

Advanced Compiler

Techniques

<http://lamp.epfl.ch/teaching/advancedCompiler/>

Erik Steinman

LAMP

Introduction

- ◆ What is this course about?
- ◆ How will this be taught?
- ◆ Who is teaching the course?
- ◆ Where to find more information?
- ◆ Why is this course interesting?

Teachers

◆ Lecturer

◆ Erik Stenman

- ◆ have been hacking compilers for money since 1996.
Have been hacking for fun since 1980.
- ◆ Office: INR315, 021-69 37593.

◆ Assistant

◆ Michel Schinz

- ◆ whom you all know and love.
- ◆ Office: INR318, 021-69 34209.

Course Content

- ◆ Optimization Techniques
- ◆ Implementation techniques for high level languages (HLL).

Course Content

- ◆ Optimization Techniques
 - ◆ Theory for analysis and optimization
 - ◆ Optimization algorithms
- ◆ Implementation techniques for high level languages (HLL).
 - ◆ Virtual Machines
 - ◆ Memory Management
 - ◆ High level constructs

Course Goals

- ◆ Give some theoretical framework for compiler optimizations.
- ◆ Give a general orientation on optimization techniques.
- ◆ Give an understanding of how some higher level constructs are implemented.

Non-Goals and Requirements

- ◆ This course will not try to teach you all possible optimizations, or even all common optimizations.
- ◆ We will not talk about parallel machines.
- ◆ You are supposed to be familiar with basic compiler concepts: scanning, parsing, semantic analysis, and simple code generation. (These topics will not be touched.)
- ◆ You are supposed to be used to programming in Java.

Course Structure

- ◆ The course will be made up of lectures, articles, two projects, and an oral exam.
- ◆ The lectures will be given with slides like this one, and the slides will be available on the web:
<http://lamp.epfl.ch/teaching/advancedCompiler/>
- ◆ I will try to have the slides on the web at least a day before the lecture.

Preliminary Schedule

1. Introduction
2. Control-Flow Analysis & Foundations of Data-Flow Analysis
3. Reaching Definitions, Available Expressions, and Liveness Analyses. Introduction to Abstract Interpretation.
4. Static Single Assignment Form, SSA-based Dead Code Elimination & Sparse Conditional Constant Propagation
5. [cont] Static Single Assignment Form, SSA-based Dead Code Elimination & Sparse Conditional Constant Propagation
6. Partial Redundancy Elimination & Lazy Code Motion
7. Loop Optimizations
8. Global Register Allocation
9. Code Scheduling
10. Implementation of higher order functions, processes, and objects
11. Automatic Memory Management
12. Virtual Machines, Interpretation Techniques, and Just-In-Time Compilers
13. Presentations

The Projects

- ◆ There will be two projects in the course and you may work in groups of two persons.
- ◆ Project 1: A simple register allocator.
 - ◆ The main goal of the first project is to get familiar with the compiler framework that we will use for the second project.
 - ◆ The task is to implement a Sethi-Ullman tree-based register allocator for a given compiler.
- ◆ Project 2: Optimizations.
 - ◆ The goal of the second project is to get a concrete understanding of different optimization techniques.
 - ◆ The task will be to implement different optimizations in the given compiler in order to achieve a given speedup on a set of benchmarks.

Literature

Expect to read a lot for this class,
especially in order to complete the
projects.

- ◆ Course Book:
 - ◆ Andrew W. Appel,
Modern compiler implementation in Java (second edition).
Cambridge University Press, 2002, ISBN 052182060X.
- ◆ Alternative:
 - ◆ Keith Cooper and Linda Torczon,
Engineering a Compiler, Morgan Kaufmann, October 2003.
- ◆ Reference:
 - ◆ Steven Muchnick,
Advanced Compiler Design and Implementation,
Morgan Kaufmann, August 1997.
- ◆ Additional articles that will be handed out.

The Exam

- ◆ There will be an oral exam **during the last week** of the course.
- ◆ The exam will concentrate on the understanding of the concepts taught in the course, and not on details of specific algorithms.

Note:

The examination form of the course has changed from “Branche à examen (oral) avec contrôle continu” to “**Contrôle continu**”

Feedback

- ◆ This is the first time this course is given so the format and the content is not set in stone.
- ◆ At the end of next week there will be an evaluation of the course so far, and you will have chance to influence the rest of the course to a great extent.

Why is this course interesting?

- ◆ Optimization is challenging – you can not write an optimal compiler: there is always room for improvements.
- ◆ The course will give you many techniques and tools that you can use in other areas.
- ◆ You will gain a better understanding of how a compiler works and what to expect of the code generated by compilers.
- ◆ It is fun!

Foundations of Dataflow Analysis

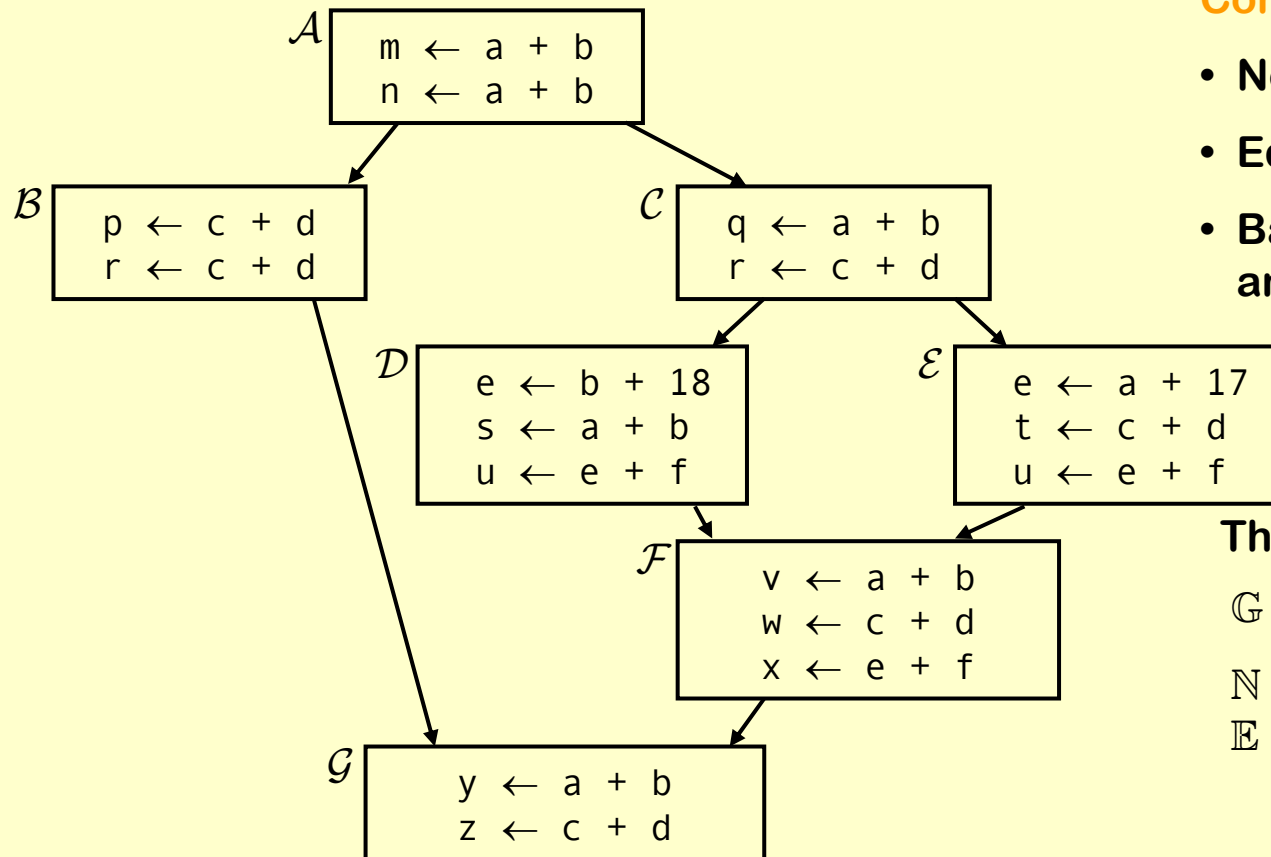
This lecture is primarily based on Konstantinos Sagonas set of slides
(Advanced Compiler Techniques, (2AD518)
at Uppsala University, January-February 2004).
Used with kind permission.

Terminology: Program Representation

Control Flow Graph (CFG):

- ◆ Nodes N – statements of program
- ◆ Edges E – flow of control
 - ◆ $pred(n)$ = set of all immediate predecessors of n
 - ◆ $succ(n)$ = set of all immediate successors of n
- ◆ Start node n_0
- ◆ Set of final nodes N_{final}

Terminology: Control-Flow Graph



Control-flow graph (CFG)

- Nodes for basic blocks
- Edges for branches
- Basis for much of program analysis & transformation

This CFG,

$$G = (N, E)$$

$$N = \{A, B, C, D, E, F, G\}$$

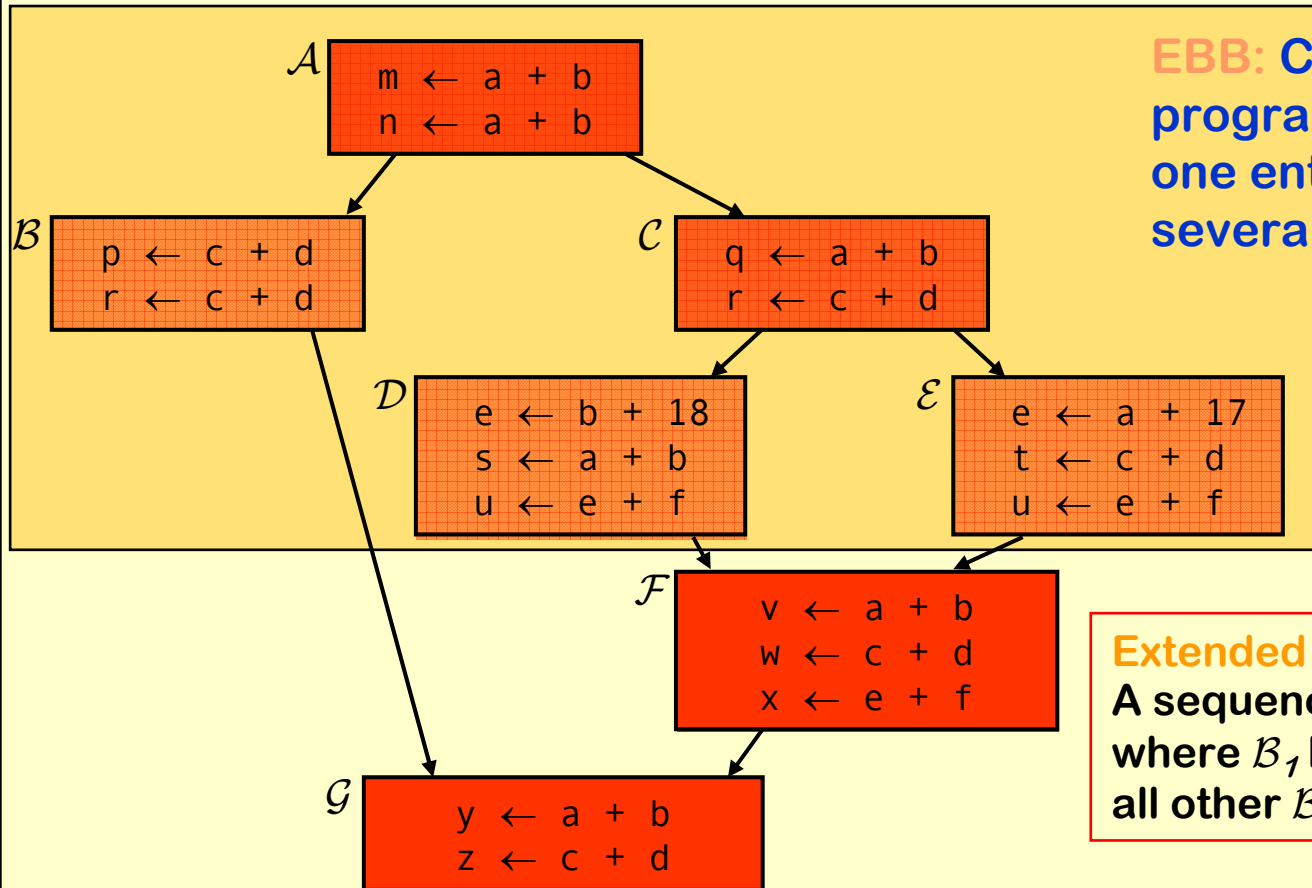
$$E = \{(A, B), (A, C), (B, G), (C, D), (C, E), (D, F), (E, F), (F, G)\}$$

$$|N| = 7$$

$$|E| = 8$$

Terminology:

Extended Basic Block



EBB: Conceptually it is a program sequence with only one entry point but possibly several exit points.

An EBB contains 1 or more paths. This EBB $(\{A, B, C, D, E\})$ contains the paths $\{A, B\}$, $\{A, C, D\}$, $\{A, C, E\}$

Extended Basic Block (EBB):

A sequence of basic blocks B_1, B_2, \dots, B_n where B_1 has more than 1 predecessor, all other B_i have a unique predecessor.

Path:

A sequence of basic blocks B_1, B_2, \dots, B_n where B_i is the predecessor of B_{i+1} .

Terminology: Program Points

- ◆ One program point before each node.
- ◆ One program point after each node.
- ◆ *Join point* – Program point with multiple predecessors.
- ◆ *Split point* – Program point with multiple successors.

Dataflow Analysis

Compile-Time Reasoning About

- ◆ Run-Time Values of Variables or Expressions at different program points:
 - ◆ Which assignment statements produced the value of the variables at this point?
 - ◆ Which variables contain values that are no longer used after this program point?
 - ◆ What is the range of possible values of a variable at this program point?

Dataflow Analysis

◆ Assumptions:

- ◆ We have a syntactically and semantically correct program (as far as compile time analysis can determine this).
- ◆ We have the “whole” program, or a clearly defined subset of the program which will only interact with the rest of the program through a predefined interface.

(That is, no *self* modifying code, and if the interface is a function then the parameters can take any value of the given type.)

Dataflow Analysis: Basic Idea

- ◆ Information about a program represented using values from an algebraic structure called *lattice*. (We will call this set of values \mathbb{P} .)
- ◆ Analysis produces a lattice value for each program point.
- ◆ Two flavors of analyses:
 - ◆ *Forward dataflow analyses.*
 - ◆ *Backward dataflow analyses.*

Forward Dataflow Analysis

- ◆ Analysis propagates values forward through control flow graph with flow of control
 - ◆ Each node has a transfer function f
 - ◆ Input – value at program point before node.
 - ◆ Output – new value at program point after node.
 - ◆ Values flow from program points after predecessor nodes to program points before successor nodes.
 - ◆ At join points, values are combined using a merge function.
- ◆ Canonical Example: **Reaching Definitions.**

Backward Dataflow Analysis

- ◆ Analysis propagates values backward through control flow graph against flow of control:
 - ◆ Each node has a transfer function f
 - ◆ Input – value at program point after node.
 - ◆ Output – new value at program point before node.
 - ◆ Values flow from program points before successor nodes to program points after predecessor nodes.
 - ◆ At split points, values are combined using a merge function.
- ◆ Canonical Example: **Live Variables**.

Partial Orders

◆ Set \mathbb{P}

◆ Partial order \leq such that $\forall x, y, z \in \mathbb{P}$

i. $x \leq x$ (reflexive)

ii. $x \leq y$ and $y \leq x \Rightarrow x = y$ (antisymmetric)

iii. $x \leq y$ and $y \leq z \Rightarrow x \leq z$ (transitive)

Upper Bounds

- ◆ If $S \subseteq \mathbb{P}$ then
 - ◆ $x \in \mathbb{P}$ is an *upper bound* of S if
$$\forall y \in S, y \leq x$$
 - ◆ $x \in \mathbb{P}$ is the *least upper bound* (lub) of S if
 - ◆ x is an upper bound of S , and
 - ◆ $x \leq y$ for all upper bounds y of S
 - ◆ \vee - *join*, least upper bound, supremum (sup)
 - ◆ $\bigvee S$ is the least upper bound of S
 - ◆ $x \vee y$ is the least upper bound of $\{x, y\}$

Lower Bounds

- ◆ If $S \subseteq \mathbb{P}$ then
 - ◆ $x \in \mathbb{P}$ is a *lower bound* of S if $\forall y \in S, x \leq y$
 - ◆ $x \in \mathbb{P}$ is the *greatest lower bound (glb)* of S if
 - ◆ x is a lower bound of S , and
 - ◆ $y \leq x$ for all lower bounds y of S
 - ◆ \wedge - *meet*, greatest lower bound, infimum (inf)
 - ◆ $\wedge S$ is the greatest lower bound of S
 - ◆ $x \wedge y$ is the greatest lower bound of $\{x, y\}$

Coverings

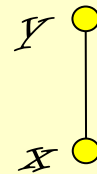
- ◆ Notation: $x < y$ if $x \leq y$ and $x \neq y$
- ◆ x is covered by y (y covers x) if
 - ◆ $x < y$, and
 - ◆ $x \leq z < y \Rightarrow x = z$
- ◆ Conceptually, y covers x if there are no elements between x and y

Dataflow Analysis: Basic Idea

- ◆ Information about a program represented using values from an algebraic structure called *lattice*. (We will call this set of values \mathbb{P} .)
- ◆ Analysis produces a lattice value for each program point.
- ◆ Two flavors of analyses:
 - ◆ *Forward dataflow analyses.*
 - ◆ *Backward dataflow analyses.*

Hasse Diagram

- ◆ We can visualize a partial order with a Hasse Diagram.
- ◆ For each element x we draw a circle: ●
- ◆ If y covers x
 - ◆ Line from y to x
 - ◆ y above x in diagram

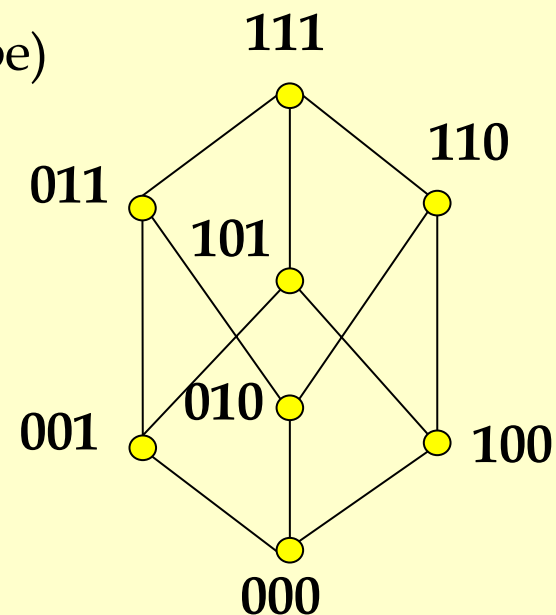


Hasse Diagram: Example

$$\mathbb{P} = \{000, 001, 010, 011, 100, 101, 110, 111\}$$

$$x \leq y \text{ if } (x \text{ bitwise_and } y) = x$$

(standard boolean lattice, also called hypercube)



Lattices

- ◆ If $x \wedge y$ and $x \vee y$ exist for all $x, y \in \mathbb{P}$, then \mathbb{P} is a *lattice*.
- ◆ If $\bigwedge S$ and $\bigvee S$ exist for all $S \subseteq \mathbb{P}$, then \mathbb{P} is a *complete lattice*.
- ◆ Theorem: **All finite lattices are complete.**
- ◆ Example of a lattice that is not complete
 - ◆ Integers \mathbb{Z}
 - ◆ For any $x, y \in \mathbb{Z}$, $x \vee y = \max(x, y)$, $x \wedge y = \min(x, y)$
 - ◆ But $\bigvee \mathbb{Z}$ and $\bigwedge \mathbb{Z}$ do not exist
 - ◆ $\mathbb{Z} \cup \{+\infty, -\infty\}$ is a complete lattice

Top and Bottom

- ◆ Greatest element of \mathbb{P} (if it exists) is *top* (\top).
- ◆ Least element of \mathbb{P} (if it exists) is *bottom* (\perp).

