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

郑馥丹
zhengfd5@mail.sysu.edu.cn

CONTENTS

目 录

1. 中间代码生成[Intermediate Code Generation]

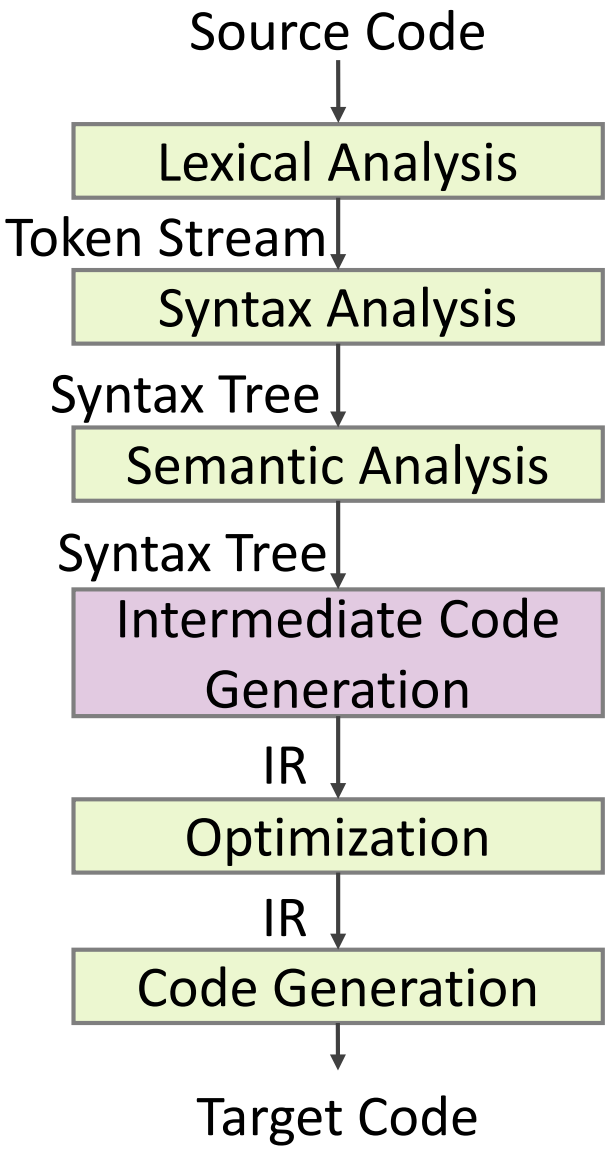
——从这里开始真正做翻译工作

- 初步翻译，生成**等价**于源程序的中间表示（IR）
  - 输入：语法树，输出：IR
  - 建立源和目标语言的桥梁，易于翻译过程的实现， 利于实现某些优化算法
  - IR形式：通常为**三地址码（TAC）**

```
void main()
{
    int arr[10], i, x = 1;

    for (i = 0; i < 10; i++)
        arr[i] = x * 5;
}
```

```
i := 0
loop:
    t1 := x * 5
    t2 := &arr
    t3 := sizeof(int)
    t4 := t3 * i
    t5 := t2 + t4
    *t5 := t1
    i := i + 1
    if i < 10 goto loop
```

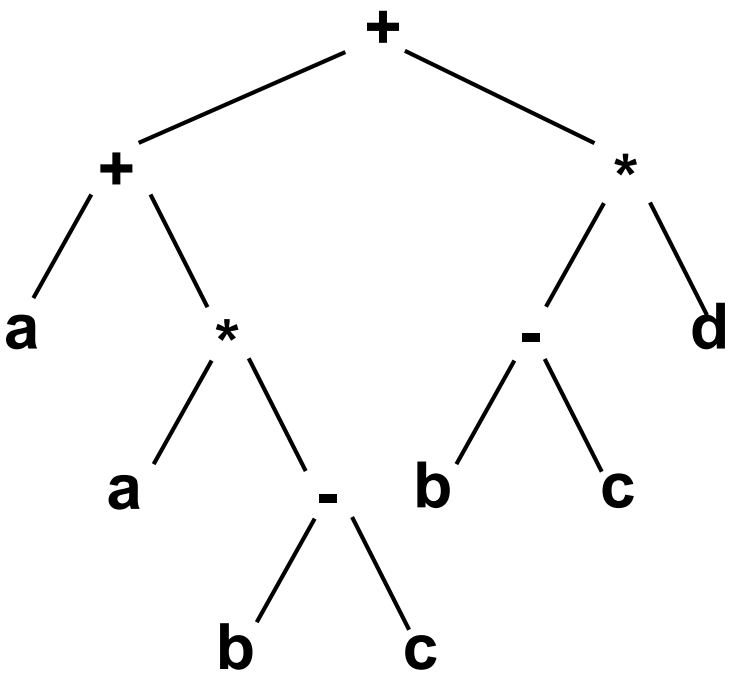


2. 中间表示[Intermediate Representation, IR]

- 高级中间表示
  - AST(Abstract Syntax Tree, 抽象语法树)和DAG(Directed Acyclic Code, 有向无环图)
  - 适用于静态类型检查等任务
- 低级中间表示
  - **3-地址码(Three-Address Code, TAC)**：x = y op z
  - 适用于依赖于机器的任务，如寄存器分配和指令选择。
- IR的选择/设计都是针对具体应用的
  - LLVM IR：通用
  - TensorFlow XLA IR：专门针对机器学习计算图优化
  - 常用C语言（AT&T贝尔实验室高级C++）

2. 中间表示[Intermediate Representat

- AST和DAG
  - 例：算术表达式a+a*(b-c)+(b-c)*d

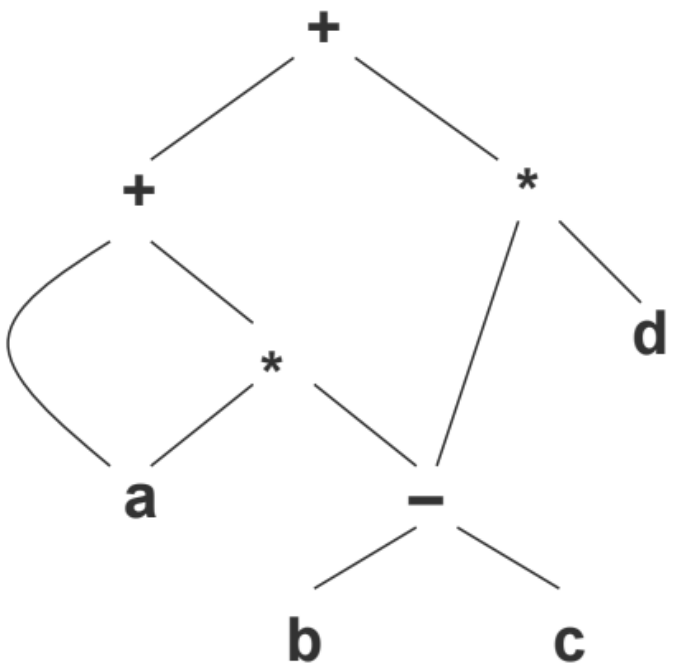


AST

(1) p1 = **new** Leaf(**id**, entry-a)
(2) **p2 = new Leaf(id, entry-a)**
(3) p3 = **new** Leaf(**id**, entry-b)
(4) p4 = **new** Leaf(**id**, entry-c)
(5) p5 = **new** Node('–', p3 , p4)
(6) p6 = **new** Node('*', p2 , p5)
(7) p7 = **new** Node('+', p1 , p6)
(8) **p8 = new Leaf(id, entry-b)**
(9) **p9 = new Leaf(id, entry-c)**
(10) **p10 = new Node('–', p8 , p9)**
(11) p11 = **new** Leaf(**id**, entry-d)
(12) p12 = **new** Node('*', p10, p11)
(13) p13 = **new** Node('+', p7 , p12)

No.	Productions	Semantic Rules
1	E → E_1 + T	E.node = **new** Node('+', E_1.node, T.node)
2	E → E_1 – T	E.node = **new** Node('–', E_1.node, T.node)
3	E → T	E.node = T.node
4	T → **(** E **)**	T.node = E.node
5	T → **id**	T.node = **new** Leaf(**id**, **id**.entry)
6	T → **num**	T.node = **new** Leaf(**num**, **num**.val)

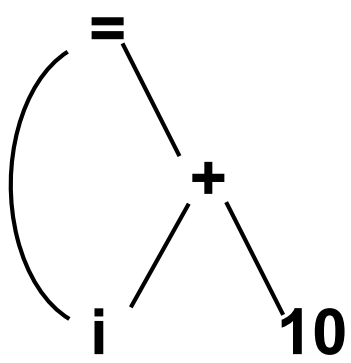
SDD



DAG

2. 中间表示[Intermediate Representation, IR]

