本地优化

Local Optimization

SSA

钮鑫涛

基本块&控制流图

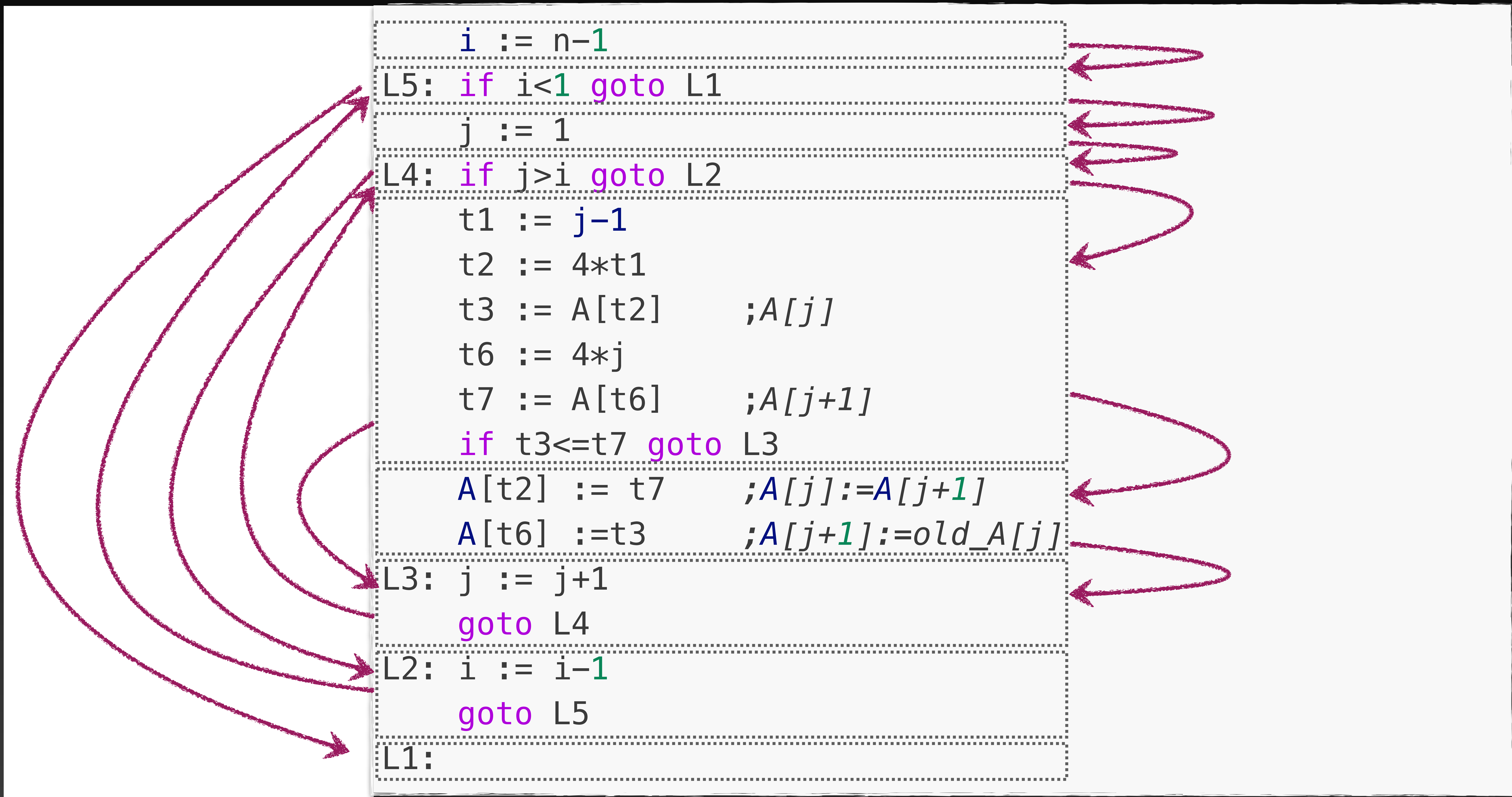
- 基本块 = 三地址码的指令序列
  - 只有基本块的第一句可以从外部到达（内部的语句不会被外部跳转到达）
  - 只要基本块的第一句被执行，那么基本块内的其他语句都可以被执行（只有最后一句才会跳出基本块）
  - 我们需要最大的基本块！
- 控制流图
  - 节点：基本块
  - 边：$B_i \rightarrow B_j$ 当切仅当 B_j 在某些执行下紧跟在B_i之后
    - 要么B_j的第一条指令是B_i最后一句跳转的目标
    - 要么B_j就是天然的跟在B_i后面，其中B_i不是以无条件跳转结尾

构建基本块

- 找到基本块的leader：
  - 第一条指令
  - 一个跳转指令的目标
  - 跳转指令的下一个指令
- 基本块从一个leader开始，终止于某个leader的上一个指令（或者最后一个指令）

```
    i := n-1
L5: if i<1 goto L1
    j := 1
L4: if j>i goto L2
     t1 := j-1
     t2 := 4*t1
     t3 := A[t2]     ;A[j]
     t6 := 4*j
     t7 := A[t6]     ;A[j+1]
     if t3<=t7 goto L3
     A[t2] := t7     ;A[j]:=A[j+1]
     A[t6] :=t3      ;A[j+1]:=old_A[j]
L3: j := j+1
    goto L4
L2: i := i-1
    goto L5
L1:
```

