北京大学计算机学院

2024 年春季学期　　《编译原理》

第五讲　中间表示

Intermediate Representations

主要内容

- 中间表示的作用
- 中间表示的设计
- 中间表示的生成

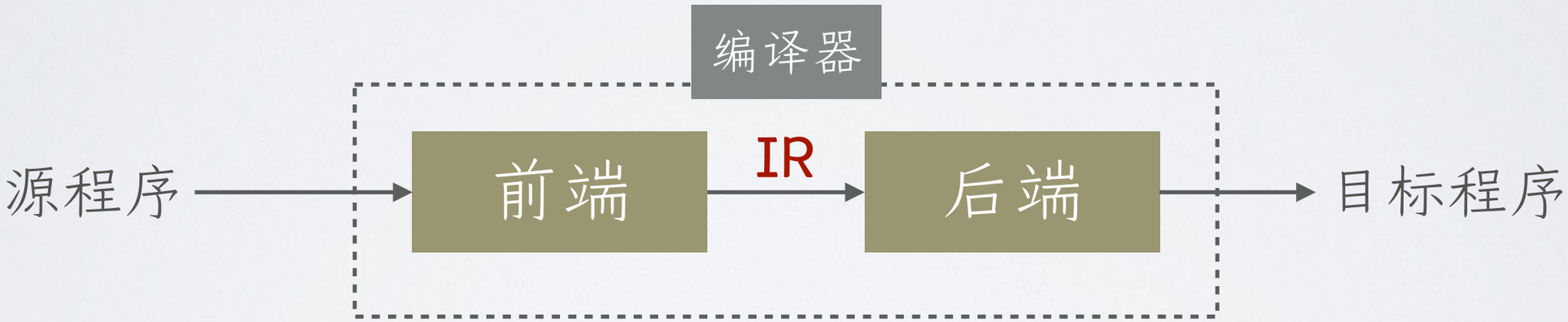
- 对应章节：第 6 章

主要内容

- **中间表示的作用**
- 中间表示的设计
- 中间表示的生成

回顾：编译器的基本结构

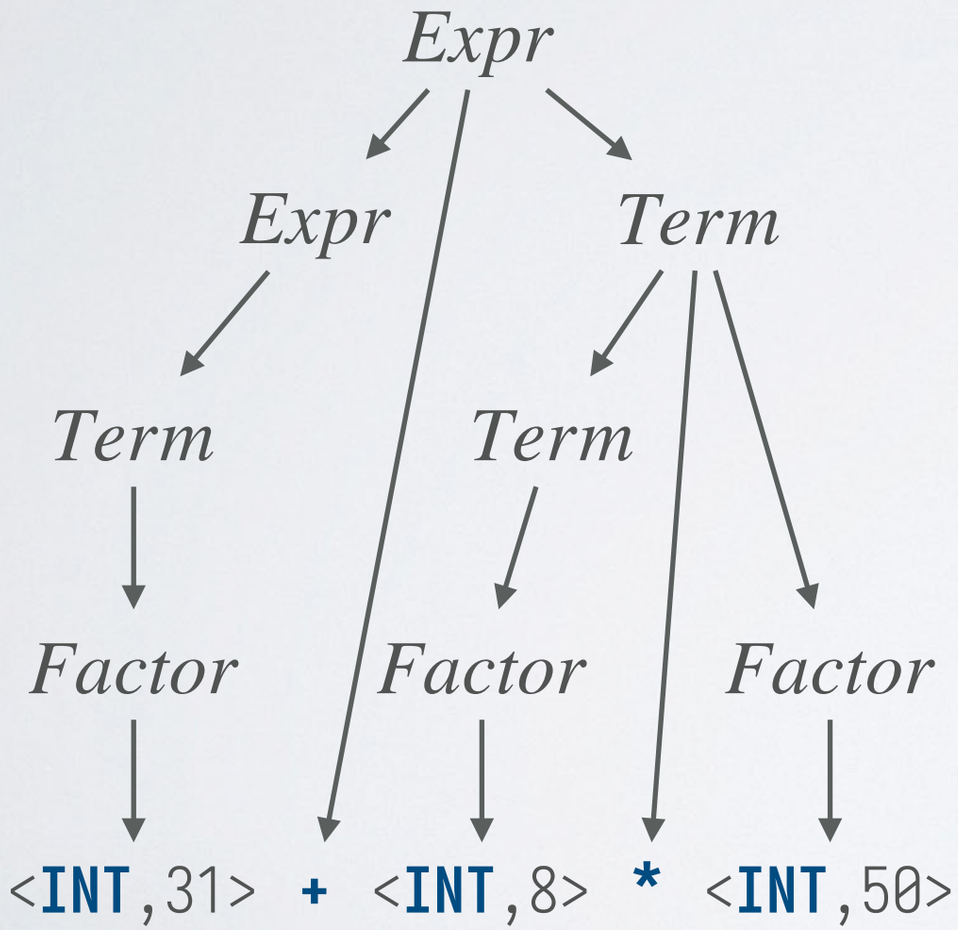
- 翻译过程要求编译器既要理解源程序，也要理解目标机器
- 两个不同的任务驱动了基于**前端**和**后端**的设计



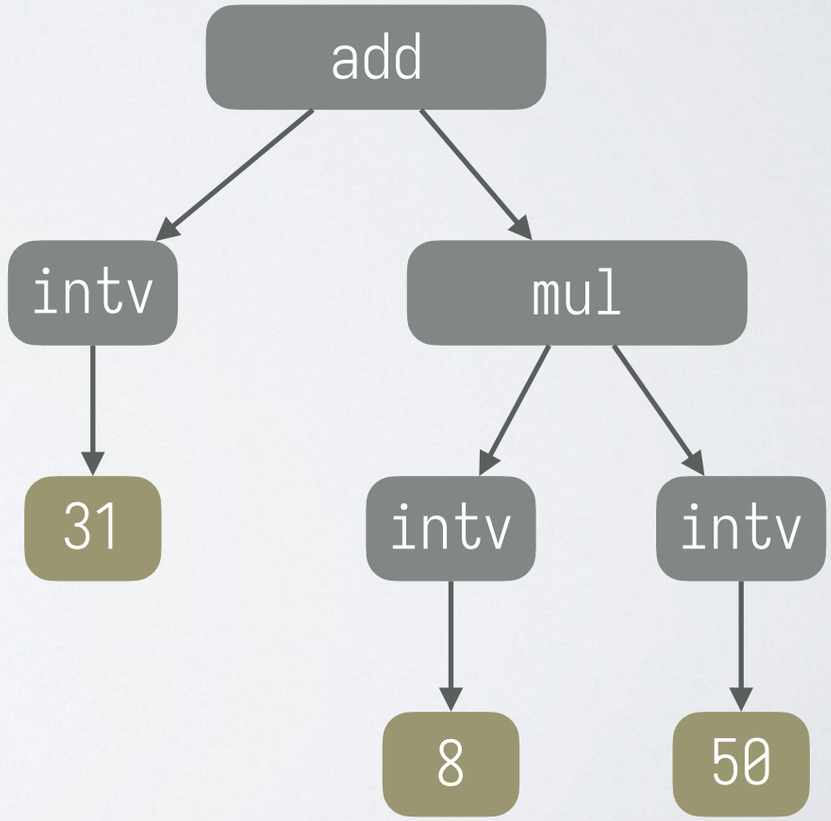
- **前端**：理解源程序，得到语义
- **后端**：把语义翻译为目标程序
- **中间表示**(Intermediate Representation，IR)：编译器对语义的表示形式

回顾：抽象语法树

```
31 + 8 * 50
```

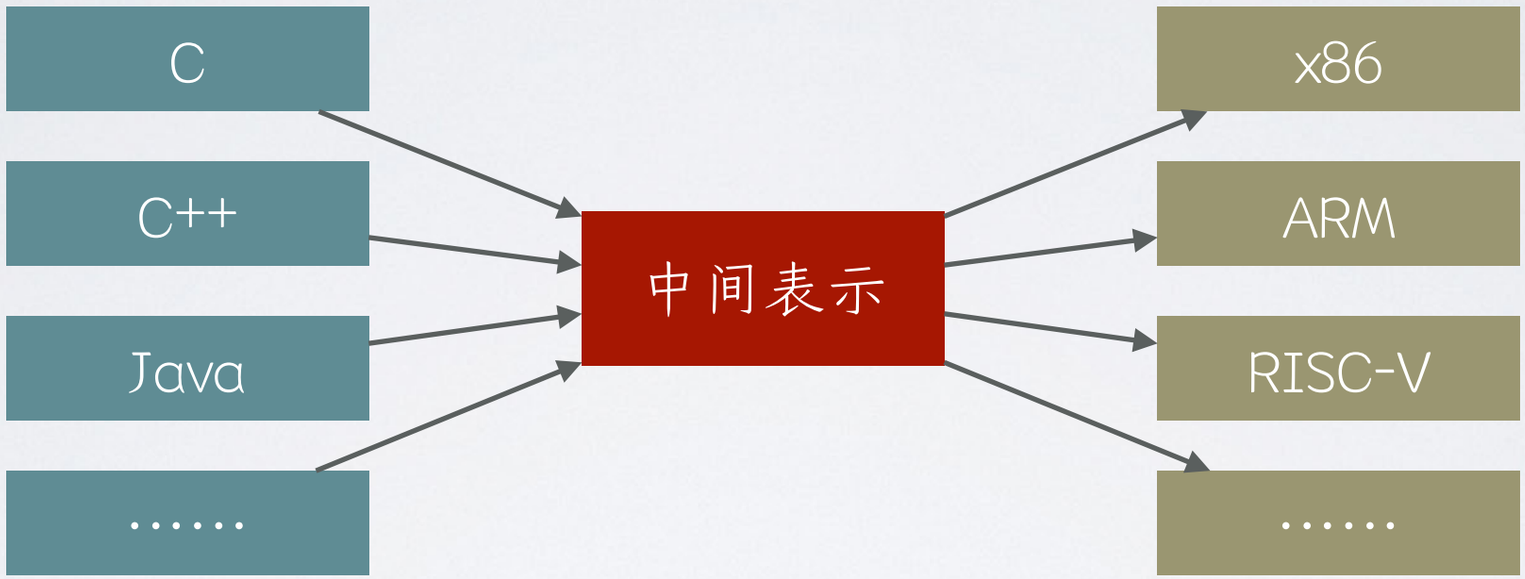



```
add(intv(31), mul(intv(8), intv(50)))
```



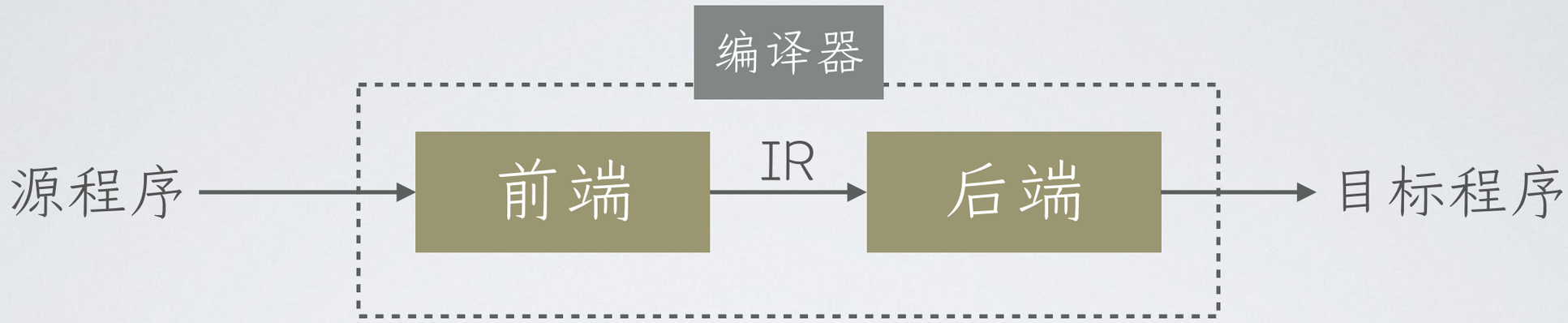
回顾：重定目标（Retargeting）

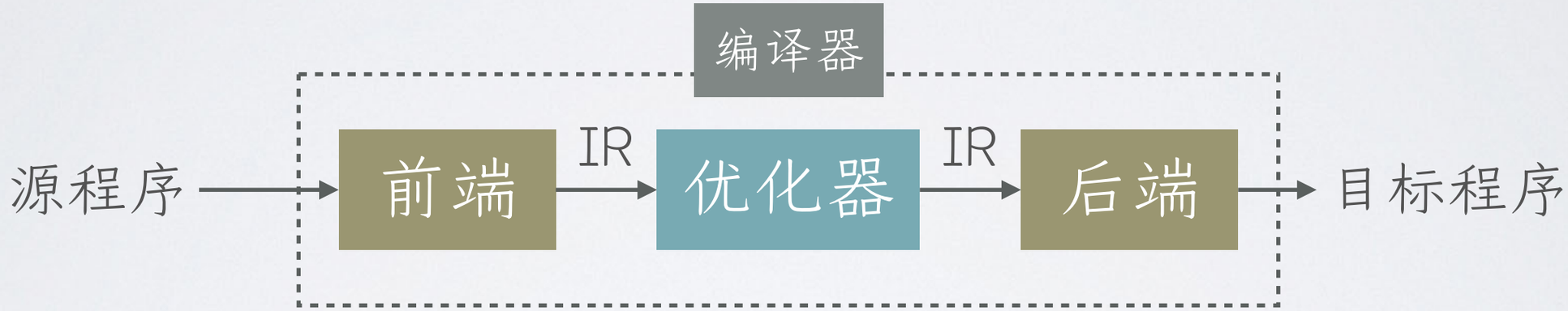
- 重定目标一般指改变编译器使得其为别的处理器生成代码
- 现代编译器构造同时面对**多种源语言**和**多种目标机器**



- 通过引入中间表示(IR)，可以在不同的编译器间复用代码
- **LLVM**：一种流行的编译器中间表示形式

回顾：从两阶段到三阶段





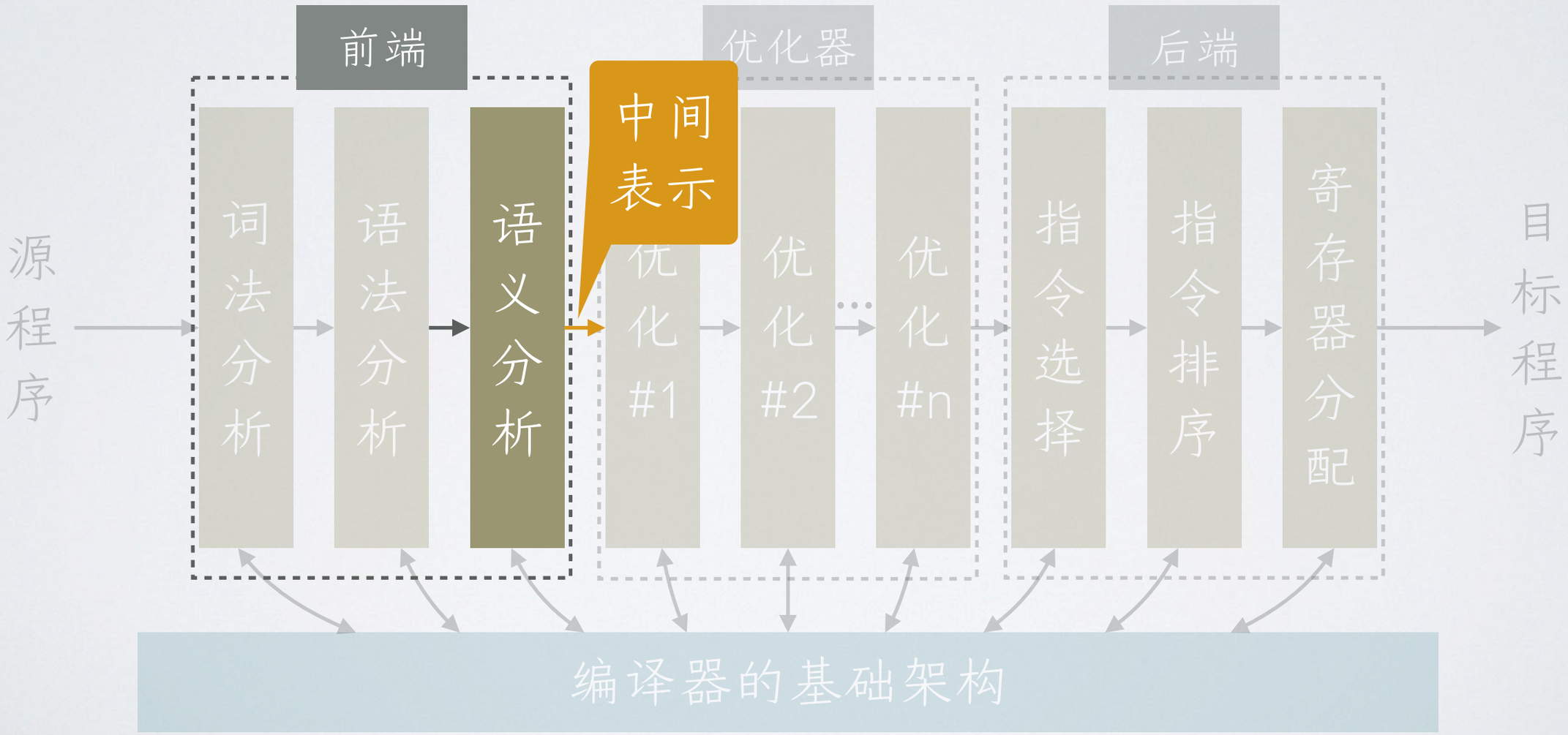
- **优化器：**负责分析、改进、转换中间表示(IR)
- 现代编译器中，优化是至关重要的阶段

中间表示的作用

- Intermediate Representation, IR
- 表示程序语义
  - 例：抽象语法树
- 解耦前端和后端
  - 通用 IR：既与源语言无关，也与目标机器无关
- 简化优化器的设计
  - 多趟处理：每一趟在 IR 上进行简单的分析或转换
  - IR 中可存放额外信息来帮助优化

中间表示的作用

- 在本讲中，重点关注**语义分析**阶段输出的 IR
- 即：如何设计 IR 表示程序语义，如何通过语义分析生成 IR



主要内容

- 中间表示的作用
- **中间表示的设计**
- 中间表示的生成

中间表示的设计考量

- 应该使用什么样的**结构**来组织 IR?
  - 图状 IR、线性 IR、混合 IR

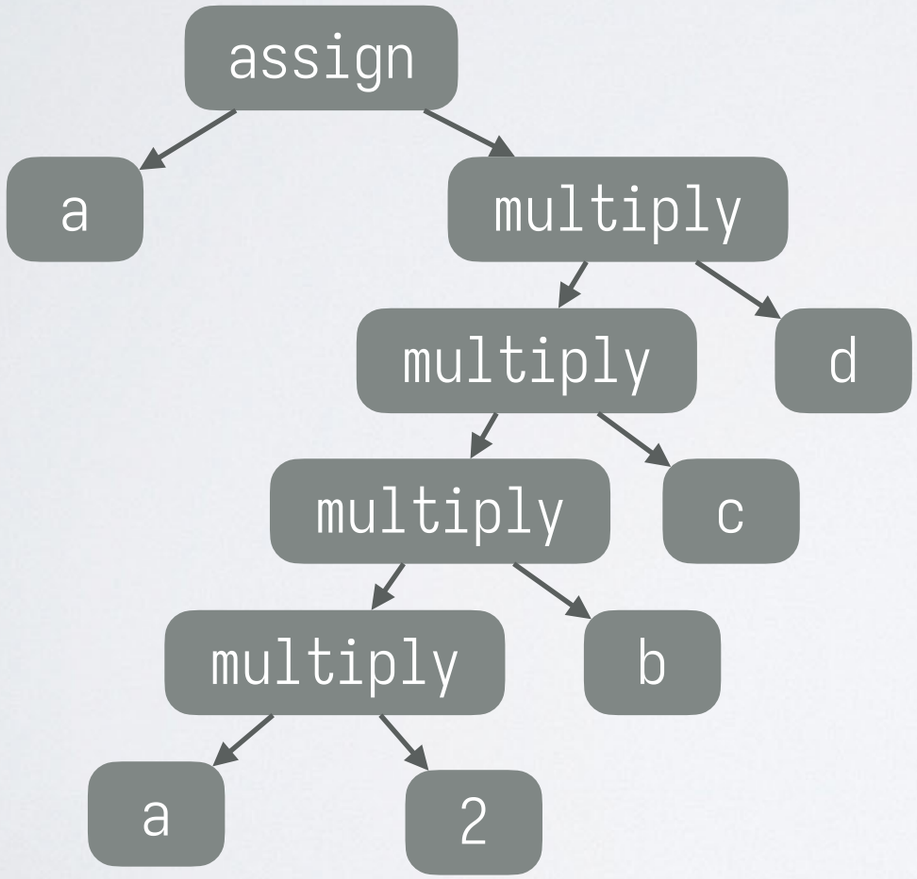
- IR 应该提供何种粒度的**抽象**层次?
  - 更接近源语言、更接近目标机器

- IR 中产生的中间值应该如何**命名**?