Connection between \leq , \wedge , and \vee

The following 3 properties are equivalent:

- ◆ $x \leq y$

- ◆ $x \vee y = y$

- ◆ $x \wedge y = x$

- ◆ Will prove:

- ◆ $x \leq y \Rightarrow x \vee y = y$ and $x \wedge y = x$

- ◆ $x \vee y = y \Rightarrow x \leq y$

- ◆ $x \wedge y = x \Rightarrow x \leq y$

- ◆ By Transitivity,

- ◆ $x \vee y = y \Rightarrow x \wedge y = x$

- ◆ $x \wedge y = x \Rightarrow x \vee y = y$

Connecting Lemma Proofs (1)

- ◆ Proof of $x \leq y \Rightarrow x \vee y = y$
 - ◆ $x \leq y \Rightarrow y$ is an upper bound of $\{x, y\}$.
 - ◆ Any upper bound z of $\{x, y\}$ must satisfy $y \leq z$.
 - ◆ So y is least upper bound of $\{x, y\}$ and $x \vee y = y$
- ◆ Proof of $x \leq y \Rightarrow x \wedge y = x$
 - ◆ $x \leq y \Rightarrow x$ is a lower bound of $\{x, y\}$.
 - ◆ Any lower bound z of $\{x, y\}$ must satisfy $z \leq x$.
 - ◆ So x is the greatest lower bound of $\{x, y\}$,
that is $x \wedge y = x$

Connecting Lemma Proofs (2)

- ◆ Proof of $x \vee y = y \Rightarrow x \leq y$
 - ◆ y is an upper bound of $\{x, y\} \Rightarrow x \leq y$
- ◆ Proof of $x \wedge y = x \Rightarrow x \leq y$
 - ◆ x is a lower bound of $\{x, y\} \Rightarrow x \leq y$

Lattices as Algebraic Structures

- ◆ Have defined \vee and \wedge in terms of \leq .
- ◆ Now define \leq in terms of \vee and \wedge :
 - ◆ Start with \vee and \wedge as arbitrary algebraic operations that satisfy associative, commutative, idempotence, and absorption laws.
 - ◆ Will define \leq using \vee and \wedge .
 - ◆ Will show that \leq is a partial order.

Algebraic Properties of Lattices

Assume arbitrary operations \vee and \wedge such that

- ◆ $(x \vee y) \vee z = x \vee (y \vee z)$ (associativity of \vee)
- ◆ $(x \wedge y) \wedge z = x \wedge (y \wedge z)$ (associativity of \wedge)
- ◆ $x \vee y = y \vee x$ (commutativity of \vee)
- ◆ $x \wedge y = y \wedge x$ (commutativity of \wedge)
- ◆ $x \vee x = x$ (idempotence of \vee)
- ◆ $x \wedge x = x$ (idempotence of \wedge)
- ◆ $x \vee (x \wedge y) = x$ (absorption of \vee over \wedge)
- ◆ $x \wedge (x \vee y) = x$ (absorption of \wedge over \vee)

Connection Between \wedge and \vee

Theorem: $x \vee y = y$ if and only if $x \wedge y = x$

◆ Proof of $x \vee y = y \Rightarrow x = x \wedge y$

$$\begin{aligned} x &= x \wedge (x \vee y) && \text{(by absorption)} \\ &= x \wedge y && \text{(by assumption)} \end{aligned}$$

◆ Proof of $x \wedge y = x \Rightarrow y = x \vee y$

$$\begin{aligned} y &= y \vee (y \wedge x) && \text{(by absorption)} \\ &= y \vee (x \wedge y) && \text{(by commutativity)} \\ &= y \vee x && \text{(by assumption)} \\ &= x \vee y && \text{(by commutativity)} \end{aligned}$$

Properties of \leq

- ◆ Define $x \leq y$ if $x \vee y = y$
- ◆ Proof of transitive property. Show that

$$x \vee y = y \text{ and } y \vee z = z \Rightarrow x \vee z = z$$

$$\begin{aligned} x \vee z &= x \vee (y \vee z) && \text{(by assumption)} \\ &= (x \vee y) \vee z && \text{(by associativity)} \\ &= y \vee z && \text{(by assumption)} \\ &= z && \text{(by assumption)} \end{aligned}$$

Properties of \leq

- ◆ Proof of asymmetry property. Show that

$$x \vee y = y \text{ and } y \vee x = x \Rightarrow x = y$$

$$x = y \vee x \quad (\text{by assumption})$$

$$= x \vee y \quad (\text{by commutativity})$$

$$= y \quad (\text{by assumption})$$

- ◆ Proof of reflexivity property. Show that

$$x \vee x = x$$

$$x \vee x = x \quad (\text{by idempotence})$$

Properties of \leq

- ◆ Induced operation \leq agrees with original definitions of \vee and \wedge , i.e.,
 - ◆ $x \vee y = \sup \{x, y\}$
 - ◆ $x \wedge y = \inf \{x, y\}$

Proof of $x \vee y = \sup \{x, y\}$

- ◆ Consider any upper bound u for x and y .
- ◆ Given $x \vee u = u$ and $y \vee u = u$,

show $x \vee y \leq u$,

i.e., $(x \vee y) \vee u = u$

$$u = x \vee u \quad (\text{by assumption})$$

$$= x \vee (y \vee u) \quad (\text{by assumption})$$

$$= (x \vee y) \vee u \quad (\text{by associativity})$$

Proof of $x \wedge y = \inf \{x, y\}$

- Consider any lower bound l for x and y .
- Given $x \wedge l = l$ and $y \wedge l = l$,

show $l \leq x \wedge y$,

i.e., $(x \wedge y) \wedge l = l$

$$l = x \wedge l$$

(by assumption)

$$= x \wedge (y \wedge l)$$

(by assumption)

$$= (x \wedge y) \wedge l$$

(by associativity)

Chains

- ◆ A set S is a *chain* if $\forall x, y \in S. y \leq x$ or $x \leq y$
- ◆ \mathbb{P} has no infinite chains if every chain in \mathbb{P} is finite
- ◆ \mathbb{P} satisfies the *ascending chain condition* if for all sequences $x_1 \leq x_2 \leq \dots$ there exists n such that $x_n = x_{n+1} = \dots$
That is, all increasing sequences in \mathbb{P} eventually becomes constant.

Dataflow Analysis (repetition)

- ◆ Information about a program represented using values from a *lattice* (\mathbb{P}). Analysis propagates values through control flow graph, either forwards or backwards.
- ◆ For forward analysis:
 - ◆ Each node has a transfer function f ,
 - ◆ Input – value at program point before node.
 - ◆ Output – new value at program point after node.
 - ◆ Values flow from program points after predecessor nodes to program points before successor nodes.
 - ◆ At join points, values are combined using a merge function.

Transfer Functions

- ◆ Assume a lattice \mathbb{P} of abstract values.
- ◆ Transfer function $f: \mathbb{P} \rightarrow \mathbb{P}$ for each node in control flow graph.
- ◆ f models the effect of the node on the program information.

Properties of Transfer Functions

Each dataflow analysis problem has a set \mathbb{F} of transfer functions $f:\mathbb{P}\rightarrow\mathbb{P}$

- ◆ Identity function $i\in\mathbb{F}$
- ◆ \mathbb{F} must be closed under composition:
 $\forall f,g\in\mathbb{F}$, the function $h = \lambda x.f(g(x))\in\mathbb{F}$
- ◆ Each $f\in\mathbb{F}$ must be monotone: $x \leq y \Rightarrow f(x) \leq f(y)$
- ◆ Sometimes all $f\in\mathbb{F}$ are distributive:
 $f(x \vee y) = f(x) \vee f(y)$
- ◆ **Distributivity \Rightarrow monotonicity**

Distributivity Implies Monotonicity

Proof:

- ◆ Assume $f(x \vee y) = f(x) \vee f(y)$
- ◆ Show: $x \vee y = y \Rightarrow f(x) \vee f(y) = f(y)$
 $f(y) = f(x \vee y)$ (by assumption)
 $= f(x) \vee f(y)$ (by distributivity)

Forward Dataflow Analysis

- ◆ Simulates forward execution of a program
- ◆ For each node n , we have
 - in_n – value at program point before n
 - out_n – value at program point after n
 - f_n – transfer function for n (given in_n , computes out_n)
- ◆ Require that solutions satisfy
 - i. $\forall n, out_n = f_n(in_n)$
 - ii. $\forall n \neq n_0, in_n = \vee \{ out_m \mid m \in pred(n) \}$
 - iii. $in_{n_0} = \perp$

Dataflow Equations

- ◆ Result is a set of dataflow equations

$$\text{out}_n := f_n(\text{in}_n)$$

$$\text{in}_n := \vee \{ \text{out}_m \mid m \in \text{pred}(n) \}$$

- ◆ Conceptually separates analysis problem from program.

Worklist Algorithm for Solving Forward Dataflow Equations

for each $n \in \mathbb{N}$ do $out_n := f_n(\perp)$

$worklist := \mathbb{N}$

while $worklist \neq \emptyset$ do:

 remove a node n from $worklist$

$in_n := \vee \{ out_m \mid m \in pred(n) \}$

$out_n := f_n(in_n)$

 if out_n changed then

$worklist := worklist \cup succ(n)$

Correctness Argument

Why result satisfies dataflow equations?

- ◆ Whenever we process a node n ,
set $out_n := f_n(in_n)$
Algorithm ensures that $out_n = f_n(in_n)$
- ◆ Whenever out_m changes, put $succ(m)$ on **worklist**.
Consider any node $n \in succ(m)$.
It will eventually come off the **worklist** and the
algorithm will set

$$in_n := \vee \{ out_m \mid m \in pred(n) \}$$
 to ensure that $in_n = \vee \{ out_m \mid m \in pred(n) \}$

Termination Argument

Why does the algorithm terminate?

- ◆ Sequence of values taken on by in_n or out_n is a chain. If values stop increasing, the worklist empties and the algorithm terminates.
- ◆ If the lattice has the ascending chain property, the algorithm terminates
 - ◆ Algorithm terminates for finite lattices.
 - ◆ For lattices without the ascending chain property, we must use a *widening* operator.

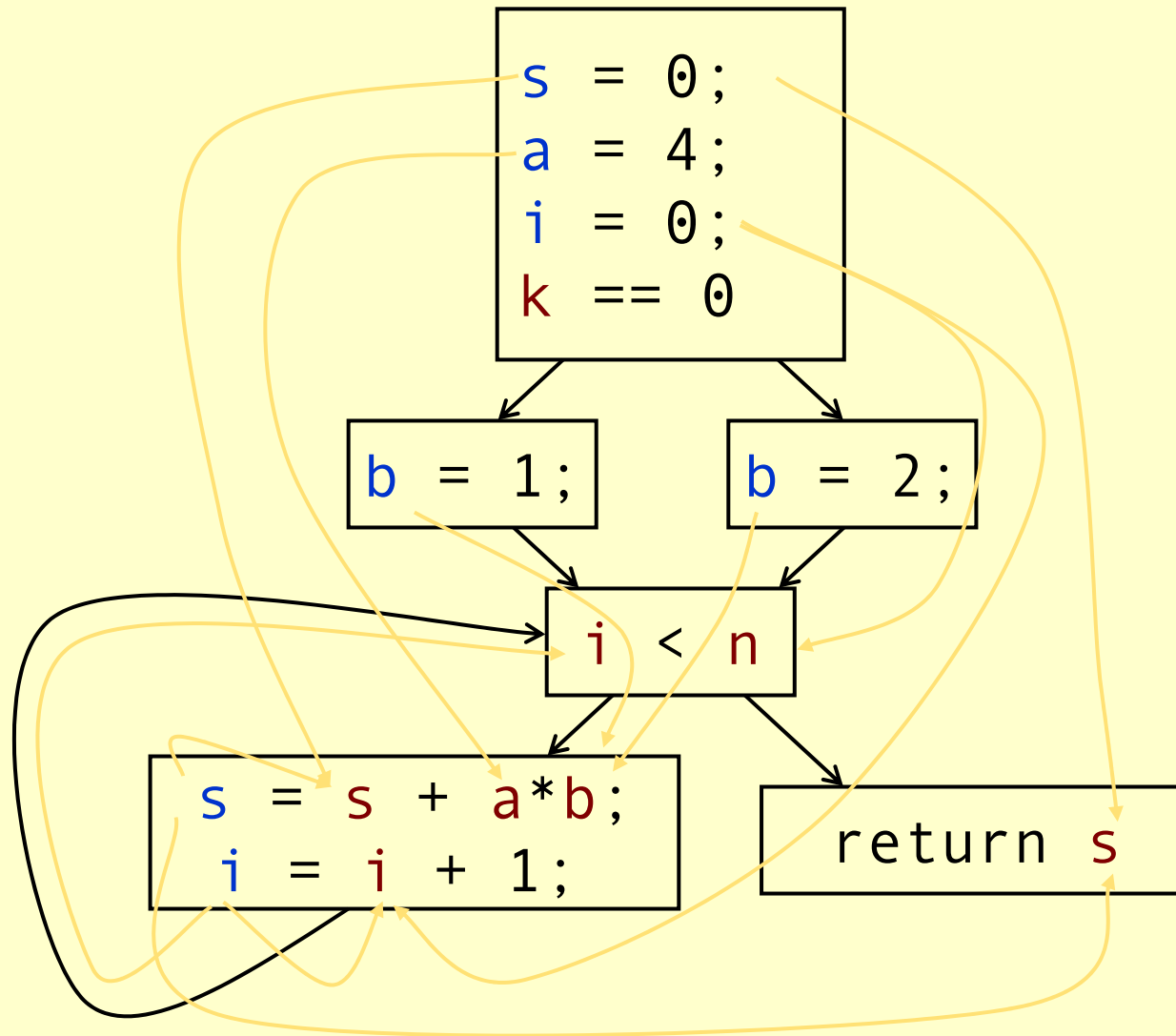
Widening Operators

- ◆ Detect lattice values that may be part of an infinitely ascending chain.
- ◆ Artificially raise value to least upper bound of the chain.
- ◆ Example:
 - ◆ Lattice is set of all subsets of integers.
 - ◆ Widening operator might raise all sets of size n or greater to **TOP** (the set of all integers).
 - ◆ Could be used to collect possible values taken on by a variable during execution of the program.

Reaching Definitions

- ◆ Concept of *definition* and *use*
 - ◆ $z = x + y$
 - ◆ is a **definition** of z
 - ◆ is a **use** of x and y
- ◆ A definition (**d**) reaches a use (**u**) if the value written by **d** may be read by **u**.

Reaching Definitions



Reaching Definitions Framework

- ◆ $\mathbb{P} = \wp$ (the powerset) of the set of definitions in the program (all subsets of the set of definitions).
- ◆ $\vee = \cup$ (order is \subseteq)
- ◆ $\perp = \emptyset$
- ◆ $\mathbb{F} =$ all functions f of the form $f(\mathbf{x}) = \mathbf{a} \cup (\mathbf{x} - \mathbf{b})$
 - ◆ \mathbf{b} is the set of definitions that the node kills.
 - ◆ \mathbf{a} is the set of definitions that the node generates.

General pattern for many transfer functions

- ◆ $f(\mathbf{x}) = \text{GEN} \cup (\mathbf{x} - \text{KILL})$

Does Reaching Definitions Framework Satisfy Properties?

- ◆ \subseteq satisfies conditions for \leq

$$x \subseteq y \text{ and } y \subseteq z \Rightarrow x \subseteq z \quad (\text{transitivity})$$

$$x \subseteq y \text{ and } y \subseteq x \Rightarrow y = x \quad (\text{asymmetry})$$

$$x \subseteq x \quad (\text{reflexivity})$$

- ◆ \mathbb{F} satisfies transfer function conditions

$$\lambda x. \emptyset \cup (x - \emptyset) = \lambda x. x \in \mathbb{F} \quad (\text{identity})$$

$$\text{Will show } f(x \cup y) = f(x) \cup f(y) \quad (\text{distributivity})$$

$$\begin{aligned} f(x) \cup f(y) &= (a \cup (x - b)) \cup (a \cup (y - b)) \\ &= a \cup (x - b) \cup (y - b) \\ &= a \cup ((x \cup y) - b) \\ &= f(x \cup y) \end{aligned}$$

Does Reaching Definitions Framework Satisfy Properties?

What about **composition**?

- ◆ Given $f_1(x) = a_1 \cup (x - b_1)$ and $f_2(x) = a_2 \cup (x - b_2)$
- ◆ Show $f_1(f_2(x))$ can be expressed as $a \cup (x - b)$

$$\begin{aligned}
 f_1(f_2(x)) &= a_1 \cup ((a_2 \cup (x - b_2)) - b_1) \\
 &= a_1 \cup ((a_2 - b_1) \cup ((x - b_2) - b_1)) \\
 &= (a_1 \cup (a_2 - b_1)) \cup ((x - b_2) - b_1) \\
 &= (a_1 \cup (a_2 - b_1)) \cup (x - (b_2 \cup b_1))
 \end{aligned}$$

Let $a = (a_1 \cup (a_2 - b_1))$ and $b = b_2 \cup b_1$

Then $f_1(f_2(x)) = a \cup (x - b)$

General Result

All GEN/KILL transfer function frameworks satisfy the properties:

- ◆ Identity
- ◆ Distributivity
- ◆ Compositionality

Available Expressions Framework

- ◆ $\mathbb{P} = \wp$ (the powerset) of the set of all expressions in the program (all subsets of set of expressions).
- ◆ $\vee = \cap$ (order is \supseteq)
- ◆ $\perp = \wp$ (but $\text{in}_{n0} = \emptyset$)
- ◆ \mathbb{F} = all functions f of the form

$$f(x) = a \cup (x-b).$$
 - ◆ b is set of expressions that node kills.
 - ◆ a is set of expressions that node generates.
- ◆ Another GEN/KILL analysis

Concept of Conservatism

- ◆ Reaching definitions use \cup as join
 - ◆ Optimizations must take into account all definitions that reach along ANY path
- ◆ Available expressions use \cap as join
 - ◆ Optimization requires expression to reach along ALL paths
- ◆ Optimizations must conservatively take all possible executions into account.
- ◆ Structure of analysis varies according to the way the results of the analysis are to be used.

Backward Dataflow Analysis

- Simulates execution of program backward against the flow of control.
- For each node n , we have
 - in_n – value at program point before n .
 - out_n – value at program point after n .
 - f_n – transfer function for n (given out_n , computes in_n).
- Require that solutions satisfy:
 - $\forall n. in_n = f_n(out_n)$
 - $\forall n \notin N_{final}. out_n = \vee \{ in_m \mid m \in succ(n) \}$
 - $\forall n \in N_{final}. out_n = \perp$

Worklist Algorithm for Solving Backward Dataflow Equations

for each $n \in N$ do $in_n := f_n(\perp)$

$worklist := N$

while $worklist \neq \emptyset$ do

remove a node n from $worklist$

$out_n := \vee \{ in_m \mid m \in succ(n) \}$

$in_n := f_n(out_n)$

if in_n changed then

$worklist := worklist \cup pred(n)$

Live Variables Analysis Framework

- ◆ \mathbb{P} = powerset of the set of all variables in the program (all subsets of the set of variables).
- ◆ $\vee = \cup$ (order is \subseteq)
- ◆ $\perp = \emptyset$
- ◆ \mathbb{F} = all functions f of the form $f(x) = a \cup (x-b)$
 - ◆ b is set of variables that the node kills.
 - ◆ a is set of variables that the node reads.

Meaning of Dataflow Results

- ◆ Connection between executions of program and dataflow analysis results.
- ◆ Each execution generates a trajectory of states:
 - ◆ $s_0; s_1; \dots; s_k$, where each $s_i \in \mathcal{S}$
- ◆ Map current state s_k to
 - ◆ Program point n where execution located.
 - ◆ Value x in dataflow lattice.
- ◆ Require $x \leq \text{in}_n$

Abstraction Function for Forward Dataflow Analysis

- ◆ Meaning of analysis results is given by an abstraction function $AF: S \rightarrow \mathbb{P}$
- ◆ Require that for all states s

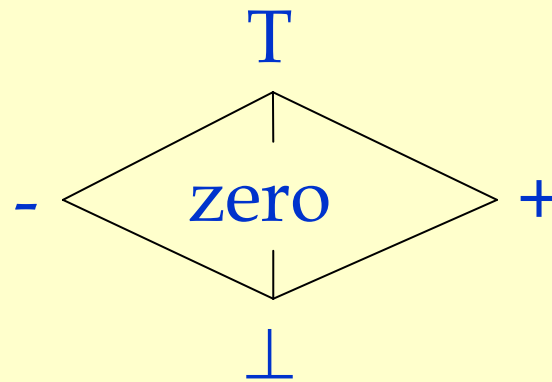
$$AF(s) \leq in_n$$

where n is the program point where the execution is located at in state s , and in_n is the abstract value before that point.

Sign Analysis Example

Sign analysis - compute sign of each variable v

- ◆ Base Lattice: flat lattice on $\{-, \text{zero}, +\}$



- ◆ Actual lattice records a value for each variable
 - ◆ Example element: $[a \rightarrow +, b \rightarrow \text{zero}, c \rightarrow -]$

Interpretation of Lattice Values

If value of v in lattice is:

- ◆ \perp : no information about the sign of v .
- ◆ $-$: variable v is negative.
- ◆ zero : variable v is 0 .
- ◆ $+$: variable v is positive.
- ◆ T : v may be positive or negative or 0.