- 构造DAG的值编码方法
  - 语法树或DAG中的结点存放在一个记录数组中
  - 数组的每一行表示一个记录，即一个结点
  - 每个记录中，都有结点编号
  - 叶子结点：一个附加字段，存放标识符的词法值lexval
  - 内部结点：两个附加字段，分别指明其左右结点
  - 例：i=i+10



DAG

1	id	i	
2	num	10	
3	+	1	2
4	=	1	3
5	…	…	…

值编码

随堂练习（1）

- 为下列表达式构造DAG，并指出其值编码，假定+是左结合的。

  (1) a+b+(a+b)

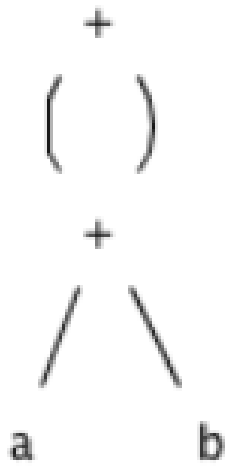
  (2) a+b+a+b

  (3) a+a+(a+a+a+(a+a+a+a))

随堂练习（1）

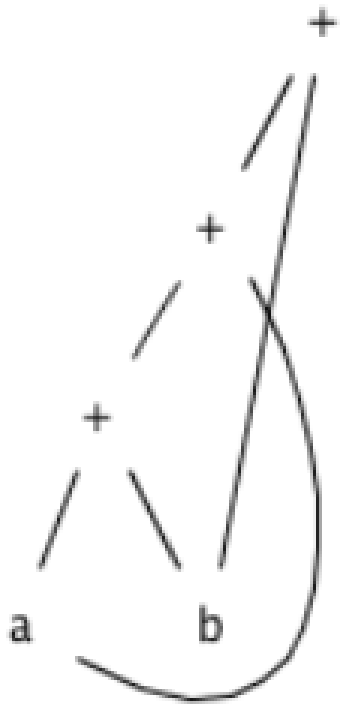
- 参考答案

(1) a+b+(a+b)



1	id	a	
2	id	b	
3	+	1	2
4	+	3	3

(2) a+b+a+b



1	id	a	
2	id	b	
3	+	1	2
4	+	3	1
5	+	4	2

随堂练习（1）

- 参考答案

(3) a+a+(a+a+a+(a+a+a+a))

1	id	a	
2	+	1	1
3	+	2	1
4	+	3	1
5	+	3	4
6	+	2	5

2. 中间表示[Intermediate Representation, IR]

- 三地址码[Three-Address Code, **TAC**]
  - 三地址码中，**一条指令右侧最多有一个运算符**，即，不允许出现组合的算术表达式
  - 像x + y * z，需翻译成如下三地址指令序列：
    - t1 = y * z
    - t2 = x + t1
  - 例：算术表达式a+a*(b-c)+(b-c)*d

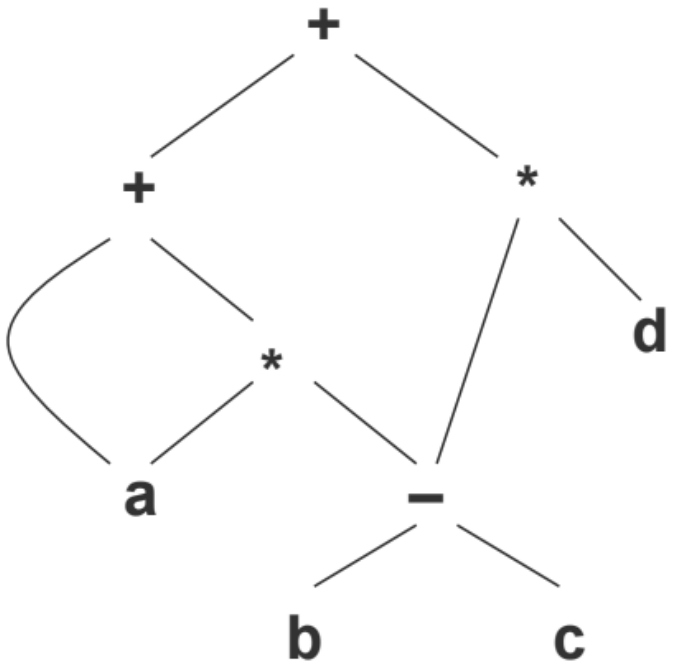
t1 = b – c

t2 = a * t1

t3 = a + t2

t4 = t1 * d

t5 = t3 + t4



DAG

2. 中间表示[Intermediate Representation, IR]

- 三地址码[Three-Address Code, TAC]
  - 两个基本概念：地址和指令
  - 地址：
    - ✓ 名字：源程序的名字作为三地址码中的地址：源程序名字被替换为指向符号表条目的指针，关于该名字的所有信息均存放在该条目中
    - ✓ 常量：需考虑表达式中的类型转换问题
    - ✓ 编译器生成的临时变量

2. 中间表示[Intermediate Representation, IR]

- 三地址码[Three-Address Code, TAC]
  - 指令
    - ✓ x = y *op* z // 双目运算或逻辑运算，x、y、z为地址
    - ✓ x = *op* y // 单目运算：单目减(取负)、逻辑非、转换运算(整数转成浮点数等)
    - ✓ x = y // 赋值运算
    - ✓ **goto** L // 无条件跳转
    - ✓ **if** x **goto** L // 有条件跳转
    - ✓ **ifFalse** x **goto** L // 有条件跳转
    - ✓ **if** x *op* y **goto** L // 关系运算跳转

2. 中间表示[Intermediate Representation, IR]

- 三地址码[Three-Address Code, TAC]
  - 指令
    - ✓ **param** x_1 // 参数传递
    - ✓ **param** x_2
    - ✓ …
    - ✓ **param** x_n
    - ✓ **call** p, n // 过程调用
    - ✓ y = **call** p, n // 函数调用
    - ✓ **return** y // 返回值

2. 中间表示[Intermediate Representation, IR]

- 三地址码[Three-Address Code, TAC]
  - 指令
    - ✓ x = y[i]　　// 带下标的赋值指令，**注意i代表内存单元位置，而非数组位置**
    - ✓ x[i] = y　　// 同上
    - ✓ x = &y　　// 取y变量的地址赋值给x
    - ✓ x = *y　　// 取y指针中的内容赋值给x
    - ✓ *x = y　　// 取y的值赋值给x指针所指空间

2. 中间表示[Intermediate Representation, IR]

- 三地址码[Three-Address Code, TAC]
  - 例：源码 **do** i=i+1;
    while(a[i]>v);

带符号标号的三地址码：

```
L: t1 = i + 1
   i = t1
   t2 = i * 8 //假设数组中每个元素
              //占8个存储单元
   t3 = a[t2]
   if t3 < v goto L
```

带位置号的三地址码：

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

2. 中间表示[Intermediate Representation, IR]

- 三地址码[Three-Address Code, TAC]
  - ① 三元式[triple]
  - ② 间接三元式[indirect triple]
  - ③ 四元式[quadruple]

2. 中间表示[Intermediate Representation, IR]

- 三地址码[Three-Address Code, TAC]

① 三元式[triple]

✓ 三个字段：**op, arg1, arg2**

✓ 用运算x op y的位置来表示其结果，而不是用一个显式的临时名字

✓ 带括号的数字表示指向相应三元式结构的指针

2. 中间表示[Intermediate Representation, IR]

- 三地址码[Three-Address Code, TAC]

① 三元式[triple]

✓ 例：源码 a = b * – c + b * – c

三地址码：

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

三元式：

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

对运算结果的引用是通过位置完成的，因此如果改变一条指令的位置，则引用该指令结果的所有指令都要相应修改！

2. 中间表示[Intermediate Representation, IR]

- 三地址码[Three-Address Code, TAC]
  - ② 间接三元式[indirect triple]
    - ✓ 同样三个字段：**op, arg1, arg2**
    - ✓ 包含**一个指向三元式的指针的列表**，而不是列出三元式序列本身，从而化解了前述三元式由于指令改变所引起的问题

2. 中间表示[Intermediate Representation, IR]

- 三地址码[Three-Address Code, TAC]

② 间接三元式[indirect triple]

✓ 例：源码 a = b * – c + b * – c

三地址码：

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

间接三元式：

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

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

优化编译器可以通过对指令列表的重新排序来移动指令位置，而不影响三元式本身。

2. 中间表示[Intermediate Representation, IR]

- 三地址码[Three-Address Code, TAC]

③ 四元式[quadruple]

