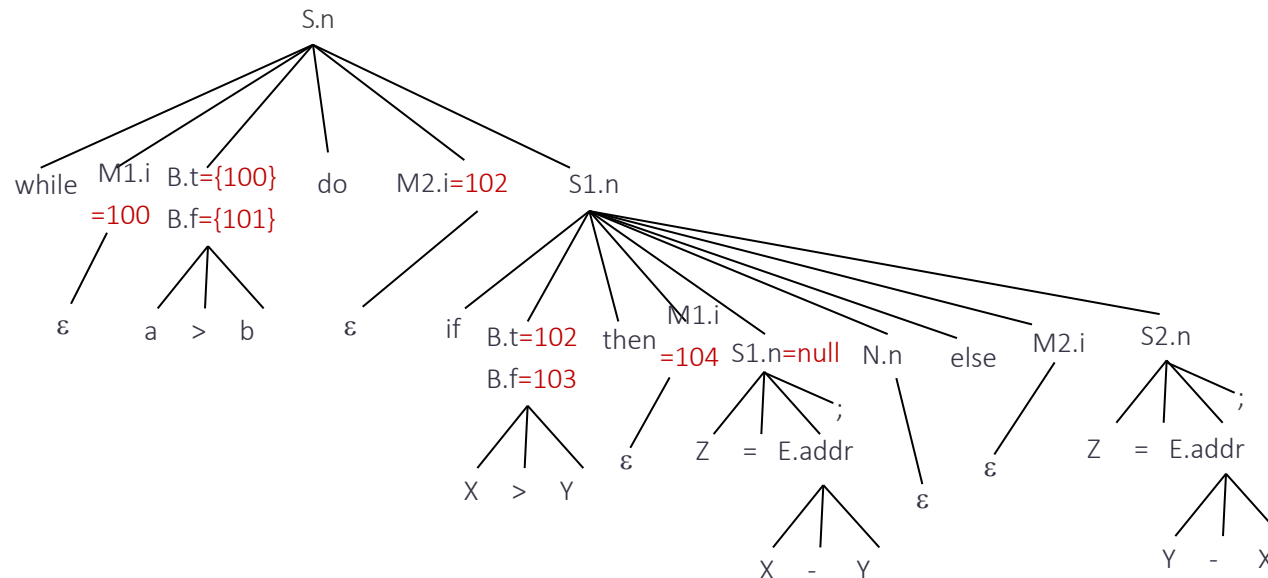


100: if a>b goto \_  
101: goto\_  
102: if X>Y goto \_  
103: goto\_  
104: t1 = X-Y  
105: Z = t1  
106:


  
 while a>b do if X>Y then Z=X-Y; else Z=Y-X;

$S \rightarrow \text{if } (B) \textcolor{red}{M_1} S_1 \textcolor{red}{N} \text{ else } \textcolor{red}{M_2} S_2$   
 $\{ \textcolor{blue}{backpatch}(B.\text{truelist}, M_1.\text{instr});$   
 $\textcolor{blue}{backpatch}(B.\text{falselist}, M_2.\text{instr});$   
 $S.\text{nextlist} = \textcolor{blue}{merge}(S_1.\text{nextlist}, \textcolor{blue}{N}.\text{nextlist}, S_2.\text{nextlist}) \}$

$\textcolor{red}{M} \rightarrow \epsilon \{ \textcolor{blue}{M}.\text{instr} = \textcolor{blue}{nextinstr}; \}$



100: if a>b goto \_  
 101: goto\_  
 102: if X>Y goto \_  
 103: goto\_  
 104: t1 = X-Y  
 105: Z = t1  
 106:

while a>b do if X>Y then Z=X-Y; else Z=Y-X;

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

$M \rightarrow \epsilon \{ \underline{M.instr} = \underline{nextinstr}; \}$

$N \rightarrow \epsilon$   
 $\{ \underline{N.nextlist} = \underline{makelist}(\underline{nextinstr});$   
 $\text{gen}(\text{'goto _'}) \}$

