

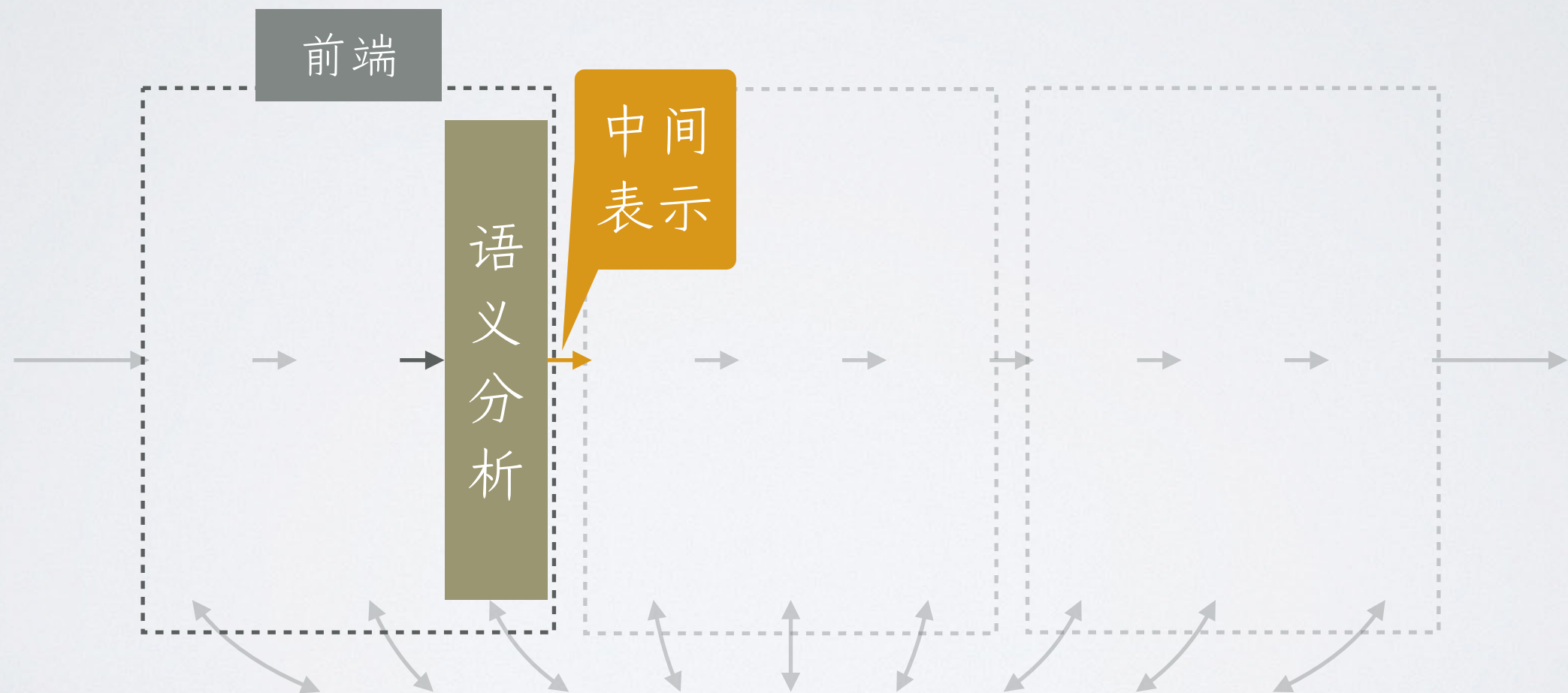


第十二讲 中间表示（进阶版）

Advanced Topics in Intermediate Representations

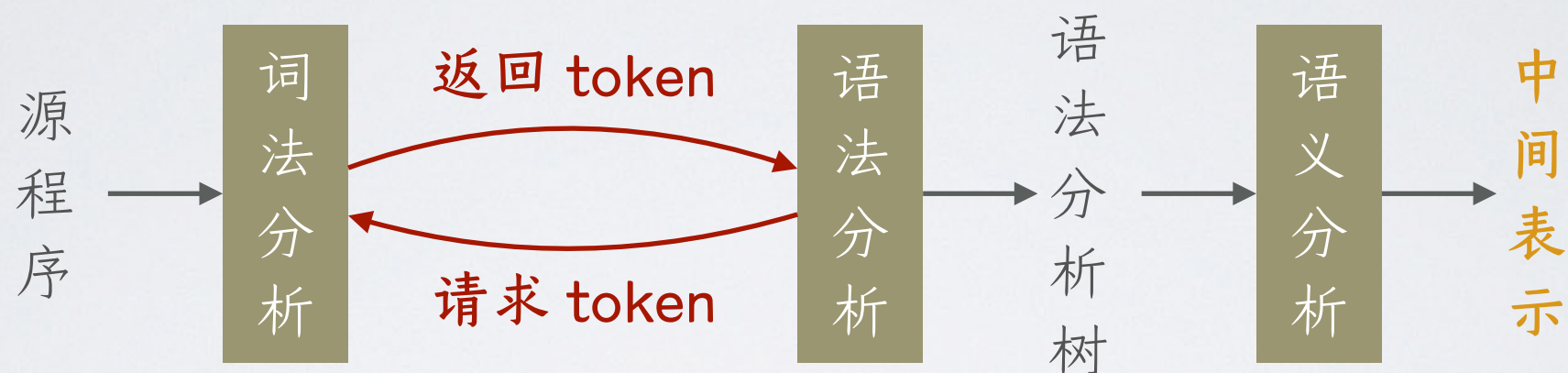
回顾：中间表示的作用

- 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

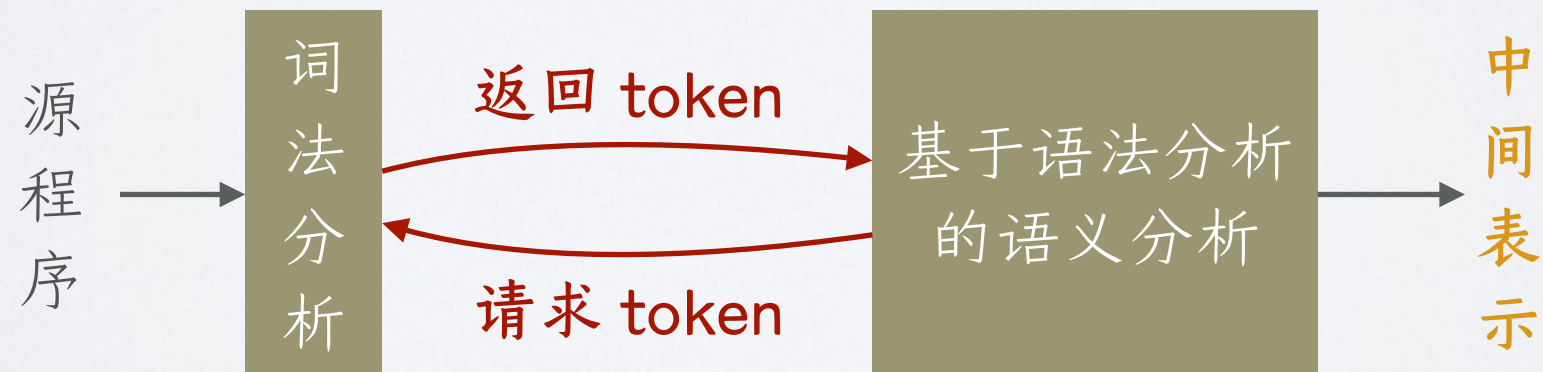


回顾：基于语法分析实现语义分析

- 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



主要内容

- ◎ 基于语法制导的翻译方案的中间表示生成
- ◎ 非三地址代码形式的中间表示的生成
- ◎ 静态单赋值形式(进阶版)
- ◎ Multi-Level IR (MLIR)

主要内容

- 基于语法制导的翻译方案的中间表示生成
- 非三地址代码形式的中间表示的生成
- 静态单赋值形式(进阶版)
- Multi-Level IR (MLIR)

回顾：语法制导的翻译方案

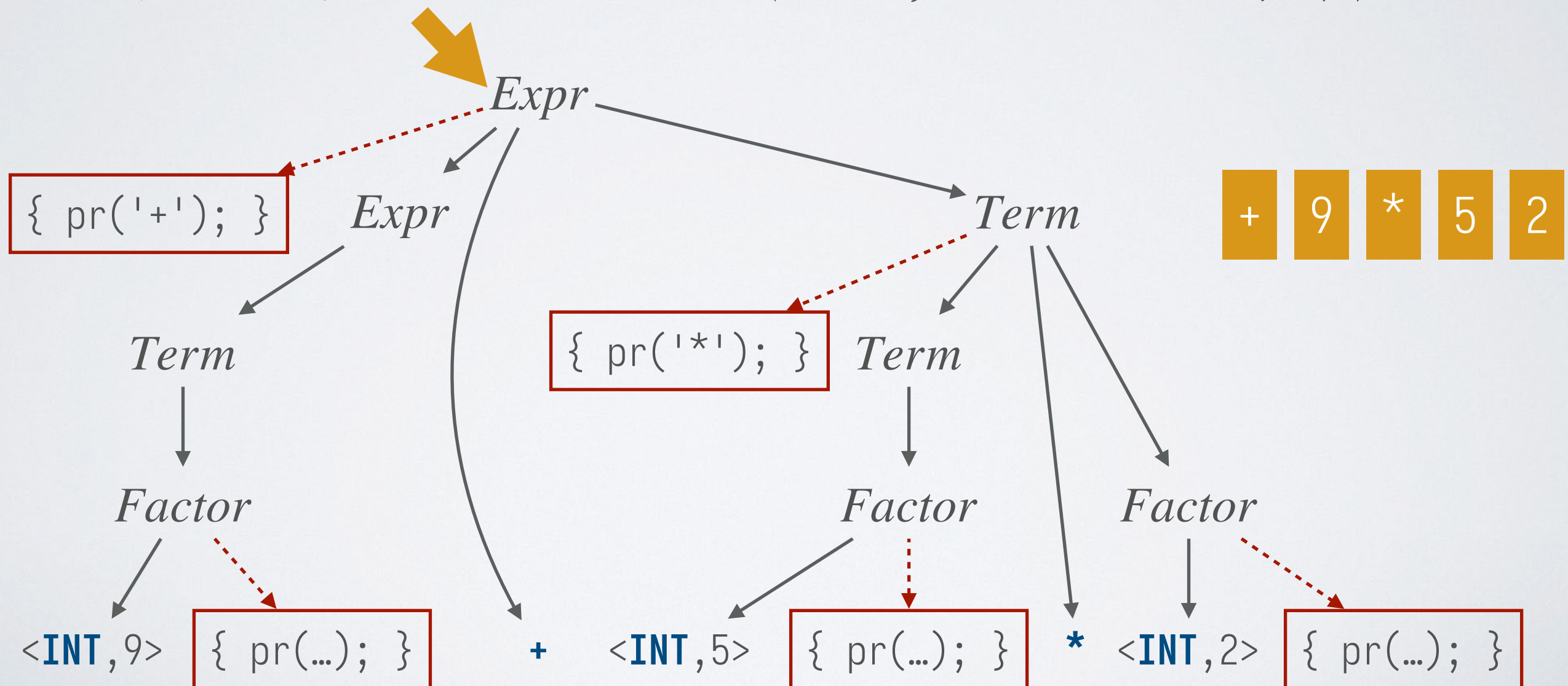
- 把属性计算规则改写为计算属性值的**程序片段**，并用花括号 $\{ \}$ 括起来，插入到产生规则**右侧**的任何合适的位置上
- 理解为深度优先遍历语法分析树时，何时执行程序片段

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

也可以写在一行上：
 $Expr \rightarrow \{ \text{print}(' + '); \} Expr_1 + Term$

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



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

genvar 函数为程序变量生成地址	语义动作
$E \rightarrow ID$	$\{ E.addr = \text{genvar}(); E.code = \{E.addr\}; \}$
$E \rightarrow INT$	$\{ E.addr = \text{gencst}(INT.intval); E.code = \{E.addr\}; \}$
$E \rightarrow E_1 + E_2$	$\{ E.addr = \text{gentmp}(); E.code = E_1.code E_2.code \{E.addr\} = \{E_1.addr\} + \{E_2.addr\}; \}$
$E \rightarrow (E_1)$	$\{ E.addr = E_1.addr; E.code = E_1.code; \}$
$S \rightarrow ID = E;$	$\{ S.code = E.code \{ \text{genvar}(ID.lexeme) \} = \{E.addr\}; \}$

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： $S \rightarrow X Y \{ a \}$ 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
$E \rightarrow \text{ID}$	{ $E.\text{addr} = \text{genvar}(\text{ID.lexeme});$ }
$E \rightarrow \text{INT}$	{ $E.\text{addr} = \text{gencst}(\text{INT.intval});$ }
$E \rightarrow E_1 + E_2$	{ $E.\text{addr} = \text{gentmp}();$ emit (`{ $E.\text{addr}$ } = { $E_1.\text{addr}$ } + { $E_2.\text{addr}$ }`); }
$E \rightarrow (E_1)$	{ $E.\text{addr} = E_1.\text{addr};$ }
$S \rightarrow \text{ID} = E;$	{ emit (`{ $\text{genvar}(\text{ID.lexeme})$ } = { $E.\text{addr}$ }`); }

