



第五讲 中间表示（基础版）

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

- ◎ 对应龙书章节：第 6 章

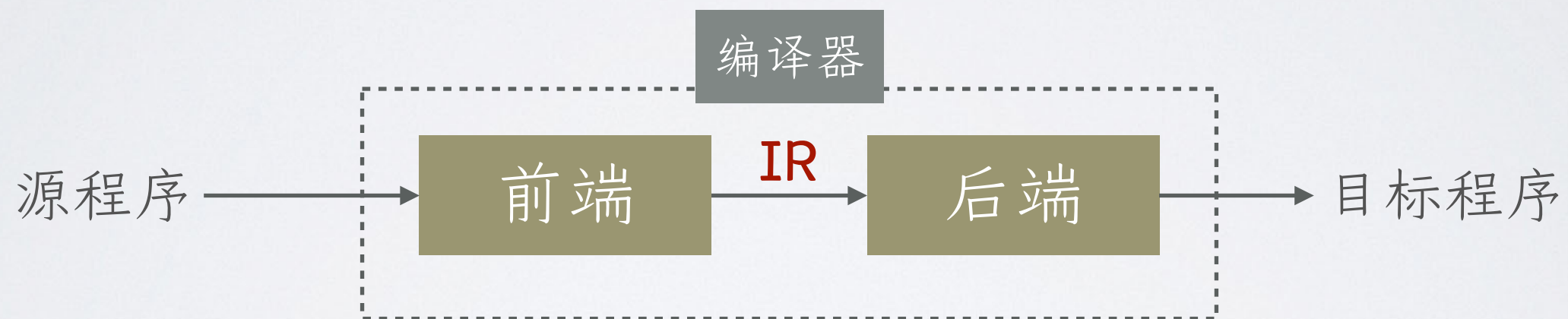
主要内容

- ◎ 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



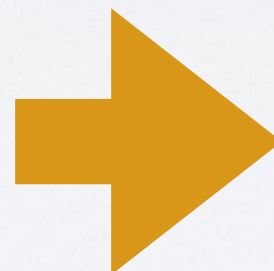
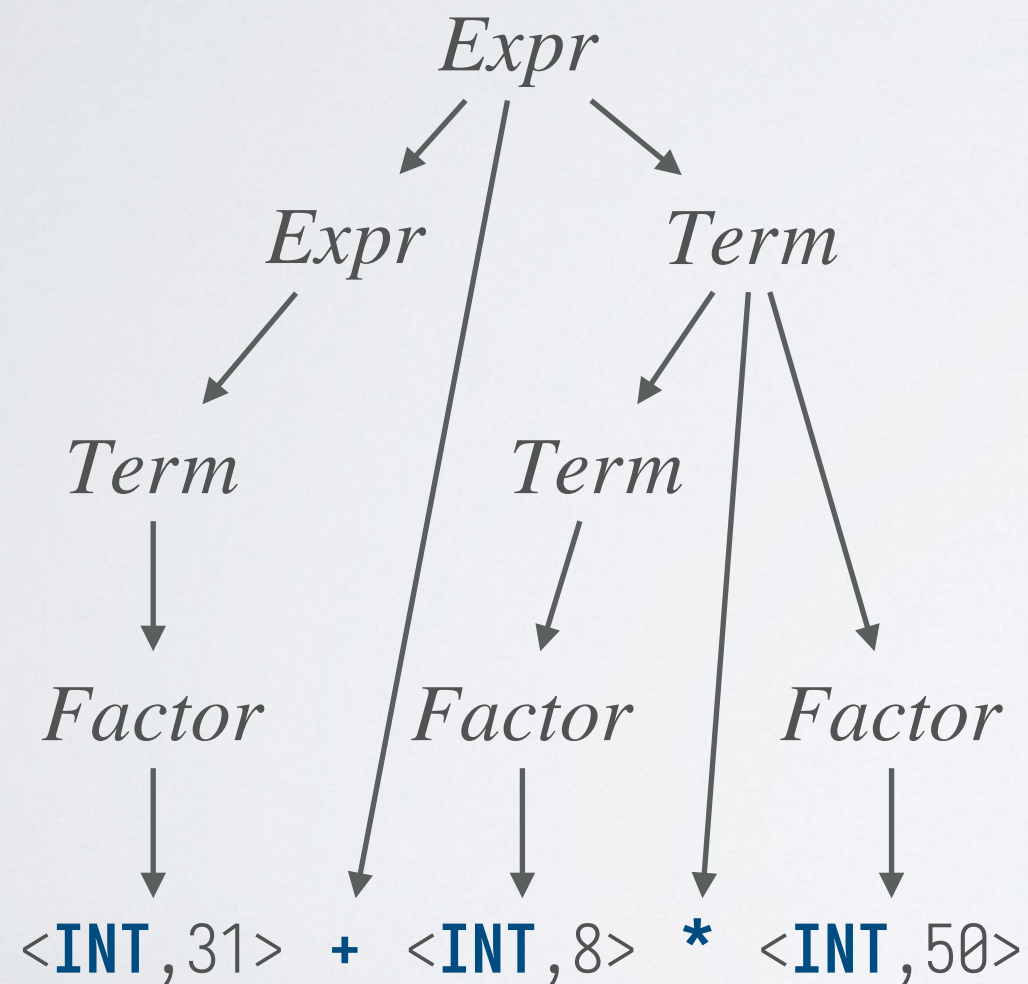
- 翻译过程要求编译器既要理解源程序,也要理解目标机器
- 两个不同的任务驱动了基于**前端**和**后端**的设计



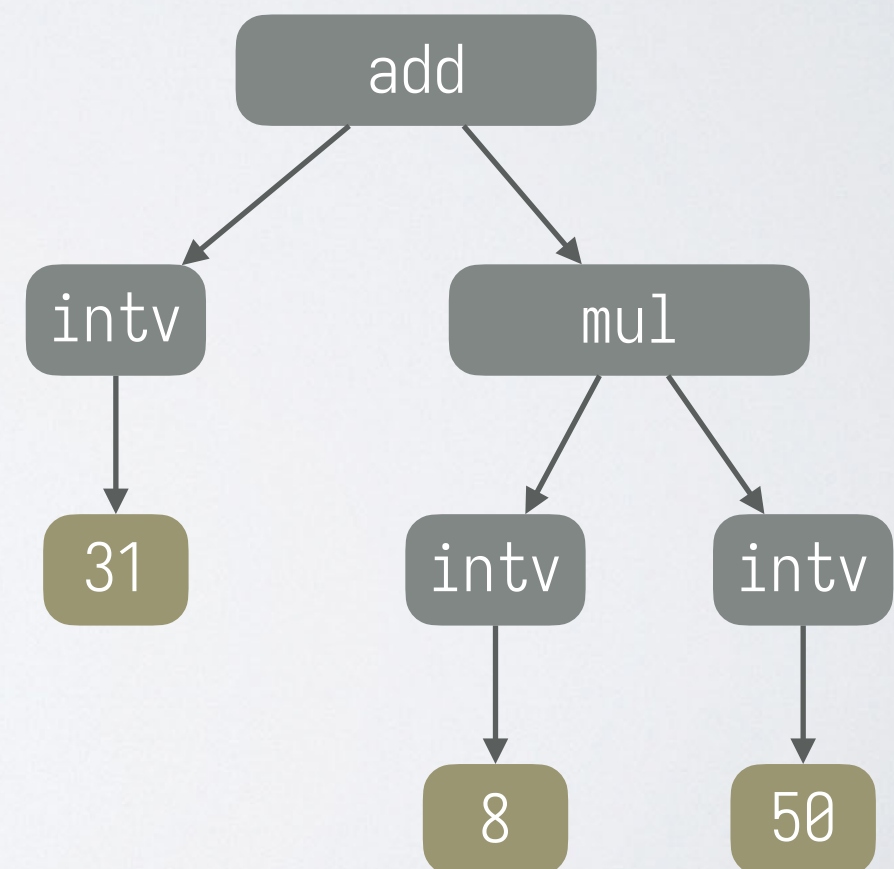
- ❖ **前端**: 理解源程序, 得到语义
- ❖ **后端**: 把语义翻译为目标程序
- ❖ **中间表示** (Intermediate Representation, IR): 编译器对语义的表示形式

回顾：抽象语法树

31 + 8 * 50

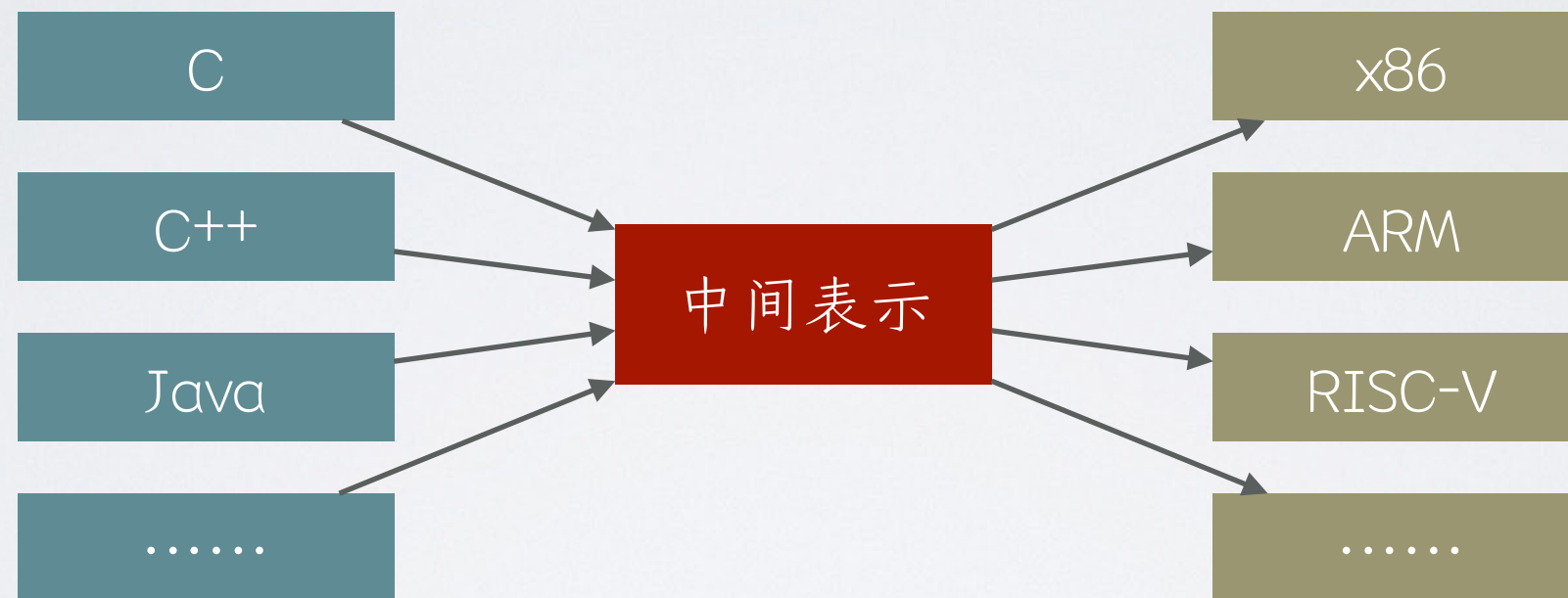


add(intv(31), mul(intv(8), intv(50)))