控制流图



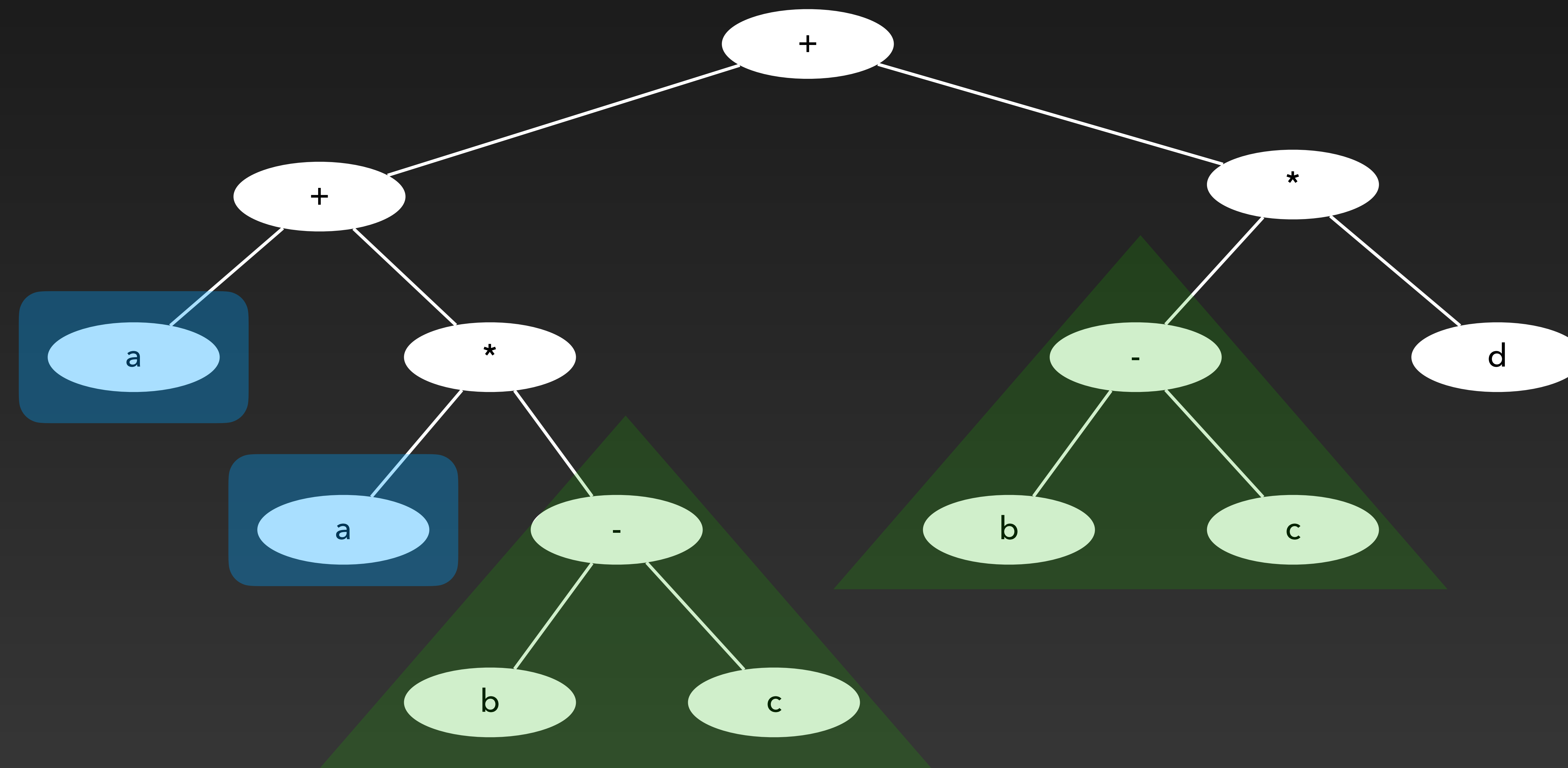
本地优化

在基本块内

- 公共子表达式（Common Subexpression）消除
  - Array expressions
  - Filed access in records
  - Access to parameters

图抽象

- 例子：
  - 文法: $E \to E + T \mid E - T \mid T, T \to T * F \mid F, F \to (E) \mid \text{id}$
  - expression: a + a * (b - c) + (b - c) * d



图抽象

- 例子：
  - 文法: $E \rightarrow E + T \mid E - T \mid T,\ T \rightarrow T * F \mid F,\ F \rightarrow (E) \mid \text{id}$
  - expression: a + a * (b - c) + (b - c) * d

Parse Tree

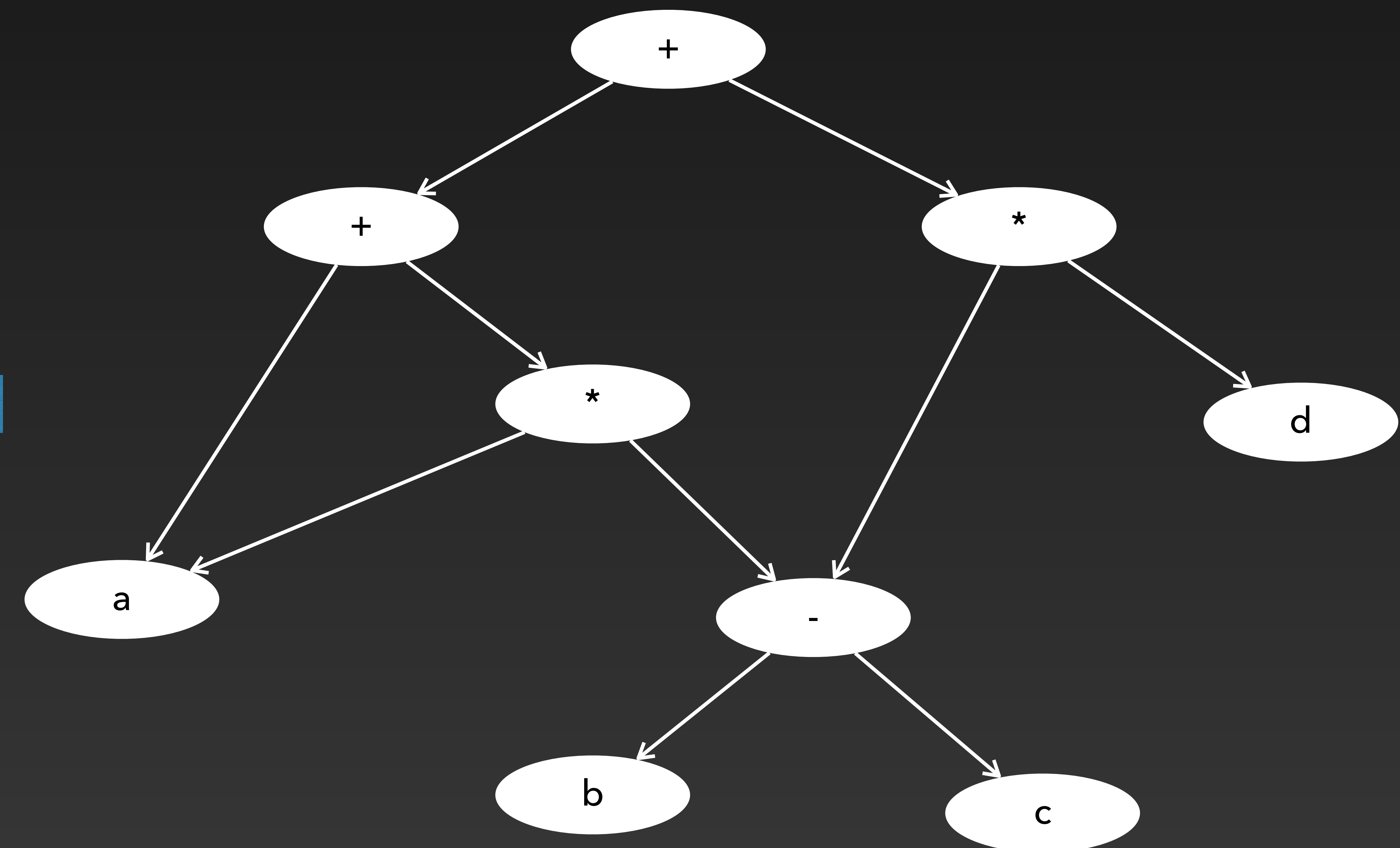
Expression DAG

图抽象

- expression: a + a * (b - c) + (b - c) * d
- Optimized code:

```
t1 = b − c
t2 = a ∗ t1
t3 = a + t2
t4 = t1 ∗ d
t5 = t3 + t4
```

