Lecture 11: 中间代码生成-I

Xiaoyuan Xie 谢晓园
xxie@whu.edu.cn
计算机学院E301

中间表示形式

后缀表达式

$E \rightarrow E\,opE \mid uopE \mid (E) \mid \text{id} \mid \text{num}$

表达式E的后缀表示可以如下归纳定义：

表达式E	后缀式E'
id	id
num	num
$E_1\,opE_2$	$E_1'\,E_2'\,op$
$uopE$	$E'uop$
(E)	E'

后缀表达式

- **后缀表示不需要括号**
  - (8 – 5) + 2 的后缀表示是8 5 –2 +
- **后缀表示的最大优点是便于计算机处理表达式**

计算栈	输入串
	8 5 –2 +
8	5 –2 +
8 5	–2 +
3	2 +
3 2	+
5	

后缀表达式

- **后缀表示不需要括号**
  - (8 – 5) + 2 的后缀表示是8 5 –2 +
- **后缀表示的最大优点是便于计算机处理表达式**
- **后缀表示也可以拓广到表示赋值语句和控制语句，但很难用栈来描述控制语句的计算**

图形表示

- **语法树是一种图形化的中间表示**
  - 语法树中，公共子表达式每出现一次，就有一个对应的子树
- **有向无环图也是一种中间表示---有向无环图(Directed Acyclic Graph, DAG)**
  - 能够指出表达式中的公共子表达式，更简洁地表示表达式

DAG构造

- **可以用和构造抽象语法树一样的SDD来构造**

	PRODUCTION	SEMANTIC RULES
1)	$E \to E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \to E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \to T$	$E.node = T.node$
4)	$T \to (E)$	$T.node = E.node$
5)	$T \to \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \to \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

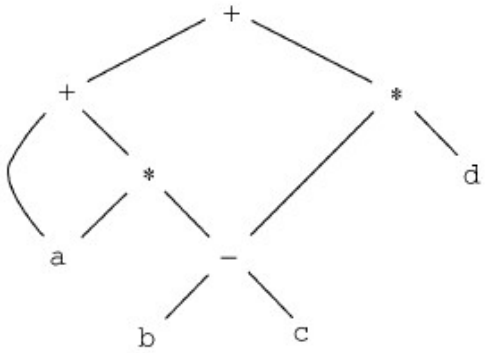


图 6-3　表达式 a + a * (b - c) + (b - c) * d 的 DAG

DAG构造

- **不同的处理**
  - 在函数Leaf和Node每次被调用时，构造新节点前先检查是否已存在同样的节点，如果已经存在，则返回这个已有的节点
- **构造过程示例**

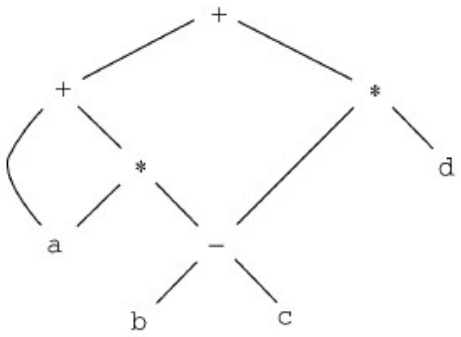


图 6-3　表达式 a + a * (b - c) + (b - c) * d 的 DAG

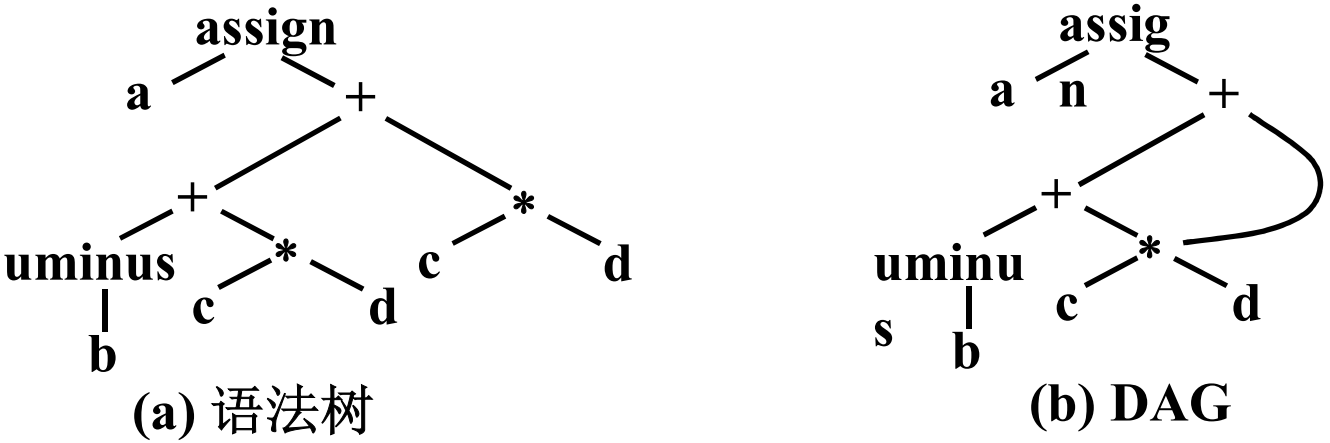
1) $p_1 = Leaf(\mathbf{id}, entry\text{-}a)$
2) $p_2 = Leaf(\mathbf{id}, entry\text{-}a) = p_1$
3) $p_3 = Leaf(\mathbf{id}, entry\text{-}b)$
4) $p_4 = Leaf(\mathbf{id}, entry\text{-}c)$
5) $p_5 = Node('-', p_3, p_4)$
6) $p_6 = Node('*', p_1, p_5)$
7) $p_7 = Node('+', p_1, p_6)$
8) $p_8 = Leaf(\mathbf{id}, entry\text{-}b) = p_3$
9) $p_9 = Leaf(\mathbf{id}, entry\text{-}c) = p_4$
10) $p_{10} = Node('-', p_3, p_4) = p_5$
11) $p_{11} = Leaf(\mathbf{id}, entry\text{-}d)$
12) $p_{12} = Node('*', p_5, p_{11})$
13) $p_{13} = Node('+', p_7, p_{12})$

