



第六章 中间代码生成（1）

Intermediate Code Generation



主要内容

- 中间代码的表示
 - 抽象语法树 (AST)
 - 有向无环图 (DAG)
 - 三地址代码
- 静态类型检查
- 中间代码生成
 - 类型和声明的翻译
 - 表达式和赋值语句的翻译
 - 控制语句的翻译
 - 回填



中间代码生成简介

- 中间代码是介于源代码和目标代码之间的一种代码形式，它既不依赖于具体的程序语言，也不依赖于具体的目标机。
 - 对不同的程序语言进行编译时，可以采用同一种形式的中间代码。
 - 同一种形式的中间代码，可以变换成不同目标机的目标代码。
 - 在中间代码上可以进行各种不依赖于目标机的优化，这些优化程序可以在不同的程序语言和不同的目标机的编译程序中重复使用。



编译器前端的逻辑结构

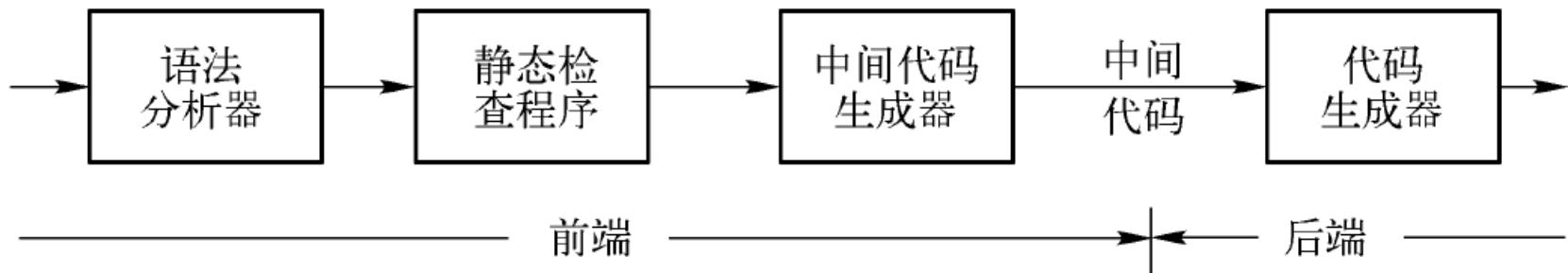


图 6-1 一个编译器前端的逻辑结构

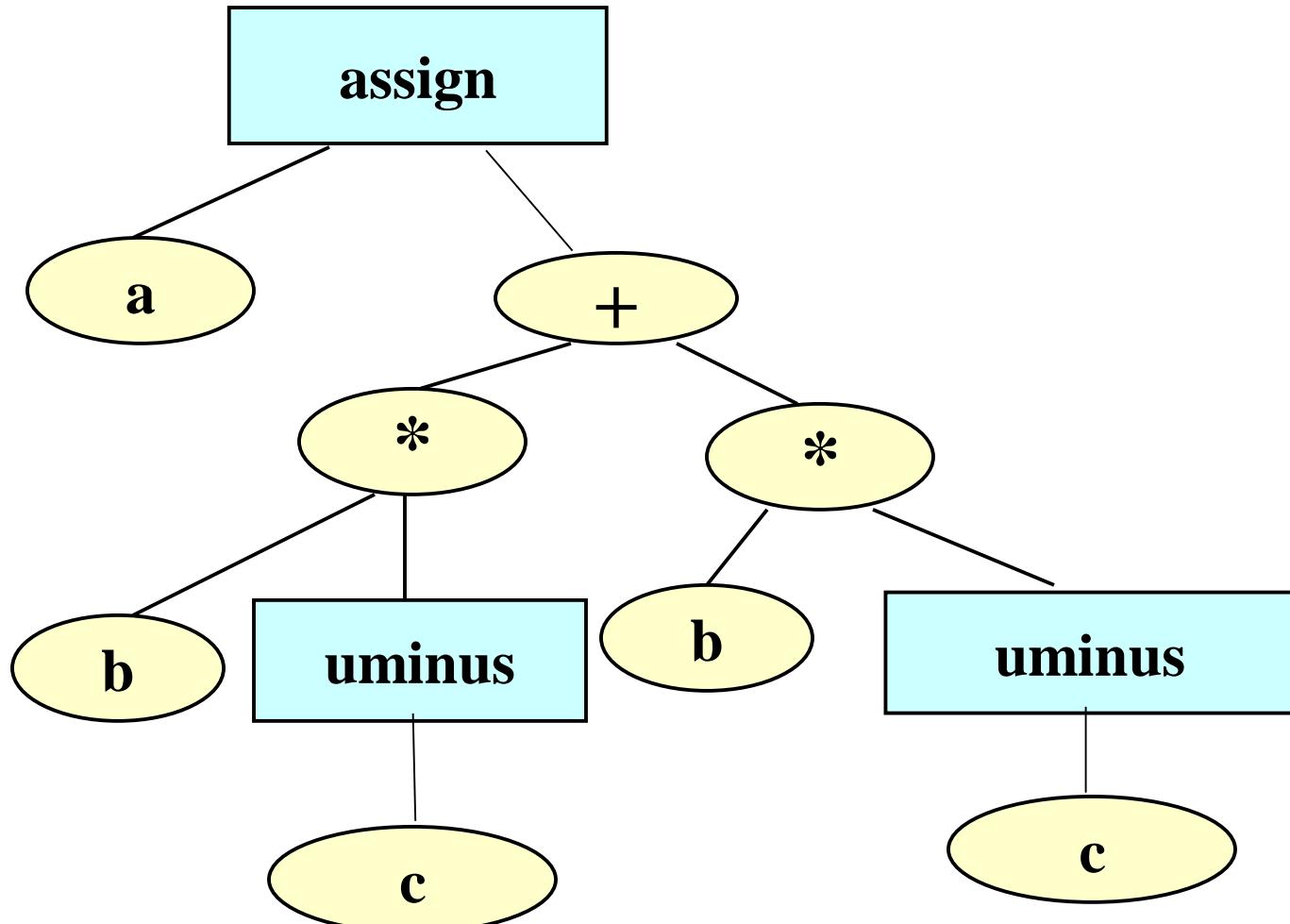