基于有向无环图的拓扑序



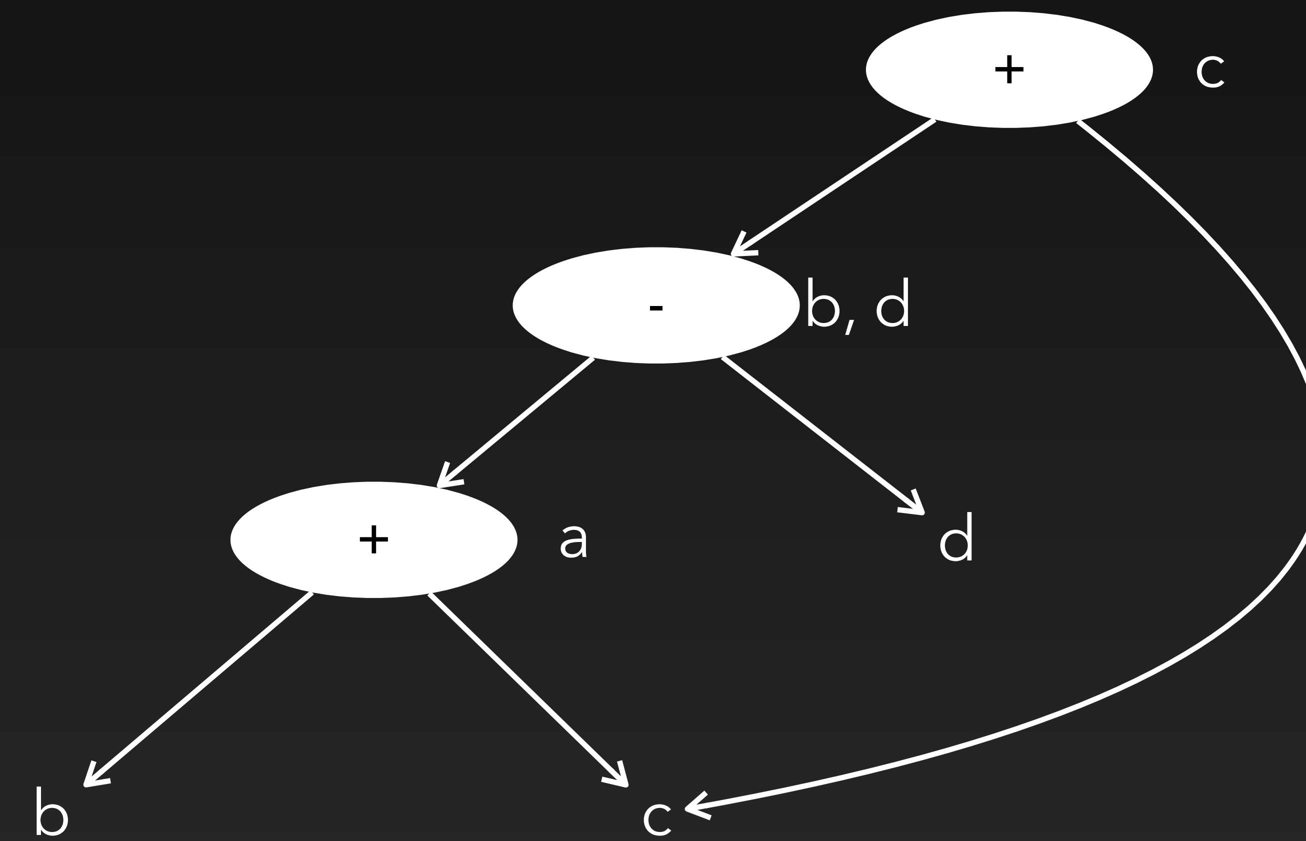
但如果是语句？DAG能运用在多条语句吗？

- 例子：

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

可以优化为下面吗？

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



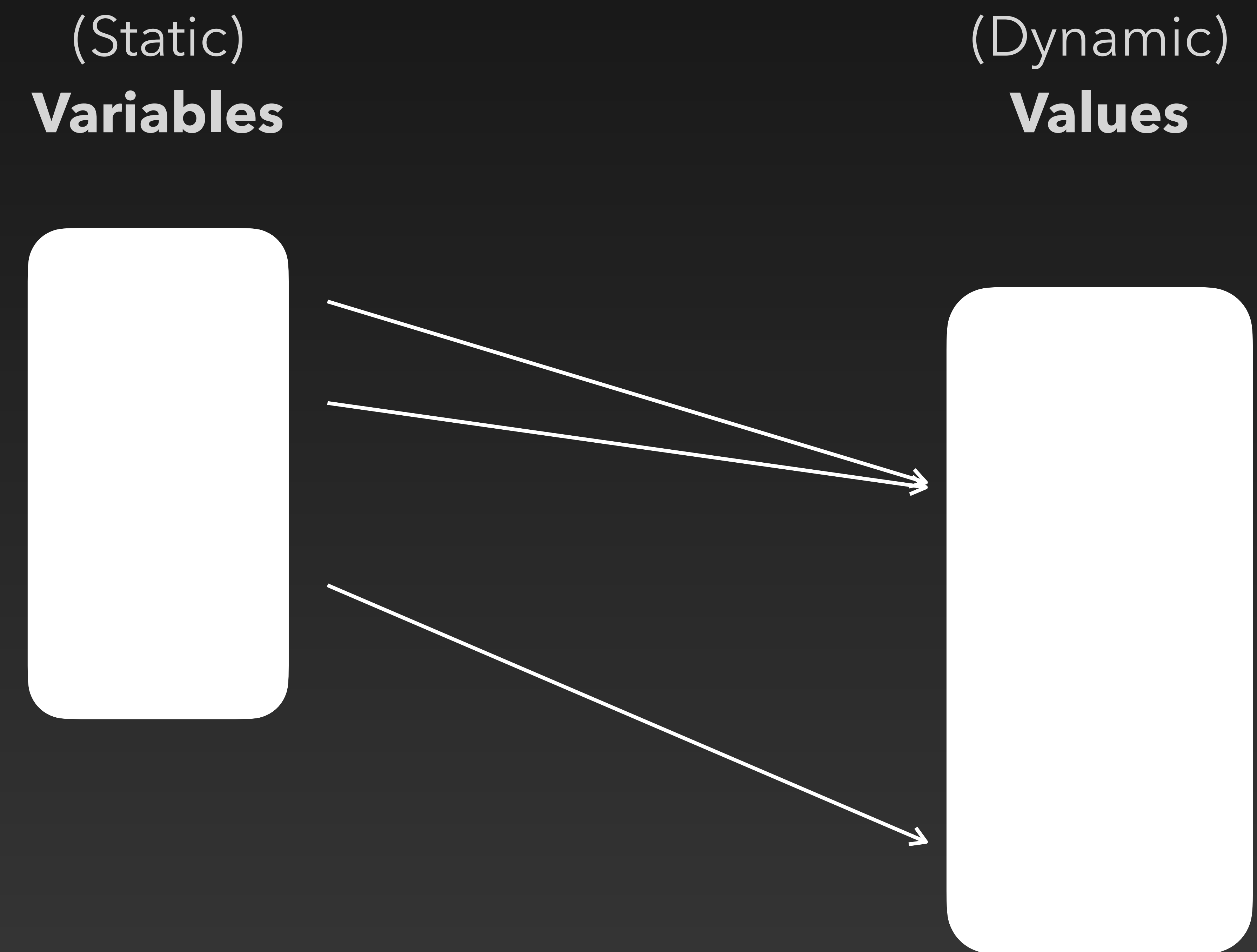
取决于b是否在该基本块外面还需要用到

DAG的局限

- 问题的起因：
  - 赋值语句会改变变量的值
  - 变量的值和具体出现的时间相关
- 如何修复这个问题
  - 建立一个和执行顺序相关的图
  - 将变量名和latest的赋值相关联
- 随之而来的问题是，最终构建的图信息太多，缺失了我们一开始对图这个的“抽象”需求