✓ 四个字段：**op, arg1, arg2, result**

✓ 一些特例：

- 形如x=minus y的单目运算符指令和赋值指令x=y，不使用arg2
- 像param这样的运算既不使用arg2，也不使用result
- 条件或非条件转移指令将目标标号放入result字段

2. 中间表示[Intermediate Representation, IR]

- 三地址码[Three-Address Code, TAC]

③ 四元式[quadruple]

✓ 例：源码 a = b * – c + b * – c

三地址码：

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

四元式：

	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
…	…			

相对于三元式的优势：当移动一个计算临时变量t的指令时，那些使用t的指令不需要做任何改变

随堂练习（2）

- 对于以下的表达式，分别给出三元式和四元式序列

  (1) a=b[i]+c[j]

  (2) a[i]=b*c-b*d

  (3) x=f(y+1)+2

  (4) x=*p+&y

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

随堂练习（2）

- 参考答案

(1) a=b[i]+c[j]

#	OP	ARG1	ARG2
1	=[]	b	i
2	=[]	c	j
3	+	(1)	(2)
4	=	a	(3)

三元式

#	OP	ARG1	ARG2	RESULT
1	=[]	b	i	t1
2	=[]	c	j	t2
3	+	t1	t2	t3
4	=	t3	—	a

四元式

随堂练习（2）

- 参考答案

(2) a[i]=b*c-b*d

#	OP	ARG1	ARG2
1	*	b	c
2	*	b	d
3	-	(1)	(2)
4	[]=	a	i
5	=	(4)	(3)

三元式

#	OP	ARG1	ARG2	RESULT
1	*	b	c	t1
2	*	b	d	t2
3	-	t1	t2	t3
4	[]=	a	i	t4
5	=	t3	—	*t4

四元式

随堂练习（2）

- 参考答案

(3) x=f(y+1)+2

#	OP	ARG1	ARG2
1	+	y	1
2	param	(1)	—
3	call	f	1
4	+	(3)	2
5	=	x	(4)

三元式

#	OP	ARG1	ARG2	RESULT
1	+	y	1	t1
2	param	t1	—	—
3	call	f	1	t2
4	+	t2	2	t3
5	=	t3	—	x

四元式

随堂练习（2）

- 参考答案

(4) x=*p+&y

#	OP	ARG1	ARG2
1	*	p	—
2	&	y	—
3	+	(1)	(2)
4	=	x	(3)

三元式

#	OP	ARG1	ARG2	RESULT
1	*	p	—	t1
2	&	y	—	t2
3	+	t1	t2	t3
4	=	t3	—	x

四元式

随堂练习（2）

- 参考答案

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

#	OP	ARG1	ARG2
1	+	a	b
2	@	—	(1)
3	+	c	d
4	*	(2)	(3)
5	+	a	b
6	+	(5)	c
7	-	(4)	(6)

三元式

#	OP	ARG1	ARG2	RESULT
1	+	a	b	t1
2	@	t1	—	t2
3	+	c	d	t3
4	*	t2	t3	t4
5	+	a	b	t5
6	+	t5	c	t6
7	-	t4	t6	t7

四元式

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]**

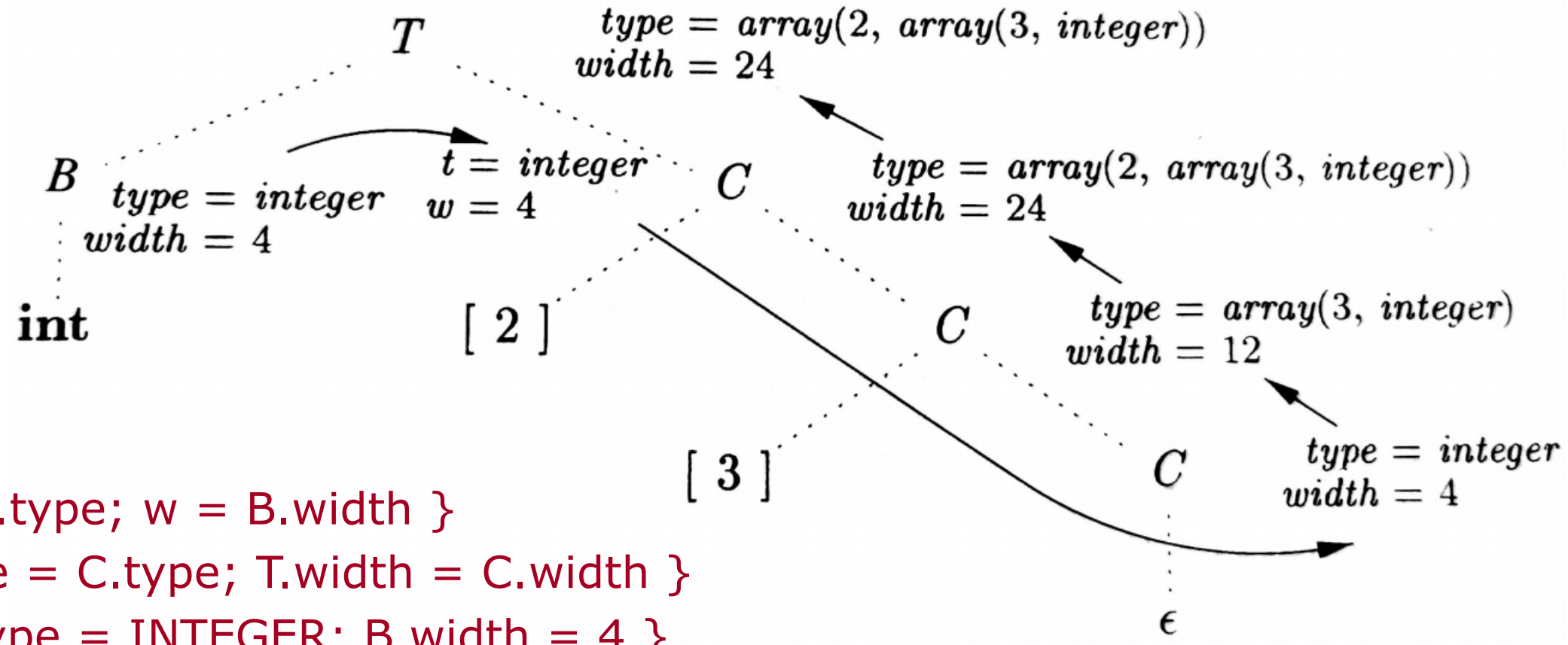
3. 类型的存储

- 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

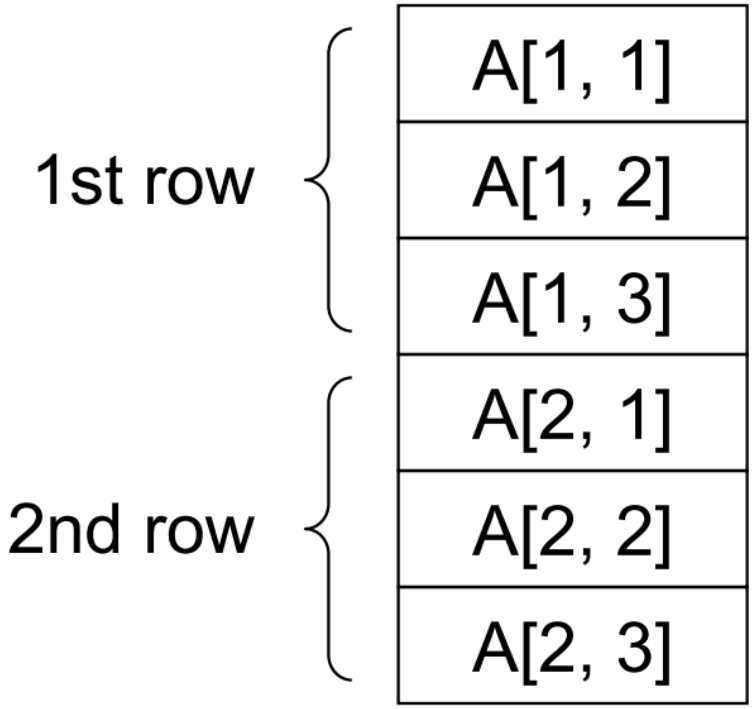
- 中间代码生成
  - 增量生成[Incremental generation]
    - ✓ emit(...)
    - ✓ 或重载gen(...)
    - ✓ **不再用到code属性**
    - ✓ 例：上述表达式的例子，可采用增量生成方式

	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 = ""