- **注意：以上三个维度是相对独立的**

中间表示的组织结构

- **图状 IR**：用树、图等结构表示
  - 例：语法分析树、抽象语法树
  - 例：有向无环图、控制流图
- **线性 IR**：类似汇编的列表形式
  - 例：LLVM IR、Koopa IR
  - 例：三地址代码

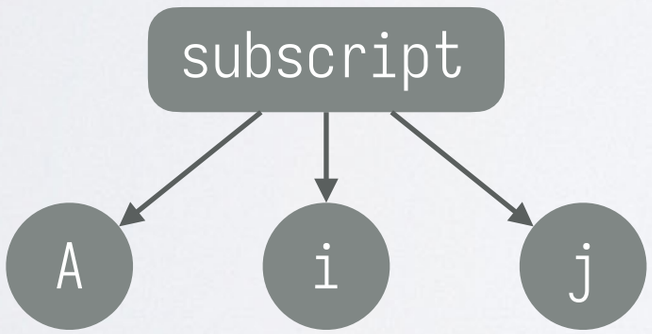



```
t0 = a * 2
t1 = t0 * b
t2 = t1 * c
t3 = t2 * d
a  = t3
```

中间表示的抽象层次

- **所谓「抽象」，不过是隐藏一定的「细节」**
- 例：对于二维数组数组 A[1...10,1...10]，每个元素 4 字节，考虑数组访问 A[i,j]

- **更接近源语言：**
  - 「数组访问」作为整体进行表示
  - 方便对数组进行分析和优化



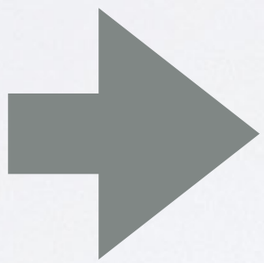
- **更接近目标机器：**
  - 显式地计算数组下标的偏移
  - 方便进行机器有关优化

```
t0 = i - 1
t1 = t0 * 10
t2 = j - 1
t3 = t1 + t2
t4 = t3 * 4
t5 = A [ t4 ]
```

案例：Rust 编译器的

- 图状 IR（类似 AST）
- 更接近源语言
- 简化了语法(比如 for 循环替换为 loop 循环)


```
for elem in vec {
  process(elem);
}
```



```
let mut iterator = vec.into_iter();
loop {
    match iterator.next() {
        Some(elem) => process(elem),
        None => break,
    }
}
```

案例：Rust IR 设计

- 混合 IR（类似控制流图）
- 更接近目标机器，但带有 Rust 的类型信息
- 拥有完整的控制流信息

AST → High-Level IR (HIR) → Mid-Level IR (MIR) → LLVM IR

…

```
bb4:  _12 = &mut _10;
      _11 = …::next(_12) -> [return: bb5, …]

bb5:  _13 = discriminant(_11);
      switchInt(move _13) -> [0: bb8, 1: bb7, …];

bb8:  drop(_10) -> [return: bb9, …];

bb9:  return;

bb7:  _14 = move ((_11 as Some).0: Data);
      _15 = process(move _14) -> [return: bb13, …];

bb13: goto -> bb4;
```

案例：Rust 编译器的 IR 设计

- 线性 IR（类似控制流图）
- 更接近目标机器
- 通用 IR，无 Rust 特定信息


```
......

bb5:                                              ; preds = %bb4
  %11 = extractvalue { i32, i32 } %6, 0, !dbg !2490
  %12 = extractvalue { i32, i32 } %6, 1, !dbg !2490
  store i32 %11, ptr %_11, align 4, !dbg !2490
  %13 = getelementptr inbounds i8, ptr %_11, i64 4, !dbg !2490
  store i32 %12, ptr %13, align 4, !dbg !2490
  %14 = load i32, ptr %_11, align 4, !dbg !2490
  %_13 = zext i32 %14 to i64, !dbg !2490
  %15 = icmp eq i64 %_13, 0, !dbg !2490
  br i1 %15, label %bb8, label %bb7, !dbg !2490

......
```

抽象语法树

- **抽象语法树(Abstract Syntax Tree，AST)**
  - 与语法分析树类似，但去掉了不影响语义的冗余信息

$$
\begin{aligned}
Expr &::= Expr + Term \\
&\mid Expr - Term \\
&\mid Term \\
Term &::= Term * Factor \\
&\mid Term / Factor \\
&\mid Factor \\
Factor &::= (\ Expr\ ) \\
&\mid \texttt{ID} \mid \texttt{INT}
\end{aligned}
$$

语法分析树

AST

$$
\begin{aligned}
E &::= \texttt{add}(E, E) \mid \texttt{sub}(E, E) \mid \texttt{mul}(E, E) \\
&\mid \texttt{div}(E, E) \mid x \mid i
\end{aligned}
$$

$$
\begin{aligned}
BDisj &::= BDisj \mid\mid BConj \\
&\mid BConj \\
BConj &::= BConj \ \&\& \ BCmp \\
&\mid BCmp \\
BCmp &::= Expr == Expr \mid Expr <= Expr \\
&\mid BAtom \\
BAtom &::= (\ BDisj\ ) \mid !\ BAtom \\
&\mid \texttt{true} \mid \texttt{false}
\end{aligned}
$$

语法分析树

AST

$$
\begin{aligned}
B &::= \texttt{or}(B, B) \mid \texttt{and}(B, B) \mid \texttt{eq}(E, E) \mid \texttt{le}(E, E) \\
&\mid \texttt{not}(B) \mid \texttt{true} \mid \texttt{false}
\end{aligned}
$$

抽象语法树

- **抽象语法树(Abstract Syntax Tree，AST)**
  - 与语法分析树类似，但去掉了不影响语义的冗余信息

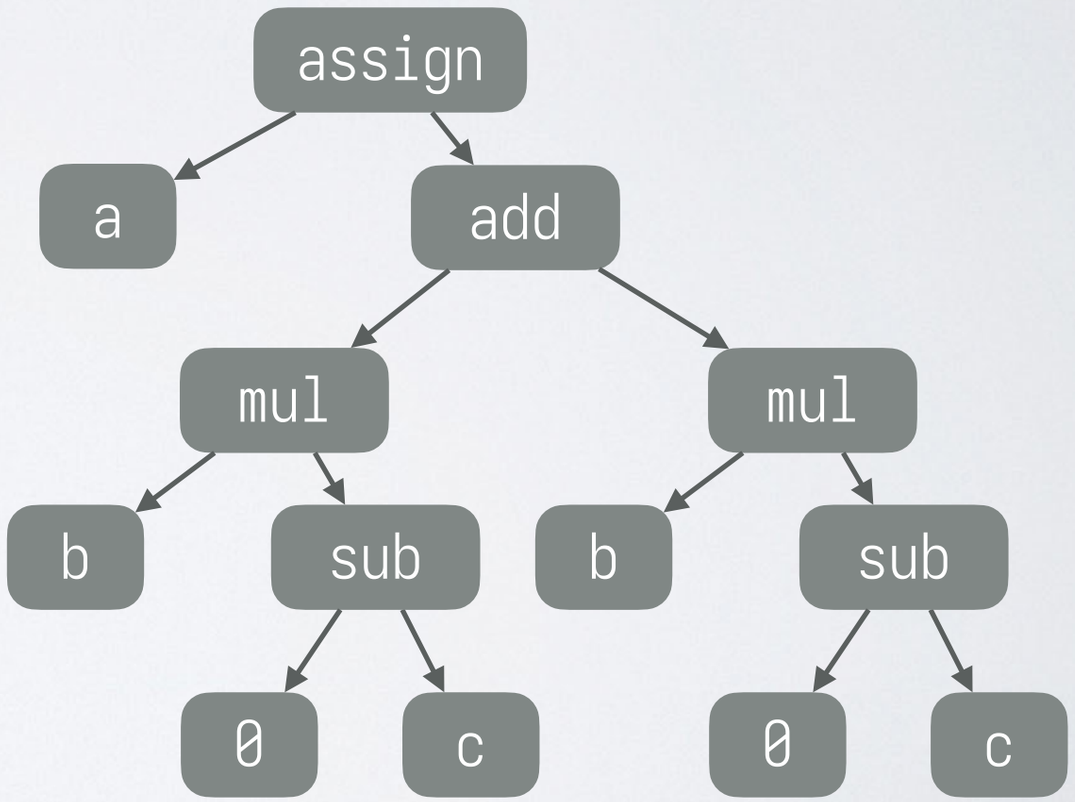
语法分析树

$$
\begin{aligned}
Stmt &::= \{\ Block\ \} \\
&\mid \texttt{ID} = Expr\ ; \\
&\mid \texttt{if}\ (\ BDisj\ )\ Stmt\ \texttt{else}\ Stmt \\
&\mid \texttt{while}\ (\ BDisj\ )\ Stmt \\
&\mid \texttt{return}\ Expr\ ; \\
Block &::= \epsilon \\
&\mid Stmt\ Block
\end{aligned}
$$

AST

$$
\begin{aligned}
S ::= &\ \texttt{block}(\vec{S}) \mid \texttt{assign}(x, E) \mid \texttt{if}(B, S, S) \\
&\mid \texttt{while}(B, S) \mid \texttt{ret}(E)
\end{aligned}
$$

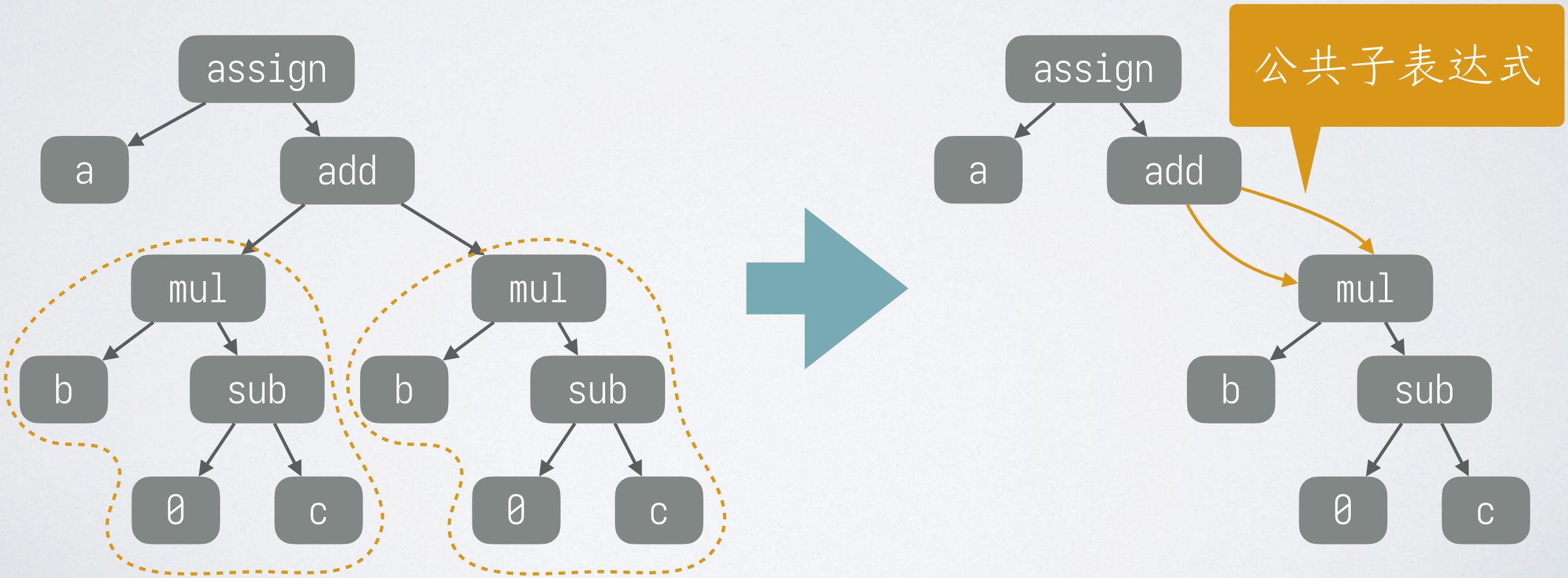
```
a = b * -c + b * -c;
```



有向无环图

- 有向无环图(Directed Acyclic Graph，DAG)
  - 与抽象语法树类似，但对相同的子树进行了合并

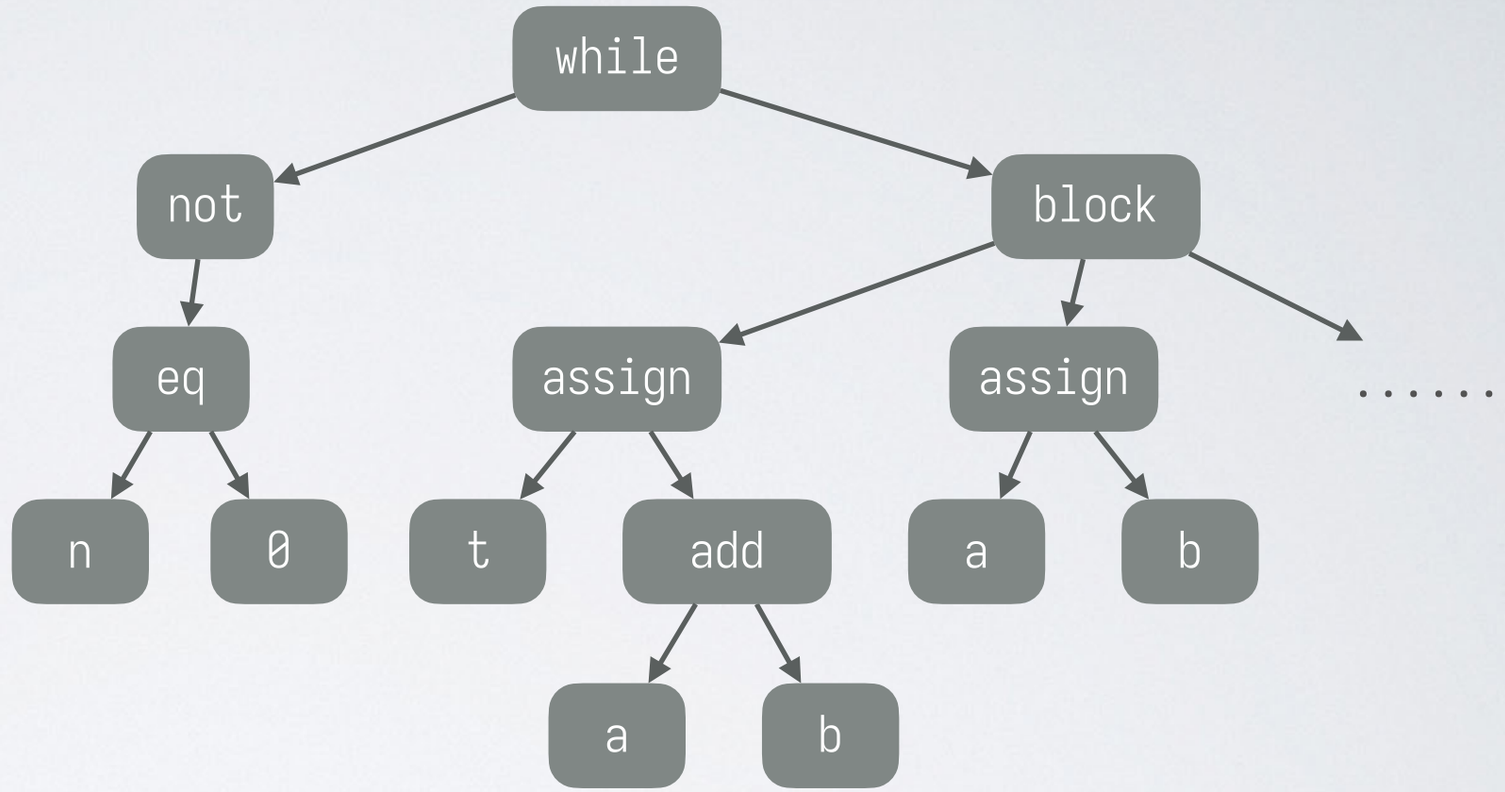
```
a = b * -c + b * -c;
```



控制流图

- 问题：AST、DAG 基本保留了语法结构，没有明确控制流信息

```
while (!(n == 0)) {
  t = a + b; a = b; b = t;
  n = n - 1;
}
```

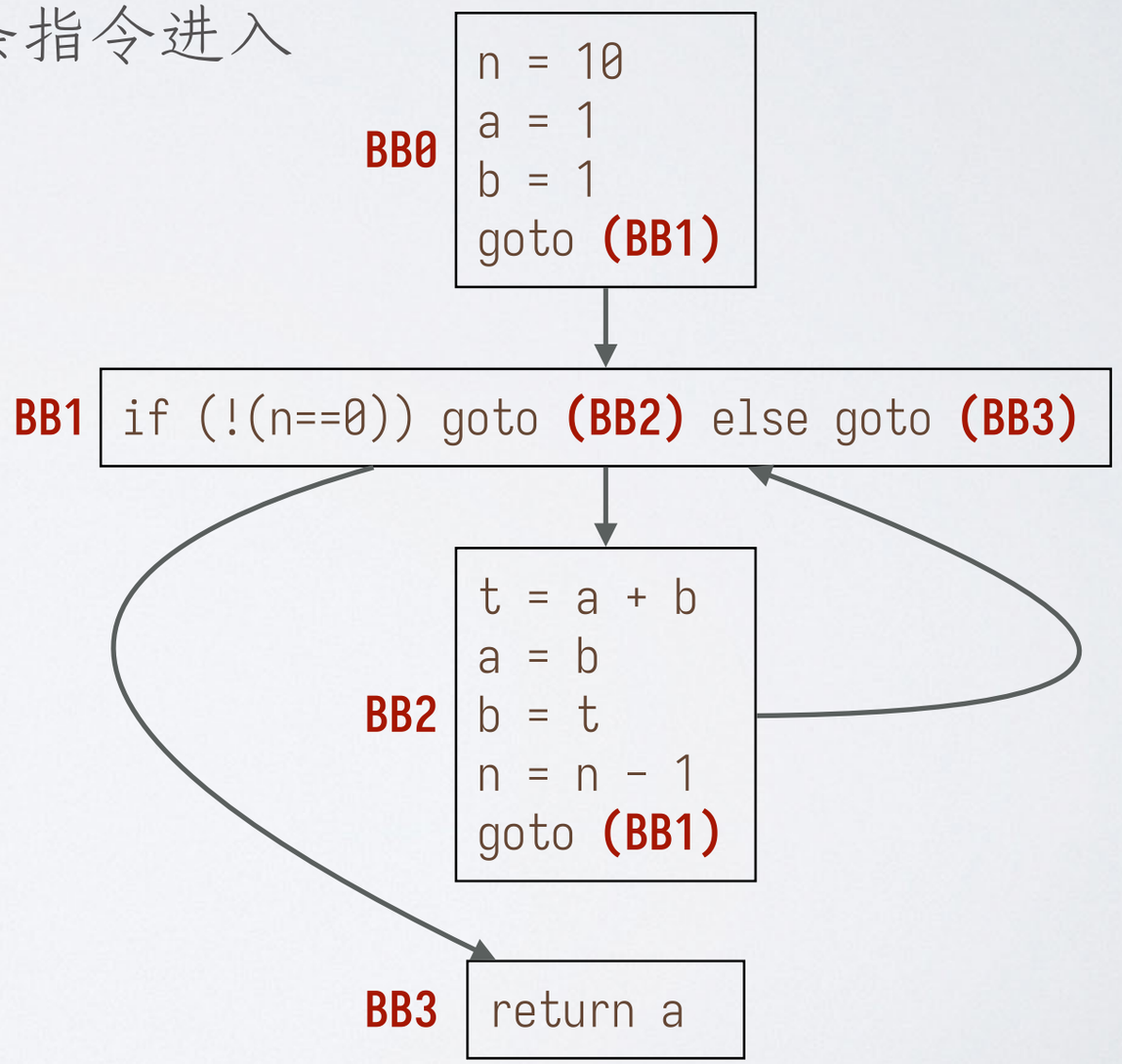


- 控制流图通过**有向图**中边的流向来表示程序的控制流
- 图中结点为语句，或者一串不发生控制流跳转的语句

控制流图

- **控制流图(Control-Flow Graph，CFG)**
  - 有向图，图中结点为**基本块**(basic block)，边为控制流跳转
  - 基本块具有线性结构，其中最后一条语句为**跳转**或者**过程/函数返回**
  - 控制流只能从基本块的第一条指令进入

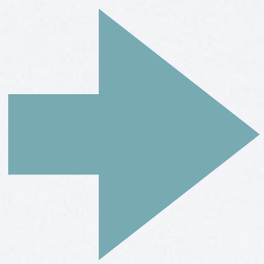
```
{
  n = 10; a = 1; b = 1;
  while (!(n == 0)) {
    t = a + b; a = b; b = t;
    n = n - 1;
  }
  return a;
}
```



三地址代码

- **线性 IR**：类似汇编的列表形式
  - 设计动机：编译器通过简单的线性扫描生成目标代码
- **三地址代码(three-address code)**
  - 每条指令最多涉及三个**地址**(程序变量、临时变量、常量)
  - 所以不支持嵌套表达式等结构

```
a = b * -c + b * -c;
```



$$
\begin{aligned}
t_0 &= 0 - c \\
t_1 &= b * t_0 \\
t_2 &= 0 - c \\
t_3 &= b * t_2 \\
t_4 &= t_1 + t_3 \\
a &= t_4
\end{aligned}
$$

三地址代码的形式

ID 表示地址

I ::= ID = ID *bop* ID | ID = *uop* ID | ID = ID | goto LABEL
| if ID goto LABEL | ifFalse ID goto LABEL | if ID *rop* ID goto LABEL

- $x = y\ bop\ z$: 双目运算符
- $x = uop\ y$: 单目运算符(例如单目减、逻辑非)
- $x = y$: 复制指令
- goto L: 无条件跳转指令, L 表示一条三地址指令的序号
- if/ifFalse x goto L: 条件跳转指令
  - 当 x 为真/假时, 跳转到 L 处
- if/ifFalse $x\ rop\ y$ goto L: 基于关系运算符的条件跳转指令
  - 当 x 和 y 满足/不满足 rop 关系(<、==、>= 等), 跳转到 L 处

三地址代码的形式：过程调用和返回

I ::= … | param ID | call ID, INT | ID = call ID, INT | return ID

- param x：进行参数传递
- call p, n 或 y = call p, n：过程调用和函数调用
  - n 是参数的个数
- return y：返回指令
- 示例：调用有 n 个参数的过程 p

```
param x1
param x2
...
param xn
call p,n
```

三地址代码的形式：数组和指针操作

$$I ::= \cdots \mid \texttt{ID} = \texttt{ID}[\texttt{ID}] \mid \texttt{ID}[\texttt{ID}] = \texttt{ID} \mid \texttt{ID} = \&\,\texttt{ID} \mid \texttt{ID} = *\,\texttt{ID} \mid *\,\texttt{ID} = \texttt{ID}$$

- $x = y[i]$: 把数组 y 中第 i 个元素赋给 x
- $x[i] = y$: 把数组 x 中第 i 个元素设置为 y 的值
- $x = \&\, y$: 把 y 的地址赋给 x
- $x = *y$: 把指针 y 指向的值赋给 x
- $*x = y$: 把指针 x 指向的值设置为 y 的值