另一个抽象：值编号（Value Number）

- 值编码将Values和时间显式地进行关联
- 每个值由自己的编号
  - 公共子表达式意味着相同的值编号
- **var2value**: 将变量映射到当前值的函数
  - 基于此可以进一步获得表达式的编号
    - r1 + r2 -> var2value(r1) + var2value(r2)



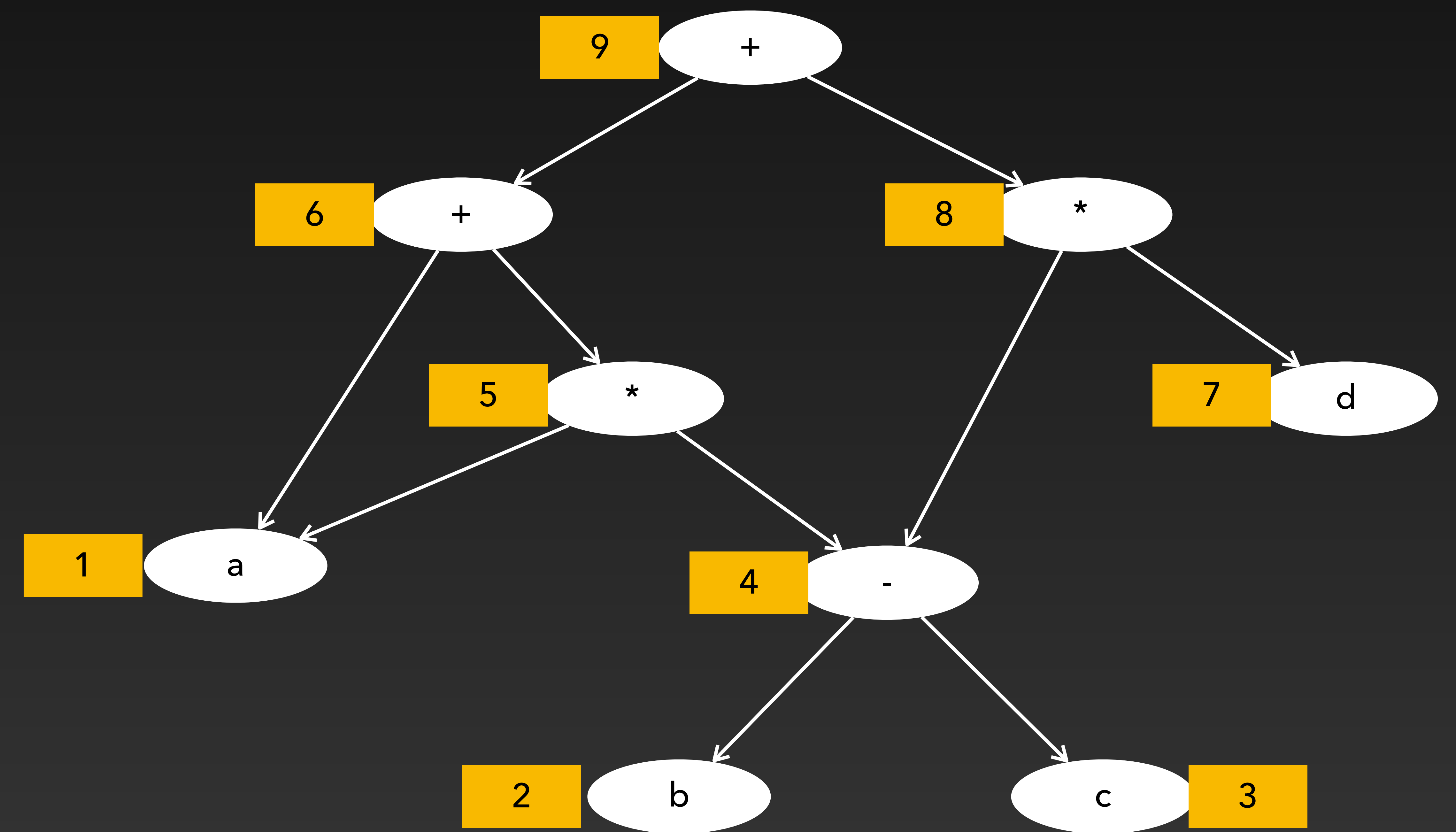
概念来自： John Cocke & Jack Schwartz的没有正式出版的书: “Programming Languages and their Compilers”, (1970)

值编号

例子

- 表达式：
  - a + a * (b - c) + (b - c) * d
- 优化后的代码

```
t4 = b − c
t5 = a ∗ t4
t6 = a + t5
t8 = t4 ∗ d
t9 = t6 + t8
```



值编号算法

```
Data structure:
  VALUES = Table of
    expression /* [OP, valnum1, valnum2] */
    var        /* name of variable currently holding expr */

For each instruction (dst = src1 OP src2) in execution order
    valnum1=var2value(src1); valnum2=var2value(src2)

    IF [OP, valnum1, valnum2] is in VALUES
         v = the index of expression
         Replace instruction with: dst = VALUES[v].var
    ELSE
         Add
            expression = [OP, valnum1, valnum2]
            var = tv
         to VALUES
         v = index of new entry; tv is new temporary for v
         Replace instruction with: tv = VALUES[valnum1].var OP VALUES[valnum2].var
         dst = tv


    set_var2value (dst, v)
```

更多细节

- 这些变量的初始值是多少?
  - values at beginning of the basic block
- 可能的实现
  - Initialization: create “initial values” for all variables
  - Or dynamically create them as they are used
- 数据结构**Values** 和映射函数 **var2value**的实现: **hash table**