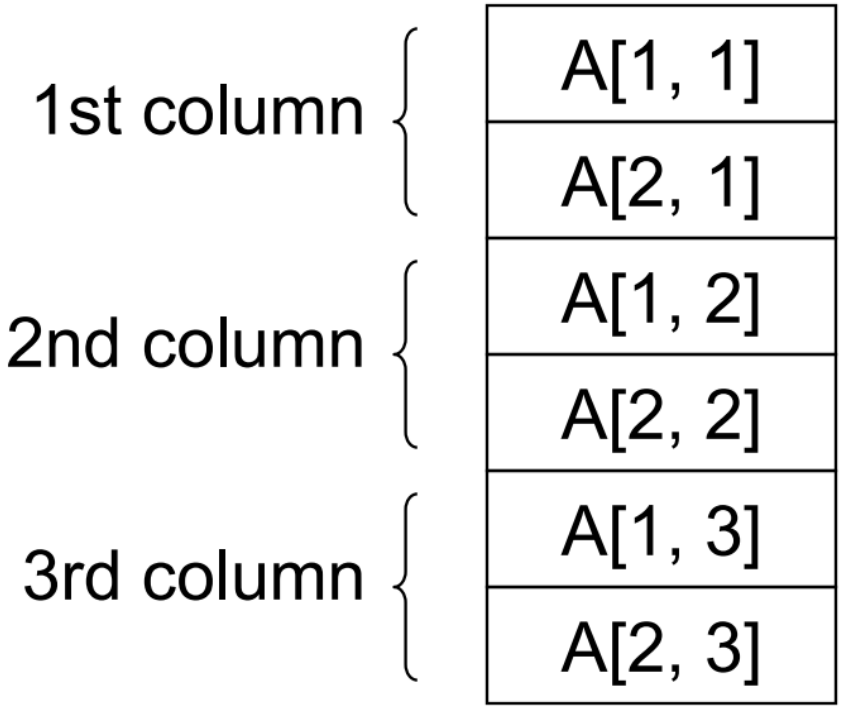
```
S → id = E ;      { gen(top.get(id.lexeme) '=' E.addr) }
E → E1 + E2       { E.addr = new Temp();
                    gen(E.addr '=' E1 .addr '+' E2 .addr) }
E → – E1          { E.addr = new Temp();
                    gen(E.addr '=' 'minus' E1 .addr) }
E → ( E1 )        { E.addr = E1 .addr }
E →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

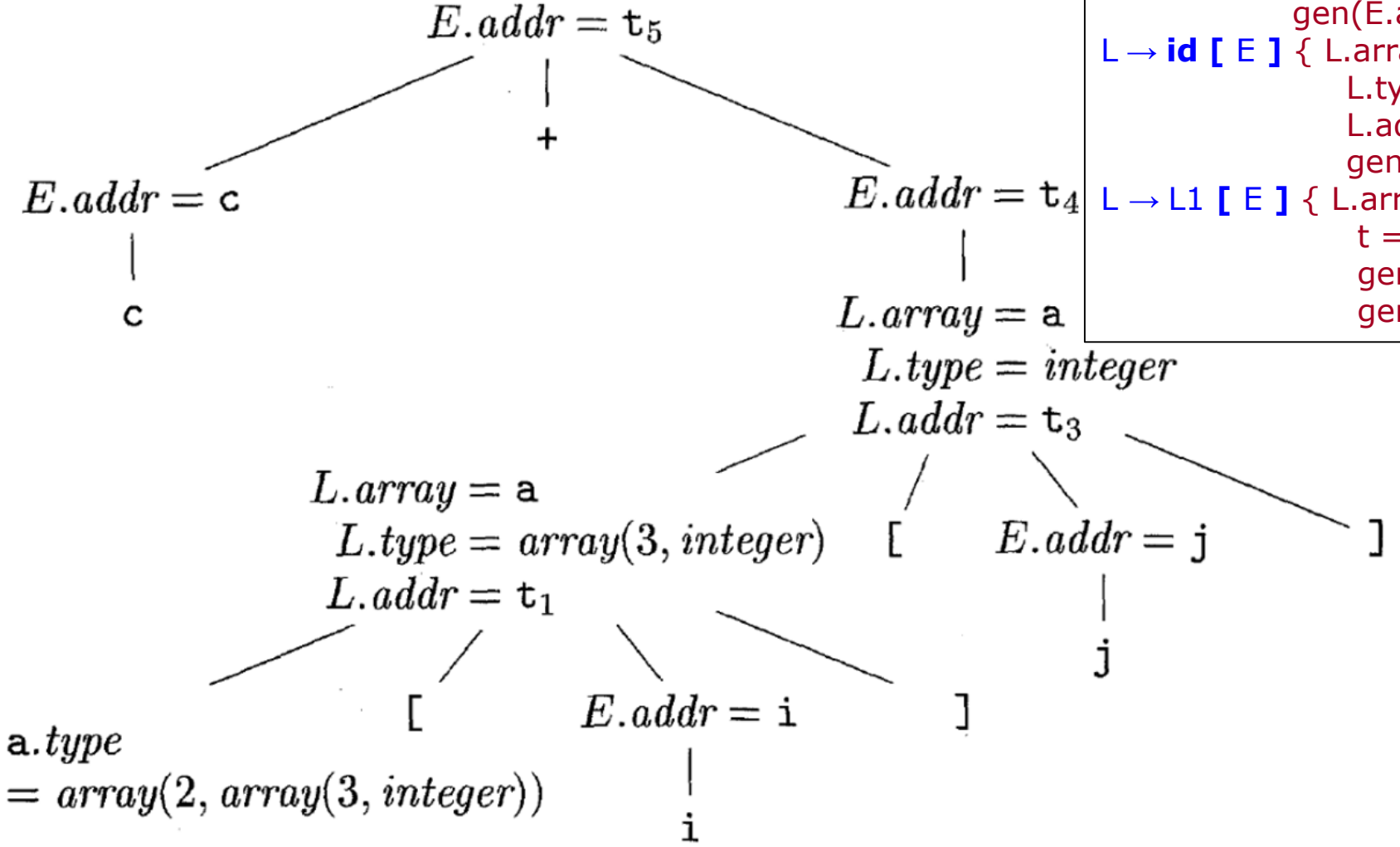
- 处理数组引用的文法及语义动作