三地址代码示例

- 考虑程序语句：
  - `do i = i + 1; while (a[i] < v);`
- 两种可能的翻译：

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

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
```

位置标号

- 通过一次扫描可将符号化标号替换为实际的指令位置

三地址代码的具体实现

- 可以实现为对象(object)或带有相应字段的结构体(struct)

- 四元式(quadruple)
  - $op\ arg_1\ arg_2\ result$

- 三元式(triple)
  - $op\ arg_1\ arg_2$

- 间接三元式(indirect triple)
  - 间接码表 + 三元式表

四元式表示

- 四个字段：*op*、arg_1、arg_2、*result*
  - *op*：运算符的内部编码
  - arg_1、arg_2、*result*：地址
  - 例：$x = y + z$ 对应的四元式为 add　y　z　x

- 一些特例：
  - 单目运算符不使用 arg_2
  - 参数传递指令 param 不使用 arg_2 和 *result*
  - 条件或非条件跳转指令将目标标号放在 *result* 字段

示例：三地址代码的四元式表示

```
a = b * -c + b * -c;
```

	op	arg_1	arg_2	*result*
0	sub	0	c	t_0
1	mul	b	t_0	t_1
2	sub	0	c	t_2
3	mul	b	t_2	t_3
4	add	t_1	t_3	t_4
5	assign	t_4		a

三元式表示

- 三个字段：op、arg_1、arg_2
  - 使用三元式的**位置**来引用三元式的运算结果
- 三元指令需要拆分为两个条目
  - $x[i] = y$：先求 $x[i]$ 的地址，然后再赋值
  - $x = y\ op\ z$：先求 $y\ op\ z$ 的值，然后再赋值
    - (?)　op　y　z
    - 　　assign　x　(?)

示例：三地址代码的三元式表示

```
a = b * -c + b * -c;
```

三元式中使用指向指令的指针

	op	arg_1	arg_2
0	sub	0	c
1	mul	b	**(0)**
2	sub	0	c
3	mul	b	**(2)**
4	add	**(1)**	**(3)**
5	assign	a	**(4)**

间接三元式表示

- **三元式的问题:** 在优化时经常需要移动/删除/添加指令，导致指令位置的移动，引用该指令结果的其它指令都要相应修改
- 用一个单独的列表表示三元式的执行顺序
  - 指令的移动仅需要改变这个执行列表

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

	op	arg_1	arg_2
0	sub	0	c
1	mul	b	(0)
2	sub	0	c
3	mul	b	(2)
4	add	(1)	(3)
5	assign	a	(4)

间接三元式便于进行优化

```
a = b * -c + b * -c;
```

通过提取公共表达式进行优化

instruction	
35	(0)
36	(1)
37	**(4)**
38	**(5)**
39	
40	

	op	arg_1	arg_2
0	sub	0	c
1	mul	b	(0)
2	sub	0	c
3	mul	b	(2)
4	add	(1)	**(1)**
5	assign	a	(4)

三地址代码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

- 优化三地址代码时，四元式比三元式更为方便

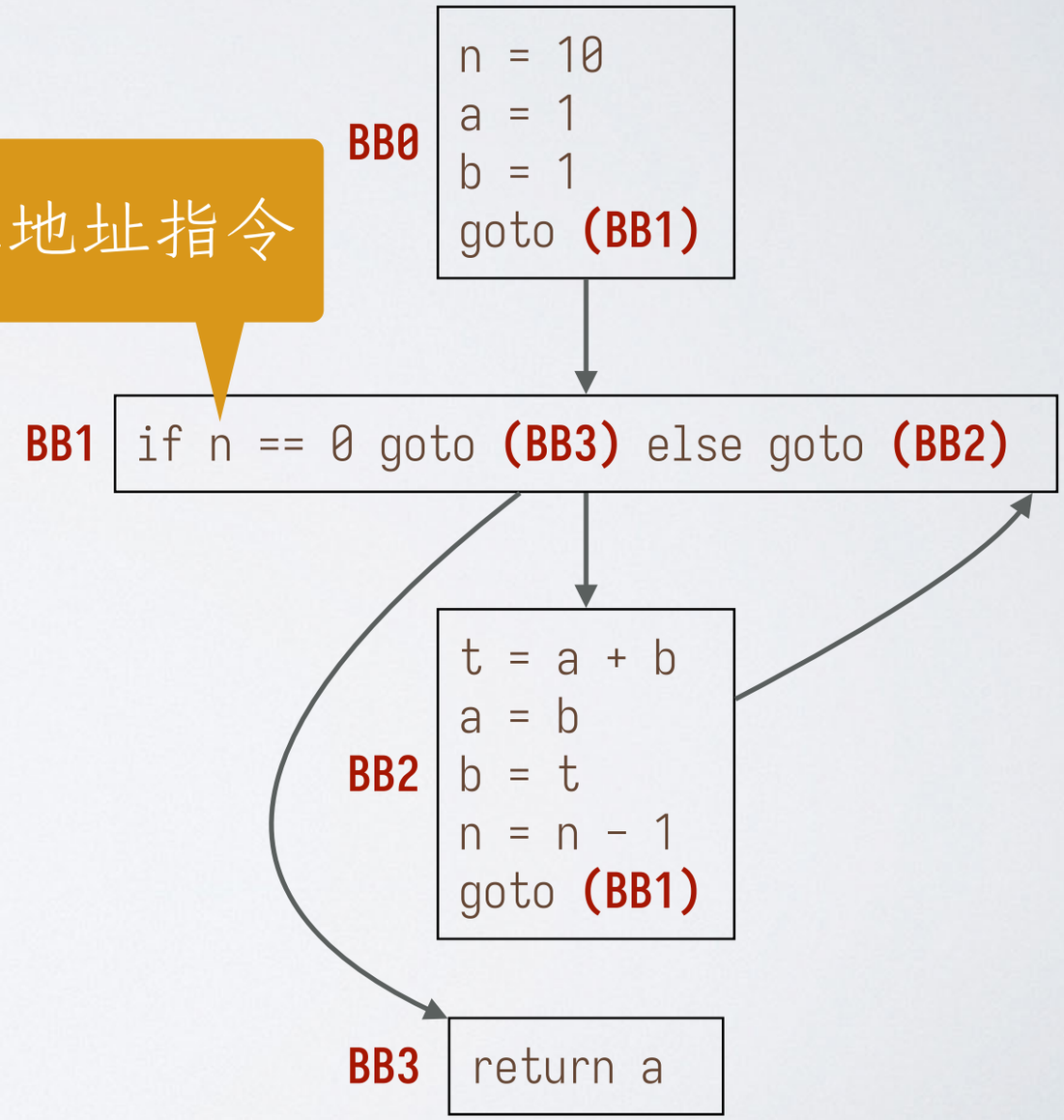
- 间接三元式与四元式同样方便，两种实现方式需要的存储空间大体相同

控制流图 + 三地址代码

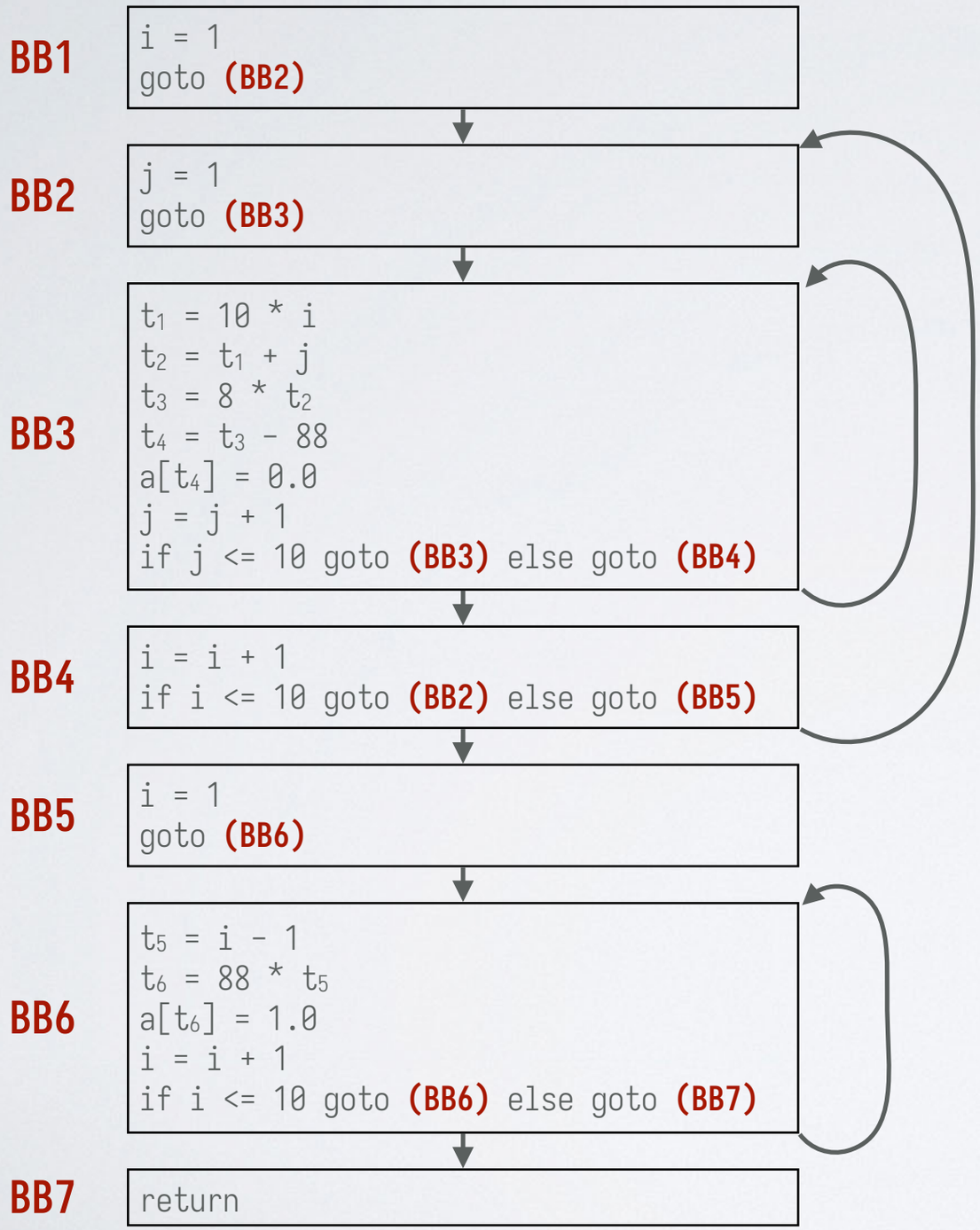
- 控制流图的每个基本块内部为三地址代码
  - 跳转指令的目标为基本块(而不是指令标号)
  - 一种常见的**混合 IR**

```
{
  n = 10; a = 1; b = 1;
  while (!(n == 0)) {
    t = a + b; a = b; b = t;
    n = n - 1;
  }
  return a;
}
```

不是三地址指令



循环的控制流图

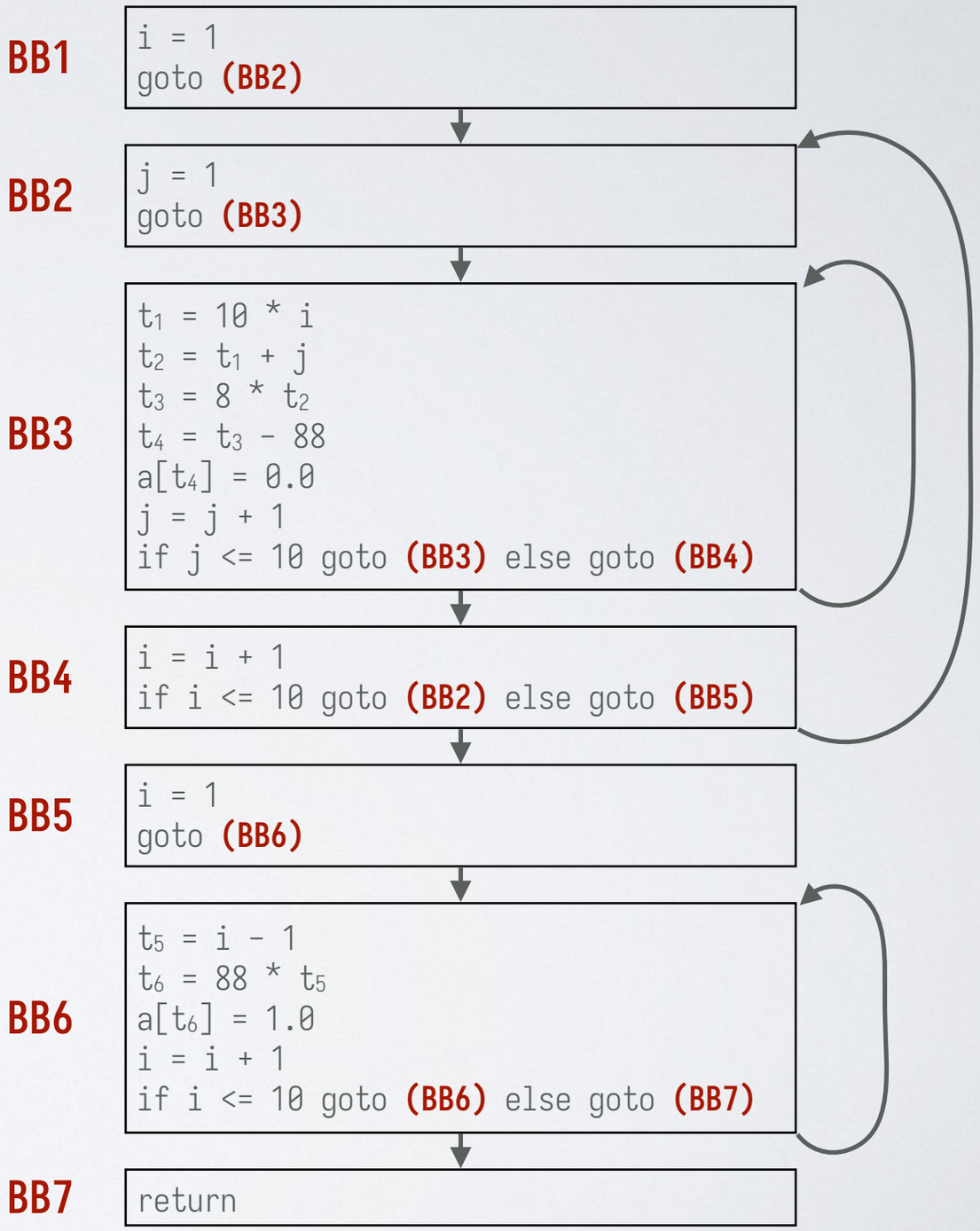


- **循环的定义：**
  - 一个结点集合 L
  - 存在一个**循环入口 (loop entry)** 结点，唯一的前驱可以在 L 之外的结点
  - 每个结点都有到达入口结点的非空路径，且该路径都在 L 中
- 左边控制流图中的循环：
  - {BB3}
  - {BB6}
  - {BB2, BB3, BB4}
    - BB2 为入口结点

从三地址代码构造控制流图

```
(1)  i = 1
(2)  j = 1
(3)  t1 = 10 * i
(4)  t2 = t1 + j
(5)  t3 = 8 * t2
(6)  t4 = t3 - 88
(7)  a[t4] = 0.0
(8)  j = j + 1
(9)  if j <= 10 goto (3)
(10) i = i + 1
(11) if i <= 10 goto (2)
(12) i = 1
(13) t5 = i - 1
(14) t6 = 88 * t5
(15) a[t6] = 1.0
(16) i = i + 1
(17) if i <= 10 goto (13)
```

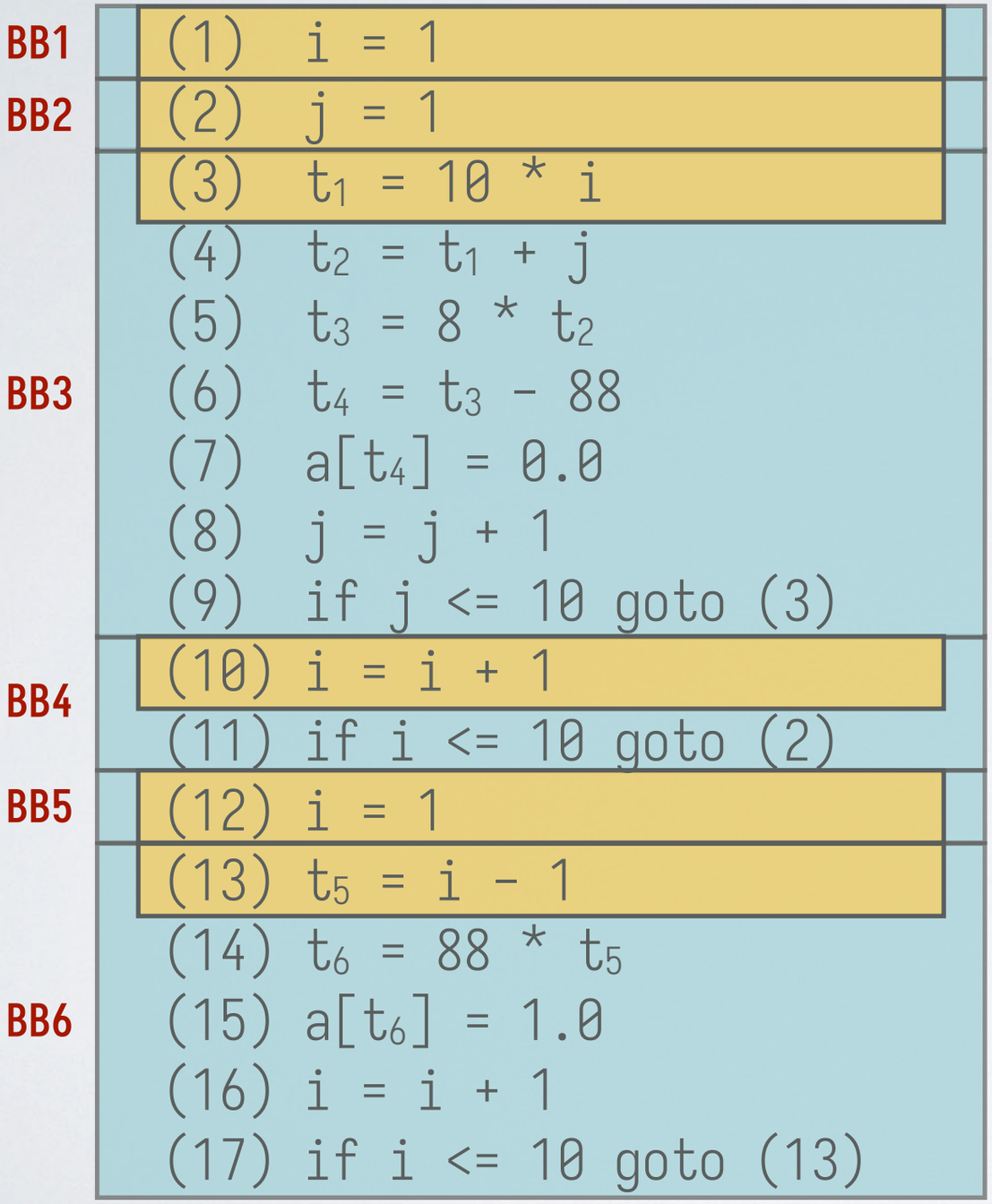
?



划分基本块的算法

- **输入**：三地址指令序列
- **输出**：基本块的列表
- **方法**：
  - 确定**首指令**(leader，基本块的第一条指令)
    - 第一条三地址指令
    - 任意一个条件或无条件跳转指令的目标指令
    - 紧跟在一个条件或无条件跳转指令之后的指令
  - 确定基本块
    - 每条首指令对应一个基本块：从首指令开始到下一个首指令

划分基本块示例



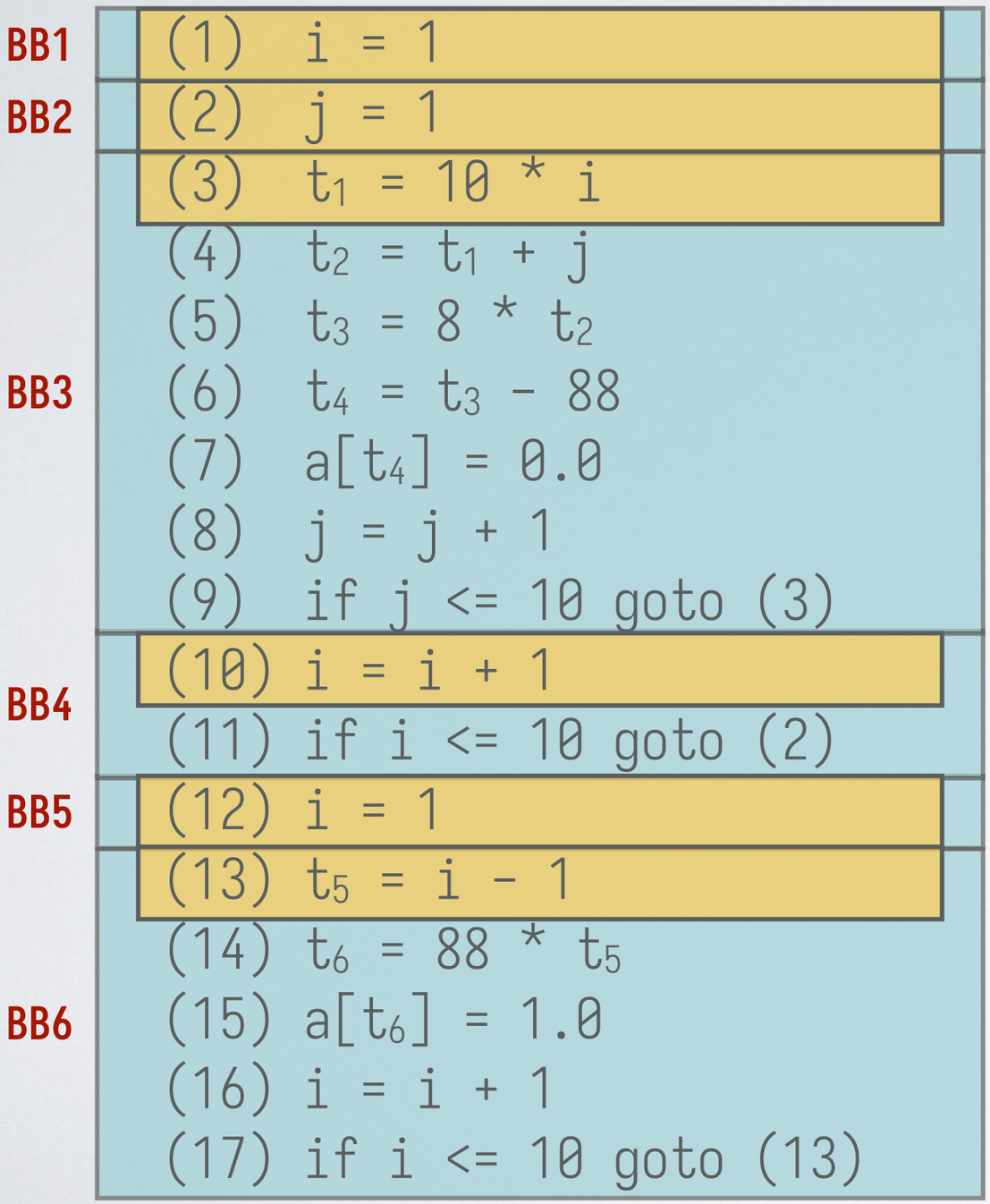
首指令

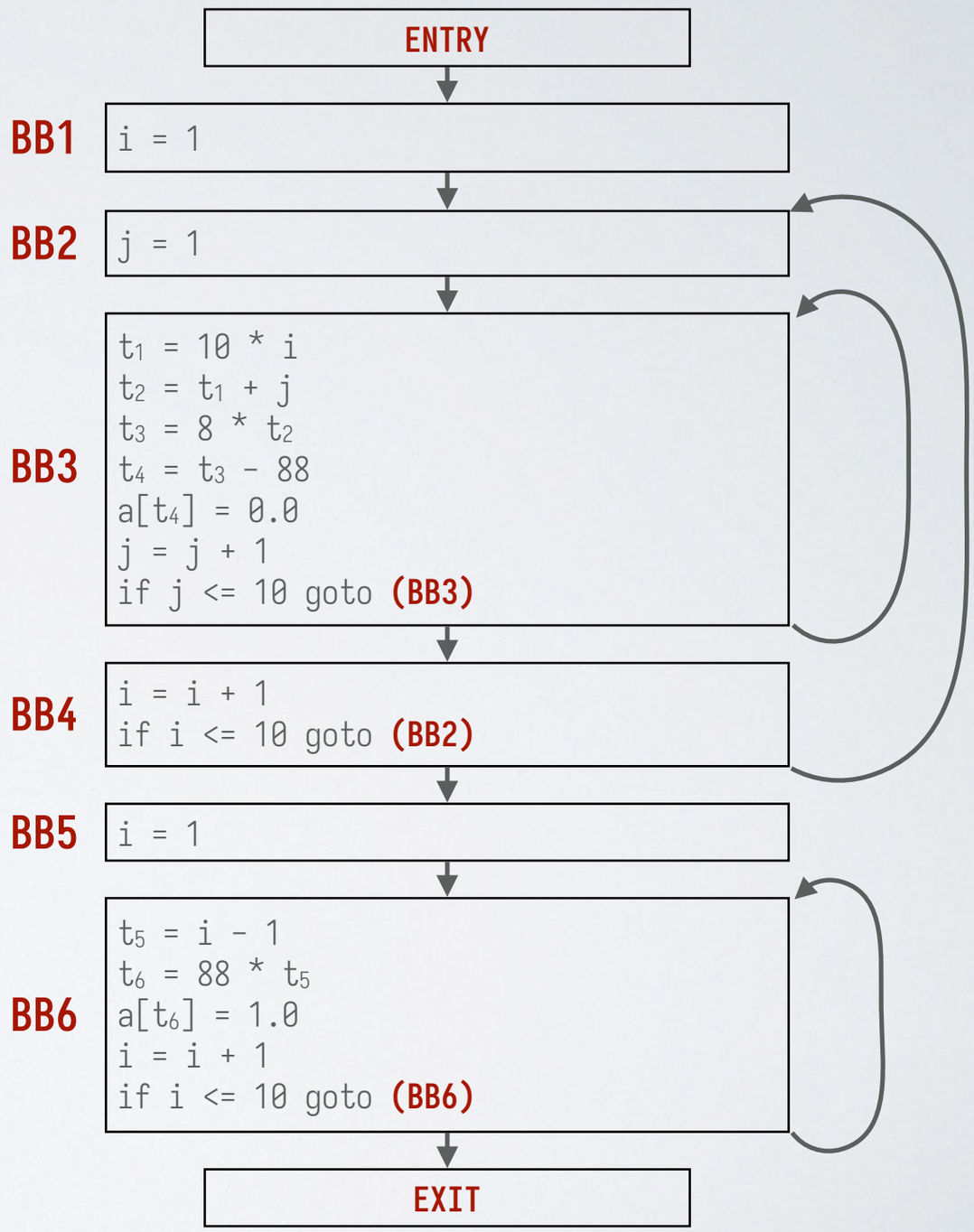
基本块

构造控制流图

- 前面提到的「控制流图 + 三地址代码」中不完全是三地址形式
  - 比如 `if j <= 10 goto (BB3) else goto (BB4)`
- 基于三地址跳转指令的流图：
  - 两个基本块 B 和 C 之间存在一条有向边**当且仅当**基本块 C 的第一条指令可能在 B 的最后一条指令之后执行
  - **情况 1：** B 的结尾跳转到 C 的开头
  - **情况 2：** B 的结尾不是无条件跳转，且 C 在原来的序列中紧跟 B 之后
- 可以额外添加**入口 (entry)** 和**出口 (exit)** 结点
  - 不包含指令

构造控制流图示例


BB1
(1) i = 1
BB2
(2) j = 1
(3) t1 = 10 * i
(4) t2 = t1 + j
(5) t3 = 8 * t2
BB3
(6) t4 = t3 - 88
(7) a[t4] = 0.0
(8) j = j + 1
(9) if j <= 10 goto (3)
(10) i = i + 1
BB4
(11) if i <= 10 goto (2)
BB5
(12) i = 1
(13) t5 = i - 1
(14) t6 = 88 * t5
BB6
(15) a[t6] = 1.0
(16) i = i + 1
(17) if i <= 10 goto (13)


ENTRY
BB1
i = 1
BB2
j = 1
BB3
t1 = 10 * i
t2 = t1 + j
t3 = 8 * t2
t4 = t3 - 88
a[t4] = 0.0
j = j + 1
if j <= 10 goto (BB3)
BB4
i = i + 1
if i <= 10 goto (BB2)
BB5
i = 1
BB6
t5 = i - 1
t6 = 88 * t5
a[t6] = 1.0
i = i + 1
if i <= 10 goto (BB6)
EXIT

中间表示的命名策略

- 编译器生成 IR 时，常常会生成很多的临时变量
- 如何为这些临时变量命名可能影响后续的分析和优化

源程序

