



中山大學
SUN YAT-SEN UNIVERSITY



国家超级计算广州中心
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

DCS²⁹⁰

Compilation Principle 编译原理

第六章 中间代码生成 (2)

郑馥丹

zhengfd5@mail.sysu.edu.cn

CONTENTS

目录

01

中间代码概述
Introduction

02

类型和声明
Types and
Declarations

03

表达式和语句
Assignment and
Expressions

04

类型检查
Type
Checking

05

布尔表达式
Boolean
Expressions

06

回填技术
Backpatching

1. 类型表达式[type expression]

- 类型表达式包括：
 - 基本类型，如boolean、char、integer、float、void等
 - 类名
 - 类型构造算子array，如array(3, integer)
 - 类型构造算子record，如record{float x; float y;}
 - 类型构造算子 $\rightarrow$ ，如 $s \rightarrow t$ 表示从类型s到类型t的函数
 - 笛卡尔积 $\times$ ：具有左结合性，优先级高于 $\rightarrow$
 - 取值为类型表达式的变量
- 保证运算分量的类型和运算符的预期类型相匹配
 - 例如，Java要求 $\&\&$ 运算符的两个运算分类必须是boolean型，若满足这个条件，则运算结果也是boolean型

2. 声明[declarations]

- 类型及其声明文法

$D \rightarrow T \text{ id } ; D \mid \varepsilon$	// D生成一系列声明
$T \rightarrow B \ C \mid \text{record } \{ D \}$	// T生成基本类型、数组类型或记录类型
$B \rightarrow \text{int} \mid \text{double}$	// B生成基本类型int或double
$C \rightarrow [\text{ num }] C \mid \varepsilon$	// C生成零个或多个整数，每个整数用方括号括起来

一个数组类型包含一个由B指定的基本类型，后跟一个由C指定的数组分量
如**int[2][3]**

3. 类型的存储

- 类型的宽度[width]是指该类型的一个对象所需的存储单元的数量
 - 基本类型：char、int、float、double等，需要整数多个连续字节
 - 数组和类：需要一个连续的存储字节块
 - 例：计算基本类型和数组类型及其宽度的SDT

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

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

$B \rightarrow \mathbf{int} \{ B.type = \mathbf{INTEGER}; B.width = 4 \}$

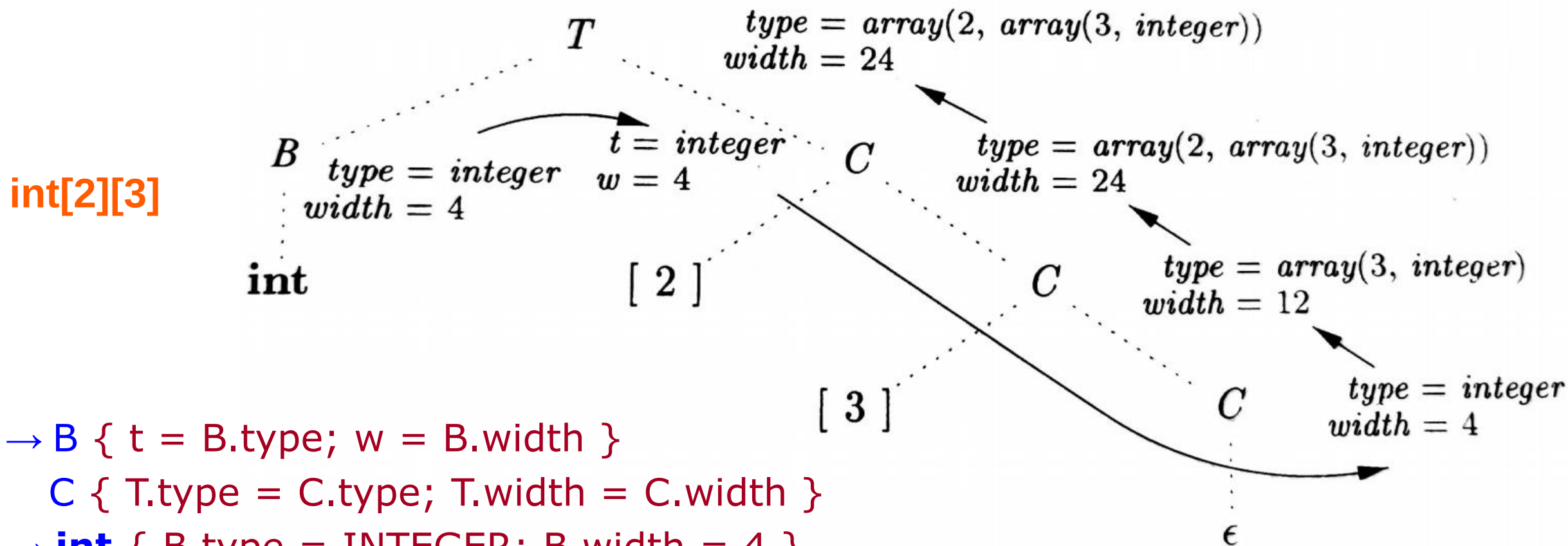
$B \rightarrow \mathbf{double} \{ B.type = \mathbf{DOUBLE}; B.width = 8 \}$