Operation \otimes on Lattice

\otimes	\perp	-	zero	+	T
\perp	\perp	-	zero	+	T
-	-	+	zero	-	T
zero	zero	zero	zero	zero	zero
+	+	-	zero	+	T
T	T	T	zero	T	T

Transfer Functions

Defined by structural induction on the shape of nodes:

- ◆ If n of the form $v = c$
 - ◆ $f_n(x) = x[v \rightarrow +]$ if c is positive
 - ◆ $f_n(x) = x[v \rightarrow \text{zero}]$ if c is 0
 - ◆ $f_n(x) = x[v \rightarrow -]$ if c is negative
- ◆ If n of the form $v_1 = v_2 * v_3$
 - ◆ $f_n(x) = x[v_1 \rightarrow x[v_2] \otimes x[v_3]]$

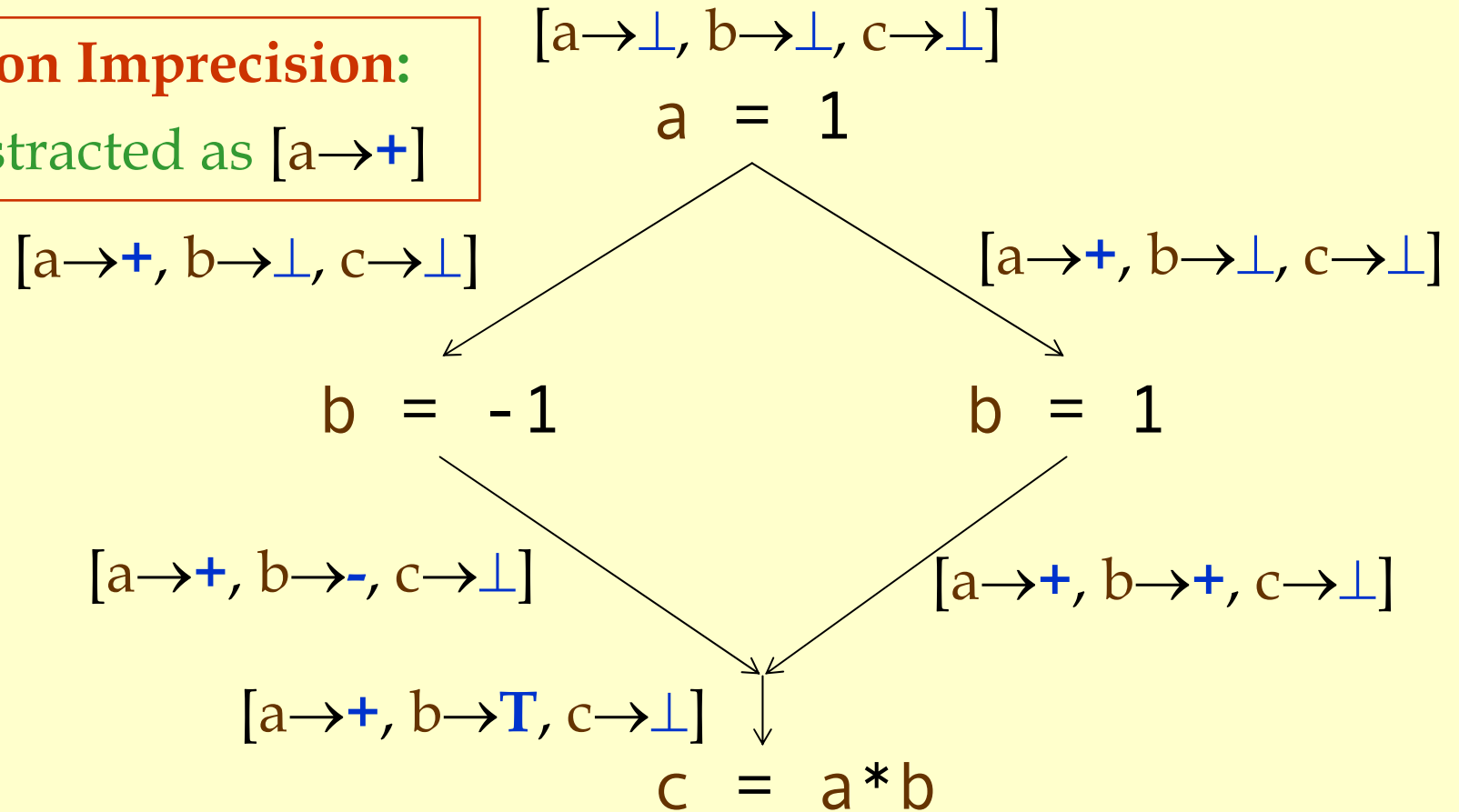
Abstraction Function

- ◆ $AF(s)[v] = \text{sign of } v$
 - ◆ $AF([a \rightarrow 5, b \rightarrow 0, c \rightarrow -2]) = [a \rightarrow +, b \rightarrow \text{zero}, c \rightarrow -]$
- ◆ Establishes meaning of the analysis results
 - ◆ If analysis says a variable v has a given sign
 - ◆ then v always has that sign in actual execution.
- ◆ Two sources of imprecision
 - ◆ **Abstraction Imprecision** – concrete values (integers) abstracted as lattice values ($-$, **zero**, and $+$);
 - ◆ **Control Flow Imprecision** – one lattice value for all different flow of control possibilities.

Imprecision Example

Abstraction Imprecision:

$[a \rightarrow 1]$ abstracted as $[a \rightarrow +]$



Control Flow Imprecision:

$[b \rightarrow \top]$ summarizes results of all executions.

In any execution state s , $AF(s)[b] \neq \top$

$[a \rightarrow +, b \rightarrow \top, c \rightarrow \top]$

General Sources of Imprecision

◆ Abstraction Imprecision

- ◆ Lattice values less precise than execution values.
- ◆ Abstraction function throws away information.

◆ Control Flow Imprecision

- ◆ Analysis result has a single lattice value to summarize results of multiple concrete executions.
- ◆ Join operation \vee moves up in lattice to combine values from different execution paths.
- ◆ Typically if $x \leq y$, then x is more precise than y .

Why Have Imprecision?

ANSWER: To make analysis tractable

- ◆ Conceptually infinite sets of values in execution.
 - ◆ Typically abstracted by finite set of lattice values.
- ◆ Execution may visit infinite set of states.
 - ◆ Abstracted by computing joins of different paths.

Augmented Execution States

- ◆ Abstraction functions for some analyses require augmented execution states.
 - ◆ **Reaching definitions**: states are augmented with the definition that created each value.
 - ◆ **Available expressions**: states are augmented with expression for each value.

Meet Over All Paths Solution

- ◆ What solution would be ideal for a forward dataflow analysis problem?
- ◆ Consider a path $p = n_0, n_1, \dots, n_k, n$ to a node n (note that for all $i, n_i \in \text{pred}(n_{i+1})$)
- ◆ The solution must take this path into account:

$$f_p(\perp) = (f_{n_k}(f_{n_{k-1}}(\dots f_{n_1}(f_{n_0}(\perp)) \dots))) \leq \text{in}_n$$

- ◆ So the solution must have the property that

$$\bigvee \{f_p(\perp) \mid p \text{ is a path to } n\} \leq \text{in}_n$$

and ideally

$$\bigvee \{f_p(\perp) \mid p \text{ is a path to } n\} = \text{in}_n$$

Soundness Proof of Analysis Algorithm

Property to prove:

For all paths p to n , $f_p(\perp) \leq \text{in}_n$

- ◆ Proof is by induction on the length of p .
 - ◆ Uses monotonicity of transfer functions.
 - ◆ Uses following lemma.

Lemma:

The worklist algorithm produces a solution such that

if $n \in \text{pred}(m)$ then $\text{out}_n \leq \text{in}_m$

(That is, what you get out of a predecessor is more precise than what will go in to the node, because precision may be lost by the join function.)

Proof

- ◆ Base case: p is of length 0
 - ◆ Then $p = n_0$ and $f_p(\perp) = \perp = \text{in}_{n_0}$
- ◆ Induction step:
 - ◆ Assume theorem for all paths of length k .
 - ◆ Show for an arbitrary path p of length $k+1$.

Induction Step Proof

- ◆ Given a path $p = n_0, \dots, n_k, n$ show $(f_{n_k}(f_{n_{k-1}}(\dots f_{n_1}(f_{n_0}(\perp)) \dots))) \leq \text{in}_n$

By induction assumption:

$$(f_{n_{k-1}}(\dots f_{n_1}(f_{n_0}(\perp)) \dots)) \leq \text{in}_{n_k}$$

Apply f_{n_k} to both sides:

$$f_{n_k}(f_{n_{k-1}}(\dots f_{n_1}(f_{n_0}(\perp)) \dots)) \quad ? \quad f_{n_k}(\text{in}_{n_k})$$

By monotonicity:

$$(f_{n_k}(f_{n_{k-1}}(\dots f_{n_1}(f_{n_0}(\perp)) \dots))) \leq f_{n_k}(\text{in}_{n_k})$$

By definition of f_{n_k} : $f_{n_k}(\text{in}_{n_k}) = \text{out}_{n_k}$

$$(f_{n_k}(f_{n_{k-1}}(\dots f_{n_1}(f_{n_0}(\perp)) \dots))) \leq \text{out}_{n_k}$$

By lemma: $\text{out}_{n_k} \leq \text{in}_n$

By transitivity:

$$(f_{n_k}(f_{n_{k-1}}(\dots f_{n_1}(f_{n_0}(\perp)) \dots))) \leq \text{in}_n$$

Distributivity

- ◆ Distributivity preserves precision.
- ◆ If framework is distributive, then the worklist algorithm produces the meet over paths solution:

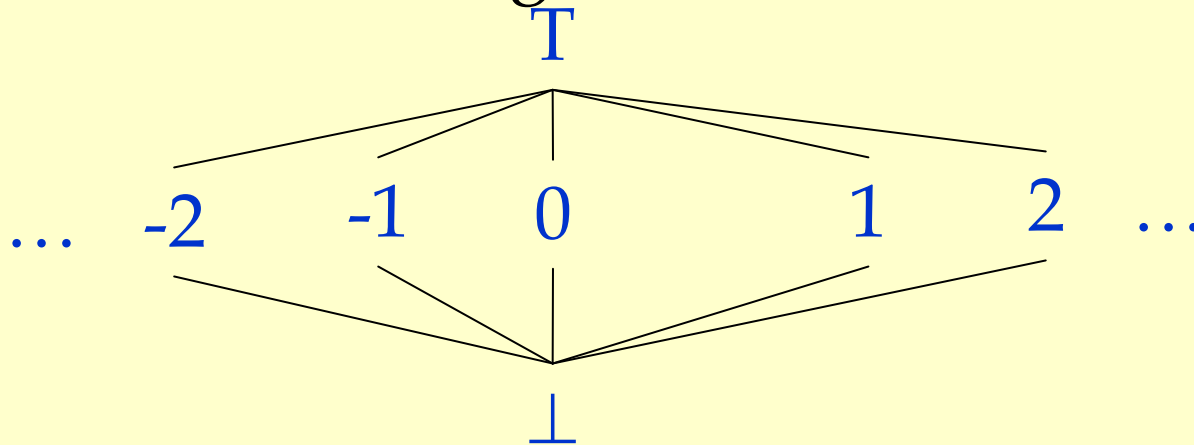
For all n :

$$\bigvee \{f_p(\perp) \mid p \text{ is a path to } n\} = in_n$$

Lack of Distributivity Example

Integer Constant Propagation (ICP)

- ◆ Flat lattice on integers

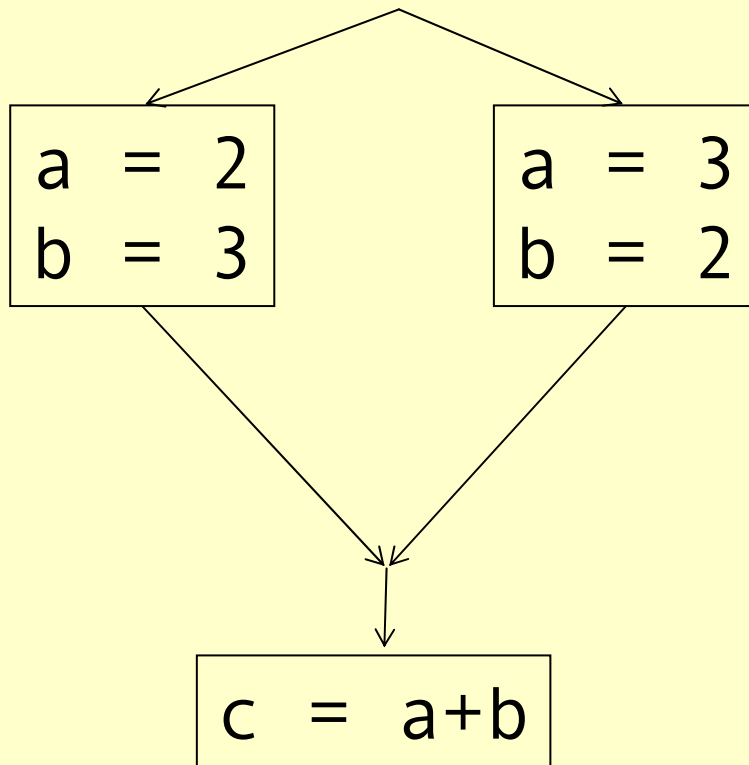


- ◆ Actual lattice records a value for each variable
 - ◆ Example element: $[a \rightarrow 3, b \rightarrow 2, c \rightarrow 5]$

Transfer Functions

- ◆ If n of the form $v = c$
 - ◆ $f_n(x) = x[v \rightarrow c]$
- ◆ If n of the form $v_1 = v_2 + v_3$
 - ◆ $f_n(x) = x[v_1 \rightarrow x[v_2] + x[v_3]]$

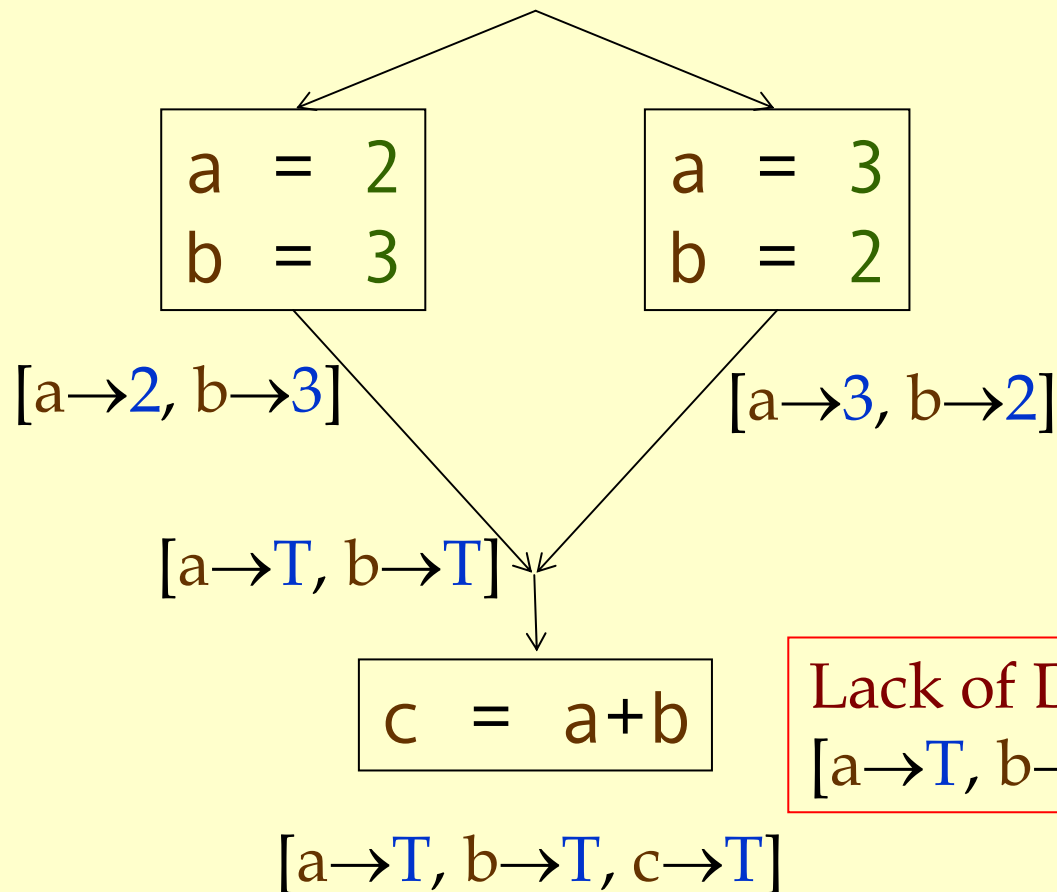
Lack of Distributivity Anomaly



Lack of distributivity of ICP

- ◆ Consider transfer function f for $c = a + b$
 $(f(x) = x[c \rightarrow x[a] + x[b]])$
- ◆ $f([a \rightarrow 3, b \rightarrow 2]) \vee f([a \rightarrow 2, b \rightarrow 3]) =$
 $[a \rightarrow 3, b \rightarrow 2] [c \rightarrow [a \rightarrow 3, b \rightarrow 2][a] + [a \rightarrow 3, b \rightarrow 2][b]] \vee$
 $[a \rightarrow 2, b \rightarrow 3] [c \rightarrow [a \rightarrow 2, b \rightarrow 3][a] + [a \rightarrow 2, b \rightarrow 3][b]] =$
 $[a \rightarrow 3, b \rightarrow 2] [c \rightarrow 3 + 2] \vee [a \rightarrow 2, b \rightarrow 3] [c \rightarrow 2 + 3] =$
 $[a \rightarrow 3, b \rightarrow 2] [c \rightarrow 5] \vee [a \rightarrow 2, b \rightarrow 3] [c \rightarrow 5] =$
 $[a \rightarrow T, b \rightarrow T, c \rightarrow 5]$
- ◆ $f([a \rightarrow 3, b \rightarrow 2] \vee [a \rightarrow 2, b \rightarrow 3]) =$
 $f([a \rightarrow T, b \rightarrow T]) =$
 $[a \rightarrow T, b \rightarrow T] [c \rightarrow [a \rightarrow T, b \rightarrow T][a] + [a \rightarrow T, b \rightarrow T][b]] =$
 $[a \rightarrow T, b \rightarrow T, c \rightarrow T]$

Lack of Distributivity Anomaly



Lack of Distributivity Imprecision:
 $[a \rightarrow T, b \rightarrow T, c \rightarrow 5]$ more precise.

Summary

- ◆ Formal dataflow analysis framework
 - ◆ Lattices, partial orders.
 - ◆ Transfer functions, joins and splits.
 - ◆ Dataflow equations and fixed point solutions.
- ◆ Connection with program
 - ◆ Abstraction function $AF: \mathcal{S} \rightarrow \mathcal{P}$
 - ◆ For any state s and program point n , $AF(s) \leq \text{in}_n$
 - ◆ Meet over paths solutions, distributivity.

Using Program Analysis for Optimization

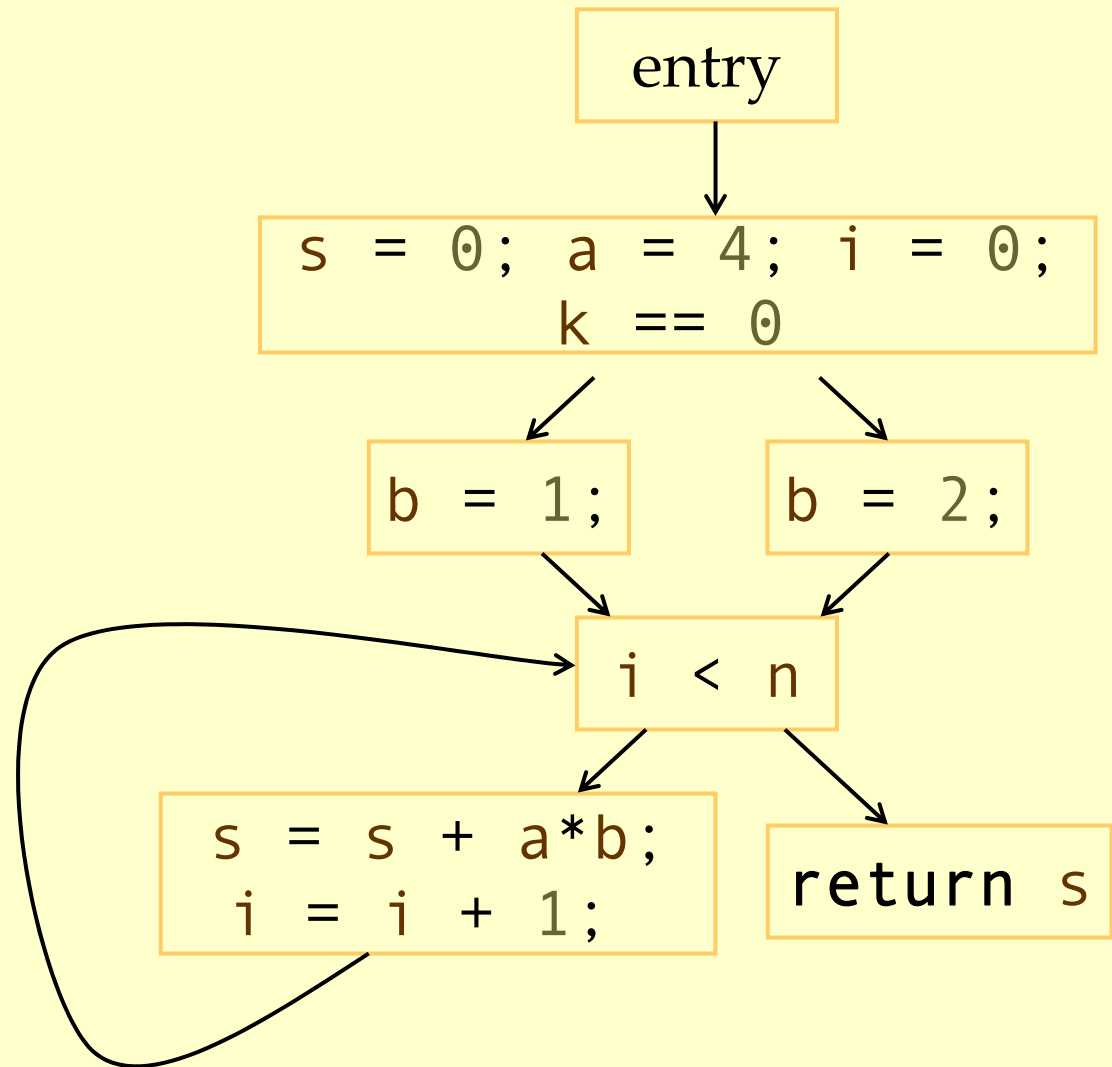
This lecture is primarily based on Konstantinos Sagonas set of slides
(Advanced Compiler Techniques, (2AD518)
at Uppsala University, January-February 2004).
Used with kind permission.

