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

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→，如s→t表示从类型s到类型t的函数
  - 笛卡尔积x：具有左结合性，优先级高于→
  - 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 → T **id ;** D | ε　　// D生成一系列声明

T → B C | **record { D }**　　// T生成基本类型、数组类型或记录类型

B → **int** | **double**　　// B生成基本类型int或double

C → **[num]** C | ε　　// 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]**

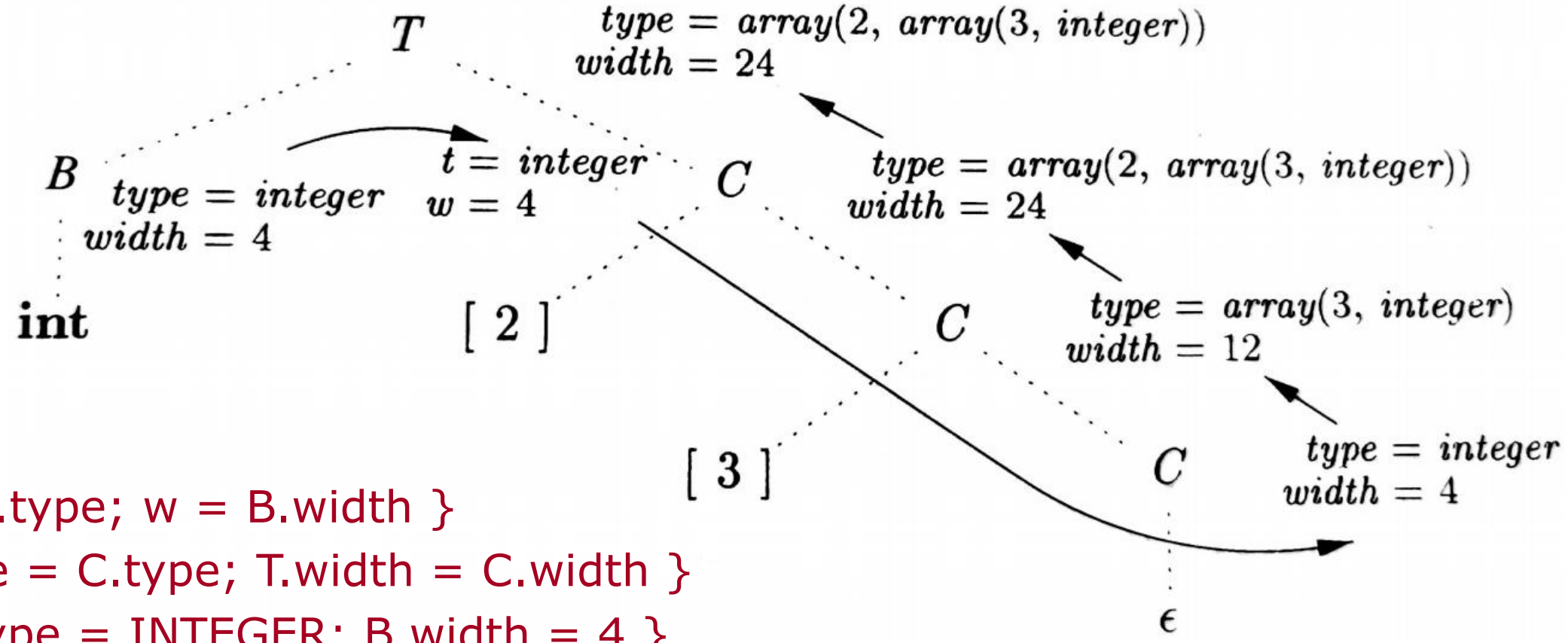
3. 类型的存储

- 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 }
$B \rightarrow$ **int** { B.type = INTEGER; B.width = 4 }
$B \rightarrow$ **double** { B.type = DOUBLE; B.width = 8 }
$C \rightarrow$ **[num]** C1 { C.type = array(**num**.value, C1 .type);
C.width = **num**.value × 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 → B { t = B.type; w = B.width }
C { T.type = C.type; T.width = C.width }

B → **int** { B.type = INTEGER; B.width = 4 }

B → **double** { B.type = DOUBLE; B.width = 8 }

C → **[num]** C1 { C.type = array(**num**.value, C1 .type);
C.width = **num**.value × C1 .width }

C → ε {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 → { 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 → T **id** ; { top.put(**id**.lexeme, T.type, offset);
offset += T.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 → ε

4. 记录和类中的字段

- 记录类型对应的产生式：T → **record '{' D '}'**
  - 一个记录中各个字段的名字必须互不相同，即**D中声明的名字必须不重复**
  - 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