图 6-5　图 6-3 所示的 DAG 的构造过程

DAG构造

构造赋值语句语法树的语法制导定义

修改构造结点的函数可生成有向无环图



a = (–b + c*d) + c*d的图形表示

DAG构造

构造赋值语句语法树的语法制导定义

修改构造结点的函数可生成有向无环图

产 生 式	语 义 规 则
$S \rightarrow \mathrm{id} = E$	*S.nptr* = *mkNode*('assign', *mkLeaf* (id, id.*entry*), *E.nptr*)
$E \rightarrow E_1 + E_2$	*E.nptr* = *mkNode*('+', E_1.*nptr*, E_2.*nptr*)
$E \rightarrow E_1 * E_2$	*E.nptr* = *mkNode*('*', E_1.*nptr*, E_2.*nptr*)
$E \rightarrow -E_1$	*E.nptr* = *mkUNode*('uminus', E_1.*nptr*)
$E \rightarrow (E_1)$	*E.nptr* = E_1.*nptr*
$F \rightarrow \mathrm{id}$	*E.nptr* = *mkLeaf* (id, id.*entry*)

三地址代码

- **每条指令右侧最多有一个运算符**
  - 一般情况可以写成x = y op z
- **允许的运算分量（地址）最多3个**
  - 名字：源程序中的名字作为三地址代码的地址
  - 常量：源程序中出现或生成的常量
  - 编译器生成的临时变量

三地址代码

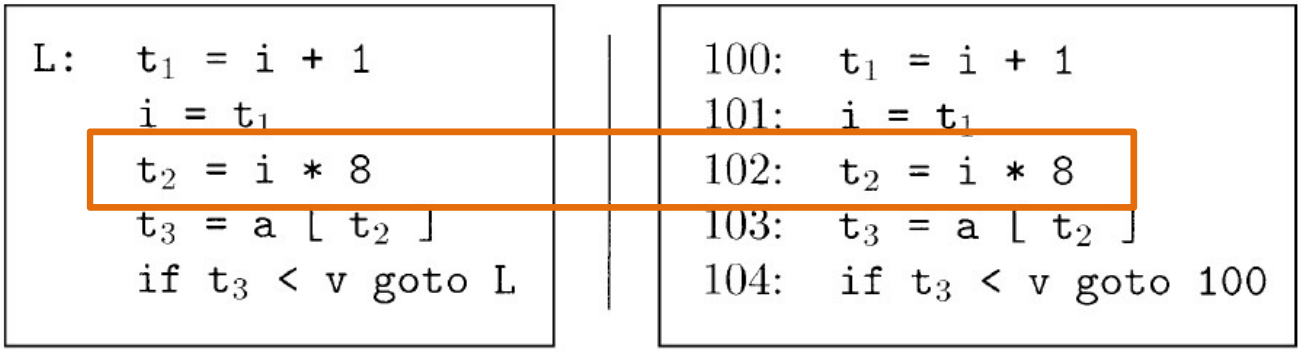
- **指令集合（1）**
  - 运算/赋值指令：x=y op z　　x = op y
  - 复制指令：x=y
  - 无条件转移指令：goto L
  - 条件转移指令：if x goto L　　if False x goto L
  - 条件转移指令：if x relop y goto L

三地址代码

- **指令集合（2）**
  - 过程调用/返回 p(x1, x2, …, xn)
    - param x1　　//设置参数
    - param x2
    - …
    - param xn
    - call p, n　　//调用子过程p，n为参数个数
  - 带下标的复制指令：x=y[i]　　x[i]=y
    - 注意：i表示离开数组位置第i个字节，而不是数组的第i个元素
  - 地址/指针赋值指令：
    - x=&y　　x=*y　　*x=y

三地址代码实例

- **语句**
  - do i = i + 1; while (a[i]<v) ;



a) 符号标号　　b) 位置号

三地址代码实例

- **三地址代码是语法树或DAG的一种线性表示**
  - 例：a = (–b + c∗d) + c∗d

语法树的代码

$t_1 = -b$

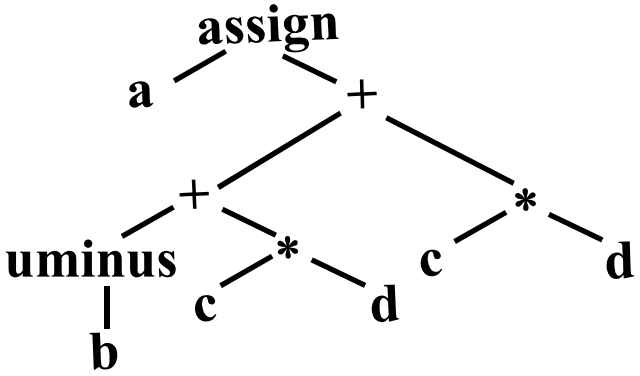
$t_2 = c * d$

$t_3 = t_1 + t_2$

$t_4 = c * d$

$t_5 = t_3 + t_4$

$a = t_5$



三地址代码实例

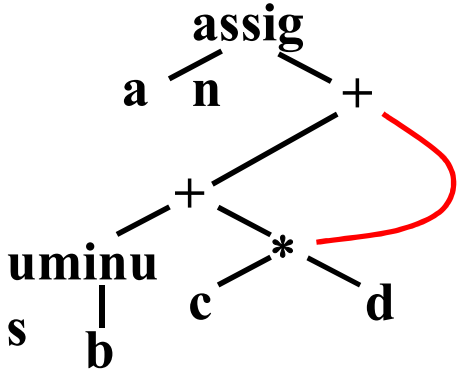
- **三地址代码是语法树或DAG的一种线性表示**
  - 例　　a = (–b + c∗d) + c∗d