翻译声明语句和语句块的 SDT

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

```

{ 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

$$T ::= \dots \mid \text{struct} \{ D \}$$

$$D ::= \epsilon \mid T \text{ ID} ; D$$

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

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

$T ::= T_B C$

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

数组分量

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

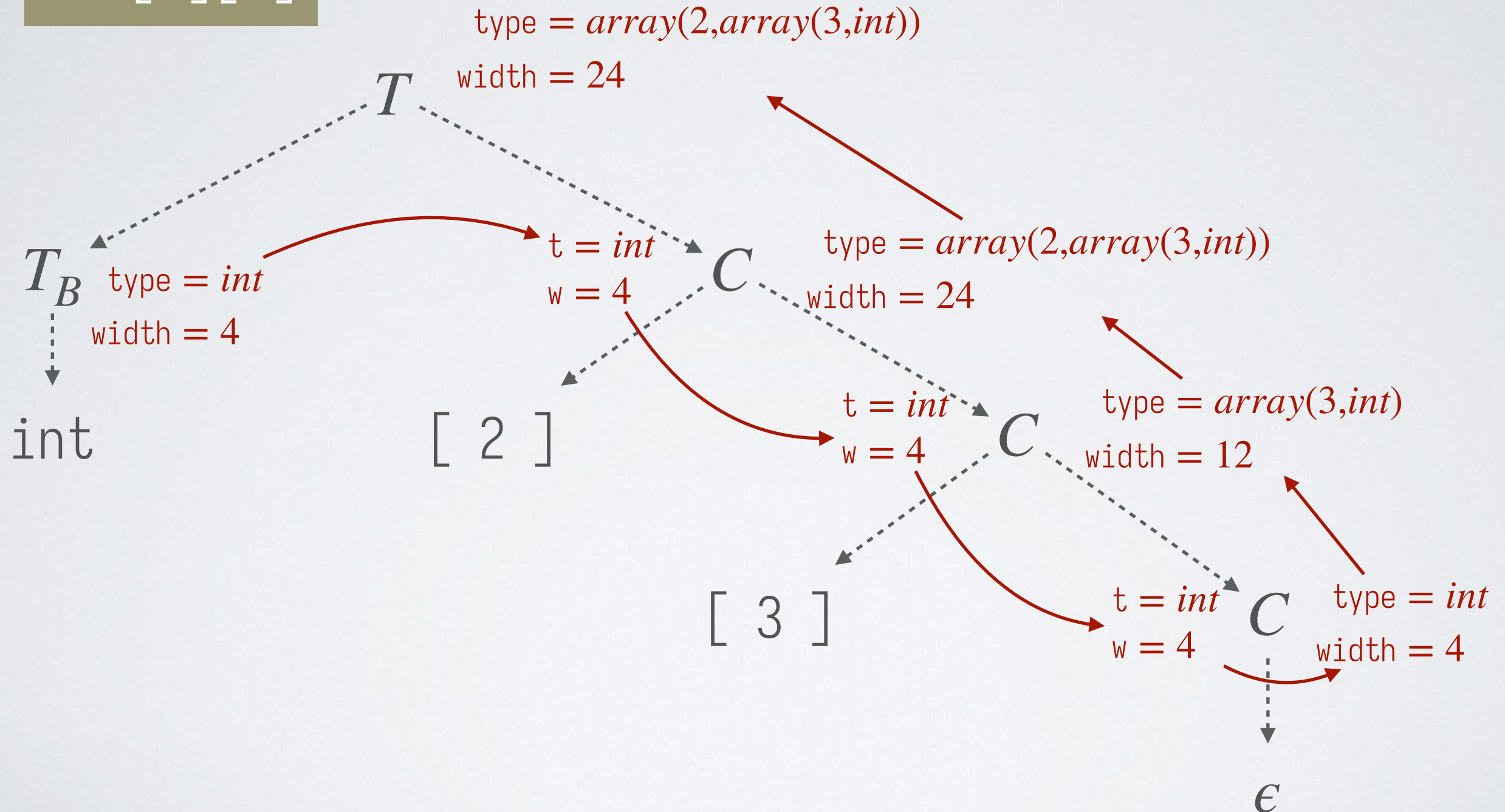
产生规则

用继承属性 t 和 w 记录数组元素的类型和宽度

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

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: $R ::= R[E] \mid \text{ID}[E]$
- ❖ 数组读取: $E ::= \dots \mid R$
- ❖ 数组赋值: $S ::= \dots \mid R = E;$

◎ 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 $A[0][i_2] \dots [i_k]$, 然后存放 $A[1][i_2] \dots [i_k]$, $\dots$
 - ❖ 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: $w_k = w$
 - ❖ $w_{j-1} = n_j \times w_j$
- ◎ $A[i_1][i_2] \dots [i_k]$ 的地址
 - ❖ $base + i_1 \times w_1 + i_2 \times w_2 + \dots + i_k \times w_k$
 - ❖ 或者
 - ❖ $base + ((\dots((i_1 \times n_2 + i_2) \times n_3 + i_3) \dots) \times n_k + i_k) \times w$

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 $base - low \times w$

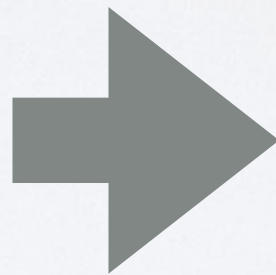
数组引用的翻译

$$R ::= R[E] \mid \text{ID}[E]$$

$$E ::= \dots \mid R$$

$$S ::= \dots \mid R = E;$$

```
int[2][3] a;
int[5] b;
...
a[i * j + k][1] = b[m - n];
```



```
t0 = i * j
t1 = t0 + k
t2 = t1 * 3
t3 = t2 + 1
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

$$R ::= R[E] \mid ID[E]$$

$$E ::= \dots \mid R$$

$$S ::= \dots \mid R = E;$$

产生规则	例: <i>array(2,array(3,int))</i>	例: <i>array(3,int)</i>
$R \rightarrow ID[E]$	<pre>{ R.addr = E.addr; R.base = genvar(ID.lexeme); R.type = lookup(ID.lexeme).type.elem; R.code = E.code; }</pre>	
$R \rightarrow R_1[E]$	<pre>{ R.addr = gentmp(); R.base = R₁.base; R.type = R₁.type.elem; t = gentmp(); R.code = R₁.code E.code `{t} = {R<sub>1</sub>.addr} * {R<sub>1</sub>.type.len}` `{R.addr} = {t} + {E.addr}`; }</pre>	
$E \rightarrow R$	<pre>{ E.addr = gentmp(); t = gentmp(); E.code = R.code `{t} = {R.addr} * {R.type.width}` `{E.addr} = {R.base} [ {t} ]`; }</pre>	

