第十章 静态单赋值形式
Static Single Assignment Form

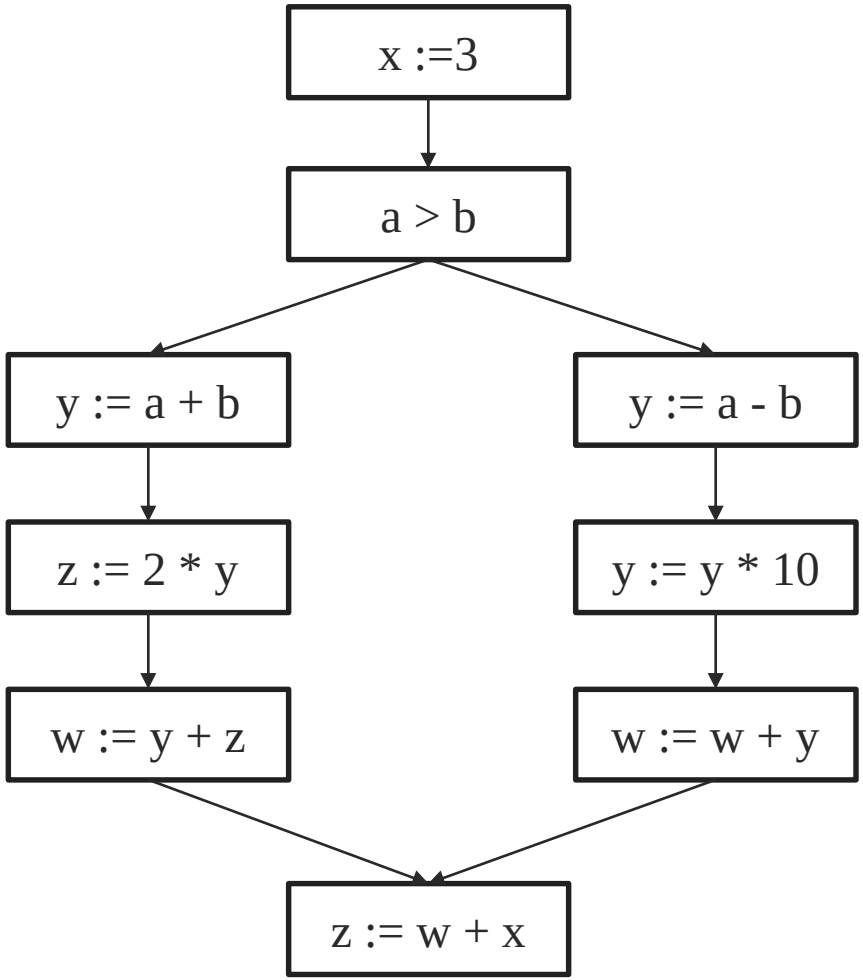
冯 洋

为什么要有静态单赋值？

- 数据流分析通常需要记录程序在任意程序点上各个变量的状态

- 那么，如果。。。
  - 有很多变量
  - 并且，有很多程序点呢？

来一个例子？



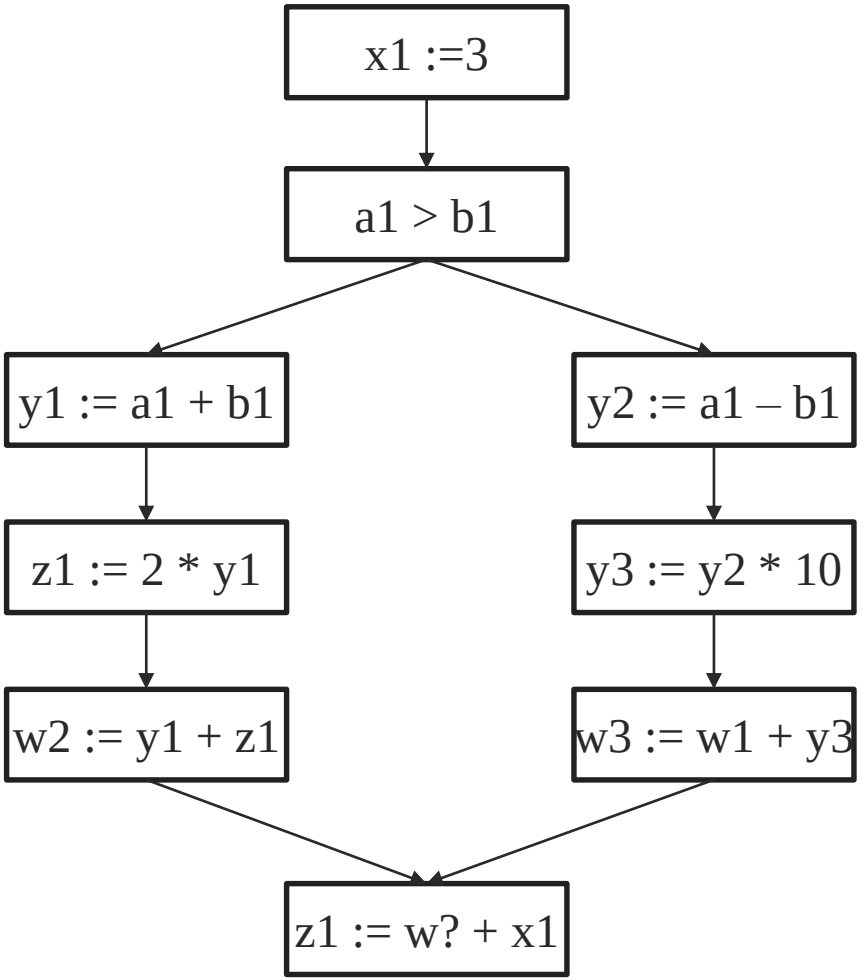
稀疏表示

- 我们使用的稀疏表示形式？How？
  - 我们只在 x 被使用的才记录？

- 使用SSA（Static Single Assignment）形式
  - 每个变量，只被定义一次