$C \rightarrow [\mathbf{num}] C1 \{ C.type = \mathbf{array}(\mathbf{num.value}, C1.type);$
 $C.width = \mathbf{num.value} \times C1.width \}$

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

试分析int[2][3]的T.type和T.width

3. 类型的存储



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

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

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

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

$C \rightarrow [\text{num}] C1 \{ C.type = \text{array}(\text{num.value}, C1.type);$

$C.width = \text{num.value} \times C1.width \}$

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

T.type=integer
T.width=24

3. 类型的存储

- 计算相对地址

- 例：计算被声明变量相对地址的SDT

$P \rightarrow \{ \text{offset} = 0 \}$ // offset表示存储变量的相对地址
// 在声明的最开始初始化为0

D

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

// top表示当前符号栈

// 每声明一个变量x，即将x加入符号表，保存x的类型，并将x的相对地

// 址设置为offset，并将x的宽度叠加到offset上

D1

$D \rightarrow \varepsilon$

4. 记录和类中的字段

- 记录类型对应的产生式： $T \rightarrow \text{record } \{ D \}$
 - 一个记录中各个字段的名称必须互不相同，即**D中声明的名称必须不重复**
 - 字段名的offset是相对于该记录的数据区字段而言的
- 以下命名并不冲突
 - float x;
 - record{ float x; float y; } p;
 - record{ float x; float y; } q;
 - x=p.x+q.x;
- 采用**专用的符号表来记录各个字段的类型和相对地址**

4. 记录和类中的字段

- 记录的翻译方案:

```
T → record '{' { Env.push(top);           // 保存top指向的已有符号表
                  top=new Env();           // 让top指向新的符号表
                  Stack.push(offset);      // 保存当前offset值
                  offset=0;}               // 将offset置为0

D '{'           // D生成的声明会使类型和offset被保存到新的符号表中(如前所述)
{ T.type = record(top); // 使用top创建一个记录类型
  T.width =offset;        // T.width记录整个record所需的存储空间
  top=Env.pop();          // 恢复早先保存好的符号表
  offset=Stack.pop();}    // 恢复早先保存好的offset
```

CONTENTS

目录

01

中间代码概述
Introduction

02

类型和声明
Types and
Declarations

03

表达式和语句
Assignment and
Expressions

04

类型检查
Type
Checking

05

布尔表达式
Boolean
Expressions

06

回填技术
Backpatching

1. 表达式中的运算

- 中间代码生成
 - 代码拼接[Code concatenation]
 - 增量生成[Incremental generation]

1. 表达式中的运算

- 中间代码生成

- 代码拼接[Code concatenation]

- ✓ 使用记号`gen(...)`来表示三地址指令，例如`gen(x='y'+z)`表示三地址指令 $x=y+z$
 - ✓ `||`作为代码拼接符号

1. 表达式中的运算

• 中间代码生成

– 代码拼接[Code concatenation]

✓ 例：表达式的三地址码

	Productions	Semantic Rules
1	$S \rightarrow \mathbf{id} = E ;$	$S.code = E.code \parallel \text{gen}(\text{top.get}(\mathbf{id.lexeme}) \text{'=' } E.addr)$
2	$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new Temp}();$ $E.code = E_1.code \parallel E_2.code \parallel \text{gen}(E.addr \text{'=' } E_1.addr \text{'+' } E_2.addr)$
3	$E \rightarrow - E_1$	$E.addr = \mathbf{new Temp}();$ $E.code = E_1.code \parallel \text{gen}(E.addr \text{'=' } \mathbf{'minus'} E_1.addr)$
4	$E \rightarrow (E_1)$	$E.addr = E_1.addr;$ $E.code = E_1.code$
5	$E \rightarrow \mathbf{id}$	$E.addr = \text{top.get}(\mathbf{id.lexeme});$ $E.code = ""$