```
a = b + c;
b = a - d;
c = b + c;
d = a - d;
```

按源程序变量命名

```
t0 = b
t1 = c
t2 = t0 + t1
a  = t2
t3 = d
t0 = t2 - t3
b  = t0
t1 = t0 + t1
c  = t1
t3 = t2 - t3
d  = t3
```

字面意义上相同的表达式计算的值一定相同

按值命名

```
t0 = b
t1 = c
t2 = t0 + t1
a  = t2
t3 = d
t4 = t2 - t3
b  = t4
t5 = t4 + t1
c  = t5
t4 = t2 - t3
d  = t4
```

b 和 d 最终的值相同

静态单赋值形式（SSA）

- **Static Single Assignment**，简称 SSA
- 现代编译器常常设计 SSA 形式的中间表示
  - LLVM IR、Koopa IR
- SSA 中的所有赋值都针对**不同名字**的变量
  - 对一个名字的使用可以找到唯一的定值(definition)

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

三地址代码

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

SSA 形式

SSA 形式中的控制流

- 同一个变量可能在不同的路径中被定值
  - `if ( flag ) x = -1; else x = 1;  y = x * a;`
- 使用 φ **函数**来合并不同路径中的定值
  - `if ( flag ) x1 = -1; else x2 = 1; x3 = φ(x1,x2); y = x3 * a;`
  - LLVM IR 采用这种设计
- 使用**带参数的跳转指令**来传递不同路径中的定值

```
if ( flag ) { x1 = -1; goto L(x1); }
else { x2 = 1; goto L(x2); }
L(z): y = z * a;
```

  - Koopa IR 采用这种设计

SSA 形式中的循环

- 虽然名字的定值只有一处，但是其值的计算可能不止一次
- 例：循环

源程序

```
x = ...;
y = ...;
while (x < 100) {
  x = x + 1;
  y = y + x;
}
```

SSA 形式

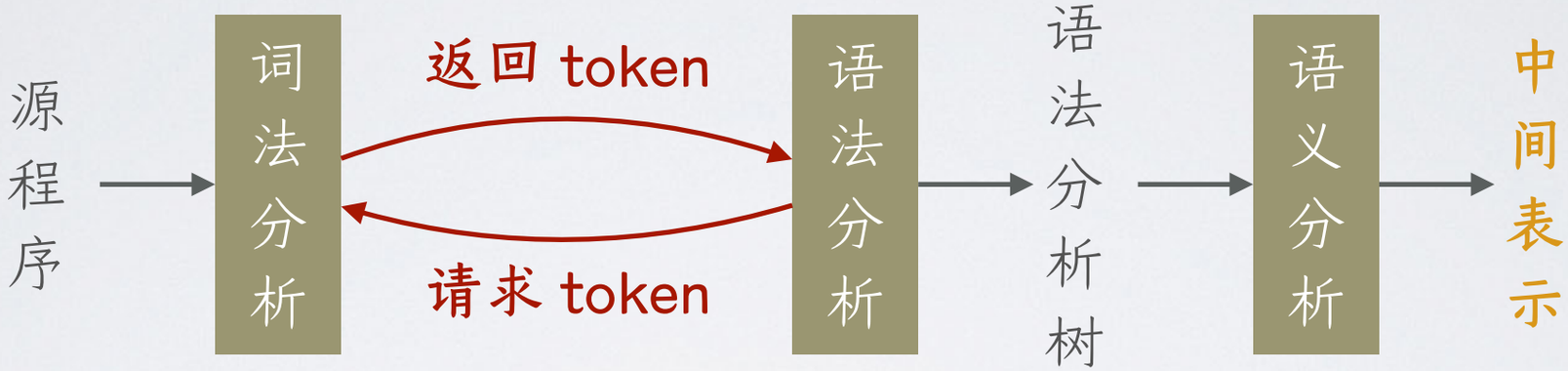
```
      x0 = ...
      y0 = ...
      if x0 >= 100 goto next
loop: x1 = φ(x0,x2)
      y1 = φ(y0,y2)
      x2 = x1 + 1
      y2 = y1 + x2
      if x2 < 100 goto loop
next: x3 = φ(x0,x2)
      y3 = φ(y0,y2)
```

主要内容

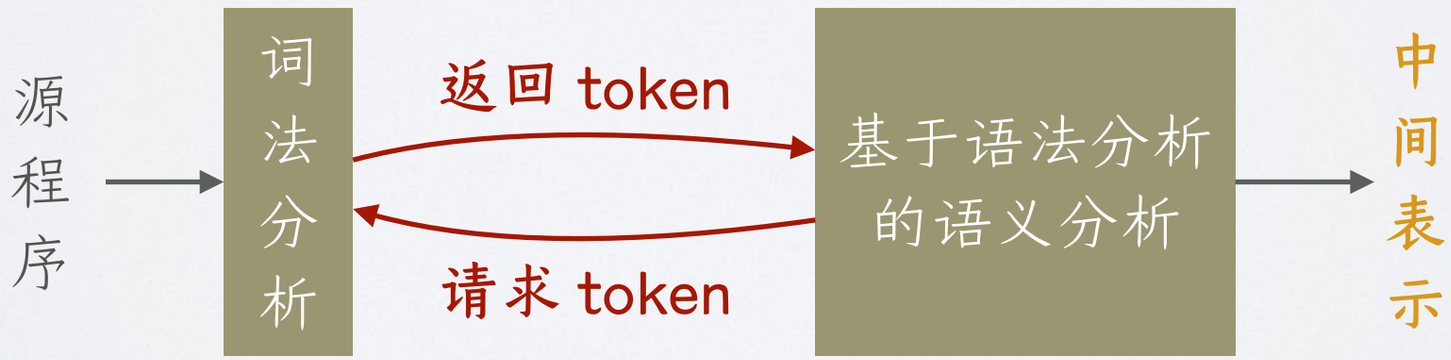
- 中间表示的作用
- 中间表示的设计
- **中间表示的生成**

回顾：基于语法分析实现语义分析

- 如果分开实现语法和语义分析，后者可以通过遍历语法分析树来实现(通常可以使用深度优先遍历)



- 也可以实现为同步进行的语法、语义分析



中间表示的生成策略

- 通过基于语法分析的语义分析进行
  - **优点**：不用显式构造语法分析树，伴随语法分析进行，效率高
  - **缺点**：需要精妙地设计属性文法(即 SDD) 或翻译方案(即 SDT)

- 先构造抽象语法树，再通过遍历抽象语法树进行
  - **优点**：脱离语法分析，通过遍历语法树进行翻译，耦合度低
  - **缺点**：需要显式构造抽象语法树，效率可能不够高

- 本部分侧重通过基于语法分析的语义分析生成三地址代码

回顾：属性文法/语法制导定义

- **属性文法 = 上下文无关文法 + 属性计算规则**
- 综合属性、继承属性
- S 属性的文法、L 属性的文法

产生规则	属性计算规则
$Expr \rightarrow Expr_1 + Term$	$Expr.\mathtt{val} = Expr_1.\mathtt{val} + Term.\mathtt{val}$
$Expr \rightarrow Expr_1 - Term$	$Expr.\mathtt{val} = Expr_1.\mathtt{val} - Term.\mathtt{val}$
$Expr \rightarrow Term$	$Expr.\mathtt{val} = Term.\mathtt{val}$
$Term \rightarrow Term_1 * Factor$	$Term.\mathtt{val} = Term_1.\mathtt{val} \times Factor.\mathtt{val}$
$Term \rightarrow Term_1 / Factor$	$Term.\mathtt{val} = Term_1.\mathtt{val} \div Factor.\mathtt{val}$
$Term \rightarrow Factor$	$Term.\mathtt{val} = Factor.\mathtt{val}$
$Factor \rightarrow (Expr)$	$Factor.\mathtt{val} = Expr.\mathtt{val}$
$Factor \rightarrow \mathtt{INT}$	$Factor.\mathtt{val} = \mathtt{INT}.\mathtt{intval}$

Expr、*Term*、*Factor* 的属性 val 都是综合属性

回顾：属性文法/语法制导定义

- **属性文法 = 上下文无关文法 + 属性计算规则**
- 综合属性、继承属性
- S 属性的文法、L 属性的文法

继承属性 inh 记录左侧已经计算的值

产生规则	属性计算规则
$Expr \rightarrow Term\ Expr'$	$Expr'.\mathtt{inh} = Term.\mathtt{val}$ $Expr.\mathtt{val} = Expr'.\mathtt{syn}$
$Expr' \rightarrow +\ Term\ Expr'_1$	$Expr'_1.\mathtt{inh} = Expr'.\mathtt{inh} + Term.\mathtt{val}$ $Expr'.\mathtt{syn} = Expr'_1.\mathtt{syn}$
$Expr' \rightarrow \epsilon$	$Expr'.\mathtt{syn} = Expr'.\mathtt{inh}$
……	……
$Factor \rightarrow (\ Expr\ )$	$Factor.\mathtt{val} = Expr.\mathtt{val}$
$Factor \rightarrow \mathtt{INT}$	$Factor.\mathtt{val} = \mathtt{INT}.\mathtt{intval}$

综合属性 syn 记录在 inh 基础上计算的值

回顾：语法制导的翻译方案

- 把属性计算规则改写为计算属性值的**程序片段**，并用花括号{ }括起来，插入到产生规则**右侧**的任何合适的位置上
- 理解为深度优先遍历语法分析树时，何时执行程序片段

产生规则	语义动作
$Expr \rightarrow Expr_1 + Term$	{ print('+'); }
$Expr \rightarrow Term$	
$Term \rightarrow Term_1 * Factor$	{ print('*'); }
$Term \rightarrow Factor$	
$Factor \rightarrow (Expr)$	
$Factor \rightarrow$ INT	{ print(INT.lexeme); }

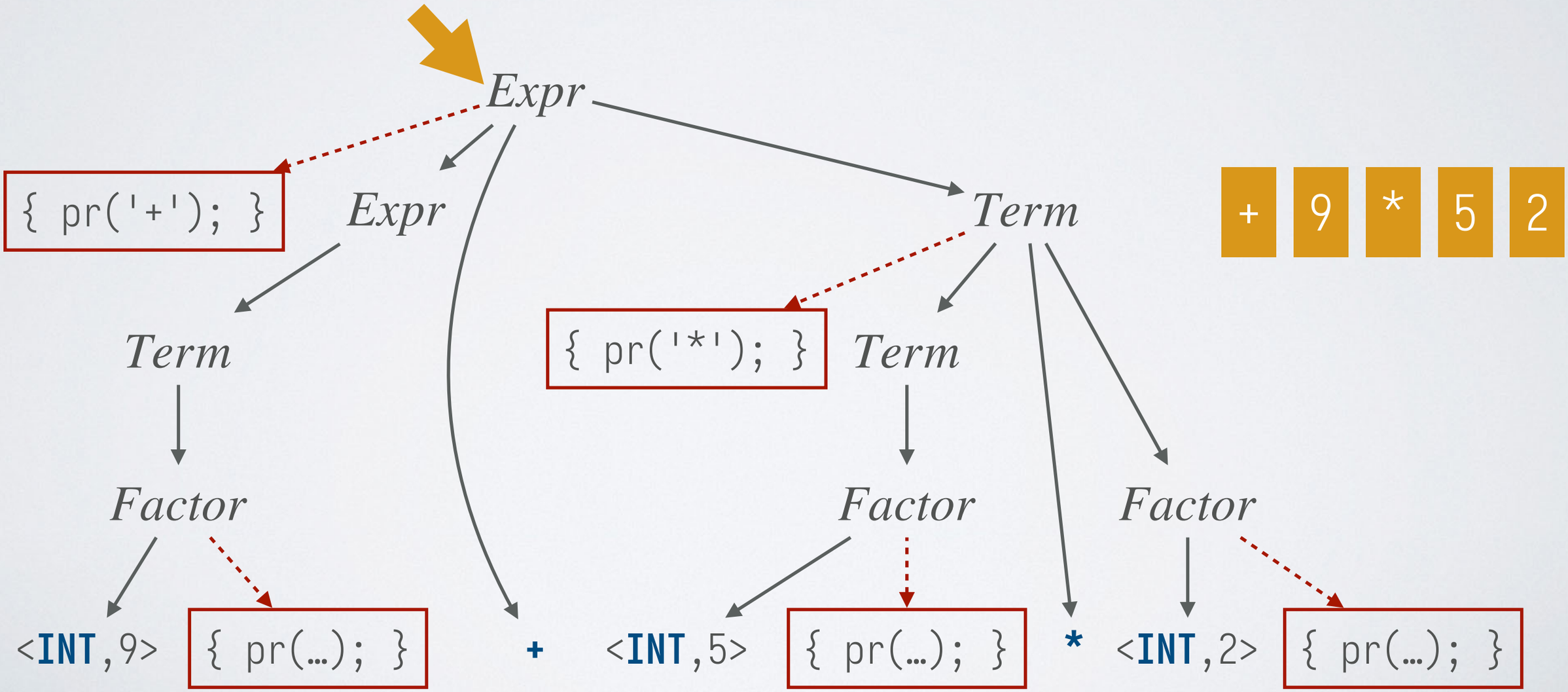
也可以写在一行上：

$Expr \rightarrow$ { print('+'); } $Expr_1 + Term$

回顾：语法制导的翻译方案

- 把属性计算规则改写为计算属性值的**程序片段**，并用花括号 { } 括起来，插入到产生规则**右侧**的任何合适的位置上

- 理解为深度优先遍历语法分析树时，何时执行程序片段



表达式和赋值语句

- 为了简化讨论，我们使用等价的有二义性的文法来说明

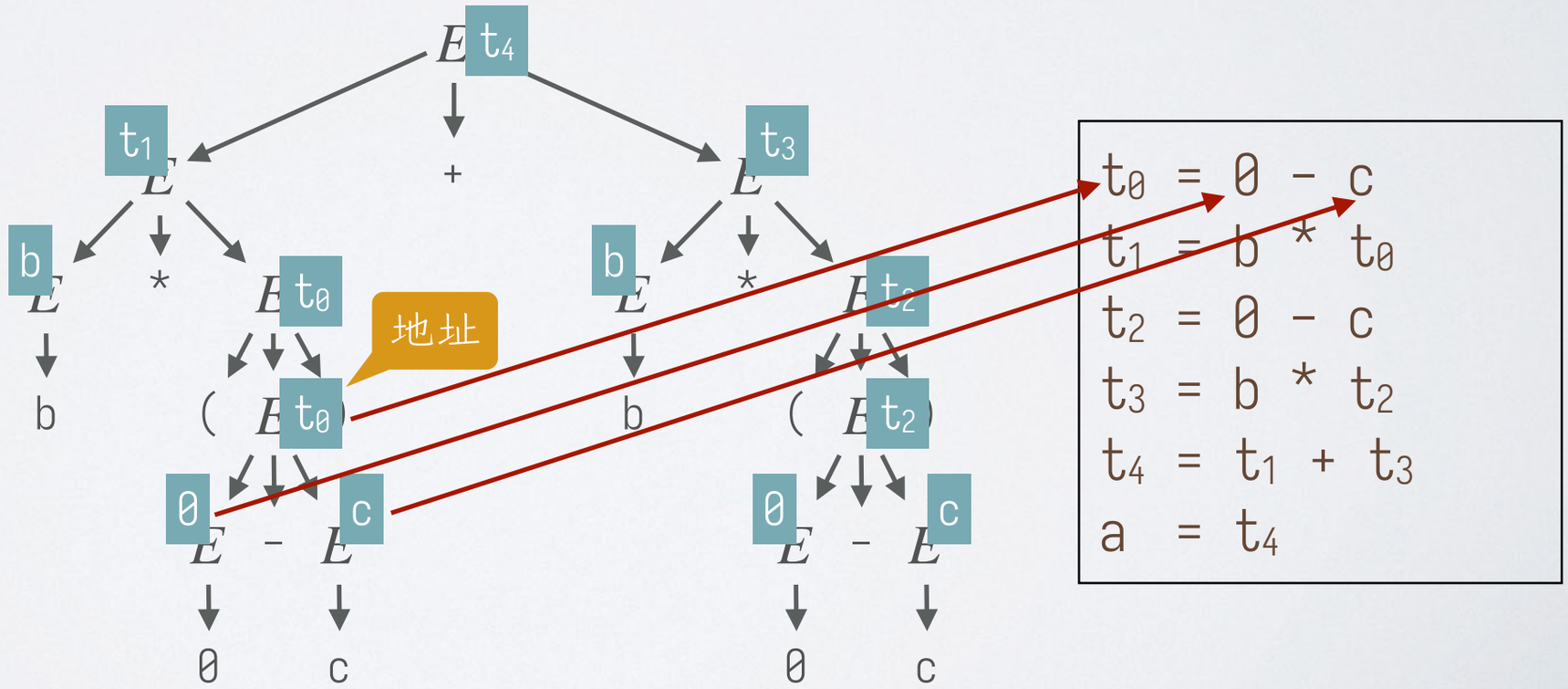
$$
\begin{aligned}
E &::= E + T \mid E - T \mid T \\
T &::= T * F \mid T / F \mid F \\
F &::= (E) \mid \texttt{ID} \mid \texttt{INT} \\
S &::= \texttt{ID} = E ;
\end{aligned}
$$

$$
\Rightarrow
$$

$$
\begin{aligned}
E &::= E + E \mid E - E \\
&\mid E * E \mid E / E \\
&\mid (E) \mid \texttt{ID} \mid \texttt{INT} \\
S &::= \texttt{ID} = E ;
\end{aligned}
$$

- 思路：如何通过遍历语法分析树来生成三地址代码？