例: *int*

例: 3

例: 4

数组读取翻译示例

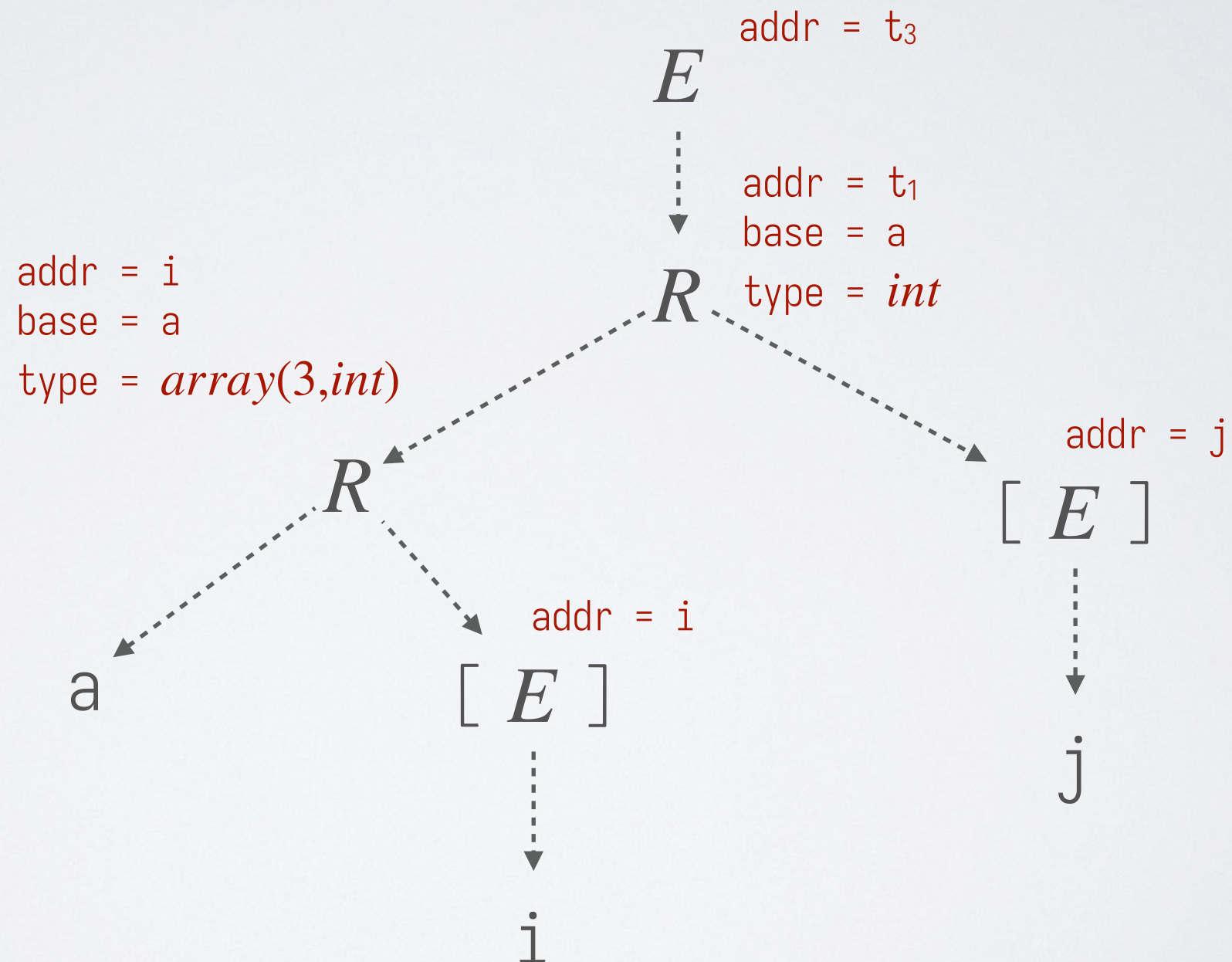
a 是 $\text{int}[2][3]$ 的数组

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

$a[i][j]$

```

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

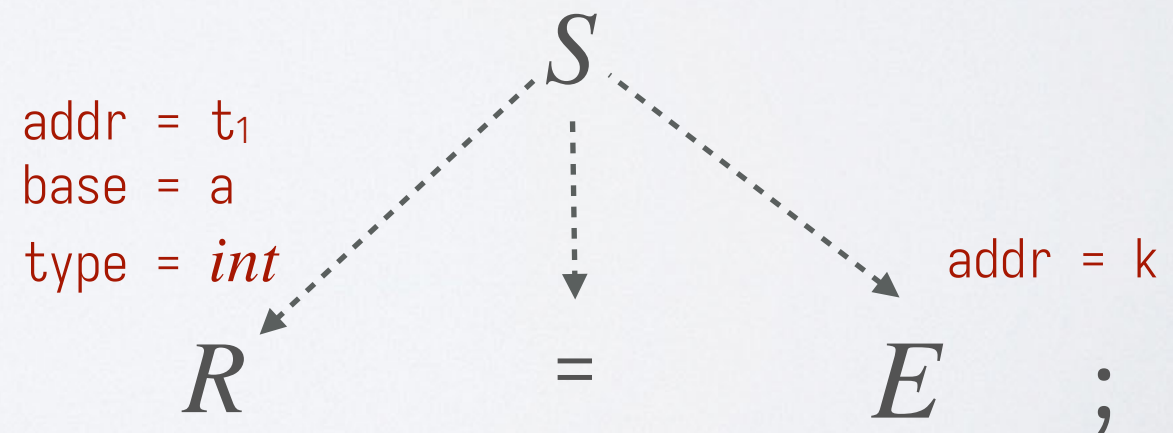
$$R ::= R[E] \mid ID[E]$$

$$E ::= \dots \mid R$$

$$S ::= \dots \mid R = E ;$$

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

$a[i][j] = k;$

$$\begin{aligned} t_0 &= i * 3 \\ t_1 &= t_0 + j \\ t_2 &= t_1 * 4 \\ a[t_2] &= k \end{aligned}$$


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： $\text{if}(B) S$
 - ❖ 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 $S.\text{next}$ 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 $\text{if}(B)S$

- ❖ 记录 B 的代码中跳转指令 $\text{goto } S.\text{next}$ 、 $\text{if} \dots \text{goto } S.\text{next}$ 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 $B.\text{falselist}$ 中
- ❖ 当 $S.\text{next}$ 的值确定时(即 S 的代码生成完毕时), 把 $B.\text{falselist}$ 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
$B \rightarrow \text{true}$	{ $B.\text{truelist} = \text{makelist}(\text{nextinstr}); B.\text{falselist} = \text{NULL};$ $\text{emit}(\text{'goto _'});$ }
$B \rightarrow \text{false}$	{ $B.\text{truelist} = \text{NULL}; B.\text{falselist} = \text{makelist}(\text{nextinstr});$ } $\text{emit}(\text{'goto _'});$ }
$B \rightarrow (B_1)$	{ $B.\text{truelist} = B_1.\text{truelist}; B.\text{falselist} = B_1.\text{falselist};$ }
$B \rightarrow !B_1$	{ $B.\text{truelist} = B_1.\text{falselist}; B.\text{falselist} = B_1.\text{truelist};$ }

- 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 $B.\text{truelist}$: 其中的跳转指令在 B 的值为真时执行
- 综合属性 $B.\text{falselist}$: 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
$B \rightarrow E_1 == E_2$	<pre>{ B.truelist = makelist(nextinstr); emit(`if {E<sub>1</sub>.addr} == {E<sub>2</sub>.addr} goto _`); B.falselist = makelist(nextinstr); emit(`goto _`); }</pre>

- 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
$B \rightarrow B_1 \&\& B_2$	<pre>{ backpatch(B_1.truelist, nextinstr); } { B.truelist = B_2.truelist; B.falselist = merge(B_1.falselist, B_2.falselist); }</pre>
$B \rightarrow B_1 B_2$	<pre>{ backpatch(B_1.falselist, nextinstr); } { B.truelist = merge(B_1.truelist, B_2.truelist); B.falselist = B_2.falselist; }</pre>

- 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

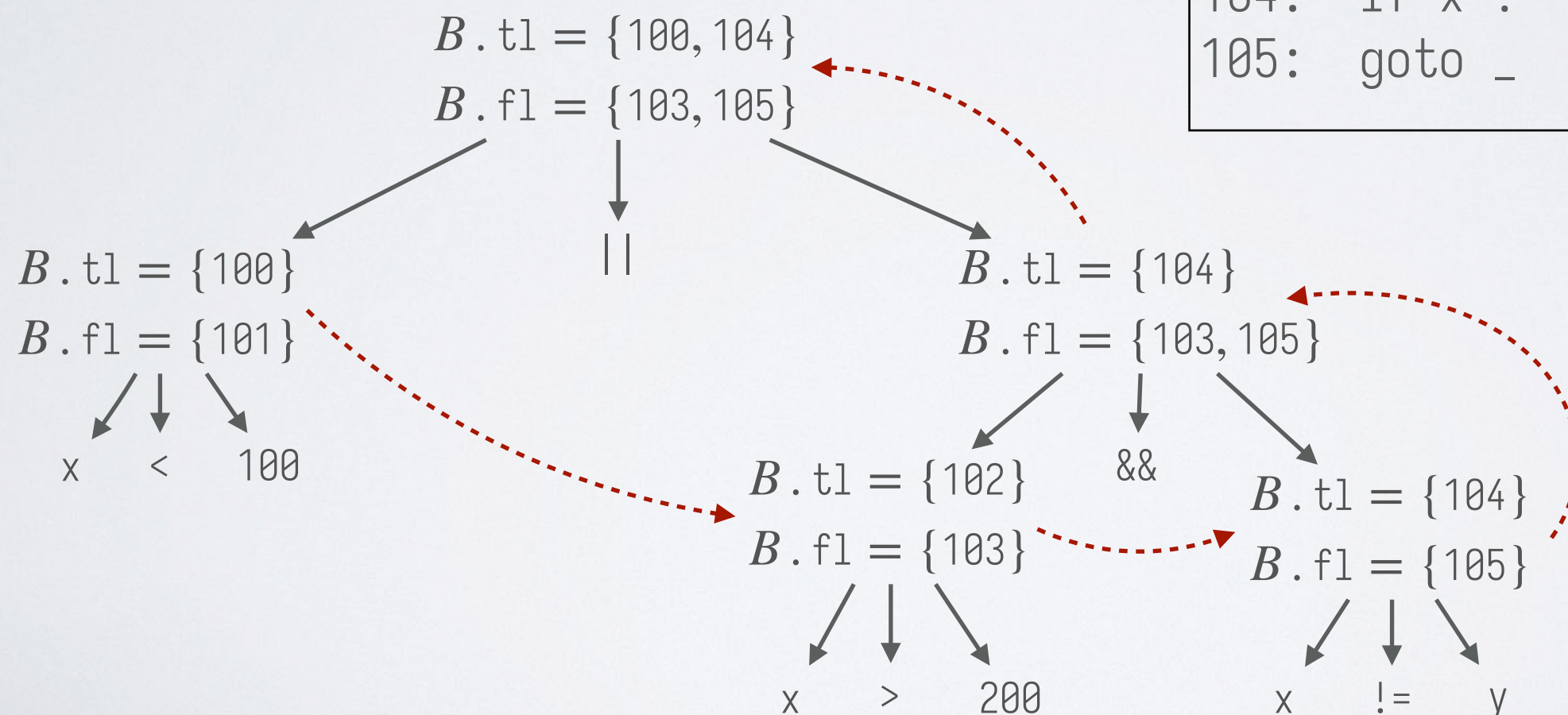
布尔表达式的回填翻译示例

再次考虑表达式:

❖ $x < 100 \ || \ (x > 200 \ \&\& \ x \neq y)$

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

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

- 前面我们通过继承属性 $S.next$ 和 $L.next$ 记录跳转目标
- 为进行回填, 我们转而使用综合属性 $S.nextlist$ 和 $L.nextlist$, 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
$S \rightarrow ID = E;$	{ $S.nextlist = NULL;$ }
$S \rightarrow T ID;$	{ $S.nextlist = NULL;$ }
$S \rightarrow \{L\}$	{ $S.nextlist = L.nextlist;$ }
$L \rightarrow \epsilon$	{ $L.nextlist = NULL;$ }
$L \rightarrow L_1$	{ $backpatch(L_1.nextlist, nextinstr);$ }
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
$S \rightarrow \text{if}(B)$ S_1	<pre>{ backpatch(B.truelist, nextinstr); }</pre> <pre>{ S.nextlist = merge(B.falselist, S_1.nextlist); }</pre>
$S \rightarrow \text{if}(B)$ $S_1 \text{ else}$ S_2	<pre>{ backpatch(B.truelist, nextinstr); }</pre> <pre>{ S_2.temp = merge(S_1.nextlist, makelist(nextinstr));</pre> <pre>emit(`goto _`);</pre> <pre>backpatch(B.falselist, nextinstr); }</pre> <pre>{ S.nextlist = merge(S_2.temp, S_2.nextlist) ; }</pre>

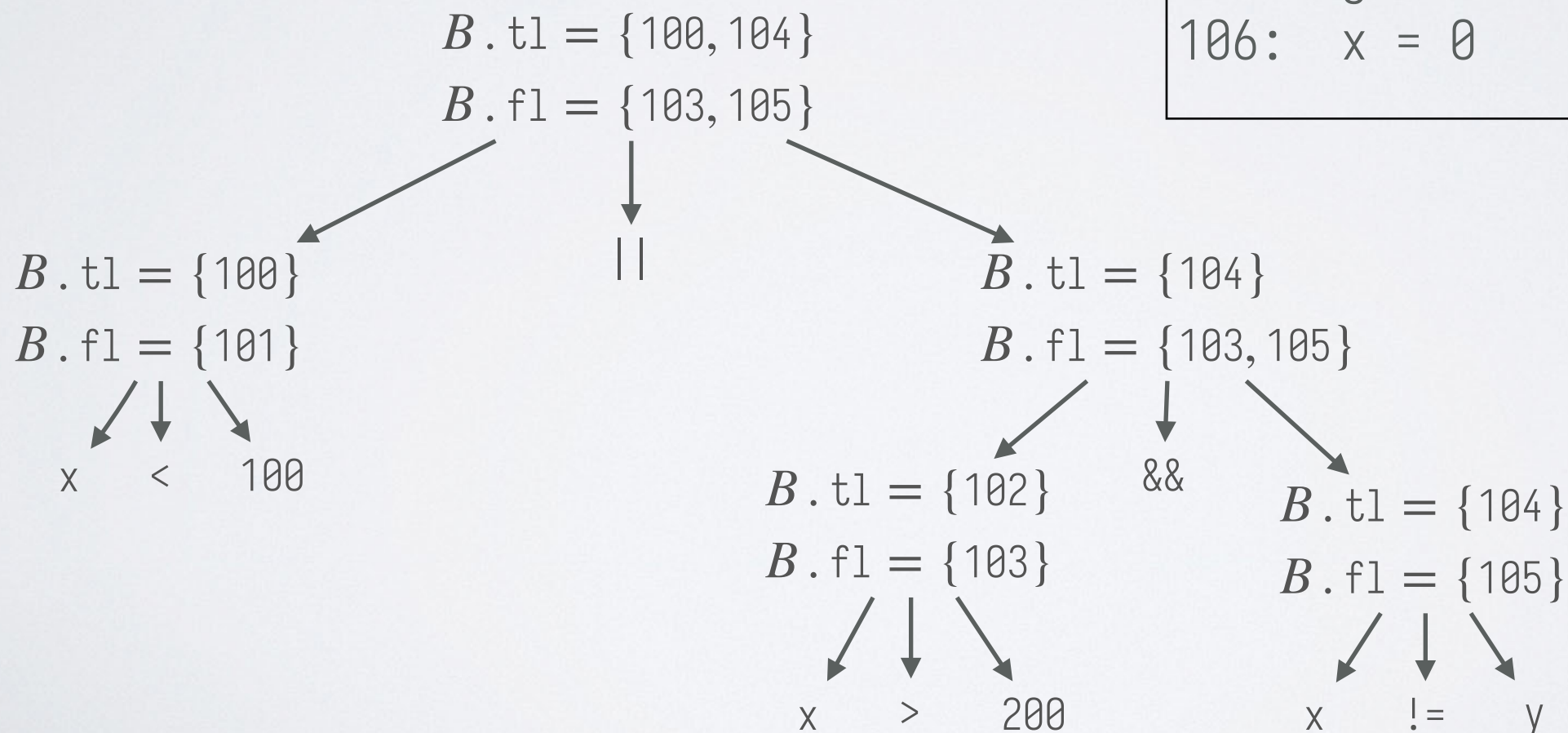
- 采用增量式翻译, 假设全局变量 `nextinstr` 记录下一条语句的序号
- 综合属性 `$B$ .truelist`: 其中的跳转指令在 B 的值为真时执行
- 综合属性 `$B$ .falselist`: 其中的跳转指令在 B 的值为假时执行
- 综合属性 `$S/L$ .nextlist`: 其中的跳转指令的目标是紧接着 S/L 的下一条指令
- `makelist( $i$ )`: 创建一个只包含指令 i 的列表
- `merge( $p1$ ,  $p2$ )`: 合并两个列表 $p1$ 和 $p2$
- `backpatch( $p$ ,  $i$ )`: 用指令 i 回填列表 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
$S \rightarrow \text{while} ($ $B)$ S_1	<pre> { $S_1.begin = \text{nextinstr};$ } { $\text{backpatch}(B.truelist, \text{nextinstr});$ } { $\text{backpatch}(S_1.nextlist, S_1.begin);$ $\text{emit}(\text{'goto } \{S_1.begin\}')$; $S.nextlist = B.falselist;$ } </pre>

- 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

- ```
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

|              |                    |           |                     |
|--------------|--------------------|-----------|---------------------|
|              | $B_1$              | $S_3 B_3$ | $B_2$               |
| <b>if</b>    | $((x + y) > z)$    | $\&\&$    | $(a == b)$          |
|              | $B_4$              | $S_1$     |                     |
| <b>while</b> | $m < n$            | $S_2$     | $\{ m = n + 10; \}$ |
| <b>else</b>  | $\{ a = b - m; \}$ |           |                     |

```

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

```

```

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

```

```

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

```

$B_3 B_1 \text{truelist} = \{103\}$   
 $B_3 B_1 \text{fafalselist} = \{102, 104\}$

$B_4 \text{truelist} = \{105\}$   
 $S_1 \text{nextlist} = \{106\}$   
 $B_4 \text{fafalselist} = \{106\}$

$S_3 \text{nextlist} = \{106\}$   
 $S_2 \text{nextlist} = \{106\}$

这句是多余的，在优化过程中可被删除

# 主要内容

---

- 基于语法制导的翻译方案的中间表示生成
- **非三地址代码形式的中间表示的生成**
- 静态单赋值形式(进阶版)
- Multi-Level IR (MLIR)

# 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

# 回顾：静态单赋值形式 (SSA)

- ◎ 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  $\varphi$  函数来合并不同路径中的定值

- ❖ **if** ( flag )  $x_1 = -1$ ; **else**  $x_2 = 1$ ;  $x_3 = \varphi(x_1, x_2)$ ;  $y = x_3 * a$ ;

- ❖ LLVM IR 采用这种设计

- ◎ 使用块参数来传递不同路径中的定值

```
if (flag) { $x_1 = -1$; goto L(x_1); }
else { $x_2 = 1$; goto L(x_2); }
L(z): $y = z * a$;
```

- ❖ Koopa IR 采用这种设计

# 回顾：静态单赋值形式 (SSA)

- 虽然名字的定值只有一处，但是其值的计算可能不止一次

## 源程序

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

## SSA 形式( $\varphi$ 函数)

```
x0 = ...
y0 = ...
if x0 >= 100 goto next
 else goto loop
loop:
 x1 = $\varphi(x_0, x_2)$
 y1 = $\varphi(y_0, y_2)$
 x2 = x1 + 1
 y2 = y1 + x2
 if x2 < 100 goto loop
 else goto next
next:
 x3 = $\varphi(x_0, x_2)$
 y3 = $\varphi(y_0, y_2)$
```

## SSA 形式(块参数)

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

# SSA 形式 IR 的生成

- ◎ 生成高质量的 SSA 代码不是那么容易的
  - ❖ 后面会再进行讨论
- ◎ **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                                                                                                                                                                       |
|-------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $S \rightarrow T ID;$   | { <code>insert</code> ( <code>ID.lexeme</code> , <code>T.type</code> );<br><code>emit</code> (`{ <code>genvar</code> ( <code>ID.lexeme</code> )} = <code>alloc</code> ); } |
| $S \rightarrow ID = E;$ | { <code>emit</code> (` <code>store</code> { <code>E.addr</code> } { <code>genvar</code> ( <code>ID.lexeme</code> )}`); }                                                   |
| $E \rightarrow ID$      | { <code>E.addr</code> = <code>gentmp</code> ();<br><code>emit</code> (`{ <code>E.addr</code> } = <code>load</code> { <code>genvar</code> ( <code>ID.lexeme</code> )}`); }  |

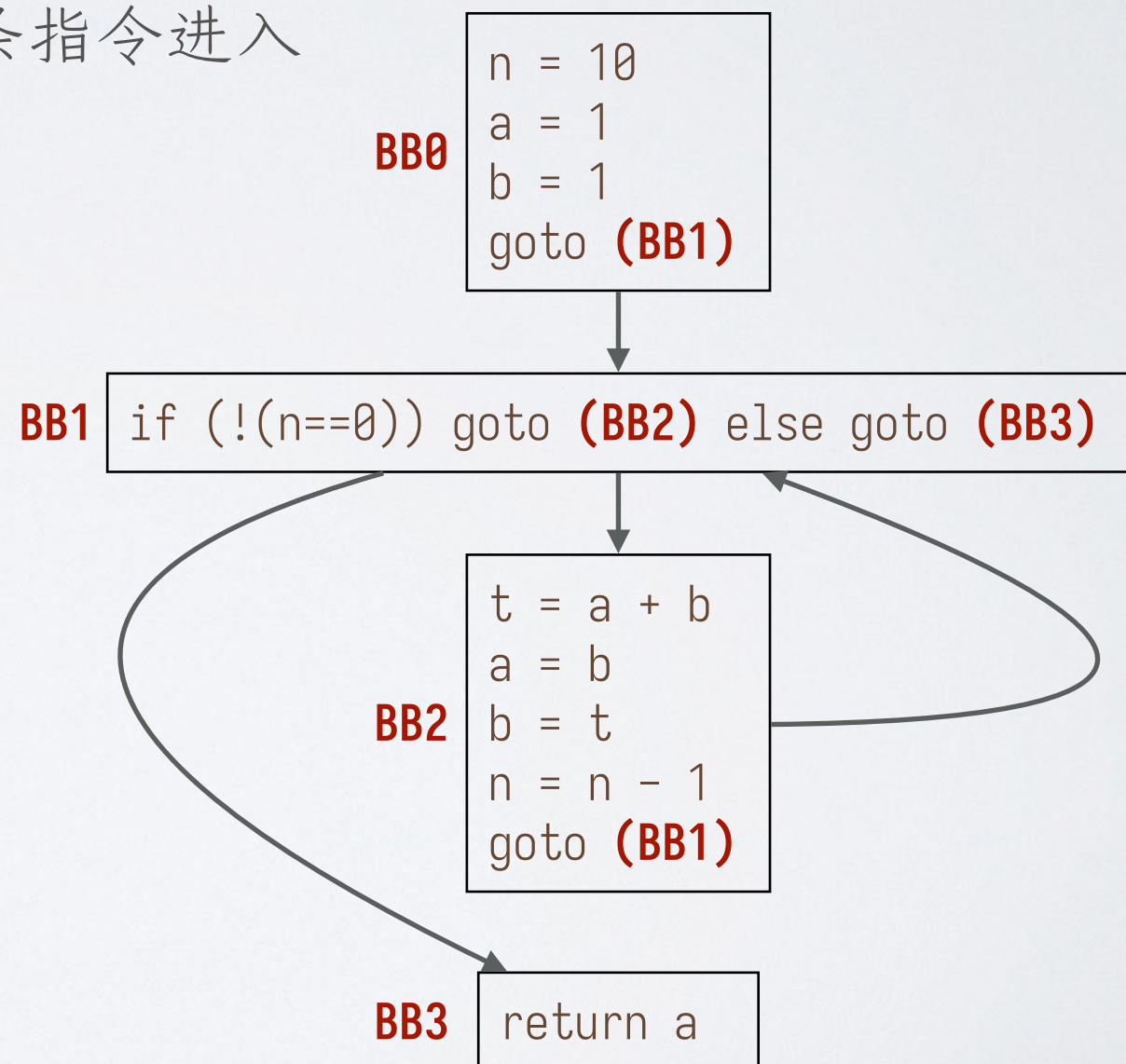
- 可以进行的**优化**:
  - ❖ 在符号表中记录每个程序变量的**当前值**的地址
  - ❖ 在上一步的基础上, 尽可能推迟往内存写回的操作