  - 但是可以被使用很多次！

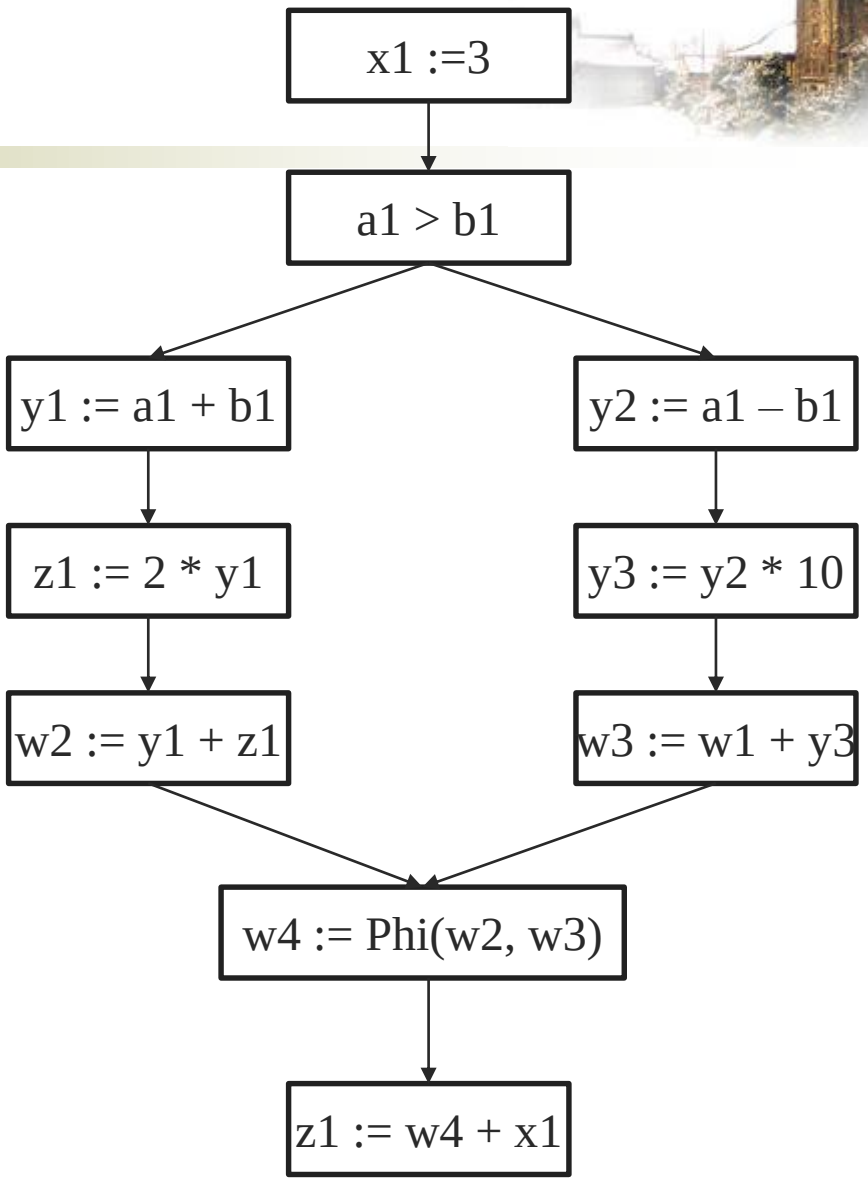
静态单赋值形式

- 添加从x定义到使用的边
  - def-use
  - 注意，是从使用，到上一次复制之间的关联
  - 通过SSA-edge，可以保证所有的变量在程序点上的分析均是安全的



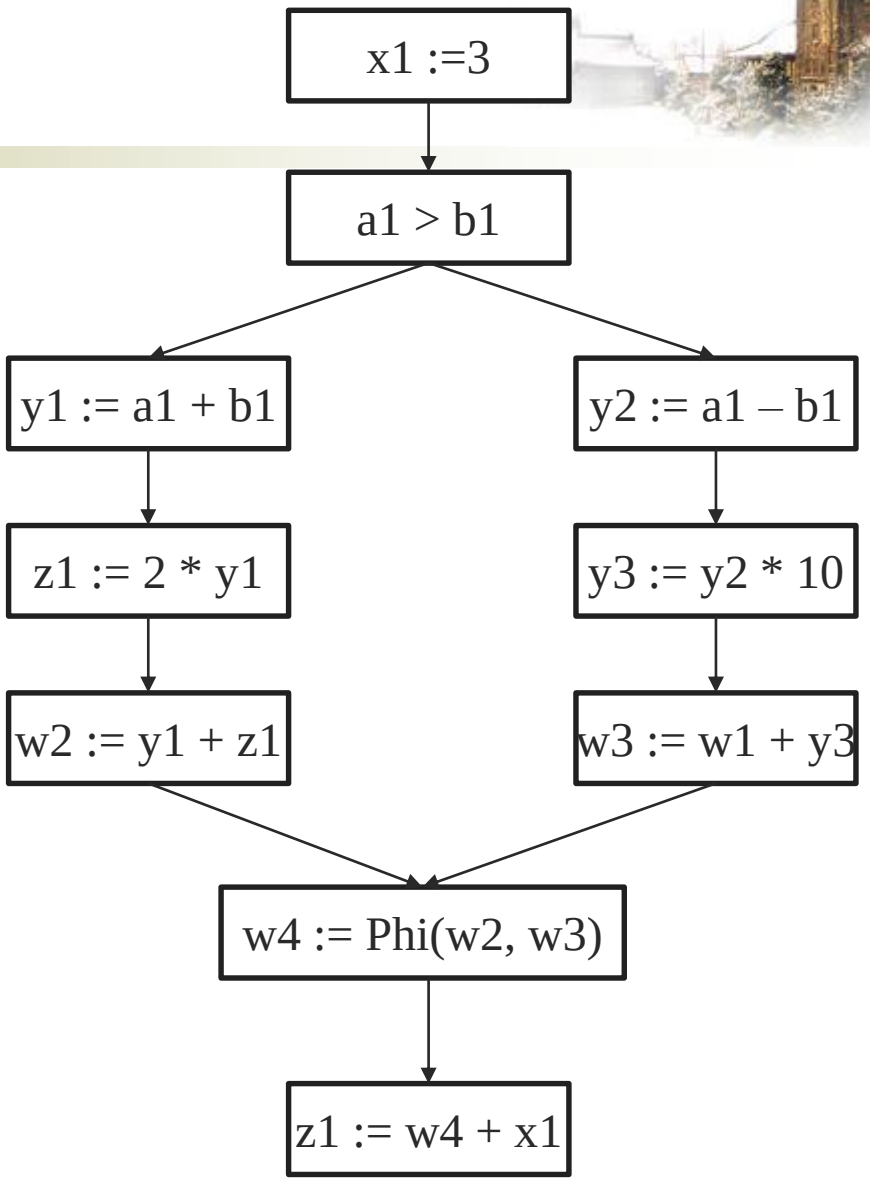
静态单赋值形式

- 我们通过添加 Φ 函数用以模拟join 操作
- 什么是 Φ 函数？
  - 根据Kenny Zadeck的说法，Φ函数最初被称为伪函数，而SSA是在20世纪80年代在IBM Research开发的。Φ功能的正式名称仅在首次发表在学术论文中时才被采用