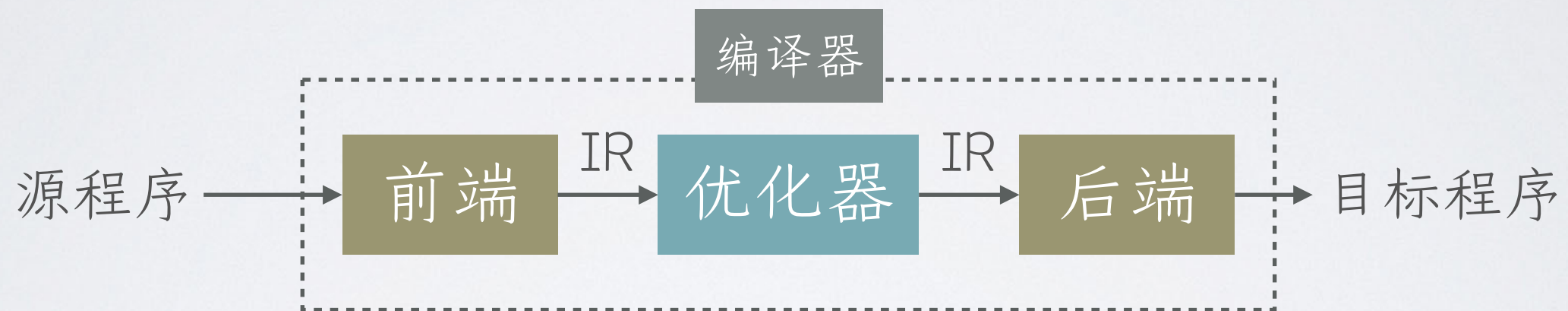
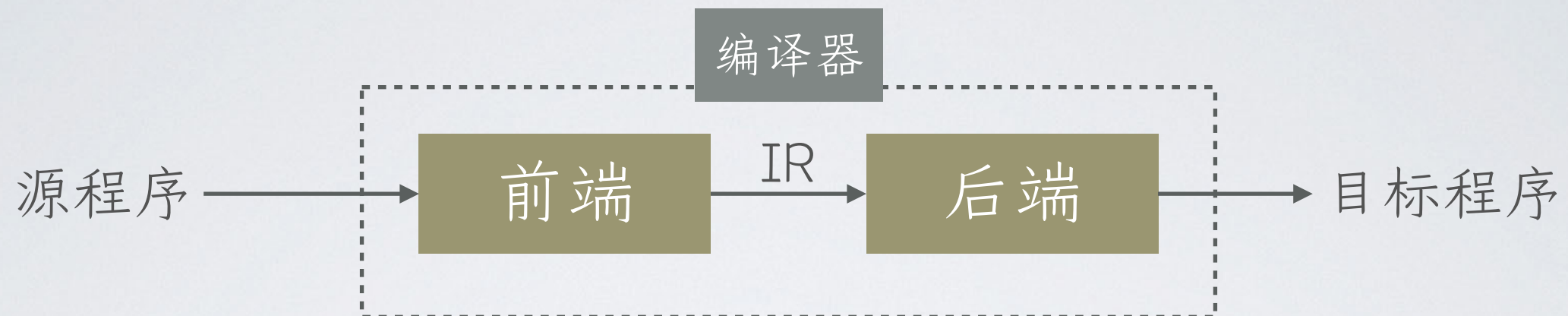
回顾：重定目标 (Retargeting)

- 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

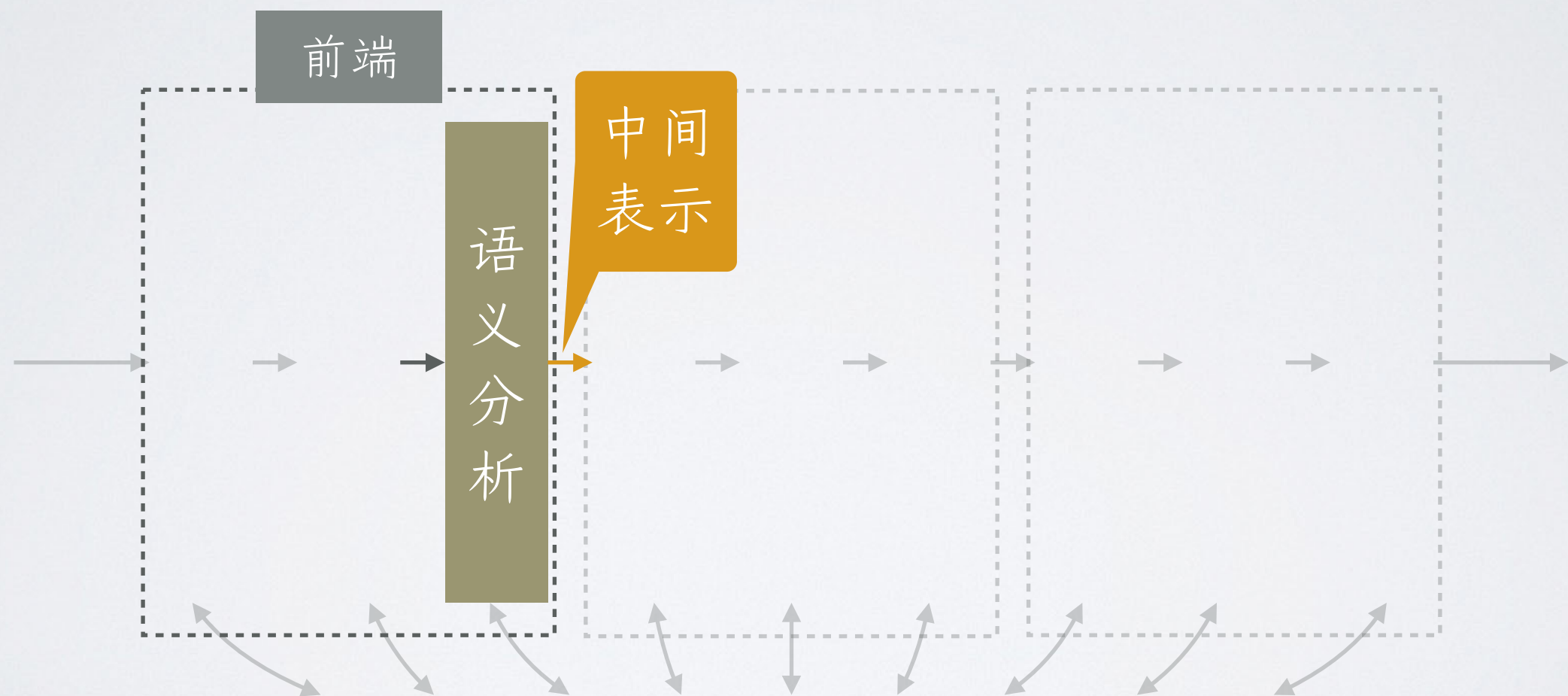
◎ 简化优化器的设计

- ❖ 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

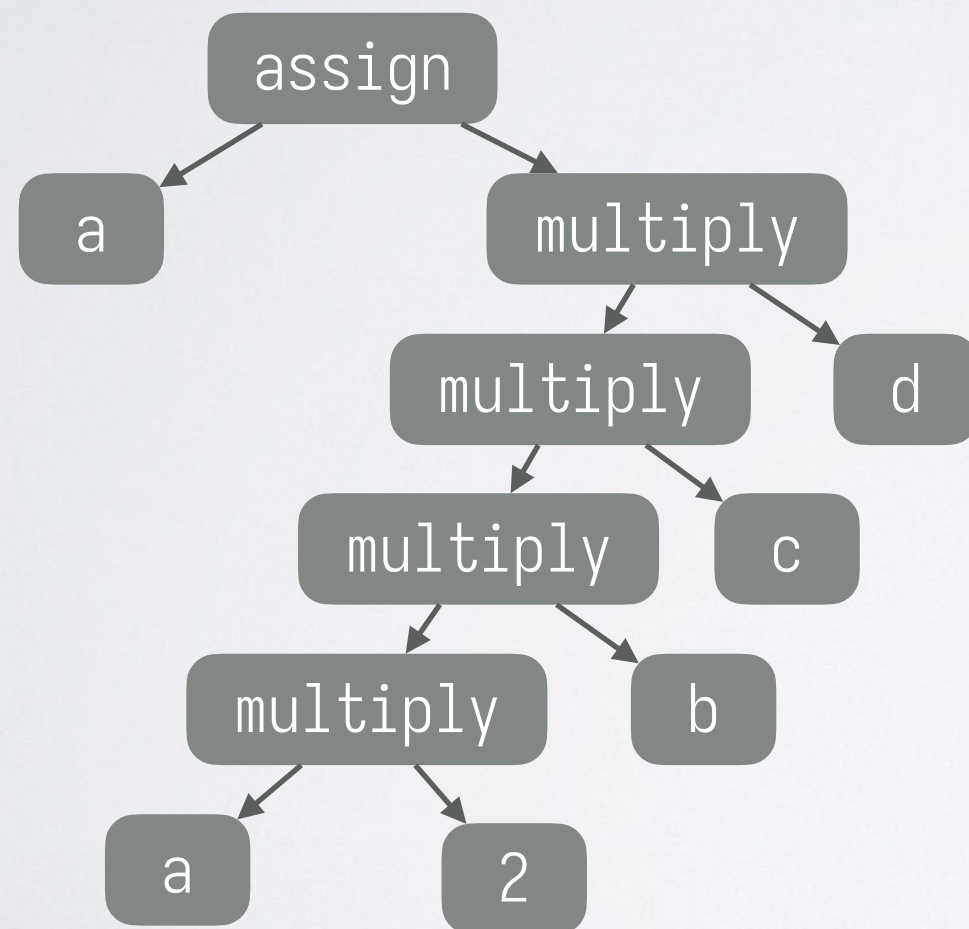
中间表示的设计考量

- ◎ 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
  
```

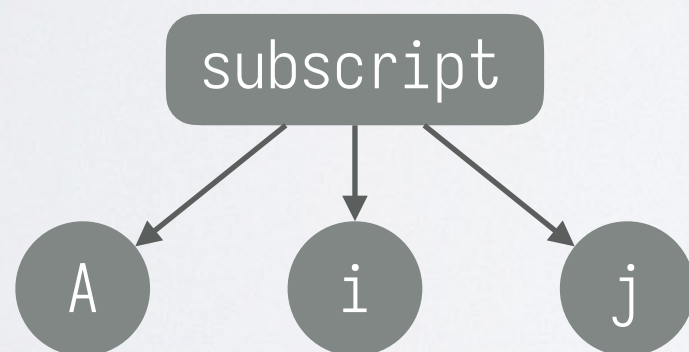
中间表示的抽象层次

● 所谓「抽象」, 不过是隐藏一定的「细节」

● 例: 对于二维数组数组 $A[1 \dots 10, 1 \dots 10]$, 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

```

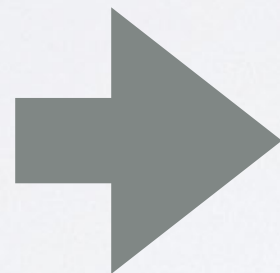
t0 = i - 1
t1 = t0 * 10
t2 = j - 1
t3 = t1 + t2
t4 = t3 * 4
t5 = A [ t4 ]
  
```

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);  
}
```

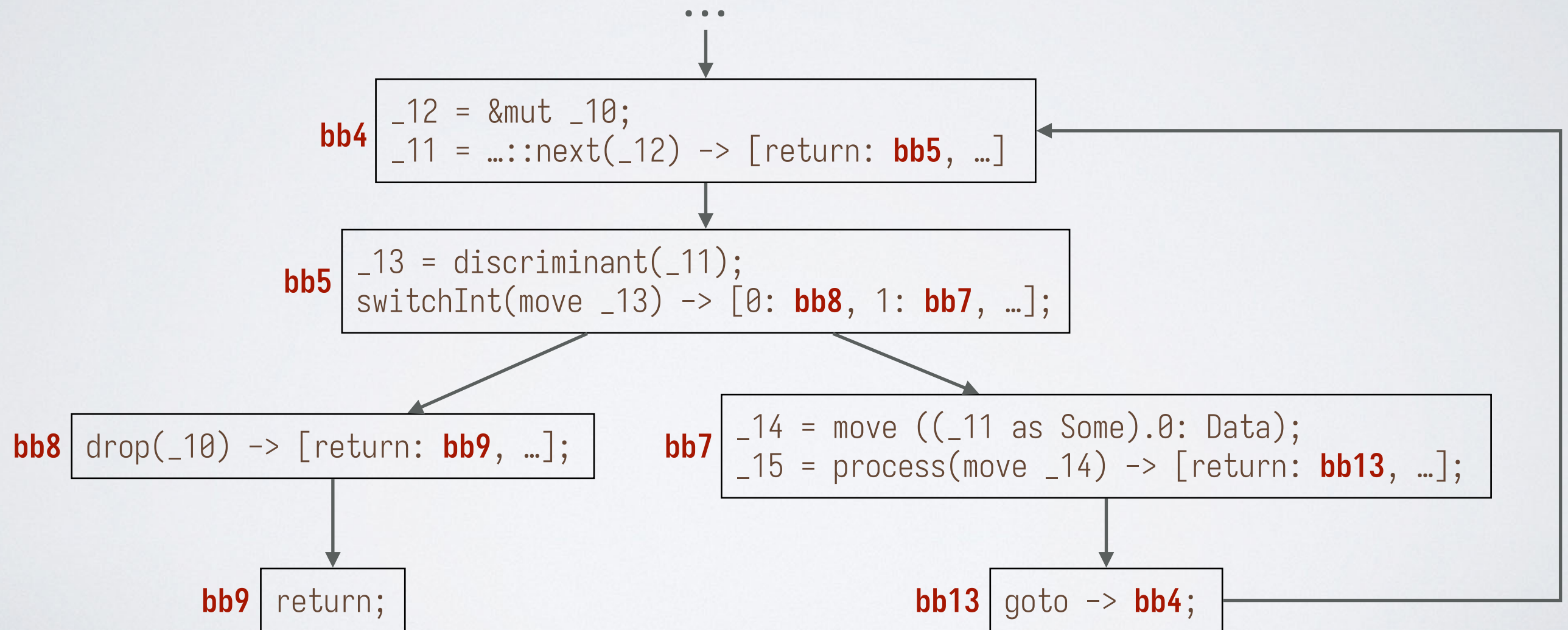


```
let mut iterator = vec.into_iter();  
loop {  
    match iterator.next() {  
        Some(elem) => process(elem),  
        None => break,  
    }  
}
```

案例：

- 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

IR 设计



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



.....