100: if a>b goto \_

101: goto \_

102: if X>Y goto \_

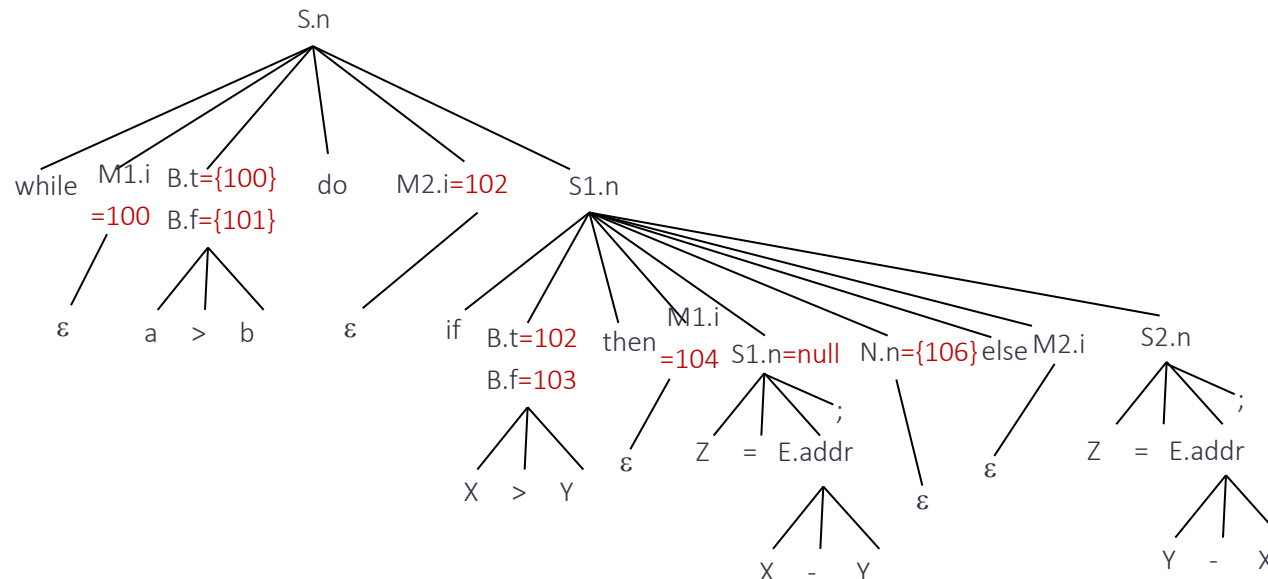
103: goto \_

104: t1 = X-Y

105: Z = t1

106: goto \_

107

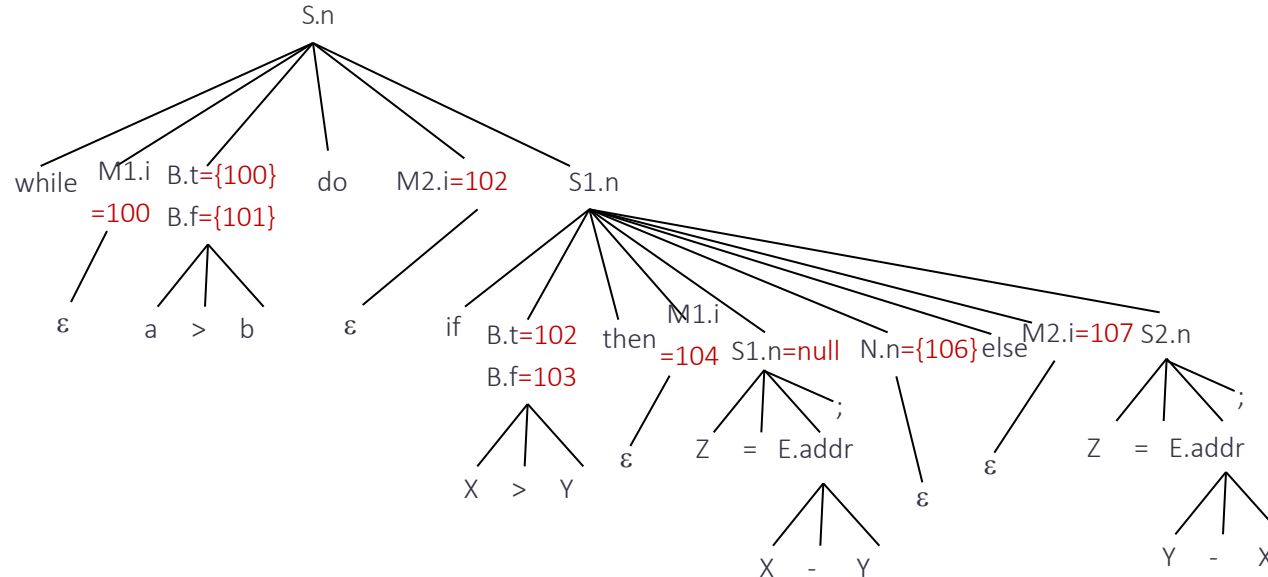


while a>b do if X>Y then Z=X-Y; else Z=Y-X;