Analysis and Optimizations

- ◆ Program Analysis
 - ◆ Discover properties of a program.
- ◆ Optimizations
 - ◆ Use analysis results to transform the program.
 - ◆ Goal: improve some aspect of the program
 - ◆ number of executed instructions, number of cycles
 - ◆ cache hit rate
 - ◆ memory space (code or data)
 - ◆ power consumption
 - ◆ Has to be safe: Keep the semantics of the program.

Control Flow Graph

```
int add(n, k) {  
    s = 0; a = 4; i = 0;  
    if (k == 0)  
        b = 1;  
    else  
        b = 2;  
    while (i < n) {  
        s = s + a*b;  
        i = i + 1;  
    }  
    return s;  
}
```



Control Flow Graph

- ◆ Nodes represent computation.
 - ◆ Each node is a Basic Block (BB).
 - ◆ Basic Block is a sequence of instructions with:
 - ◆ No branches out of middle of basic block.
 - ◆ No branches into middle of basic block.
 - ◆ Basic blocks should be maximal.
 - ◆ Execution of basic block starts with first instruction.
 - ◆ Includes all instructions in basic block.
- ◆ Edges represent control flow.

Two Kinds of Variables

- ◆ **Temporaries (temps, a tmp):**
 - ◆ Introduced by the compiler.
 - ◆ Transfer values only within basic block.
 - ◆ Introduced as part of instruction flattening.
 - ◆ Introduced by optimizations/transformations.
- ◆ **Program variables (vars, a var):**
 - ◆ Declared in original program.
 - ◆ May transfer values between basic blocks.

Basic Block Optimizations

(Local Optimizations)

◆ Common Sub-Expression Elimination (CSE)

```
a=(x+y)+z; b=x+y;  
t=x+y; a=t+z; b=t;
```

◆ Constant Propagation

```
x=5; b=x+y;  
b=5+y;
```

◆ Algebraic Simplification

```
a=x*1;  
a=x;
```

◆ Copy Propagation

```
a=x+y; b=a; c=b+z;  
a=x+y; b=a; c=a+z;
```

◆ Dead Code Elimination

```
a=x+y; b=a; c=a+z;  
a=x+y; c=a+z
```

◆ Strength Reduction

```
t=i*4;  
t=i<<2;
```

Value Numbering

- ◆ Normalize BB so that all statements are of the form:
 - ◆ $\text{var} = \text{var op var}$ (where op is a binary operator)
 - ◆ $\text{var} = \text{op var}$ (where op is a unary operator)
 - ◆ $\text{var} = \text{var}$
(I.E., no complex statements like $x=a+b*c$.)
- ◆ Simulate execution of basic block:
 - ◆ Assign a virtual value to each variable.
 - ◆ Assign a virtual value to each expression.
 - ◆ Assign a temporary variable to hold value of each computed expression.

Value Numbering for CSE

As we simulate execution of program,
generate a new version of program:

- ◆ Each new value assigned to temporary
 $a = x + y$; becomes
 $a = x + y$; $t_1 = a$;
- ◆ Temporary preserves value for use later in
program even if original variable rewritten
 $a = x + y$; $a = a + z$; becomes
 $a = x + y$; $t_1 = a$; $a = a + z$; $t_2 = a$;

CSE Example

◆ Original

$a = x + y$

$b = a + z$

$b = b + y$

$c = a + z$

◆ After CSE

$a = x + y$

$t_1 = a$

$b = a + z$

$t_2 = b$

$b = b + y$

$t_3 = b$

$c = t_2$

◆ Issues:

◆ CSE with different names:

$a = x; b = x + y; c = a + y;$

◆ Excessive temp generation and use.

Original Basic Block

```

a = x + y
b = a + z
b = b + y
c = a + z
    
```

New Basic Block

```

a = x + y
t1 = a
b = a + z
t2 = b
b = b + y
t3 = b
c = t2
    
```

Var to Val

```

x → V1
y → V2
a → V3
z → V4
b → V6
c → V5
    
```

Exp to Val

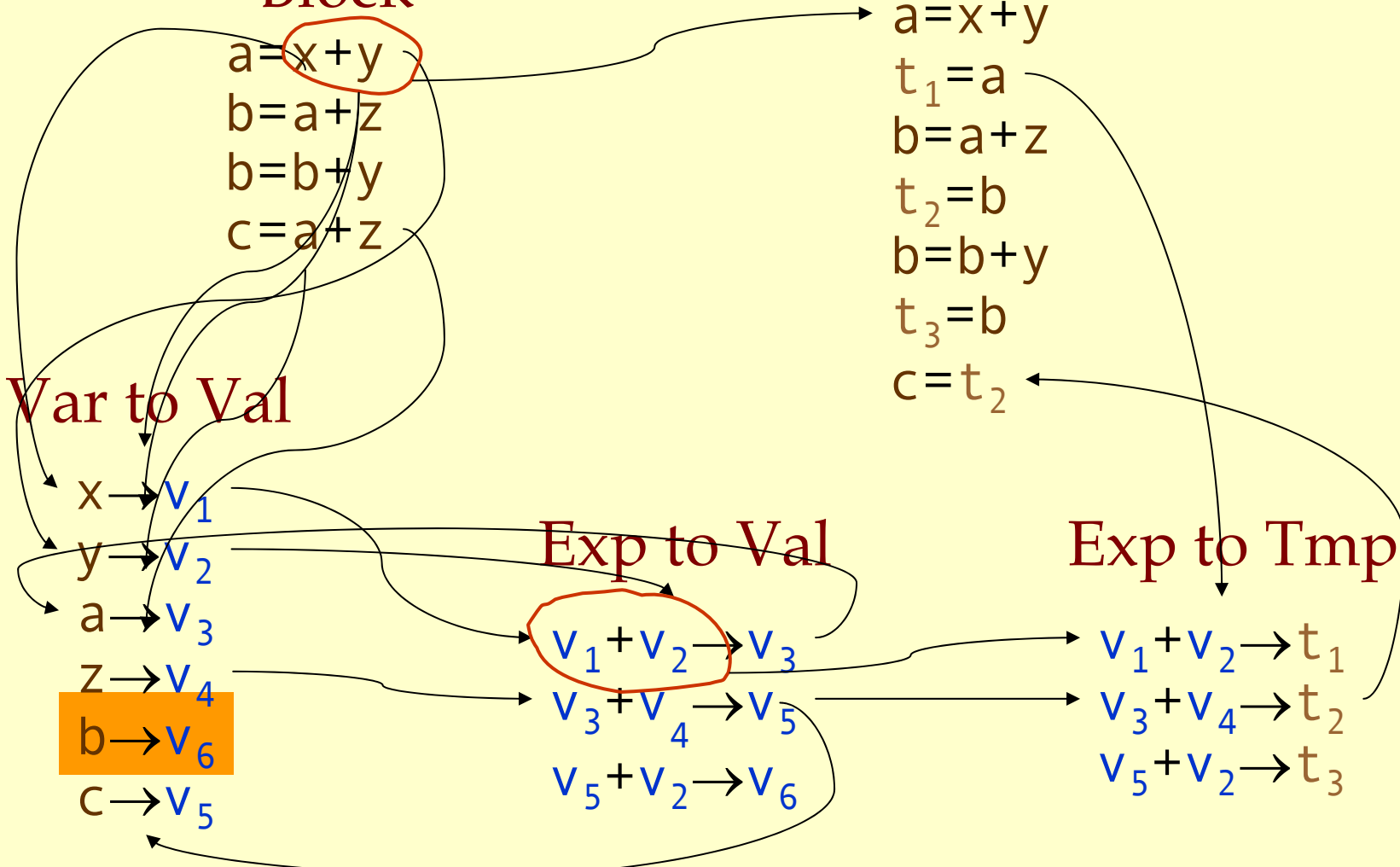
```

V1 + V2 → V3
V3 + V4 → V5
V5 + V2 → V6
    
```

Exp to Tmp

```

V1 + V2 → t1
V3 + V4 → t2
V5 + V2 → t3
    
```



Original Basic Block

$a = x + y$
 $b = a + z$
 $b = b + y$
 $c = a + z$

New Basic Block

$a = x + y$
 $t_1 = a$
 $b = a + z$
 $t_2 = b$
 $b = b + y$
 $t_3 = b$
 $c = t_2$

Var to Val

$x \rightarrow v_1$
 $y \rightarrow v_2$
 $a \rightarrow v_3$
 $z \rightarrow v_4$
 $b \rightarrow v_5$
 $c \rightarrow v_5$

$b \rightarrow v_6$

Exp to Val

$v_1 + v_2 \rightarrow v_3$
 $v_3 + v_4 \rightarrow v_5$
 $v_5 + v_2 \rightarrow v_6$

Exp to Tmp

$v_1 + v_2 \rightarrow t_1$
 $v_3 + v_4 \rightarrow t_2$
 $v_5 + v_2 \rightarrow t_3$

Problems

- ◆ Algorithm has a temporary for each value.
 $a = x + y; \quad t_1 = a;$
- ◆ Introduces
 - ◆ lots of temporaries.
 - ◆ lots of copy statements to temporaries.
- ◆ In many cases, temporaries and copy statements are unnecessary.
- ◆ So we eliminate them with copy propagation and dead code elimination.

Copy Propagation (CP)

- ◆ Once again, simulate execution of program
- ◆ If possible, use the original variable instead of a temporary
 - ◆ $a=x+y$; $b=x+y$;
 - ◆ After CSE becomes $a=x+y$; $t_1=a$; $b=t_1$;
 - ◆ After CP becomes $a=x+y$; $b=a$;
- ◆ **Key idea**: determine when original variables are **NOT** overwritten between computation of stored value and use of stored value.

Copy Propagation Maps

- ◆ Maintain two maps
 - ◆ tmp to var: tells which variable to use instead of a given temporary variable.
 - ◆ var to set: inverse of tmp to var. Tells which temps are mapped to a given variable by tmp to var.

Copy Propagation Example

◆ Original

$a = x + y$

$b = a + z$

$c = x + y$

$a = b$

◆ After CSE

$a = x + y$

$t_1 = a$

$b = a + z$

$t_2 = b$

$c = t_1$

$a = b$

◆ After CSE and Copy Propagation

$a = x + y$

$t_1 = a$

$b = a + z$

$t_2 = b$

$c = a$

$a = b$

Copy Propagation Example

Basic Block
After CSE

```
a=x+y
t1=a
b=a+z
t2=b
c=t1
a=b
```

tmp to var

```
t1→a
t2→b
```

Basic Block After
CSE and Copy Prop

```
a=x+y
t1=a
b=a+z
t2=b
c=a
a=b
```

var to set

```
a→{t1}
b→{t2}
```


Copy Propagation Example

Basic Block
After CSE

```
a=x+y
t1=a
b=a+z
t2=b
c=t1
a=b
```

tmp to var

```
t1→t1
t2→b
```

Basic Block After
CSE and Copy Prop

```
a=x+y
t1=a
b=a+z
t2=b
c=a
a=b
```

var to set

```
a→{ }
b→{ t2 }
```

Dead Code Elimination

- ◆ Copy propagation keeps all temporaries.
- ◆ There may be temps that are never read.
- ◆ Dead Code Elimination removes them.

Basic block after
CSE and Copy Prop.

```
a=x+y  
t1=a  
b=a+z  
t2=b  
c=a  
a=b
```

Basic block after
CSE, CP, &
Dead Code Elimination

```
a=x+y  
b=a+z  
c=a  
a=b
```

Dead Code Elimination

- ◆ Basic idea:
 - ◆ Process code in **reverse** execution order.
 - ◆ Maintain a set of variables that are needed later in computation.
 - ◆ On encountering an assignment to a temporary that is not needed, we remove the assignment.

Basic Block After CSE and Copy Prop

a=x+y
t1=a
b=a+z
t2=b
c=a
⇒ a=b

Needed Set

{ a , z }

{ a , z }

{ a , b , z }

{ a , b }

{ a , b }

{ b }

Interesting Properties

- ◆ Analysis and optimization algorithms simulate execution of the program.
 - ◆ CSE and Copy Propagation go forward.
 - ◆ Dead Code Elimination goes backwards.
- ◆ Optimizations are stacked.
 - ◆ Group of basic transformations.
 - ◆ Work together to get good result.
 - ◆ Often, one transformation creates inefficient code that is cleaned up by following transformations.

Other Basic Block Transformations

- ◆ Constant Propagation.
- ◆ Strength Reduction:
 - ◆ $a * 4; \Rightarrow a \ll 2;$
 - ◆ $3 * a; \Rightarrow a + a + a;$
- ◆ Algebraic Simplification:
 - ◆ $a * 1; \Rightarrow a;$
 - ◆ $b + 0; \Rightarrow b;$
- ◆ Unified transformation framework.

Dataflow Analysis

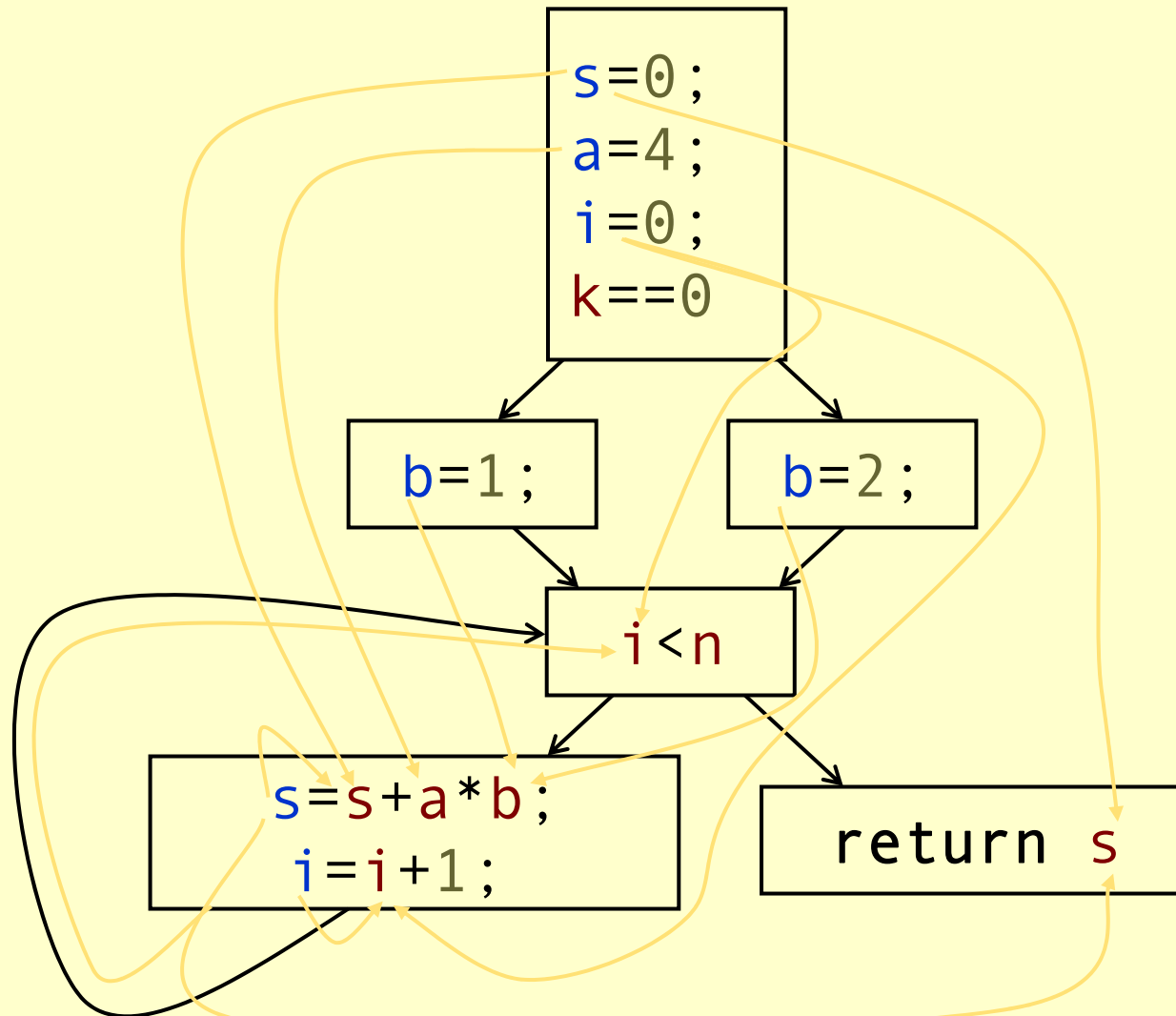
(Global Analysis)

- ◆ Used to determine properties of programs that involve multiple basic blocks.
- ◆ Typically used to enable transformations.
 - ◆ common sub-expression elimination.
 - ◆ constant and copy propagation.
 - ◆ dead code elimination.
- ◆ Analysis and transformation often come in pairs.

Reaching Definitions

- ◆ Concept of *definition* and *use*
 - ◆ $a = x + y$
 - ◆ is a definition of a .
 - ◆ is a use of x and y .
- ◆ A *definition* reaches a *use* if value written by *definition* may be read by *use*.

Reaching Definitions



Reaching Definitions and Constant Propagation

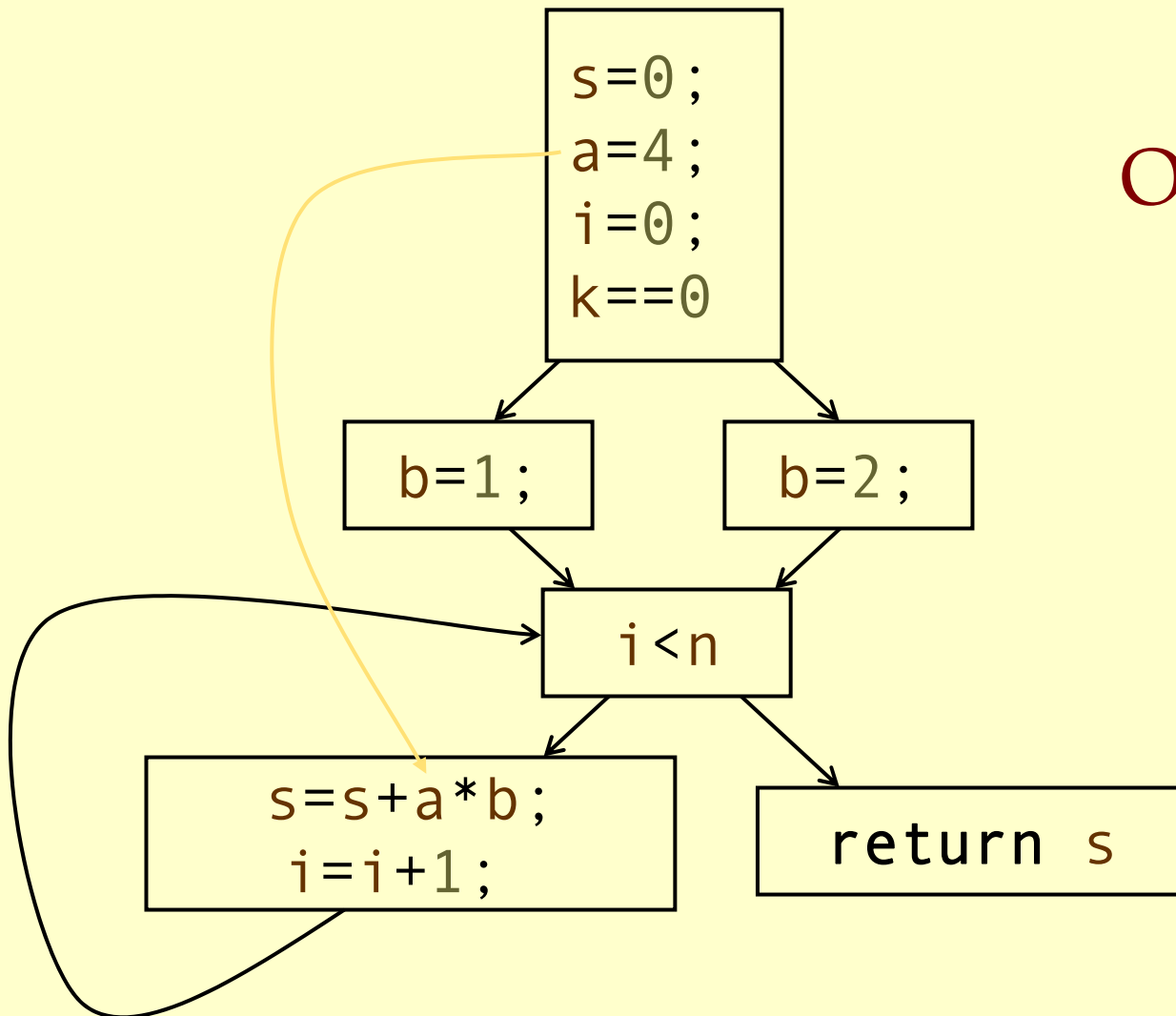
- ◆ Is a **use** of a variable a constant?
 - ◆ Check all reaching definitions.
 - ◆ If all assign variable to same constant.
 - ◆ Then use is in fact a constant.
- ◆ Can replace variable with constant.