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 → **id** = E ;	S.code = E.code || gen(top.get(**id**.lexeme) '=' E.addr)
2	E → E_1 + E_2	E.addr = **new** Temp(); E.code = E_1.code || E_2.code || gen(E.addr '=' E_1.addr '+' E_2.addr)
3	E → – E_1	E.addr = **new** Temp(); E.code = E_1.code || gen(E.addr '=' '**minus**' E_1.addr)
4	E → (E_1)	E.addr = E_1.addr; E.code = E_1.code
5	E → **id**	E.addr = top.get(**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 = minus c
t2 = b + t1
a = t2
```

1. 表达式中的运算

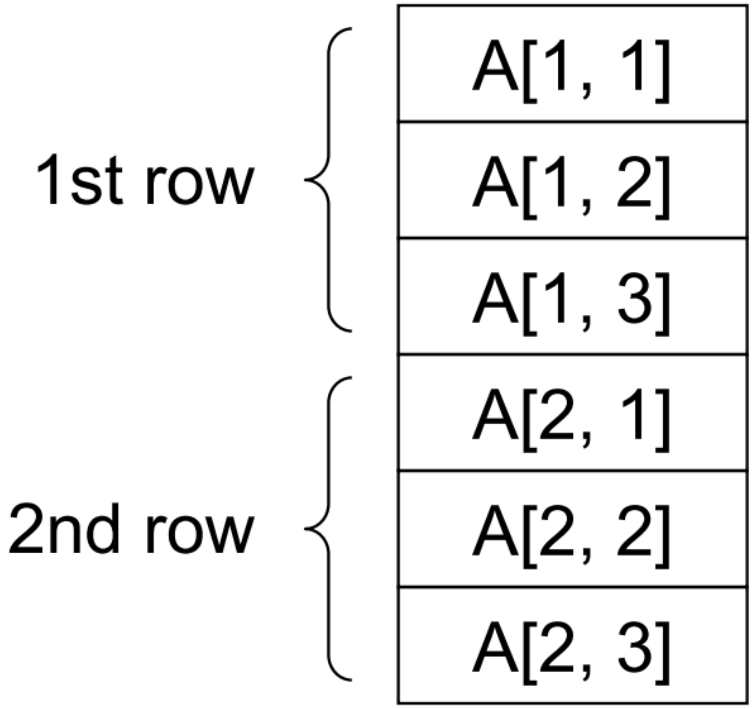
- 中间代码生成
  - 增量生成[Incremental generation]
    - ✓ emit(...)
    - ✓ 或重载gen(...)
    - ✓ **不再用到code属性**
    - ✓ 例：上述表达式的例子，可采用增量生成方式

	Productions	Semantic Rules
1	S → **id** = E ;	S.code = E.code || gen(top.get(**id**.lexeme) '=' E.addr)
2	E → E_1 + E_2	E.addr = **new** Temp(); E.code = E_1.code || E_2.code || gen(E.addr '=' E_1.addr '+' E_2.addr)
3	E → – E_1	E.addr = **new** Temp(); E.code = E_1.code || gen(E.addr '=' '**minus**' E_1.addr)
4	E → (E_1)	E.addr = E_1.addr; E.code = E_1.code
5	E → **id**	E.addr = top.get(**id**.lexeme); E.code = ""

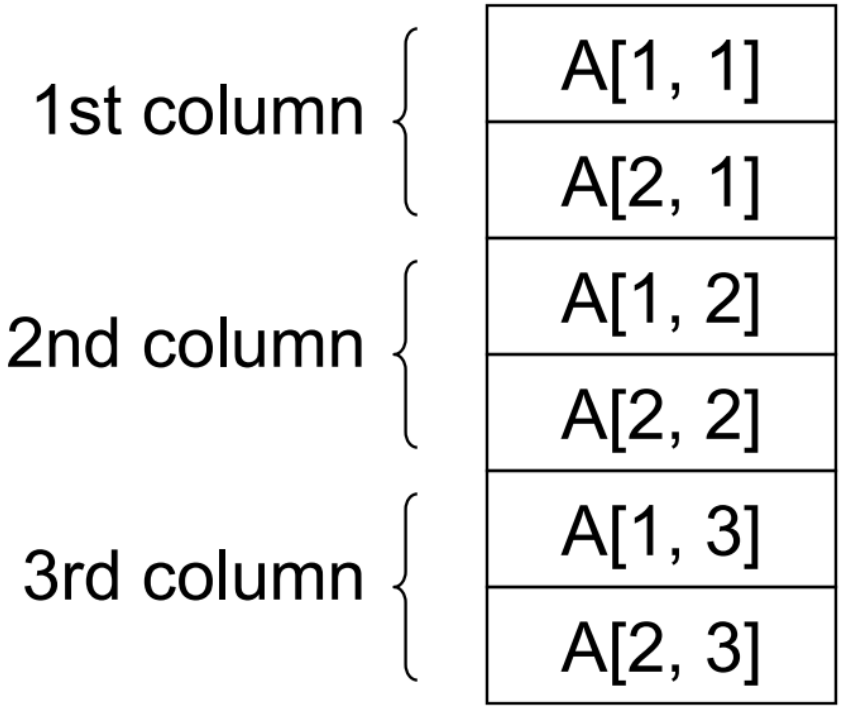
```
S → id = E ;      { gen(top.get(id.lexeme) '=' E.addr) }
E → E1 + E2       { E.addr = new Temp();
                    gen(E.addr '=' E1 .addr '+' E2 .addr) }
E → – E1          { E.addr = new Temp();
                    gen(E.addr '=' 'minus' E1 .addr) }
E → ( E1 )        { E.addr = E1 .addr }
E → id            { E.addr = top.get(id.lexeme) }
```

2. 数组元素的寻址

- 二维数组的存储布局



按行存储



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])
      - base+i×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])
      - $base+(i_1\times n_2+i_2)\times w$　　(n_2为第2维上数组元素的个数)
    - ✓ $A[i_1][i_2]\ldots[i_k]$ (base为A[0][0]…[0])
      - $base+((\ldots((i_1\times n_2+i_2)\times n_3+i_3)\ldots)\times n_k+i_k)\times w$

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[low])
      - base+(i-low)×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$A[low_1][low_2]$)
      - $base+((i_1-low_1)\times n_2+(i_2-low_2))\times w$　　(n_2为第2维上数组元素的个数)
    - ✓ $A[i_1][i_2]\ldots[i_k]$ (base为$A[low_1][low_2]\ldots[low_k]$)
      - $base+((\ldots(((i_1-low_1)\times n_2+(i_2-low_2))\times n_3+(i_3-low_3))\ldots)\times n_k+(i_k-low_k))\times w$

3. 数组引用的翻译

- 处理数组引用的文法及语义动作
  - L → L [E] | id [E]