中间代码的表示形式

- 抽象语法树 (AST)
- DAG (Directed Acyclic Graph 有向无环图)
- 后缀式 (也称逆波兰表示)
- 三地址代码

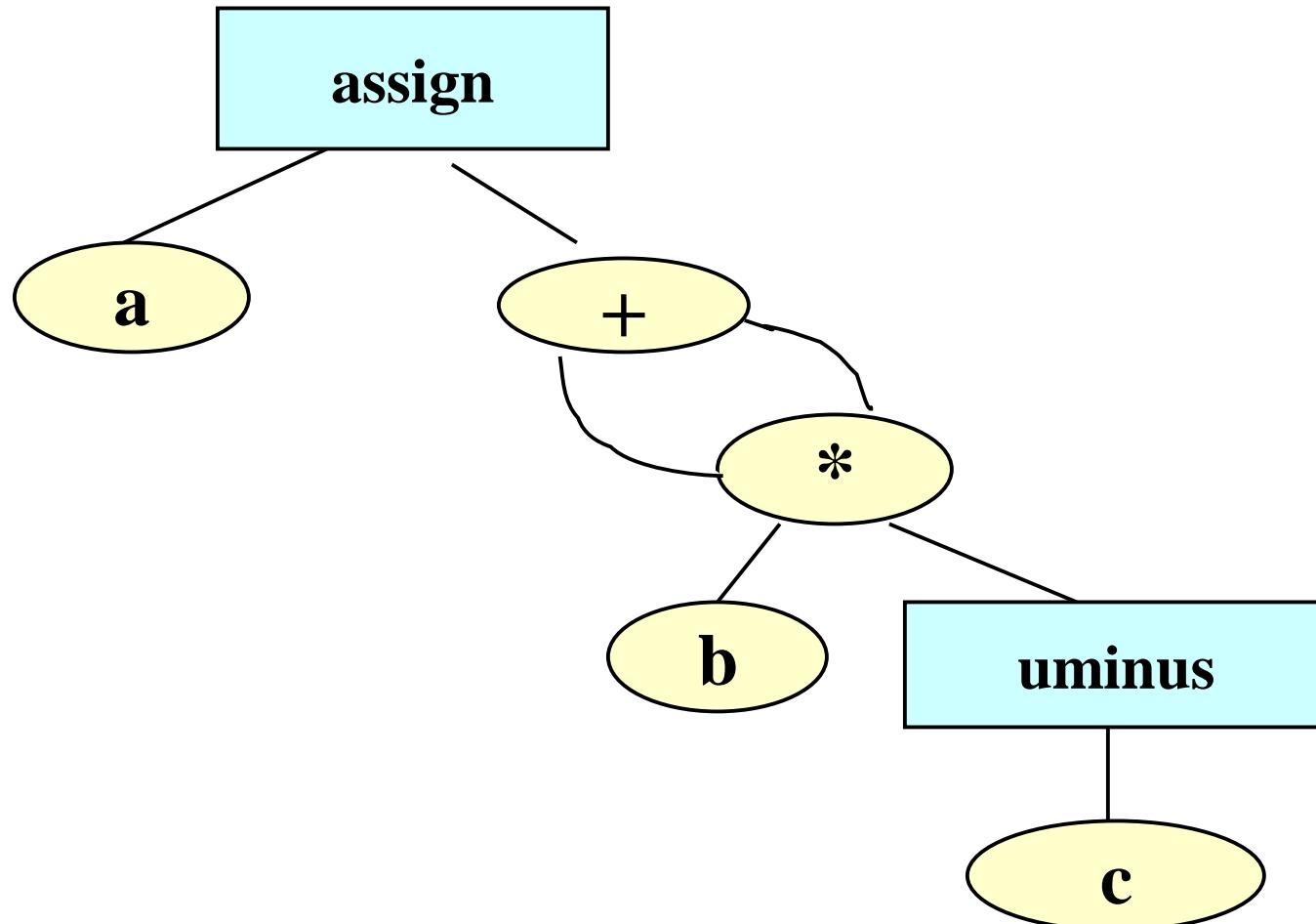


$a = b * -c + b * -c$ 的抽象语法树





$a = b * -c + b * -c$ 的DAG



产生表达式DAG的翻译方案

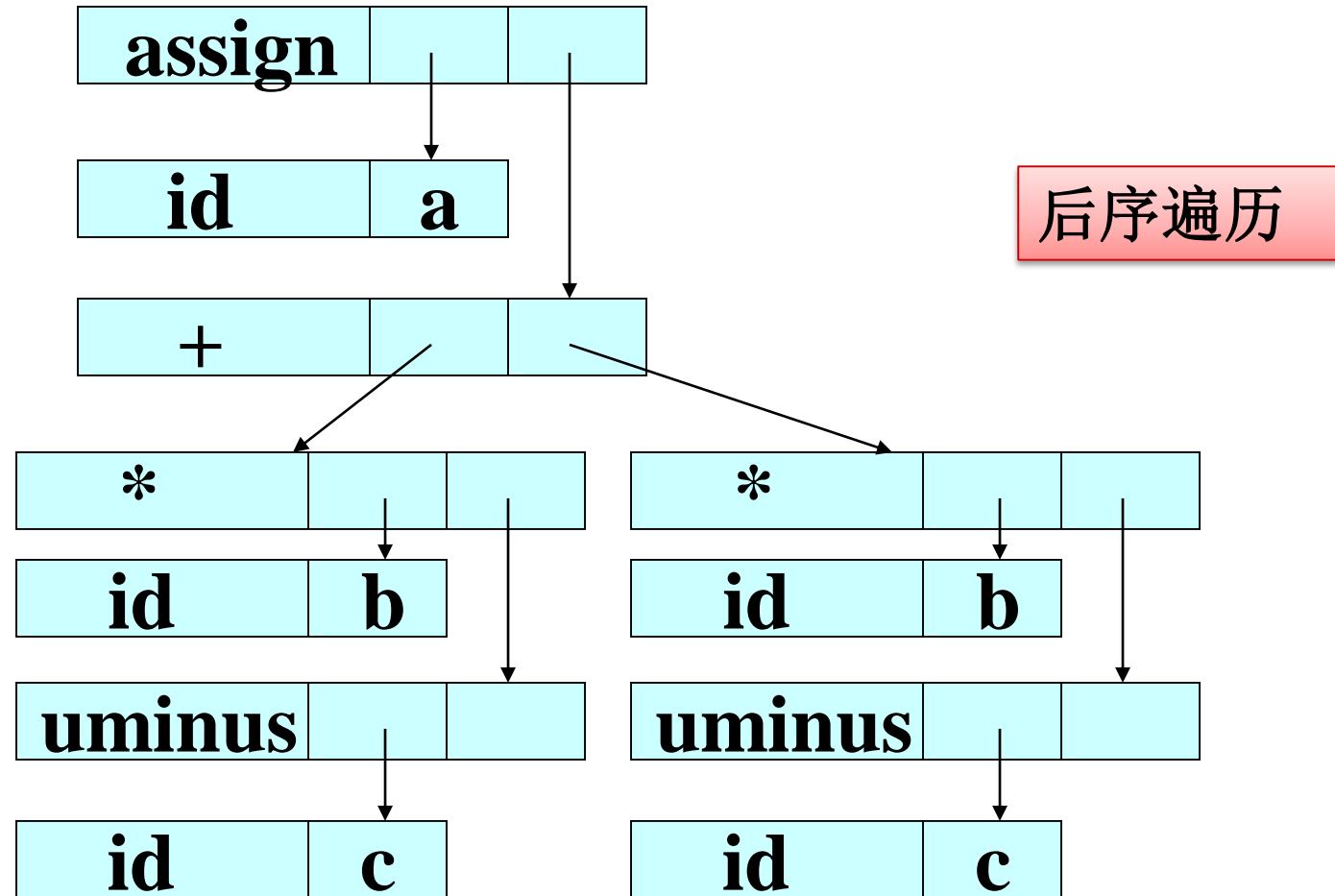


$E \rightarrow E_1 + T$	{ E.node = new Node ('+', E ₁ .node, T.node); }
$E \rightarrow T$	{ E.node = T.node; }
$T \rightarrow T_1 * F$	{ T.node = new Node ('*', T ₁ .node, F.node); }