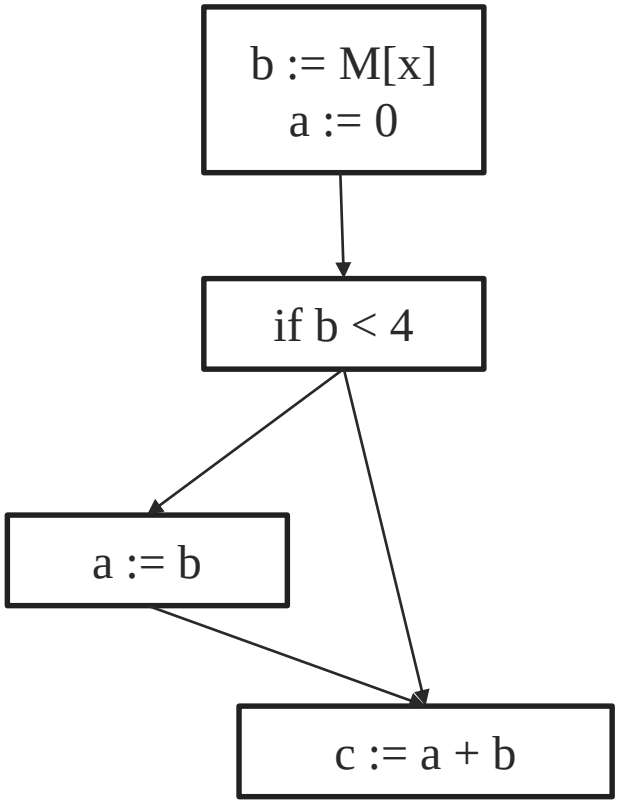


静态单赋值形式

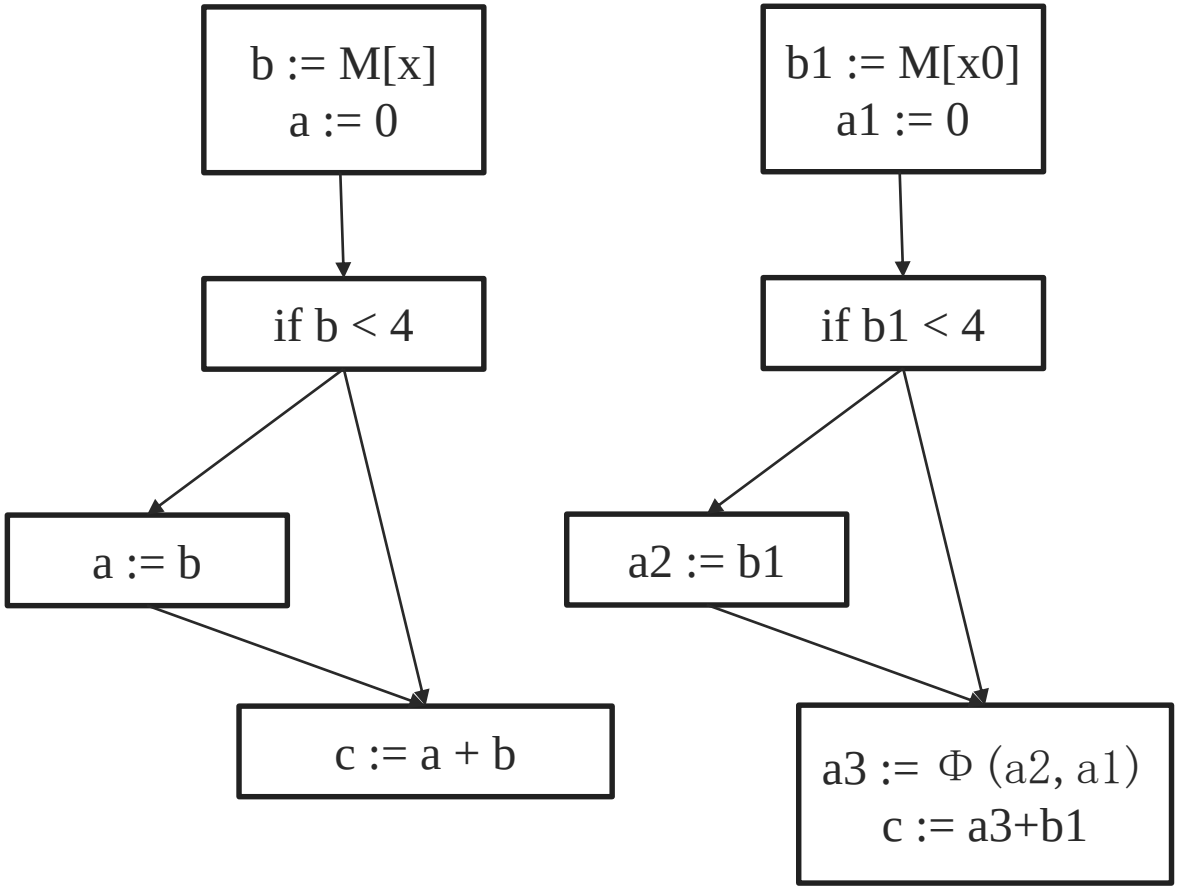
- 我们通过添加 Φ 函数用以模拟join 操作
- 什么是 Φ 函数?
  - 有若干个参数，每个参数来源于一个分支的出口
  - 根据动态行为确定选择哪个参数对变量进行赋值
  - 在代码生成阶段，需要“消灭” Φ 函数