语法树的代码

$t_1 = -b$
$t_2 = c * d$
$t_3 = t_1 + t_2$
$t_4 = c * d$
$t_5 = t_3 + t_4$
$a = t_5$

DAG的代码

$t_1 = -b$
$t_2 = c * d$
$t_3 = t_1 + t_2$
$t_4 = t_3 + t_2$
$a = t_4$



三地址指令的四元式表示方法

- **在实现时，可以使用四元式/三元式/间接三元式来表示三地址指令**
- **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**

三地址指令的四元式表示方法

- **例：赋值语句：a=b* -c + b* -c**

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

a) 三地址代码

	op	arg_1	arg_2	*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
		...		

b) 四元式

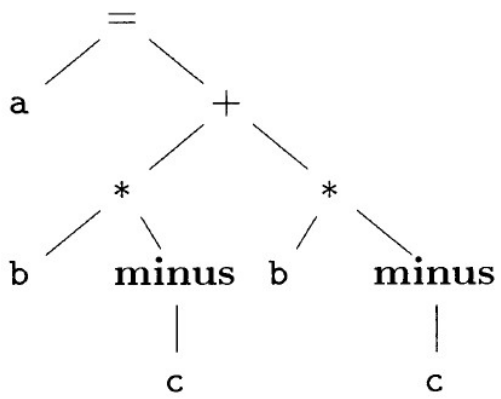
图 6-10　三地址代码及其四元式表示

三地址指令的三元式表示方法

- **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
  - 求x[i]的地址，然后再赋值
  - x=y op z需要拆分为（这里？是编号）
    - (?)　　op　　y　　z
    - 　　=　　x　　?

三地址指令的三元式表示方法

- **例：a=b*-c + b * -c**



a) 语法树

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

b) 三元式

图 6-11　$a = b^* - c + b^* - c$ 的表示

问题：在优化时经常需要移动/删除/添加三元式，导致三元式的移动

间接三元式

- 包含了一个指向三元式的指针的列表
- 我们可以对这个列表进行操作，完成优化功能；操作时不需要修改三元式中的参数

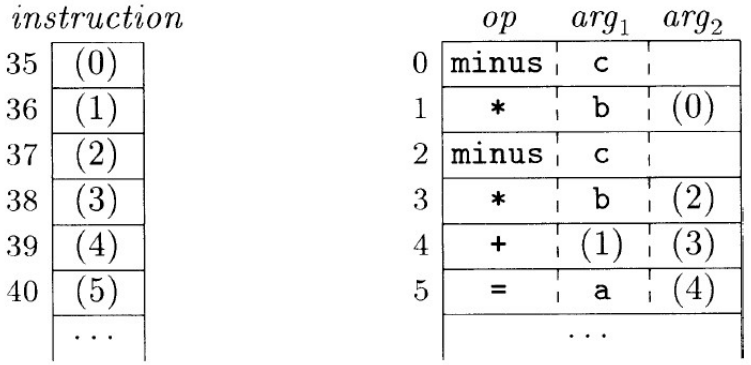


图 6-12　三地址代码的间接三元式表示

静态单赋值（SSA）

- **一种便于某些代码优化的中间表示**
- **和三地址代码的主要区别**
  - 所有赋值指令都是对不同名字的变量的赋值

三地址代码	静态单赋值形式
$p = a + b$	$p_1 = a + b$
$q = p - c$	$q_1 = p_1 - c$
$p = q * d$	$p_2 = q_1 * d$
$p = e - p$	$p_3 = e - p_2$
$q = p + q$	$q_2 = p_3 + q_1$

静态单赋值（SSA）

- **一种便于某些代码优化的中间表示**
- **和三地址代码的主要区别**
  - 所有赋值指令都是对不同名字的变量的赋值
  - 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

    if (flag) x = –1; else x = 1;

    y = x * a;

  改成

    if (flag) x_1 = –1; else x_2 = 1;

    $x_3 = \phi(x_1, x_2)$; //由flag的值决定用x_1还是x_2

类型系统

什么是类型

- **一个程序变量在程序执行期间的值可以设想为有一个范围，这个范围的一个界叫做该变量的类型。**
  - 变量都被给定类型的语言叫做类型化语言（typed language）。
  - 语言若不限制变量值的范围，则被称作未类型化的语言（untyped language ）

类型系统

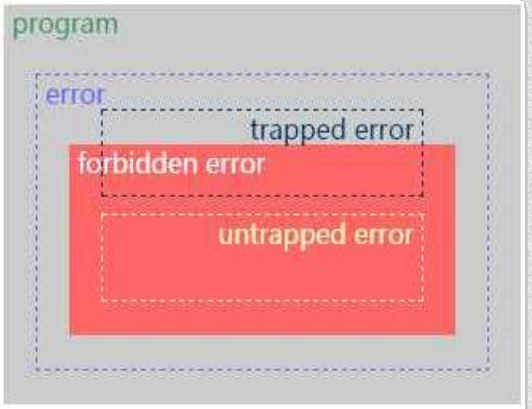
- 类型化语言的类型系统（type system）是该语言的一个组成部分，它始终监视着程序中变量的类型，通常还包括所有表达式的类型。
- 一个类型系统主要由一组定型规则（typing rules）构成，这组规则用来给各种语言构造（程序、语句、表达式等）指派类型。