```
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 (...) { ... }
             }
```

CONTENTS

目 录

1. 布尔表达式

- 布尔表达式的用途
  - 计算逻辑值
  - 控制流：作为控制语句(如if-then,while)的条件表达式
- 形式
  - **E→E or E | E and E | not E | (E) | id rop id | true | false**
  - 布尔算符的优先顺序（从高到低）为：not, and, or，且and和or都服从左结合，not服从右结合
  - rop是关系算符（<=, <, = , !=, >, >= 等）；id rop id是关系式，关系式中的id是算术量。**关系算符的优先级都相同，而且高于任何布尔算符，低于任何算术算符。**

1. 布尔表达式

- 布尔表达式的计算方法
  - 数值表示的直接计算
    - ✓ 1 or 0 and 1 = 1 or 0 = 1
  - 逻辑表示的短路计算
    - ✓ 布尔表达式计算到某一部分就可以得到结果，而**无需对布尔表达式进行完全计算**(作为条件控制的情况)，可以用if-then-else来解释：
      - A or B　　if A then 1 else B
      - A and B　if A then B else 0
      - not A　　if A then 0 else 1

2. 直接计算的翻译

- 如：A or B and not C被翻译成：

(not, C, - , t1)

(and, B, t1, t2)

(or, A, t2, t3)

- 对关系表达式a<b，可翻译成如下固定的三地址代码（四元式）序列：

a<b 等价于 if a<b then 1 else 0

(1) if a<b then goto (4)
(2) t:=false
(3) goto (5)
(4) t:=true
(5) .....

(1) (j<, a, b, (4))
(2) (:=, 0, - , t1)
(3) (jump, - , - , (5))
(4) (:=, 1, - , t1)
(5) ...

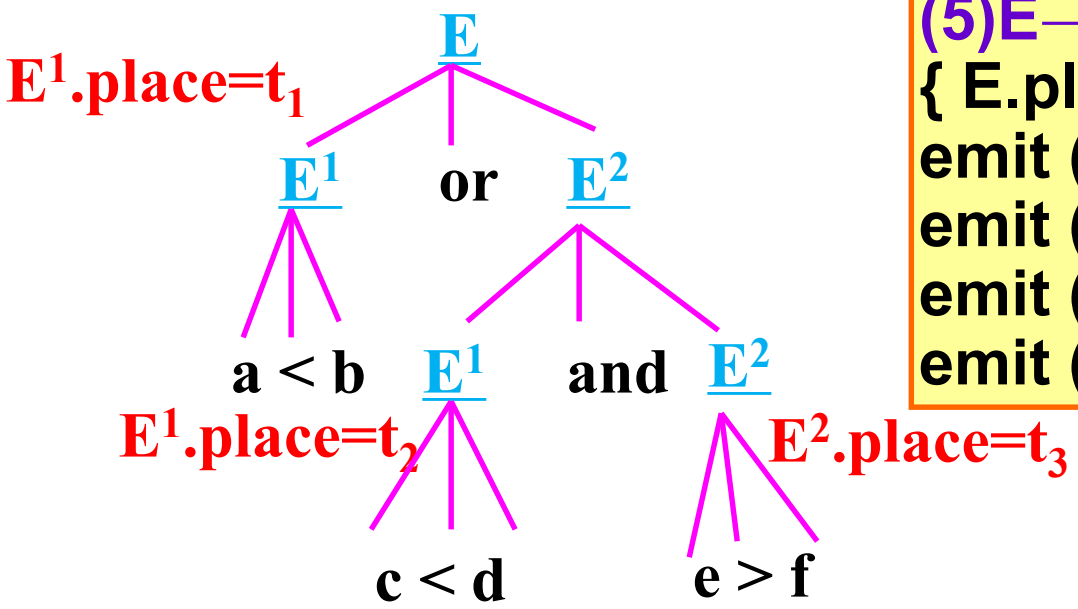
2. 直接计算的翻译

(1) $E \rightarrow E^1$ or E^2	{ E.place : = newtemp ; emit (or , E^1.place , E^2.place , E.place) }
(2) $E \rightarrow E^1$ and E^2	{ E.place : = newtemp ; emit (and , E^1.place , E^2.place , E.place) }
(3) $E \rightarrow$ not E^1	{ E.place : = newtemp ; emit (not , E^1.place ,—, E.place) }
(4) $E \rightarrow (E^1)$	{ E.place : = E^1.place }
(5) $E \rightarrow id_1$ rop id_2	{ E.place : = newtemp ; emit (jrop , id_1.place , id_2.place , nextstat+3) ; emit (: = , 0 , - , E.place) ; emit (jump ,—,—, nextstat+2) ; emit (: = , 1 , - , E.place) }
(6) E→true	{ E.place: = newtemp;emit(:=,1,- ,E.place) }
(7) E→false	{E.place:=newtemp;emit(:=,0,- ,E.place)}

(1) (j<, a, b, (4))
(2) (:=, 0, - , t_1)
(3) (jump, - , - , (5))
(4) (:=, 1, - , t_1)
(5) ...

2. 直接计算的翻译

- 例：布尔表达式a<b or c<d and e>f的翻译

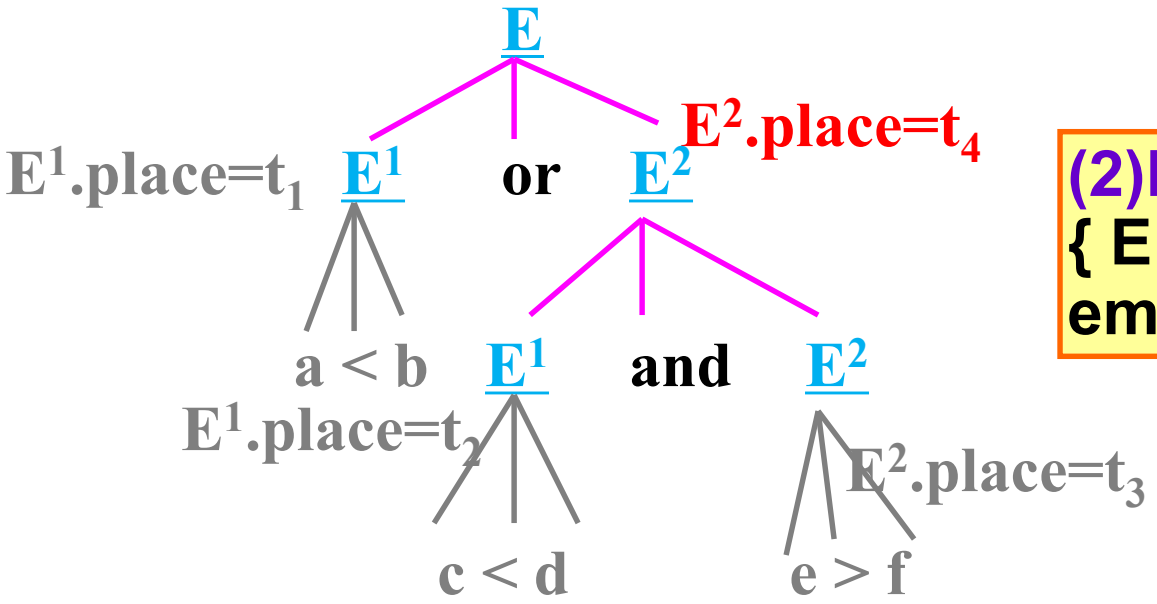


(5)E→id$_1$ rop id$_2$
{ E.place : = newtemp ;
emit (jrop, id$_1$.place, id$_2$.place, nextstat+3);
emit (: = , 0 , - , E.place) ;
emit (jump , - , - , nextstat+2) ;
emit (: = , 1 , - , E.place) }

(1)(j<, a, b, (4))
(2)(:=, 0, - , t_1)
(3)(jump,- ,- ,(5))
(4)(:=, 1, - , t_1)
(5)(j<, c, d, (8))
(6)(:=, 0, - , t_2)
(7)(jump, - , - ,(9))
(8)(:=, 1, - , t_2)
(9)(j>, e, f, (12))
(10)(:=, 0, - , t_3)
(11)(jump,- ,- ,(13))
(12)(:=, 1, - , t_3)

2. 直接计算的翻译

- 例：布尔表达式a<b or c<d and e>f的翻译



(2)E→E1 and E2
{ E.place : = newtemp ;
emit(and, E1.place, E2.place , E.place)}

(1)(j<, a, b, (4))

(2)(:=, 0, - , t_1)

(3)(jump,- ,- ,(5))

(4)(:=, 1, - , t_1)

(5)(j<, c, d, (8))

(6)(:=, 0, - , t_2)

(7)(jump, - , - ,(9))

(8)(:=, 1, - , t_2)

(9)(j>, e, f, (12))

(10)(:=, 0, - , t_3)

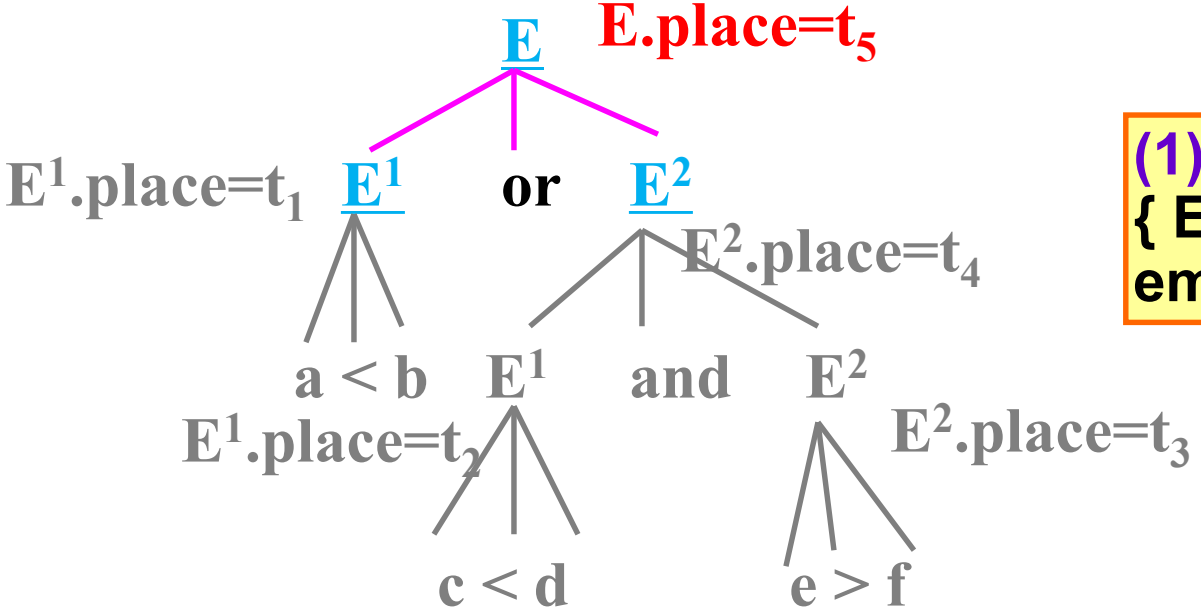
(11)(jump,- ,- ,(13))

(12)(:=, 1, - , t_3)

(13)(and, t_2, t_3, t_4)

2. 直接计算的翻译

- 例：布尔表达式a<b or c<d and e>f的翻译



(1)E→E1 or E2
{ E.place : = newtemp ;
emit (or, E1.place , E2.place , E.place)}

(1)(j<, a, b, (4))

(2)(:=, 0, - , t_1)

(3)(jump,- ,- ,(5))

(4)(:=, 1, - , t_1)

(5)(j<, c, d, (8))

(6)(:=, 0, - , t_2)

(7)(jump, - , - ,(9))

(8)(:=, 1, - , t_2)

(9)(j>, e, f, (12))

(10)(:=, 0, - , t_3)

(11)(jump,- ,- ,(13))

(12)(:=, 1, - , t_3)

(13)(and, t_2, t_3, t_4)

(14)(or, t_1, t_4, t_5)

2. 直接计算的翻译

- 例：布尔表达式a<b or c<d and e>f的翻译