$T \rightarrow F$	{ T.node = F.node; }
$F \rightarrow (E)$	{ F.node = E.node; }
$F \rightarrow id$	{ F.node = new Leaf(ID, id.name); }
$F \rightarrow num$	{ F.node = new Leaf(NUM, num.val); }

与抽象语法树构造的区别：

- 每次调用Leaf()和Node()的构造函数时，要检查是否已存在相同结构的结点，如果存在，则返回找到的已有结点，否则构造新结点。

后缀式: a b c uminus * b c uminus * + assign



•为什么用后缀式作为中间代码的表示，而不用中缀式？



三地址代码

$$a = b^* - c + b^* - c$$

```
t1 = uminus c  
t2 = b * t1  
t3 = uminus c  
t4 = b * t3  
t5 = t2 + t4  
a = t5
```



三地址代码的形式

- 基本形式: $x = y \text{ op } z$
- 特殊形式:
 1. $x = \text{op } y$ op是一元运算符,如一元减、逻辑非等
 2. $x = y$ 复制指令
 3. **goto L** 转移到L处的无条件转移语句
 4. **if x goto L** 或 **if False x goto L**
 - 仅当 x 为真 (或假) 时转移到L处
 5. **if x ROP y goto L** 条件转移语句
 - 仅当 $x \text{ ROP } y$ 成立时转移到L处
 - ROP是关系运算符 <、<=、>、>= 等



三地址代码的形式 (cont.)

6. param x_1

param x_2

....

param x_n

call p, n

- 调用过程p的过程调用语句：
 - **param x_i** 表示n个实在参数
 - **call p, n** 表示调用过程p并且有n个实在参数



三地址代码的形式 (cont.)