```
a = b * (0 - c) +
    b * (0 - c);
```



翻译表达式和赋值语句的 SDT

- 用 `...` 表示三地址代码，用 {...} 表示进行插值
  - 类似 Python 中的 f"Hello {name}"，字符串会代入变量 name 的值
- 属性 addr 表示表达式的值所在的地址，属性 code 表示翻译表达式或语句得到的三地址代码
- 用 || 把多个三地址代码块连接起来

	语义动作
$E \to \text{ID}$	{ E.addr = genv...; }
$E \to \text{INT}$	{ E.addr = gencst(INT.intval); E.code = ``; }
$E \to E_1 + E_2$	{ E.addr = gentmp(); E.code = E_1.code || E_2.code || `{$E$.addr} = {$E_1$.addr} + {$E_2$.addr}`; }
$E \to (E_1)$	{ E.addr = E_1.addr; E.code = E_1.code; }
$S \to \text{ID} = E;$	{ S.code = E.code || `{genvar(ID.lexeme)} = {$E$.addr}`; }

genvar 函数为程序变量生成地址

gencst 函数为常量生成地址

gentmp 函数生成一个新的临时变量的地址

增量式翻译

- 属性 code 的计算有一定的冗余，特别是 || 连接操作
- on-the-fly 的生成策略：通过 SDT 的副作用生成三地址代码
  - 回顾：$S \to X\ Y\ \{\ a\ \}$ 意味着在处理完 X 和 Y 的动作之后再执行 a
- 用 emit 函数表示直接输出生成的三地址指令

产生规则	语义动作
$E \to$ ID	{ E.addr = genvar(ID.lexeme); }
$E \to$ INT	{ E.addr = gencst(INT.intval); }
$E \to E_1 + E_2$	{ E.addr = gentmp(); emit(`{$E$.addr} = {$E_1$.addr} + {$E_2$.addr}`); }
$E \to (\ E_1\ )$	{ E.addr = E_1.addr; }
$S \to$ ID = E ;	{ emit(`{genvar(ID.lexeme)} = {$E$.addr}`); }

声明语句和语句块

- 许多语言要求变量要先**声明**再使用
- 许多语言支持**语句块**，且语句块限制了局部变量的作用域

```
{ int n;
  int a;
  int b;
  n = 10; a = 1; b = 1;
  while (!(n == 0)) {
    int t;
    t = a + b; a = b; b = t;
    n = n - 1;
  }
  return a;
}
```

```
{ int n;
  n 的作用域
  int a;
  int b;
  n = 10; a = 1; b = 1;
  a 的作用域
  while (!(n == 0)) {
    { int n;
    b 的作用域
      n = a + b; a = b; b = n;
    }
    t 的作用域
    n = n - 1;
  }
  return a;
}
```

- **问题：如何正确处理作用域？比如重复的变量名？**

翻译中使用的符号表

- 编译器通过**符号表(symbol table)**来汇总不同名字的信息
  - 在本部分我们重点考虑程序变量
  - 前面的 genvar 函数需要生成变量在三地址代码中对应的名字

```
{ int n;
  int a;
  int b;
  ...
}
```

```
{ int n;
  int a;
  int b;
  { int n; ...
  }
}
```

?

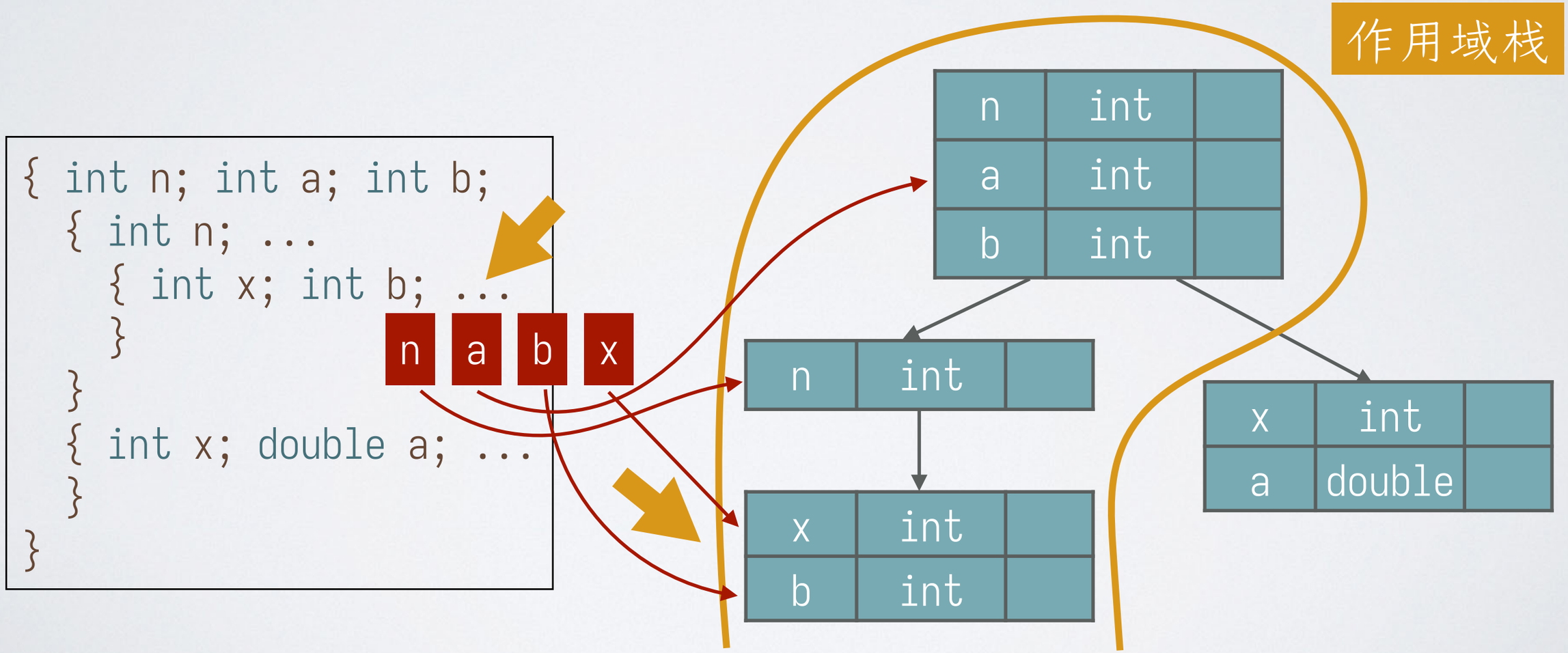
?

名字	类型	在 IR 中的名字	声明位置	其它信息
n	int	n	…	
a	int	a	…	
b	int	b	…	

名字	类型	在 IR 中的名字	声明位置	其它信息
n	int	n	…	
a	int	a	…	
b	int	b	…	
n	int	n2	…	

通过符号表管理作用域

- **每个作用域对应一个符号表**：多个符号表形成树状结构
- 在翻译时，可以通过栈来存放当前符号表及其祖先
  - 对程序变量 x 的使用需查看**最靠近栈顶**的声明了 x 的作用域



符号表提供的接口

- push_scope(): 在栈顶压入一个新的作用域
- pop_scope(): 弹出栈顶的作用域
- insert(name, info): 在栈顶作用域中记录名字和它的信息
- lookup(name): 查找最靠近栈顶的名字对应的信息
  - 可用于实现 genvar 函数

```
{ int n; int a; int b;
  { int n; ...
    { int x; int b;
      x = n + a + b;
      ...
    }
  }
  { int x; double a; ...
  }
}
```

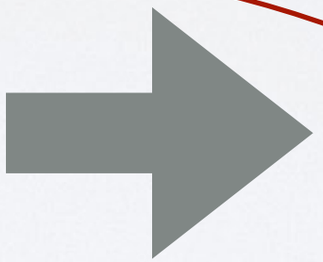
```
1. push_scope()
2. insert(n)
3. insert(a)
4. insert(b)
5. push_scope()
6. insert(n)
7. push_scope()
8. insert(x)
9. insert(b)
10.lookup(n)
```

```
11.lookup(a)
12.lookup(b)
13.lookup(x)
14.pop_scope()
15.pop_scope()
16.push_scope()
17.insert(x)
18.insert(a)
19.pop_scope()
20.pop_scope()
```

声明语句和语句块的翻译

$$
\begin{aligned}
S &::= \cdots \mid T\ \texttt{ID}\ \texttt{;} \mid \texttt{\{}\ L\ \texttt{\}} \\
T &::= \texttt{int} \mid \texttt{double} \\
L &::= \epsilon \mid L\ S
\end{aligned}
$$

```
{ int n; int a; int b;
  n = 10; a = 1; b = 1;
  { int n;
    n = a + b;
    a = b;
    b = n;
  }
  n = n - 1;
}
```



```
n  = 10
a  = 1
b  = 1
t0 = a + b
n2 = t0
a  = b
b  = n2
t1 = n - 1
n  = t1
```

翻译声明语句和语句块的 SDT

产生规则	语义动作
$S \to T$ ID ;	{ insert(ID.lexeme, T.type); S.code = `` ; }
$S \to \{$	{ push_scope(); }
L	{ pop_scope(); }
$\}$	{ S.code = L.code; }
$L \to \epsilon$	{ L.code = ``; }
$L \to L_1 S$	{ L.code = L_1.code || S.code; }
$T \to$ int	{ T.type = int; }
$T \to$ double	{ T.type = $double$; }

push_scope()

push_scope()

pop_scope()

pop_scope()

```
{ int n; int a; int b;
  n = 10; a = 1; b = 1;
  { int n;
    n = a + b;
    a = b;
    b = n;
  }
  n = n - 1;
}
```

支持更多的类型：结构体

- 结构体类型中有若干**字段的声明**
  - 其形式和处理方法跟声明语句差不多
- 设 pop_scope 函数会返回弹出的栈顶作用域，*struct* 函数通过作用域构造结构体类型的表达式

$$T ::= \cdots \mid \texttt{struct}\{D\}$$
$$D ::= \epsilon \mid T\ \texttt{ID}\ ;\ D$$

产生规则	语义动作
$T \to \texttt{struct}\{$ D $\}$	{ push_scope(); } { T.type = *struct*(pop_scope()); }
$D \to \epsilon$	
$D \to T\ \texttt{ID}\ ;$ D_1	{ insert(ID.lexeme, T.type); }

支持更多的类型：数组

- 计算数组类型的表达式和数组的**宽度(width)**
  - 相关信息可以记录在符号表中
  - 便于后续生成目标代码时进行存储空间的分配

基本类型

数组分量

$$T ::= T_B\, C$$

$$T_B ::= \texttt{int} \mid \texttt{double}$$

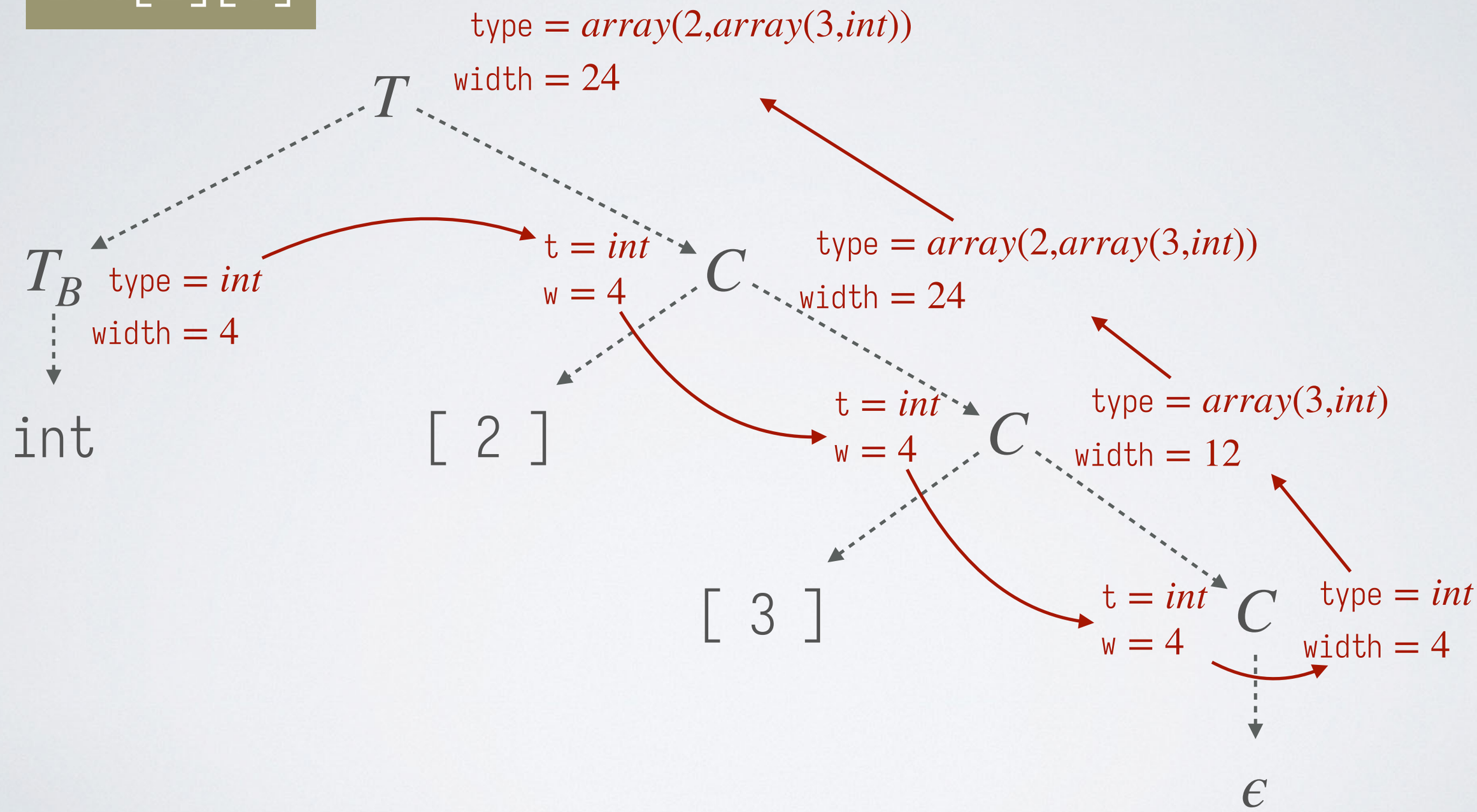
$$C ::= \epsilon \mid \texttt{[INT]}\, C$$

用继承属性 t 和 w 记录数组元素的类型和宽度

产生规则	
$T \rightarrow T_B\ C$	{ C.t = T_B.type; C.w = T_B.width; }
	{ T.type = C.type; T.width = C.width; }
$C \rightarrow \epsilon$	{ C.type = C.t; C.width = C.w; }
$C \rightarrow$ [INT] C_1	{ C_1.t = C.t; C_1.w = C.w; }
	{ C.type = *array*(INT.intval, C_1.type);
	C.width = INT.intval * C_1.width; }
$T_B \rightarrow$ int	{ T_B.type = *int*; T_B.width = 4; }
$T_B \rightarrow$ double	{ T_B.type = *double*; T_B.width = 8; }

数组类型处理示例

int[2][3]



数组引用

- 处理数组引用的文法：
  - 数组引用：$R ::= R[E] \mid \texttt{ID}[E]$
  - 数组读取：$E ::= \cdots \mid R$
  - 数组赋值：$S ::= \cdots \mid R = E;$

- **问题：如何根据数组引用计算相对于数组基址的偏移？**

- 计算方法取决于数组的存储方式

数组寻址

- 假设数组元素被存放在连续的存储空间中
  - 一个数组长度为 n，元素从 0 到 $n-1$ 编号
  - 每个数组元素的宽度是 w，第 i 个元素的地址为 $base + i \times w$
  - $base$ 是分配给数组 A 的内存块的相对地址
- 推广到二维数组
  - 假设一行的宽度是 w_1，同一行中每个元素的宽度是 w_2
  - $A[i_1][i_2]$ 的相对地址为 $base + i_1 \times w_1 + i_2 \times w_2$

数组寻址

- **k 维数组**的寻址：假设数组按行存放，即首先存放 $A[0][i_2]\ldots[i_k]$，然后存放 $A[1][i_2]\ldots[i_k]$，…
  - 设 n_j 为第 j 维上数组元素的个数
  - w_j 为第 j 维的每个子数组元素的宽度
  - w_k 为每个数组元素的宽度：$w_k = w$
  - $w_{j-1} = n_j \times w_j$
- $A[i_1][i_2]\ldots[i_k]$ 的地址
  - $base + i_1 \times w_1 + i_2 \times w_2 + \ldots + i_k \times w_k$
  - 或者
  - $base + ((\ldots((i_1 \times n_2 + i_2) \times n_3 + i_3)\ldots) \times n_k + i_k) \times w$

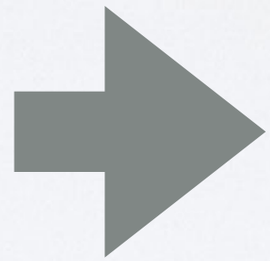
数组寻址

- 多维数组的存放方法
  - 前面描述的地址计算是基于**按行存放**的
  - 有的语言采取**按列存放**，例如 Fortran
- 有的语言中，数组元素的下标不一定从 0 开始
  - 比如 Pascal 语言中声明数组时可以指定下标区间
  - `VAR a: ARRAY [low..high] OF real;`
  - 求 $a[i]$ 的地址：$base + (i - low) \times w$
  - 可以在编译时刻预先计算 $base - low \times w$

数组引用的翻译

$$
\begin{aligned}
R &::= R[E] \mid \texttt{ID}[E] \\
E &::= \cdots \mid R \\
S &::= \cdots \mid R = E;
\end{aligned}
$$

```
int[2][3] a;
int[5] b;
...
a[i * j + k][l] = b[m - n];
```



```
t0 = i * j
t1 = t0 + k
t2 = t1 * 3
t3 = t2 + l
t4 = t3 * 4
t5 = m - n
t6 = t5 * 4
t7 = b [ t6 ]
a [ t4 ] = t7
```

翻译数组引用的 SDT

- 给 R 三个额外的综合属性：
  - R.addr 为一个保存数组偏移的地址
  - R.base 为数组的基地址
  - R.type 记录 R 对应的子数组的类型

$$R ::= R[E] \mid \texttt{ID}[E]$$
$$E ::= \cdots \mid R$$
$$S ::= \cdots \mid R = E;$$

例：*array*(2,*array*(3,*int*))

例：*array*(3,*int*)

产生规则	
$R \to \texttt{ID}[E]$	{ R.addr = E.addr; R.base = genvar(ID.lexeme); R.type = lookup(ID.lexeme).type.elem; R.code = E.code;
$R \to R_1[E]$	{ R.addr = gentmp(); R.base = R_1.base; R.type = R_1.type.elem; t = gentmp(); R.code = R_1.code ‖ E.code ‖ `{t} = {$R_1$.addr} * {$R_1$.type.len}` ‖ `{$R$.addr} = {t} + {$E$.addr}`; }
$E \to R$	{ E.addr = gentmp(); t = gentmp(); E.code = R.code ‖ `{t} = {$R$.addr} * {$R$.type.width}` ‖ `{$E$.addr} = {$R$.base} [ {t} ]`; }

例：*int*

例：3

例：4

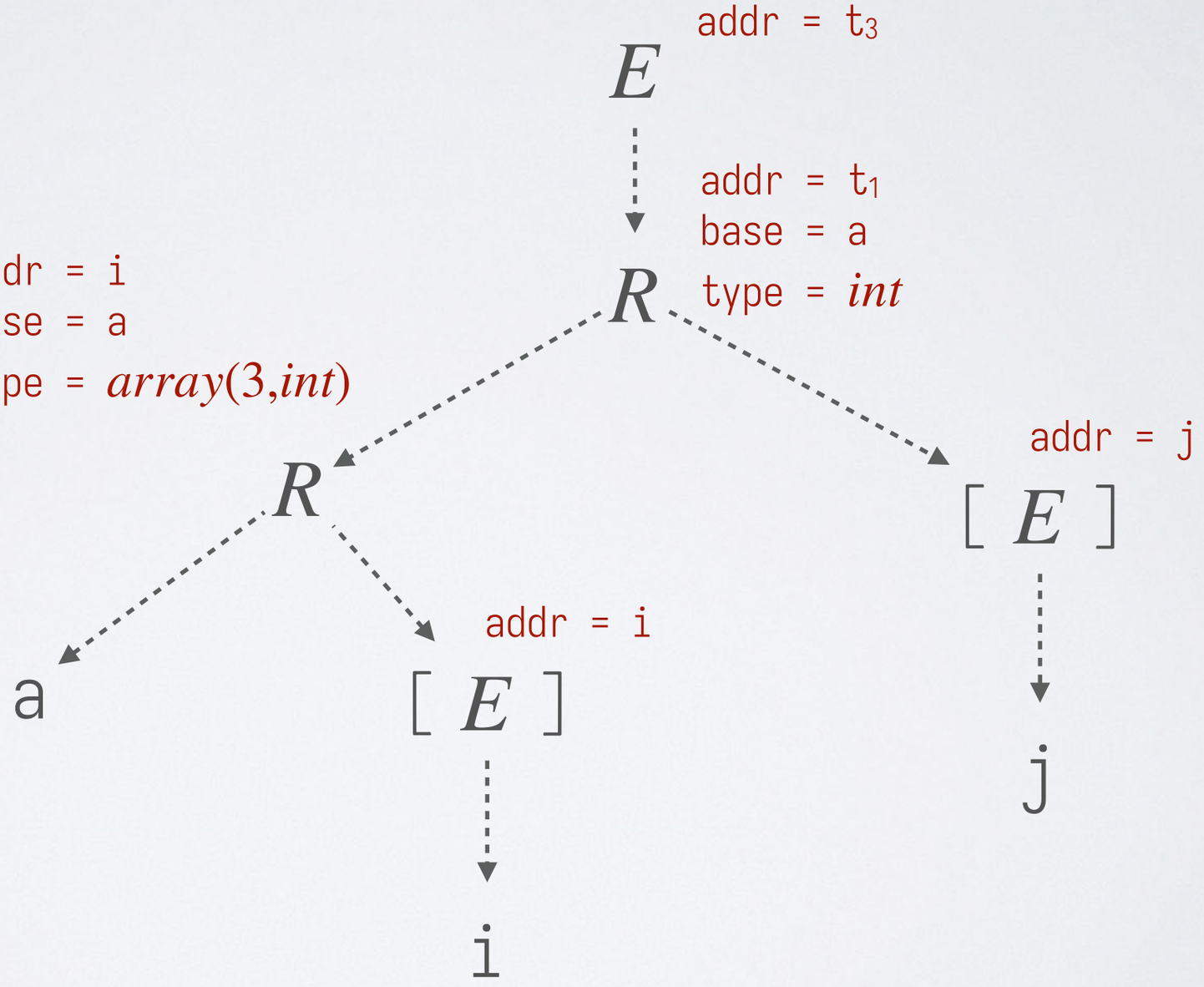
数组读取翻译示例

a 是 int[2][3] 的数组

- a 的类型是 $array(2,array(3,int))$
- a[i] 的类型是 $array(3,int)$

a[i][j]