值编号

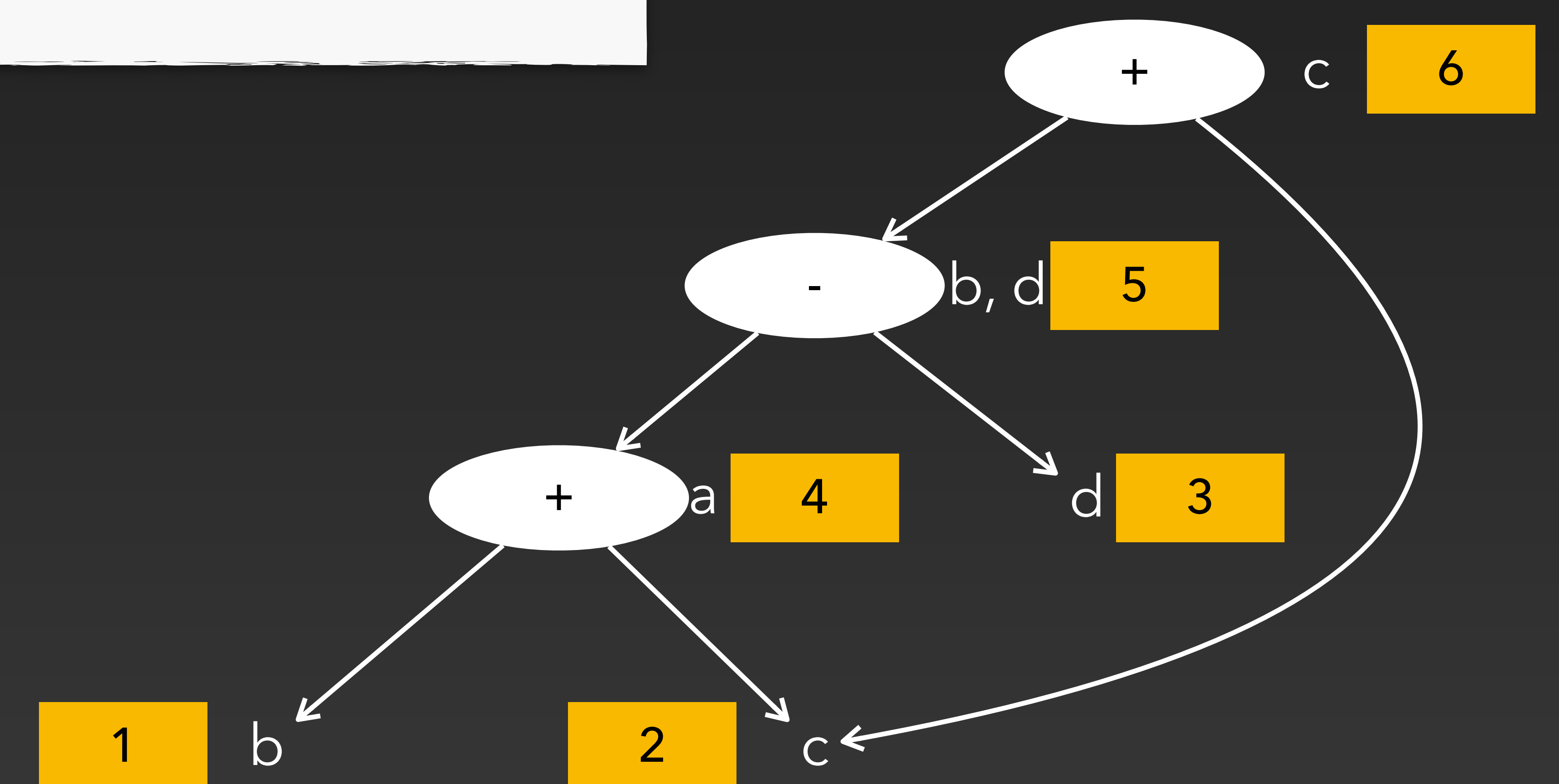
```
a = b + c

b = a - d

c = b + c

d = a - d
```

```
t4 = b + c  // tv = VALUES[valnum1].var OP VALUES[valnum2].var
a  = t4     // dst = tv
t5 = t4 - d
b = t5
t6 = t5 + c
c = t6
d = t5    // dst = VALUES[v]. var
```