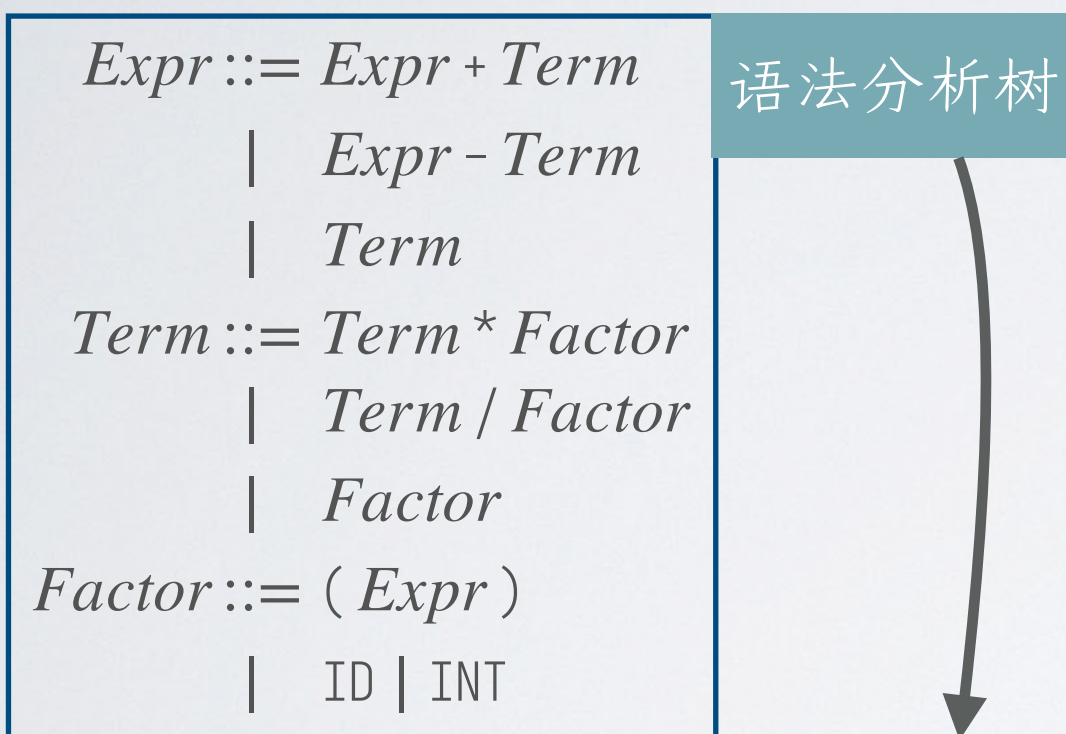
```
bb5:                                     ; preds = %bb4
%11 = extractvalue { i32, i32 } %6, 0, !dbg !2490
%12 = extractvalue { i32, i32 } %6, 1, !dbg !2490
store i32 %11, ptr %_11, align 4, !dbg !2490
%13 = getelementptr inbounds i8, ptr %_11, i64 4, !dbg !2490
store i32 %12, ptr %13, align 4, !dbg !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
```

.....

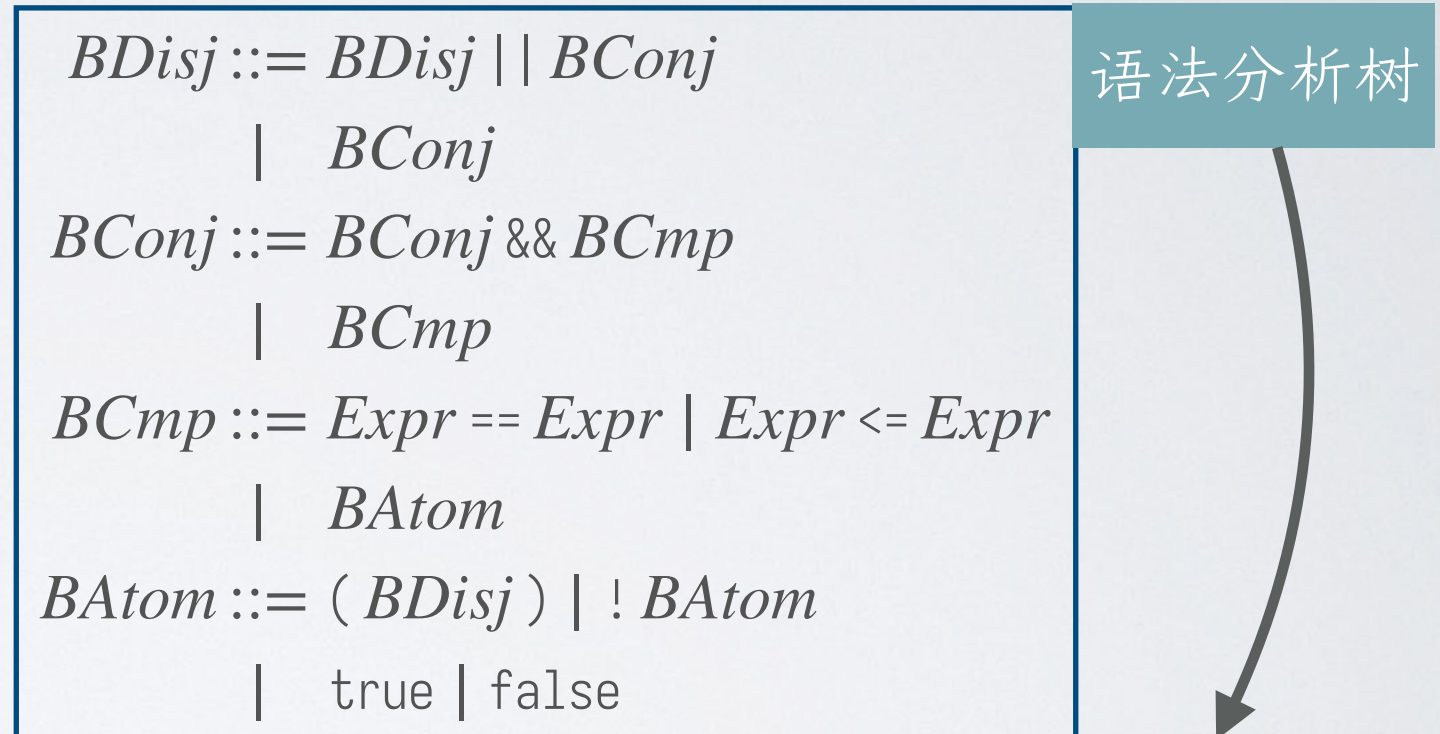
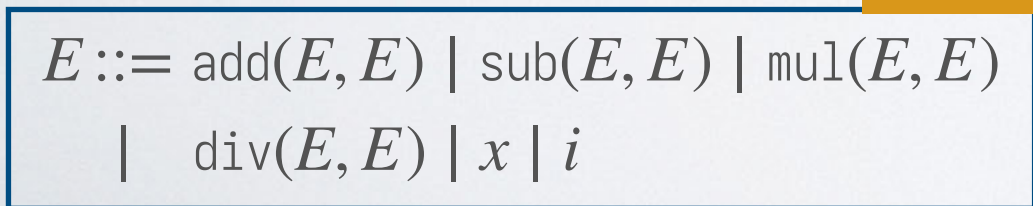
抽象语法树

◎ 抽象语法树 (Abstract Syntax Tree, AST)