```
t0 = i * 3
t1 = t0 + j
t2 = t1 * 4
t3 = a [ t2 ]
```



翻译数组引用的 SDT

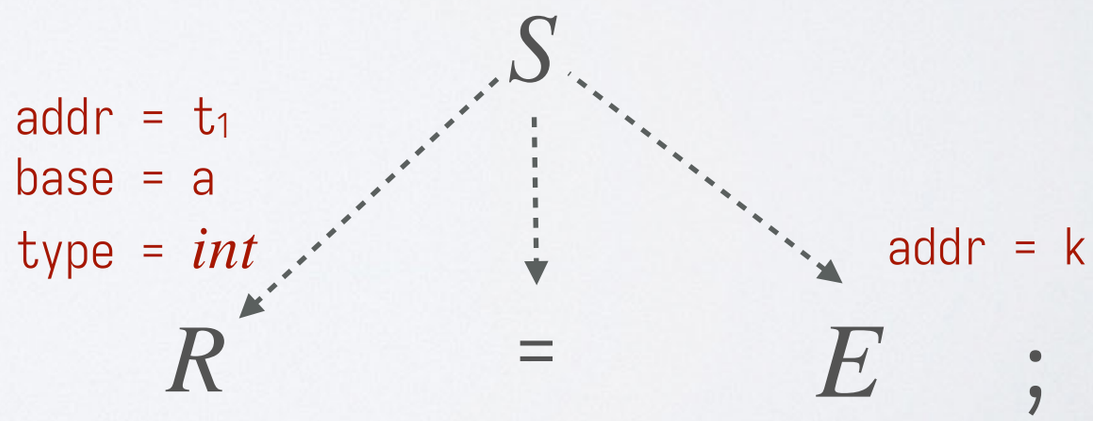
- 给 R 三个额外的综合属性：
  - R.addr 为一个保存数组偏移的地址
  - R.base 为数组的基地址
  - R.type 记录 R 对应的子数组的类型

$$R ::= R[E] \mid \texttt{ID}[E]$$
$$E ::= \cdots \mid R$$
$$S ::= \cdots \mid R = E;$$

产生规则	语义动作
$S \to R = E;$	{ t = gentmp(); S.code = R.code ‖ `{t} = {$R$.addr} * {$R$.type.width}` ‖ E.code ‖ `{$R$.base} [ {t} ] = {$E$.addr}`; }

a[i][j] = k;

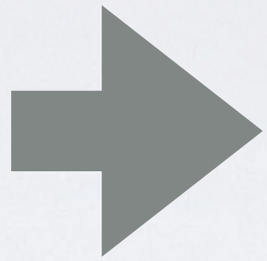
```
t0 = i * 3
t1 = t0 + j
t2 = t1 * 4
a [ t2 ] = k
```



布尔表达式

- 为了简化讨论，我们使用等价的有二义性的文法来说明

$$
\begin{aligned}
B &::= B \,||\, BC \mid BC \\
BC &::= BC \,\&\&\, BR \mid BR \\
BR &::= E == E \mid E <= E \mid BA \\
BA &::= (\, B \,) \mid !\, BA \mid \texttt{true} \mid \texttt{false}
\end{aligned}
$$



$$
\begin{aligned}
B &::= B \,||\, B \\
&\mid\ B \,\&\&\, B \\
&\mid\ E == E \mid E <= E \\
&\mid\ (\, B \,) \mid !\, B \mid \texttt{true} \mid \texttt{false}
\end{aligned}
$$

- 如果只是对布尔表达式求值，其翻译过程与算术表达式相似
- 但是，布尔表达式通常需要被**短路求值**
  - $B_1 \,||\, B_2$ 中 B_1 为真时，不用计算 B_2，整个表达式为真
    - 当 B_1 为真时应该跳过计算 B_2 的代码
  - $B_1 \,\&\&\, B_2$ 中 B_1 为假时，不用计算 B_2，整个表达式为假
    - 当 B_1 为假时应该跳过计算 B_2 的代码

短路求值示例

- 源程序：
  - `if (x < 100 || (x > 200 && x != y)) x = 0;`
- 三地址代码：

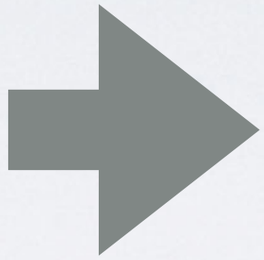
```
        if          x < 100       goto L1
        ifFalse     x > 200       goto L0
        ifFalse     x != y        goto L0
L1:     x = 0
L0:     接下来的代码
```

- **回顾：三地址代码显式给出子表达式的计算顺序**

条件语句的翻译

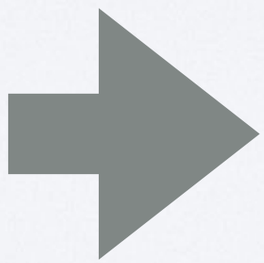
$$S ::= \cdots \mid \texttt{if}\,(B)\,S \mid \texttt{if}\,(B)\,S\,\texttt{else}\,S$$

```
if (n <= 0) {
  n = 0 - n;
}
```



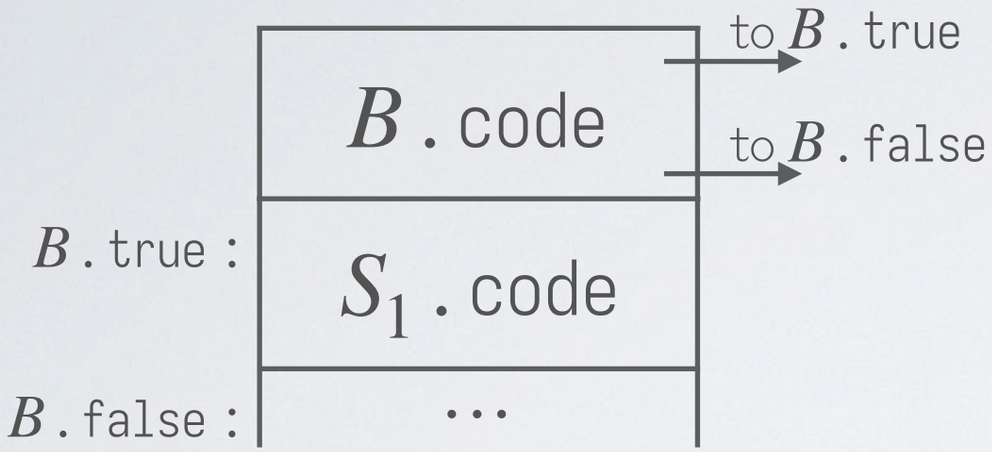
```
    if n <= 0 goto L1
    goto L0
L1: t0 = 0 - n
    n  = t0
L0:
```

```
if (n <= 0) {
  n = 0 - n;
} else {
  n = n + 1;
}
```

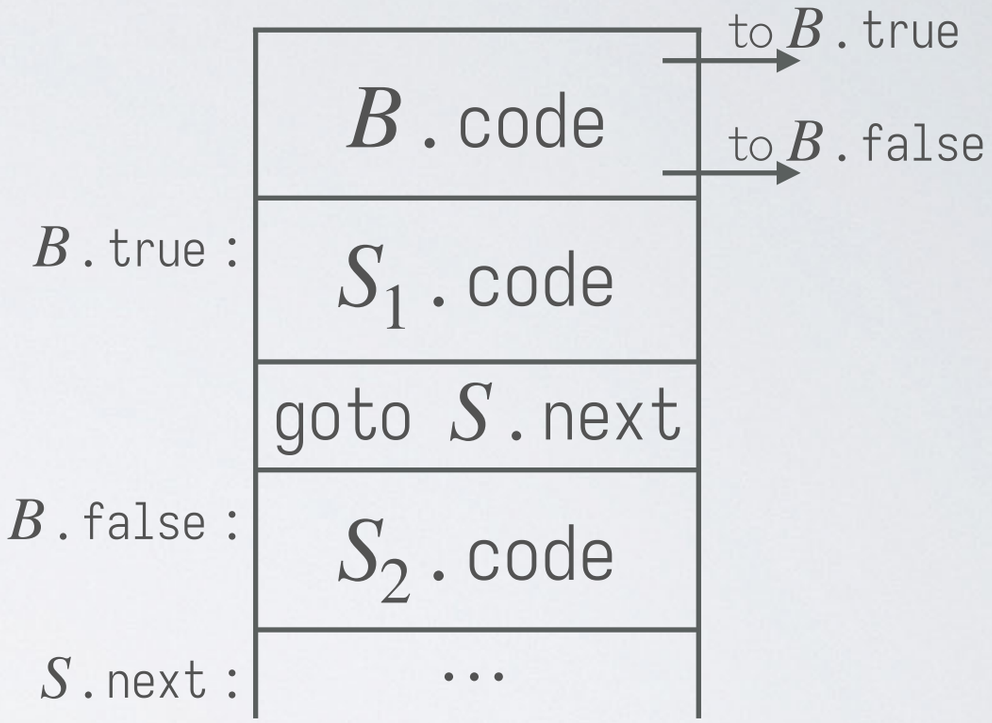


```
    if n <= 0 goto L1
    goto L2
L1: t0 = 0 - n
    n  = t0
    goto L0
L2: t1 = n + 1
    n  = t1
L0:
```

条件语句的翻译思路



$$S \rightarrow \texttt{if}\,(\,B\,)\,S_1$$



$$S \rightarrow \texttt{if}\,(\,B\,)\,S_1\,\texttt{else}\,S_2$$

继承属性：

- B.true：B 为真时的跳转标号
- B.false：B 为假时的跳转标号
- S.next：紧跟在 S 之后的指令标号

计算翻译条件语句所需的继承属性

genlabel 函数生成新的标号

在三地址代码中插入标号

产生规则	动作
$S' \rightarrow$	
S EOF	{ S.next = genlabel(); }
	{ S'.code = S.code || `{$S$.next}:`; }
$S \rightarrow$ {	{ push_scope(); L.next = S.next; }
L	{ pop_scope(); }
}	{ S.code = L.code; }
$L \rightarrow$	{ L_1.next = genlabel(); }
L_1	{ S.next = L.next; }
S	{ L.code = L_1.code || `{$L_1$.next}:` || S.code; }

继承属性：

- B.true：B 为真时的跳转标号
- B.false：B 为假时的跳转标号
- S/L.next：紧跟在 S/L 之后的指令标号

翻译条件语句的 SDT

产生规则	语义动作
$S \rightarrow$ if (B) S_1	{ B.true = genlabel(); B.false = S.next; } { S_1.next = S.next; } { S.code = B.code ‖ `{$B$.true}:` ‖ S_1.code; }
$S \rightarrow$ if (B) S_1 else S_2	{ B.true = genlabel(); B.false = genlabel(); } { S_1.next = S.next; } { S_2.next = S.next; } { S.code = B.code ‖ `{$B$.true}:` ‖ S_1.code ‖ `goto {$S$.next}` ‖ `{$B$.false}:` ‖ S_2.code; }

继承属性：

- B.true：B 为真时的跳转标号
- B.false：B 为假时的跳转标号
- S/L.next：紧跟在 S/L 之后的指令标号

布尔表达式的翻译

- 生成的计算 B 的代码执行时跳转到两个标号之一：
  - 表达式的值为真时，跳转到 B.true
  - 表达式的值为假时，跳转到 B.false

- B.true 和 B.false 是两个继承属性，根据 B 所在的上下文指向不同的位置标号
  - 如果 B 是条件语句的条件表达式，则分别指向 then 分支和 else 分支
    - 如果没有 else 分支，则 B.false 指向 if 语句的下一条指令
  - 如果 B 是 while 循环语句的条件表达式，则分别指向循环体的开头和循环出口处

翻译布尔表达式的 SDT

产生规则	语义动作
$B \to$ true	{ B.code = `goto {$B$.true}`; }
$B \to$ false	{ B.code = `goto {$B$.false}`; }
$B \to$ (B_1)	{ B_1.true = B.true; B_1.false = B.false; } { B.code = B_1.code; }
$B \to$! B_1	{ B_1.true = B.false; B_1.false = B.true; } { B.code = B_1.code; }
$B \to E_1 == E_2$	{ B.code = E_1.code || E_2.code || `if {$E_1$.addr} == {$E_2$.addr} goto {$B$.true}` || `goto {$B$.false}`; }

继承属性：

- B.true：B 为真时的跳转标号
- B.false：B 为假时的跳转标号

翻译布尔表达式的 SDT

产生规则	语义动作
$B \rightarrow$ B_1 && B_2	{ B_1.true = genlabel(); B_1.false = B.false; } { B_2.true = B.true; B_2.false = B.false; } { B.code = B_1.code || B_2.code; }
$B \rightarrow$ B_1 || B_2	{ B_1.true = B.true; B_1.false = genlabel(); } { B_2.true = B.true; B_2.false = B.false; } { B.code = B_1.code || `{$B_1$.false}:` || B_2.code; }

当 B_1 为假时，逻辑与的结果一定为假

当 B_1 为真时，逻辑或的结果一定为假

继承属性：

- B.true：B 为真时的跳转标号
- B.false：B 为假时的跳转标号

布尔表达式翻译示例

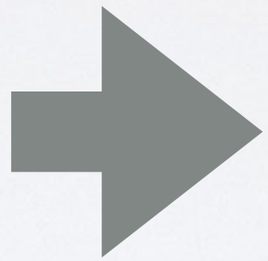
- 源程序：
  - `if (x < 100 || (x > 200 && x != y)) x = 0;`
- 三地址代码：

```
        if x < 100 goto L1
        goto L2
L2:     if x > 200 goto L3
        goto L0
L3:     if x != y  goto L1
        goto L0
L1:     x = 0
L0:
```

循环语句的翻译

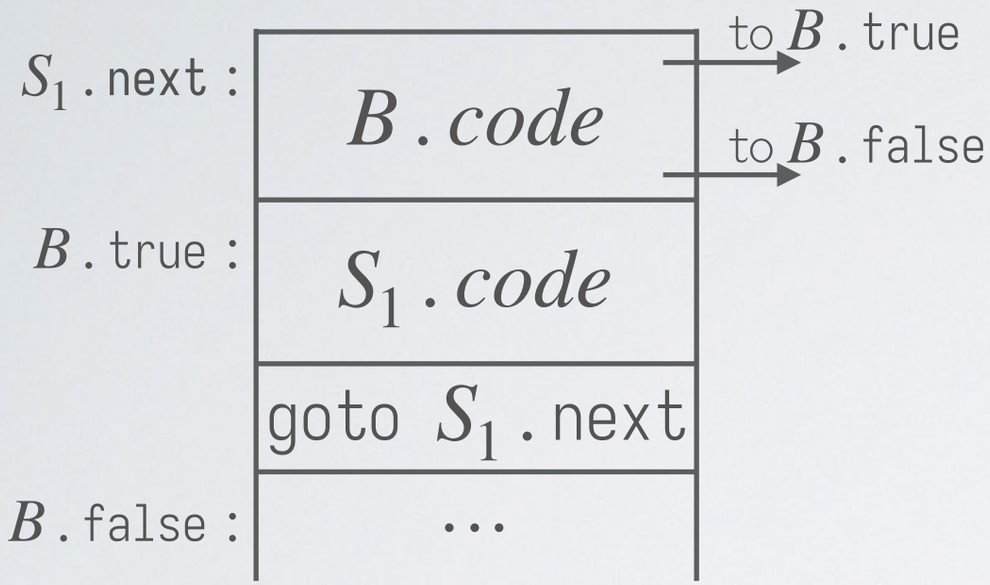
$$S ::= \cdots \mid \texttt{while}\,(\,B\,)\,S$$

```
while (!(n == 0)) {
  int t;
  t = a + b; a = b; b = t;
  n = n - 1;
}
```



```
L1: if n == 0 goto L0
    goto L2
L2: t0 = a + b
    t  = t0
    a  = b
    b  = t
    t1 = n - 1
    n  = t1
    goto L1
L0:
```

翻译循环语句的 SDT



$S \rightarrow \texttt{while(}\ B\ \texttt{)}\ S_1$

产生规则	语义动作
$S \rightarrow$ while(	{ B.true = genlabel(); B.false = S.next; }
B)	{ S_1.next = genlabel(); }
S_1	{ S.code = `{$S_1$.next}:` ‖ B.code ‖ `{$B$.true}:` ‖ S_1.code ‖ `goto {$S_1$.next}`; }

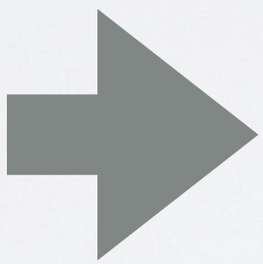
练习：如何支持 break 和 continue?

继承属性：

- B.true：B 为真时的跳转标号
- B.false：B 为假时的跳转标号
- S.next：紧跟在 S 之后的指令标号

多分支选择语句的翻译思路

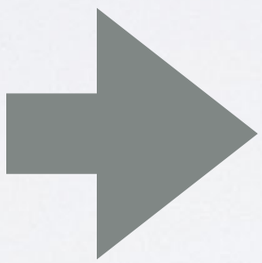
```
switch ( E ) {
  case V1: S1
  case V2: S2
           …
  case Vn−1: Sn−1
  default: Sn
}
```



```
        计算 E 并把结果赋给临时变量 t
        goto test
L1:     S1 的代码
        goto next
L2:     S2 的代码
        goto next
        ...
Ln-1:   Sn−1 的代码
        goto next
Ln:     Sn 的代码
        goto next
test:   if t = V1 goto L1
        if t = V2 goto L2
        ...
        if t = Vn−1 goto Ln-1
        goto Ln
next:
```

过程/函数调用的翻译方案

$x = f(E_1, E_2, \ldots, E_n);$



计算 E_1 并把结果赋给临时变量 t_1
param t_1
计算 E_2 并把结果赋给临时变量 t_2
param t_2
...
计算 E_n 并把结果赋给临时变量 t_n
param t_n
t = call f, n
x = t

过程/函数声明的翻译

```
int fib(int n) {
  int a; a = 1;
  int b; b = 1;
  while (!(n == 0)) {
    int t;
    t = a + b; a = b; b = t;
    n = n - 1;
  }
  return a;
}

int main() {
  return fib(10);
}
```

```
  a  = 1
  b  = 1
L1:
  if n == 0 goto L0
  goto L2
L2:
  t0 = a + b
  t  = t0
  a  = b
  b  = t
  t1 = n - 1
  n  = t1
  goto L1
L0:
  return a
```

为每个过程/函数生成中间代码