类型系统

- **程序运行时的执行错误分成两类**
  - 会被捕获的错误（*trapped error*）
    - 例：非法指令错误、非法内存访问、除数为零
    - 引起计算立即停止
  - 不会被捕获的错误（*untrapped error*）
    - 例：下标变量的访问越过了数组的末端
    - 例：跳到一个错误的地址，该地址开始的内存正好代表一个指令序列
    - 错误可能会有一段时间未引起注意

类型系统

- **禁止错误（*forbidden error*）**
  - *untrapped error*集合 + *trapped error*的一个子集
  - 为语言设计类型系统的目标是在排除禁止错误



- **良行为的程序(well-behaved)**
  - A program fragment that will not produce forbidden errors at run time (不同场合对良行为的定义略有区别)
- **安全语言**
  - 任何合法程序都没有`forbidden error`

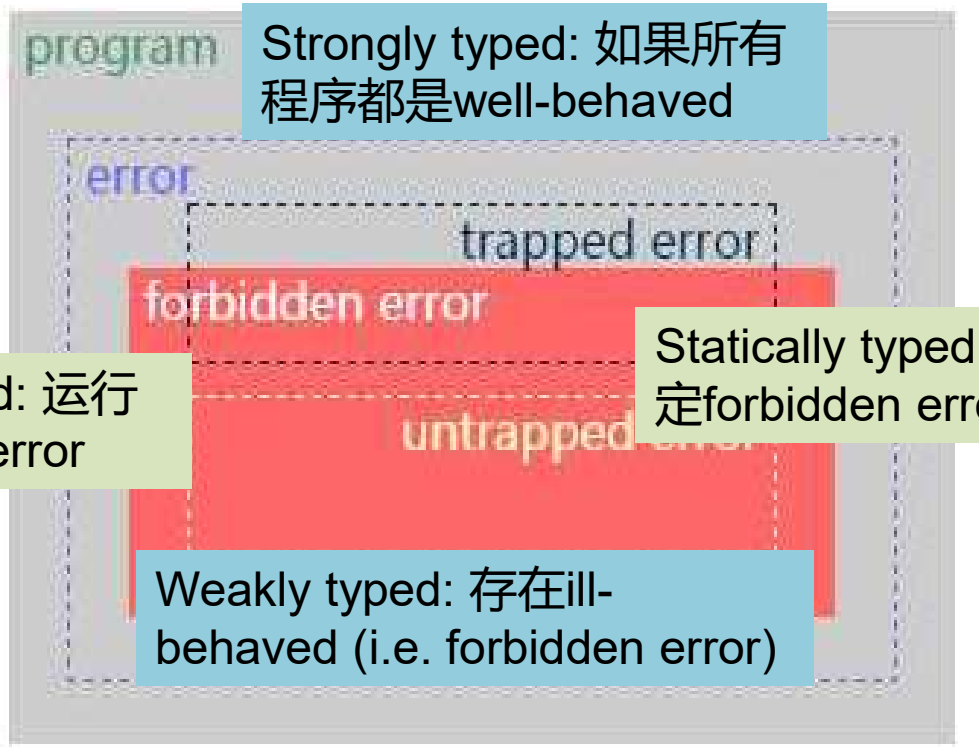
类型系统

- **类型化的语言**
  - 变量都被给定类型的语言：表达式、语句等程序构造的类型都可以静态确定，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
- **未类型化的语言**
  - 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

类型系统

- **显式类型化语言**
  - 类型是语法的一部分
- **隐式类型化的语言**
  - 不存在隐式类型化的主流语言，但可能存在忽略类型信息的程序片段，例如不需要程序员声明函数的参数类型

类型系统


program
error
trapped error
forbidden error
untrapped
Strongly typed: 如果所有程序都是well-behaved
Statically typed: 编译时确定forbidden error
Dynamically typed: 运行时确定forbidden error
Weakly typed: 存在ill-behaved (i.e. forbidden error)

静态类型信息在编译中的作用

- **应用一：静态类型检查** (下周介绍)
  - 编译时确定forbidden error

类型表达式

- **类型本身也有结构，我们使用类型表达式 (type expression)来表示这种结构**
  - 基本类型：Boolean, integer, float, char, void；或
  - 类名；或

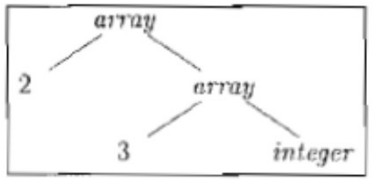


图 6-14　int[2][3]的类型表达式

类型表达式

- **类型本身也有结构，我们使用类型表达式 (type expression)来表示这种结构**
  - 通过将“类型构造算子”作用于类型表达式而得，例如：
    - array[数字，类型表达式]
    - record[字段/类型对的列表]（可以用符号表表示）
    - 函数类型构造算子→ ：参数类型→结果类型
    - 笛卡尔积：s X t: struct { int a[10]; float f;} st 对应于：record((a × array(0..9, int)) × (f × real))
    - 可以包含取值为类型表达式的变量

类型表达式的例子

- **类型例子**
  - 元素个数为3X4的二维数组
  - 数组的元素的记录类型
  - 该记录类型中包含两个字段: x和y,其类型分别是float和integer
- **类型表达式**
  - array[3,
    array[4, record[(x,float),(y,integer)]
    ]
    ]