# 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;
}
```



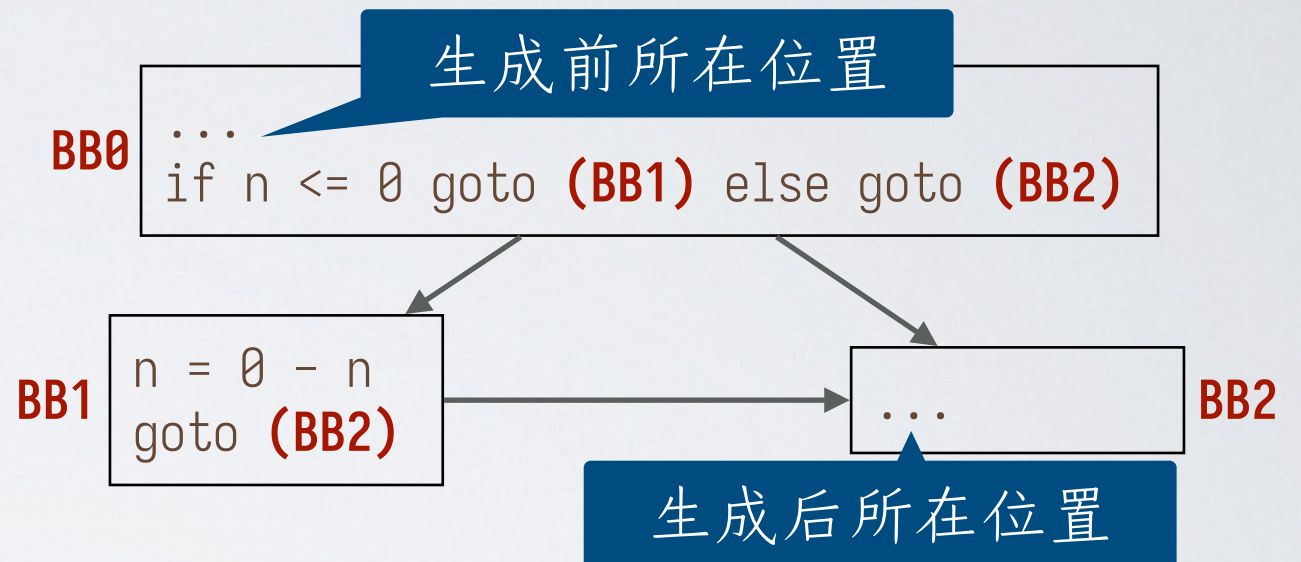
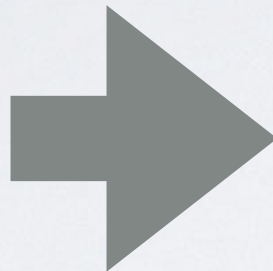
# 控制流图形式 IR 的生成

- ◎ 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**

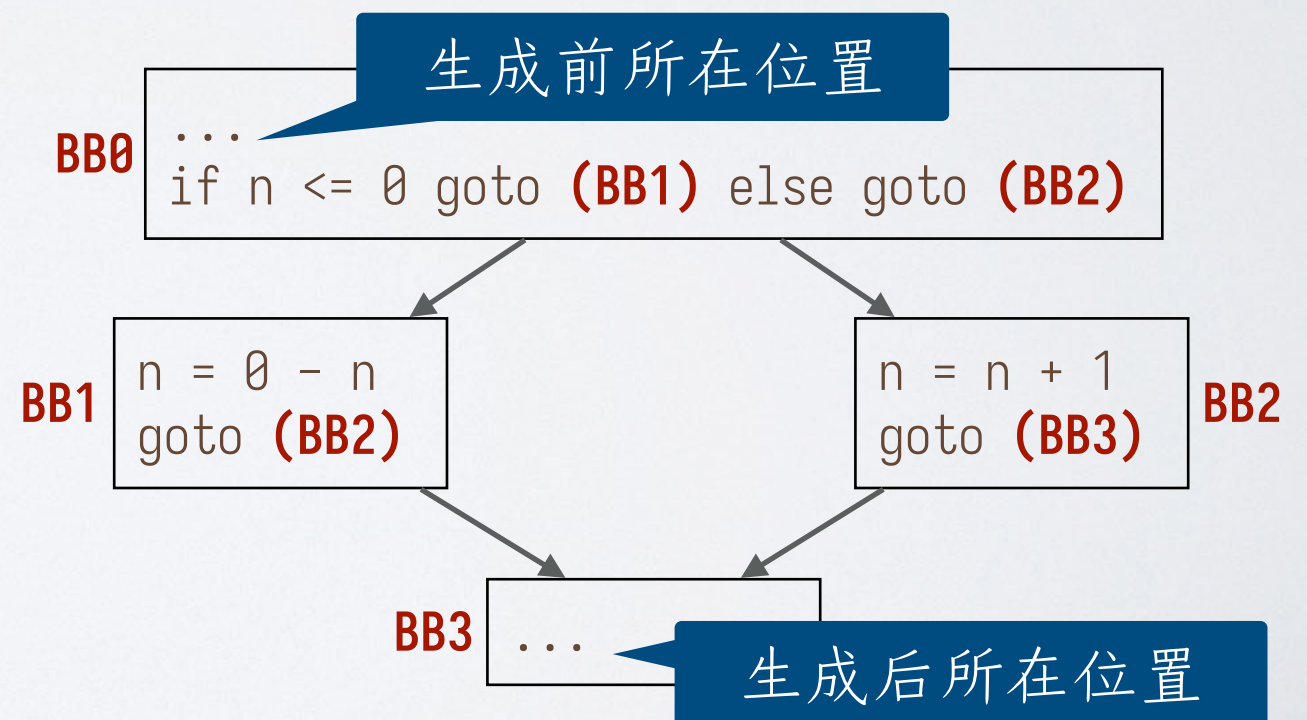
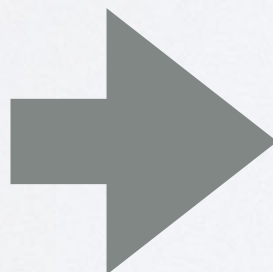
# 生成控制流图形式 IR 的 SDT (条件)

$$S ::= \dots \mid \text{if}(B) S \mid \text{if}(B) S \text{ else } S$$

```
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                                                                                                                                                                                                                                                                                                   |
|--------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $S \rightarrow \text{if} ($<br>$B)$<br>$S_1$                       | $\{ B.true = \text{new\_bb}(); B.false = S_1.next = \text{new\_bb}(); \}$<br>$\{ \text{set\_bb}(B.true); \}$<br>$\{ \text{emit}(\text{'goto } \{S_1.next\}'); \text{set\_bb}(S_1.next); \}$                                                                                                            |
| $S \rightarrow \text{if} ($<br>$B)$<br>$S_1 \text{ else}$<br>$S_2$ | $\{ B.true = \text{new\_bb}(); B.false = \text{new\_bb}(); S_1.next = S_2.next = \text{new\_bb}(); \}$<br>$\{ \text{set\_bb}(B.true); \}$<br>$\{ \text{emit}(\text{'goto } \{S_1.next\}'); \text{set\_bb}(B.false); \}$<br>$\{ \text{emit}(\text{'goto } \{S_2.next\}'); \text{set\_bb}(S_2.next); \}$ |

## 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) ...
```

BB0

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

生成前所在位置

```
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                                                                                       |
|------------------------------|--------------------------------------------------------------------------------------------|
| $B \rightarrow \text{true}$  | { emit(`goto { $B$ .true}`); }                                                             |
| $B \rightarrow \text{false}$ | { emit(`goto { $B$ .false}`); }                                                            |
| $B \rightarrow ($<br>$B_1)$  | { $B_1$ .true = $B$ .true; $B_1$ .false = $B$ .false; }                                    |
| $B \rightarrow !$<br>$B_1$   | { $B_1$ .true = $B$ .false; $B_1$ .false = $B$ .true; }                                    |
| $B \rightarrow E_1 == E_2$   | { emit(`if { $E_1$ .addr} == { $E_2$ .addr} goto { $B$ .true} else goto { $B$ .false}`); } |
| $B \rightarrow E_1 <= E_2$   | { 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                                                                                                                                                                                                |
|------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $B \rightarrow$<br>$B_1 \ \&\&$<br>$B_2$ | $\{ B_1.\text{true} = \text{new\_bb}(); B_1.\text{false} = B.\text{false}; \}$<br>$\{ B_2.\text{true} = B.\text{true}; B_2.\text{false} = B.\text{false};$<br>$\text{set\_bb}(B_1.\text{true}); \}$ |
| $B \rightarrow$<br>$B_1 \   $<br>$B_2$   | $\{ B_1.\text{true} = B.\text{true}; B_1.\text{false} = \text{new\_bb}(); \}$<br>$\{ B_2.\text{true} = B.\text{true}; B_2.\text{false} = B.\text{false};$<br>$\text{set\_bb}(B_1.\text{false}); \}$ |

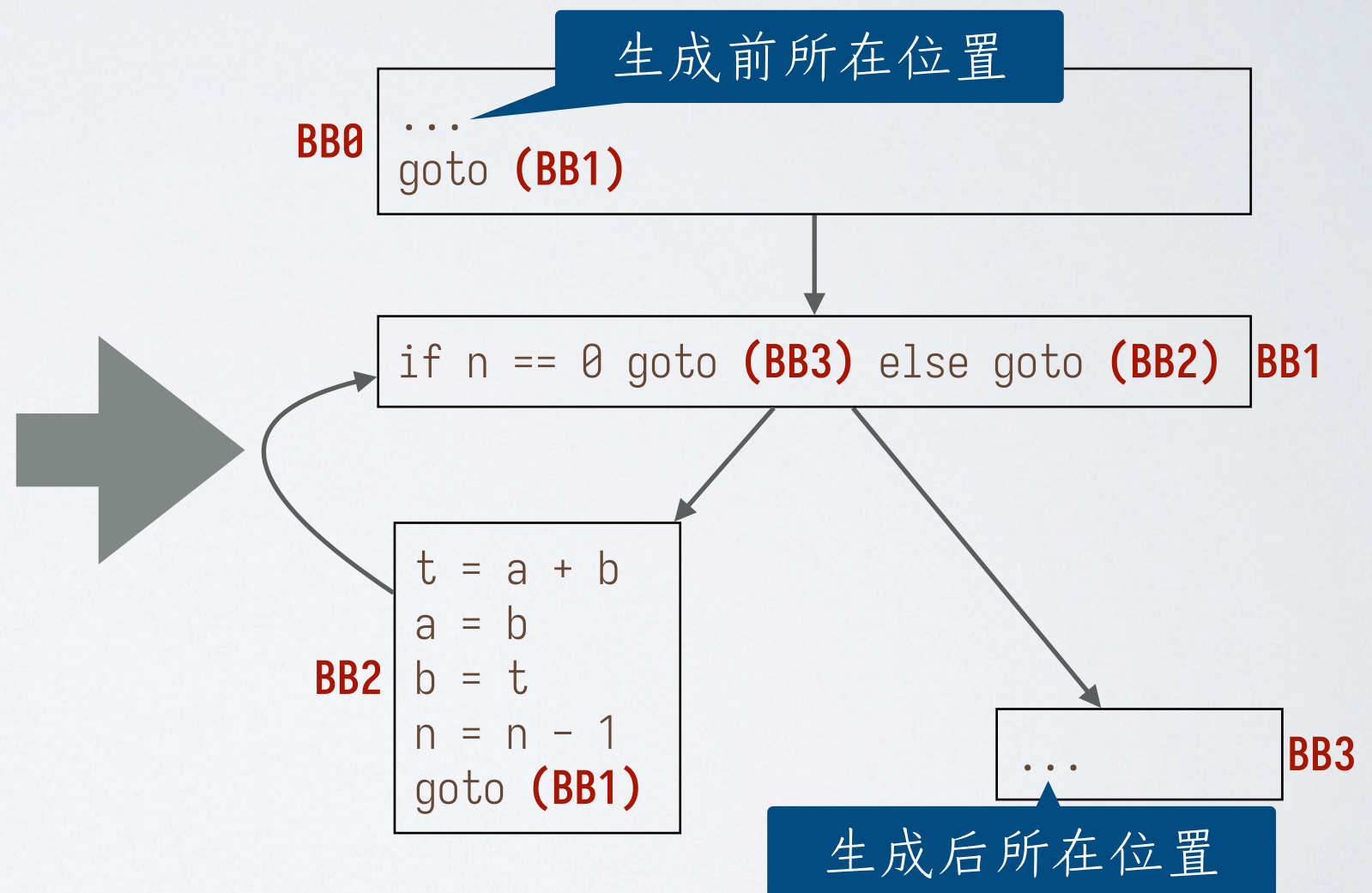
## 继承属性:

- $B.\text{true}$ :  $B$  为真时跳转到的基本块
- $B.\text{false}$ :  $B$  为假时跳转到的基本块

# 生成控制流图形式 IR 的 SDT (循环)

$$S ::= \dots \mid \text{while}(B) S$$

```
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                                                                                                                                                                                                         |
|----------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $S \rightarrow \text{while}(\text{B})$ | $\{ B.\text{true} = \text{new\_bb}(); B.\text{false} = \text{new\_bb}(); S_1.\text{next} = \text{new\_bb}();$<br>$\text{emit}(\text{'goto } \{S_1.\text{next}\}'); \text{set\_bb}(S_1.\text{next});$<br>$\}$ |
| $S_1$                                  | $\{ \text{set\_bb}(B.\text{true});$<br>$\text{emit}(\text{'goto } \{S_1.\text{next}\}'); \text{set\_bb}(B.\text{false}); \}$                                                                                 |

练习：如何支持  
break 和 continue?

继承属性：

- ◉  $B.\text{true}$ :  $B$  为真时跳转到的基本块
- ◉  $B.\text{false}$ :  $B$  为假时跳转到的基本块

# 主要内容

---

- 基于语法制导的翻译方案的中间表示生成
- 非三地址代码形式的中间表示的生成
- 静态单赋值形式(进阶版)
- Multi-Level IR (MLIR)

# SSA 形式 IR 的生成

◎ 前面讲的生成算法：为每个程序变量开辟内存空间

◎ 问题：生成代码的性能太差

对于单个基本块，可以使用前面讲过的基于 DAG 的优化方法进行 SSA 生成(也称为 local value numbering)

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

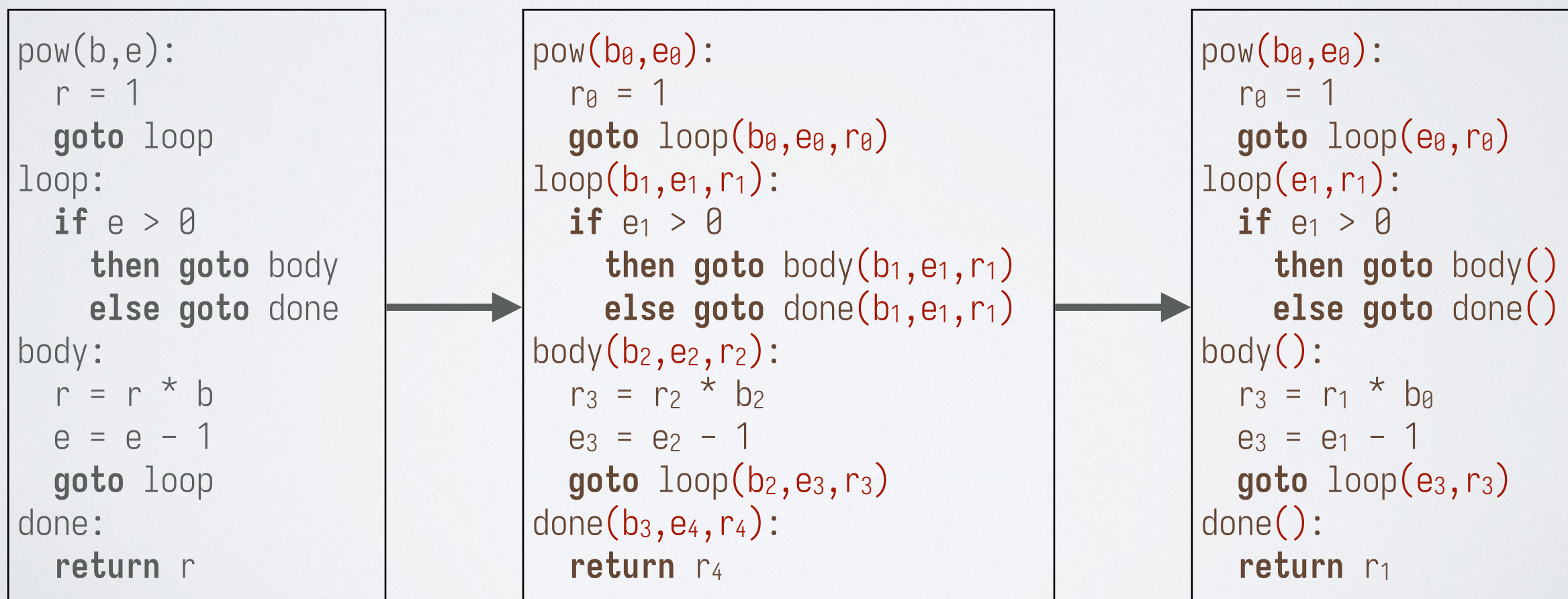
```
a = 1
b = 1
c = a + b
a = b
b = c
```

```
a1 = 1
b1 = 1
c1 = a1 + b1
a2 = b1
b2 = c1
```

如何处理多个基本块？  
(也称为 global value numbering)

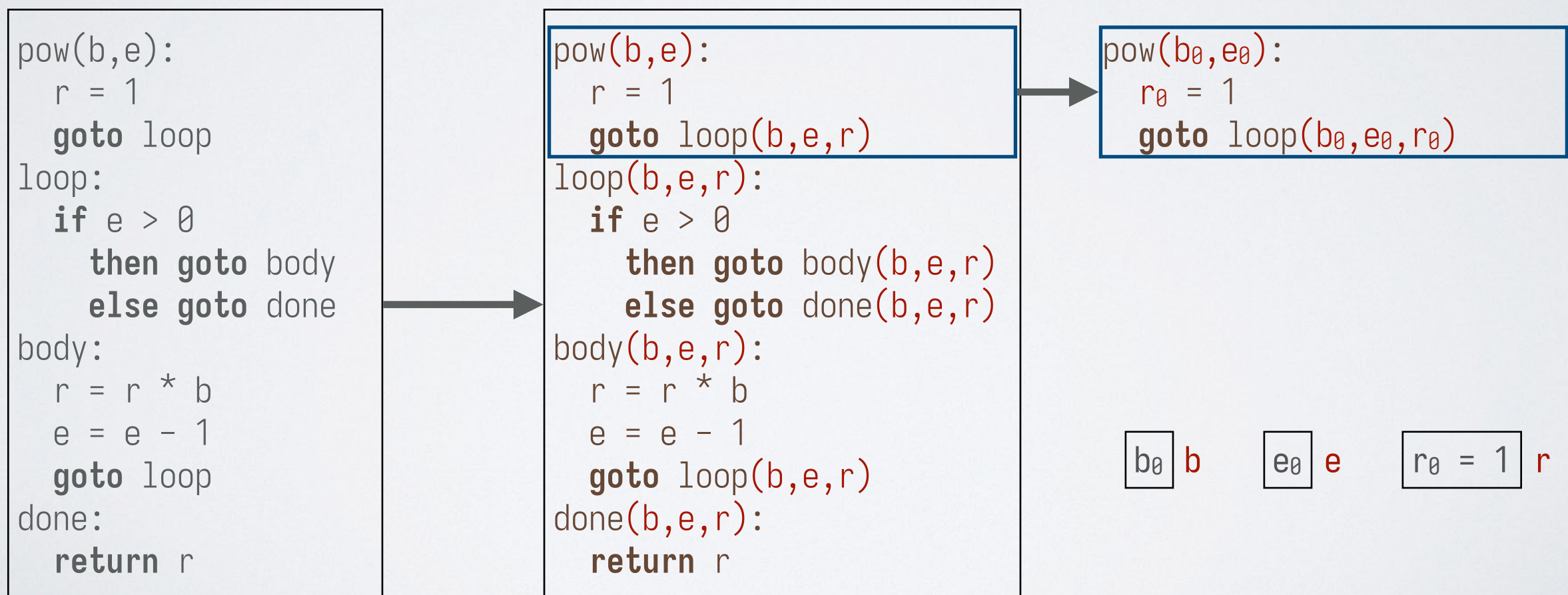
# 一种两趟生成方法

- 以块参数 SSA 形式为例 (即 Koopa IR 的形式)
- 第一趟: 每个基本块以所有 (块起始处的活跃) 变量为参数
- 第二趟: 尝试消除冗余的块参数



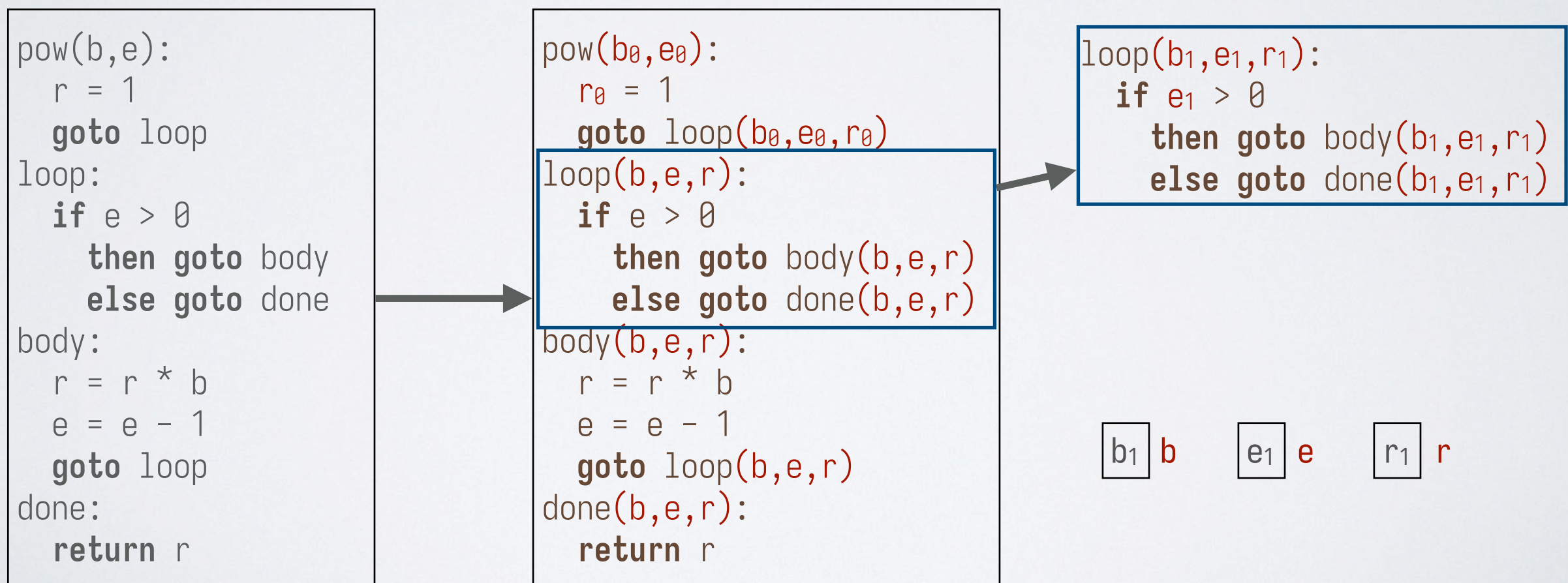
# 第一趟：粗粒度生成

- 为每个基本块添加块参数，原则上只需要添加活跃变量
  - ❖ 简单起见，可以添加块起始处所有的可用变量
- 对每个基本块，分别进行 SSA 转换(即 local value numbering)



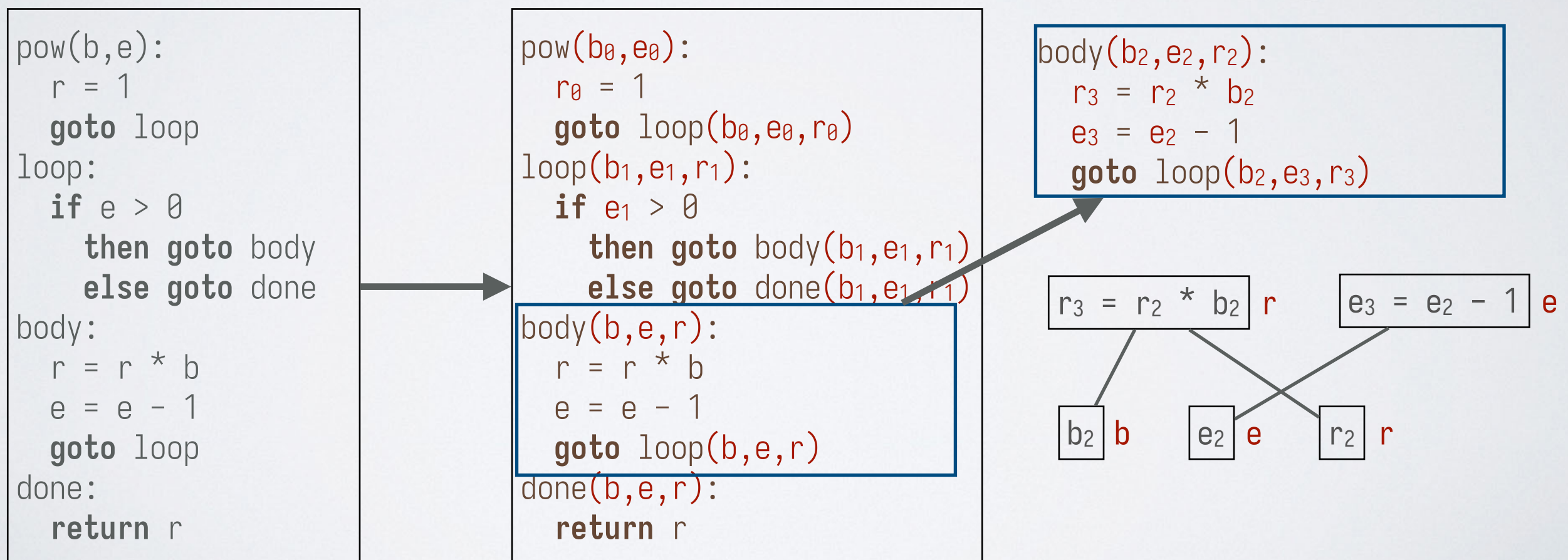
# 第一趟：粗粒度生成

- 为每个基本块添加块参数，原则上只需要添加活跃变量
  - ❖ 简单起见，可以添加块起始处所有的可用变量
- 对每个基本块，分别进行 SSA 转换(即 local value numbering)



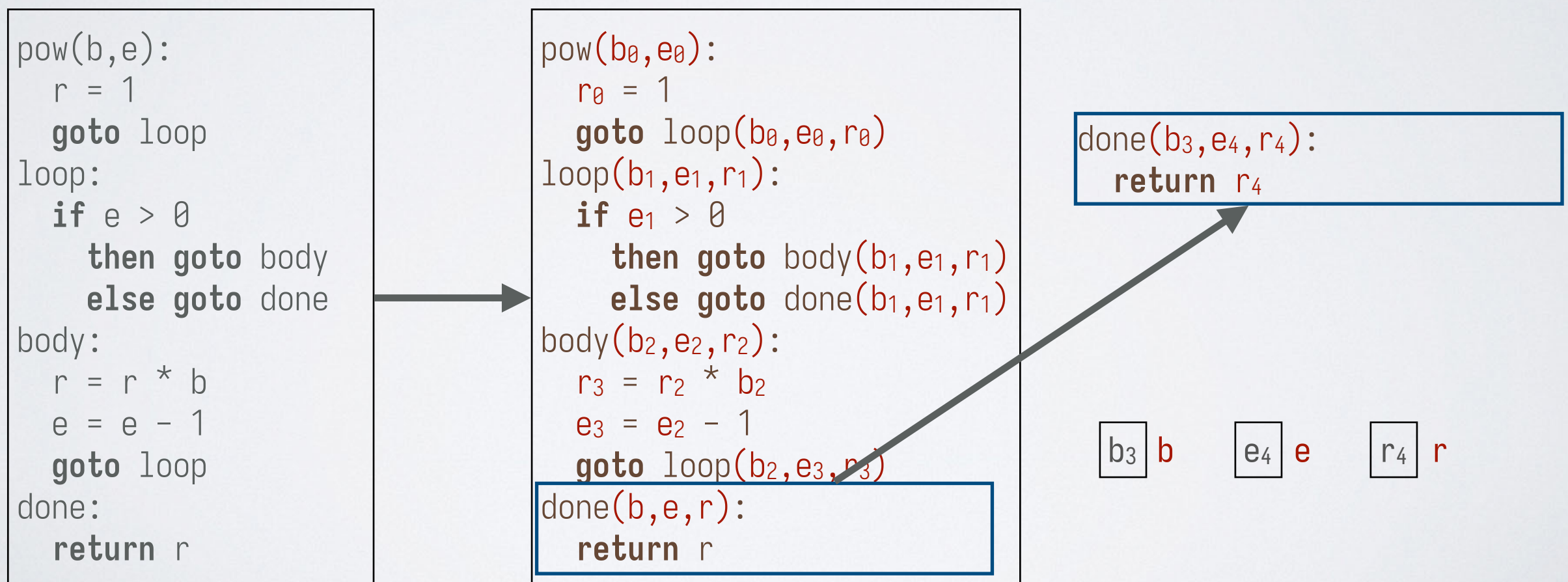
# 第一趟：粗粒度生成

- 为每个基本块添加块参数，原则上只需要添加活跃变量
  - ❖ 简单起见，可以添加块起始处所有的可用变量
- 对每个基本块，分别进行 SSA 转换(即 local value numbering)



# 第一趟：粗粒度生成

- 为每个基本块添加块参数，原则上只需要添加活跃变量
  - ❖ 简单起见，可以添加块起始处所有的可用变量
- 对每个基本块，分别进行 SSA 转换(即 local value numbering)



# 第二趟：细粒度优化

◎ 对于带参数基本块  $l(\dots, x_i, \dots)$ ：而言，如果存在  $k$  满足任意的跳转到  $l$  的指令都是下列两种形式之一：

❖  $\text{goto } l(\dots, x_i, \dots)$  或者  $\text{goto } l(\dots, x_k, \dots)$

则可以移除参数  $x_i$ ，并把所有  $x_i$  替换为  $x_k$