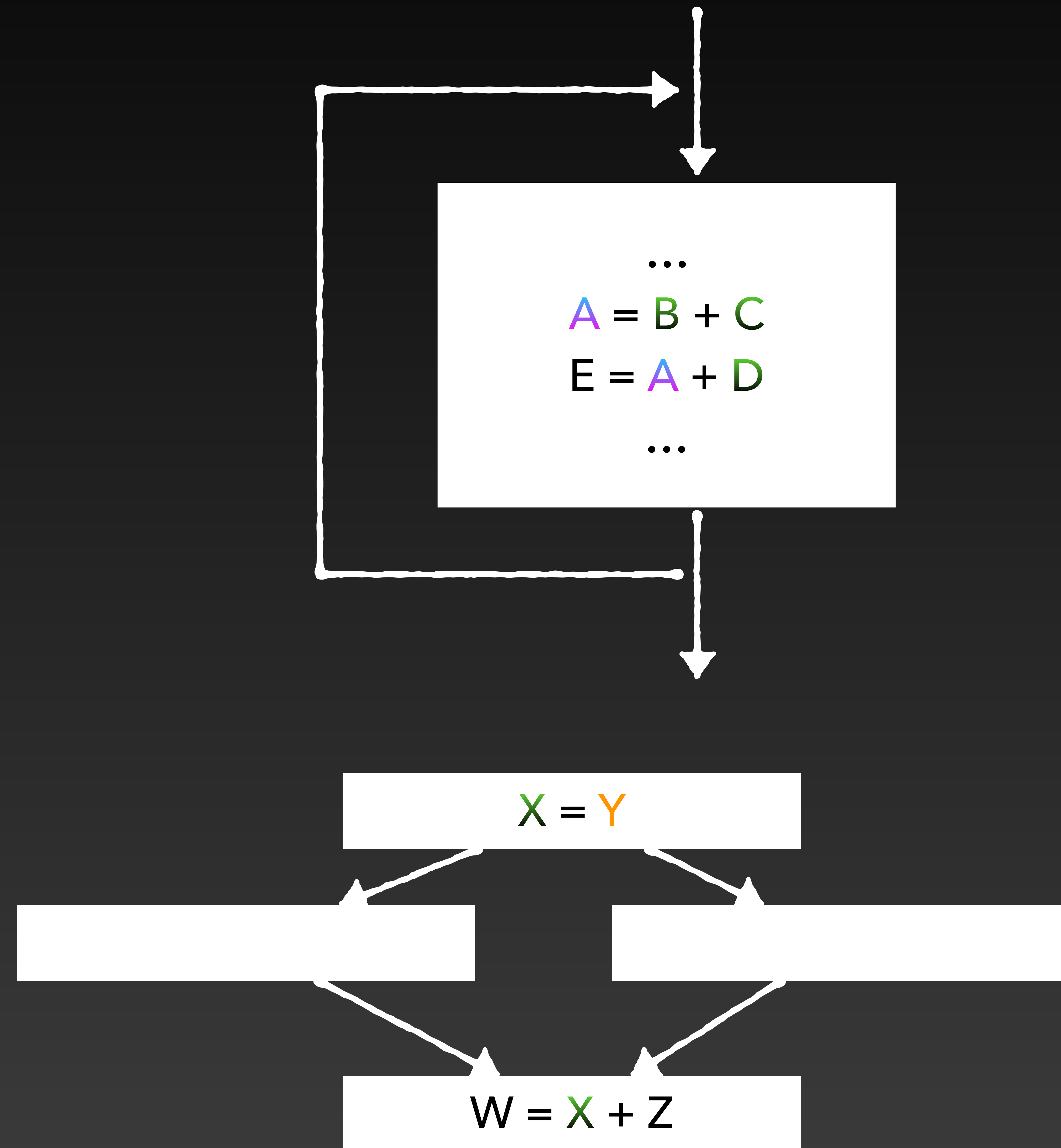
SSA简介

全局优化

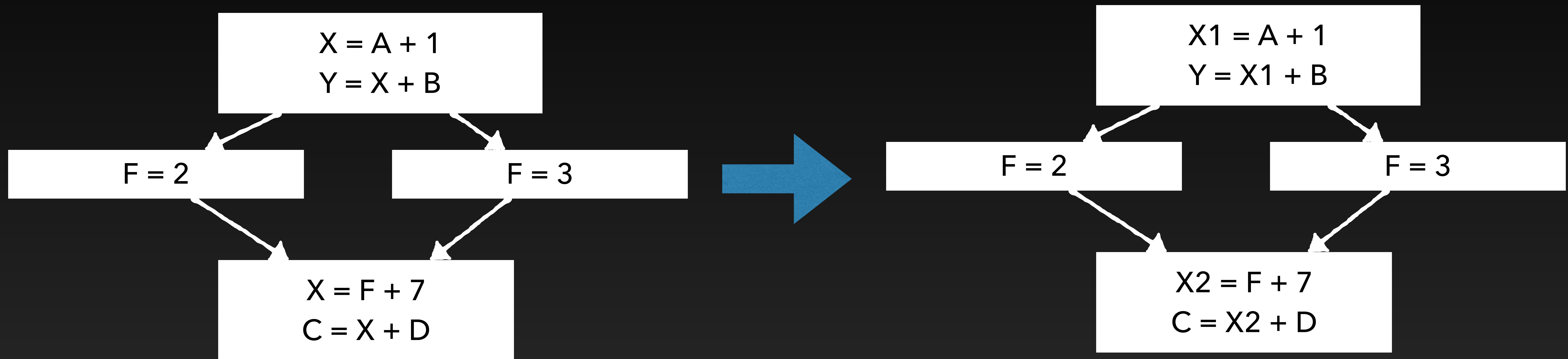
- 全局优化：跨多个block的优化
  - 全局版本的“本地优化”
    - 全局版本的公共子表达式消除
    - 全局版本的常量传播
    - 死代码消除
  - 循环优化
    - 减少每次迭代需要的代码
    - 代码移动（比如循环不变式外提）
    - 归纳变量消除

优化中非常重要的概念： def and use

- 例子1：循环不变式外提（Loop-Invariant Code Motion）
  - B, C, 和 D 是否只在循环外定义?
  - A在循环内部有没有其他的定义?
  - A在循环部分的使用?
- 例子2：拷贝传播(Copy Propagation)
  - 对于一个X的使用点：
    - 是否所有可达的X定义点都使用了同样的拷贝？（比如Y？）
    - 期间Y是否没有被重新定义过?
    - 如果是，就可以将X替换为Y



相同的变量名可能完全无关



- 使用同一个变量的可能完全是独立的值
- 编译器一般会对这些变量进行重新署名来区分

Def-Use (DU) 和 Use-Def (UD) 链

- 一个常见的分析方式：可以遍历变量的定义使用对
  - 这样可以让我们只关心变量的变动，而免去一些不重要的点（Sparse code analysis）
  - Def-Use chain
    - 给定一个变量X的定义点，其所有的使用点在哪里?
  - Use-Def chain
    - 给定一个变量X的使用点，其所有能到达这个使用点的定义有哪些，在哪里?

DU和UD的获得往往很昂贵

```
foo(int i, int j) {
  …
  switch (i) {
  case 0: x=3;break;
  case 1: x=1; break;
  case 2: x=6; break;
  case 3: x=7; break;
  default: x = 11;
}
switch (j) {
  case 0: y=x+7; break;
  case 1: y=x+4; break;
  case 2: y=x-2; break;
  case 3: y=x+1; break;
  default: y=x+9;
}
…
```

对于有N个defs和M个uses，需要O(MN)复杂度

- 一个解决办法：限制每个变量只有一个定义点（defintion site）

限制每个变量只有一个定义点

```
foo(int i, int j) {
  …
  switch (i) {
  case 0: x=3;break;
  case 1: x=1; break;
  case 2: x=6; break;
  case 3: x=7; break;
  default: x = 11;
}
switch (j) {
  case 0: y=x+7; break;
  case 1: y=x+4; break;
  case 2: y=x-2; break;
  case 3: y=x+1; break;
  default: y=x+9;
}
…
```

```
foo(int i, int j) {
  …
  switch (i) {
  case 0: x=3;break;
  case 1: x=1; break;
  case 2: x=6; break;
  case 3: x=7; break;
  default: x = 11;
}
x1 is one of the above x's
switch (j) {
  case 0: y=x1+7; break;
  case 1: y=x1+4; break;
  case 2: y=x1-2; break;
  case 3: y=x1+1; break;
  default: y=x1+9;
}
…
```

静态单赋值形式 (Static Single Assignment, SSA)

- 静态单赋值形式 (Static Single Assignment, SSA)是一种中间码形式，其限制每个变量至多只被赋值一次
- 对于基本块是容易实现的
  - 对每个指令进行遍历
    - 赋值操作左端(LHS): 对一个变量给赋予一个没有使用过的名字(fresh version)
    - 赋值操作左端(RHS): 使用变量的最近的version