6. $x = y[z]$ 表示把数组元素的值赋给x; y和z分别表示数组元素地址的不变部分和可变部分
7. $x[y] = z$ 表示把z的值赋给数组元素; x和y分别表示数组元素地址的不变部分和可变部分
8. $x = \&y$ 表示把y的地址赋给x
9. $x = *y$ 表示把y值为地址的存储空间的值赋给x
10. $*x = y$ 表示把y值赋给x值为地址的存储空间



三地址代码的例子

□ 语句

■ **do i = i + 1; while (a[i]<v);**

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

a) 符号标号

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

b) 位置号



三地址代码的具体实现

1. 四元式 (quadruple)

op arg1 arg2 result

2. 三元式 (triple)

op arg1 arg2

3. 间接三元式 (indirect triple)

间接码表 + 三元式表



四元式表示

- 四元式：可以实现为纪录（或结构）
 - 格式（字段）：**op arg1 arg2 result**
 - **op**: 运算符的内部编码
 - **arg1, arg2, result**是地址
 - $x=y+z$: + y z x
- 单目运算符不使用**arg2**
- **param**运算不使用**arg2**和**result**
- 条件转移/非条件转移将目标标号放在**result**字段



例：三地址代码的四元式表示

语句 $a = b * -c + b * -c$ 的表示方法：

	op	arg1	arg2	result
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	assign	t_5		a



三元式表示

- 三元式 (triple) op arg1 arg2
 - 使用三元式的位置来引用三元式的运算结果
- $x[i] = y$ 需要拆分为两个三元式
 - 先求 $x[i]$ 的地址，然后再赋值
- $x=y \text{ op } z$ 需要拆分为 (这里?是语句编号)
 - (?) op y z
 - = x ?
- 存在的问题：在优化时经常需要移动/删除/添加三元式，导致三元式位置的移动。



例：三地址代码的三元式表示

语句 $a = b * -c + b * -c$ 的表示方法：

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

注：三元式中使用指向三元式语句的指针



三地址代码的间接三元式表示

- 用一个单独的列表表示三元式的执行顺序
 - 语句的移动仅改变左边的语句表
- 例：语句 $a = b * -c + b * -c$ 的表示方法：

statement	
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

op	arg1	arg2
uminus	c	
*	b	(14)
uminus	c	
*	b	(16)
+	(15)	(17)
assign	a	(18)



代码优化时的间接三元式

- 对语句 $a = b^* - c + b^* - c$ 的三元式进行优化

	statement		op	arg1	arg2
(0)	(14)	(14)	uminus	c	
(1)	(15)	(15)	*	b	(14)
(2)	(18)	(16)	uminus	e	
(3)	(19)	(17)	*	b	(16)
		(18)	+	(15)	(15)
		(19)	assign	a	(18)



不同表示方法的对比

- 四元式需要利用较多的临时单元, 四元式之间的联系通过临时变量实现;
- 中间代码优化处理时, 四元式比三元式更为方便;
- 间接三元式与四元式同样方便, 两种实现方式需要的存储空间大体相同。



静态单赋值 (SSA)

- SSA (Static Single Assignment) 中的所有赋值都针对不同的变量

```
p = a + b  
q = p - c  
p = q * d  
p = e - p  
q = p + q
```