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

$M \rightarrow \epsilon \quad \{ \text{M.instr} = \text{nextinstr}; \}$

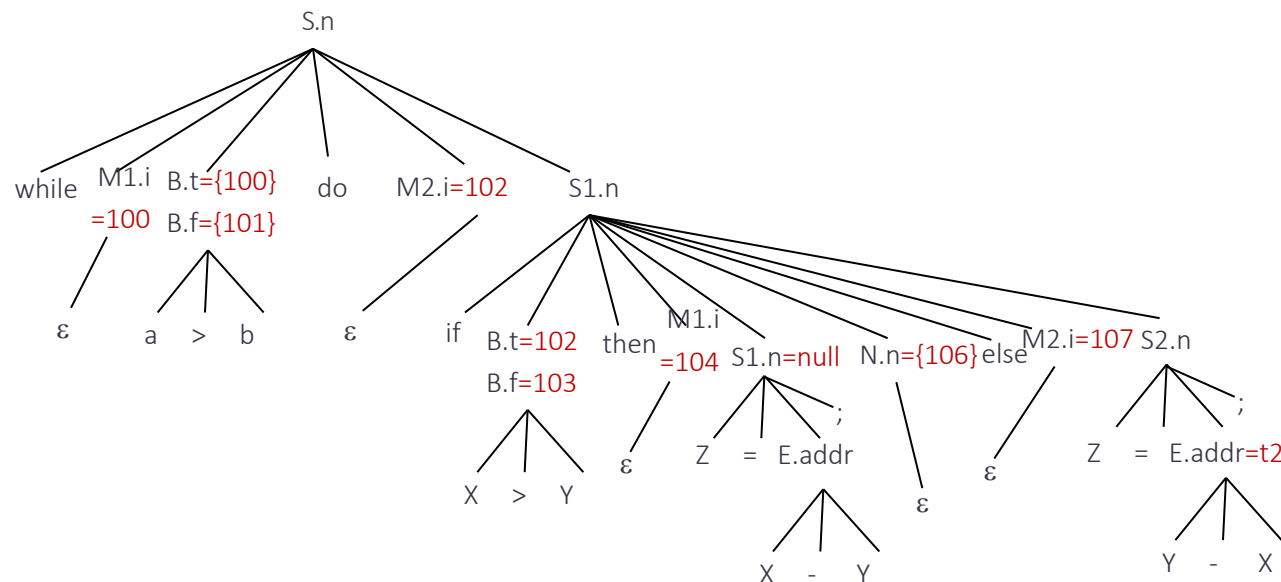
$N \rightarrow \epsilon$   
 $\{ N.\text{nextlist} = \text{makelist}(\text{nextinstr});$   
 $\text{gen}(\text{'goto\_'}) \}$



100: if a>b goto \_  
 101: goto\_  
 102: if X>Y goto \_  
 103: goto\_  
 104: t1 = X-Y  
 105: Z = t1  
 106: goto \_  
 107