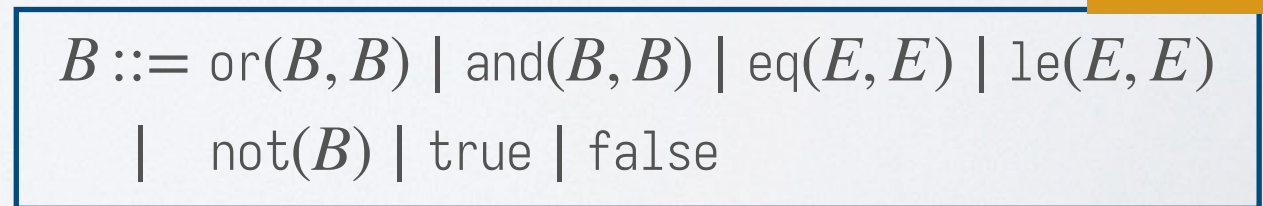
❖ 与语法分析树类似, 但去掉了不影响语义的冗余信息



AST



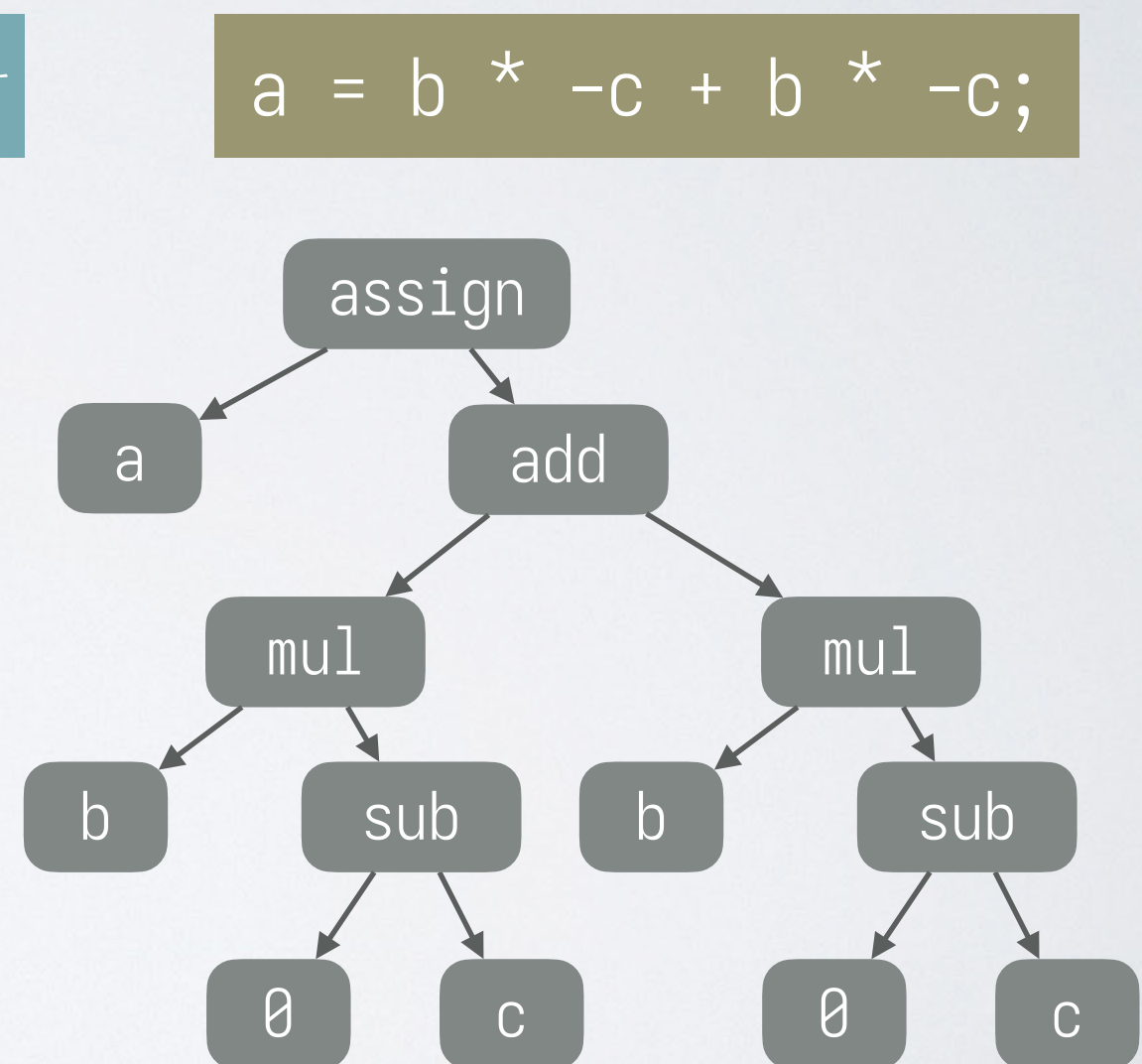
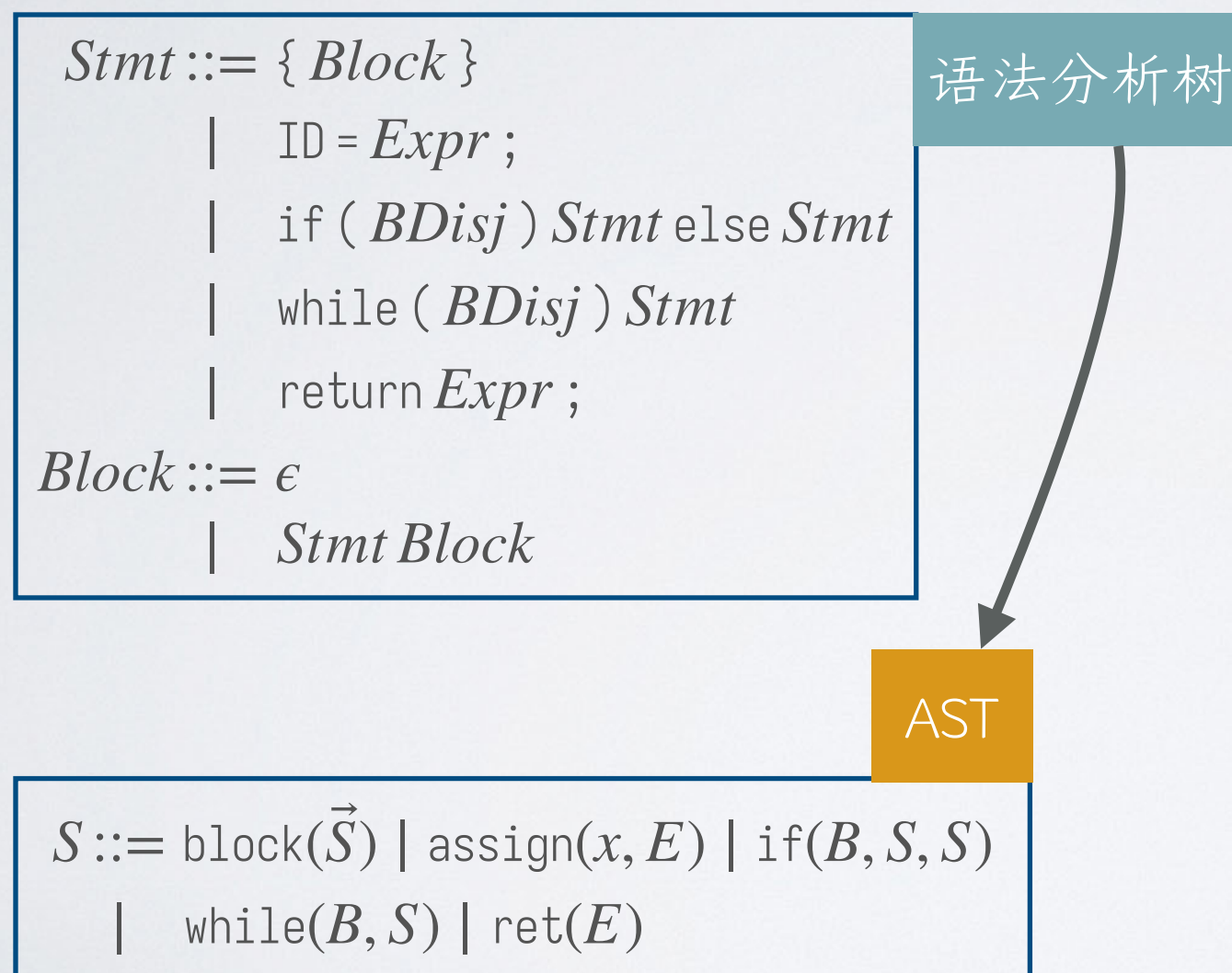
AST



抽象语法树

◎ 抽象语法树 (Abstract Syntax Tree, AST)

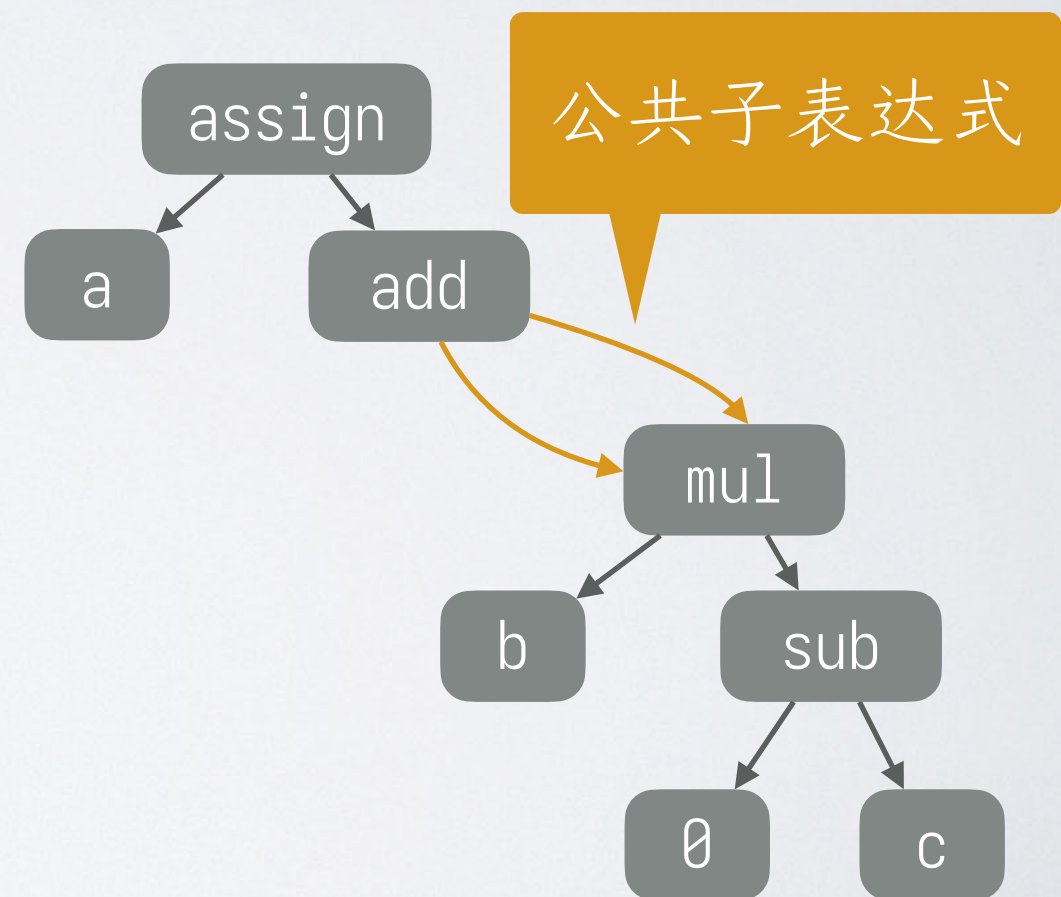
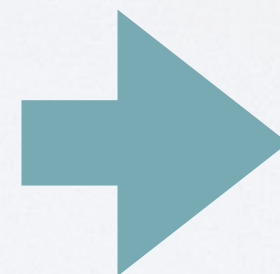
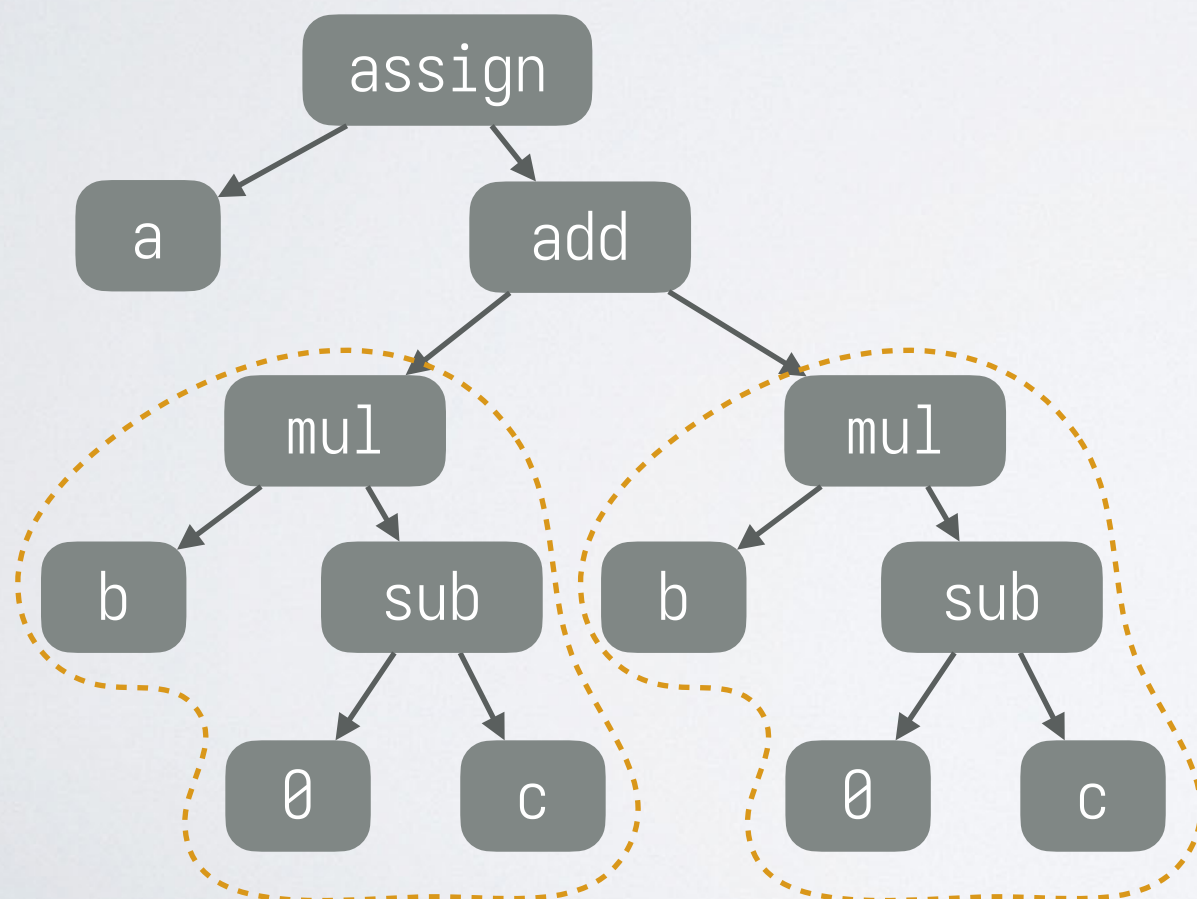
❖ 与语法分析树类似, 但去掉了不影响语义的冗余信息



有向无环图

有向无环图 (Directed Acyclic Graph, DAG)