Is a constant **in** $s=s+a*b$?

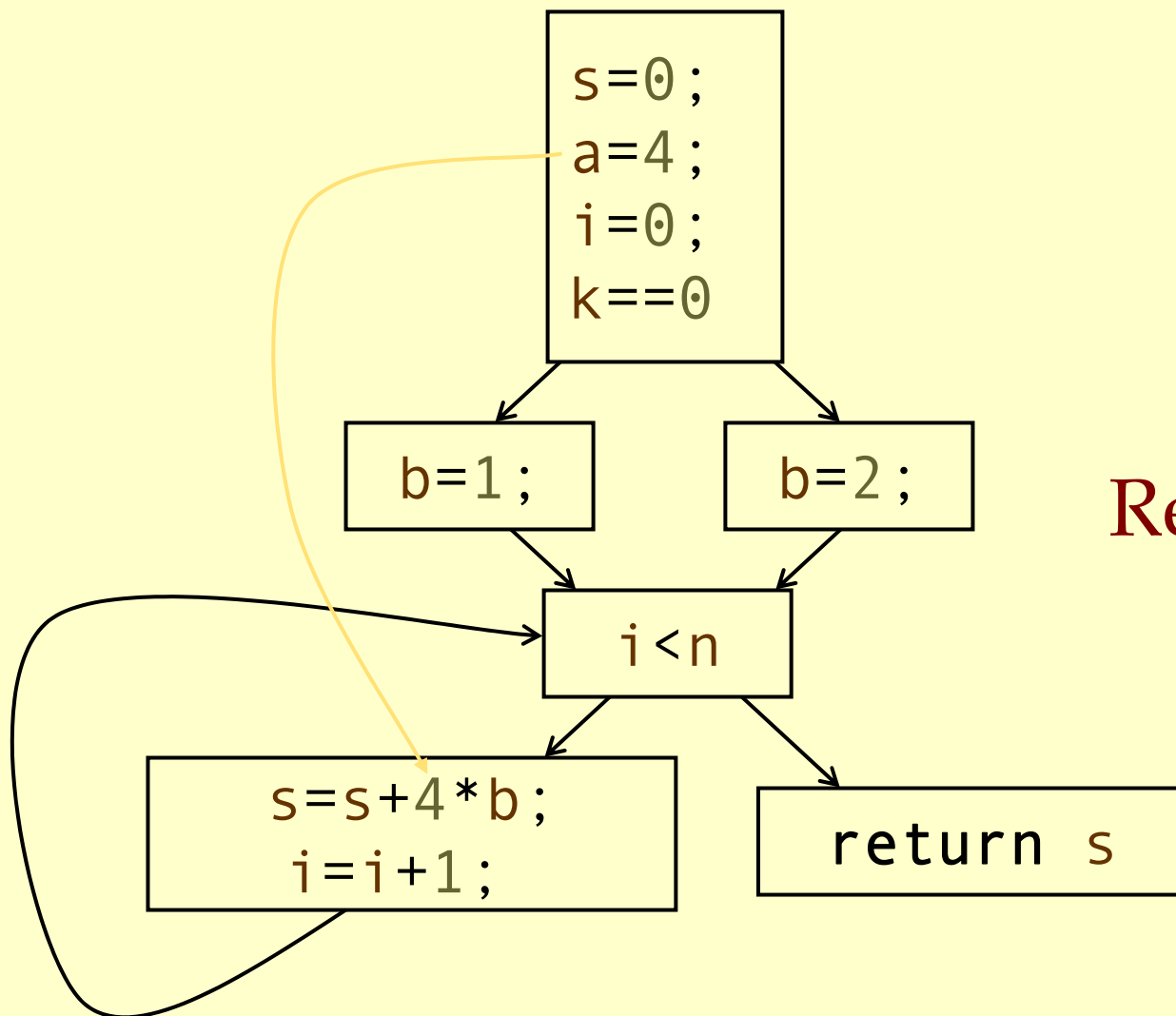
Yes!

On all reaching
definitions

$a=4$



Constant Propagation Transform



Yes!

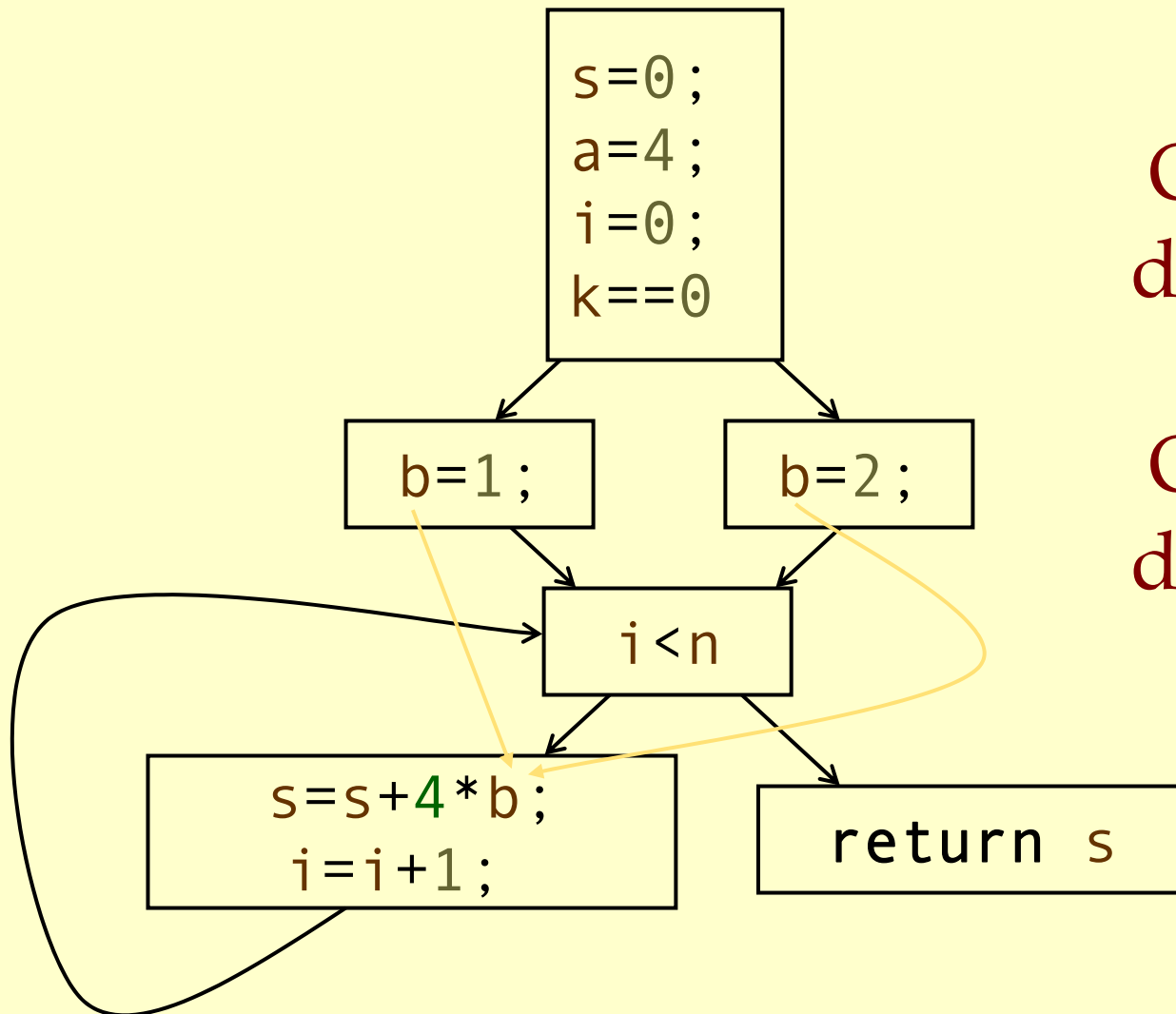
`a=4`

in

`s=s+a*b`

Replace use of `a`
with `4`.

Is b constant in $s = s + 4 * b$?



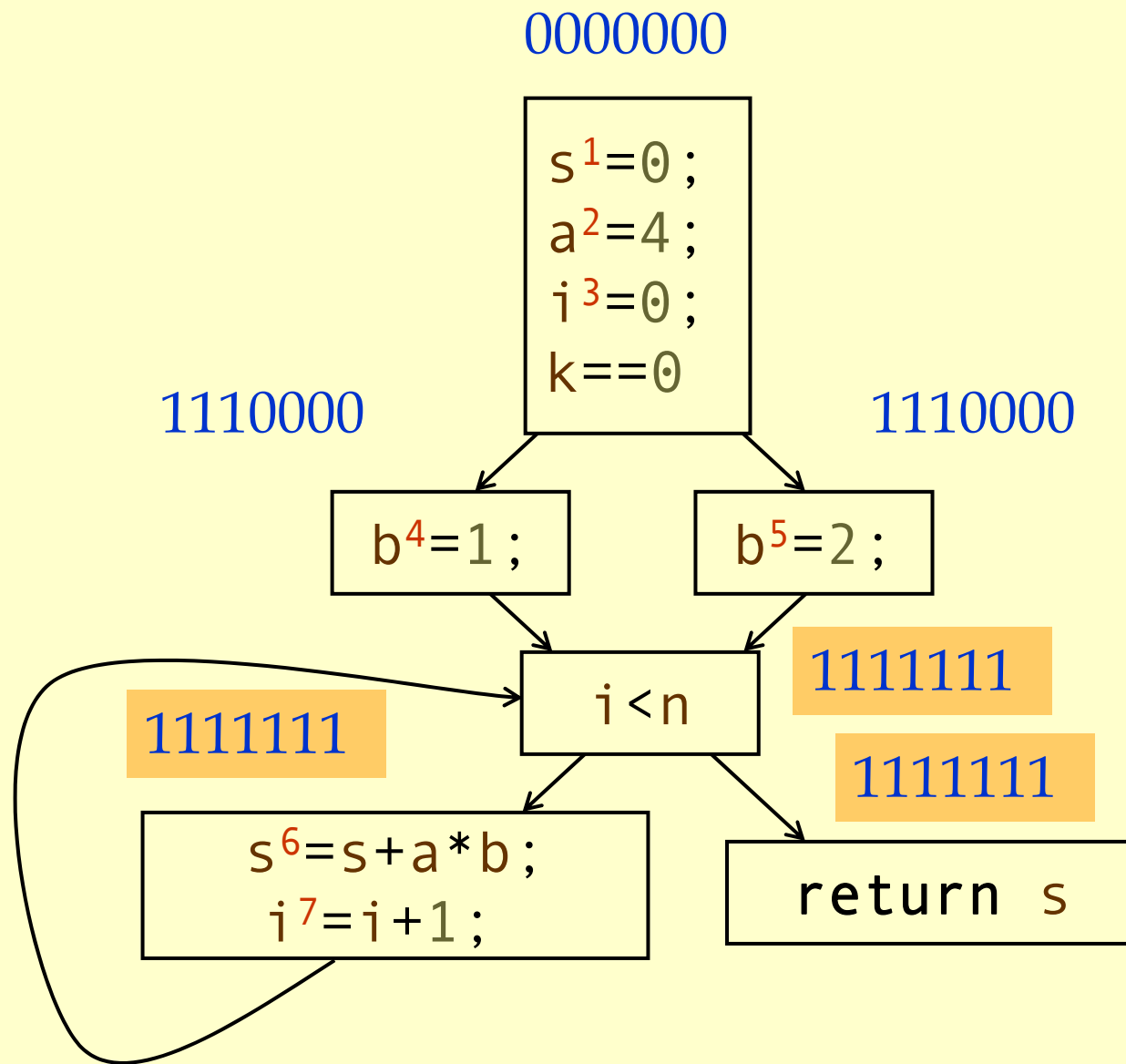
No!

One reaching
definition with
 $b=1$

One reaching
definition with
 $b=2$

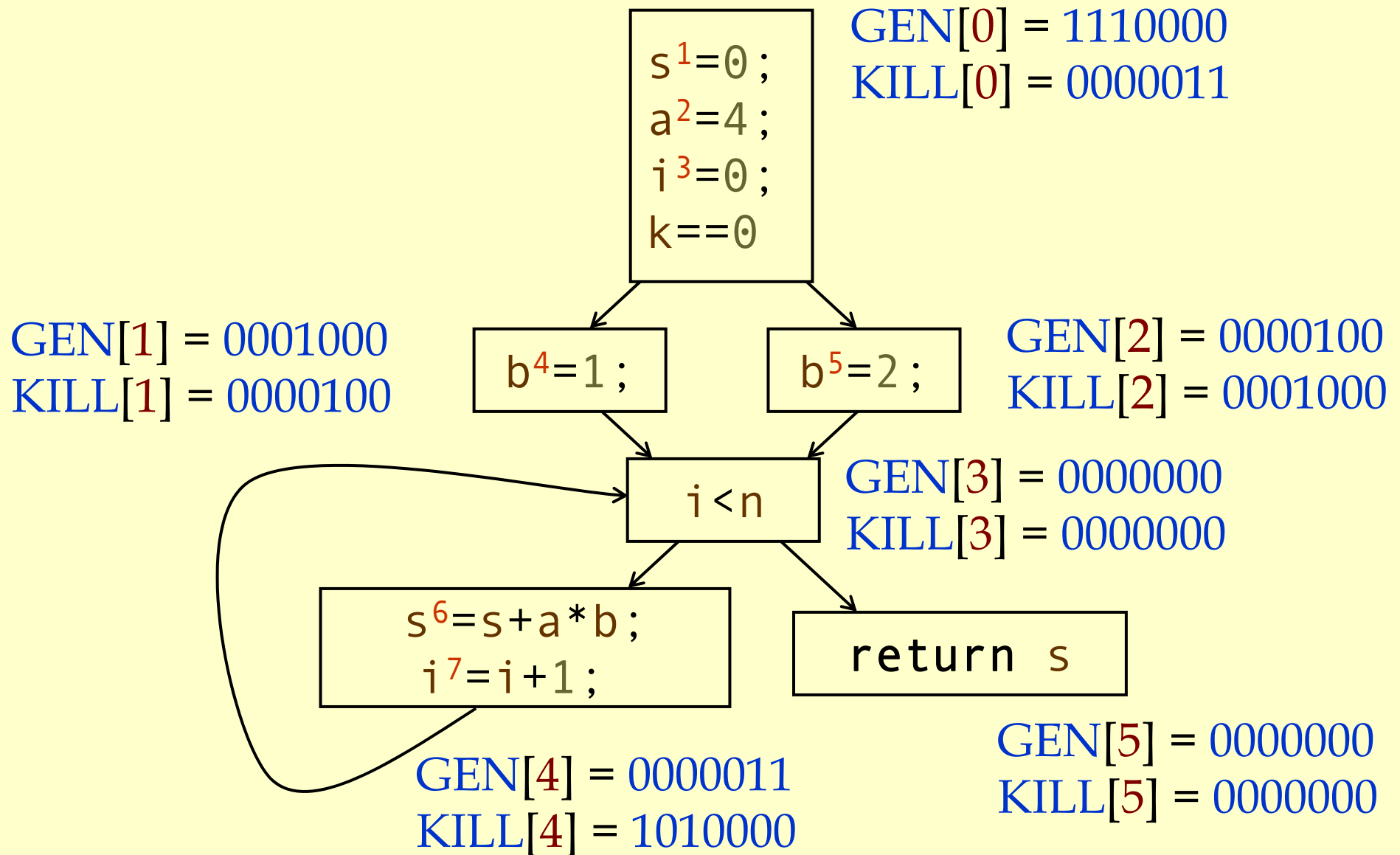
Computing Reaching Definitions

- ◆ Compute with sets of definitions:
 - ◆ Represent sets using bit vectors.
 - ◆ Each definition has a position in bit vector.
- ◆ At each basic block, compute:
 - ◆ Definitions that reach start of block.
 - ◆ Definitions that reach end of block.
- ◆ Do computation by simulating execution of program until the fixed point is reached.



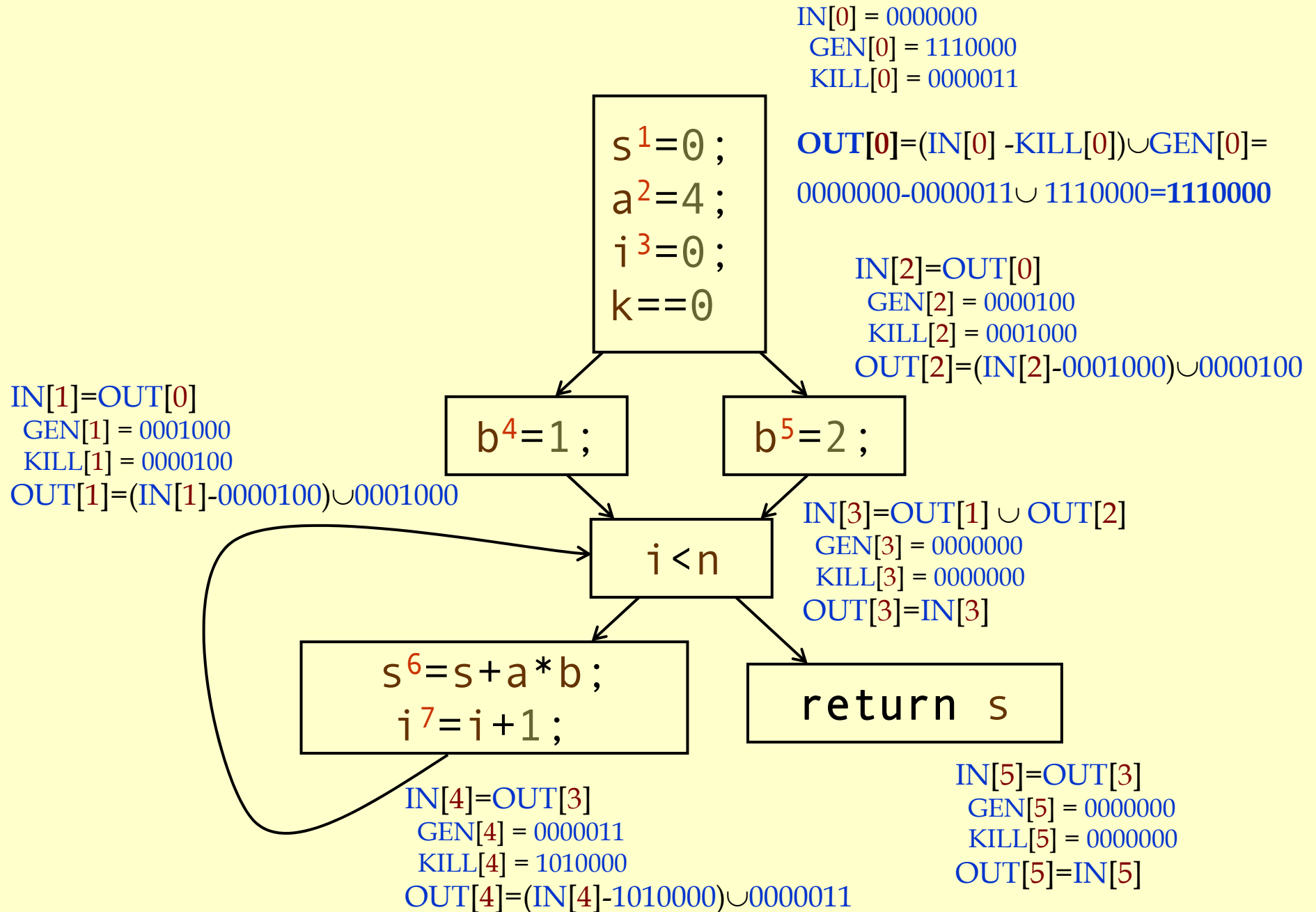
Formalizing Analysis

- ◆ Each basic block has
 - ◆ **IN** - set of definitions that reach beginning of block
 - ◆ **OUT** - set of definitions that reach end of block
 - ◆ **GEN** - set of definitions generated in block
 - ◆ **KILL** - set of definitions killed in the block
- ◆ $\text{GEN}[s^6 = s + a * b; i^7 = i + 1;] = 0000011$
- ◆ $\text{KILL}[s^6 = s + a * b; i^7 = i + 1;] = 1010000$
- ◆ Compiler scans each basic block to derive **GEN** and **KILL** sets.