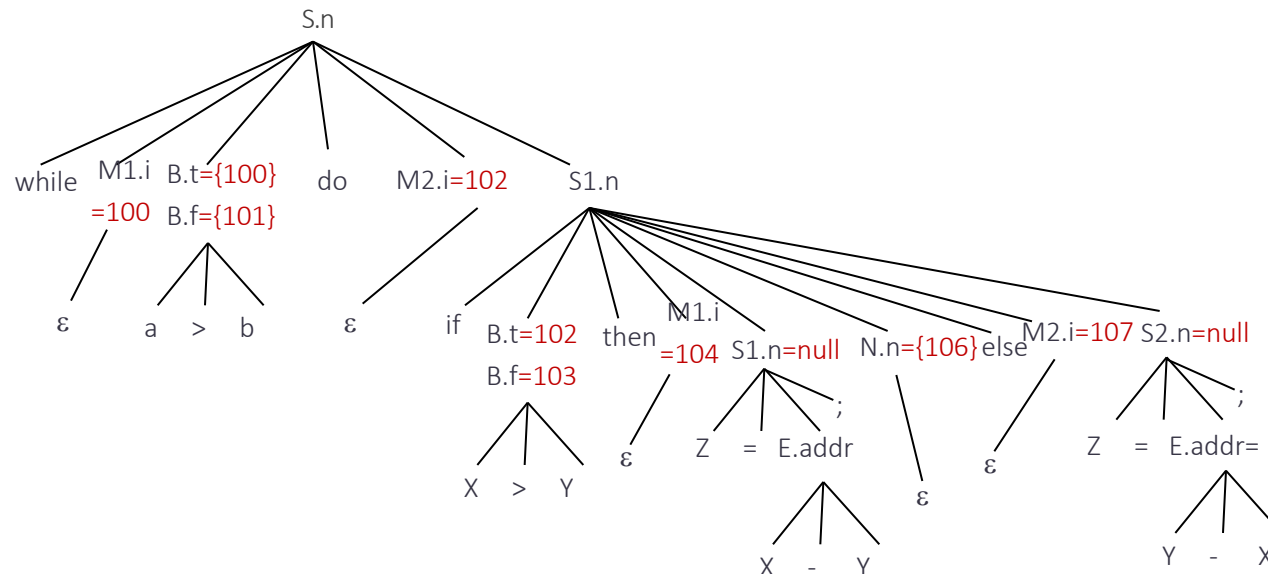
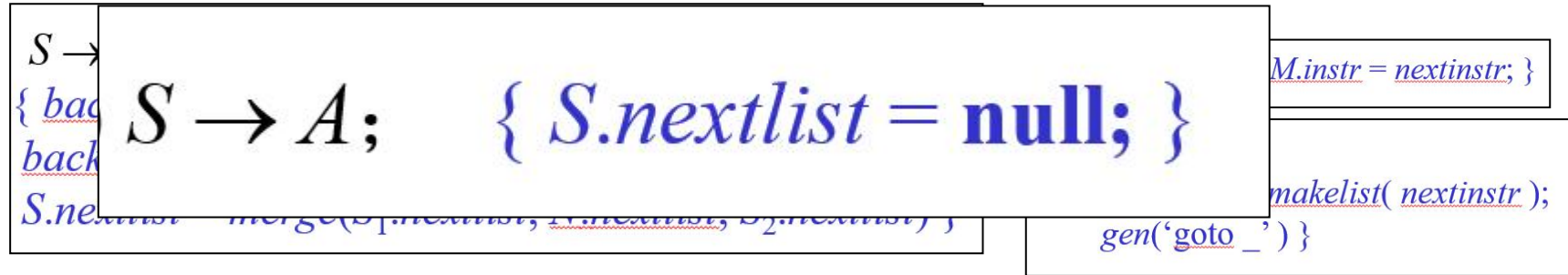
while a>b do if X>Y then Z=X-Y; else Z=Y-X;

$S \rightarrow id = E ;$	{ <u>gen(top.get(id.lexeme) '=' E.addr);</u> }
$E \rightarrow E_1 - E_2$	{ <u>E.addr = new Temp()</u> <u>gen(E.addr '=' E<sub>1</sub>.addr - E<sub>2</sub>.addr);</u> }



101: goto\_  
 102: if X>Y goto \_  
 103: goto\_  
 104: t1 = X-Y  
 105: Z = t1  
 106: goto \_  
 107: t2 = Y-X  
 108: Z = t2

while a>b do if X>Y then Z=X-Y; else Z=Y-X;