(1)(j<, a, b, (4))
(2)(:=, 0, - , t_1)
(3)(jump,- ,- ,(5))
(4)(:=, 1, - , t_1)
(5)(j<, c, d, (8))
(6)(:=, 0, - , t_2)
(7)(jump, - , - ,(9))
(8)(:=, 1, - , t_2)
(9)(j>, e, f, (12))
(10)(:=, 0, - , t_3)
(11)(jump,- ,- ,(13))
(12)(:=, 1, - , t_3)
(13)(and, t_2, t_3, t_4)
(14)(or, t_1, t_4, t_5)

100: if a < b goto 103
101: t_1 = 0
102: goto 104
103: t_1 = 1
104: if c < d goto 107
105: t_2 = 0
106: goto 108
107: t_2 = 1
108: if e < f goto 111
109: t_3 = 0
110: goto 112
111: t_3 = 1
112: t_4 = t_2 and t_3
113: t_5 = t_1 or t_4

四元式编号从100开始

3. 短路计算

- 条件控制语句

  S → **if** (B) S1

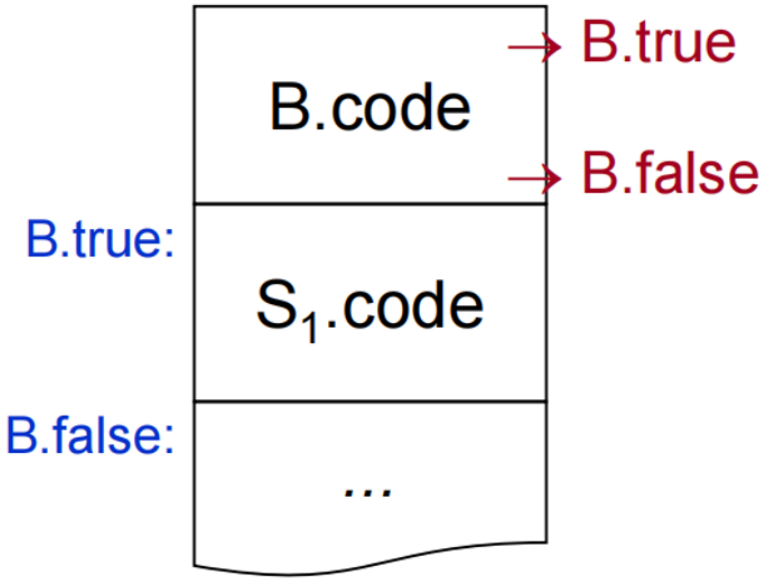
  S → **if** (B) S1 **else** S2

  S → **while** (B) S1

- 短路计算
  - 布尔表达式计算到某一部分就可以得到结果，而无需对布尔表达式进行完全计算
    - ✓ A or B　　if A then 1 else B
    - ✓ A and B　　if A then B else 0
    - ✓ not A　　if A then 0 else 1

3. 短路计算

- 为布尔表达式B引入两个新的属性：
  - B.true：表达式的真出口，它指向表达式为真时的转向
  - B.false：表达式的假出口，它指向表达式为假时的转向

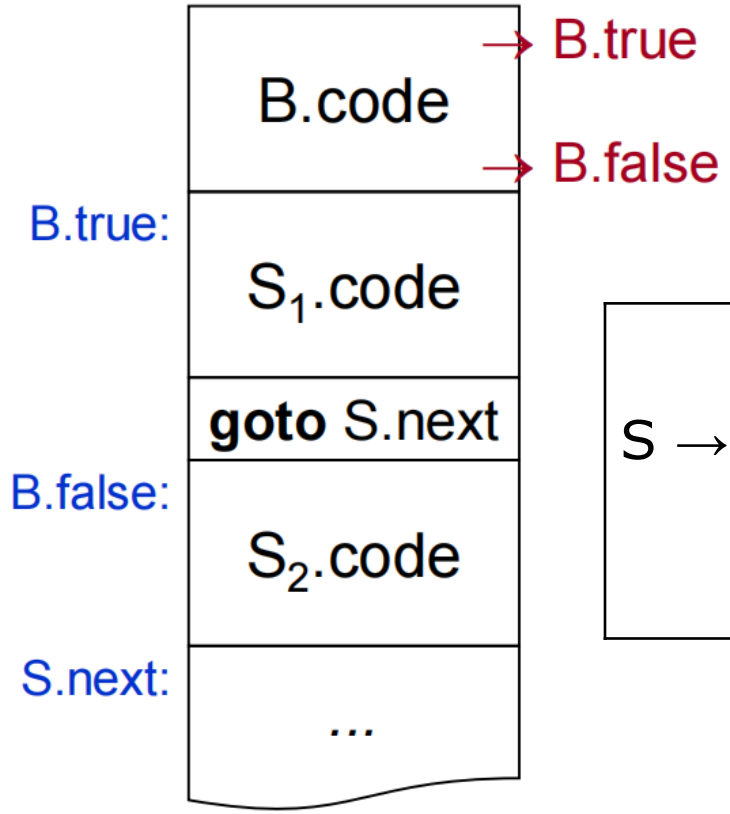


S → if (B) S1

S → **if** (B) S1	B.true = **new** Label(); B.false = S1.next = S.next; S.code = B.code || label(B.true) || S1.code

3. 短路计算

- 为布尔表达式B引入两个新的属性：
  - B.true：表达式的真出口，它指向表达式为真时的转向
  - B.false：表达式的假出口，它指向表达式为假时的转向



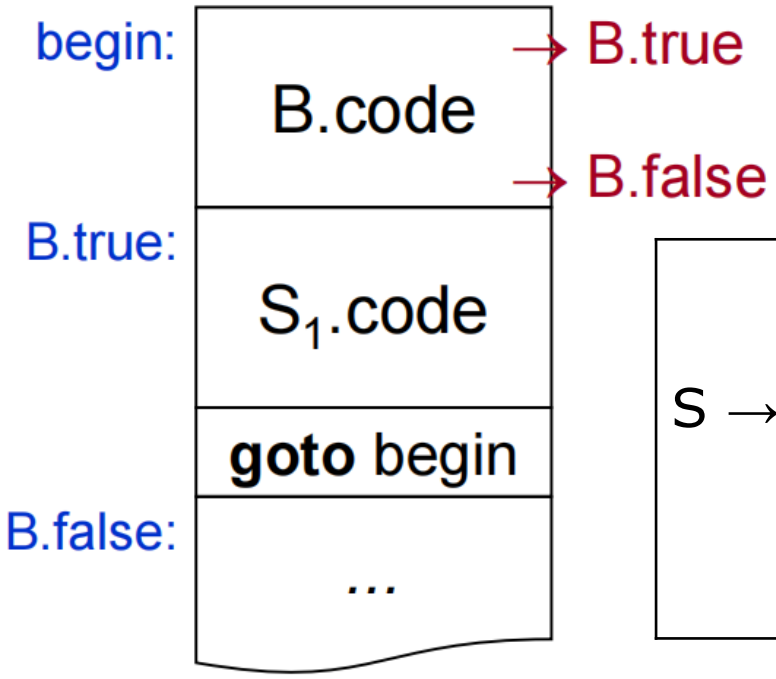
S → **if** (B) S1 **else** S2

S.code = B.code || label(B.true) || S1.code || gen('**goto**' S.next) || label(B.false) || S2.code

S → if (B) S1 else S2

3. 短路计算

- 为布尔表达式B引入两个新的属性：
  - B.true：表达式的真出口，它指向表达式为真时的转向
  - B.false：表达式的假出口，它指向表达式为假时的转向



S → while (B) S1

S → **while** (B) S1	begin = **new** Label(); B.true = **new** Label(); B.false = S.next; S1.next = begin; S.code = label(begin) || B.code || label(B.true) || S1.code || gen('**goto**' begin)

3. 短路计算

Productions	Semantic Rules
P → S	S.next = **new** Label(); P.code = S.code || label(S.next)
S → **assign**	S.code = **assign**.code
S → S_1 S_2	S_1.next = **new** Label(); S_2.next = S.next; S.code = S_1.code || label(S_1.next) || S_2.code
S → **if** (B) S_1	B.true = **new** Label(); B.false = S_1.next = S.next; S.code = B.code || label(B.true) || S_1.code
S → **if** (B) S_1 **else** S_2	B.true = **new** Label(); B.false = **new** Label(); S_1.next = S_2.next = S.next; S.code = B.code || label(B.true) || S_1.code || gen('**goto**' S.next) || label(B.false) || S_2.code
S → **while** (B) S_1	begin = **new** Label(); B.true = **new** Label(); B.false = S.next; S_1.next = begin; S.code = label(begin) || B.code || label(B.true) || S1 .code || gen('**goto**' begin)

控制流语句的语法制导定义

3. 短路计算

为布尔表达式生成三地址码

Productions	Semantic Rules
B → B1 **||** B2	B_1 .true = B.true; B_1 .false = **new** Label(); B_2 .true = B.true; B_2 .false = B.false; B.code = B_1 .code || label(B_1 .false) || B_2 .code
B → B1 **&&** B2	B_1 .true = **new** Label(); B_1 .false = B.false; B_2 .true = B.true; B_2 .false = B.false; B.code = B_1 .code || label(B_1 .true) || B_2 .code
B → **!** B1	B_1 .true = B.false; B_1 .false = B.true; B.code = B_1 .code
B → E1 **relop** E2	B.code = E_1 .code || E_2 .code || gen('**if**' E_1 .addr **relop**.op E_2 .addr '**goto**' B.true) || gen('**goto**' B.false)
B → **true**	B.code = gen('**goto**' B.true)
B → **false**	B.code = gen('**goto**' B.false)

3. 短路计算

- 例：条件控制语句**if (x < 100 || x > 200 && x != y) x = 0**的翻译