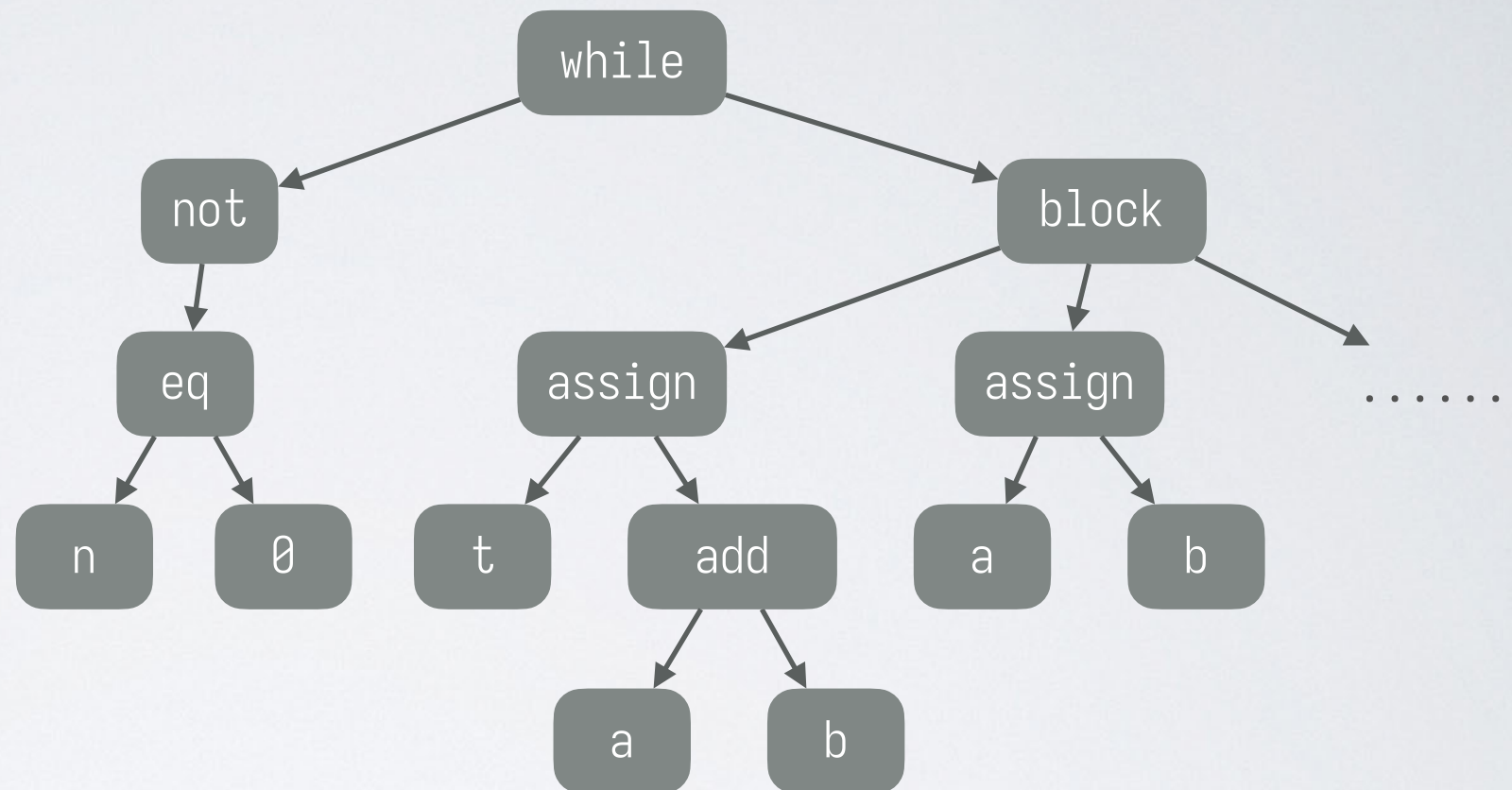
❖ 与抽象语法树类似, 但对相同的子树进行了合并

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


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;
}
```



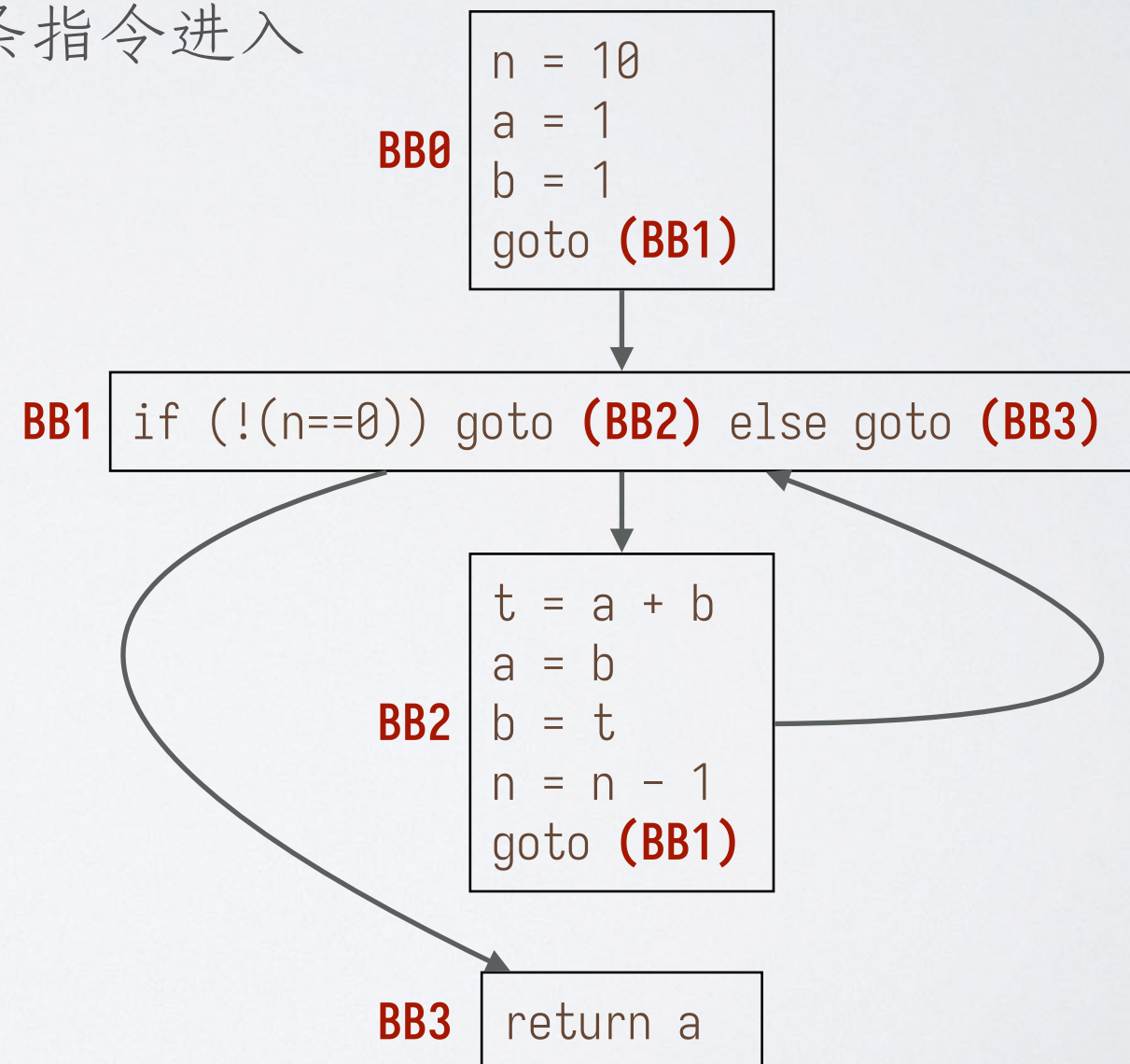
- 控制流图通过**有向图**中边的流向来表示程序的控制流
- 图中结点为语句, 或者一串不发生控制流跳转的语句

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

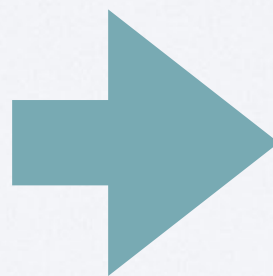
❖ 设计动机: 编译器通过简单的线性扫描生成目标代码

● 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;
```



```
t0 = 0 - c
t1 = b * t0
t2 = 0 - c
t3 = b * t2
t4 = t1 + t3
a = t4
```

三地址代码的形式

ID 表示地址

$$I ::= ID = ID \text{ } bop \text{ } ID \mid ID = uop \text{ } ID \mid ID = ID \mid goto \text{ } LABEL \\ \mid \text{ if } ID \text{ goto } LABEL \mid \text{ ifFalse } ID \text{ goto } LABEL \mid \text{ if } ID \text{ } rop \text{ } ID \text{ goto } LABEL$$

- $x = y \text{ } bop \text{ } z$: 双目运算符
- $x = uop \text{ } y$: 单目运算符(例如单目减、逻辑非)
- $x = y$: 复制指令
- $goto \text{ } L$: 无条件跳转指令, L 表示一条三地址指令的序号
- $\text{if/ifFalse } x \text{ goto } L$: 条件跳转指令
 - ❖ 当 x 为真/假时, 跳转到 L 处
- $\text{if/ifFalse } x \text{ } rop \text{ } y \text{ goto } L$: 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

$$I ::= \dots \mid \text{param ID, INT} \mid \text{call ID, INT} \mid \text{ID} = \text{call ID, INT} \mid \text{return ID}$$

- $\text{param } x, i$: 进行参数传递

- ❖ i 表示是第几个参数

- $\text{call } p, n$ 或 $y = \text{call } p, n$: 过程调用和函数调用

- ❖ n 是参数的个数

- $\text{return } y$: 返回指令

- 示例：调用有 n 个参数的过程 p

```
param x1, 1  
param x2, 2  
...  
param xn, n  
call p, n
```

三地址代码的形式：数组和指针操作

$$I ::= \dots \mid ID = ID[ID] \mid ID[ID] = ID \mid ID = \& ID \mid ID = * ID \mid * ID = ID$$

- ◎ $x = y[i]$: 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:     $t_1 = i + 1$   
         $i = t_1$   
         $t_2 = i * 8$   
         $t_3 = a[t_2]$   
        if  $t_3 < v$  goto L
```

符号标号