Dataflow Equations

- ◆ $IN[b_i] = OUT[b_1] \cup \dots \cup OUT[b_n]$
where b_1, \dots, b_n are predecessors of b_i
- ◆ $OUT[b_i] = (IN[b_i] - KILL[b_i]) \cup GEN[b_i]$
- ◆ $IN[entry] = 00000000$
- ◆ **Result:** system of equations.



Solving Equations

- ◆ Use fix point algorithm.
- ◆ Initialize with solution of $OUT[b_i] = 0000000$
- ◆ Repeatedly apply equations:
 - ◆ $IN[b_i] = OUT[b_1] \cup \dots \cup OUT[b_n]$
 - ◆ $OUT[b_i] = (IN[b_i] - KILL[b_i]) \cup GEN[b_i]$
- ◆ Until reach fixed point, i.e., until equation application has no further effect.
- ◆ Use a worklist to track which equation applications may have further effect.

Reaching Definitions Algorithm

```

for all nodes  $n \in \mathbb{N}$ 
   $OUT[n] = \emptyset;$ 
   $Changed = \mathbb{N};$ 
while ( $Changed \neq \emptyset$ )
  choose  $n \in Changed;$ 
   $Changed = Changed - \{n\};$ 
   $OldOut = OUT[n]$ 
   $IN[n] = \emptyset;$ 
  for all nodes  $p \in predecessors(n)$ 
     $IN[n] = IN[n] \cup OUT[p];$ 
   $OUT[n] = (IN[n] - KILL[n]) \cup GEN[n];$ 
  if ( $OUT[n] \neq OldOut$ )
    for all nodes  $s \in successors(n)$ 
       $Changed = Changed \cup \{s\};$ 
  // Or  $OUT[n] = GEN[n];$ 
  //  $\mathbb{N}$  = all nodes in graph
  // Until fixed point reached.
  // Node from worklist
  // Remove from worklist
  // Remember old result
  // Calculate  $IN$  as join
  //   of predecessors.
  // Recalculate  $OUT$ 
  // If  $OUT[n]$  changed
  // Add succs to worklist

```

Questions

- ◆ Does the algorithm halt?
 - ◆ yes, because transfer function is monotonic.
 - ◆ if increase **IN**, increase **OUT**.
 - ◆ in limit, all bits are **1**.
- ◆ If bit is **1**, is there always an execution in which corresponding definition reaches basic block?
- ◆ If bit is **0**, does the corresponding definition ever reach basic block?
- ◆ Concept of conservative analysis.

Available Expressions

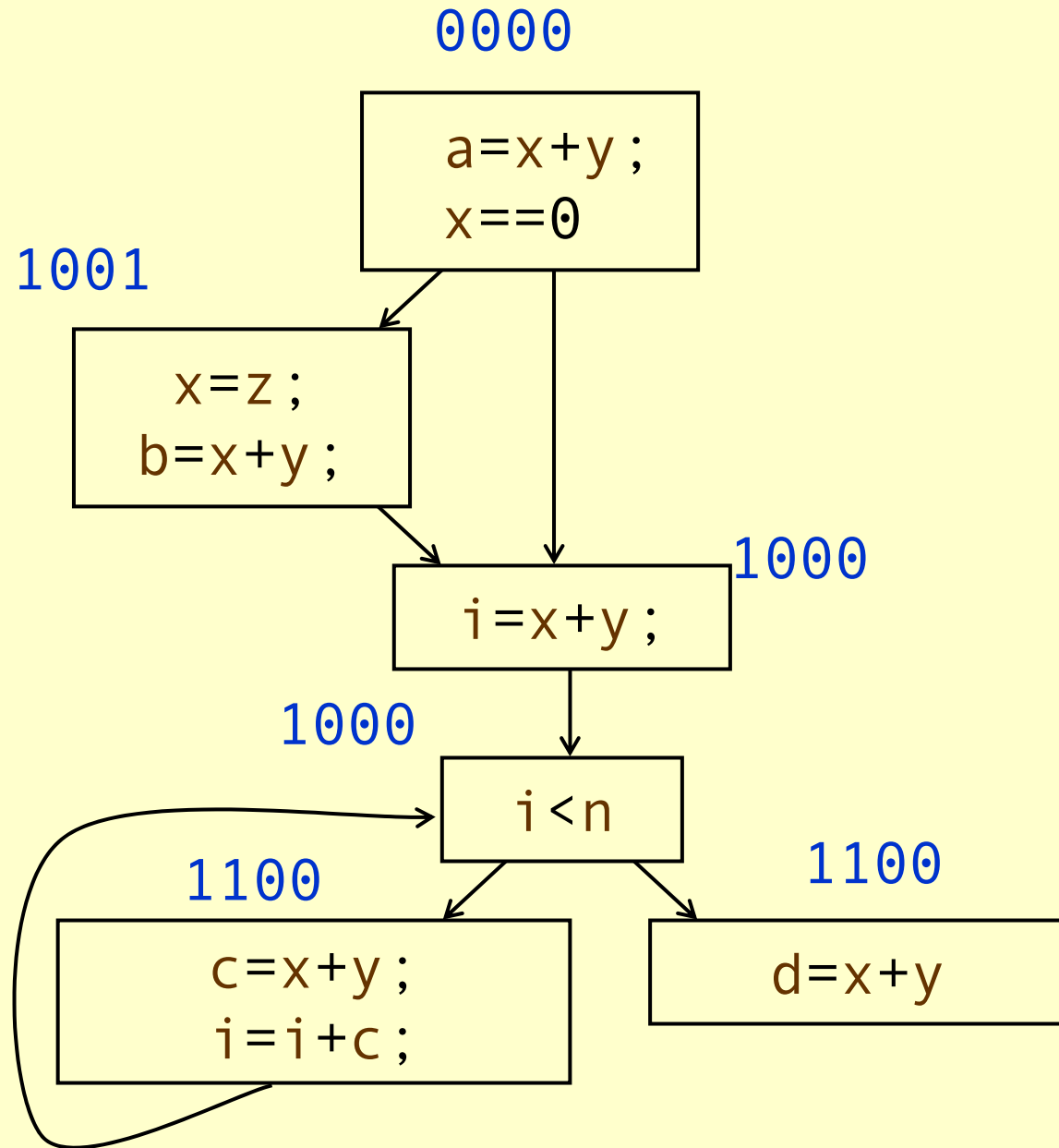
- ◆ An expression $x+y$ is available at a point p if
 - ◆ **every** path from the initial node to p evaluates $x+y$ before reaching p ,
 - ◆ and there are **no assignments** to x or y after the evaluation but before p .
- ◆ Available Expression information can be used to do global (across basic blocks) CSE.
- ◆ If an expression is available at use, there is no need to re-evaluate it.

Computing Available Expressions

- ◆ Represent sets of expressions using bit vectors.
- ◆ Each expression corresponds to a bit.
- ◆ Run dataflow algorithm similar to reaching definitions.
- ◆ **Big difference:**
 - ◆ **Definition** reaches a basic block if it comes from **ANY** predecessor in CFG.
 - ◆ **Expression** is available at a basic block only if it is available from **ALL** predecessors in CFG.

Expressions

- 1: $x+y$
- 2: $i < n$
- 3: $i+c$
- 4: $x == 0$

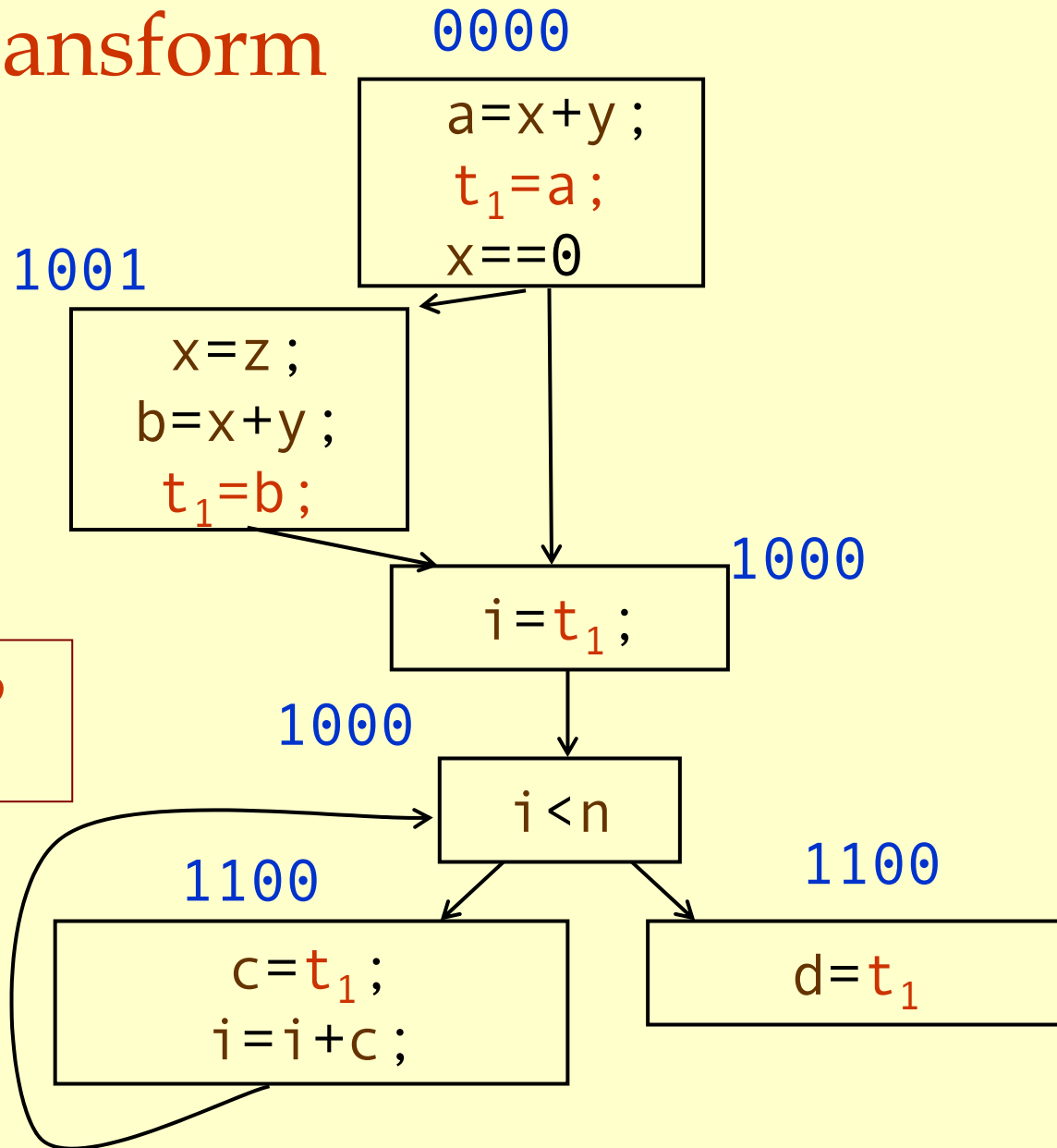


Global CSE Transform

Expressions

- 1: $x+y$
- 2: $i < n$
- 3: $i+c$
- 4: $x == 0$

Must use same temp for CSE in all blocks



Formalizing Analysis

- ◆ Each basic block has
 - IN** - set of expressions that reach beginning of block.
 - OUT** - set of expressions that reach end of block.
 - GEN** - set of expressions generated in block.
 - KILL** - set of expressions killed in the block.
- ◆ $\text{GEN}[x=z ; b=x+y] = 1000$
- ◆ $\text{KILL}[x=z ; b=x+y] = 1001$
- ◆ Compiler scans each basic block to derive **GEN** and **KILL** sets.

Dataflow Equations

- ◆ $IN[b_i] = OUT[b_1] \cap \dots \cap OUT[b_n]$
 - ◆ where b_1, \dots, b_n are predecessors of b_i
- ◆ $OUT[b_i] = (IN[b_i] - KILL[b_i]) \cup GEN[b_i]$
- ◆ $IN[entry] = 0000$
- ◆ **Result:** system of equations.

Solving Equations

- ◆ Use fix point algorithm.
- ◆ $IN[entry] = 0000$
- ◆ Initialize with solution of $OUT[b_i] = 1111$
- ◆ Repeatedly apply equations:
 - ◆ $IN[b_i] = OUT[b_1] \cap \dots \cap OUT[b_n]$
 - ◆ $OUT[b_i] = (IN[b_i] - KILL[b_i]) \cup GEN[b_i]$
- ◆ Use a worklist to track which equation applications may have further effect.

Available Expressions Algorithm

```

for all nodes  $n \in N$            //  $\mathbb{E}$  is set of all expressions.
     $OUT[n] = \mathbb{E}$ ;           //  $OUT[n] = \mathbb{E} - KILL[n]$ ;
Changed =  $N$ ;                 //  $N$  = all nodes in graph
while (Changed  $\neq \emptyset$ )
    choose  $n \in \text{Changed}$ ;
    Changed = Changed -  $\{n\}$ ;
     $IN[n] = \mathbb{E}$ ;
    OldOut =  $OUT[n]$ 
    for all nodes  $p \in \text{predecessors}(n)$ 
         $IN[n] = IN[n] \cap OUT[p]$ ;
     $OUT[n] = (IN[n] - KILL[n]) \cup GEN[n]$ ;
    if ( $OUT[n] \neq \text{OldOut}$ )
        for all nodes  $s \in \text{successors}(n)$  Changed = Changed  $\cup \{s\}$ ;

```

Questions

- ◆ Does algorithm always halt?
- ◆ If expression is available in some execution, is it always marked as available in analysis?
- ◆ If expression is not available in some execution, can it be marked as available in analysis?
- ◆ In what sense is the algorithm conservative?

Duality In Two Algorithms

- ◆ Reaching definitions
 - ◆ Confluence operation is set **union**.
 - ◆ **OUT**[b] initialized to **empty set**.
- ◆ Available expressions
 - ◆ Confluence operation is set **intersection**.
 - ◆ **OUT**[b] initialized to **set of available expressions**.
- ◆ General framework for dataflow algorithms.
- ◆ Build parameterized dataflow analyzer once, use for all dataflow problems.

Liveness Analysis

- ◆ A variable v is live at point p if
 - ◆ v is used along some path starting at p , and
 - ◆ no definition of v along the path before the use.
- ◆ When is a variable v dead at point p ?
 - ◆ No use of v on any path from p to exit node, or
 - ◆ If all paths from p , redefine v before using v .

What Use is Liveness Information?

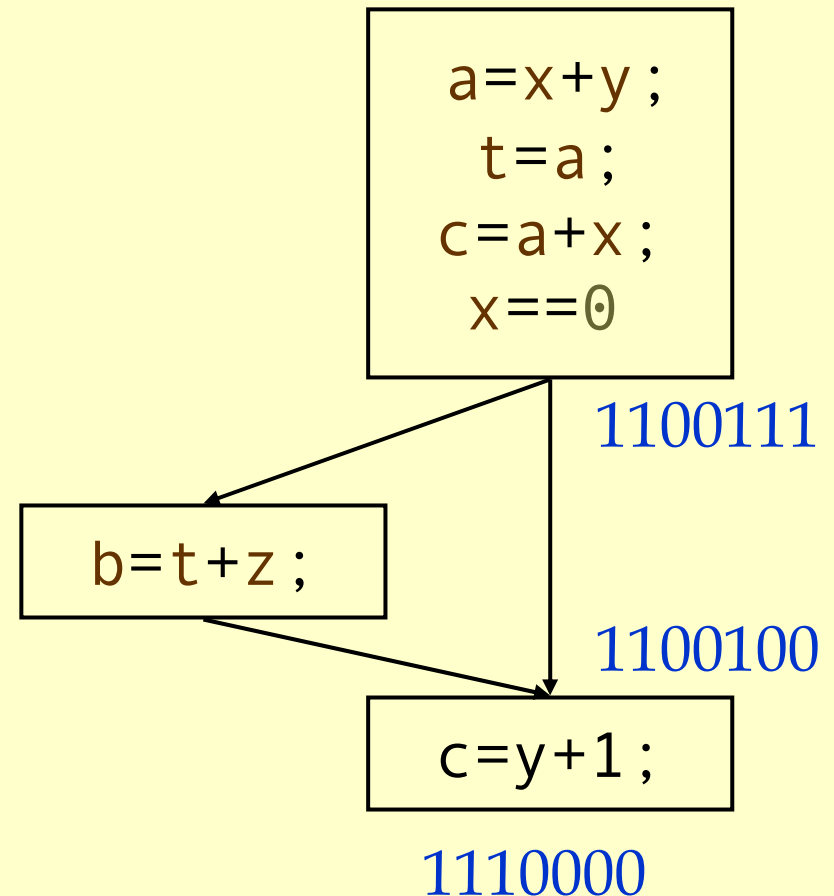
- ◆ Register allocation.
 - ◆ If a variable is dead, we can reassign its register.
- ◆ Dead code elimination.
 - ◆ Eliminate assignments to variables not read later.
 - ◆ But must not eliminate last assignment to variable (such as instance variable) visible outside CFG.
 - ◆ Can eliminate other dead assignments.
 - ◆ Handle by making all externally visible variables live on exit from CFG.

Conceptual Idea of Analysis

- ◆ Simulate execution.
- ◆ But start from exit and go **backwards** in CFG.
- ◆ Compute liveness information from end to beginning of basic blocks.

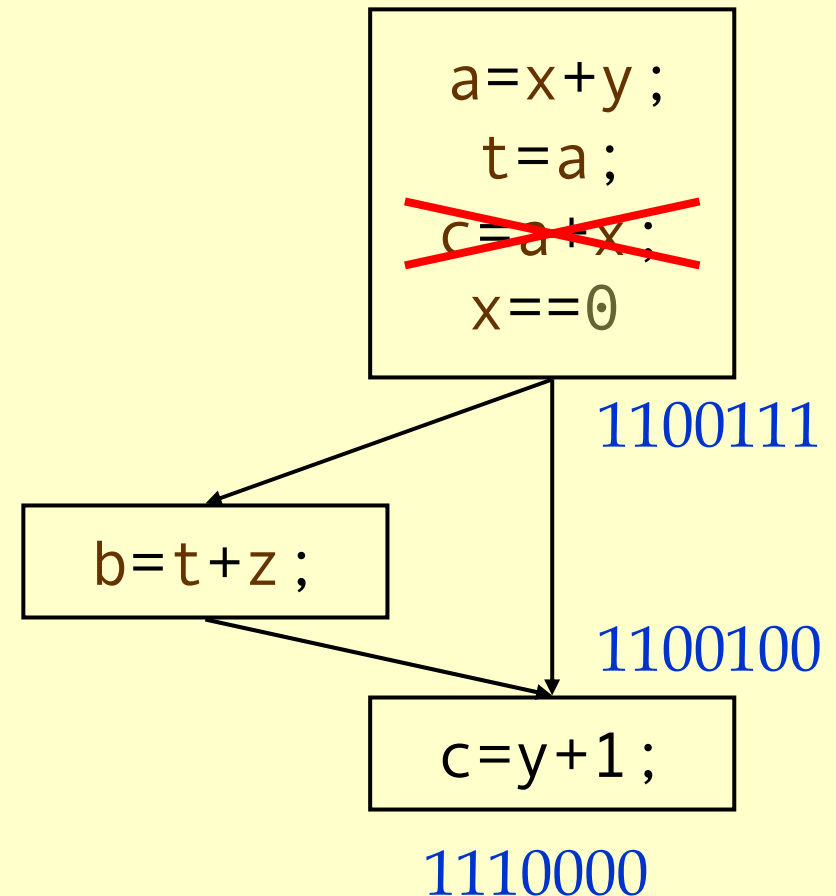
Liveness Example

- ◆ Assume a, b, c visible outside function. They are live on exit.
- ◆ Assume x, y, z, t are not visible.
- ◆ Represent liveness using a bit vector: order is $abcxyz t$.



Using Liveness Information for Dead Code Elimination

- ◆ Assume a, b, c visible outside function. They are live on exit.
- ◆ Assume x, y, z, t are not visible.
- ◆ Represent liveness using a bit vector: order is $abcxyz t$.



Formalizing Analysis

- ◆ Each basic block has
 - IN** - set of variables live at start of block.
 - OUT** - set of variables live at end of block.
 - USE** - set of variables with upwards exposed uses in block.
(**GEN**)
 - DEF** - set of variables defined in block. (**KILL**)
- ◆ $USE[x=z ; x=x+1 ; y=1 ;] = \{z\}$ (x not in **USE**)
- ◆ $DEF[x=z ; x=x+1 ; y=1 ;] = \{x, y\}$
- ◆ Compiler scans each basic block to derive **USE** and **DEF** sets.