```
        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 :    …
```

3. 短路计算

- 例：条件控制语句 a<b or c<d and e>f 的翻译

(1) (j<, a, b, E.true)
(2) (jump, - , - , (3))
(3) (j<, c , d , (5))
(4) (jump, - , - , E.false)
(5) (j>, e , f , E.true)
(6) (jump, - , - , E.false)

CONTENTS

目 录

1. 回填技术的提出

- 遇到的问题
  - 在把布尔式翻译成一串条件转和无条件转四元式时，真假出口**未能**在生成四元式时确定
  - 多个四元式可能有**相同的出口**
- 解决办法：
  - 真假出口的**拉链与回填[backpatching]**

1. 回填技术的提出

- 例：条件控制语句 **a<b or c<d and e>f** 的翻译

(1) (j<, a, b, E.true)

(2) (jump, - , - , (3))

(3) (j<, c , d , (5))

(4) (jump, - , - , E.false)

(5) (j>, e , f , E.true)

(6) (jump, - , - , E.false)

  - E.true和E.false不能在产生四元式时确定，要等**将来目标明确时再回填**，为此要记录这些要回填的四元式
  - 通常采用“**拉链**”的办法，把需要回填**E.true**的四元式拉成一条“**真**”**链**，把需要回填**E.false**的四元式拉成一条“**假**”**链**

2. 拉链方式

若有四元式序列：	则链接成为：
(10) (*, *, *, E.true)	(10) (*, *, *, **0**)
……	……
(20) (*, *, *, E.true)	(20) (*, *, *, **10**)
……	……
(30) (*, *, *, E.true)	(30) (*, *, *, **20**)

– 把地址（30）作为链首，地址（10）作为链尾，0为链尾标志。

– 四元式的第四个区段存放链指针。

– **E.true 和E.false用于存放“真”链和“假”链的链首。**

3. 回填的翻译方案

- 语义：
  - 函数**merge (p1, p2)** 用于把p1和p2为链首的两条链合并成1条，返回合并后的链首值。
    - ✓ 当p2为空链时，返回p1；
    - ✓ 当p2不为空链时，把p2的链尾第四区段改为p1，返回p2。
  - 函数**backpatch (p, t)** 用于把链首p所链接的每个四元式的第四区段都填为转移目标t【**在知道t具体在哪里之后**】

3. 回填的翻译方案

- 对布尔表达式

$B \rightarrow B_1$ **||** $M\ B_2$ { backpatch(B_1.falseList, M.instruction);
B.trueList = merge(B_1.trueList, B_2.trueList);
B.falseList = B_2.falseList; }

$B \rightarrow B_1$ **&&** $M\ B_2$ { backpatch(B_1.trueList, M.instruction);
B.trueList = B_2.trueList;
B.falseList = merge(B_1.falseList, B_2.falseList); }

$B \rightarrow$ **!** 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; }

$B \rightarrow E_1$ **relop** E_2 { B.trueList = **new** List(nextInstruction);
B.falseList = **new** List(nextInstruction + 1);
emit('**if**' E_1.addr relop.op E_2.addr '**goto** __');
emit('**goto** __'); }

$B \rightarrow$ **true** { B.trueList = **new** List(nextInstruction);
emit('**goto** __'); }

$B \rightarrow$ **false** { B.falseList = **new** List(nextInstruction);
emit('**goto** __'); }

$M \rightarrow \varepsilon$ { M.instruction = nextInstruction; }

3. 回填的翻译方案

- 对布尔表达式

1) $E \rightarrow E^1$ or E^2

{ E.codebegin: = E^1.codebegin ;
backpatch (E^1.false , E^2.codebegin) ;
E.true: = merge (E^1.true , E^2.true) ;
E.false: = E2.false }

2) $E \rightarrow E^1$ and E^2

{ E.codebegin:＝E^1.codebigin ;
backpatch (E^1.true , E^2.codebegin) ;
E.true:＝E^2.true ;
E.false:＝merge (E^1.fasle , E^2.false) }

3) $E \rightarrow$ not E^1

{ E.codebegin:＝E^1.codebegin ;
E.true:＝E^1.false ;
E.false:＝E^1.true }

4) $E \rightarrow (E^1)$

{ E.codebegin:＝E^1.codebegin ;
E.true:＝E^1.true ;
E.false:＝E^1.false }

3. 回填的翻译方案

- 对布尔表达式

5) $E \rightarrow id_1$ rop id_2

{ E.codebegin:＝nextstat ;
E.true:＝nextstat ;
E.false:＝nextstat+1;
emit (jrop , id_1.place , id_2.place , 0) ;
emit (jump ,－,－, 0) }

6) E→true