```
pow(b0, e0):
 r0 = 1
 goto loop(b0, e0, r0)
loop(b1, e1, r1):
 if e1 > 0
 then goto body(b1, e1, r1)
 else goto done(b1, e1, r1)
body(b2, e2, r2):
 r3 = r2 * b2
 e3 = e2 - 1
 goto loop(b2, e3, r3)
done(b3, e4, r4):
 return r4
```

```
pow(b0, e0):
 r0 = 1
 goto loop(b0, e0, r0)
loop(b1, e1, r1):
 if e1 > 0
 then goto body()
 else goto done()
body():
 r3 = r1 * b1
 e3 = e1 - 1
 goto loop(b1, e3, r3)
done():
 return r1
```

```
pow(b0, e0):
 r0 = 1
 goto loop(e0, r0)
loop(e1, r1):
 if e1 > 0
 then goto body()
 else goto done()
body():
 r3 = r1 * b0
 e3 = e1 - 1
 goto loop(e3, r3)
done():
 return r1
```

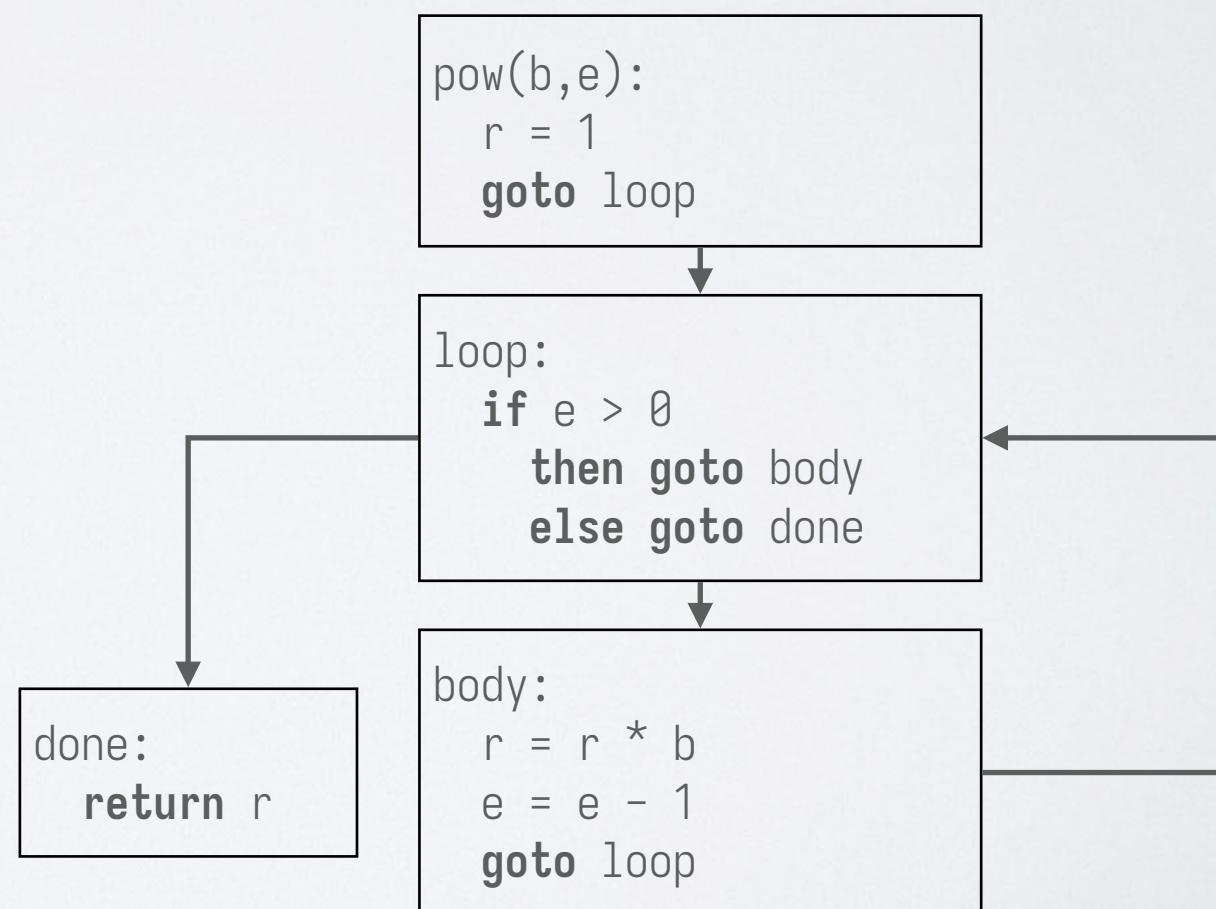
# 一种一趟生成方法

- 在每个基本块内通过 local value numbering 转换为 SSA 形式
- 如果有一个变量  $x$  的使用不在基本块内, 考虑前驱基本块:
  - ❖ 如果只有一个前驱, 则递归处理该前驱基本块
  - ❖ 如果有多个前驱, 则添加  $x$  为基本块参数, 并递归处理前驱基本块并传递实际的块参数

```

pow(b,e):
 r = 1
 goto loop
loop:
 if e > 0
 then goto body
 else goto done
body:
 r = r * b
 e = e - 1
 goto loop
done:
 return r

```



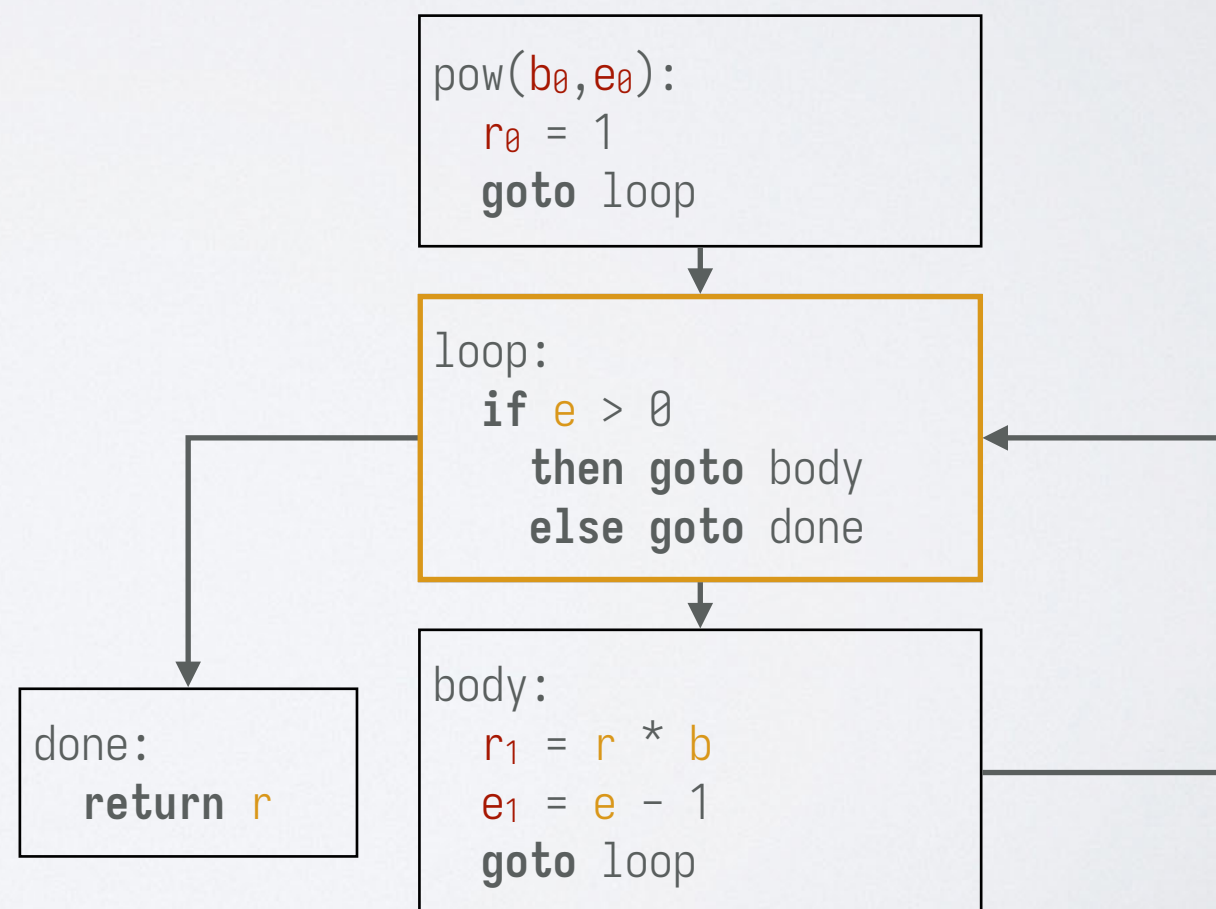
# 一种一趟生成方法

- 在每个基本块内通过 local value numbering 转换为 SSA 形式
- 如果有一个变量  $x$  的使用不在基本块内, 考虑前驱基本块:
  - ❖ 如果只有一个前驱, 则递归处理该前驱基本块
  - ❖ 如果有多个前驱, 则添加  $x$  为基本块参数, 并递归处理前驱基本块并传递实际的块参数

```

pow(b,e):
 r = 1
 goto loop
loop:
 if e > 0
 then goto body
 else goto done
body:
 r = r * b
 e = e - 1
 goto loop
done:
 return r

```



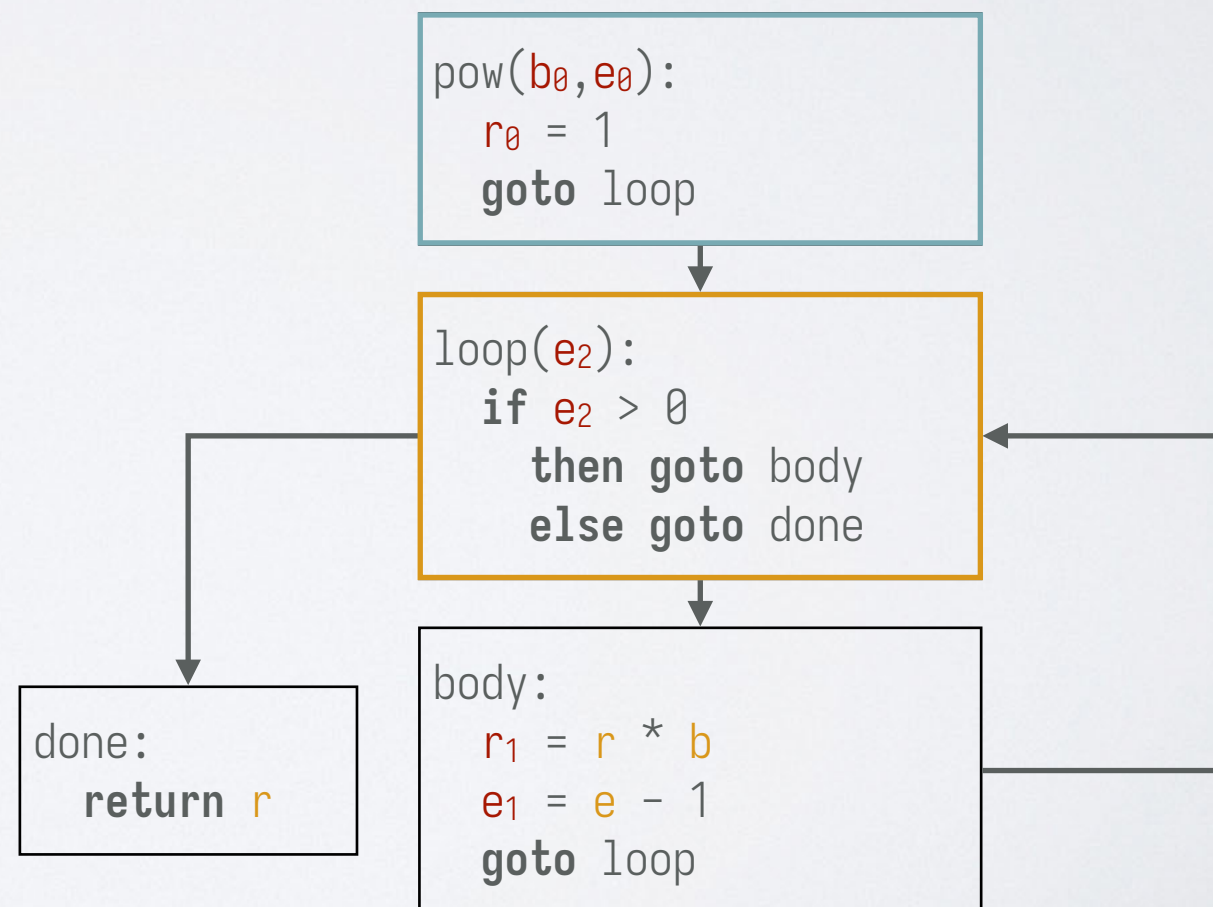
# 一种一趟生成方法

- 在每个基本块内通过 local value numbering 转换为 SSA 形式
- 如果有一个变量  $x$  的使用不在基本块内, 考虑前驱基本块:
  - ❖ 如果只有一个前驱, 则递归处理该前驱基本块
  - ❖ 如果有多个前驱, 则添加  $x$  为基本块参数, 并递归处理前驱基本块并传递实际的块参数