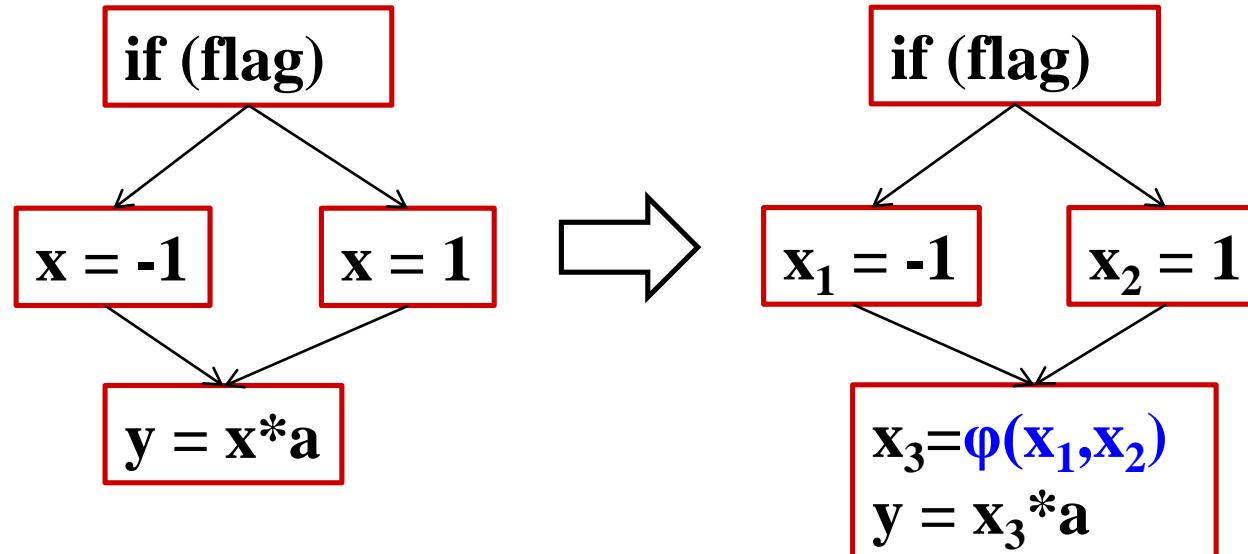


```
p1 = a + b  
q1 = p1 - c  
p2 = q1 * d  
p3 = e - p2  
q2 = p3 + q1
```

SSA形式

静态单赋值 (SSA) : Phi函数

- 对于同一个变量在不同路径中定值的情况，
可以使用 φ 函数来合并不同的定值
- 例： if (flag) $x = -1$; else $x = 1$; $y = x * a$



SSA形式



类型和声明

□ 类型检查(Type Checking)

- 利用一组规则来检查运算分量的类型和运算符的预期类型是否匹配

□ 类型信息的作用

- 检查错误、确定名字需要的内存空间、计算数组元素的地址、类型转换、选择正确的运算符

□ 主要问题

- 如何确定名字的类型？
- 如何确定变量的存储空间布局（相对地址）？



类型化的语言

- **类型化的语言**: 变量都被给定类型的语言
 - 表达式、语句等语法构造的类型都是可以静态确定的
 - 例如, 类型 *boolean* 的变量 *x* 在程序每次运行时的值只能是布尔值, *not (x)* 总有意义
- **非类型化的语言**: 不限制变量值范围的语言
 - 一个运算可以作用到任意的运算对象, 其结果可能是一个有意义的值、一个错误、一个异常或一个语言未加定义的结果
 - 例如: LISP语言



类型表达式

- **类型表达式(*type expression*)**用来表示源程序中变量、常量、表达式、语句等语言成分的类型。
。 主要可以分为如下的种类：
 - 基本类型： **boolean, char, integer, float, etc.**
 - 类名
 - 数组类型： **array**
 - 记录(结构)类型： **record**
 - 函数类型 →： 从s到t的函数表示为 $s \rightarrow t$
 - 笛卡尔积 (×) : 列表或元组 (例如函数参数)
 - 指针类型
 - 类型表达式的变量



类型表达式的例子

□ C语言的类型

```
struct {  
    int no;  
    char name[20];  
}
```

的类型表达式为：

record ((no × integer) × (name × array (20, char)))



类型等价 (type equivalence)

- **类型等价**: 两个类型的值集合相等并且作用于其上的运算集合相等
 - 类型等价具有对称性
- **类型等价主要可以分为两种:**
 - **按名字等价**: 两个类型名字相同，或者被定义成等价的两个名字
 - **按结构等价**: 两个类型的结构完全相同，但是名字不一定相同
 - 按名字等价一定是按结构等价的



类型等价的例子

```
TYPE int = integer;
TARRAY = ARRAY [1..10] OF integer;
VAR i:integer; j:integer; k:int; x:real;
    r1, r2 : ARRAY [1..10] OF integer;
    r3: ARRAY[1..10] OF integer;
    r4: TARRAY;
    r5: TARRAY;
    r6: ARRAY[1..11] OF integer;
```

- 变量i、j和k的类型是按名字等价的；
- r1和r2的类型是按名字等价的；
- r4和r5的类型名字都是TARRAY，也是按名字等价的；
- r3与r1, r2, r4, r5中的每一个的类型都不是按名字等价的；
- r1, r2, r3, r4, r5 的类型是按结构等价的；
- r6与r1, r2, r3, r4, r5中的任何一个都不是类型等价的。