类型等价

- **不同的语言有不同的类型等价的定义**
- **结构等价**
  - 或者它们是相同的基本类型
  - 或者是相同的构造算子作用于结构等价的类型而得到的。
  - 或者一个类型是另一个类型表达式的名字
- **名等价**
  - 类型名仅仅代表其自身

静态类型信息在编译中的作用

- **应用二：翻译时应用** (后面介绍不同语句翻译的时候会用到)
  - 根据一个名字类型(查表可确定), 编译器即可实现存储空间(相对地址)的布局，包括
    - 确定名字所需内存空间、计算数组元素地址、类型转换、选择正确运算符
    - 绝对地址在运行时才可得；此处指的是相对地址，即，一个名字或某个数据结构分量相对于数据区域开始位置的偏移量(offset)

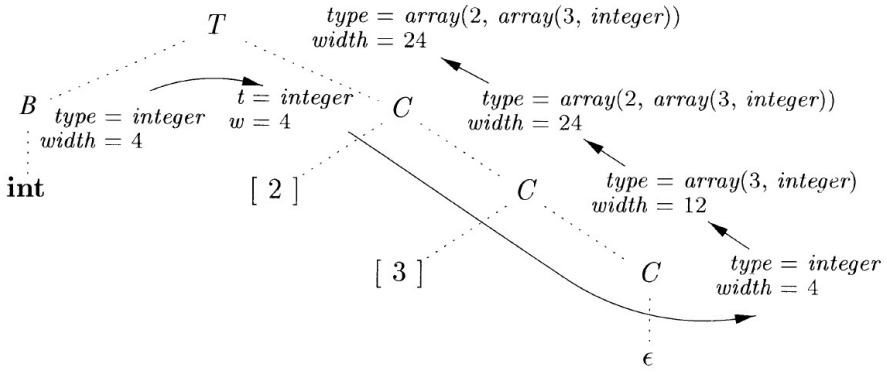
局部变量名的存储布局

- **变量的类型可以确定变量需要的内存**
  - 类型的宽度(width)：指该类型的一个对象所需的存储单元的数量
    - 基本类型 (int, float, char)：整数多个字节；
    - 组合类型 (数组，类)：连续的存储字节块
    - 可变大小的数据结构只需要考虑指针
- **多字节数据对象、函数的局部变量往往被存储在一段连续字节中，并以初始字节的地址作为该数据对象的地址**
  - 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**

局部变量名的存储布局

- **计算类型和宽度的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	$\{\ T.type = C.type;\ T.width = C.width\ \}$
$B \rightarrow$	**int**	$\{\ B.type = integer;\ B.width = 4;\ \}$
$B \rightarrow$	**float**	$\{\ B.type = float;\ B.width = 8;\ \}$
$C \rightarrow$	ϵ	$\{\ C.type = t;\ C.width = w;\ \}$
$C \rightarrow$	[**num**] C_1	$\{\ C.type = array(\mathbf{num}.value,\ C_1.type);$ $C.width = \mathbf{num}.value \times C_1.width;\ \}$



声明语句的翻译

声明

- **考虑例子**
  - 含义
    - D生成一个声明列表
    - T生成不同的类型
    - B生成基本类型int/float
    - C表示分量，生成[num]序列
    - 注意record中也是声明列表，其中字段声明和变量声明的文法一致

$$
\begin{array}{lcl}
D & \rightarrow & T\ \mathbf{id}\ ;\ D \mid \epsilon \\
T & \rightarrow & B\ C \mid \mathbf{record}\ '\{'\ D\ '\}' \\
B & \rightarrow & \mathbf{int} \mid \mathbf{float} \\
C & \rightarrow & \epsilon \mid [\ \mathbf{num}\ ]\ C
\end{array}
$$

除了确定类型和类型宽度，还有什么语义需要处理？符号表中的位置

普通字段声明序列SDT (不考虑作用域问题)

- **计算被声明名字的类型和相对地址**

$P \rightarrow$ {*offset* = 0; }
D
$D \rightarrow T$ **id ;** {*enter* (id.*lexeme*, *T.type*, *offset*);
offset = *offset* + *T.width* }
D_1
$T \rightarrow$ integer {*T.type* = *integer*; *T.width* = 4 }
$T \rightarrow$ real {*T.type* = *real*; *T.width* = 8 }

普通字段声明序列SDT (不考虑作用域问题)

- **计算被声明名字的类型和相对地址**

$P \rightarrow$ {offset = 0; }

D

$D \rightarrow T$ **id** ; {enter (id.lexeme, T.type, offset);
offset = offset + T.width }

D_1

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

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

enter是填表动作

符号表与作用域

- **作用域：一个声明起作用的程序部分称为该声明的作用域。**
  - 过程中出现的名字，如果是在该过程的一个声明的作用域内，那么这个出现称为局部于该过程的；否则叫做非局部的。
  - 局部和非局部的区分也适用于其他任何可包含声明的语法结构。
  - 作用域是名字声明的一个性质
- **需要多张符号表来实现作用域信息的保存**