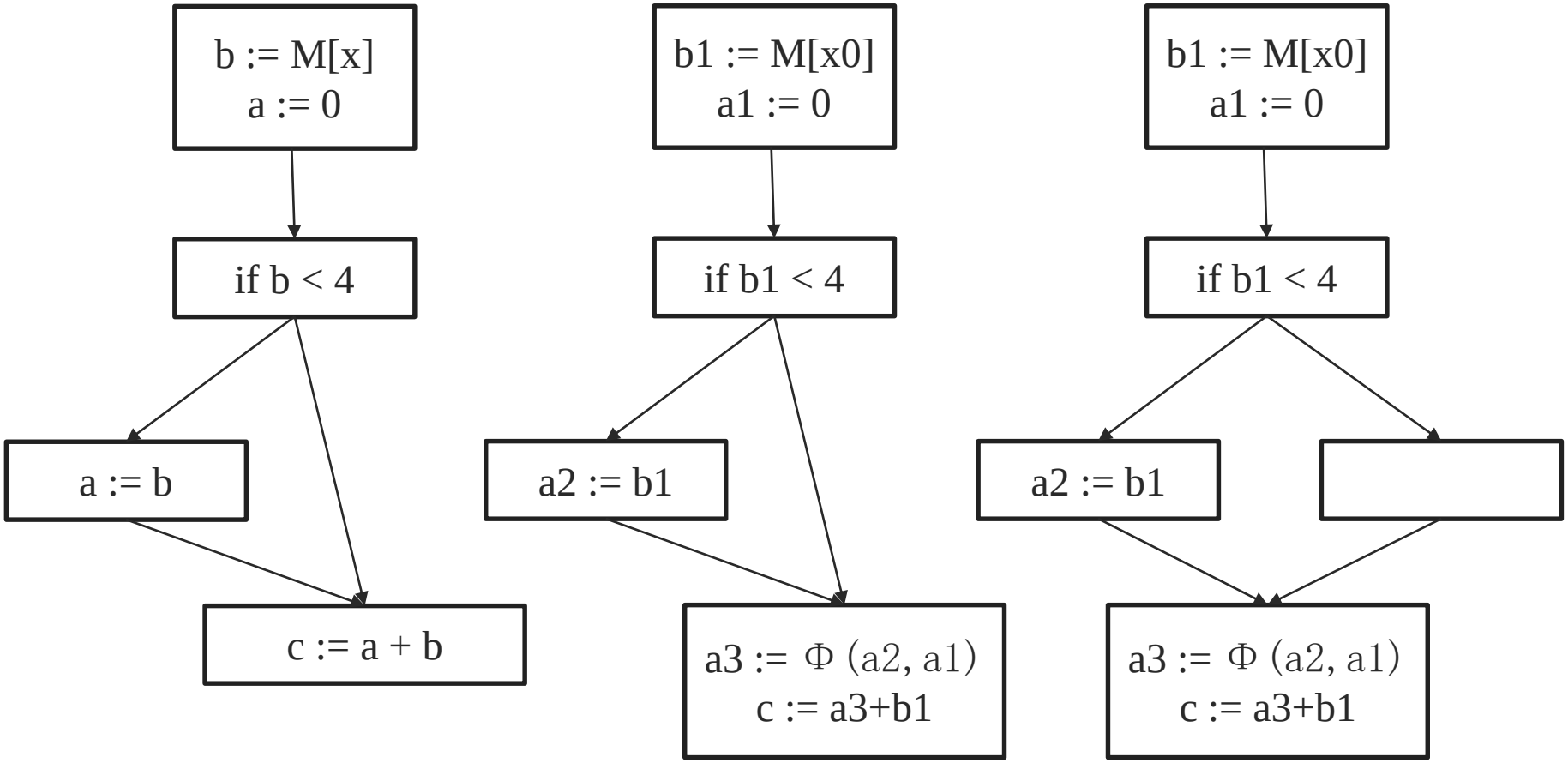
其他例子？



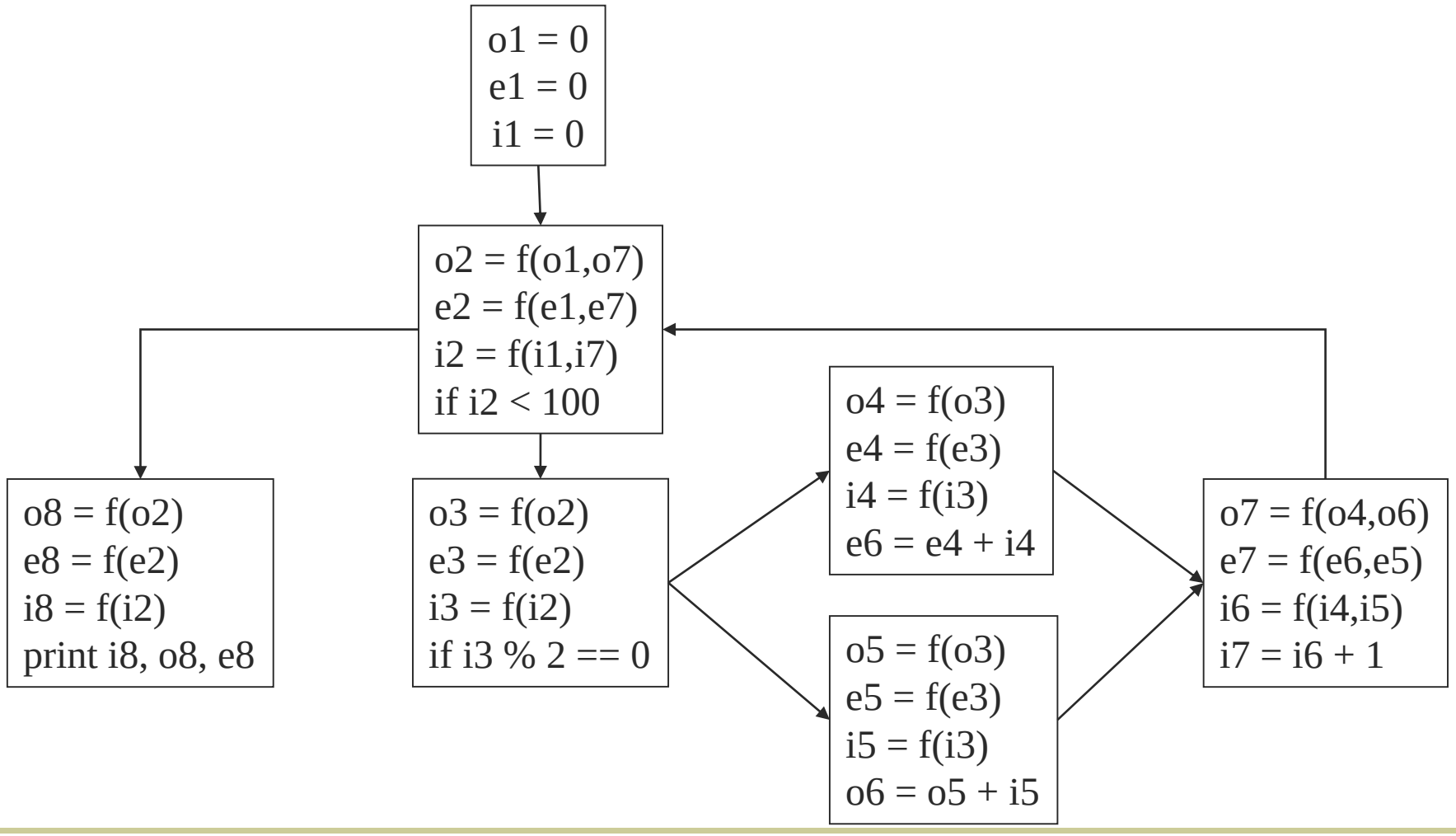
其他例子?



其他例子？


b := M[x]
a := 0
if b < 4
a := b
c := a + b
b1 := M[x0]
a1 := 0
if b1 < 4
a2 := b1
a3 := Φ (a2, a1)
c := a3+b1
b1 := M[x0]
a1 := 0
if b1 < 4
a2 := b1
a3 := Φ (a2, a1)
c := a3+b1

其他例子？


o1 = 0
e1 = 0
i1 = 0
o2 = f(o1,o7)
e2 = f(e1,e7)
i2 = f(i1,i7)
if i2 < 100
o8 = f(o2)
e8 = f(e2)
i8 = f(i2)
print i8, o8, e8
o3 = f(o2)
e3 = f(e2)
i3 = f(i2)
if i3 % 2 == 0
o4 = f(o3)
e4 = f(e3)
i4 = f(i3)
e6 = e4 + i4
o5 = f(o3)
e5 = f(e3)
i5 = f(i3)
o6 = o5 + i5
o7 = f(o4,o6)
e7 = f(e6,e5)
i6 = f(i4,i5)
i7 = i6 + 1

Phi函数的特点

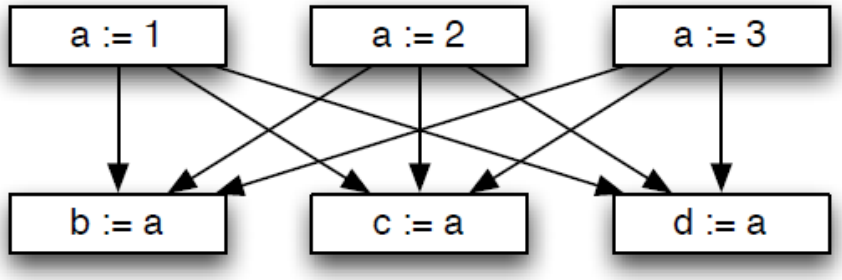
- 我们通过添加 Φ 函数用以模拟join 操作
- 必须执行该程序，或者将该程序翻译为可执行形式，那么我们可以通过 MOVE 指令来“实现” Φ 函数
- 许多情况下，我们只需要知道use-def之间的关系，而不需要“执行” Φ 函数，在这些情况下，可以忽略到底产生哪一个值的问题

Def-Use Chains vs. SSA

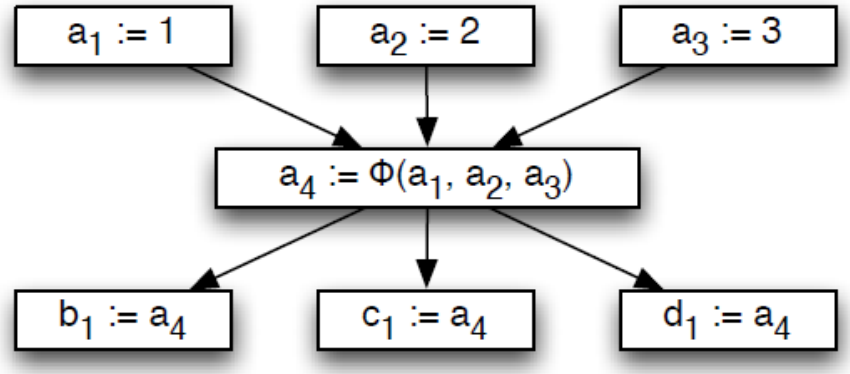
- Quadratic（二次方）

vs. Linear behavior

（实践中，通常是线性关系）



def-use chain



SSA Form 形式

重新审视常量传播

- 只要有 v ← c 的语句，那么我们只需要用 c 替换所有的 v
- 那么，对于任意 v ← Φ(c1, c2, …, cn) 若其中的ci 全部相等，那么可以用 v ← c 代替Φ