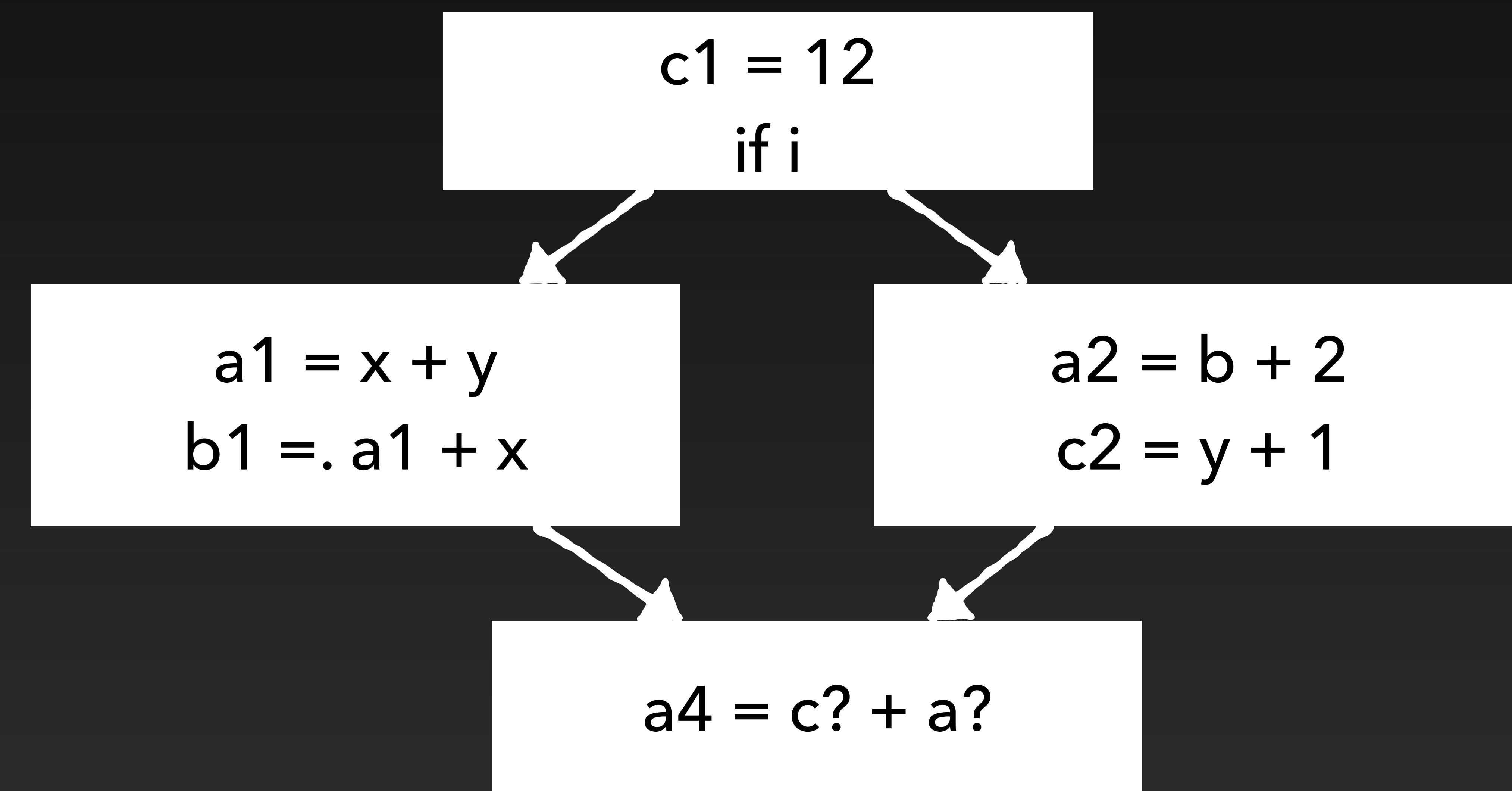
```
a = x + y
b = a + x
a = b + 2
c = y + 1
a = c + a
```

```
a1 = x + y
b1  = a1 + x
a2 = b1 + 2
c1 = y + 1
a3 = c1 + a2
```

类似Value Numbering

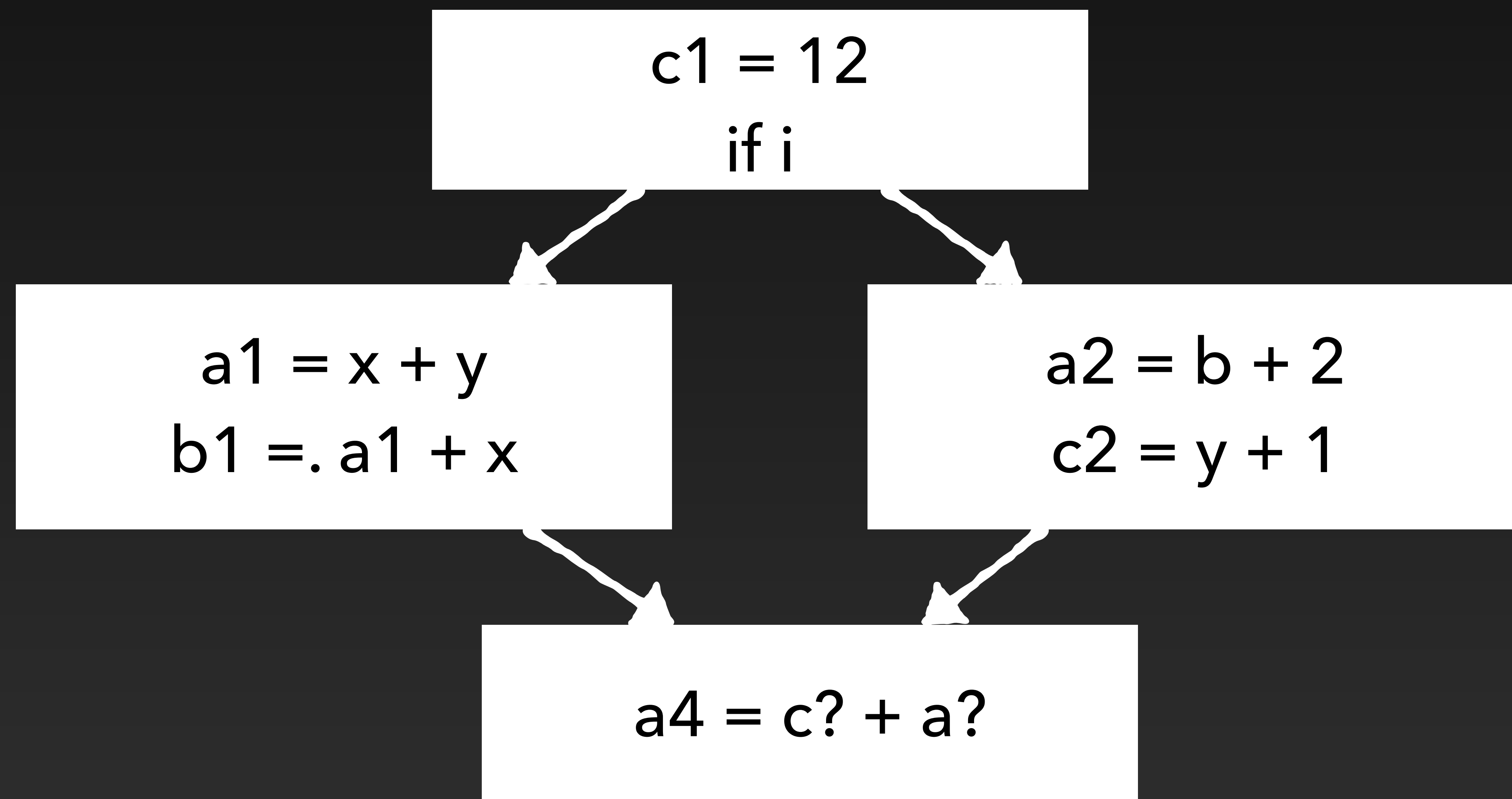
但如果是控制流图中的汇聚(joins)?