符号表与作用域

- **考虑嵌套变量的声明**

  $P \rightarrow D;\ S$

  $D \rightarrow D\ ;\ D \mid \text{id}\ :\ T \mid$

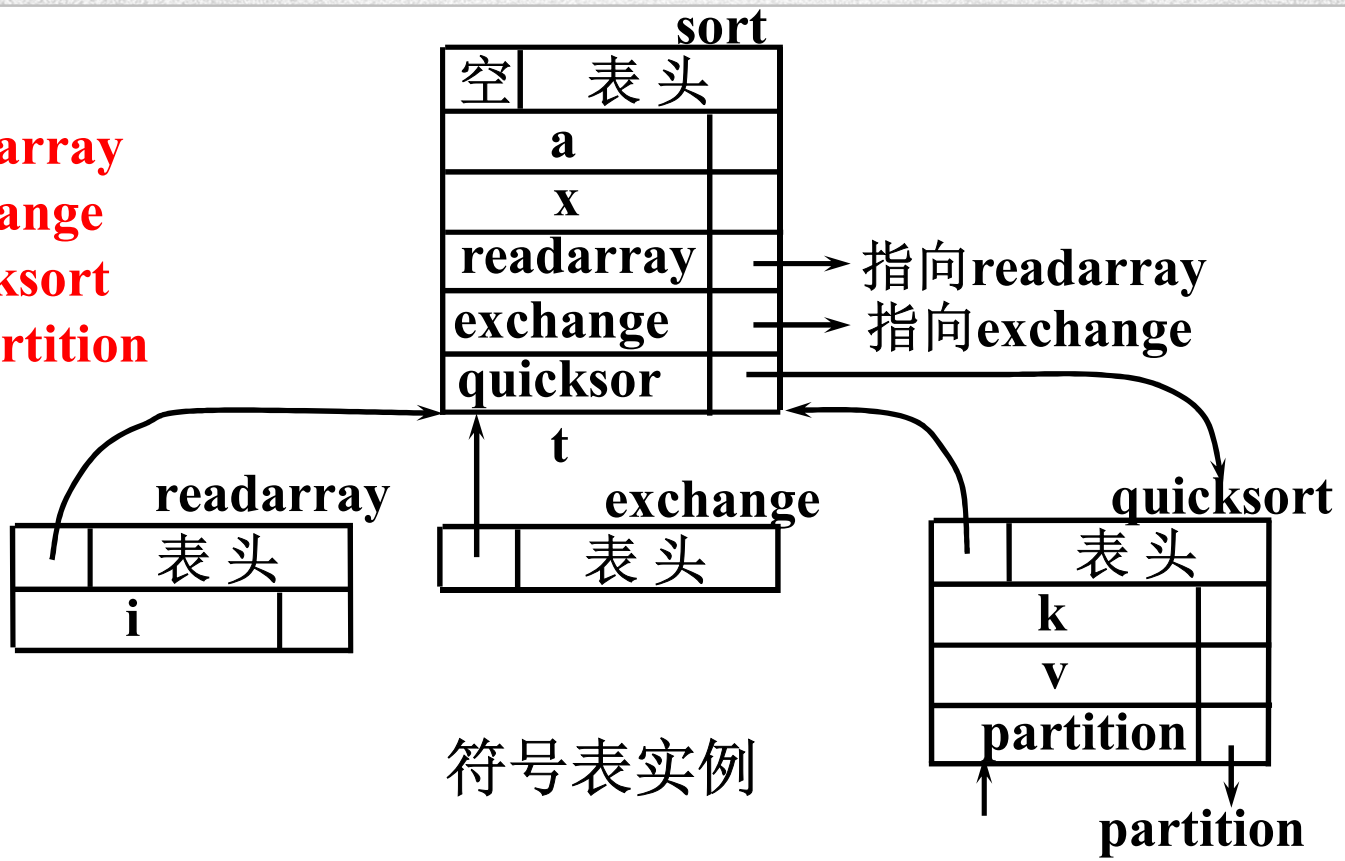
  proc id ; D ; S

```
sort
    var a:…; x:…;
    readarray
        var i:…;
    exchange
    quicksort
        var k, v:…;
        partition
                var i, j:…;
```

程序参数被略去

符号表与作用域

sort
  readarray
  exchange
  quicksort
    partition



符号表实例

符号表与作用域

- **符号表的特点**
  - 各过程有各自的符号表
  - 符号表之间有双向链
  - 构造符号表时需要符号表栈
  - 构造符号表需要活动记录栈

符号表与作用域

符号表栈（*tblptr)*

活动记录栈（*offset)*

- **语义动作用到的函数**

  mkTable(*previous*)

  enter(*table, name, type, offset*)

  addWidth(*table*, *width*)

  enterProc(*table*, *name*, *newtable*)

符号表与作用域

$P \rightarrow M\ D;\ S$ {*addWidth* (*top* (*tblptr*), *top* (*offset*));
pop(*tblptr*); *pop* (*offset*) }

$M \rightarrow \varepsilon$ {*t* = *mkTable* (*nil*);
push(*t*, *tblprt*); *push* (0, *offset*) }

$D \rightarrow D_1\ ;\ D_2$

$D \rightarrow$ proc id ; $N\ D_1;\ S$ {*t* = *top*(*tblptr*);
addWidth(*t*, *top*(*offset*)); *pop*(*tblptr*); *pop*(*offset*);
enterProc(*top*(*tblptr*), id.*lexeme*, *t*) }

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

$N \rightarrow \varepsilon$ {*t* = *mkTable*(*top*(*tblptr*));
push(*t*, *tblptr*); *push*(0, *offset*) }

符号表与作用域

$P \rightarrow M\ D;\ S$ {*addWidth* (*top* (*tblptr*), *top* (*offset*));
pop(*tblptr*); *pop* (*offset*) }

$M \rightarrow \varepsilon$ {*t* = *mkTable* (*nil*);
push(*t*, *tblprt*); *push* (0, *offset*) }