```
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) }
```

3. 数组引用的翻译

- 处理数组引用的文法及语义动作

```
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) }
```

– L的3个综合属性：

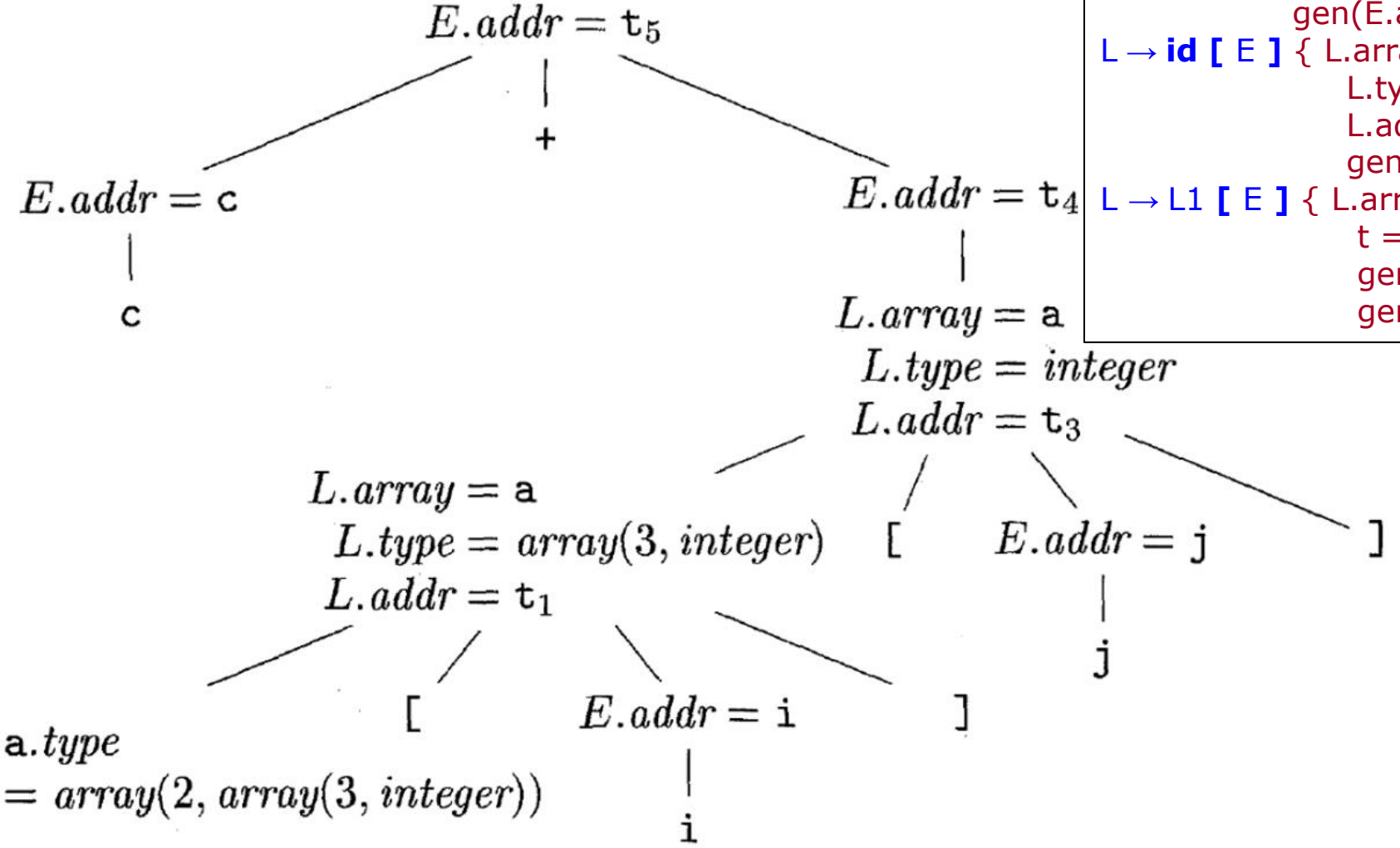
✓ L.addr：临时变量，用于计算数组引用的偏移量

✓ 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

随堂练习（3）

- 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) }
```