SSA Form 的计算

- Step1：我们需要先找到，哪些Node需要放置Φ
  - 一个解决方案：在每个基本块开始出，均放置Φ函数？

SSA Form 的计算

- Step1：我们需要先找到，哪些Node需要放置Φ

  一个解决方案：在每个基本块开始出，均放置Φ函数（请思考：该方案可行吗？）

SSA Form 的计算

- Step1：我们需要先找到，哪些Node需要放置Φ

~~一个解决方案：在每个基本块开始出，均放置Φ函数？~~

- Step2：重命名变量，这样每个变量只会被定义一次

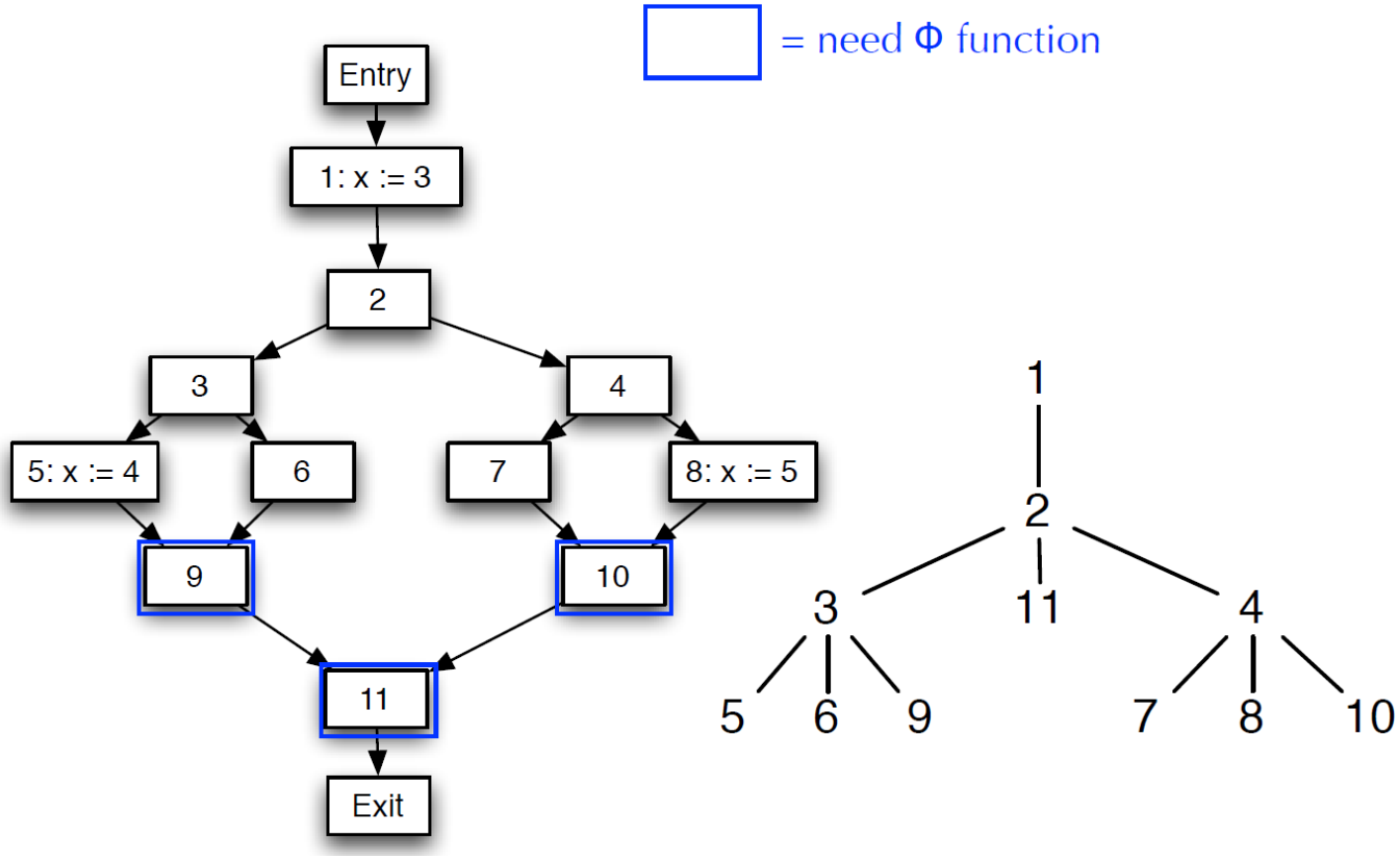
SSA Form 的计算

- Step1：我们需要先找到，哪些Node需要放置Φ

  ~~一个解决方案：在每个基本块开始出，均放置Φ函数？？？？？？？~~

  - Step1a：计算Dominance Frontier（DF）
  - Step1b：通过DF节点来放置Φ
    - 可能存在一个问题？如果节点 X 包含一个变量a，为a在X中放置Φ，那么，添加Φ会不会引入额外的Φ添加需求？
- Step2：重命名变量，这样每个变量只会被定义一次

SSA Form 的计算

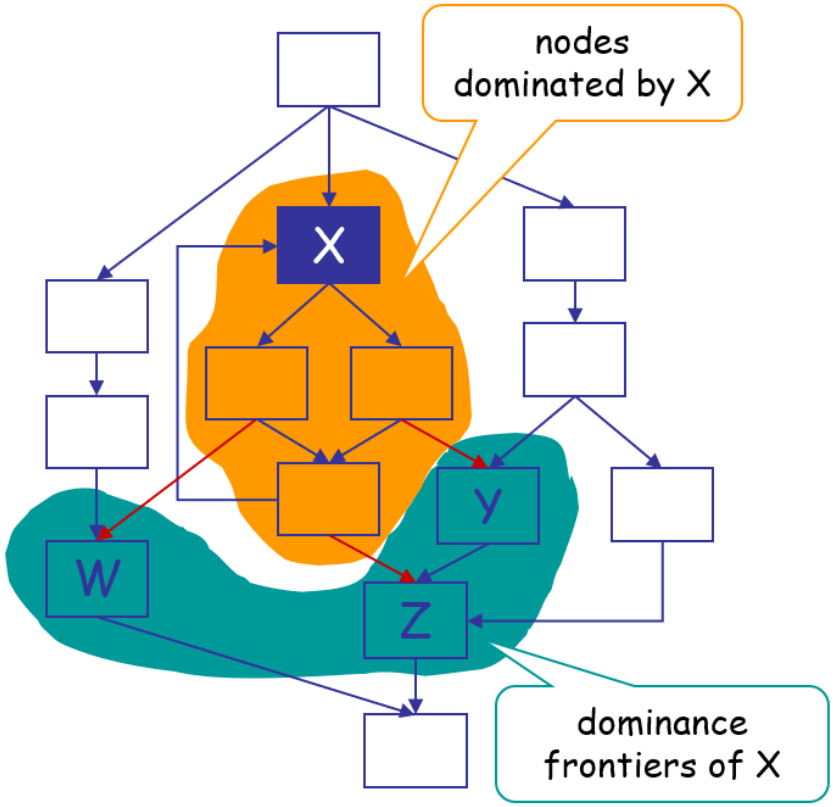

= need Φ function
Entry
1: x := 3
2
3
4
5: x := 4
6
7
8: x := 5
9
10
11
Exit
1
2
3
11
4
5
6
9
7
8
10

SSA Form 的计算

- 控制流图上的节点X的 *Dominance frontier* 定义：
  - The set of all CFG nodes Y such that X dominates a predecessor of Y but does not strictly dominate Y
- X的Dominance Frontiers 形成了X可以dominate和不可以dominate的边界

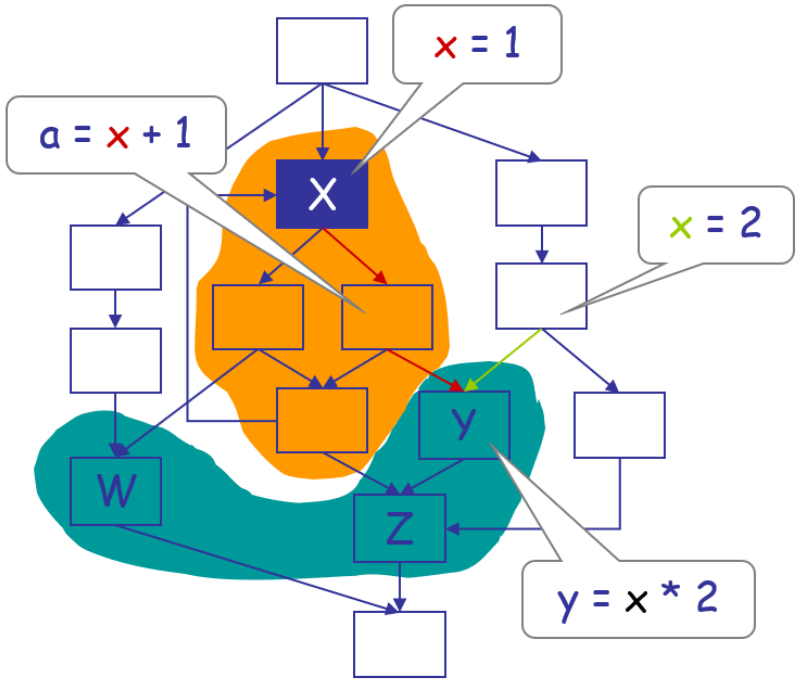
SSA Form 的计算

- 控制流图上的节点X的 *Dominance frontier* 定义：
  - The set of all CFG nodes Y such that X dominates a predecessor of Y but does not strictly dominate Y
- X的Dominance Frontiers 形成了X可以dominate和不可以dominate的边界