$D \rightarrow D_1\ ;\ D_2$

$D \rightarrow$ proc id ; $N\ D_1;\ S$ {*t* = *top*(*tblptr*);
addWidth(*t*, *top*(*offset*)); *pop*(*tblptr*); *pop*(*offset*);
enterProc(*top*(*tblptr*), id.*lexeme*, *t*) }

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

$N \rightarrow \varepsilon$ {*t* = *mkTable*(*top*(*tblptr*));
push(*t*, *tblptr*); *push*(0, *offset*) }

符号表与作用域

$P \rightarrow M\ D;\ S$ {*addWidth* (*top* (*tblptr*), *top* (*offset*));
pop(*tblptr*); *pop* (*offset*) }

$M \rightarrow \varepsilon$ {*t* = *mkTable* (*nil*);
push(*t*, *tblprt*); *push* (0, *offset*) }

$D \rightarrow D_1 ; D_2$

$D \rightarrow$ proc id ; $N\ D_1$; S {*t* = *top*(*tblptr*);
addWidth(*t*, *top*(*offset*)); *pop*(*tblptr*); *pop*(*offset*);
enterProc(*top*(*tblptr*), id.*lexeme*, *t*) }

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

$N \rightarrow \varepsilon$ {*t* = *mkTable*(*top*(*tblptr*));
push(*t*, *tblptr*); *push*(0, *offset*) }

记录的处理

- **T → 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

$T \rightarrow$ record ‘{‘ *L D* ‘}’
{*T.type* = *record* (*top*(*tblptr*));
T.width = *top*(*offset*);
pop(*tblptr*); *pop*(*offset*) }
$L \rightarrow \varepsilon$ {*t* = *mkTable*(*nil*);
push(*t*, *tblprt*); *push*(0, *offset*) }

$T \rightarrow$	record ‘{’	{ *Env.push*(*top*); *top* = new *Env*(); *Stack.push*(*offset*); *offset* = 0; }
	D ‘}’	{ *T.type* = *record*(*top*); *T.width* = *offset*; *top* = *Env.pop*(); *offset* = *Stack.pop*(); }

Env是存储表的栈
Stack是存储offset值的栈
top是当前栈顶所指的表 (top.get, top.put)
offset是当前栈顶所指的表的整体偏移量

赋值语句的翻译

表达式代码的SDD

- **将表达式翻译成三地址指令序列**

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

code表示代码

addr表示存放表达式结果的地址（临时变量）

new Temp()可以生成一个临时变量

gen(...)生成一个指令

表达式代码的SDD

- **例：赋值语句：a=b* -c + b* -c**

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

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

a) 三地址代码

表达式代码的SDD

- **可做重名检查**

top.get(id.lexeme)

$S \rightarrow$ id := E {*p* = *lookup*(id.*lexeme*);
if *p* != *nil* then
gen(*p*, '=', *E.addr*)
else *error* }

$E \rightarrow$ id {*p* = *lookup*(id.*lexeme*);
if *p* != *nil* then *E.addr* = *p* else *error* }

增量式翻译方案

- **主属性code满足增量式翻译的条件**

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

gen依次执行直接生成指令序列

$S \rightarrow \mathbf{id} = E\ ;$	$\{\ gen(\ top.get(\mathbf{id}.lexeme)\ '='\ E.addr);\ \}$
$E \rightarrow E_1 + E_2$	$\{\ E.addr = \mathbf{new}\ Temp();$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);\ \}$
$\mid\ - E_1$	$\{\ E.addr = \mathbf{new}\ Temp();$ $gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr);\ \}$
$\mid\ (\ E_1\ )$	$\{\ E.addr = E_1.addr;\ \}$
$\mid\ \mathbf{id}$	$\{\ E.addr = top.get(\mathbf{id}.lexeme);\ \}$

数组元素的寻址

- **数组元素存储在一块连续的存储空间中，以方便快速的访问它们**

重要

  - n个数组元素是0,1,…,n-1进行顺序编号的
  - 假设每个数组元素宽度是w，那么数组A的第i个元素的开始地址为base + i*w
    - base是A[0]的相对地址

数组元素的寻址

- **二维数组**
  - $A[i_1][i_2]$表示第i_1行第i_2个元素。假设一行的宽度是w_1，同一行中每个元素的宽度是w_2，$A[i_1][i_2]$的相对地址是$base+i_1*w_1+i_2*w_2$
  - 根据第j维上的数组元素的个数n_j和该数组每个元素的宽度w进行计算的，如二维数组$A[i_1][i_2]$的地址$base+(i_1*n_2+i_2)*w$
- **对于k维数组$A[i_1][i_2]\ldots[i_k]$，推广**
  - $base+i_1*w_1+i_2*w2+\ldots+i_k*w_k$
  - 于k维数组$A[i_1][i_2]\ldots[i_k]$ 的地址$base+((\ldots(i_1*n_2+i_2)*n_3+i_3)\ldots)*n_k+i_k)*w$

数组元素的寻址

- **有时下标不一定从0开始，**
  - 比如一维数组编号low,low+1,…,high，此时base是A[low]的相对地址。计算A[i]的地址变成base+(i-low)*w
- **预先计算技术**
  - base+i*w和base+(i-low)*w都可以改写成i*w+c的形式，其中c=base-low*w可以在编译时刻预先计算出来，计算A[i]的相对地址只要计算i*w再加上c就可以了