```
  param 10
  t0 = call fib,1
  return t0
```

翻译函数声明和调用的 SDT

函数声明

形参列表

语句序列

类型

$$F ::= T\ \texttt{ID}\ (\ Params\ )\ \{\ L\ \}$$

$$T ::= \cdots$$

$$Params ::= \epsilon \mid T\ \texttt{ID}\ ,\ Params$$

$$L ::= \epsilon \mid L\ S$$

$$S ::= \cdots \mid \texttt{return}\ E\ ;$$

$$E ::= \cdots \mid \texttt{ID}\ (\ Args\ )$$

$$Args ::= \epsilon \mid E\ ,\ Args$$

为方便演示，使用了产生额外逗号的文法

实参列表

这里也会产生多余逗号

如何确定跳转指令的目标？

- 为布尔表达式和控制流语句生成三地址代码的关键问题：**某些跳转指令应该跳转到哪里？**

- 例如：`if` (B) S
  - 按照短路求值的翻译，B 的代码中有一些跳转指令在 B 为假时执行
  - 这些跳转指令的目标应该跳过 S 对应的代码
  - 生成这些指令时，S 的代码尚未生成，因此目标不确定
  - 如果通过语句的继承属性 S`.next` 来传递，当中间表示不允许符号标号时，则需要第二趟处理

- **问题：如何一趟处理完毕呢？**

回填

- 基本思想：考虑 if (B) S
  - 记录 B 的代码中跳转指令 goto S.next、if … goto S.next 的位置(比如在指令数组中的下标)，但暂不生成跳转目标
  - 这些位置被记录到 B 的综合属性 B.falselist 中
  - 当 S.next 的值确定时(即 S 的代码生成完毕时)，把 B.falselist 中的所有指令的目标都填上这个值
- **回填技术**
  - 生成跳转指令时暂时不指定跳转目标的标号，而是使用列表记录这些不完整的指令
  - 等知道确定的目标标号时再回过头去填写
  - **每个这样的列表中的指令都跳转到相同的目标**

布尔表达式的回填翻译

- 前面我们通过继承属性 B.true 和 B.false 记录跳转目标
- 为进行回填，我们转而使用两个综合属性：
  - B.truelist：其中的跳转指令在 B 的值为真时执行
  - B.falselist：其中的跳转指令在 B 的值为假时执行

- 辅助函数：
  - makelist(i)：创建一个只包含指令 i 的列表
  - merge(p1, p2)：合并两个列表 p1 和 p2
  - backpatch(p, i)：用指令 i 回填列表 p 中跳转指令的目标标号

通过回填翻译布尔表达式的 SDT

产生规则	语义动作
$B \to$ true	{ B.truelist = makelist(nextinstr); B.falselist = NULL; emit(`goto _`); }
$B \to$ false	{ B.truelist = NULL; B.falselist = makelist(nextinstr); } emit(`goto _`); }
$B \to (B_1)$	{ B.truelist = B_1.truelist; B.falselist = B_1.falselist; }
$B \to !B_1$	{ B.truelist = B_1.falselist; B.falselist = B_1.truelist; }

- 采用增量式翻译，假设全局变量 nextinstr 记录下一条语句的序号
- 综合属性 B.truelist：其中的跳转指令在 B 的值为真时执行
- 综合属性 B.falselist：其中的跳转指令在 B 的值为假时执行
- makelist(i)：创建一个只包含指令 i 的列表
- merge(p1, p2)：合并两个列表 p1 和 p2
- backpatch(p, i)：用指令 i 回填列表 p 中跳转指令的目标标号

通过回填翻译布尔表达式的 SDT

产生规则	语义动作
$B \to E_1 == E_2$	{ B.truelist = makelist(nextinstr); emit(`if {$E_1$.addr} == {$E_2$.addr} goto _`); B.falselist = makelist(nextinstr); emit(`goto _`); }

- 采用增量式翻译，假设全局变量 nextinstr 记录下一条语句的序号
- 综合属性 B.truelist：其中的跳转指令在 B 的值为真时执行
- 综合属性 B.falselist：其中的跳转指令在 B 的值为假时执行
- makelist(i)：创建一个只包含指令 i 的列表
- merge(p1, p2)：合并两个列表 p1 和 p2
- backpatch(p, i)：用指令 i 回填列表 p 中跳转指令的目标标号

通过回填翻译布尔表达式的 SDT

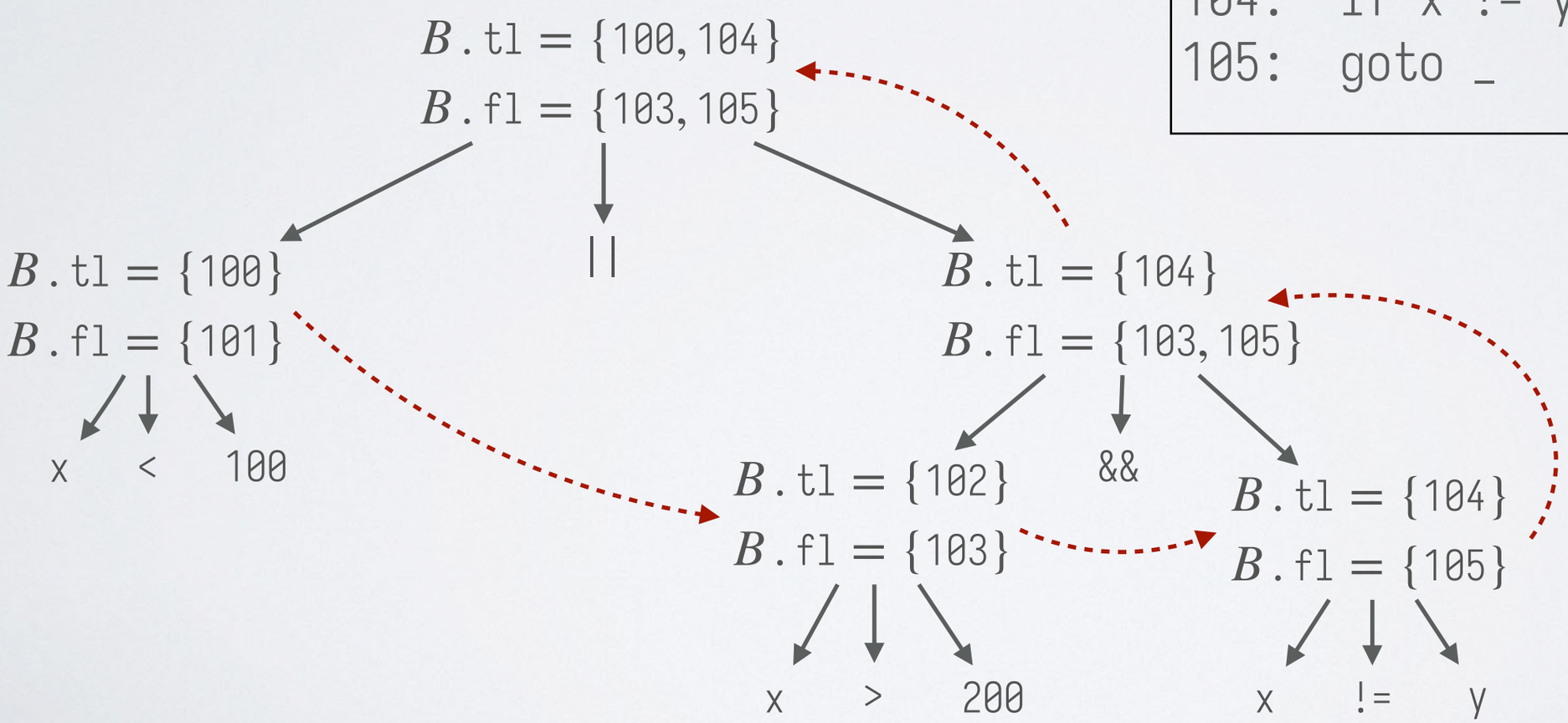
产生规则	语义动作
$B \to B_1$ && B_2	{ backpatch(B_1.truelist, nextinstr); } { B.truelist = B_2.truelist; B.falselist = merge(B_1.falselist, B_2.falselist); }
$B \to B_1$ || B_2	{ backpatch(B_1.falselist, nextinstr); } { B.truelist = merge(B_1.truelist, B_2.truelist); B.falselist = B_2.falselist; }

- 采用增量式翻译，假设全局变量 nextinstr 记录下一条语句的序号
- 综合属性 B.truelist：其中的跳转指令在 B 的值为真时执行
- 综合属性 B.falselist：其中的跳转指令在 B 的值为假时执行
- makelist(i)：创建一个只包含指令 i 的列表
- merge(p1, p2)：合并两个列表 p1 和 p2
- backpatch(p, i)：用指令 i 回填列表 p 中跳转指令的目标标号

布尔表达式的回填翻译示例

- 再次考虑表达式：
  - `x < 100 || (x > 200 && x != y)`
- 假设从 100 开始为指令编号

```
100:  if x < 100 goto _
101:  goto 102
102:  if x > 200 goto 104
103:  goto _
104:  if x != y goto _
105:  goto _
```



控制流语句的回填翻译

$$
\begin{aligned}
S ::= & \ \texttt{ID} = E \ ; \mid T \ \texttt{ID} \ ; \mid \{ \ L \ \} \\
\mid & \ \ \texttt{if} \ (\ B \ ) \ S \mid \texttt{if} \ (\ B \ ) \ S \ \texttt{else} \ S \\
\mid & \ \ \texttt{while} \ (\ B \ ) \ S \\
L ::= & \ \epsilon \mid L \ S
\end{aligned}
$$

- 前面我们通过继承属性 $S.\texttt{next}$ 和 $L.\texttt{next}$ 记录跳转目标
- 为进行回填，我们转而使用综合属性 $S.\texttt{nextlist}$ 和 $L.\texttt{nextlist}$，其中的跳转指令的目标是 S 和 L 执行完后紧接着执行的下一条指令

通过回填翻译控制流语句的 SDT

产生规则	语义动作
$S \to \texttt{ID} = E\ ;$	{ S.nextlist = NULL; }
$S \to T\ \texttt{ID}\ ;$	{ S.nextlist = NULL; }
$S \to \{ L \}$	{ S.nextlist = L.nextlist; }
$L \to \epsilon$	{ L.nextlist = NULL; }
$L \to L_1$	{ backpatch(L_1.nextlist, nextinstr); }
S	{ L.nextlist = S.nextlist; }

- 采用增量式翻译，假设全局变量 nextinstr 记录下一条语句的序号
- 综合属性 B.truelist：其中的跳转指令在 B 的值为真时执行
- 综合属性 B.falselist：其中的跳转指令在 B 的值为假时执行
- 综合属性 S/L.nextlist：其中的跳转指令的目标是紧接着 S/L 的下一条指令
- makelist(i)：创建一个只包含指令 i 的列表
- merge(p1, p2)：合并两个列表 p1 和 p2
- backpatch(p, i)：用指令 i 回填列表 p 中跳转指令的目标标号

通过回填翻译控制流语句的 SDT

产生规则	语义动作
$S \rightarrow$ if (B)	{ backpatch(B.truelist, nextinstr); }
S_1	{ S.nextlist = merge(B.falselist, S_1.nextlist); }
$S \rightarrow$ if (B)	{ backpatch(B.truelist, nextinstr); }
S_1 else	{ S_2.temp = merge(S_1.nextlist, makelist(nextinstr)); emit(`goto _`); backpatch(B.falselist, nextinstr); }
S_2	{ S.nextlist = merge(S_2.temp, S_2.nextlist) ; }

- 采用增量式翻译，假设全局变量 nextinstr 记录下一条语句的序号
- 综合属性 B.truelist：其中的跳转指令在 B 的值为真时执行
- 综合属性 B.falselist：其中的跳转指令在 B 的值为假时执行
- 综合属性 S/L.nextlist：其中的跳转指令的目标是紧接着 S/L 的下一条指令
- makelist(i)：创建一个只包含指令 i 的列表
- merge(p1, p2)：合并两个列表 p1 和 p2
- backpatch(p, i)：用指令 i 回填列表 p 中跳转指令的目标标号

通过回填翻译控制流语句的 SDT

产生规则	语义动作
$S \rightarrow$ while (	{ S_1.begin = nextinstr; }
B)	{ backpatch(B.truelist, nextinstr); }
S_1	{ backpatch(S_1.nextlist, S_1.begin);
	emit(`goto {$S_1$.begin}`);
	S.nextlist = B.falselist; }

- 采用增量式翻译，假设全局变量 nextinstr 记录下一条语句的序号
- 综合属性 B.truelist：其中的跳转指令在 B 的值为真时执行
- 综合属性 B.falselist：其中的跳转指令在 B 的值为假时执行
- 综合属性 S/L.nextlist：其中的跳转指令的目标是紧接着 S/L 的下一条指令
- makelist(i)：创建一个只包含指令 i 的列表
- merge(p1, p2)：合并两个列表 p1 和 p2
- backpatch(p, i)：用指令 i 回填列表 p 中跳转指令的目标标号

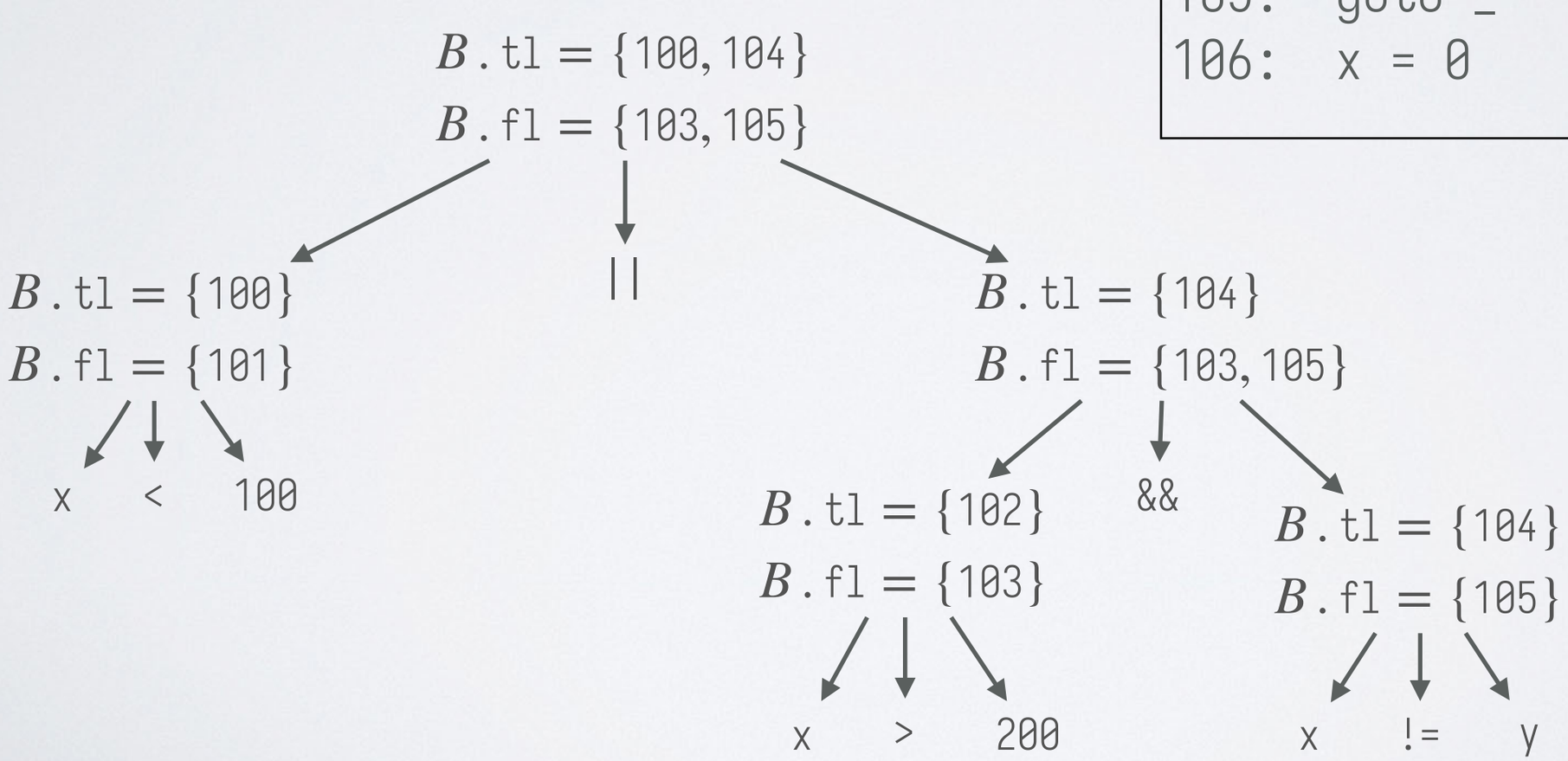
一个小例子

- 再次考虑语句 S:

```
if (x < 100 || (x > 200 && x != y))
  x = 0;
```

- 假设从 100 开始为指令编号
- $S.\mathtt{nl} = \{103, 105\}$

```
100:  if x < 100 goto 106
101:  goto 102
102:  if x > 200 goto 104
103:  goto _
104:  if x != y goto 106
105:  goto _
106:  x = 0
```



一个大例子

B_1 S_3 B_3 B_2

```
if ((x + y  >  z)  &&  (a == b))
    while  m <  n  { m = n + 10; }
else { a = b - m; }
```

B_4 S_1 S_2

```
100:  t1 = x + y
101:  if t1 > z goto 103
102:  goto 111
103:  if a == b goto 105
104:  goto 111
```

$B_3.\text{truelist} = \{103\}$

$B_3.\text{falselist} = \{104\}$

$B_1.\text{truelist} = \{103\}$

$B_1.\text{falselist} = \{102, 104\}$

```
105:  if m < n goto 107
106:  goto
107:  t2 = n + 10
108:  m = t2
109:  goto 105
110:  goto
```

$B_4.\text{truelist} = \{105\}$

$S_1.\text{nextlist} = \{106\}$

$B_4.\text{falselist} = \{106\}$

这句是多余的，在优化过程中可被删除

```
111:  t3 = b - m
112:  a = t3
```

$S_3.\text{nextlist} = \{110\}$

$S_2.\text{nextlist} = \{106\}$

小结

- 一个支持算术、函数、数组、分支、循环的类 C 语言
- 通过**语法制导的翻译方案(SDT)**来(**增量式**)翻译到三地址代码
  - 使用**符号表**管理名字和作用域
  - 使用**继承属性**或**回填**来处理控制流语句的短路求值(以及 break/continue)

$$
\begin{aligned}
P &::= \epsilon \mid FP \\
F &::= T\ \texttt{ID}\ (\ Params\ )\ \{\ L\ \} \\
T &::= \texttt{int}\ C \mid \texttt{double}\ C \\
C &::= \epsilon \mid [\ \texttt{INT}\ ]\ C \\
Params &::= \epsilon \mid T\ \texttt{ID}\ ,\ Params \\
L &::= \epsilon \mid L\ S \\
S &::= \texttt{ID} = E\ ; \mid R = E\ ; \mid T\ \texttt{ID}\ ; \mid \{\ L\ \} \mid \texttt{return}\ E\ ; \mid \texttt{if}\ (\ B\ )\ S \mid \texttt{if}\ (\ B\ )\ S\ \texttt{else}\ S \mid \texttt{while}\ (\ B\ )\ S \\
E &::= E + E \mid E - E \mid E * E \mid E / E \mid (\ E\ ) \mid R \mid \texttt{ID} \mid \texttt{INT} \mid \texttt{ID}\ (\ Args\ ) \\
R &::= R\ [\ E\ ] \mid \texttt{ID}\ [\ E\ ] \\
B &::= B \mid\mid B \mid B\ \&\&\ B \mid E == E \mid E <= E \mid (\ B\ ) \mid !\ B \mid \texttt{true} \mid \texttt{false} \\
Args &::= \epsilon \mid E\ ,\ Args
\end{aligned}
$$

主要内容

- 中间表示的作用
- 中间表示的设计
- 中间表示的生成

- One More Thing

主要内容

- 中间表示的作用
- 中间表示的设计
- 中间表示的生成

- ~~One More Thing~~
- 非三地址代码形式的中间表示的生成

回顾：静态单赋值形式（SSA）

- **Static Single Assignment**，简称 SSA
- LLVM IR、Koopa IR 均采用 SSA 形式的中间表示
- SSA 中的所有赋值都针对**不同名字**的变量
  - **对一个名字的使用可以找到唯一的定值(definition)**

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

三地址代码

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

SSA 形式

SSA 形式 IR 的生成

- 生成高质量的 SSA 代码不是那么容易的
  - 有机会的话，后面讲解程序的优化和分析时再进行讨论
- **问题之一：对程序本来的变量的多次赋值**
  - 临时变量总能生成新的，所以问题不大
- **解法之一：为每个程序变量开辟内存空间**
  - 使得变量对应唯一的内存位置
  - 每次使用时从内存中加载，每次更改后存储回内存
  - 比如：alloc 表示开辟内存，load 表示加载 $(x = *y)$，store 表示存储 $(*x = y)$

```
a = 1
b = 1
c = a + b
a = b
b = c
```

```
a  = alloc
store 1, a
b  = alloc
store 1, b
c  = alloc
t0 = load a
t1 = load b
t2 = t0 + t1
store t2, c
t3 = load b
store t3, a
t4 = load c
store t4, b
```

生成 SSA 形式 IR 的 SDT

- 采用增量式翻译
- genvar 和 gentmp 函数分别为程序变量和临时变量生成地址

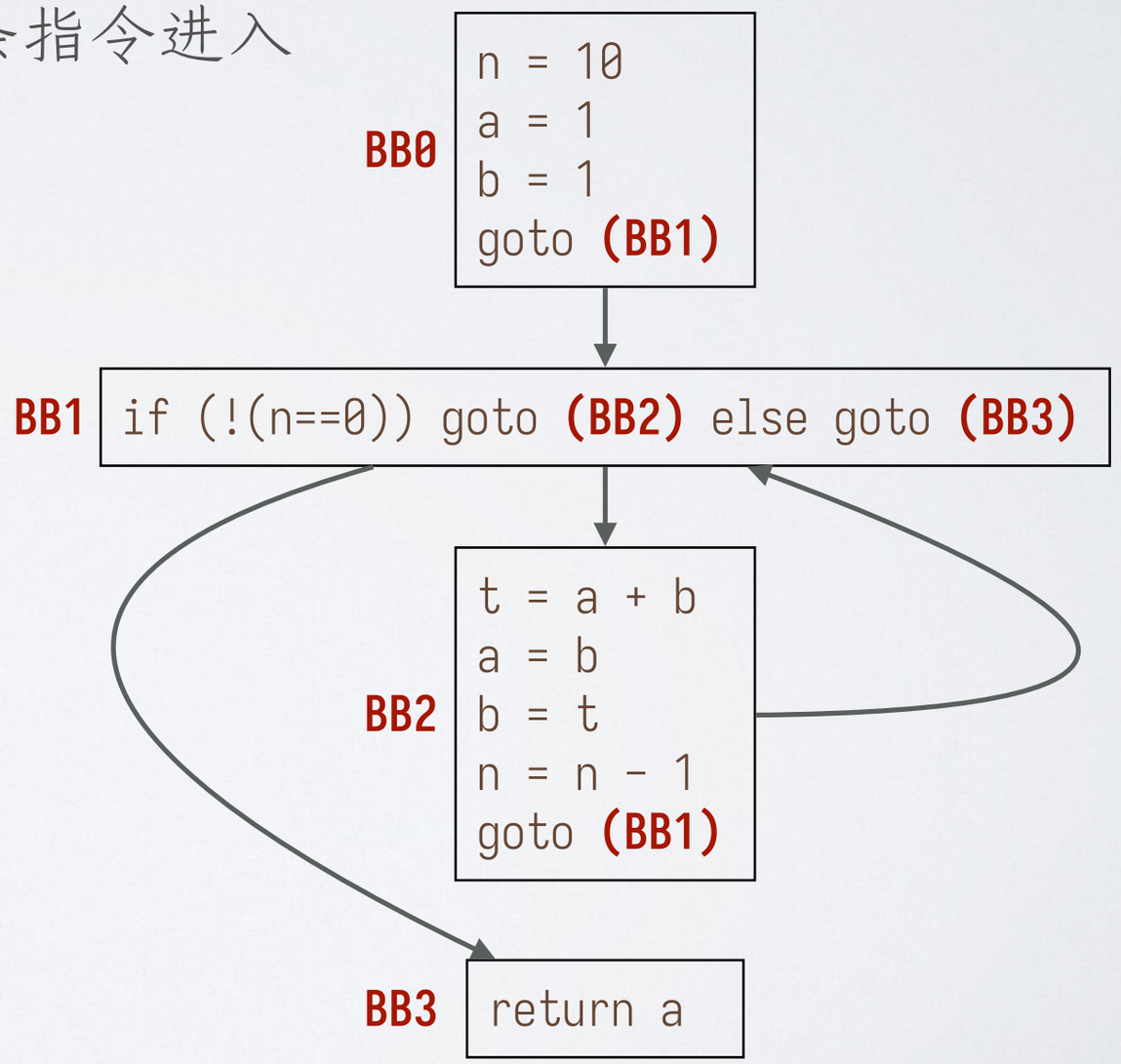
产生规则	语义动作
$S \to T$ ID ;	{ insert(ID.lexeme, T.type); emit(`{genvar(ID.lexeme)} = alloc`); }
$S \to$ ID = E ;	{ emit(`store {$E$.addr} {genvar(ID.lexeme)}`); }
$E \to$ ID	{ E.addr = gentmp(); emit(`{$E$.addr} = load {genvar(ID.lexeme)}`); }

- 可以进行的**优化**：
  - 在符号表中记录每个程序变量的**当前值**的地址
  - 在上一步的基础上，尽可能推迟往内存写回的操作

回顾：控制流图

- **控制流图(Control-Flow Graph，CFG)**
  - 有向图，图中结点为**基本块**(basic block)，边为控制流跳转
  - 基本块具有线性结构，其中最后一条语句为**跳转**或者**过程/函数返回**
  - 控制流只能从基本块的第一条指令进入