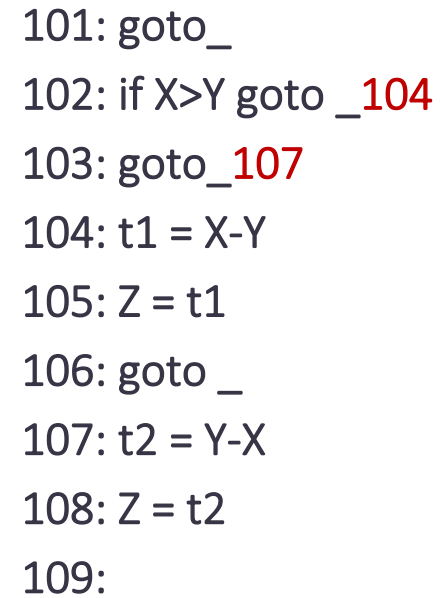
101: goto \_  
 102: if X>Y goto \_  
 103: goto \_  
 104: t1 = X-Y  
 105: Z = t1  
 106: goto \_  
 107: t2 = Y-X  
 108: Z = t2  
 109:



```

N → ε
{ N.nextlist = makelist( nextinstr );
  gen( 'goto _' ) }

```



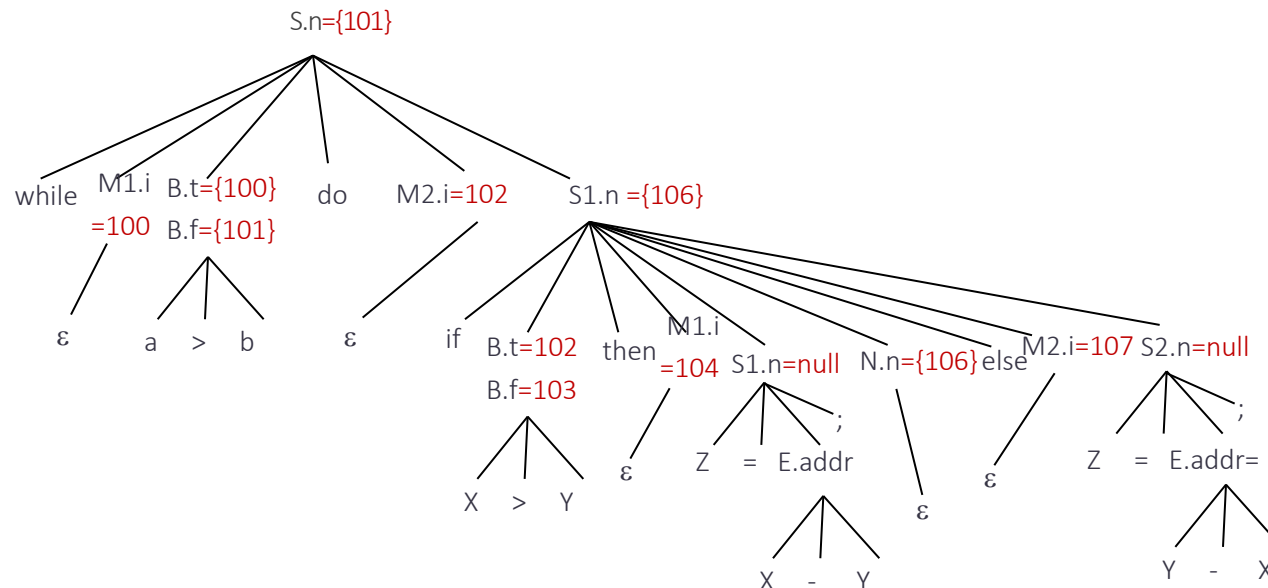


↓

while a>b do if X>Y then Z=X-Y; else Z=Y-X;

$S \rightarrow \text{while } M_1(B) \text{ do } M_2 S_1$   
 $\{ \text{backpatch}(B.\text{truelist}, M_2.\text{instr});$   
 $\text{backpatch}(S_1.\text{nextlist}, M_1.\text{instr});$   
 $S.\text{nextlist} = B.\text{falselist};$   
 $\text{gen}(\text{'goto'}, M_1.\text{instr}) \}$

$M \rightarrow \epsilon \{ \underline{M.instr} = \underline{nextinstr}; \}$



100: if a>b goto \_102

101: goto \_

102: if X>Y goto \_104

103: goto \_107

104: t1 = X-Y

105: Z = t1

106: goto \_100

107: t2 = Y-X

108: Z = t2

109: goto 100

# 学习内容

---

- 6.1 类型检查
- 6.2 中间表示
- 6.3 声明语句
- 6.4 赋值语句
- 6.5 控制流
- 6.6 回填
- **6.7 switch语句**
- 6.8 过程的中间代码