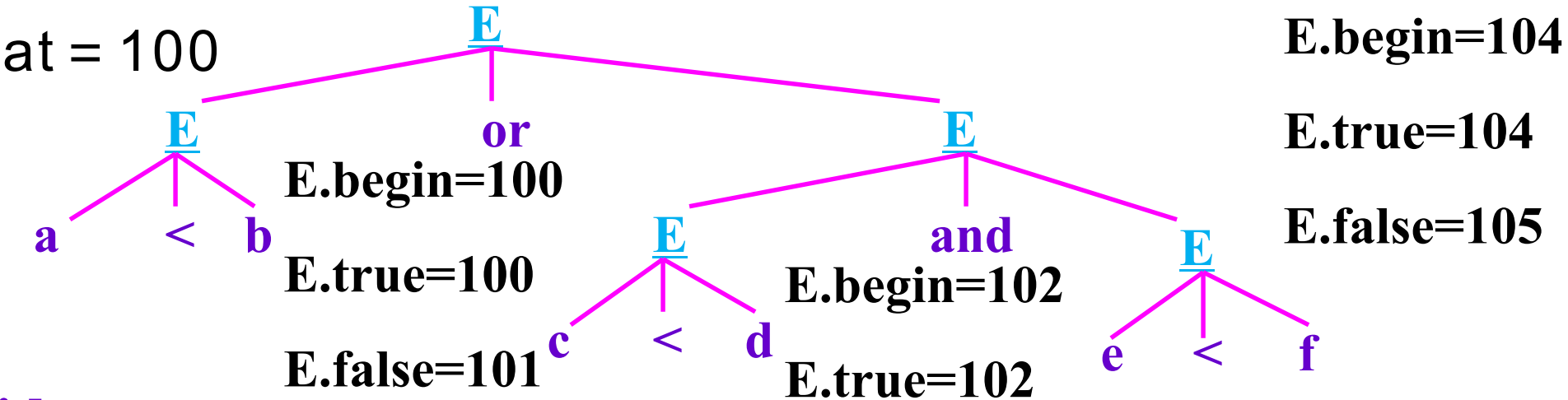
{ E.codebegin:＝nextstat ;
E.true:＝nextstat ;
E.false:＝0;
emit (jump ,－,－,0) }

7) E→false

{ E.codebegin:＝nextstat ;
E.false:＝nextstat ;
E.true:＝0;
emit (jump ,－,－,0) }

3. 回填的翻译方案

- 例：a<b or c<d and e<f 的翻译过程，假定四元式编号从100开始，即开始时nextstat = 100



5) $E \rightarrow id_1\ rop\ id_2$

{ E.codebegin:＝nextstat ;

E.true:＝nextstat ;

E.false:＝nextstat+1;

emit (jrop , id_1.place , id_2.place , 0) ;

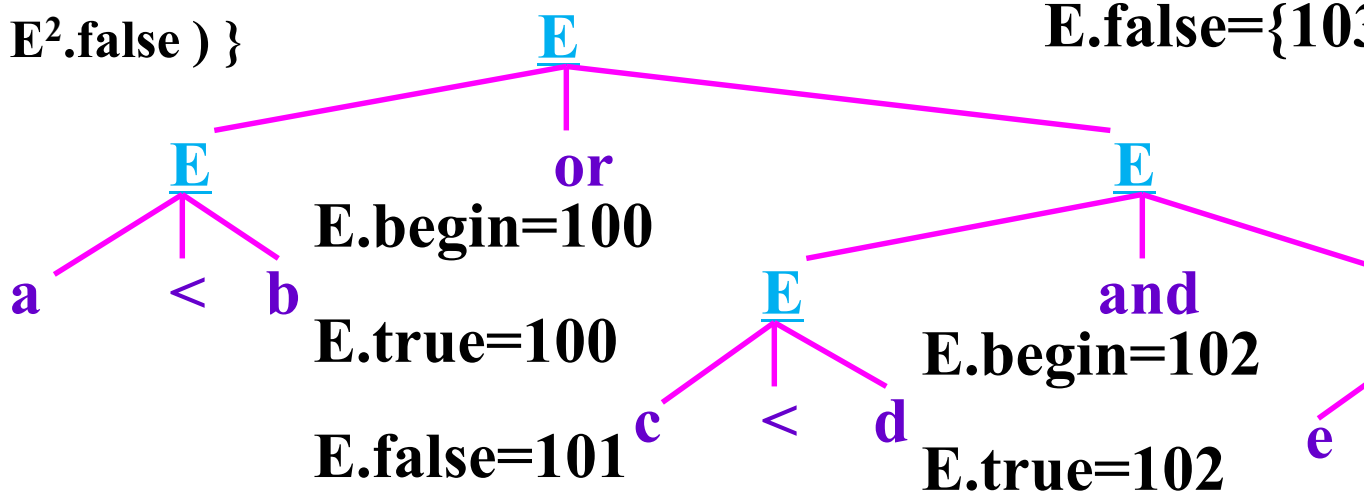
emit (jump ,－,－, 0) }

100 : (j< , a , b , 0)
101: (jump ,—,—,0)
102: (j< , c , d , 0)
103: (jump ,－,－, 0)
104: (j< , e , f , 0)
105: (jump ,－,－,0)

3. 回填的翻译方案

2) $E \rightarrow E^1$ and E^2

{ E.codebegin:＝E^1.codebigin ;
backpatch (E^1.true , E^2.codebegin) ;
E.true:＝E^2.true ;
E.false:＝merge (E^1.fasle , E^2.false) }



E.begin=102

E.true=104

E.false={103,105}=105

E.begin=100

E.true=100

E.false=101

E.begin=102

E.true=102

E.false=103

E.begin=104

E.true=104

E.false=105

100 : (j< , a , b , 0)
101: (jump ,—,—,0)
102: (j< , c , d , 0) ⟶ 102 (j< , c , d ,104)
103: (jump ,—,—, 0)
104: (j< , e , f , 0)
105: (jump ,—,—,0) ⟶ 105 (jump ,—,—,103)

3. 回填的翻译方案

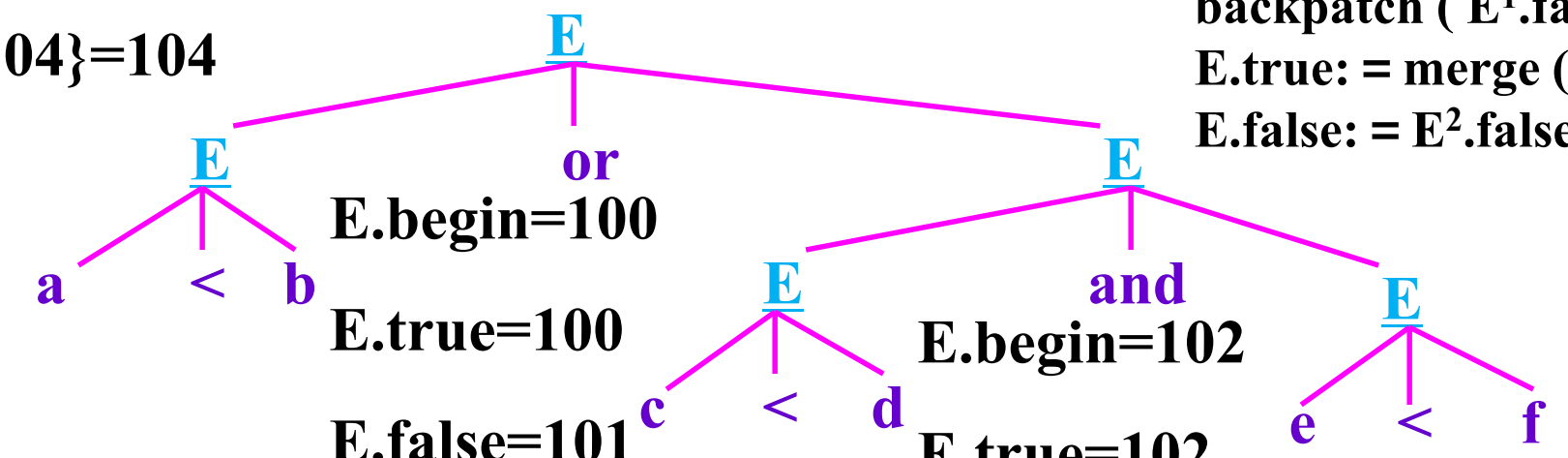
1) $E \rightarrow E^1$ or E^2

{ E.codebegin: = E^1.codebegin ;
backpatch (E^1.false , E^2.codebegin) ;
E.true: = merge (E^1.true , E^2.true) ;
E.false: = E^2.false }

E.begin=100

E.true={100,104}=104

E.false=105



100 : (j< , a , b , 0)

101: (jump ,—,—,0) ⟶ 101(jump,—,—,102)

102: (j< , c , d , 0) ⟶ 102 (j< , c , d ,104)

103: (jump ,—,—, 0)

104: (j< , e , f , 0) ⟶ 104 (j< , e , f ,100)

105: (jump ,—,—,0) ⟶ 105 (jump ,—,—,103)

3. 回填的翻译方案

- 例：a<b or c<d and e<f 的翻译过程，假定四元式编号从100开始，即开始时nextstat = 100
- 最终结果：

100：(j< , a , b , 0)

101：(jump,—,—,102)

102：(j< , c , d , 104)

103：(jump, - , - , 0)

104：(j< , e , f , 100)

105：(jump, - , - ,103)

“真”链首E.true = 104，“假”链首E.false = 105。

3. 回填的翻译方案

- 对条件控制语句

$S \rightarrow$ **if (** B **)** M S_1 { backpatch(B.trueList, M.instruction);
S.nextList = merge(B.falseList, S_1.nextList); }

$S \rightarrow$ **if (** B **)** M_1 S_1 N **else** M_2 S_2
{ backpatch(B.trueList, M_1.instruction);
backpatch(B.falseList, M_2.instruction);
S.nextList = merge(S_1.nextList, N.nextList, S_2.nextList); }

$S \rightarrow$ **while** M_1 **(** B **)** M_2 S_1 { backpatch(B.trueList, M_2.instruction);
backpatch(S_1.nextList, M_1.instruction);
S.nextList = B.falseList;
emit('**goto**' M_1.instruction); }

$S \rightarrow$ **{** L **}** { S.nextList = L.nextList; }

$S \rightarrow$ A **;** { S.nextList = **new** List(); // Assignment or Atom }

$M \rightarrow \varepsilon$ { M.instruction = nextInstruction; }

$N \rightarrow \varepsilon$ { N.nextList = **new** List(nextInstruction);
emit('**goto** __'); }

$L \rightarrow L_1$ M S { backpatch(L_1.nextList, M.instruction);
L.nextList = S.nextList; }

$L \rightarrow$ S { L.nextList = S.nextList; }