类型兼容 (Type equivalence)

- 类型兼容是针对某种运算而言，而且类型相容不具有对称性
- 例如：在有的语言中，整型类型对于赋值运算与实型类型相容
 - 即允许把整型的值赋给实型变量，但不允许把实型的值赋给整型变量



声明语句

□ 文法

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

□ 含义

- D 生成一系列声明
- T 生成不同的类型
- B 生成基本类型 int/float
- C 表示分量，生成 $[\text{num}]$ 序列
- 注意 record 中用 D 嵌套表示各个字段的声明
 - 字段声明和变量声明的文法一致



局部变量的存储布局

- 变量的类型可以确定变量需要的内存
 - 即类型的宽度
- 可变大小的数据结构只需要考虑指针
 - 函数的局部变量总是分配在连续的区间
- 给每个变量分配一个相对于这个区间开始处的相对地址
 - 变量的类型信息保存在符号表中



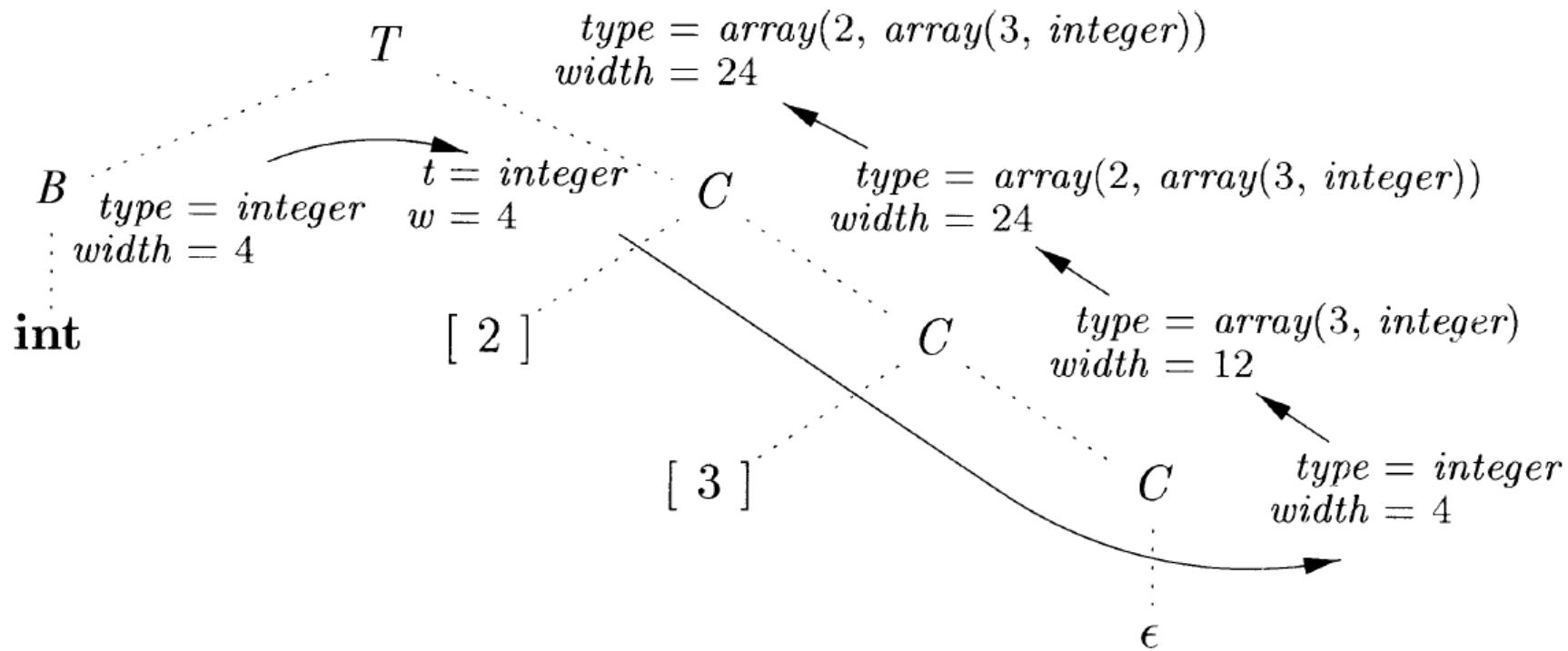
计算T的类型和宽度的SDT

- 综合属性： **type, width**
- 全局变量t和w用于将类型和宽度信息从B传递到C→ε
 - 相当于C的继承属性，因为总是通过拷贝来传递，所以在SDT中只赋值一次。也可以把t和w替换为C.t和C.w

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

SDT运行的例子

□ 输入：int[2][3]

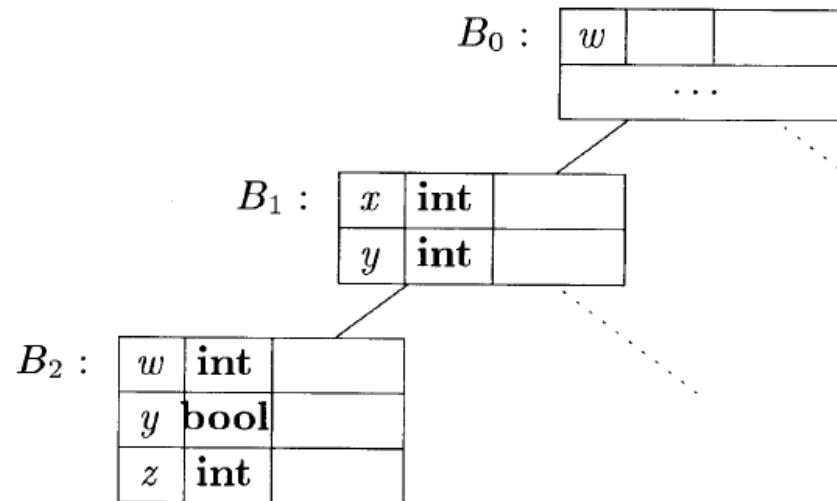




作用域和符号表（参见第2.7节）

- 在具有语句块概念的编程语言中，标识符x在最内层的x声明的作用域中
- 每个作用域对应于一个符号表；多个符号表形成树状结构
- 在语义分析时，通过栈来存放当前符号表及其祖先

```
{ int x1; int y1;  
  { int w2; bool y2; int z2;  
    ...w2...; ...x1...; ...y2...;  
  }  
  ...w0...; ...x1...; ...y1...;