# switch语句

```
switch ( E ) {  
  case  $v_1$ :  $S_1$ ;  
  case  $v_2$ :  $S_2$ ;  
  .....  
  case  $v_{n-1}$ :  $S_{n-1}$ ;  
  default:  $S_n$   
}
```

目标代码结构安排:

```
 $E$ 求值的代码  
 $E$ 的值置于 $t$   
if  $t \neq V_1$  goto  $L_1$   
   $S_1$ 的代码  
  goto next  
 $L_1$ : if  $t \neq V_2$  goto  $L_2$   
   $S_2$ 的代码  
  goto next  
 $L_2$ : ...  
  ...  
 $L_{n-2}$ : if  $t \neq V_{n-1}$  goto  $L_{n-1}$   
   $S_{n-1}$ 的代码  
  goto next  
 $L_{n-1}$ :  $S_n$ 的代码  
next:
```

# switch语句

*E*求值的代码  
*E*的值置于*t*  
if  $t \neq V_1$  goto  $L_1$   
 $S_1$ 的代码  
goto next  
 $L_1$ : if  $t \neq V_2$  goto  $L_2$   
 $S_2$ 的代码  
goto next  
 $L_2$ : ...  
...  
 $L_{n-2}$ : if  $t \neq V_{n-1}$  goto  $L_{n-1}$   
 $S_{n-1}$ 的代码  
goto next  
 $L_{n-1}$ :  $S_n$ 的代码  
next:

```
switch E { t = newtemp(); gen( t '=' E.addr ); }  
case  $V_1$ : {  $L_1$  = newlabel();  
           gen('if' t '!='  $V_1$  'goto'  $L_1$ ); }  
 $S_1$  { next = newlabel(); gen('goto' next); }  
case  $V_2$ : { label( $L_1$ );  $L_2$  = newlabel();  
           gen('if' t '!='  $V_2$  'goto'  $L_2$ ); }  
 $S_2$  { gen('goto' next); }  
      ...  
case  $V_{n-1}$ : { label( $L_{n-2}$ );  $L_{n-1}$  = newlabel();  
              gen('if' t '!='  $V_{n-1}$  'goto'  $L_{n-1}$ ); }  
 $S_{n-1}$  { gen('goto' next); }  
default: { label( $L_{n-1}$ ); }  
 $S_n$  { label(next); }
```

# switch语句

```
switch (  $E$  ) {  
  case  $v_1$ :  $S_1$ ;  
  case  $v_2$ :  $S_2$ ;  
  .....  
  case  $v_{n-1}$ :  $S_{n-1}$ ;  
  default:  $S_n$   
}
```

目标代码结构安排:  
(分支数较多时)

对 $E$ 求值并置于 $t$ 的代码

goto *test*

$L_1$ :  $S_1$ 的代码

goto *next*

$L_2$ :  $S_2$ 的代码

goto *next*

...

$L_{n-1}$ :  $S_{n-1}$ 的代码

goto *next*

$L_n$ :  $S_n$ 的代码

goto *next*

*test*: if  $t = v_1$  goto  $L_1$

if  $t = v_2$  goto  $L_2$

.....

if  $t = v_{n-1}$  goto  $L_{n-1}$

goto  $L_n$

*next*:

# switch语句

对 $E$ 求值并置于 $t$ 的代码

goto test

$L_1$ :  $S_1$ 的代码

goto next

$L_2$ :  $S_2$ 的代码

goto next

...

$L_{n-1}$ :  $S_{n-1}$ 的代码

goto next

$L_n$ :  $S_n$ 的代码

goto next

test: if  $t = v_1$  goto  $L_1$

if  $t = v_2$  goto  $L_2$

.....