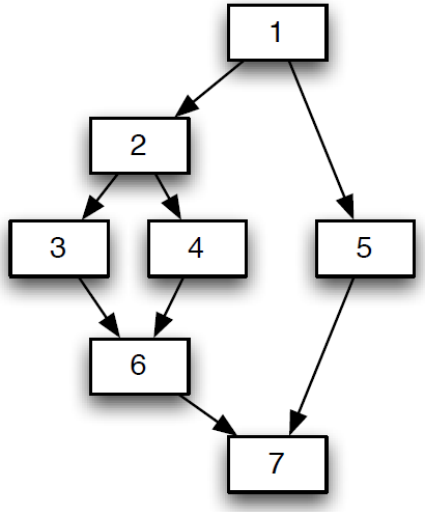
SSA Form 的计算

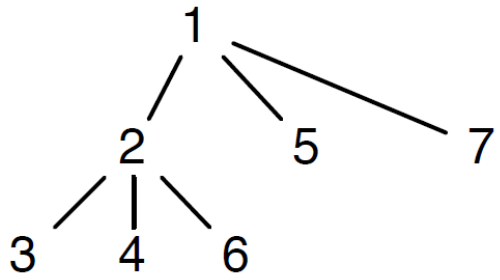
- X中定义的变量，可能在DF中与其他*节点*的同名变量，*汇合*
- 我们需要在DF(X)的节点中，对X中的定义的变量，设置Φ函数



SSA Form 的计算

- Dominator Tree（DT）
- Dominator的关系可以构成一个树形结构！
- 树上的父子节点关系可以表示Dominate关系
- 注意，DT上的边，不表示是CFG上的边





SSA Form 的计算

- R. Cytron 等人发表了一篇通过DF快速计算将源程序转化为SSA格式的论文
  - For each variable v
    - Place f-function for v at each nodes in DF+ of nodes defining v where DF+(S) = least fix point (IT.DF(T) U S) {}
  - Traversing the dominator tree in pre-order, for each node X using the parent's last map(*)
    - For each assignments in X
      - Rename used variables v --> $v_{map(v)}$
      - Rename defined variables u --> $u_{count(u)}$
      - replace map(u) = count(u) and count(u) = count(u) + 1
    - For each successor Y of X, and for each f-function for v in Y
      - Replace $v_{map(v)}$ for corresponding argument

SSA Form 的计算

- 另外一篇有意思的论文：
- Cooper, Keith D., Timothy J. Harvey, and Ken Kennedy. "A simple, fast dominance algorithm." *Software Practice & Experience* 4, no. 1-10 (2001): 1-8.
  - 论文主旨，计算DT 的算法时间复杂度理论上是O(N2)，但是实际运行中要远远快过理论值
  - 实现当中需要极其精巧（ carefully engineered ）的数据结构

Number of Nodes	*Iterative Algorithm*				*Lengauer-Tarjan/Cytron et al.*			
	Dominance		Postdominance		Dominance		Postdominance	
	DOM	DF	DOM	DF	DOM	DF	DOM	DF
> 400	3148	1446	2753	1416	7332	2241	6845	1921
201–400	1551	716	1486	674	3315	1043	3108	883
101–200	711	309	600	295	1486	446	1392	388
51–100	289	160	297	151	744	219	700	191
26–50	156	86	165	94	418	119	412	99
<= 25	49	26	52	25	140	32	134	26