```
100:     $t_1 = i + 1$   
101:     $i = t_1$   
102:     $t_2 = i * 8$   
103:     $t_3 = a[t_2]$   
104:    if  $t_3 < v$  goto 100
```

位置标号

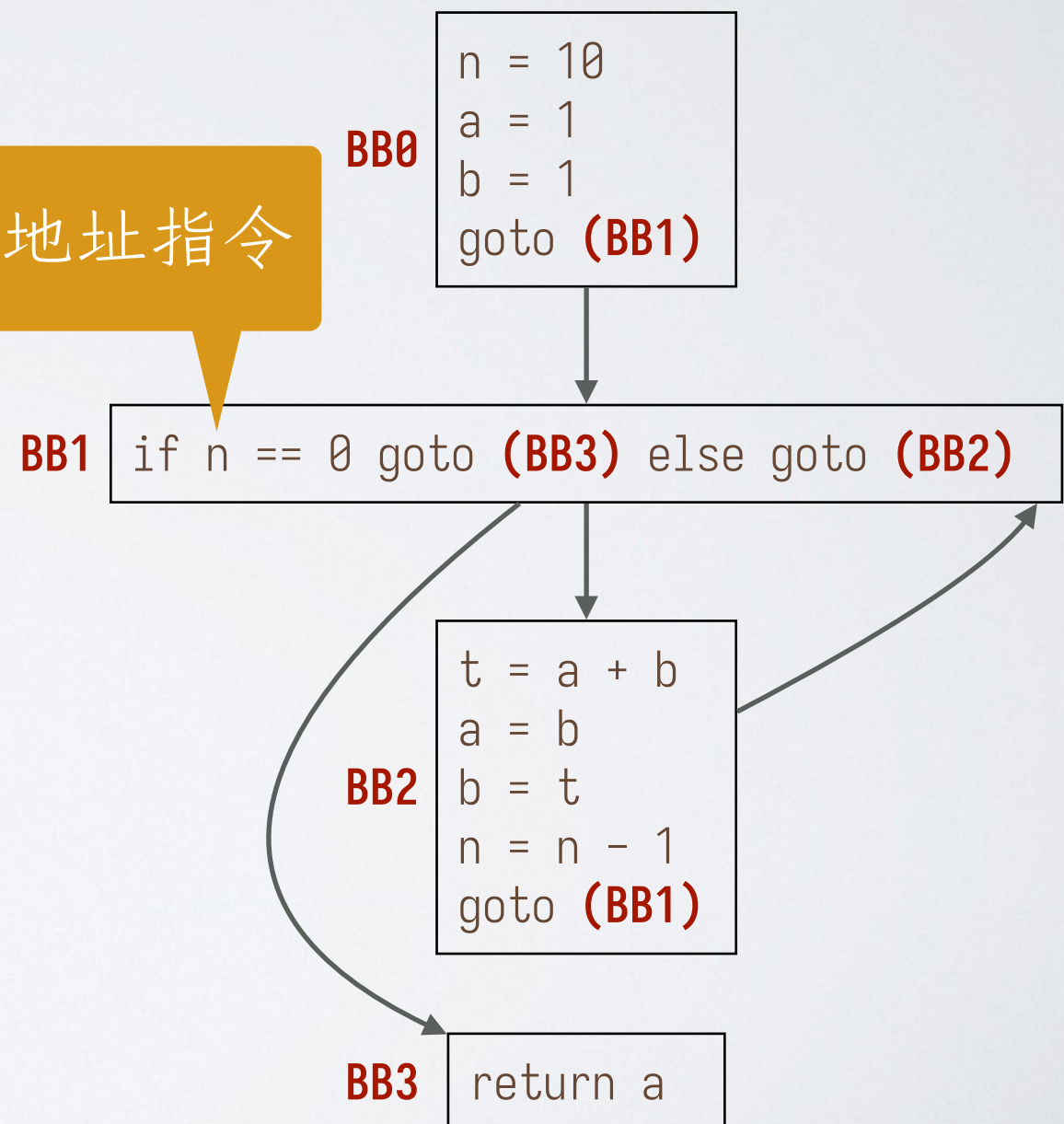
- 通过一次扫描可将符号化标号替换为实际的指令位置

控制流图 + 三地址代码

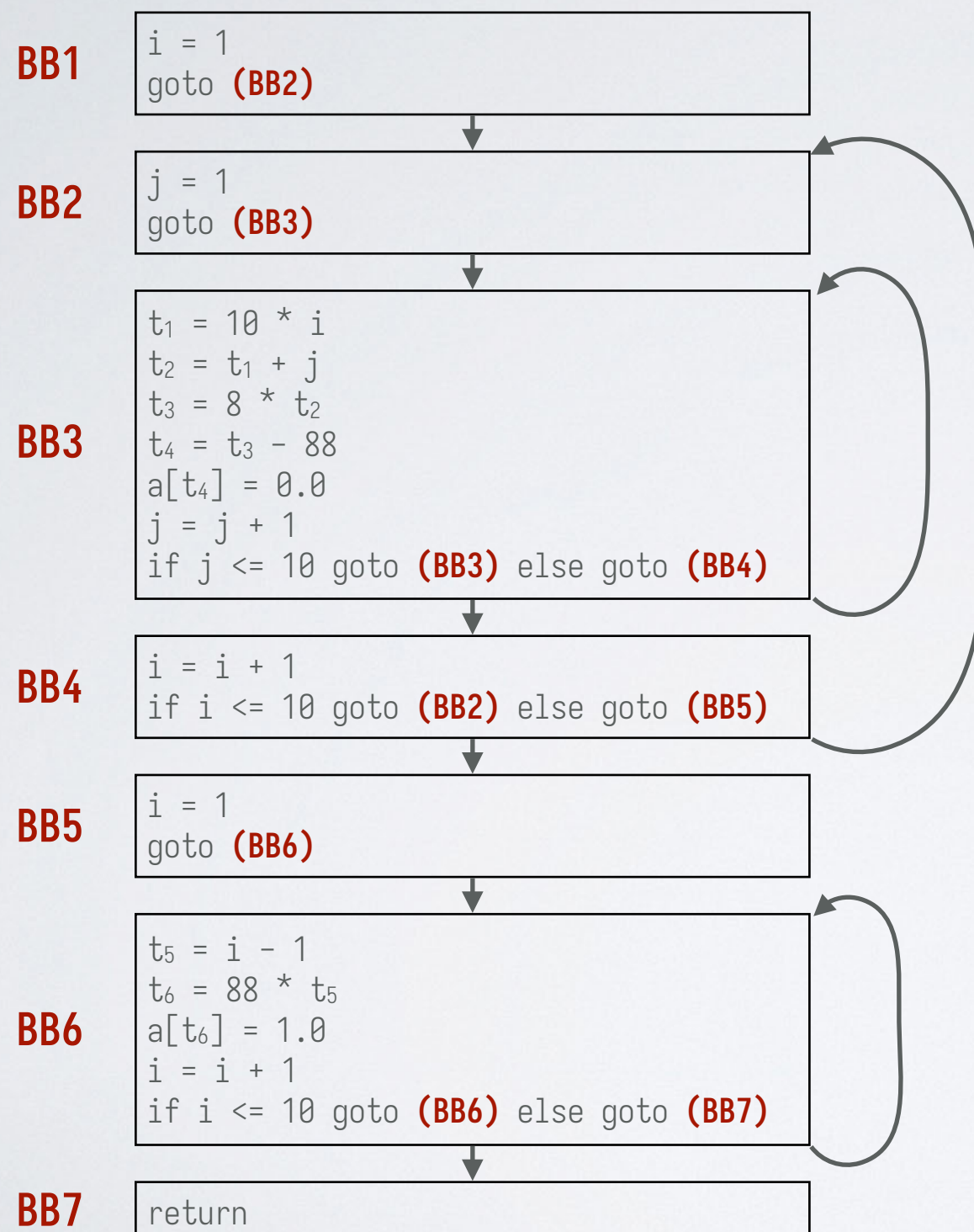
- 控制流图的每个基本块内部为三地址代码
 - ❖ 跳转指令的目标为基本块(而不是指令标号)
 - ❖ 一种常见的混合 IR