```

pow(b,e):
 r = 1
 goto loop
loop:
 if e > 0
 then goto body
 else goto done
body:
 r = r * b
 e = e - 1
 goto loop
done:
 return r

```



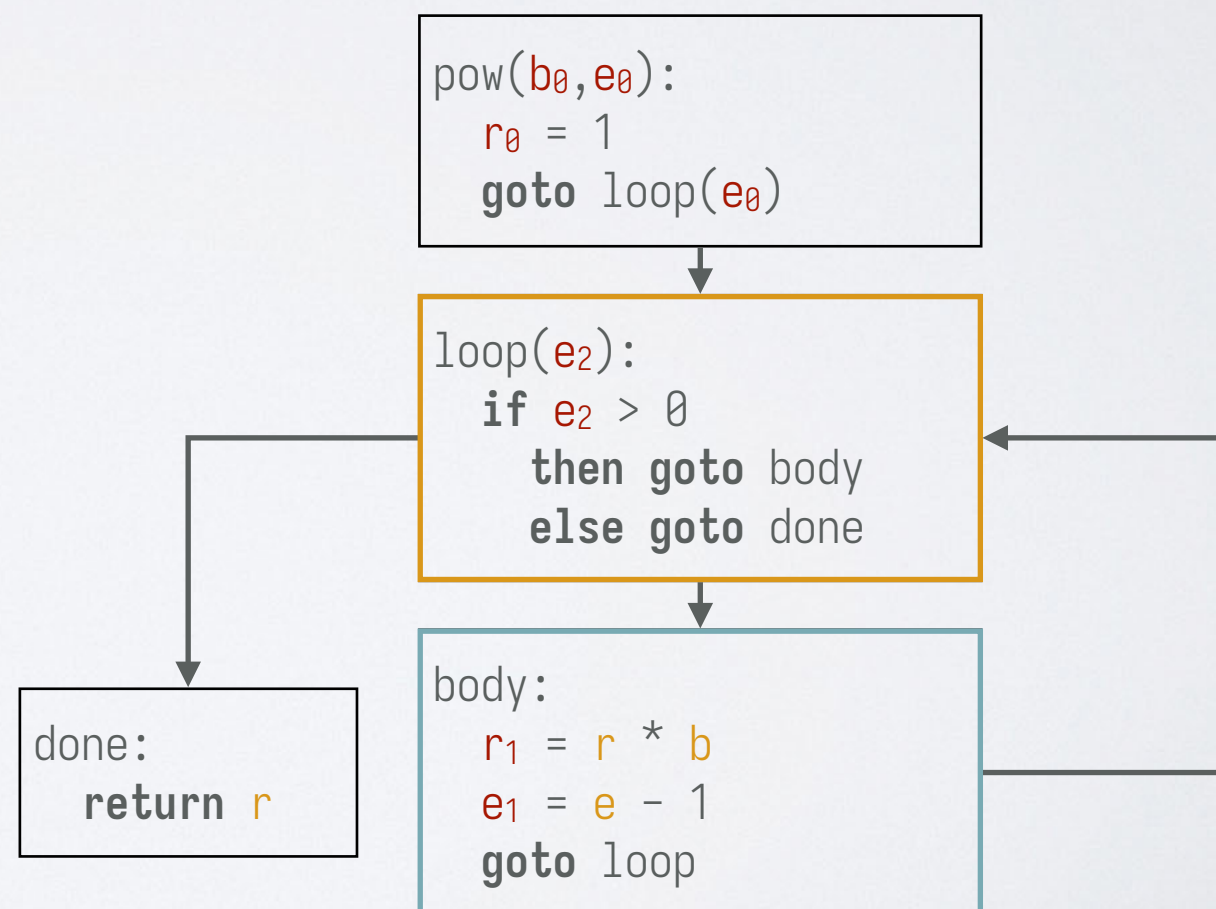
# 一种一趟生成方法

- 在每个基本块内通过 local value numbering 转换为 SSA 形式
- 如果有一个变量  $x$  的使用不在基本块内, 考虑前驱基本块:
  - ❖ 如果只有一个前驱, 则递归处理该前驱基本块
  - ❖ 如果有多个前驱, 则添加  $x$  为基本块参数, 并递归处理前驱基本块并传递实际的块参数

```

pow(b,e):
 r = 1
 goto loop
loop:
 if e > 0
 then goto body
 else goto done
body:
 r = r * b
 e = e - 1
 goto loop
done:
 return r

```



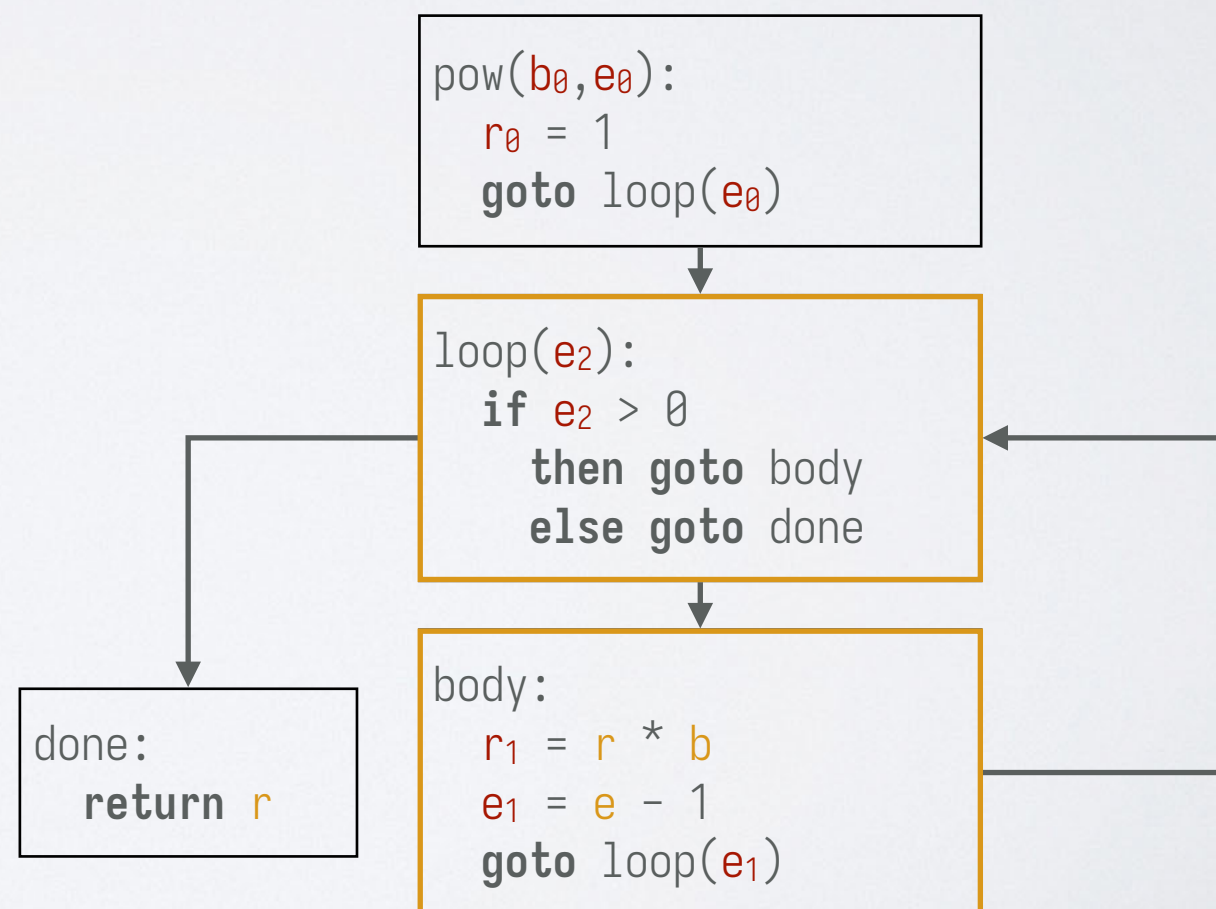
# 一种一趟生成方法

- 在每个基本块内通过 local value numbering 转换为 SSA 形式
- 如果有一个变量  $x$  的使用不在基本块内, 考虑前驱基本块:
  - ❖ 如果只有一个前驱, 则递归处理该前驱基本块
  - ❖ 如果有多个前驱, 则添加  $x$  为基本块参数, 并递归处理前驱基本块并传递实际的块参数

```

pow(b,e):
 r = 1
 goto loop
loop:
 if e > 0
 then goto body
 else goto done
body:
 r = r * b
 e = e - 1
 goto loop
done:
 return r

```



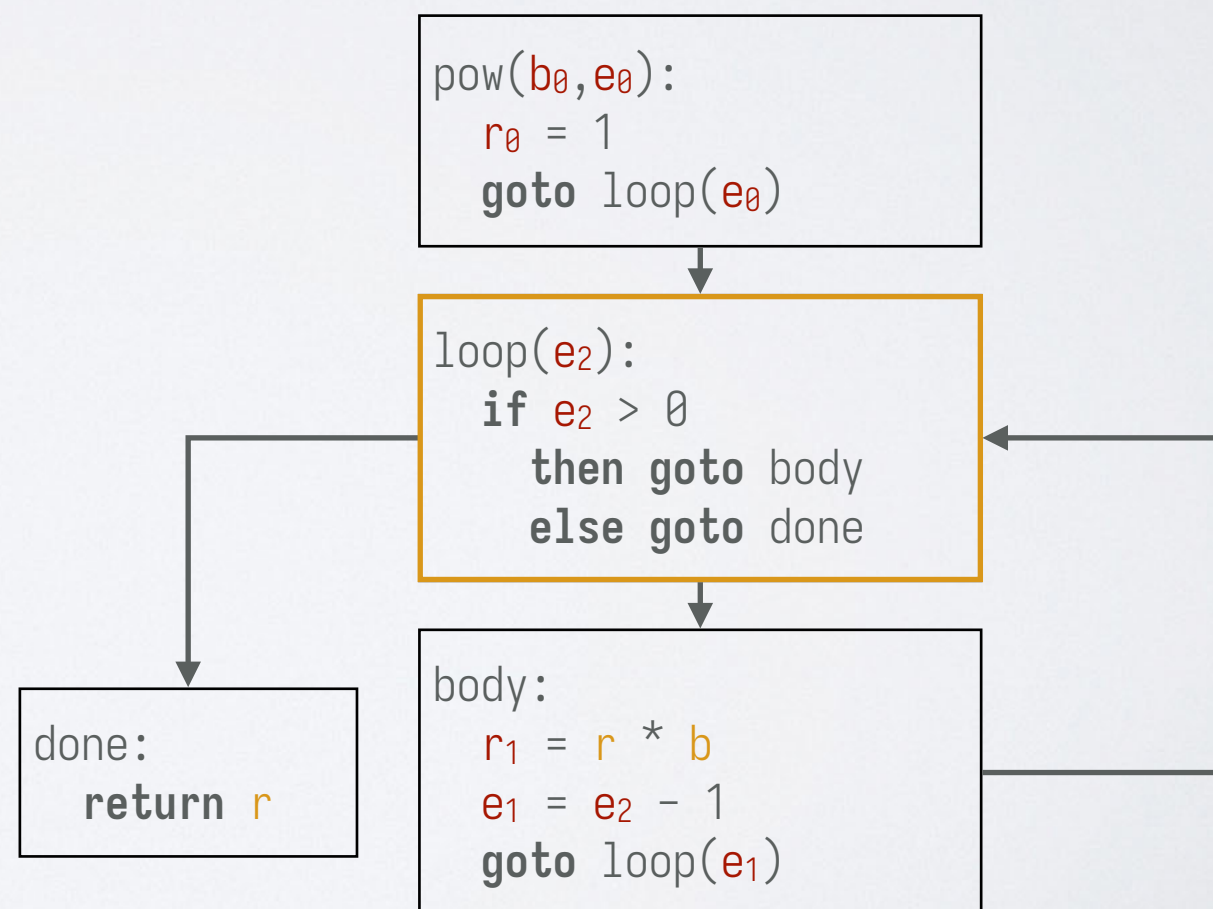
# 一种一趟生成方法

- 在每个基本块内通过 local value numbering 转换为 SSA 形式
- 如果有一个变量  $x$  的使用不在基本块内, 考虑前驱基本块:
  - ❖ 如果只有一个前驱, 则递归处理该前驱基本块
  - ❖ 如果有多个前驱, 则添加  $x$  为基本块参数, 并递归处理前驱基本块并传递实际的块参数

```

pow(b,e):
 r = 1
 goto loop
loop:
 if e > 0
 then goto body
 else goto done
body:
 r = r * b
 e = e - 1
 goto loop
done:
 return r

```



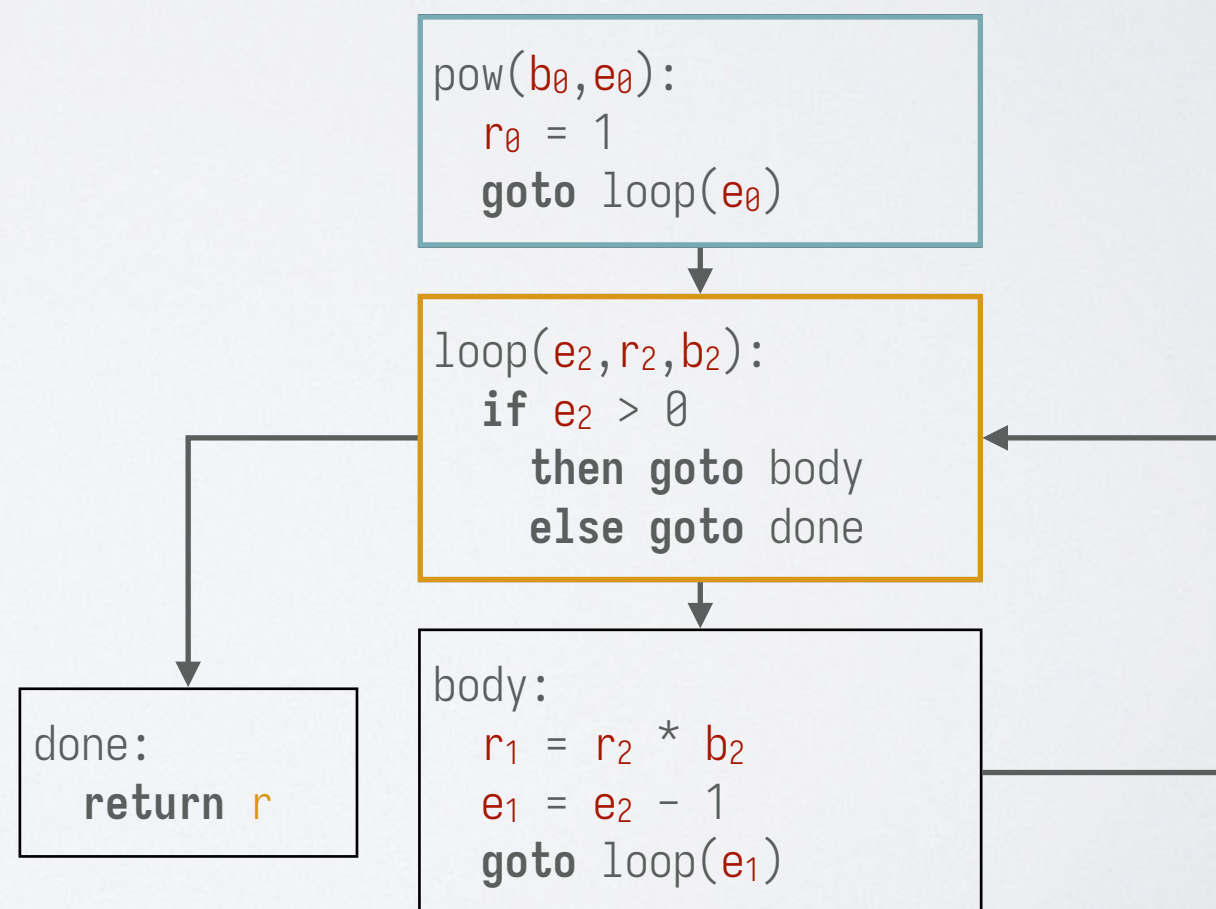
# 一种一趟生成方法

- 在每个基本块内通过 local value numbering 转换为 SSA 形式
- 如果有一个变量  $x$  的使用不在基本块内, 考虑前驱基本块:
  - ❖ 如果只有一个前驱, 则递归处理该前驱基本块
  - ❖ 如果有多个前驱, 则添加  $x$  为基本块参数, 并递归处理前驱基本块并传递实际的块参数