if  $t = v_{n-1}$  goto  $L_{n-1}$

goto  $L_n$

next:

```
switch  $E$  {  $t = newtemp()$ ; gen( $t '=' E.addr$ );  
            test = newlabel(); gen('goto' test); }  
case  $V_1$ : {  $L_1 = newlabel()$ ; label( $L_1$ ); map( $V_1, L_1$ ); }  
       $S_1$  { next = newlabel(); gen('goto' next); }  
case  $V_2$ : {  $L_2 = newlabel()$ ; label( $L_2$ ); map( $V_2, L_2$ ); }  
       $S_2$  { gen('goto' next); }  
      ...  
case  $V_{n-1}$ : {  $L_{n-1} = newlabel()$ ; label( $L_{n-1}$ ); map( $V_{n-1}, L_{n-1}$ ); }  
       $S_{n-1}$  { gen('goto' next); }  
default: {  $L_n = newlabel()$ ; label( $L_n$ ); }  
       $S_n$  { gen('goto' next);  
            label(test);  
            gen('if'  $t '=' V_1$  'goto'  $L_1$ );  
            gen('if'  $t '=' V_2$  'goto'  $L_2$ );  
            ...  
            gen('if'  $t '=' V_{n-1}$  'goto'  $L_{n-1}$ );  
            gen('goto'  $L_n$ );  
            label(next);  
          }
```

# 学习内容

---

- 6.1 类型检查
- 6.2 中间表示
- 6.3 声明语句
- 6.4 赋值语句
- 6.5 控制流
- 6.6 回填
- 6.7 switch语句
- 6.8 过程的中间代码



# 过程调用

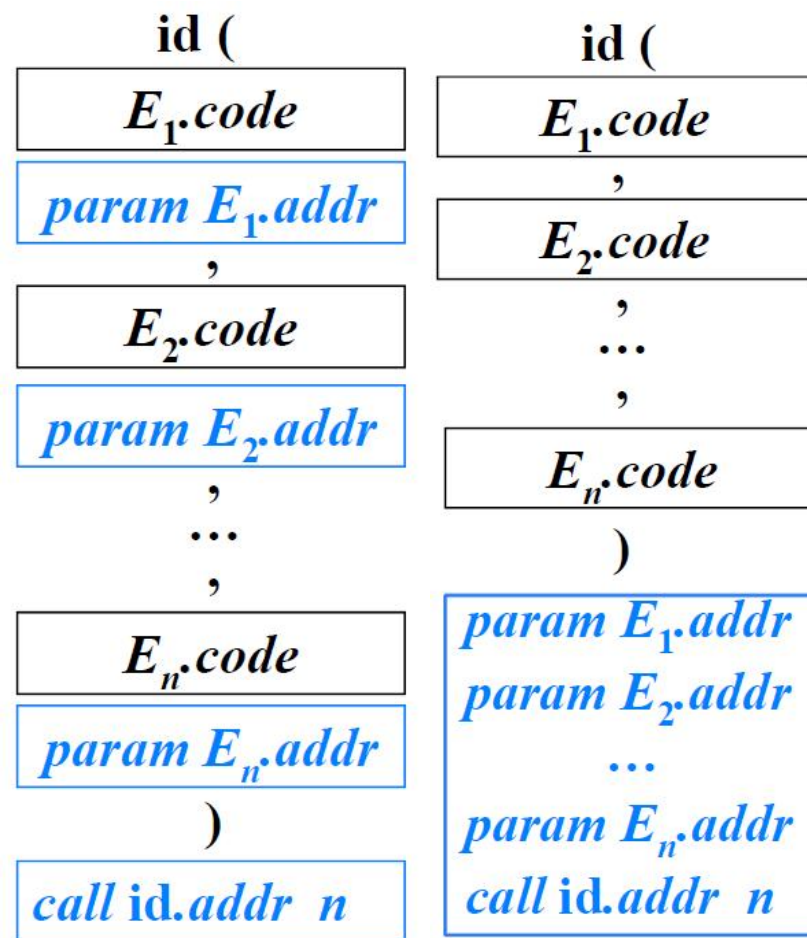
文法

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

$Elist \rightarrow Elist, E$

$Elist \rightarrow E$

$\text{id}(E_1, E_2, \dots, E_n)$





# 过程调用

---

```
 $S \rightarrow \text{call id } (Elist)$   
{   n=0;  
      for q 中的每个 t do  
      {   gen('param' t);  
          n = n+1;  
      }  
      gen('call' id.addr ', ' n);  
}  
 $Elist \rightarrow E$   
{   将 q 初始化为只包含 E.addr; }  
 $Elist \rightarrow Elist_1, E$   
{   将 E.addr 添加到 q 的队尾; }
```