Algorithm

```
OUT[Exit] =  $\emptyset$ ;  
IN[Exit] = USE[n];  
for all nodes  $n \in \mathbb{N} - \{\text{Exit}\}$   
    IN[n] =  $\emptyset$ ;  
Changed =  $\mathbb{N} - \{\text{Exit}\}$ ;  
while (Changed  $\neq \emptyset$ )  
    choose  $n \in \text{Changed}$ ;  
    Changed = Changed - {n};  
    OldIn = IN[n]  
    OUT[n] =  $\emptyset$ ;  
    for all nodes  $s \in \text{successors}(n)$  OUT[n] = OUT[n]  $\cup$  IN[s];  
    IN[n] = USE[n]  $\cup$  (OUT[n] - DEF[n]);  
    if (IN[n]  $\neq$  OldIn)  
        for all nodes  $p \in \text{predecessors}(n)$  Changed = Changed  $\cup$  {p};
```

Similar to Other Dataflow Algorithms

- ◆ Backwards analysis, not forwards.
- ◆ Still have transfer functions.
- ◆ Still have confluence operators.
- ◆ Can generalize framework to work for both forwards and backwards analyses.

Analysis Information Inside Basic Blocks

- ◆ One detail:
 - ◆ Given dataflow information at **IN** and **OUT** of node.
 - ◆ Also need to compute information at each statement of basic block.
 - ◆ Simple propagation algorithm usually works fine.
 - ◆ Can be viewed as restricted case of dataflow analysis.

Summary

- ◆ Basic blocks and basic block optimizations.
 - ◆ Copy and constant propagation.
 - ◆ Common sub-expression elimination.
 - ◆ Dead code elimination.
- ◆ Dataflow Analysis
 - ◆ Control flow graph.
 - ◆ $IN[b]$, $OUT[b]$, transfer functions, join points.
- ◆ Pairs of analyses and transformations:
 - ◆ **Reaching definitions**/constant propagation.
 - ◆ **Available expressions**/common sub-expression elimination.
 - ◆ Liveness analysis/Dead code elimination.

Building SSA Form

This lecture is primarily based on Konstantinos Sagonas set of slides
(Advanced Compiler Techniques, (2AD518)
at Uppsala University, January-February 2004).

Used with kind permission.
(In turn based on Keith Cooper's slides)

What is SSA?

SSA-form:

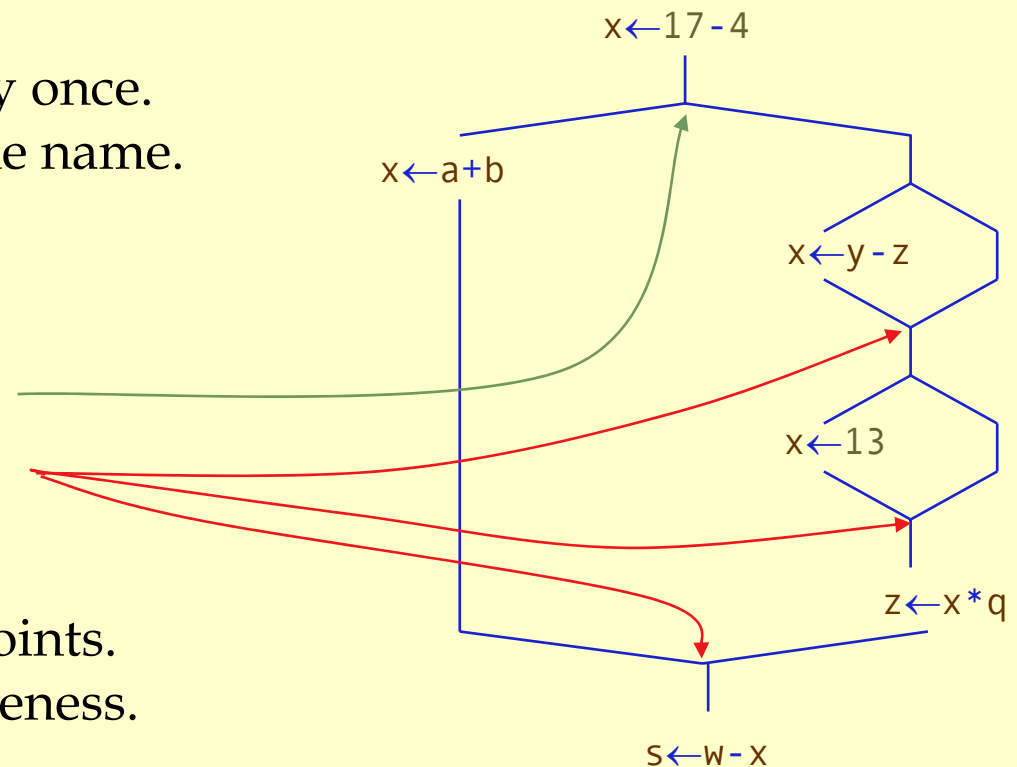
- ◆ Each name is defined exactly once.
- ◆ Each use refers to exactly one name.

What's hard?

- ◆ Straight-line code is trivial.
- ◆ Splits in the CFG are trivial.
- ◆ Joins in the CFG are hard.

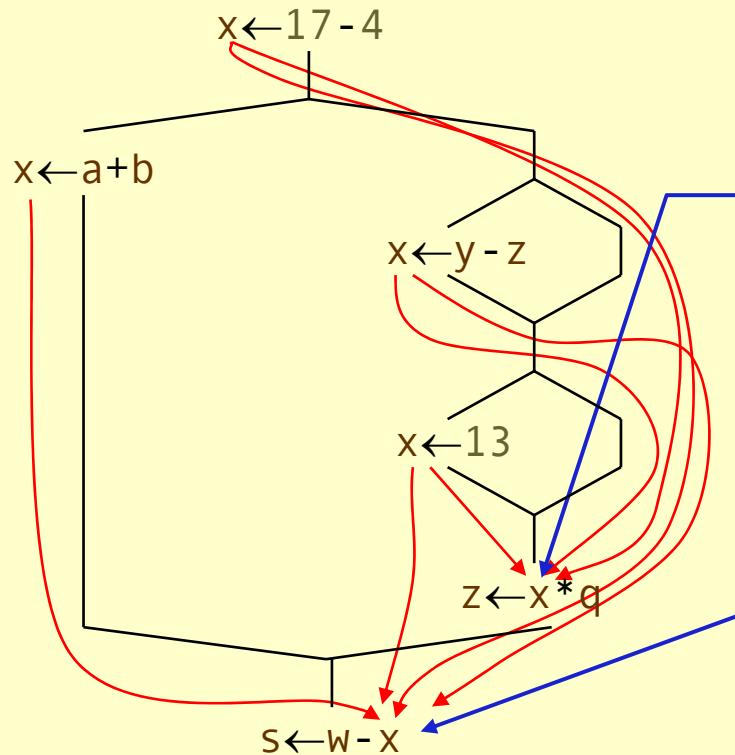
Building SSA Form:

- ◆ Insert Φ -functions at birth points.
- ◆ Rename all values for uniqueness.



Birth Points (*a notion due to Tarjan*)

Consider the flow of values in this example



The value x appears everywhere.
It takes on several values.

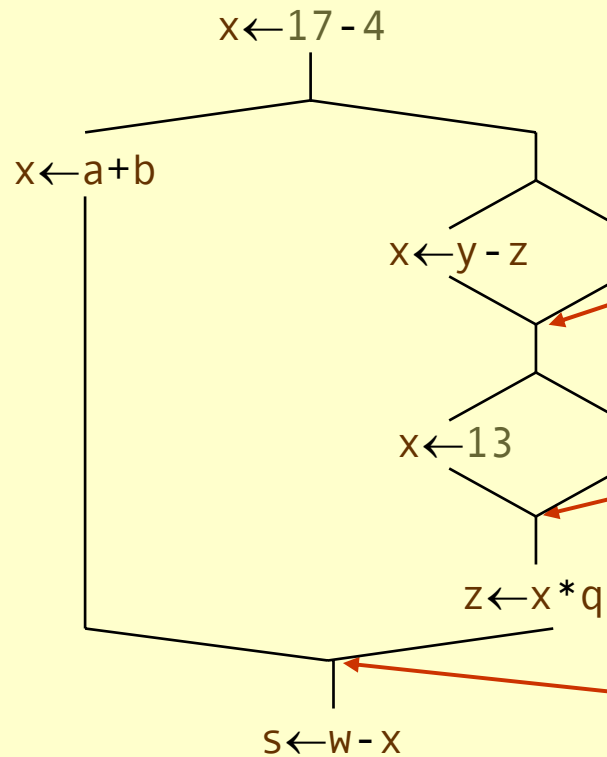
- Here, x can be 13 , $y - z$, or $17 - 4$.
- Here, it can also be $a + b$.

If each value has its own name ...

- Need a way to merge these distinct values.
- Values are “born” at merge points.

Birth Points (cont)

Consider the flow of values in this example



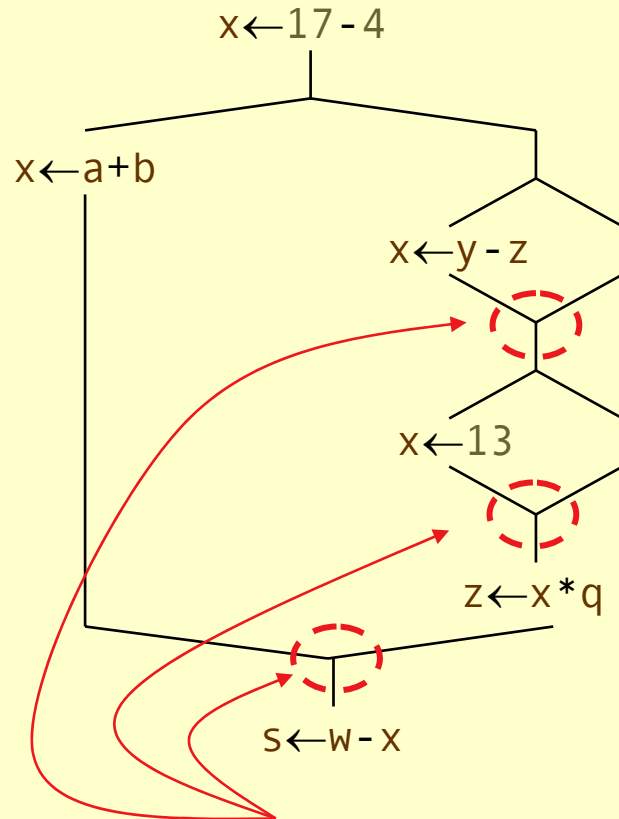
New value for x here
17 - 4 or y - 2

New value for x here
13 or (17 - 4 or y - 2)

New value for x here
a + b or ((13 or (17 - 4 or y - 2)))

Birth Points (cont)

Consider the flow of values in this example



- All birth points are join points
- Not all join points are birth points
- Birth points are value-specific ...

These are all birth points for values

Static Single Assignment Form

SSA-form:

- ◆ Each name is defined exactly once.
- ◆ Each use refers to exactly one name.

What's hard?

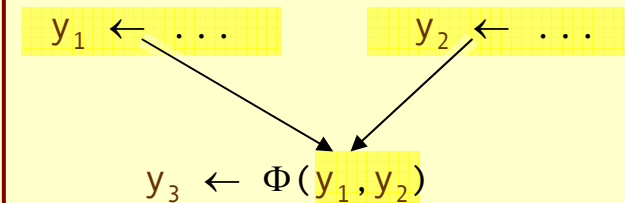
- ◆ Straight-line code is trivial.
- ◆ Splits in the CFG are trivial.
- ◆ Joins in the CFG are hard.

Building SSA Form:

- ◆ Insert Φ -functions at birth points.
- ◆ Rename all values for uniqueness.

A Φ -function is a special kind of copy that selects one of its parameters.

The choice of parameter is governed by the CFG edge along which control reached the current block.



However, real machines do not implement a Φ -function in hardware.

SSA Construction Algorithm (High-level sketch)

1. Insert Φ -functions.
2. Rename values.

... that's all ...

... of course, there is some bookkeeping to be done ...

SSA Construction Algorithm

(Less high-level)

1. Insert Φ -functions at every join for every name.
2. Solve *reaching definitions*.
3. Rename each use to the def that reaches it.
(*will be unique*)

Reaching Definitions

The equations

$$\text{REACHES}(\mathcal{N}_0) = \emptyset$$

$$\text{REACHES}(\mathcal{N}) = \bigcup_{\mathcal{P} \in \text{preds}(\mathcal{N})} \text{DEFOUT}(\mathcal{P}) \cup (\text{REACHES}(\mathcal{P}) \cap \text{SURVIVED}(\mathcal{P}))$$

Domain is |DEFINITIONS|, same as number of operations

- ◆ **REACHES**(\mathcal{N}) is the set of definitions that reach block \mathcal{N}
- ◆ **DEFOUT**(\mathcal{N}) is the set of definitions in \mathcal{N} that reach the end of \mathcal{N}
- ◆ **SURVIVED**(\mathcal{N}) is the set of definitions not obscured by a new def in \mathcal{N}

Computing **REACHES**(\mathcal{N})

- ◆ Use any data-flow method *(i.e., the iterative method)*
- ◆ This particular problem has a very-fast solution *(Zadeck)*

F.K. Zadeck, "Incremental data-flow analysis in a structured program editor," *Proceedings of the SIGPLAN 84 Conf. on Compiler Construction*, June, 1984, pages 132-143.

SSA Construction Algorithm (Less high-level)

1. Insert Φ -functions at **every join** for **every name**.
2. Solve *reaching definitions*.
3. Rename each use to the def that reaches it.

(*will be unique*)

Builds maximal SSA

What's wrong with this approach?

- ◆ Too many Φ -functions. (*precision*)
- ◆ Too many Φ -functions. (*space*)
- ◆ Too many Φ -functions. (*time*)
- ◆ Need to relate edges to Φ -functions parameters. (*bookkeeping*)

To do better, we need a more complex approach.

SSA Construction Algorithm (Less high-level)

1. Insert Φ -functions

a.) calculate dominance frontiers

Moderately complex

b.) find global names

for each name, build a list of blocks that define it

c.) insert Φ -functions

Compute list of blocks where each name is assigned & use as a worklist

\forall global name n

\forall block B in which n is defined

\forall block D in B 's dominance frontier

This adds to the worklist!

Creates the iterated dominance frontier

{ insert a Φ -function for n in D
add D to n 's list of defining blocks

Use a checklist to avoid putting blocks on the worklist twice;
keep another checklist to avoid inserting the same Φ -function twice.

SSA Construction Algorithm (Less high-level)

2. Rename variables in a pre-order walk over dominator tree

(use an array of stacks, one stack per global name)

Starting with the root block, B

a.) generate unique names for each Φ -function
and push them on the appropriate stacks

1 counter per
name for
subscripts

b.) rewrite each operation in the block

i. Rewrite uses of global names with the current version
(from the stack)

ii. Rewrite definition by inventing & pushing new name

c.) fill in Φ -function parameters of successor blocks

Need the end-of-
block name for
this path

d.) recurse on B 's children in the dominator tree

e.) <on exit from block B > pop names generated in B from stacks

Reset the state

Aside on Terminology: Dominators

Definitions

\mathcal{X} dominates \mathcal{Y} if and only if every path from the entry of the control-flow graph to the node for \mathcal{Y} includes \mathcal{X}

- ◆ By definition, \mathcal{X} dominates \mathcal{X}
- ◆ We associate a set of dominators (**Dom**) with each node
- ◆ $|\text{Dom}(x)| \geq 1$

Immediate dominators

- ◆ For any node \mathcal{X} , there must be a \mathcal{Y} in **Dom**(\mathcal{X}) closest to \mathcal{X}
- ◆ We call this \mathcal{Y} the immediate dominator of \mathcal{X}
- ◆ As a matter of notation, we write this as **IDom**(\mathcal{X})

Dominators (cont)

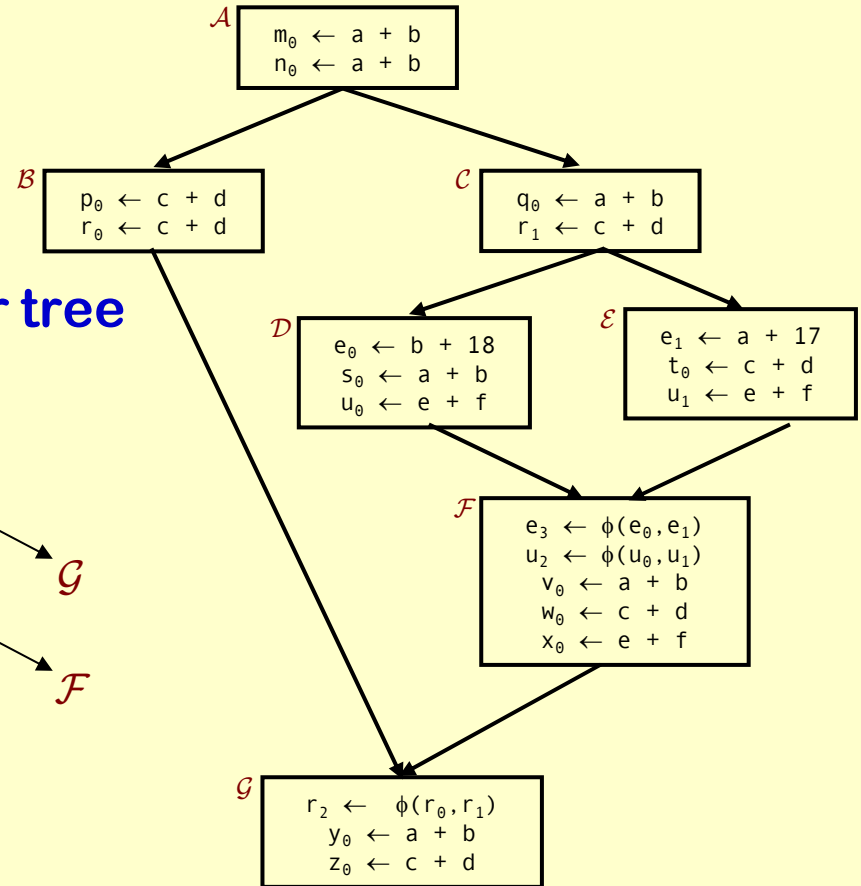
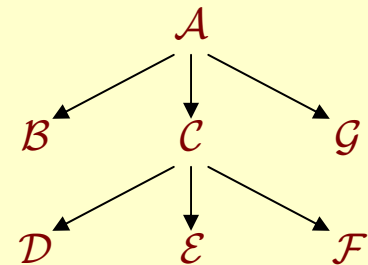
Dominators have many uses in program analysis & transformation:

- ◆ Finding loops.
- ◆ Building SSA form.
- ◆ Making code motion decisions.

Dominator sets

Block	Dom	IDom
<i>A</i>	<i>A</i>	-
<i>B</i>	<i>A, B</i>	<i>A</i>
<i>C</i>	<i>A, C</i>	<i>A</i>
<i>D</i>	<i>A, C, D</i>	<i>C</i>
<i>E</i>	<i>A, C, E</i>	<i>C</i>
<i>F</i>	<i>A, C, F</i>	<i>C</i>
<i>G</i>	<i>A, G</i>	<i>A</i>

Dominator tree



Let's look at how to compute dominators...

SSA Construction Algorithm (Low-level detail)

Computing Dominance

- ◆ First step in Φ -function insertion computes dominance.
- ◆ A node \mathcal{N} dominates \mathcal{M} iff \mathcal{N} is on every path from \mathcal{N}_0 to \mathcal{M}
 - ◆ Every node dominates itself
 - ◆ \mathcal{N} 's immediate dominator is its closest dominator, $\text{IDom}(\mathcal{M})^\dagger$

$$\text{DOM}(\mathcal{N}_0) = \{\mathcal{N}_0\}$$

$$\text{DOM}(\mathcal{M}) = \{\mathcal{M}\} \cup \left(\bigcap_{\mathcal{P} \in \text{preds}(\mathcal{M})} \text{DOM}(\mathcal{P}) \right)$$

Initially, $\text{Dom}(n) = \mathbb{N}, \forall n \neq n_0$

Computing DOM

- ◆ These equations form a rapid data-flow framework
- ◆ Iterative algorithm will solve them in $d(\mathbb{G}) + 3$ passes
 - ◆ Each pass does $|\mathbb{N}|$ unions & $|\mathbb{E}|$ intersections,
 - ◆ \mathbb{E} is $O(\mathbb{N}^2) \Rightarrow O(\mathbb{N}^2)$ work

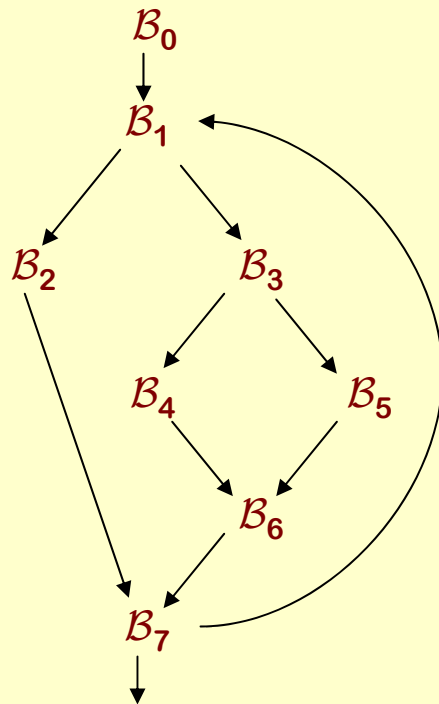
$d(\mathbb{G})$ is the loop-connectedness of the graph w.r.t a DFST

- Maximal number of back edges in an acyclic path.
- Several studies suggest that, in practice, $d(\mathbb{G})$ is small. (< 3)
- For most CFGs, $d(\mathbb{G})$ is independent of the specific DFST.