```
{
  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

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 and 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
```

字面上意义上相同的
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) $x_1 = -1$; **else** $x_2 = 1$; $x_3 = \varphi(x_1, x_2)$; $y = x_3 * a$;

- ❖ LLVM IR 采用这种设计

- ◎ 使用带参数的跳转指令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

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

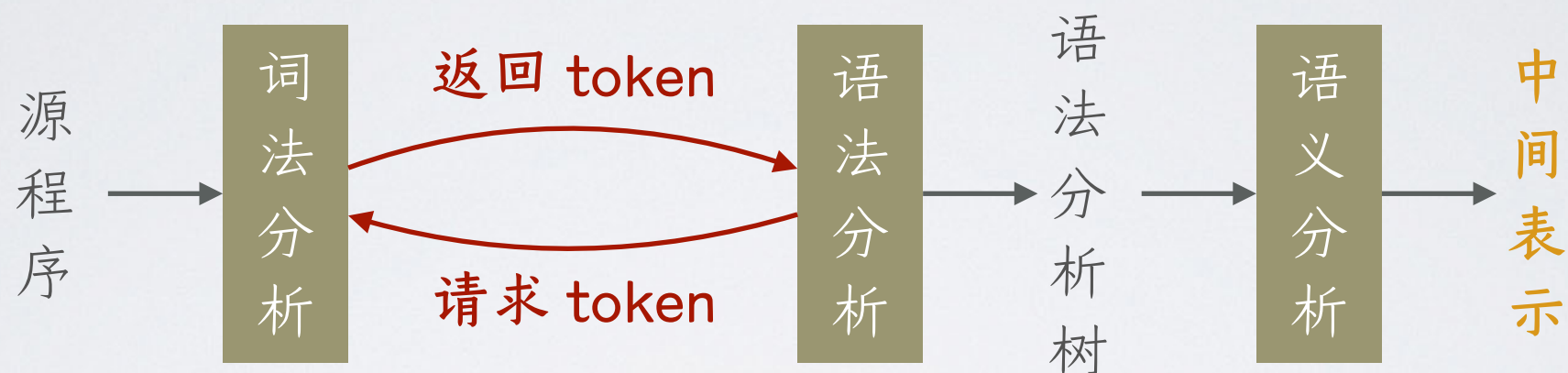
```
x0 = ...  
y0 = ...  
if x0 >= 100 goto next  
loop: x1 =  $\varphi(x_0, x_2)$   
      y1 =  $\varphi(y_0, y_2)$   
      x2 = x1 + 1  
      y2 = y1 + x2  
      if x2 < 100 goto loop  
next: x3 =  $\varphi(x_0, x_2)$   
      y3 =  $\varphi(y_0, y_2)$ 
```

主要内容

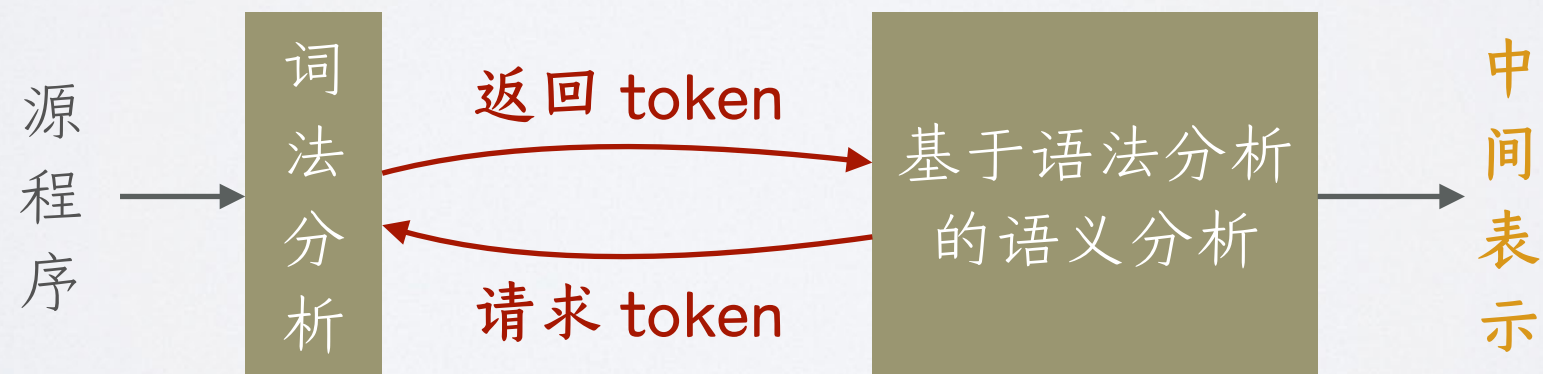
- ◎ 中间表示的作用
- ◎ 中间表示的设计
- ◎ 中间表示的生成

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)

- ◎ 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

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

Expr、*Term*、*Factor* 的属性
val 都是综合属性

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

回顾：属性文法/语法制导定义

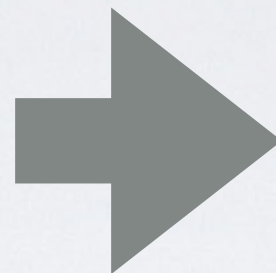
- 属性文法 = 上下文无关文法 + 属性计算规则
- 综合属性、继承属性
- S 属性的文法、L 属性的文法

产生规则	继承属性 inh 记录 左侧已经计算的值	属性计算规则
$Expr \rightarrow Term\ Expr'$		$Expr'.inh = Term.val$ $Expr.val = Expr'.syn$
$Expr' \rightarrow +Term\ Expr'_1$		$Expr'_1.inh = Expr'.inh + Term.val$ $Expr'.syn = Expr'_1.syn$
$Expr' \rightarrow \epsilon$		$Expr'.syn = Expr'.inh$
...		...
$Factor \rightarrow (Expr)$		$Factor.val = Expr.val$
$Factor \rightarrow INT$		$Factor.val = INT.intval$

综合属性 syn 记录在
inh 基础上计算的值

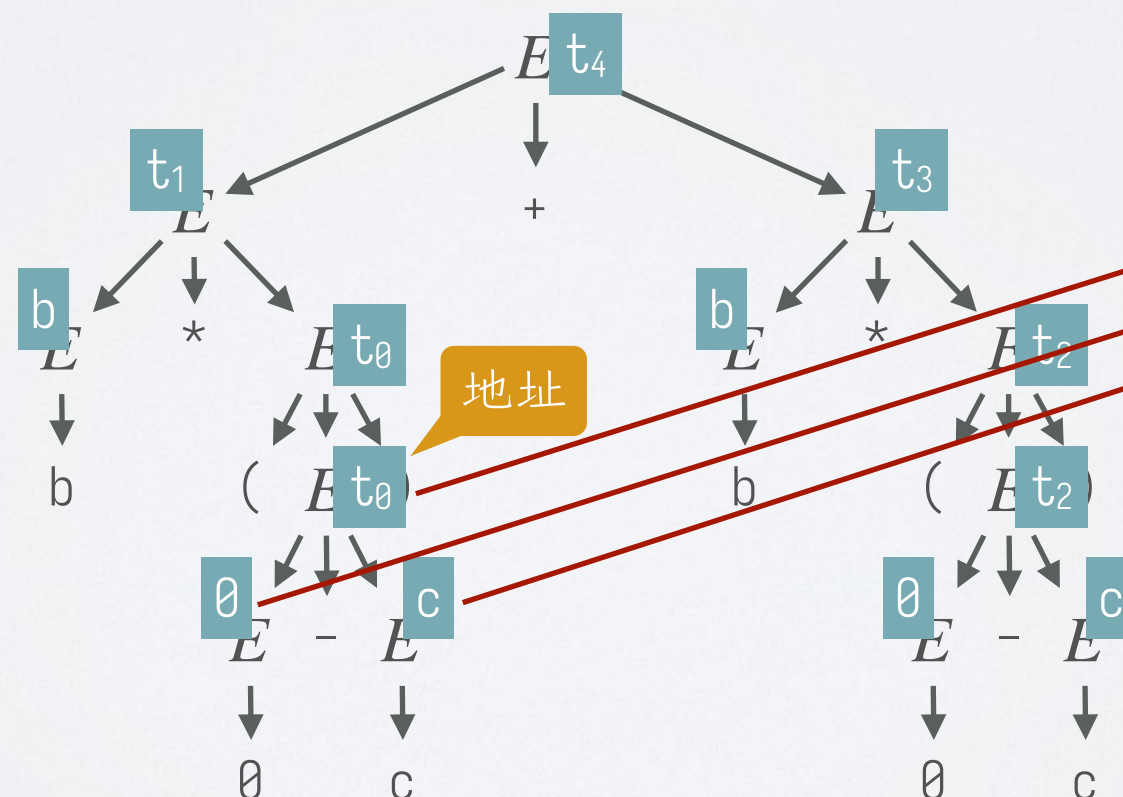
表达式和赋值语句

- 为了简化讨论, 我们使用等价的有二义性的文法来说明

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


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

- 思路: 如何通过遍历语法分析树来生成三地址代码?

$$\begin{aligned} a &= b * (\emptyset - c) + \\ &\quad b * (\emptyset - c); \end{aligned}$$


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

翻译表达式和赋值语句的属性文法

- 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
(可以基于符号表来实现)

属性计算规则

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

`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

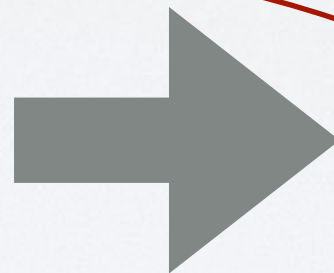
声明语句和语句块的翻译

$$S ::= \dots \mid T \text{ ID}; \mid \{ L \}$$

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

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

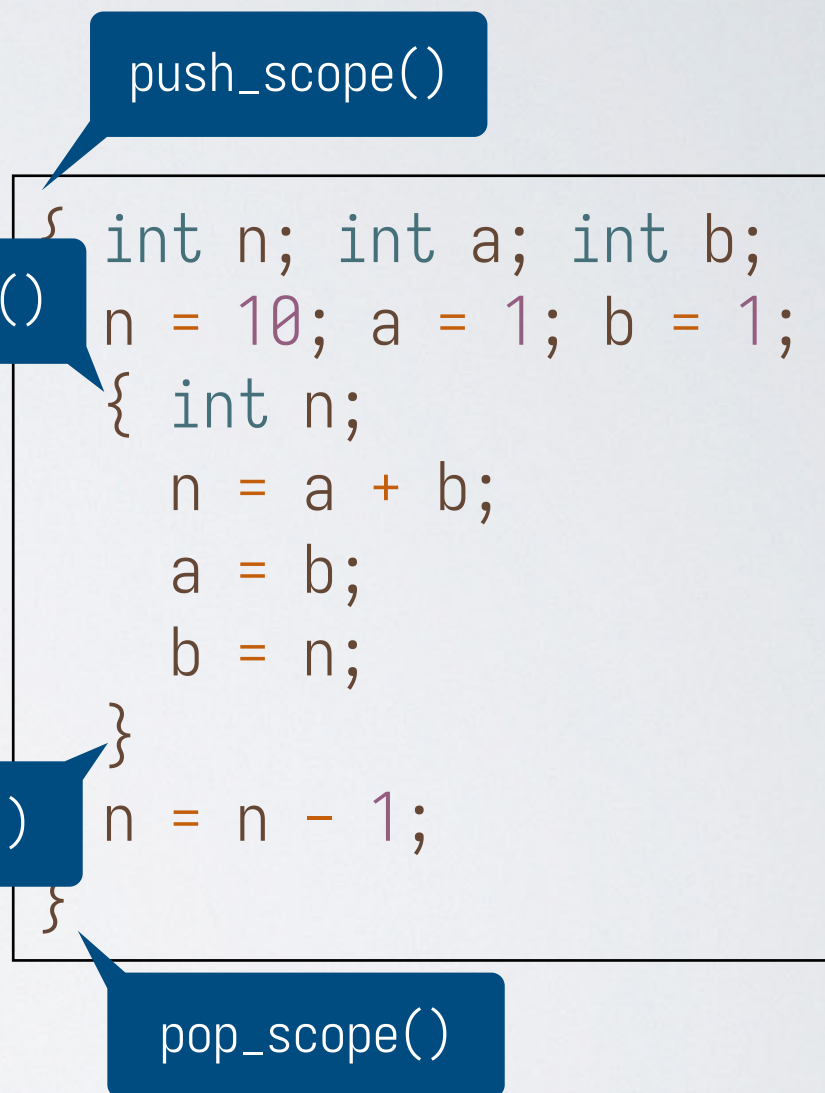
```
{ int n; int a; int b;
  n = 10; a = 1; b = 1;
  { int n;
    n = a + b;
    a = b;
    b = n;
  }
  n = n - 1;
}
```



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

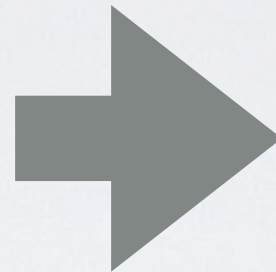
翻译声明语句和语句块的属性文法

产生规则	属性计算规则
$S \rightarrow T ID ;$	$S.table_out = S.table_in.insert(ID.lexeme, T.type)$ $S.code = ``$
$S \rightarrow \{L\}$	$L.table_in = S.table_in.push_scope()$ $S.table_out = L.table_out.pop_scope()$ $S.code = L.code$
$L \rightarrow \epsilon$	$L.table_out = L.table_in$ $L.code = ``$
$L \rightarrow L_1 S$	$L_1.table_in = L.table_in$ $S.table_in = L_1.table_out$ $L.table_out = S.table_out$ $L.code = L_1.code \parallel S.code$
$T \rightarrow int$	$T.type = int$
$T \rightarrow double$	$T.type = double$



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

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


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

- 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**
 - ❖ $B_1 \mid\mid B_2$ 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** ($x < 100 \ || \ (x > 200 \ \&\& \ x \neq y)$) $x = 0$;

- 三地址代码:

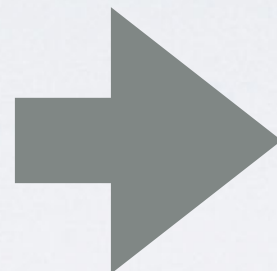
if	$x < 100$	goto L_1
ifFalse	$x > 200$	goto L_0
ifFalse	$x \neq y$	goto L_0
L_1 :	$x = 0$	
L_0 :	接下来的代码	

- 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

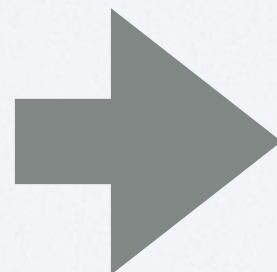
$$S ::= \dots \mid \text{if}(B) S \mid \text{if}(B) S \text{ else } S$$

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



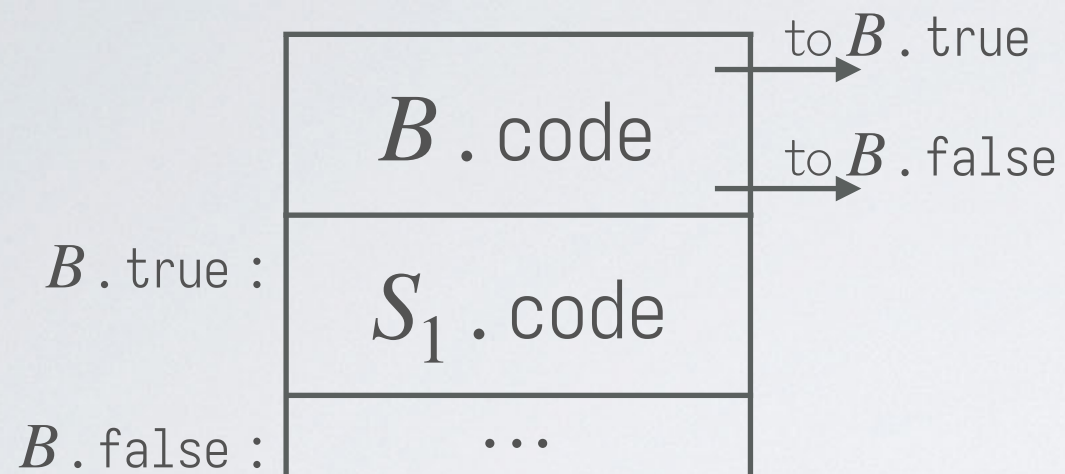
```
if n <= 0 goto L1
goto L0
L1: t0 = 0 - n
    n = t0
L0:
```

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

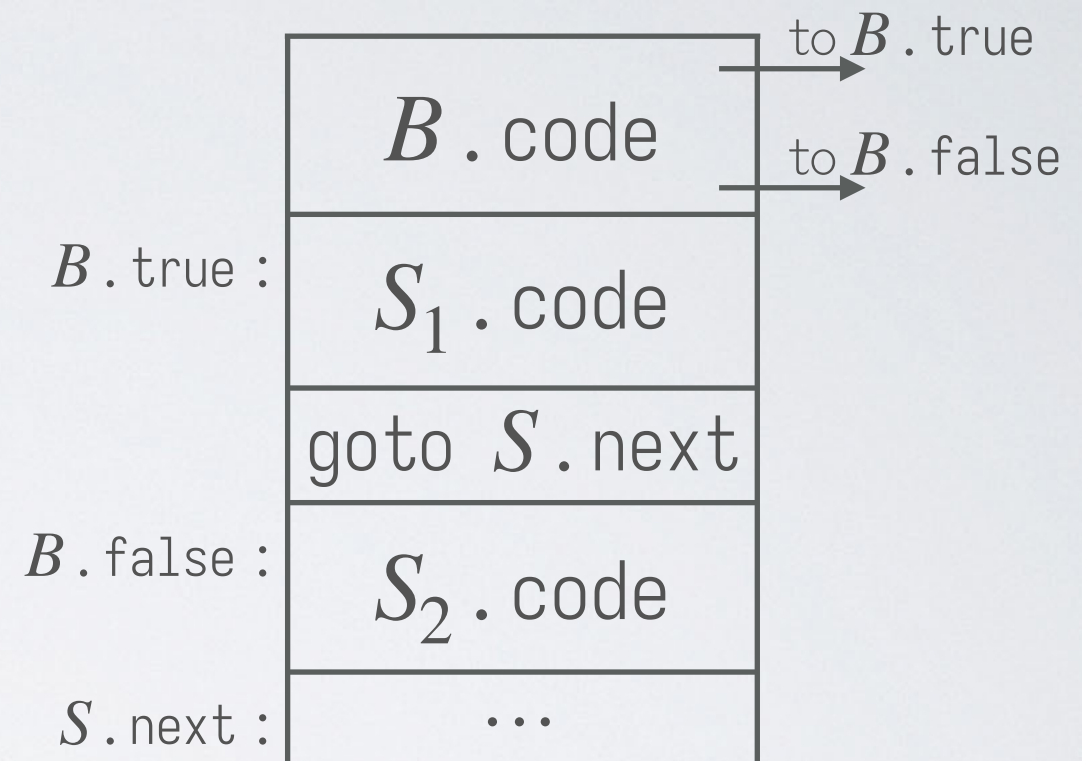