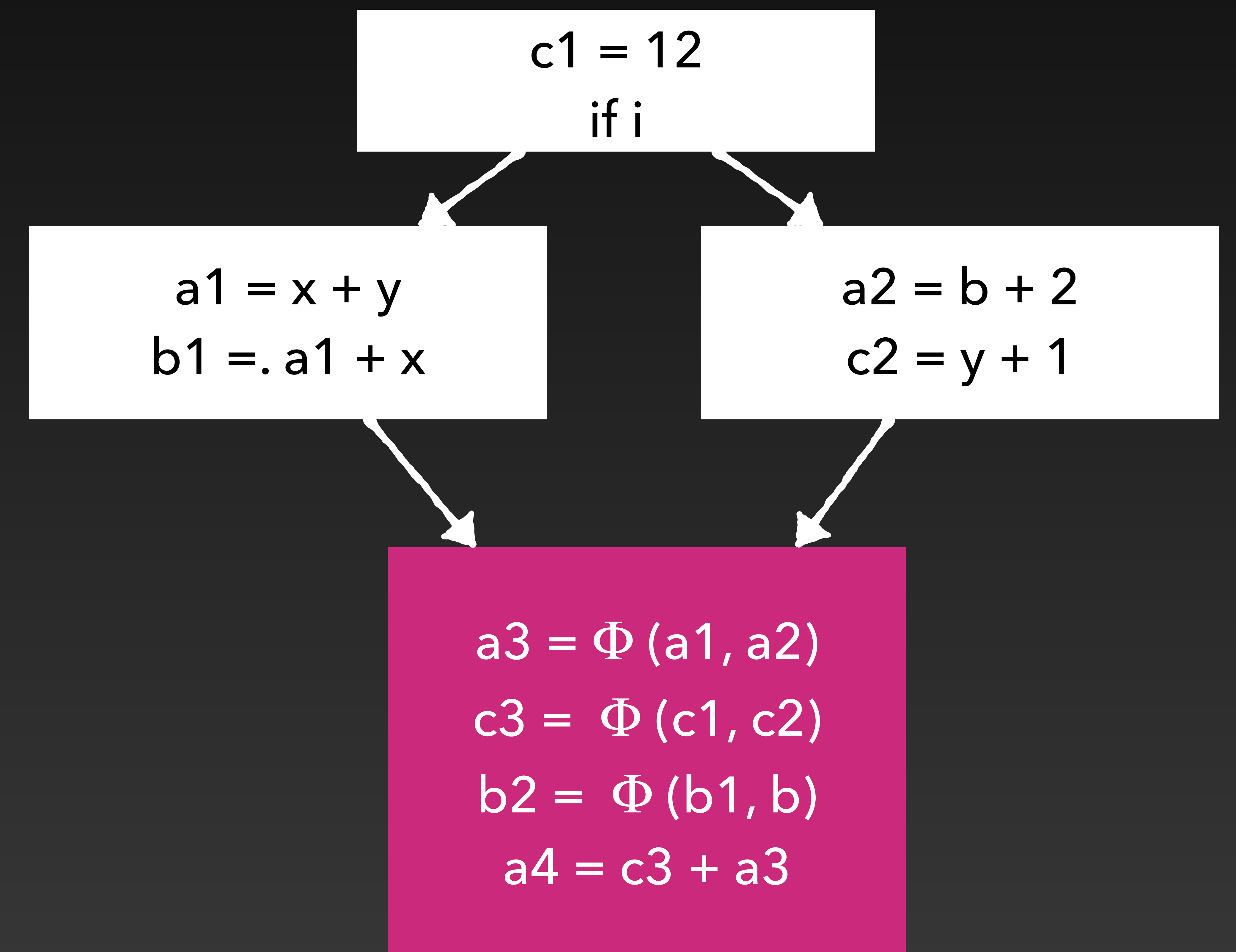
```
c = 12
if (i) {
  a = x + y
  b = a + x
} else {
  a = b + 2
  c = y + 1
}
a = c + a
```



Use a notational convention : a Φ funciton

合并汇聚： Φ 函数

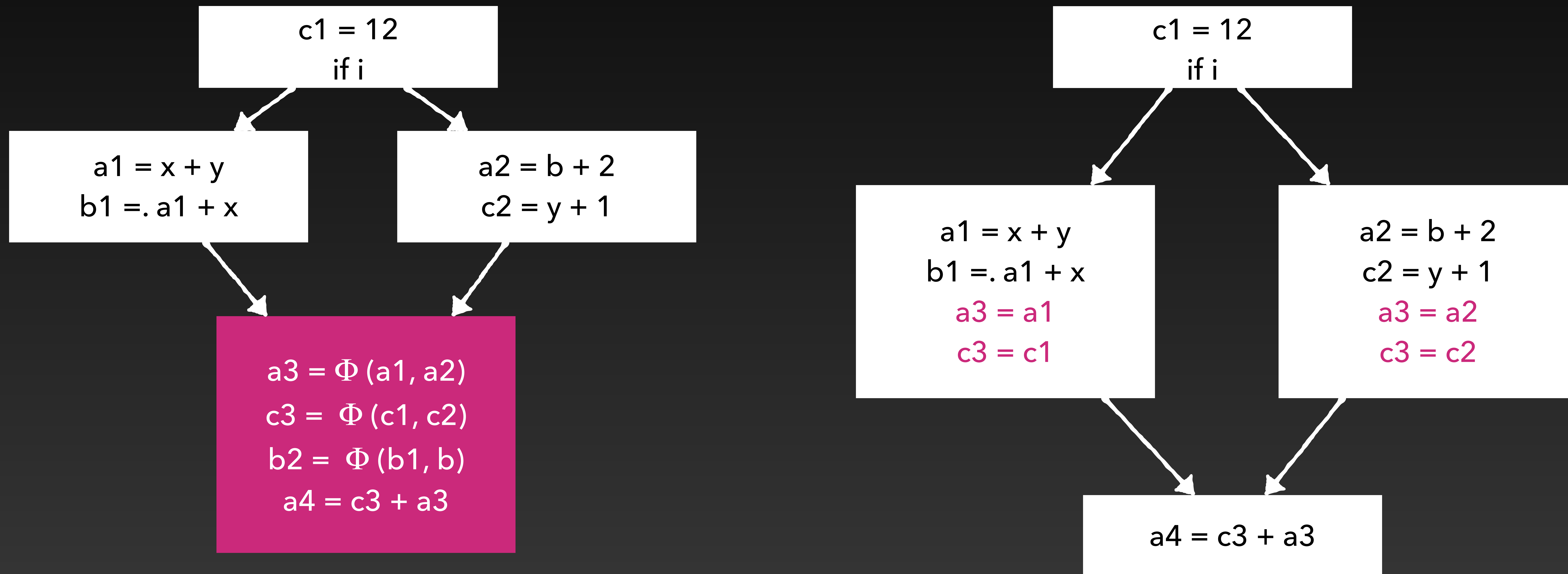




合并汇聚：Φ 函数

- Φ 函数将沿着控制流路径下的多个定义合并为一个单独的定义
- 对于个一个有p个前驱的基本块，Φ 函数就有p个参数
  - $x_{new} = \Phi(x_1, x_2, \ldots, x_p)$
- 选择哪个x_i?
  - 无需关心，由运行时决定
- 问题是，怎么生成这样的代码，使的其可以运行时判断?

Φ 函数的“实现”



注：这只是Φ的示意，实际的运行代码不需要

简单的SSA实现

- 对每一个赋值都给一个新的名字
- 每个join point都为那些活跃变量（live variables）插入Φ函数

需要插入太多的Φ函数

Minimal SSA

- 对每一个赋值都给一个新的名字
- 每个join point都为那些存在多个defs活跃变量（live variables）插入Φ函数

Q&A