```





声明序列的SDT (1)

- 在处理一个过程/函数时，局部变量应该放到单独的**符号表**中
- 这些变量的内存布局独立
 - 相对地址从0开始
 - 假设变量的放置和声明的顺序相同
- SDT的处理方法
 - 变量offset记录当前可用的相对地址
 - 每“分配”一个变量，offset的值增加相应的值
- **top.put(id.lexeme, T.type, offset)**
 - 在当前**符号表**(位于栈顶)中创建**符号表条目**，记录标识符的类型，偏移量



声明序列的SDT (2)

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

是否适合LR分析？需要怎么修改？

- LL分析中，可以把offset看作D的继承属性
 - D.offset表示D中第一个变量的相对地址
 - $P \rightarrow \{ D.\text{offset} = 0; \} D$
 - $D \rightarrow T \text{id}; \{ D_1.\text{offset} = D.\text{offset} + T.\text{width}; \} D_1$



记录(类)中字段的处理

- $T \rightarrow \text{record} \{ 'D' \}$
- 为每个记录创建单独的符号表
 - 首先创建一个新的符号表，压到栈顶
 - 然后处理对应于字段声明的D，字段都被加入到新符号表中
 - 最后根据栈顶的符号表构造出 **record** 类型表达式；
符号表出栈

```

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

$$D \}' \quad \{ T.type = record(top); T.width = offset; \\ top = Env.pop(); offset = Stack.pop(); \}$$

```



作业

- 11月13日交
- 中间代码的表示
 - 6.1.1
 - 6.2.1, 6.2.2(2)
- 类型和声明语句
 - 6.3.1