则赋值语句 $a=b+-c$ 可翻译为如下的三地址码序列：

$t1 = \text{minus } c$

$t2 = b + t1$

$a = t2$

1. 表达式中的运算

- 中间代码生成

- 增量生成[Incremental generation]

- ✓ emit(...)
 - ✓ 或重载gen(...)
 - ✓ 不再用到code属性
 - ✓ 例：上述表达式的例子，可采用增量生成方式

	Productions	Semantic Rules
1	$S \rightarrow \mathbf{id} = E ;$	$S.code = E.code \parallel \text{gen}(\text{top.get}(\mathbf{id.lexeme}) \text{'=' } E.addr)$
2	$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new Temp}();$ $E.code = E_1.code \parallel E_2.code \parallel \text{gen}(E.addr \text{'=' } E_1.addr \text{'+' } E_2.addr)$
3	$E \rightarrow - E_1$	$E.addr = \mathbf{new Temp}();$ $E.code = E_1.code \parallel \text{gen}(E.addr \text{'=' 'minus' } E_1.addr)$
4	$E \rightarrow (E_1)$	$E.addr = E_1.addr;$ $E.code = E_1.code$
5	$E \rightarrow \mathbf{id}$	$E.addr = \text{top.get}(\mathbf{id.lexeme});$ $E.code = ""$

$S \rightarrow \mathbf{id} = E ;$
 $E \rightarrow E1 + E2$

 $E \rightarrow - E1$

 $E \rightarrow (E1)$
 $E \rightarrow \mathbf{id}$

$\{ \text{gen}(\text{top.get}(\mathbf{id.lexeme}) \text{'=' } E.addr) \}$
 $\{ E.addr = \mathbf{new Temp}();$
 $\text{gen}(E.addr \text{'=' } E1 .addr \text{'+' } E2 .addr) \}$
 $\{ E.addr = \mathbf{new Temp}();$
 $\text{gen}(E.addr \text{'=' 'minus' } E1 .addr) \}$
 $\{ E.addr = E1 .addr \}$
 $\{ E.addr = \text{top.get}(\mathbf{id.lexeme}) \}$

2. 数组元素的寻址

- 二维数组的存储布局

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

按行存储

1st column	{	A[1, 1]
		A[2, 1]
2nd column	{	A[1, 2]
		A[2, 2]
3rd column	{	A[1, 3]
		A[2, 3]

按列存储

2. 数组元素的寻址

- 相对地址

- 数组下标从0开始

- ✓ $A[i]$ (base为 $A[0]$)

- $\text{base} + i \times w$ (w 为每个数组元素的宽度)

- ✓ $A[i_1][i_2]$ (第 i_1 行的第 i_2 个元素, base为 $A[0][0]$)

- $\text{base} + (i_1 \times n_2 + i_2) \times w$ (n_2 为第2维上数组元素的个数)

- ✓ $A[i_1][i_2] \dots [i_k]$ (base为 $A[0][0] \dots [0]$)

- $\text{base} + ((\dots((i_1 \times n_2 + i_2) \times n_3 + i_3) \dots) \times n_k + i_k) \times w$

2. 数组元素的寻址

- 相对地址

- 数组下标非0开始

- ✓ $A[i]$ (base为 $A[\text{low}]$)

- $\text{base} + (i - \text{low}) \times w$ (w 为每个数组元素的宽度)

- ✓ $A[i_1][i_2]$ (第 i_1 行的第 i_2 个元素, base为 $A[\text{low}_1][\text{low}_2]$)

- $\text{base} + ((i_1 - \text{low}_1) \times n_2 + (i_2 - \text{low}_2)) \times w$ (n_2 为第2维上数组元素的个数)

- ✓ $A[i_1][i_2] \dots [i_k]$ (base为 $A[\text{low}_1][\text{low}_2] \dots [\text{low}_k]$)

- $\text{base} + (((((i_1 - \text{low}_1) \times n_2 + (i_2 - \text{low}_2)) \times n_3 + (i_3 - \text{low}_3)) \dots) \times n_k + (i_k - \text{low}_k)) \times w$