数组元素的寻址

- **上述地址的计算是按行存放的**

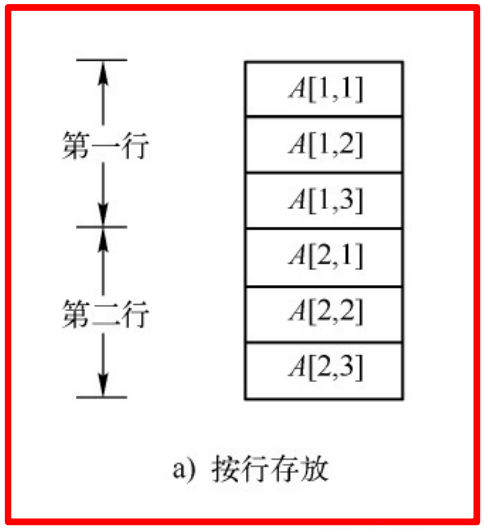


图 6-21　二维数组的存储布局

按行存放策略和按列存放策略可以推广到多维数组中

数组引用的翻译

- **为数组引用生成代码要解决的主要问题**
  - 数组引用的文法和地址计算相关联
- **假定数组编号从0开始，基于宽度来计算相对地址**
- **数组引用相关文法**
  - 非终结符号L生成一个数组名字加上一个下标表达式序列

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

数组引用生成代码的翻译方案

- **非终结符号L的三个综合属性**
  - L.addr指示一个临时变量，计算数组引用的偏移量
  - L.array是一个指向数组名字对应的符号表条目的指针，L.array.base为该数组的基地址
  - L.type是L生成的子数组的类型，对于任何数组类型t，其宽度由t.width给出，t.elem给出其数组元素的类型

数组引用生成代码的翻译方案

核心是确定数组引用的地址

$S \rightarrow$	**id** = E ;	{ $gen(top.get(\textbf{id}.lexeme)\ '='\ E.addr)$; }
|	L = E ;	{ $gen(L.array.base\ '['\ L.addr\ ']'\ '='\ E.addr)$; }
$E \rightarrow$	E_1 + E_2	{ $E.addr = \textbf{new}\ Temp()$; $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$; }
|	**id**	{ $E.addr = top.get(\textbf{id}.lexeme)$; }
|	L	{ $E.addr = \textbf{new}\ Temp()$; $gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']')$; }

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

图 6-22　处理数组引用的语义动作

数组引用翻译示例

- **基于数组引用的翻译方案，表达式c+a[i][j]的注释语法树及三地址代码序列**
- **假设a是一个2*3的整数数组，c、i、j都是整数**
  - 那么a的类型是array(2, array(3,integer)), a的宽度是24
  - a[i]的类型是array(3,integer)，宽度是12
  - a[i][j]的类型是整型

数组引用翻译示例

$L \rightarrow \mathbf{id}\ [\ E\ ]$ { $L.array = top.get(\mathbf{id}.lexeme)$;
$L.type = L.array.type.elem$;
$L.addr = \mathbf{new}\ Temp()$;
$gen(L.addr\ '='\ E.addr\ '*'\ L.type.width)$; }

| $L_1\ [\ E\ ]$ { $L.array = L_1.array$;
$L.type = L_1.type.elem$;
$t = \mathbf{new}\ Temp()$;
$L.addr = \mathbf{new}\ Temp()$;
$gen(t\ '='\ E.addr\ '*'\ L.type.width)$;
$gen(L.addr\ '='\ L_1.addr\ '+'\ t)$; }

$E \rightarrow E_1 + E_2$ { $E.addr = \mathbf{new}\ Temp()$;
$gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$; }

| $\mathbf{id}$ { $E.addr = top.get(\mathbf{id}.lexeme)$; }

| L { $E.addr = \mathbf{new}\ Temp()$;
$gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']')$; }

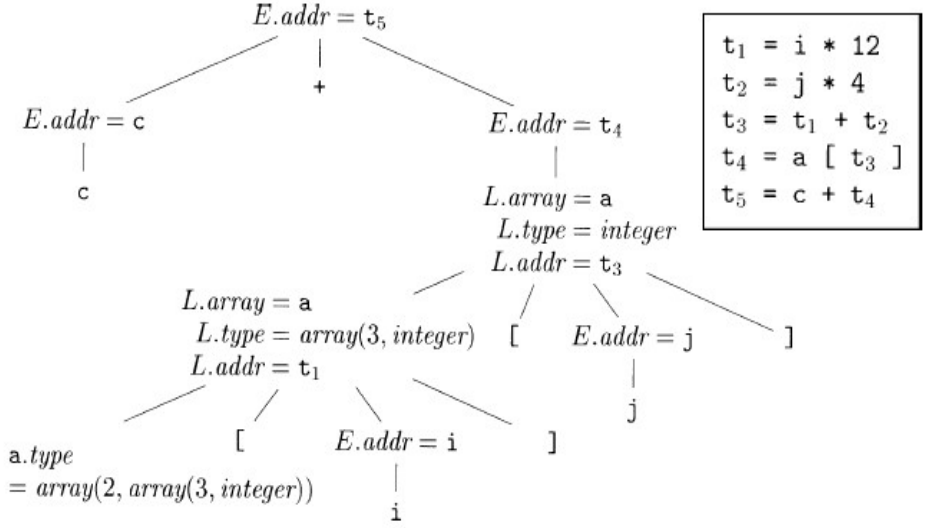


图 6-23 c+a[i][j]的注释语法分析树

L.addr指示一个临时变量，计算数组引用的偏移量
L.array是一个指向数组名字对应的符号表条目的指针，L.array.base为该数组的基地址
L.type是L生成的子数组的类型，对于任何数组类型t，其宽度由t.width给出，t.elem给出其数组元素的类型

作业

- 教材p237：6.2.2
- 教材p242：6.3.1
- 教材p247：6.4.3 (2)

Thank you!