随堂练习（3）

- 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
```

CONTENTS

目 录

1. 强类型 vs. 弱类型

- 强类型[Strong Typing]
  - 类型规则严格，**不允许隐式的类型转换**（除非语言明确允许）
  - 类型错误会在**编译时**（或**运行时**）被捕获，避免不合理的操作
  - 变量或表达式的类型在编译时通常是确定的，且操作必须符合类型约束

```
# Python 是强类型语言
x = "10" + 5   # 抛出 TypeError，不允许字符串和数字隐式拼接

// Java 也是强类型
int a = 10;
String b = "20";
int c = a + b;   // 编译错误：类型不匹配
```

1. 强类型 vs. 弱类型

- 弱类型[Weak Typing]
  - 类型规则宽松，**允许隐式的类型转换**（自动或上下文驱动的转换）
  - 编译器或解释器可能自动尝试转换类型以完成操作，可能导致意外行为
  - 更依赖程序员自行保证类型的正确性

```
// JavaScript 是弱类型语言
let x = "10" + 5; // 输出 "105"，数字被隐式转为字符串
let y = "10" * 5; // 输出 50，字符串被隐式转为数字

// C 语言是弱类型
int x = 10;
double y = 3.14;
double z = x + y; // int隐式转为double，无警告
```

2. 表达式类型的检查

- 表达式类型检查规则：
  - if f的类型为s→t且x的类型为s

    then 表达式f(x)的类型为t

```python
# python
def f(x: int) -> int:   # f 的类型是 int → int
return x * 2

x: int = 10   # x 的类型是 int
result = f(x)   # f(x) 的类型是 int

y: str = "hello"   # y 的类型是 str
result = f(y)   # 类型错误！ str 不能赋值给 int
```

3. 类型检查的翻译方案

- 类型检查、推断和隐式类型转换

```
E → E1 * E2  { E.place := new Temp();
               if (E1.type == TK_INT && E2.type == TK_INT) {
                  emit(E.place '=' E1.place '*int' E2.place);
                  E.type = TK_INT;
               } elsif (E1.type == TK_REAL && E2.type == TK_REAL) {
                  emit(E.place '=' E1.place '*real' E2.place);
                  E.type = TK_REAL;
               } elsif (E1.type == TK_INT && E2.type == TK_REAL) {
                  t := new Temp();
                  emit(t '=' 'int2real' E1.place);
                  emit(E.place '=' t '*real' E2.place);
                  E.type = TK_REAL;
               } elsif (...) { ... }
             }
```