3. 数组引用的翻译

- 处理数组引用的文法及语义动作

- $L \rightarrow L [E] \mid \text{id} [E]$

$S \rightarrow \text{id} = E ;$	{ gen(top.get(id .lexeme) '=' E.addr) }
$S \rightarrow L = E ;$	{ gen(L.array.base '[' L.addr ']' '=' E.addr) }
$E \rightarrow E1 + E2$	{ E.addr = new Temp(); gen(E.addr '=' E1 .addr '+' E2 .addr) }
$E \rightarrow \text{id}$	{ E.addr = top.get(id .lexeme) }
$E \rightarrow L$	{ E.addr = new Temp(); gen(E.addr '=' L.array.base '[' L.addr ']) }
$L \rightarrow \text{id} [E]$	{ L.array = top.get(id .lexeme); L.type = L.array.type.element; L.addr = new Temp(); gen(L.addr '=' E.addr '*' L.type.width) }
$L \rightarrow L1 [E]$	{ L.array = L1 .array; L.type = L1 .type.element; t = new Temp(); L.addr = new Temp(); gen(t '=' E.addr '*' L.type.width); gen(L.addr '=' L1 .addr '+' t) }

3. 数组引用的翻译

- 处理数组引用的文法及语义动作

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

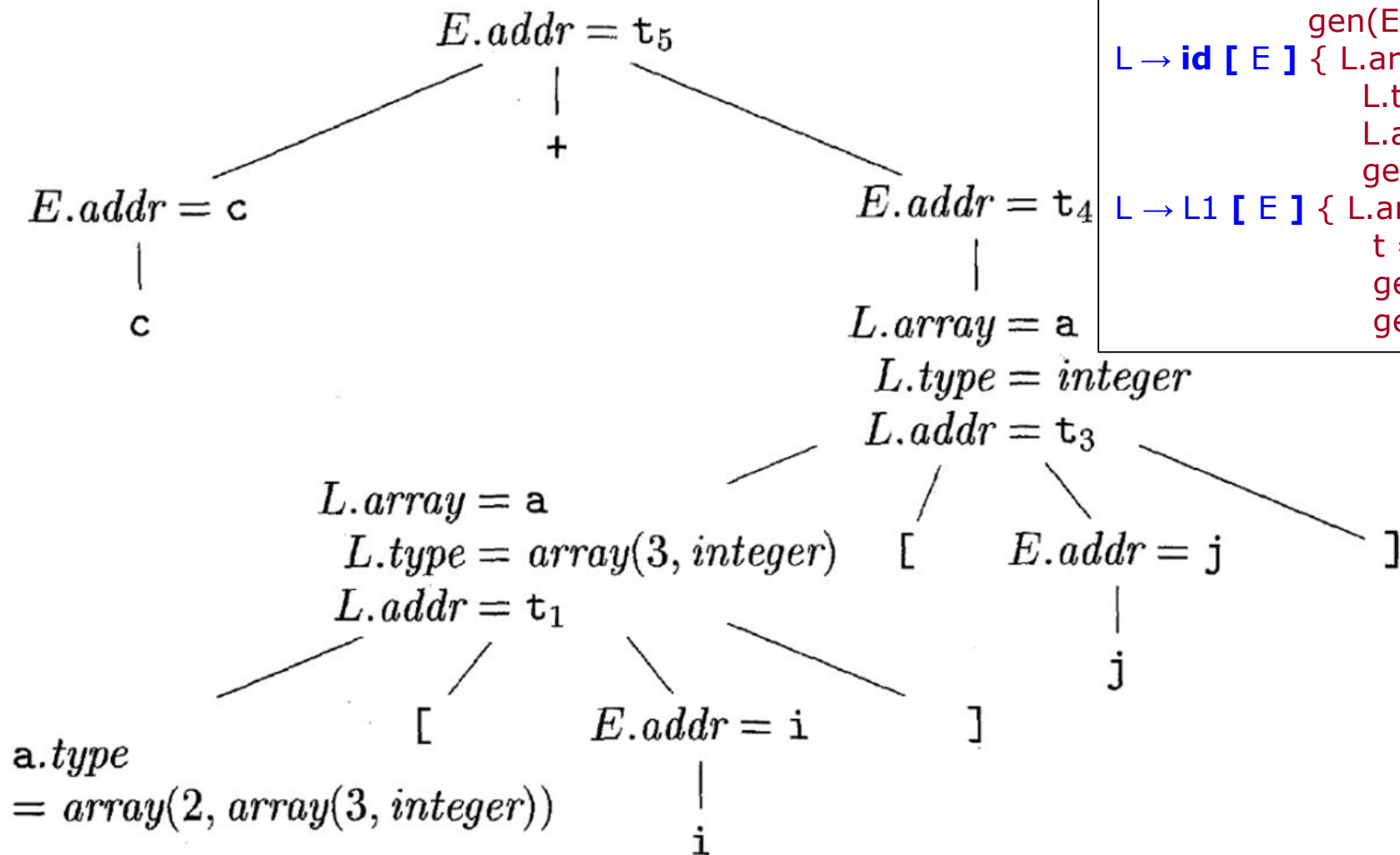
$L \rightarrow L1 [E]$ { $L.array = L1.array;$ $L.type = L1.type.element;$
 $t = \mathbf{new Temp}();$ $L.addr = \mathbf{new Temp}();$
 $\text{gen}(t '=' E.addr '*' L.type.width);$
 $\text{gen}(L.addr '=' L1.addr '+' t)$ }