Average times by graph size, measured in $\frac{1}{100}$'s of a second

SSA Form 的计算

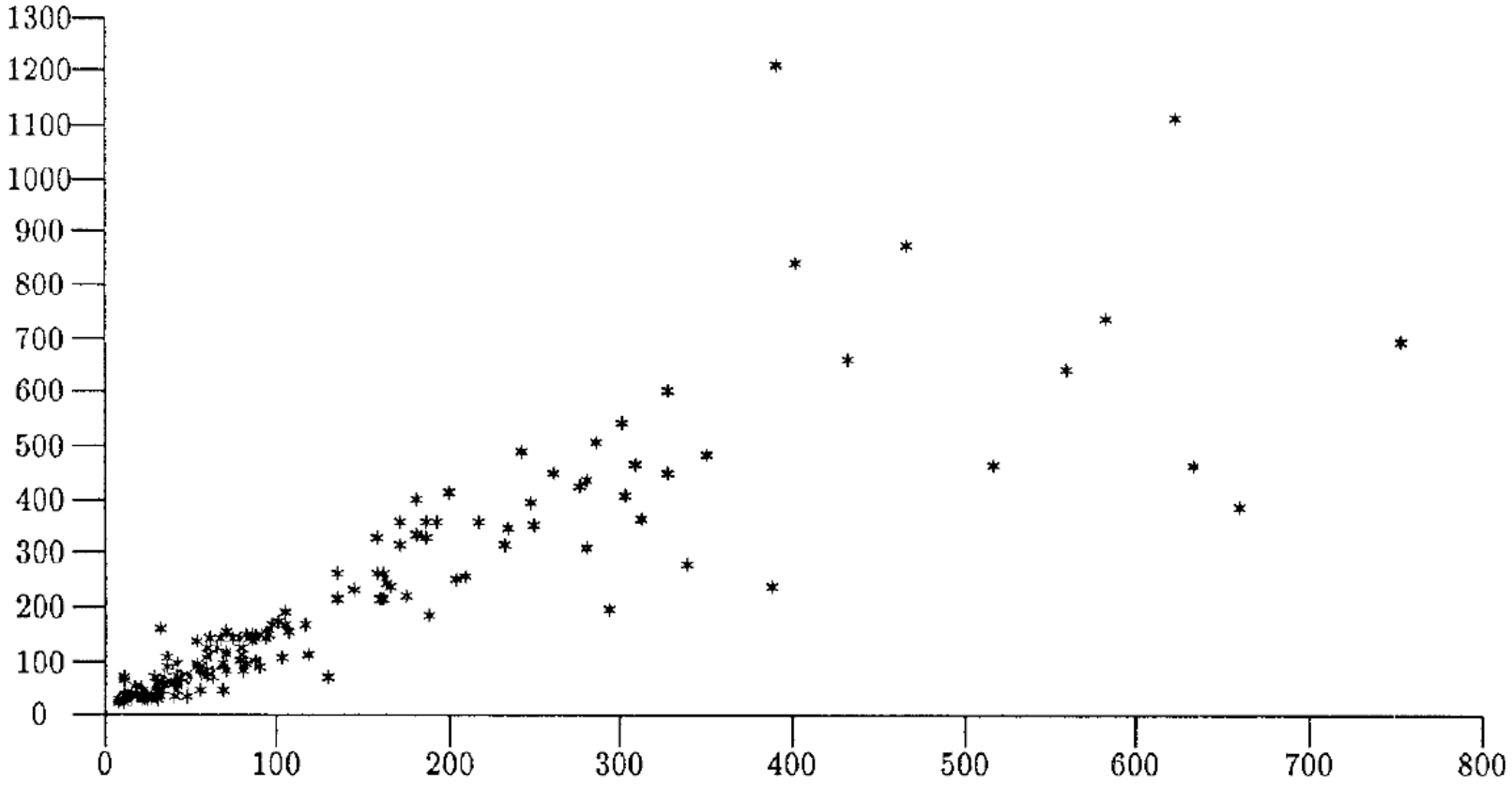


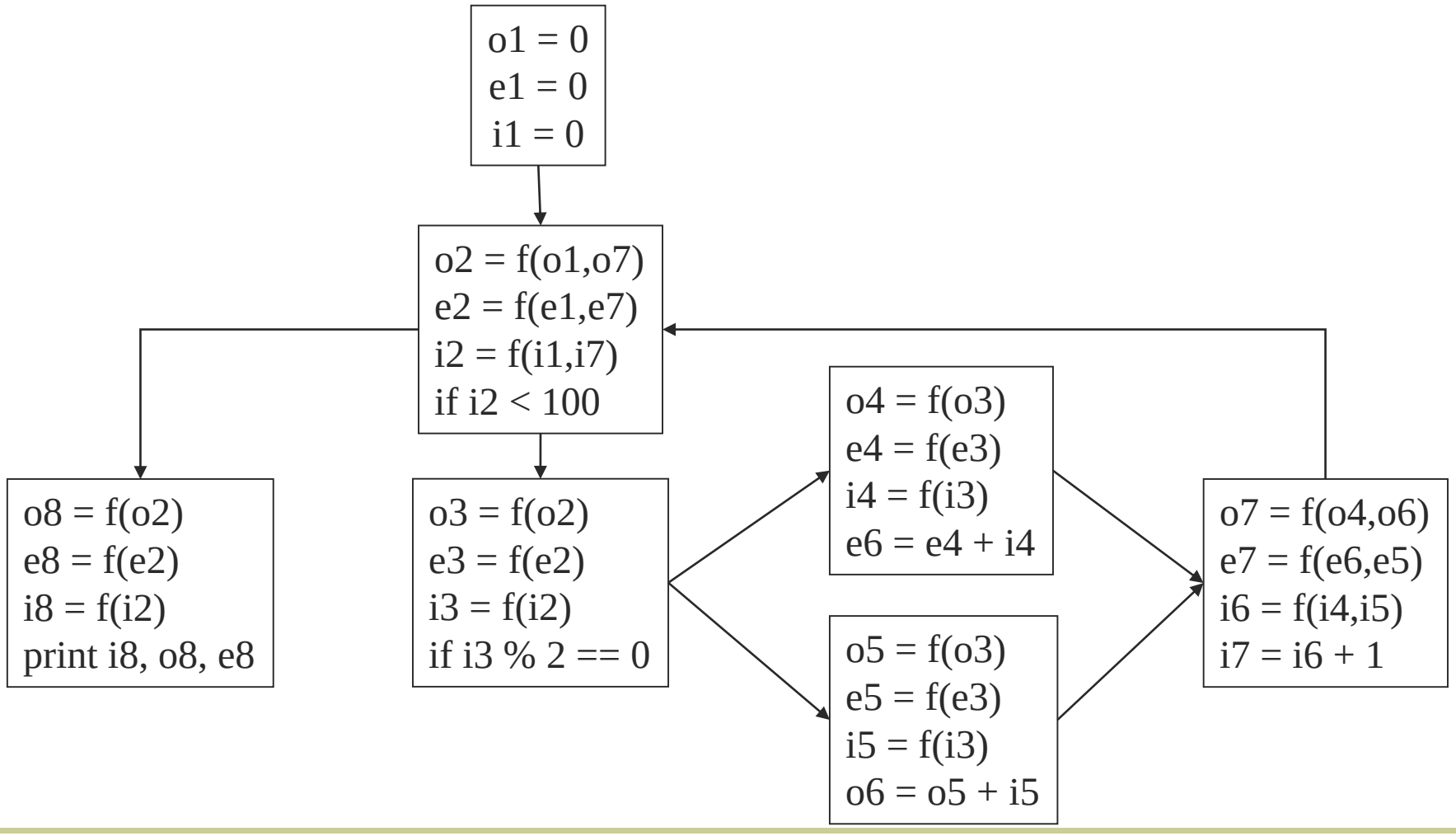
Fig. 21. Number of ϕ-functions versus number of program statements.

SSA 与函数式编程

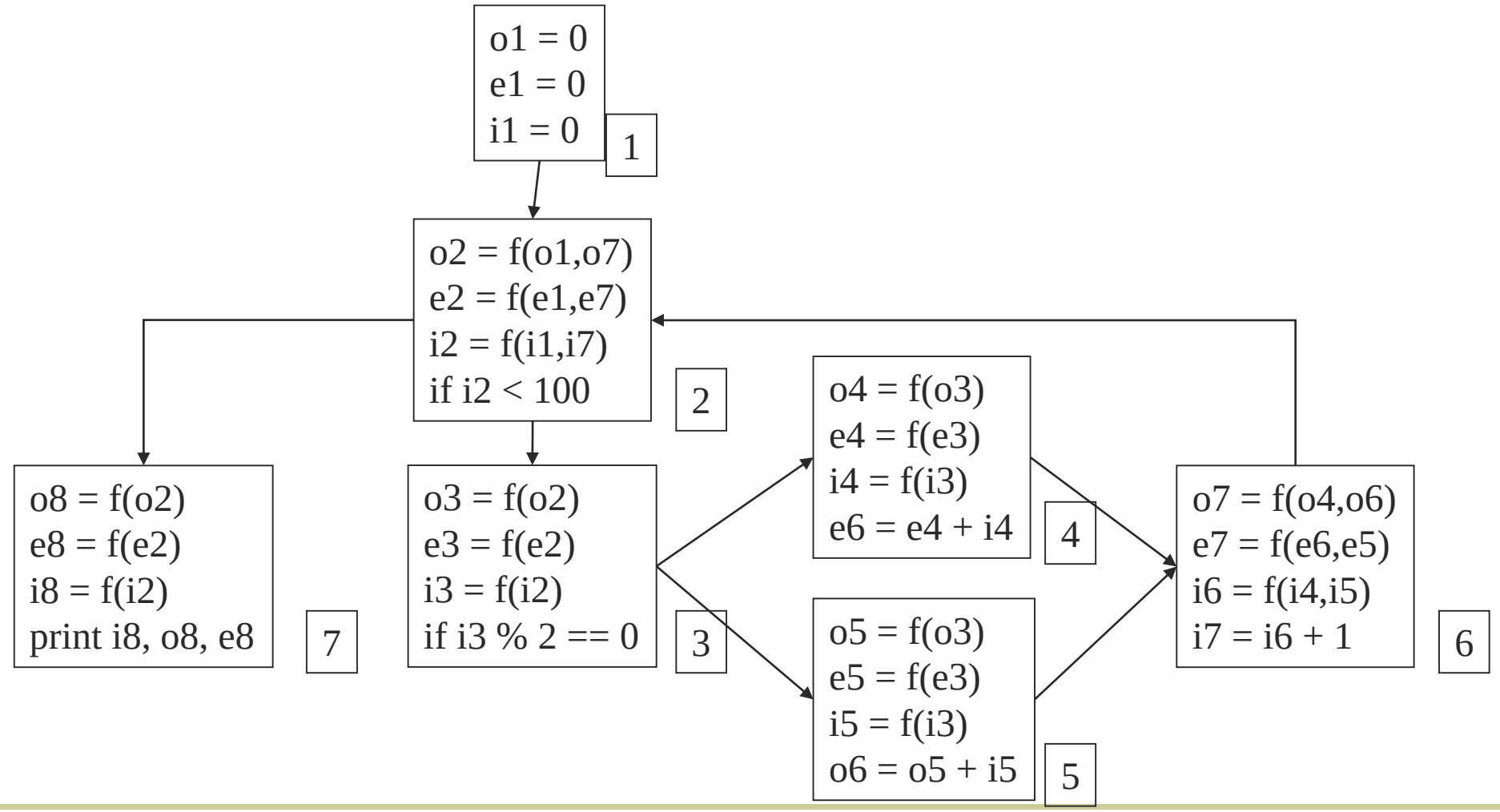
- A. Appel 已经证明了将一般程序转化为SSA，实际上是一个函数式编程的问题