```
{
  n = 10; a = 1; b = 1;
  while (!(n == 0)) {
    t = a + b; a = b; b = t;
    n = n - 1;
  }
  return a;
}
```



控制流图形式 IR 的生成

- 采用增量式翻译
  - 需要记录当前正处于的基本块
  - `emit(instr)` 函数在当前基本块的末尾插入指令 `instr`
  - `new_bb()` 生成一个新的基本块，返回值可用于生成跳转指令
  - `set_bb(bb)` 设置当前所处的基本块为 `bb`

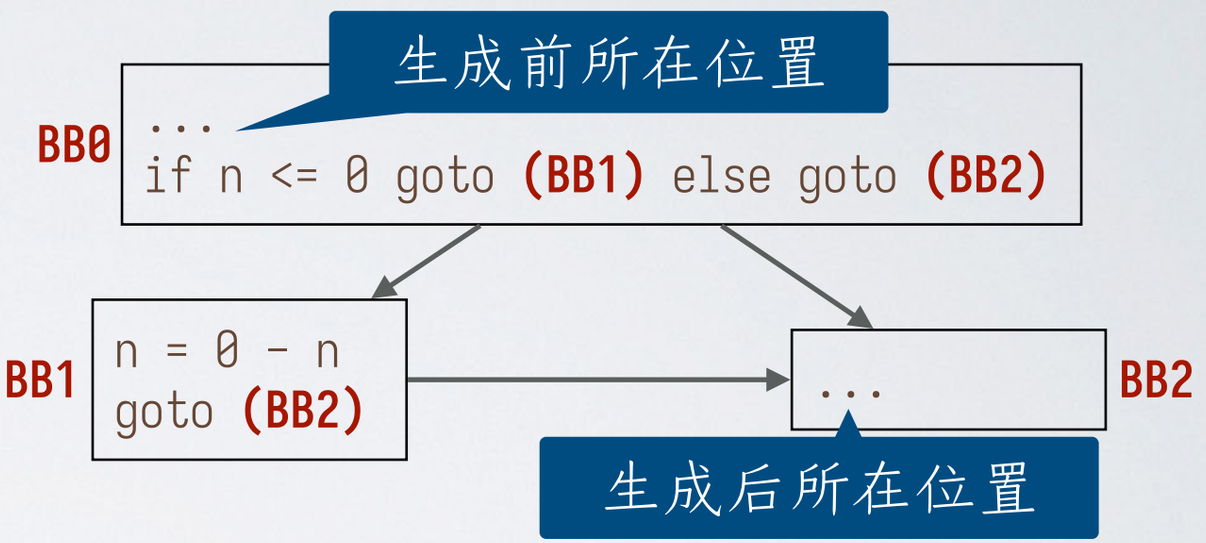
- 在生成时需确保**跳转**和**返回**指令只在基本块末尾出现

- 处理过程/函数层面时，还需确保每个过程/函数对应的控制流图有**唯一的入口基本块**

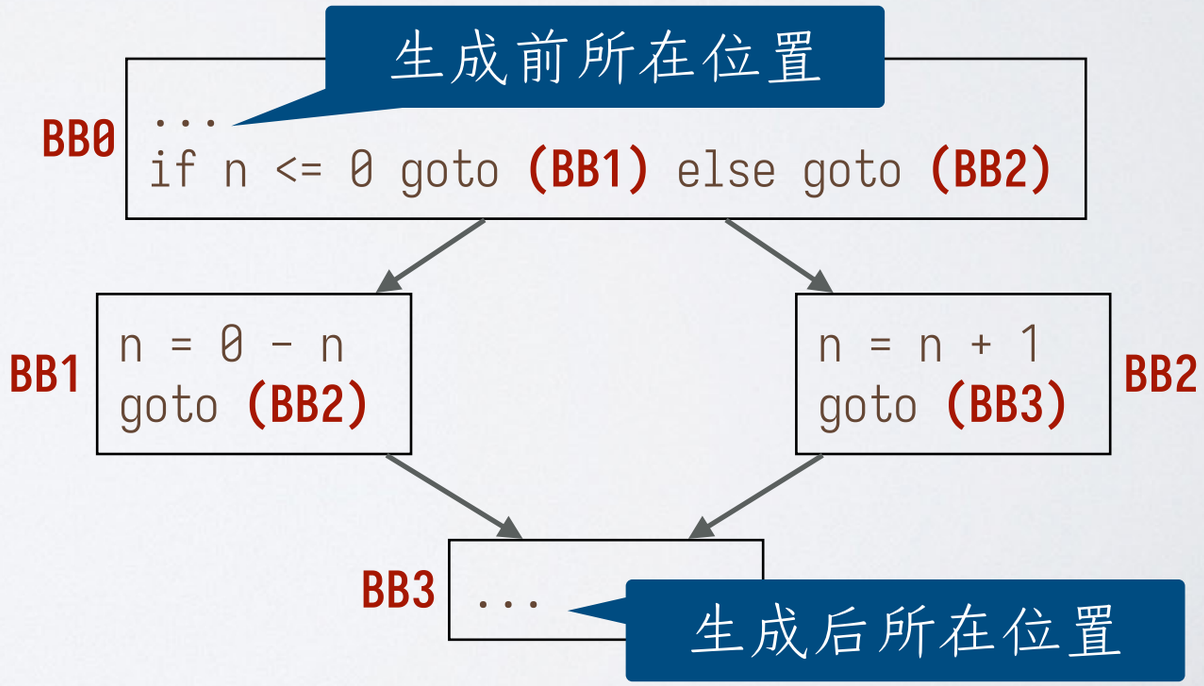
生成控制流图形式 IR 的 SDT（条件）

$$S ::= \cdots \mid \texttt{if}(B)S \mid \texttt{if}(B)S\,\texttt{else}\,S$$

```
if (n <= 0) {
  n = 0 - n;
}
```




```
if (n <= 0) {
  n = 0 - n;
} else {
  n = n + 1;
}
```



生成控制流图形式 IR 的 SDT（条件）

产生规则	语义动作
$S \to$ if (	{ B.true = new_bb(); B.false = S_1.next = new_bb(); }
B)	{ set_bb(B.true); }
S_1	{ emit(`goto {$S_1$.next}`); set_bb(S_1.next); }
$S \to$ if (	{ B.true = new_bb(); B.false = new_bb(); S_1.next = S_2.next = new_bb(); }
B)	{ set_bb(B.true); }
S_1 else	{ emit(`goto {$S_1$.next}`); set_bb(B.false); }
S_2	{ emit(`goto {$S_2$.next}`); set_bb(S_2.next); }

继承属性：

- B.true：B 为真时跳转到的基本块
- B.false：B 为假时跳转到的基本块

生成控制流图形式 IR 的 SDT（布尔）

```
if (n <= 0) ...
```

生成前所在位置

BB0

```
...
if n <= 0 goto B.true else goto B.false
```

```
if (n <= 0 && m > 0) ...
```

BB0

```
...
if n <= 0 goto (BB1) else goto B.false
```

BB1

```
if m > 0 goto B.true else goto B.false
```

```
if (n <= 0 || m > 0) ...
```

BB0

```
...
if n <= 0 goto B.true else goto (BB1)
```

BB1

```
if m > 0 goto B.true else goto B.false
```

生成控制流图形式 IR 的 SDT（布尔）

产生规则	语义动作
$B \to$ true	{ emit(`goto {$B$.true}`); }
$B \to$ false	{ emit(`goto {$B$.false}`); }
$B \to$ (B_1)	{ B_1.true = B.true; B_1.false = B.false; }
$B \to$! B_1	{ B_1.true = B.false; B_1.false = B.true; }
$B \to E_1$ == E_2	{ emit(`if {$E_1$.addr} == {$E_2$.addr} goto {$B$.true} else goto {$B$.false}`); }
$B \to E_1$ <= E_2	{ emit(`if {$E_1$.addr} <= {$E_2$.addr} goto {$B$.true} else goto {$B$.false}`); }

继承属性：

- B.true：B 为真时跳转到的基本块
- B.false：B 为假时跳转到的基本块

生成控制流图形式 IR 的 SDT（布尔）

产生规则	语义动作
$B \rightarrow$ B_1 && B_2	{ B_1.true = new_bb(); B_1.false = B.false; } { B_2.true = B.true; B_2.false = B.false; emit(`goto {$B_1$.true}`); set_bb(B_1.true); }
$B \rightarrow$ B_1 || B_2	{ B_1.true = B.true; B_1.false = new_bb(); } { B_2.true = B.true; B_2.false = B.false; emit(`goto {$B_1$.false}`); set_bb(B_1.false); }

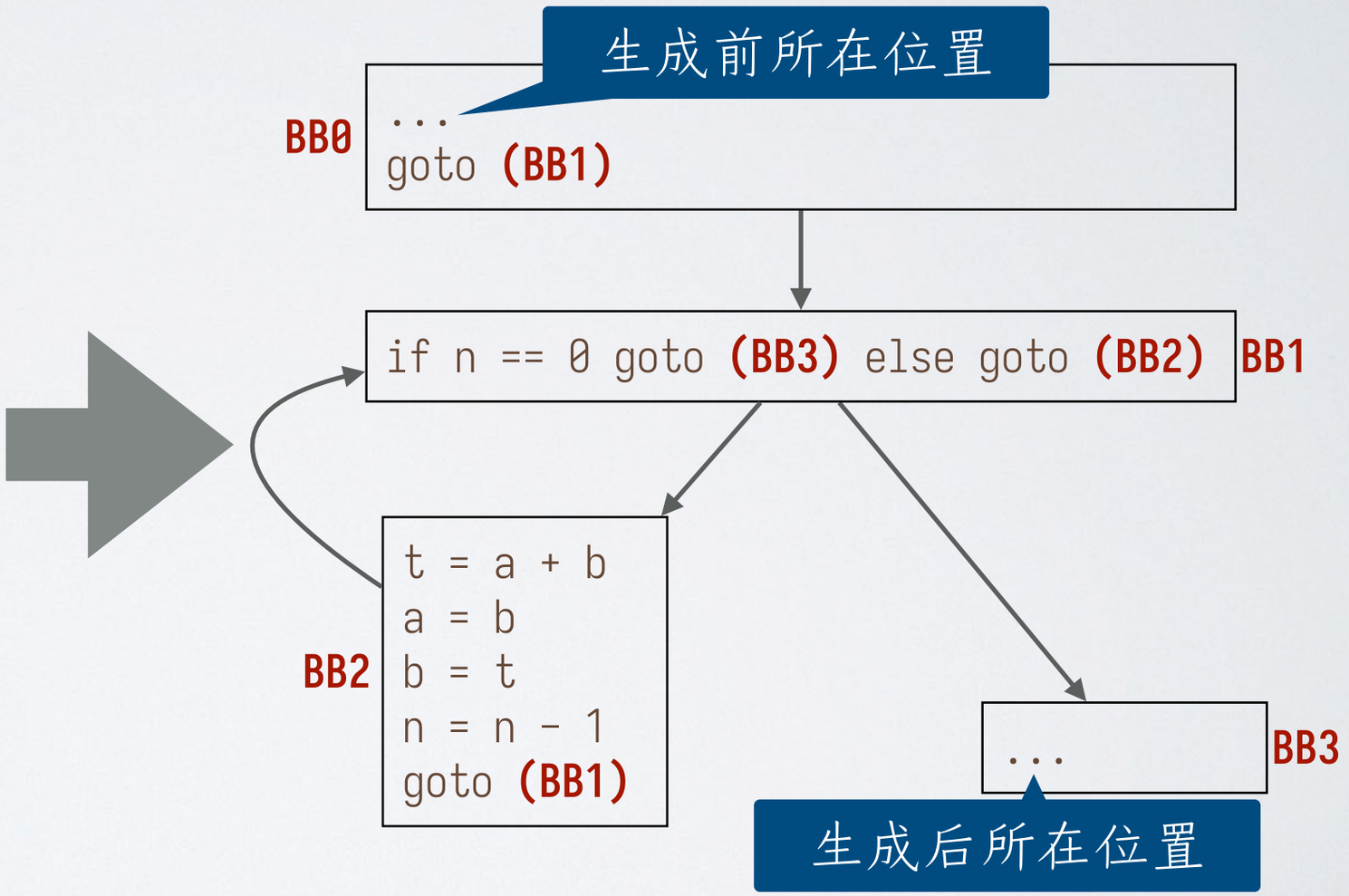
继承属性：

- B.true：B 为真时跳转到的基本块
- B.false：B 为假时跳转到的基本块

生成控制流图形式 IR 的 SDT（循环）

$$S ::= \cdots \mid \texttt{while}(B)S$$

```
while (!(n == 0)) {
  int t;
  t = a + b; a = b; b = t;
  n = n - 1;
}
```



生成控制流图形式 IR 的 SDT（循环）

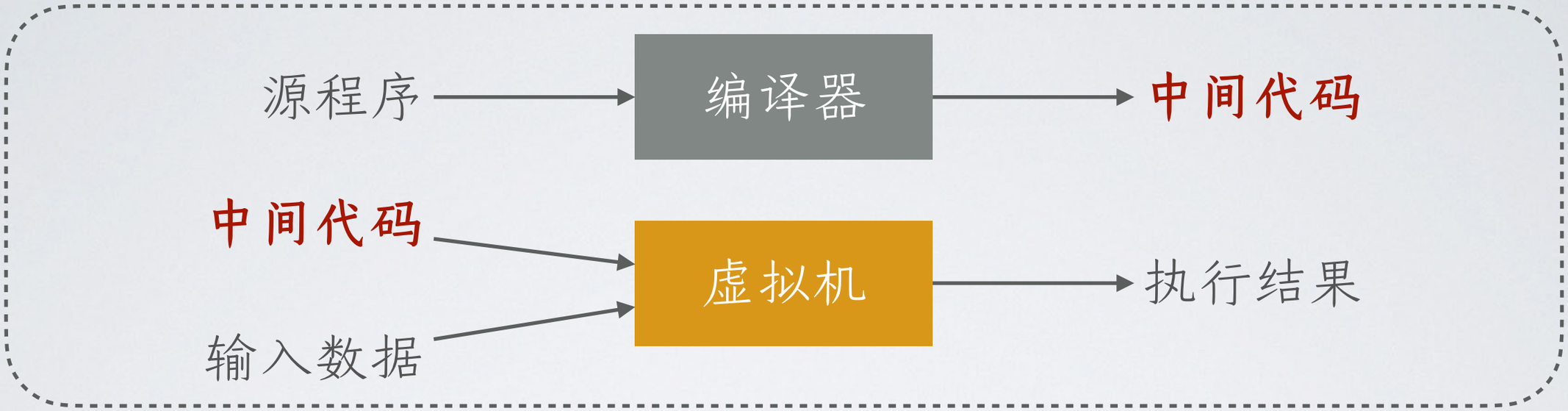
产生规则	语义动作
$S \to$ while (	{ B.true = new_bb(); B.false = new_bb(); S_1.next = new_bb(); emit(`goto {$S_1$.next}`); set_bb(S_1.next); }
B)	{ set_bb(B.true); }
S_1	{ emit(`goto {$S_1$.next}`); set_bb(B.false); }

练习：如何支持 break 和 continue?

继承属性：

- B.true：B 为真时跳转到的基本块
- B.false：B 为假时跳转到的基本块

栈式 IR



- 很多虚拟机采用了**栈式 IR**，比如 JVM
- 虚拟机状态可视为**二元组** (c, s)，其中 c 为代码、s 为栈
  - c 通常是一个指令序列，其中的指令可以对栈进行操作
  - s 通常用来保存中间计算结果(以及处理过程/函数的调用和返回)
  - 有的虚拟机会设计多种栈来负责不同的功能

栈式 IR：四则运算

指令	作用	执行前的栈	执行后的栈
CST(i)	把常数 i 压栈	s	s, i
ADD	把栈顶两数相加	s, i_1, i_2	$s, (i_1 + i_2)$
SUB	把栈顶两数相减	s, i_1, i_2	$s, (i_1 - i_2)$
MUL	把栈顶两数相乘	s, i_1, i_2	$s, (i_1 \times i_2)$
DIV	把栈顶两数相除	s, i_1, i_2	$s, (i_1 \div i_2)$

1 + 2 + 3

```
CST(1)
CST(2)
ADD
CST(3)
ADD
```

1 + 2 * 3

```
CST(1)
CST(2)
CST(3)
MUL
ADD
```

(1 + 2) * 3

```
CST(1)
CST(2)
ADD
CST(3)
MUL
```

栈式 IR：数据存取

指令	作用	执行前的栈	执行后的栈
INCST(m)	栈增长 m 个位置	s	$s, v_1, \cdots, v_m$
DECST(m)	栈缩减 m 个位置	$s, v_1, \cdots, v_m$	s
LDI	读取栈中位置的值	s, i	$s, s[i]$
STI	把值写入栈中位置	s, i, v	s, v

$s[i]$ 更改为 v

```
{ int n; int a;
  n = 10; a = 5;
  { int n;
    n = a + a;
    a = n;
  }
  n = n - 2;
}
```

```
INCST(2)
CST(0)
CST(10)
STI
DECST(1)
CST(1)
CST(5)
STI
DECST(1)
```

```
INCST(1)
CST(2)
CST(1)
LDI
CST(1)
LDI
ADD
STI
DECST(1)
CST(1)
CST(2)
LDI
STI
DECST(1)
DECST(1)
```

```
CST(0)
CST(0)
LDI
CST(2)
SUB
STI
DECST(1)
DECST(2)
```

生成栈式 IR 的 SDT

$$
\begin{aligned}
&S ::= \texttt{ID} = E\ ; \mid \texttt{int}\ \texttt{ID}\ ; \mid \{\ L\ \}\\
&L ::= \epsilon \mid L\ S\\
&E ::= E + E \mid E - E \mid E * E \mid E / E \mid (\ E\ ) \mid \texttt{ID} \mid \texttt{INT}
\end{aligned}
$$

offset 获取变量在栈中的存放位置

产生规则	
$E \to \texttt{ID}$	{ E.code = `CST({lookup(ID.lexeme).offset})` ‖ `LDI`; }
$E \to \texttt{INT}$	{ E.code = `CST({INT.intval})`; }
$E \to E_1 + E_2$	{ E.code = E_1.code ‖ E_2.code ‖ `ADD`; }
$S \to \texttt{ID} = E\ ;$	{ S.code = `CST({lookup(ID.lexeme).offset})` ‖ E.code ‖ `STI` ‖ `DECST(1)`; }
$S \to \texttt{int}\ \texttt{ID}\ ;$	{ insert(ID.lexeme, ...); 设置变量在栈中的存放位置(可以根据作用域总长度确定) S.code = `INCST(1)`; }
$S \to \{$ L $\}$	{ push_scope(); } 当前作用域的长度等于恰好在其中声明的变量数 { m = pop_scope().len; S.code = L.code ‖ `DECST({m})`; }

栈式 IR：控制流

指令	作用	执行前的栈	执行后的栈
GOTO(a)	跳转到标号 a 处	s	s
IFEQ(a)	如果栈顶为零则跳转到 a 处	s, v	s
IFLE(a)	如果栈顶非正则跳转到 a 处	s, v	s

- 栈式 IR 是线性 IR，对控制流的翻译和三地址代码相似
  - 条件跳转指令考察栈顶的值，所以要把需要的值压栈
- 前面生成三地址代码中使用**继承属性**或**回填**的技术仍然适用
- 对过程/函数的支持需要合理安排栈中内容
  - 下一讲：**运行时刻环境**

本讲小结

- 中间表示的作用
  - 表示程序语义、解耦前端和后端、简化优化器的设计
- 中间表示的设计
  - 组织结构(图状、线性)、抽象层次(靠近源或者目标)、命名策略
  - 三地址代码、控制流图、它们的相互转化
- 中间表示的生成
  - 基于语法制导的翻译方案(SDT)进行三地址代码的生成
  - 赋值语句、声明语句、布尔表达式的短路求值、条件和循环语句
  - 回填

思考问题

- 为什么编译过程需要中间表示？
- 中间表示中需要保留编程语言的哪些信息？
- 一般而言，图状 IR 抽象层次相对高，线性 IR 抽象层次相对低，为什么？
- 编译过程中大概设计多少种中间形式比较合适？
- 增量式翻译与非增量式的相比，有什么优点和缺点？
- 如何通过回填来支持 break、continue、goto 的翻译？
- 面向对象语言、函数式语言的 IR 需要考虑哪些特性？