- L的3个综合属性:

- ✓ L.addr: 临时变量, 用于计算数组引用的偏移量
 - ✓ L.array: 指向数组名对应的符号表的指针
 - ✓ L.type: L生成的子数组的类型

3. 数组引用的翻译

• 例: **c+a[i][j]**的翻译



```

S → id = E ; { gen(top.get(id.lexeme) '=' E.addr) }
S → L = E ; { gen(L.array.base '[' L.addr ']' '=' E.addr) }
E → E1 + E2 { E.addr = new Temp();
               gen(E.addr '=' E1 .addr '+' E2 .addr) }
E → id      { E.addr = top.get(id.lexeme) }
E → L      { E.addr = new Temp();
               gen(E.addr '=' L.array.base '[' L.addr ']) }
L → id [ E ] { L.array = top.get(id.lexeme);
               L.type = L.array.type.element;
               L.addr = new Temp();
               gen(L.addr '=' E.addr '*' L.type.width) }
L → L1 [ E ] { L.array = L1 .array; L.type = L1 .type.element;
               t = new Temp(); L.addr = new Temp();
               gen(t '=' E.addr '*' L.type.width);
               gen(L.addr '=' L1 .addr '+' t) }
    
```

```

t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a [ t3 ]
t5 = c + t4
    
```

注释语法分析树 (假设a为2×3的整数数组)

三地址码

- 按照所给的翻译方案，将以下赋值语句翻译成三地址码：

(1) $x = a[i] + b[j]$

假设a,b均为整型数组

(2) $x = a[i][j] + b[i][j]$

假设a,b均为 2×3 的整型数组

```

S → id = E ; { gen(top.get(id.lexeme) '=' E.addr) }
S → L = E ; { gen(L.array.base '[' L.addr ']' '=' E.addr) }
E → E1 + E2 { E.addr = new Temp();
               gen(E.addr '=' E1 .addr '+' E2 .addr) }
E → id       { E.addr = top.get(id.lexeme) }
E → L        { E.addr = new Temp();
               gen(E.addr '=' L.array.base '[' L.addr ']') }
L → id [ E ] { L.array = top.get(id.lexeme);
               L.type = L.array.type.element;
               L.addr = new Temp();
               gen(L.addr '=' E.addr '*' L.type.width) }
L → L1 [ E ] { L.array = L1 .array; L.type = L1 .type.element;
               t = new Temp(); L.addr = new Temp();
               gen(t '=' E.addr '*' L.type.width);
               gen(L.addr '=' L1 .addr '+' t) }
    
```

- 参考答案

假设a,b均为整型数组

(1) $x = a[i] + b[j]$

```
t1 = i * 4  
t2 = a[t1]  
t3 = j * 4  
t4 = b[t3]  
t5 = t2 + t4  
x = t5
```

假设a,b均为 2×3 的整数数组

(2) $x = a[i][j] + b[i][j]$

```
t1 = i * 12  
t2 = j * 4  
t3 = t1 + t2  
t4 = a[t3]  
t5 = i * 12  
t6 = j * 4  
t7 = t5 + t6  
t8 = b[t7]  
t9 = t4 + t8  
x = t9
```