```
if n <= 0 goto L1
goto L2
L1: t0 = 0 - n
    n = t0
    goto L0
L2: t1 = n + 1
    n = t1
L0:
```

条件语句的翻译思路



$S \rightarrow \text{if} (B) S_1$



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

继承属性：

- ◉ $B . \text{true}$: B 为真时的跳转标号
- ◉ $B . \text{false}$: B 为假时的跳转标号
- ◉ $S . \text{next}$: 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

产生规则	计算规则
$S' \rightarrow S \text{ EOF}$	$S.\text{next} = \text{genlabel}()$ $S'.\text{code} = S.\text{code} \parallel \text{'}\{S.\text{next}\}:\text{'}$
$S \rightarrow \{L\}$	$L.\text{next} = S.\text{next}$ $S.\text{code} = L.\text{code}$
$L \rightarrow L_1 S$	$L_1.\text{next} = \text{genlabel}()$ $S.\text{next} = L.\text{next}$ $L.\text{code} = L_1.\text{code} \parallel \text{'}\{L_1.\text{next}\}:\text{'} \parallel S.\text{code}$

继承属性:

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



继承属性：

- 2025 年春季学期

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

翻译布尔表达式的属性文法

产生规则	属性计算规则
$B \rightarrow \text{true}$	$B.\text{code} = \text{'goto \{B.true\}'}$
$B \rightarrow \text{false}$	$B.\text{code} = \text{'goto \{B.false\}'}$
$B \rightarrow (B_1)$	$B_1.\text{true} = B.\text{true}$ $B_1.\text{false} = B.\text{false}$ $B.\text{code} = B_1.\text{code}$
$B \rightarrow !B_1$	$B_1.\text{true} = B.\text{false}$ $B_1.\text{false} = B.\text{true}$ $B.\text{code} = B_1.\text{code}$
$B \rightarrow E_1 == E_2$	$B.\text{code} = E_1.\text{code} \parallel E_2.\text{code}$ $\parallel \text{'if \{E_1.addr\} == \{E_2.addr\} goto \{B.true\}'}$ $\parallel \text{'goto \{B.false\}'}$

继承属性:

- ◉ $B.\text{true}$: B 为真时的跳转标号
- ◉ $B.\text{false}$: B 为假时的跳转标号

翻译布尔表达式的属性文法

产生规则	属性计算规则
$B \rightarrow B_1 \&\& B_2$	$B_1.\text{true} = \text{genlabel}()$ $B_1.\text{false} = B.\text{false}$ $B_2.\text{true} = B.\text{true}$ $B_2.\text{false} = B.\text{false}$ $B.\text{code} = B_1.\text{code} \parallel B_2.\text{code}$
$B \rightarrow B_1 \parallel B_2$	$B_1.\text{true} = B.\text{true}$ $B_1.\text{false} = \text{genlabel}()$ $B_2.\text{true} = B.\text{true}$ $B_2.\text{false} = B.\text{false}$ $B.\text{code} = B_1.\text{code} \parallel \text{'\{B}_1.\text{false}\}:'} \parallel B_2.\text{code}$

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:

- ◉ $B.\text{true}$: B 为真时的跳转标号
- ◉ $B.\text{false}$: B 为假时的跳转标号

布尔表达式翻译示例

◎ 源程序:

❖ **if** ($x < 100 \ || \ (x > 200 \ \&\& \ x \neq y)$) $x = 0$;

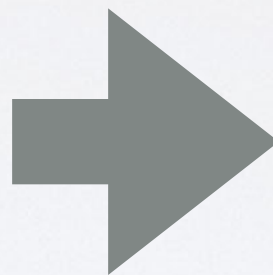
◎ 三地址代码:

```
        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

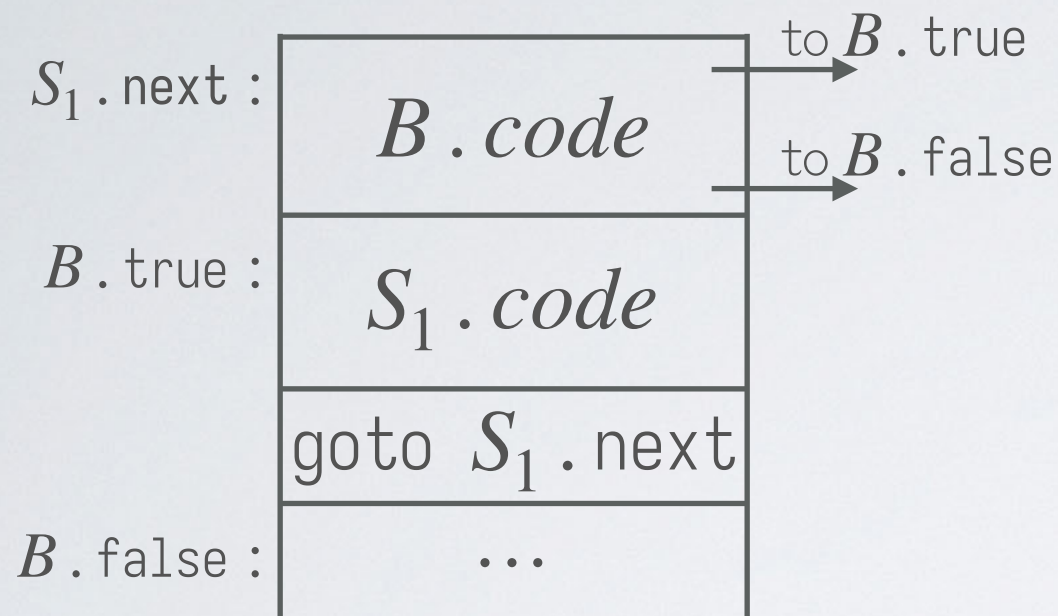
$$S ::= \dots \mid \text{while}(B) S$$

```
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:
```

翻译循环语句的属性文法



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

产生规则	属性计算规则
$S \rightarrow \text{while}(B) S_1$	$B.\text{true} = \text{genlabel}()$ $B.\text{false} = S.\text{next}$ $S_1.\text{next} = \text{genlabel}()$ $S.\text{code} = \text{'}\{S_1.\text{next}\}:\text{' } \parallel B.\text{code}$ $\quad \parallel \text{'}\{B.\text{true}\}:\text{' } \parallel S_1.\text{code}$ $\quad \parallel \text{'goto } \{S_1.\text{next}\}\text{'}$

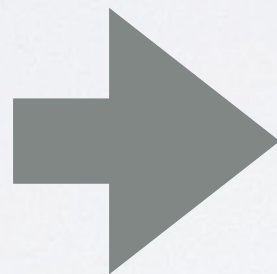
练习：如何支持
break 和 continue?

继承属性：

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

多分支选择语句的翻译思路

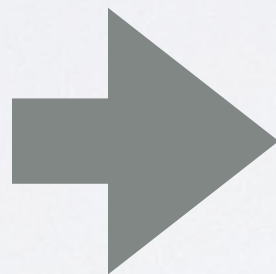
```
switch (  $E$  ) {  
    case  $V_1$ :  $S_1$   
    case  $V_2$ :  $S_2$   
        ...  
    case  $V_{n-1}$ :  $S_{n-1}$   
    default:  $S_n$   
}
```



```
    计算  $E$  并把结果赋给临时变量  $t$   
    goto test  
L1:     $S_1$  的代码  
        goto next  
L2:     $S_2$  的代码  
        goto next  
        ...  
Ln-1:   $S_{n-1}$  的代码  
        goto next  
Ln:     $S_n$  的代码  
        goto next  
test:   if  $t = V_1$  goto L1  
        if  $t = V_2$  goto L2  
        ...  
        if  $t = V_{n-1}$  goto Ln-1  
        goto Ln  
next:
```

过程/函数调用的翻译方案

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



计算 E_1 并把结果赋给临时变量 t_1
param $t_1, 1$
计算 E_2 并把结果赋给临时变量 t_2
param $t_2, 2$
...
计算 E_n 并把结果赋给临时变量 t_n
param t_n, n
 $t = \text{call } f, n$
 $x = t$

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

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

- ◎ 实践演示

玩具语言

- ◎ C 语言, 但只有主函数, 只有 int 类型, 只有直线代码

```
int main() {  
    int x = 10;  
    int y = x + 1 * x;  
    int z = x * (y + 1);  
    return z % 66;  
}
```

```
int main() {  
    int y;  
    y = 1;  
    int main;  
    main = y + 1;  
    return main;  
}
```

```
int main() {  
    int a = 1;  
    {  
        a = a + 2;  
        int a = 3;  
        a = a + 4;  
    }  
    a = a + 5;  
    return a;  
}
```

```
int main() {  
    int a = 1;  
    int b = a + 2;  
    {  
        b = a + b;  
        {  
            int a = 3;  
            b = b + a;  
            { int b = 1; }  
            { return b; }  
        }  
        int b = 1;  
    }  
}
```

玩具语言的抽象语法树表示

抽象语法

$E ::= \text{binop}(op, E, E) \mid \text{id}(x) \mid \text{number}(i)$

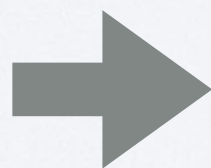
$op ::= \text{add} \mid \text{sub} \mid \text{mul} \mid \text{div} \mid \text{mod}$

$S ::= \text{scope}(S) \mid \text{decl}(x, S) \mid \text{assign}(x, E) \mid \text{ret}(E) \mid \text{seq}(S, S) \mid \text{nop}$

$F ::= \text{func}(x, S)$

$P ::= \text{prog}(F)$

```
int main() {  
  int x = 10;  
  int y = x + 1 * x;  
  int z = x * (y + 1);  
  return z % 66;  
}
```



```
prog(  
  func(main,  
    decl(x,  
      seq(assign(x, 10),  
        decl(y,  
          seq(assign(y, x + (1 * x)),  
            decl(z,  
              seq(assign(z, x * (y + 1)),  
                ret(z % 66))))))))))
```

玩具语言的类三地址 IR

「纯」表达式

$$R ::= \text{id}(x) \mid \text{num}(i) \mid \text{add}(R_1, R_2) \mid \text{sub}(R_1, R_2) \mid \text{mul}(R_1, R_2)$$

「三」地址指令

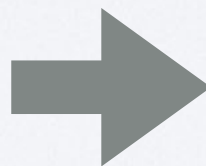
$$I ::= \text{move}(x, R) \mid \text{div}(x, R_1, R_2) \mid \text{mod}(x, R_1, R_2) \mid \text{ret}(R) \mid \text{label}(\ell)$$

函数体是
指令列表

$$F ::= \text{func}(x, \vec{I})$$

$$P ::= \text{prog}(F)$$

```
prog(
  func(main,
    decl(x,
      seq(assign(x, 10),
        decl(y,
          seq(assign(y, x + (1 * x)),
            decl(z,
              seq(assign(z, x * (y + 1)),
                ret(z % 66))))))))))
```



```
main:
// label("main")
x.1 <- 10
// move(x.1, num(10))
y.2 <- x.1 + (1 * x.1)
// move(y.2, add(id(x.1), mul(num(1), id(x.1))))
z.3 <- x.1 * (y.2 + 1)
// move(z.3, mul(id(x.1), add(id(y.2), num(1))))
%t.4 <- z.3 % 66
// mod(%t.4, id(z.3), num(66))
ret %t.4
// ret(id(%t.4))
```

产生规则	属性计算规则
$E \rightarrow \text{id}(x)$	$E.res = \text{'id}(\{E.table.lookup(x)\})'$ $E.code = ''$
$E \rightarrow \text{number}(i)$	$E.res = \text{num}(i)$ $E.code = ''$
$E \rightarrow \text{binop}(\text{add}, E_1, E_2)$	$E.res = \text{'add}(\{E_1.res\}, \{E_2.res\})'$ $E_1.table = E_2.table = E.table$ $E.code = E_1.code \parallel E_2.code$
$E \rightarrow \text{binop}(\text{div}, E_1, E_2)$	$E.res = \text{'id}(\{E.table.gentmp()\})'$ $E_1.table = E_2.table = E.table$ $E.code = E_1.code \parallel E_2.code \parallel \text{'div}(\{E.res.name\}, \{E_1.res\}, \{E_2.res\})'$
$S \rightarrow \text{scope}(S_1)$	$S_1.table_in = S.table_in.push_scope()$ $S.table_out = S_1.table_out.pop_scope()$
$S \rightarrow \text{decl}(x, S_1)$	$S_1.table_in = S.table_in.insert(x, \text{genvar}(x))$ $S.table_out = S_1.table_out.remove(x)$
$S \rightarrow \text{assign}(x, E)$	$E.table = S.table_out = S.table_in$ $S.code = S_1.code$ $S.code = E.code \parallel \text{'move}(\{S.table_in.lookup(x)\}, \{E.res\})'$
$S \rightarrow \text{ret}(E)$	$E.table = S.table_out = S.table_in$ $S.code = E.code \parallel \text{'ret}(\{E.res\})'$
$S \rightarrow \text{seq}(S_1, S_2)$	$S_1.table_in = S.table_in$ $S_2.table_in = S_1.table_out$ $S.table_out = S_2.table_out$ $S.code = S_1.code \parallel S_2.code$
$S \rightarrow \text{nop}$	$S.table_out = S.table_in$ $S.code = ''$
$F \rightarrow \text{func}(x, S)$	$S.table_in = \text{newtable}()$ $F.code = \text{'label}(\{x\})' \parallel S.code$

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、静态单赋值形式

◎ 中间表示的生成

- ❖ 基于属性文法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

◎ 后续可能讲的专题:

- ❖ 静态单赋值形式的生成

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？
- ◎ 三地址代码有什么优点和缺点？现代 IR 还会采用它吗？
- ◎ 编译过程中大概设计多少种中间形式比较合适？
- ◎ 如何支持 break、continue、goto 等非结构化控制流的翻译？
- ◎ 面向对象语言、函数式语言的 IR 需要考虑哪些特性？