- 我们用最复杂的例子进行演示？

SSA 与函数式编程


o1 = 0
e1 = 0
i1 = 0
o2 = f(o1,o7)
e2 = f(e1,e7)
i2 = f(i1,i7)
if i2 < 100
o8 = f(o2)
e8 = f(e2)
i8 = f(i2)
print i8, o8, e8
o3 = f(o2)
e3 = f(e2)
i3 = f(i2)
if i3 % 2 == 0
o4 = f(o3)
e4 = f(e3)
i4 = f(i3)
e6 = e4 + i4
o5 = f(o3)
e5 = f(e3)
i5 = f(i3)
o6 = o5 + i5
o7 = f(o4,o6)
e7 = f(e6,e5)
i6 = f(i4,i5)
i7 = i6 + 1

SSA 与函数式编程


o1 = 0
e1 = 0
i1 = 0
1
o2 = f(o1,o7)
e2 = f(e1,e7)
i2 = f(i1,i7)
if i2 < 100
2
o8 = f(o2)
e8 = f(e2)
i8 = f(i2)
print i8, o8, e8
7
o3 = f(o2)
e3 = f(e2)
i3 = f(i2)
if i3 % 2 == 0
3
o4 = f(o3)
e4 = f(e3)
i4 = f(i3)
e6 = e4 + i4
4
o5 = f(o3)
e5 = f(e3)
i5 = f(i3)
o6 = o5 + i5
5
o7 = f(o4,o6)
e7 = f(e6,e5)
i6 = f(i4,i5)
i7 = i6 + 1
6

SSA 与函数式编程

```
fun f1() =
    let val o1 = 0 and e1 = 0 and i1 = 0
    in f2(o1, e1, i1)

fun f2(o2, e2, i2) =
    if i2 < 100 then f3(o2, e2, i2)
    else f7(o2, e2, i2)

fun f3(o3, e3, i3) =
    if i3 % 2 == 0 then f4(o3, e3, i3)
    else f5(o3, e3, i3)

fun f7(o8, e8, i8) = print(i8, o8, e8)
```

```
fun f4(o4, e4, i4) =
    let val e6 = e4 + i4
    in  f6(o4, e6, i4)

fun f5(o5, e5, i5) =
    let val o6 = o5 + i5
    in  f6(o6, e5, i5)

fun f6(o7, e7, i6) =
    let val i7 = i6 + 1
    in  f2(o7, e7, i7)
```

还是不具有编程之美？

SSA 与函数式编程

```
let val o1 = 0 and e1 = 0 and i1 = 0
    fun f2(o2, e2, i2) =
        if i2 < 100
        then let fun f6(o4, e4) =
                        let val i3 = i2 + 1
                        in  f2(o4, e4, i3)
             in  if i2 % 2 == 0
                 then let val e3 = e2 + i2
                        in  f6(o2, e3)
                 else let val o3 = o2 + i2
                        in  f6(o3, e2)
        else print(i2, o2, e2)
in  f2(o1, e1, i1)
```

- 请注意，左边依然是我们前面例子中SSA的形式
- 将一般程序转化为SSA的形式的算法，只需要通过函数嵌套来完成，从而避免复杂而冗余的参数传递

SSA的实际使用

- SSA 的应用→ **简化了程序分析及优化过程**
- SSA的特性
  - 每个变量只被定义一次
  - 每次使用均可以索引到其定义
  - 不相关的变量始终有不一样的命名
- 发明于80年代后期，90年代先哲们开展了非常多的研究
- 被嵌入到很多编译器当中，已经逐渐成为了编译优化及程序分析的一种标准方法
  - — ETH Oberon 2
  - — LLVM
  - — GNU GCC 4
  - — IBM Jikes Java VM
  - — Java Hotspot VM

SSA的实际使用

- 发明于80年代后期，90年代先哲们开展了非常多的研究

- 被嵌入到很多编译器当中，已经逐渐成为了编译优化及程序分析的一种标准方法
  - — ETH Oberon 2
  - — LLVM
  - — GNU GCC 4
  - — IBM Jikes Java VM
  - — Java Hotspot VM
  - — Mono
  - — Many more…

SSA的实际使用--代码优化的本质

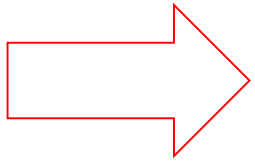
- 性能方面：提升代码执行效率
- 代码规模方面：缩减代码规模，更少的内存占用

- Tradeoffs:
  - 1） 性能 vs. 规模
  - 2） 编译速度 vs. 占用内存

- 并不存在Magic Bullet!

SSA 与常量传播

```
b := 3
c := 1 + b
d := b + c
```

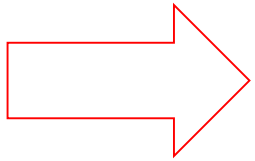


```
b := 3
c := 1 + 3
d := 3 + c
```

- 如果b被赋值了一个常量，那么后续的使用过程中，我们只需要用该常量替换b即可

SSA 与常量传播

```
b := 3
c := 1 + b
d := b + c
```

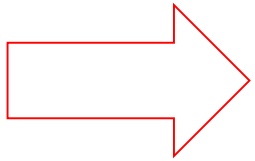


```
b := 3
c := 1 + 3
d := 3 + c
```

- 如果b被赋值了一个常量，那么后续的使用过程中，我们只需要用该常量替换b即可
- 需要较多的分析代价，因为b可以被多次赋值

SSA 与常量传播

```
b1 := 3
c := 1 + b1
d := b1 + c
```

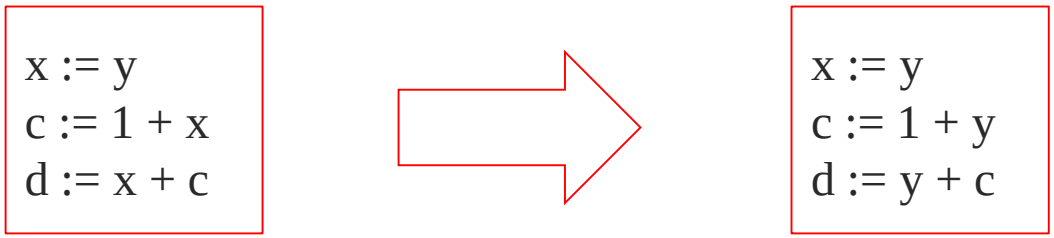


```
b1 := 3
c := 1 + 3
d := 3 + c
```

- 转化为SSA之后，所有的变量只会被赋值一次
- 我们不再需要复杂的分析代价

SSA 与赋值传播

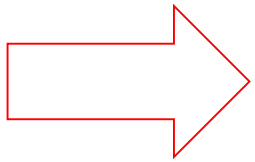
- 对于一个语句 x := y
- 如果 x 与 y 没有被修改，那么可以将后续将 x 的使用y代替



SSA 与赋值传播

- 对于一个语句 x1:= y1
- 可以将后续将 x1 的使用用y1代替

```
x1 := y1
c1 := 1 + x1
d1 := x1 + c1
```



```
x1 := y1
c1 := 1 + y1
d1 := y1 + c1
```

参考文献

- 以下文献具有一定的阅读难度，建议有兴趣的同学阅读

- Cytron, Ron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. "An efficient method of computing static single assignment form." In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Prin ciples of programming languages*, pp. 25-35. 1989
- Lengauer, Thomas, and Robert Endre Tarjan. "A fast algorithm for finding dominators in a flowgraph." ACM Trans actions on Programming Languages and Systems (TOPLAS) 1, no. 1 (1979): 121-141.
- Briggs, Preston, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. "Practical improvements to the construction and destruction of static single assignment form." Software: Practice and Experience 28, no. 8 (1998): 859-881.