```

pow(b,e):
 r = 1
 goto loop
loop:
 if e > 0
 then goto body
 else goto done
body:
 r = r * b
 e = e - 1
 goto loop
done:
 return r

```



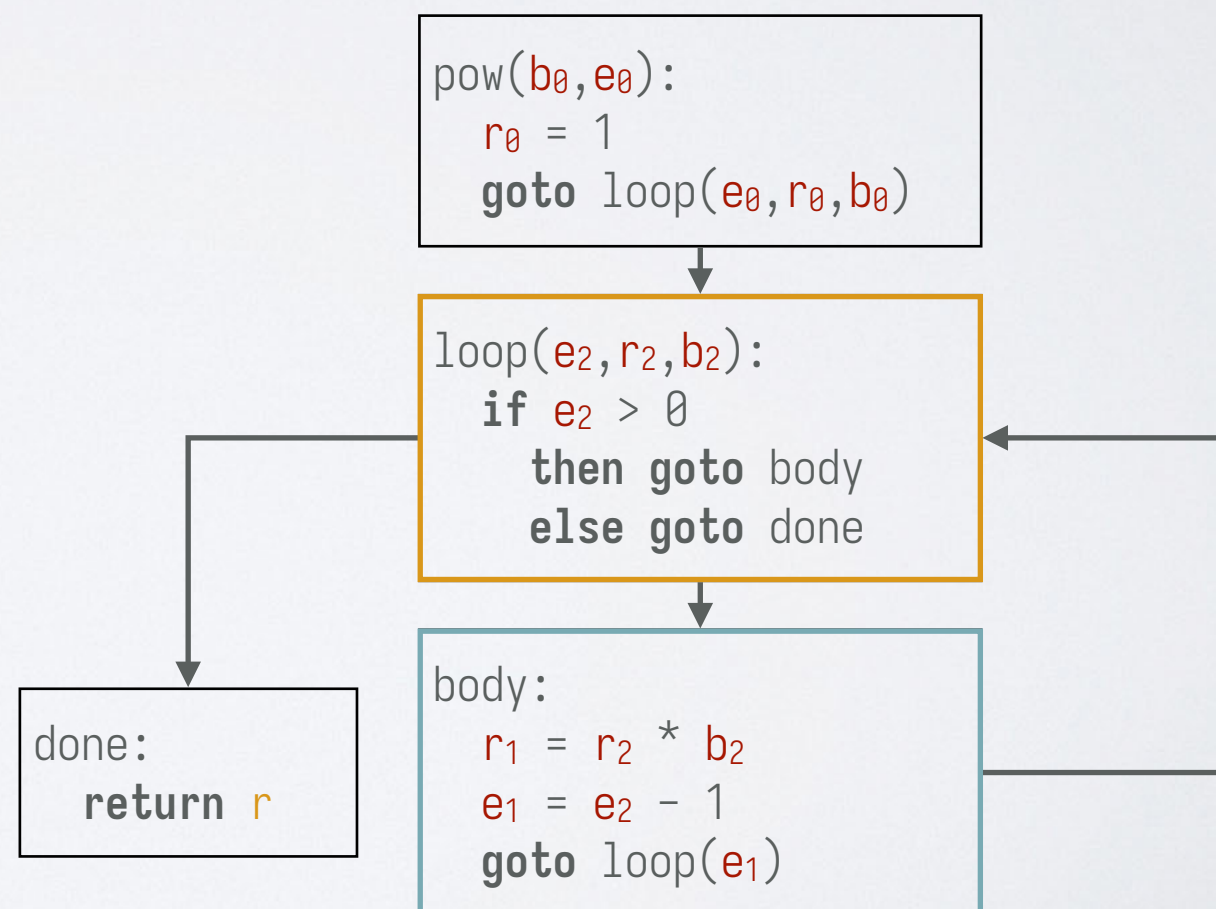
# 一种一趟生成方法

- 在每个基本块内通过 local value numbering 转换为 SSA 形式
- 如果有一个变量  $x$  的使用不在基本块内, 考虑前驱基本块:
  - ❖ 如果只有一个前驱, 则递归处理该前驱基本块
  - ❖ 如果有多个前驱, 则添加  $x$  为基本块参数, 并递归处理前驱基本块并传递实际的块参数

```

pow(b,e):
 r = 1
 goto loop
loop:
 if e > 0
 then goto body
 else goto done
body:
 r = r * b
 e = e - 1
 goto loop
done:
 return r

```



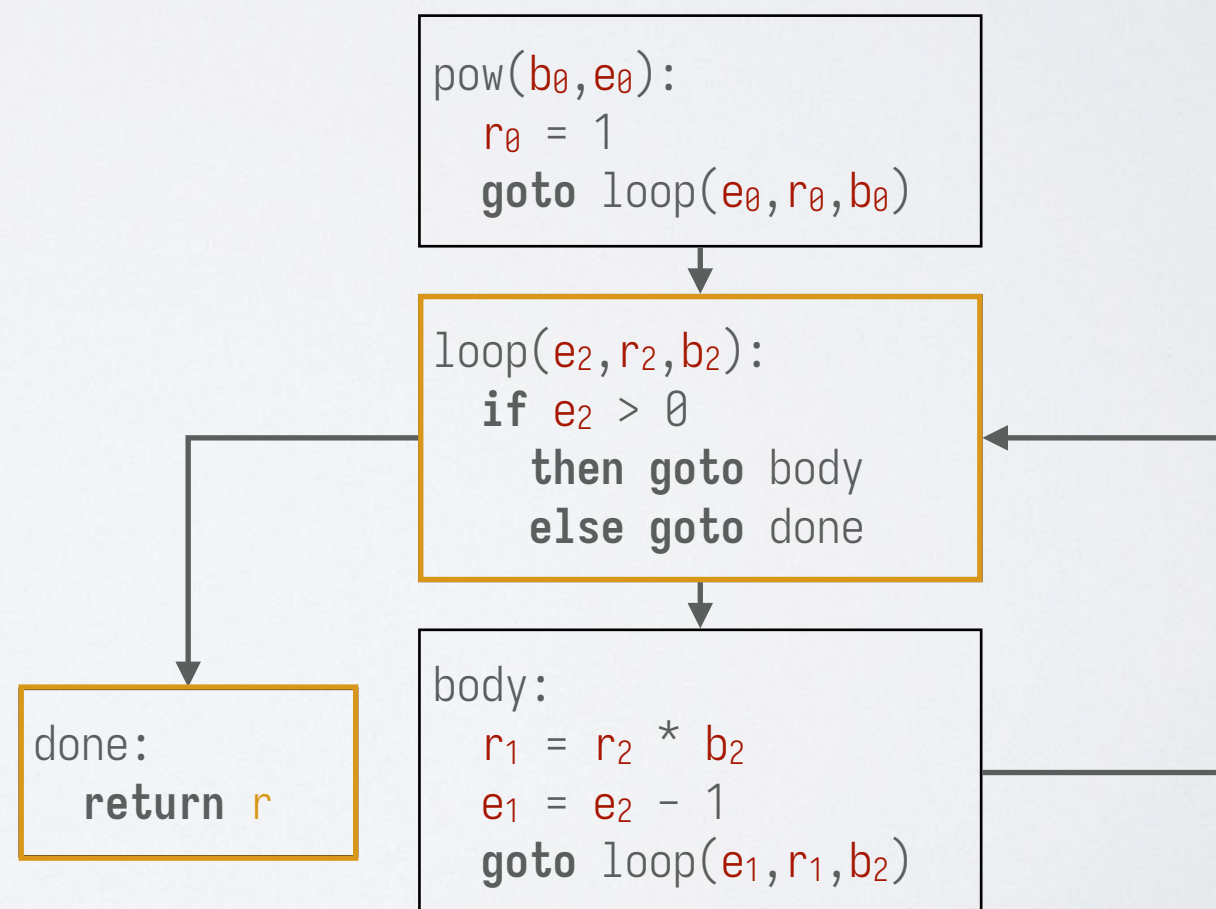
# 一种一趟生成方法

- 在每个基本块内通过 local value numbering 转换为 SSA 形式
- 如果有一个变量  $x$  的使用不在基本块内, 考虑前驱基本块:
  - ❖ 如果只有一个前驱, 则递归处理该前驱基本块
  - ❖ 如果有多个前驱, 则添加  $x$  为基本块参数, 并递归处理前驱基本块并传递实际的块参数

```

pow(b,e):
 r = 1
 goto loop
loop:
 if e > 0
 then goto body
 else goto done
body:
 r = r * b
 e = e - 1
 goto loop
done:
 return r

```



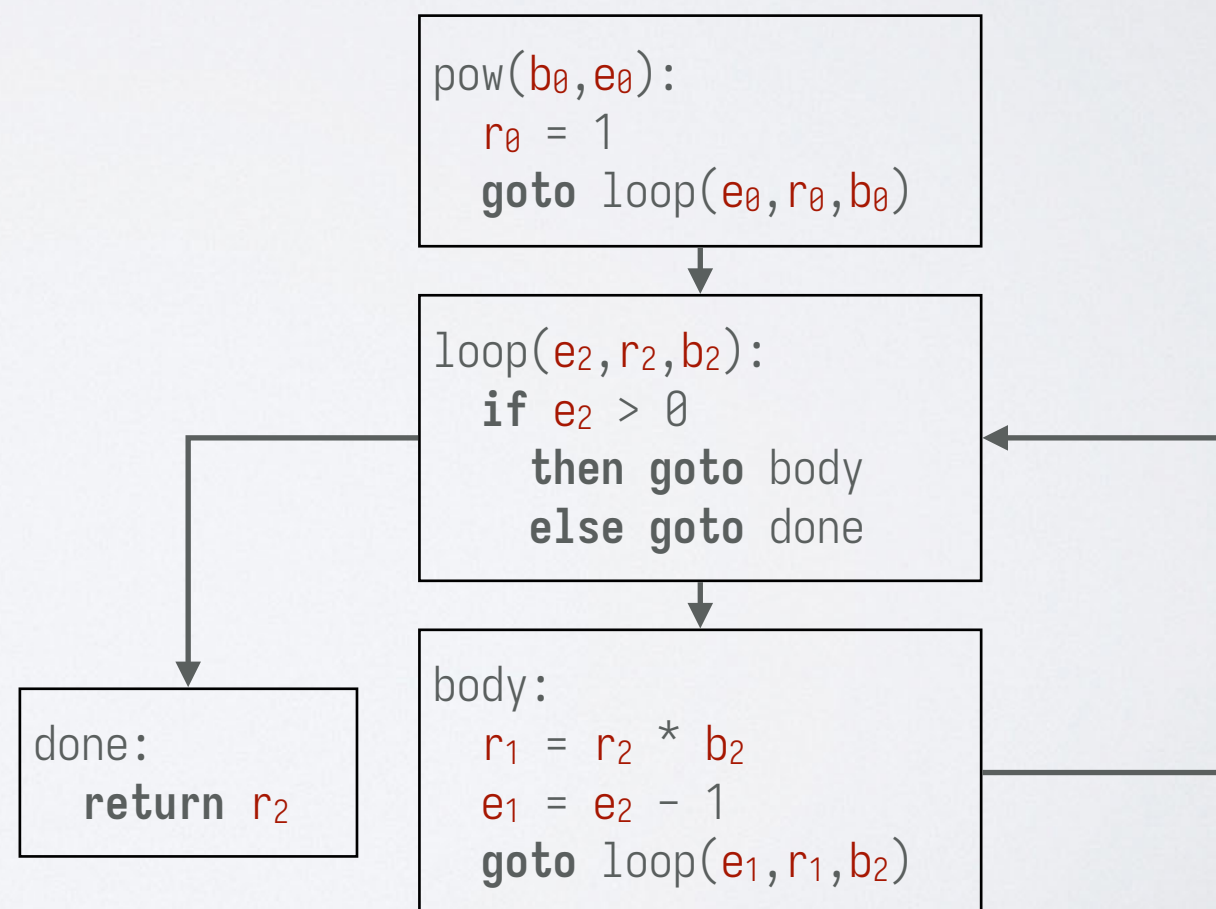
# 一种一趟生成方法

- 在每个基本块内通过 local value numbering 转换为 SSA 形式
- 如果有一个变量  $x$  的使用不在基本块内, 考虑前驱基本块:
  - ❖ 如果只有一个前驱, 则递归处理该前驱基本块
  - ❖ 如果有多个前驱, 则添加  $x$  为基本块参数, 并递归处理前驱基本块并传递实际的块参数

```

pow(b,e):
 r = 1
 goto loop
loop:
 if e > 0
 then goto body
 else goto done
body:
 r = r * b
 e = e - 1
 goto loop
done:
 return r

```



# 主要内容

---

- 基于语法制导的翻译方案的中间表示生成
- 非三地址代码形式的中间表示的生成
- 静态单赋值形式(进阶版)
- Multi-Level IR (MLIR)

# 参考文献

- M. Braun, S. Buchwald, S. Hack, R. Leißa, C. Mallon, and A. Zwinkau. 2013. Simple and Efficient Construction of Static Single Assignment Form. In Compiler Construction (CC '13).
- J. Aycock, and N. Horspool. 2000. Simple Generation of Static Single-Assignment Form. In Compiler Construction (CC '00).
- Mehdi Amini and River Riddle. MLIR: Multi-Level Intermediate Representation.