$^\dagger \text{IDom}(\mathcal{M}) \neq \mathcal{N}$, unless \mathcal{N} is \mathcal{N}_0 , by convention.

Example

Control Flow Graph



Progress of iterative solution for **DOM**

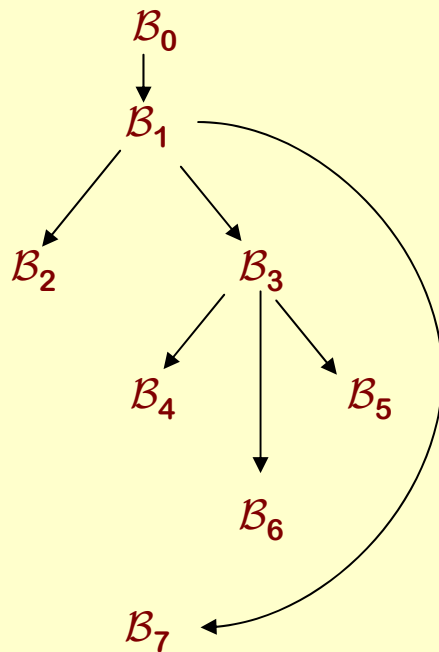
Iter- ation	DOM (<i>n</i>)							
	0	1	2	3	4	5	6	7
0	0	N	N	N	N	N	N	N
1	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7
2	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7

Results of iterative solution for **DOM** & **IDom**

	0	1	2	3	4	5	6	7
DOM	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7
IDom	0	0	1	1	3	3	3	1

Example

Dominance Tree



Progress of iterative solution for **Dom**

Iter- ation	Dom (<i>n</i>)							
	0	1	2	3	4	5	6	7
0	0	∅	∅	∅	∅	∅	∅	∅
1	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7
2	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7

Results of iterative solution for **Dom** & **IDom**

	0	1	2	3	4	5	6	7
Dom	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7
IDom	0	0	1	1	3	3	3	1

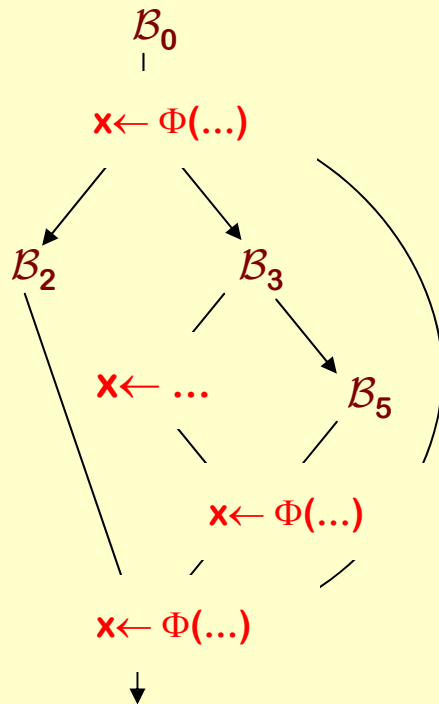
There are asymptotically faster algorithms.

With the right data structures, the iterative algorithm can be made faster.

See Cooper, Harvey, and Kennedy.

Example

Dominance Frontiers



Dominance Frontiers & Φ -Function Insertion

- A definition at \mathcal{N} forces a Φ -function at \mathcal{M} iff $\mathcal{N} \notin \text{DOM}(\mathcal{M})$ but $\mathcal{N} \in \text{DOM}(\mathcal{P})$ for some $\mathcal{P} \in \text{preds}(\mathcal{M})$
- $\text{DF}(\mathcal{M})$ is the fringe just beyond the region that \mathcal{N} dominates.

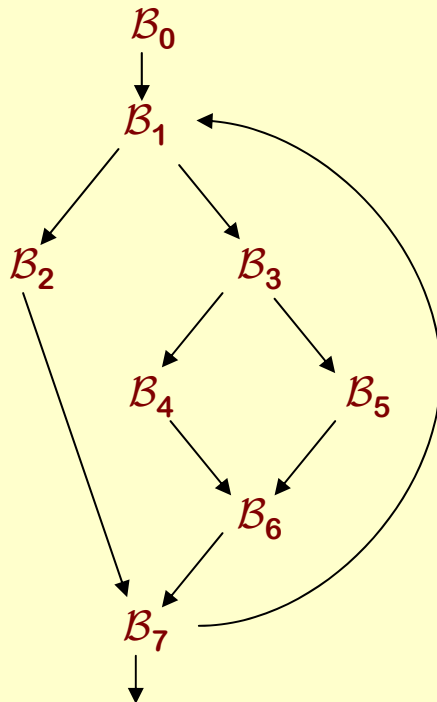
	0	1	2	3	4	5	6	7
DOM	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7
DF	-	-	7	7	6	6	7	1

- $\text{DF}(B_4)$ is $\{B_6\}$, so \leftarrow in B_4 forces a Φ -function in B_6
- \leftarrow in B_6 forces a Φ -function in $\text{DF}(B_6) = \{B_7\}$
- \leftarrow in B_7 forces a Φ -function in $\text{DF}(B_7) = \{B_1\}$
- \leftarrow in B_1 forces a Φ -function in $\text{DF}(B_1) = \emptyset$ (*halt*)

For each assignment, we insert the Φ -functions

Example

Dominance Frontiers



Computing Dominance Frontiers

- Only join points are in $DF(\mathcal{N})$ for some \mathcal{N}
- Leads to a simple, intuitive algorithm for computing dominance frontiers

For each join point \mathcal{M} (i.e., $|preds(\mathcal{M})| > 1$)

For each CFG predecessor of \mathcal{M}

Run up to $IDOM(\mathcal{M})$ in the dominator tree, adding \mathcal{M} to $DF(\mathcal{N})$ for each \mathcal{N} between \mathcal{M} and $IDOM(\mathcal{M})$

	0	1	2	3	4	5	6	7
DOM	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7
DF	-	-	7	7	6	6	7	1

- For some applications, we need post-dominance, the post-dominator tree, and reverse dominance frontiers, $RDF(\mathcal{N})$
 - > Just dominance on the reverse CFG
 - > Reverse the edges & add unique exit node
- We will use these in dead code elimination

SSA Construction Algorithm (Reminder)

1. Insert Φ -functions at every join for every name

a.) calculate dominance frontiers

b.) find global names

Needs a little more detail

for each name, build a list of blocks that define it

c.) insert Φ -functions

\forall global name n

\forall block B in which n is defined

\forall block D in B 's dominance frontier

insert a Φ -function for n in D

add D to n 's list of defining blocks

SSA Construction Algorithm

Finding global names

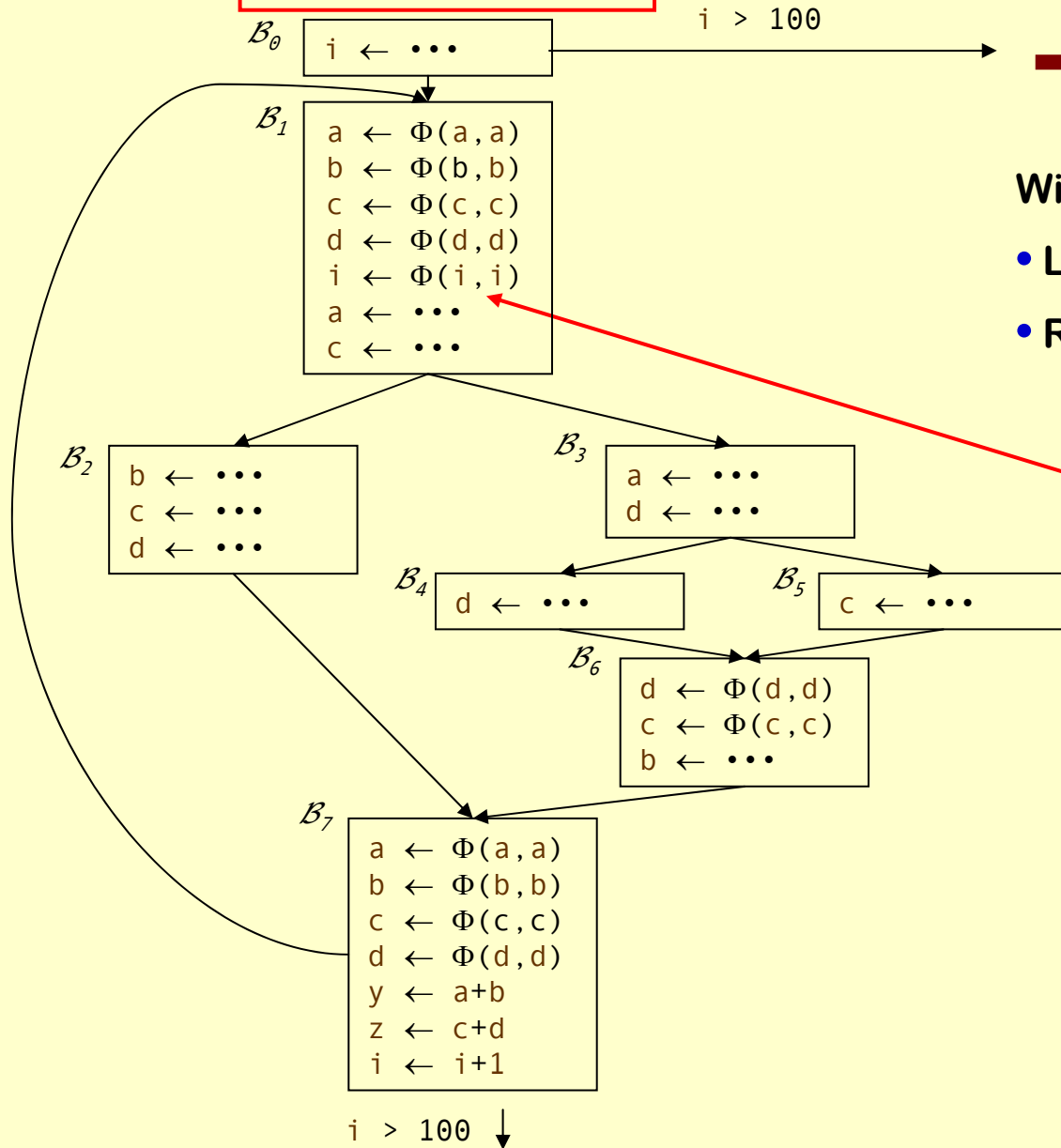
- ◆ Different between two forms of SSA
- ◆ Minimal uses all names
- ◆ Semi-pruned SSA uses names that are *live* on entry to some block
 - ◆ Shrinks name space & number of Φ -functions
 - ◆ Pays for itself in compile-time speed
- ◆ For each “global name”, need a list of blocks where it is defined
 - ◆ Drives Φ -function insertion
 - ◆ \mathcal{B} defines x implies a Φ -function for x in every $\mathcal{C} \in \text{DF}(\mathcal{B})$

Otherwise, we do not need a Φ -function

Pruned SSA adds a test to see if x is live at insertion point

Assume $a, b, c, & d$
defined before B_0

Example



With all the Φ -functions

- Lots of new ops
- Renaming is next

Excluding
local names
avoids Φ 's for
 y & z

SSA Construction Algorithm (Less high-level)

2. Rename variables in a pre-order walk over dominator tree
(use an array of stacks, one stack per global name)
Starting with the root block, \mathcal{B}
 - a.) generate unique names for each Φ -function
and push them on the appropriate stacks
 - b.) rewrite each operation in the block
 - i. Rewrite uses of global names with the current version
(from the stack)
 - ii. Rewrite definition by inventing & pushing new name
 - c.) fill in Φ -function parameters of successor blocks
 - d.) recurse on \mathcal{B} 's children in the dominator tree
 - e.) <on exit from block \mathcal{B} > pop names generated in \mathcal{B} from stacks

SSA Construction Algorithm (Less high-level)

Adding all the details ...

```

for each global name  $i$ 
  counter[ $i$ ]  $\leftarrow 0$ 
  stack[ $i$ ]  $\leftarrow \emptyset$ 
call  $Rename(\mathcal{B}_0)$ 
  
```

```

 $NewName(v)$ 
   $i \leftarrow counter[v]$ 
  counter[ $v$ ]  $\leftarrow counter[v] + 1$ 
  push  $v_i$  onto stack[ $v$ ]
  return  $v_i$ 
  
```

$Rename(\mathcal{B})$

```

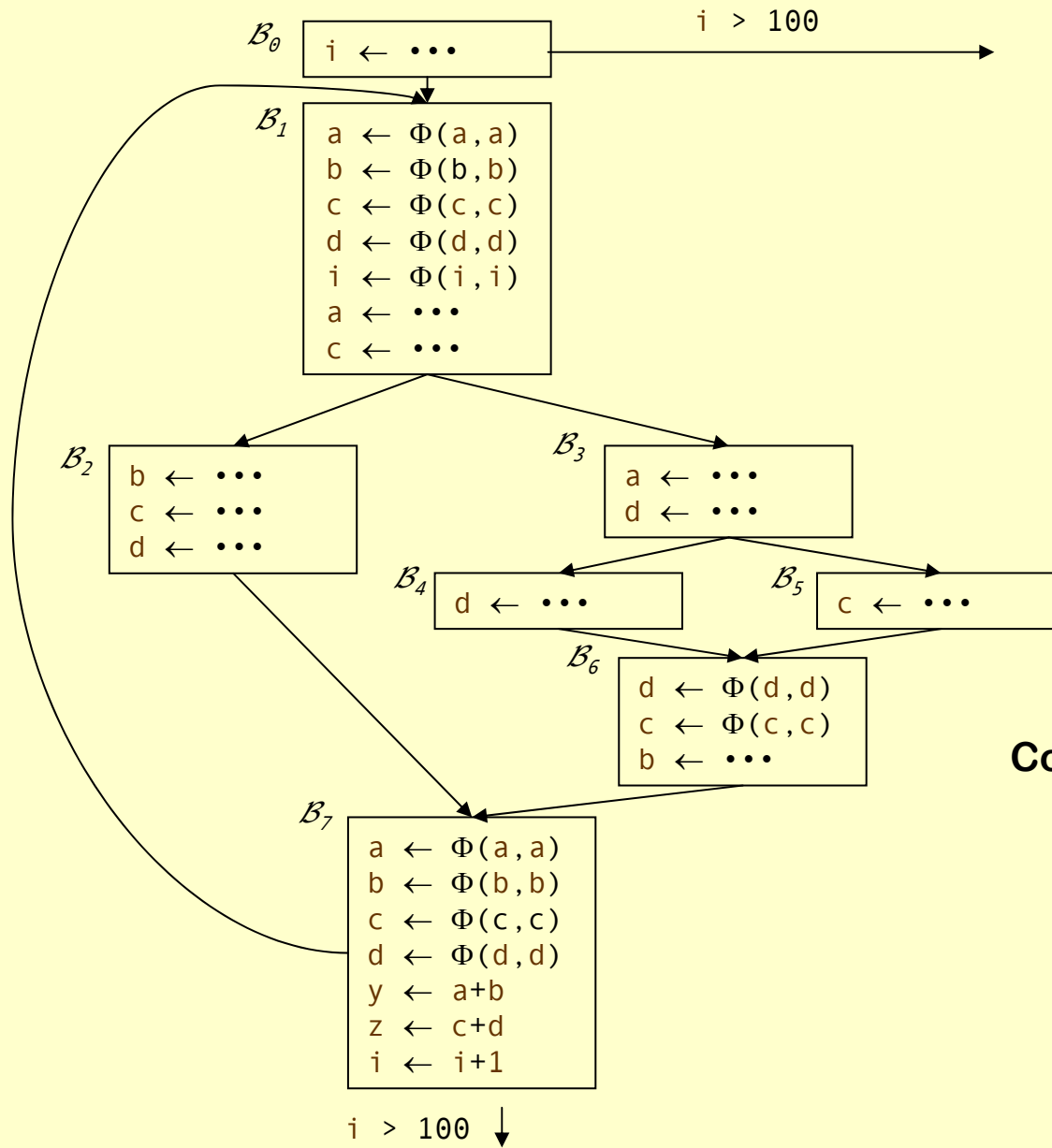
for each  $\Phi$ -function in  $\mathcal{B}$ ,  $x \leftarrow \Phi(\dots)$ 
  rename  $x$  as  $NewName(x)$ 

for each operation " $x \leftarrow y \text{ op } z$ " in  $\mathcal{B}$ 
  rewrite  $y$  as  $top(stack[y])$ 
  rewrite  $z$  as  $top(stack[z])$ 
  rewrite  $x$  as  $NewName(x)$ 

for each successor of  $\mathcal{B}$  in the CFG
  rewrite appropriate  $\Phi$  parameters

for each successor  $\mathcal{S}$  of  $\mathcal{B}$  in dom. tree
   $Rename(\mathcal{S})$ 

for each operation " $x \leftarrow y \text{ op } z$ " in  $\mathcal{B}$ 
   $pop(stack[x])$ 
  
```



Example

Before processing B_0

Assume a, b, c, & d defined before B_0

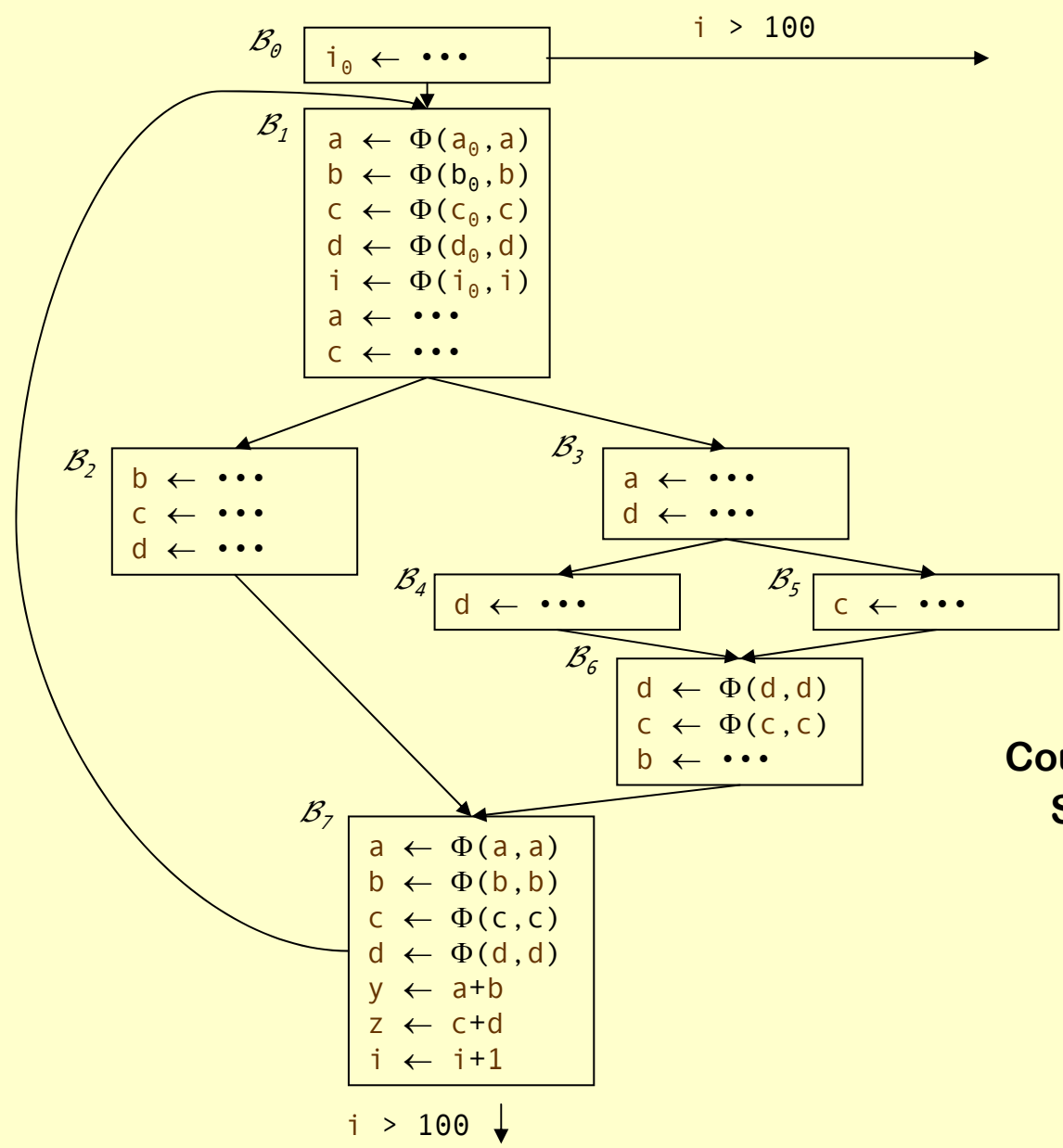
Counters
Stacks

a	b	c	d	i
1	1	1	1	0
a_0	b_0	c_0	d_0	

i has not been defined

Example

End of B_0

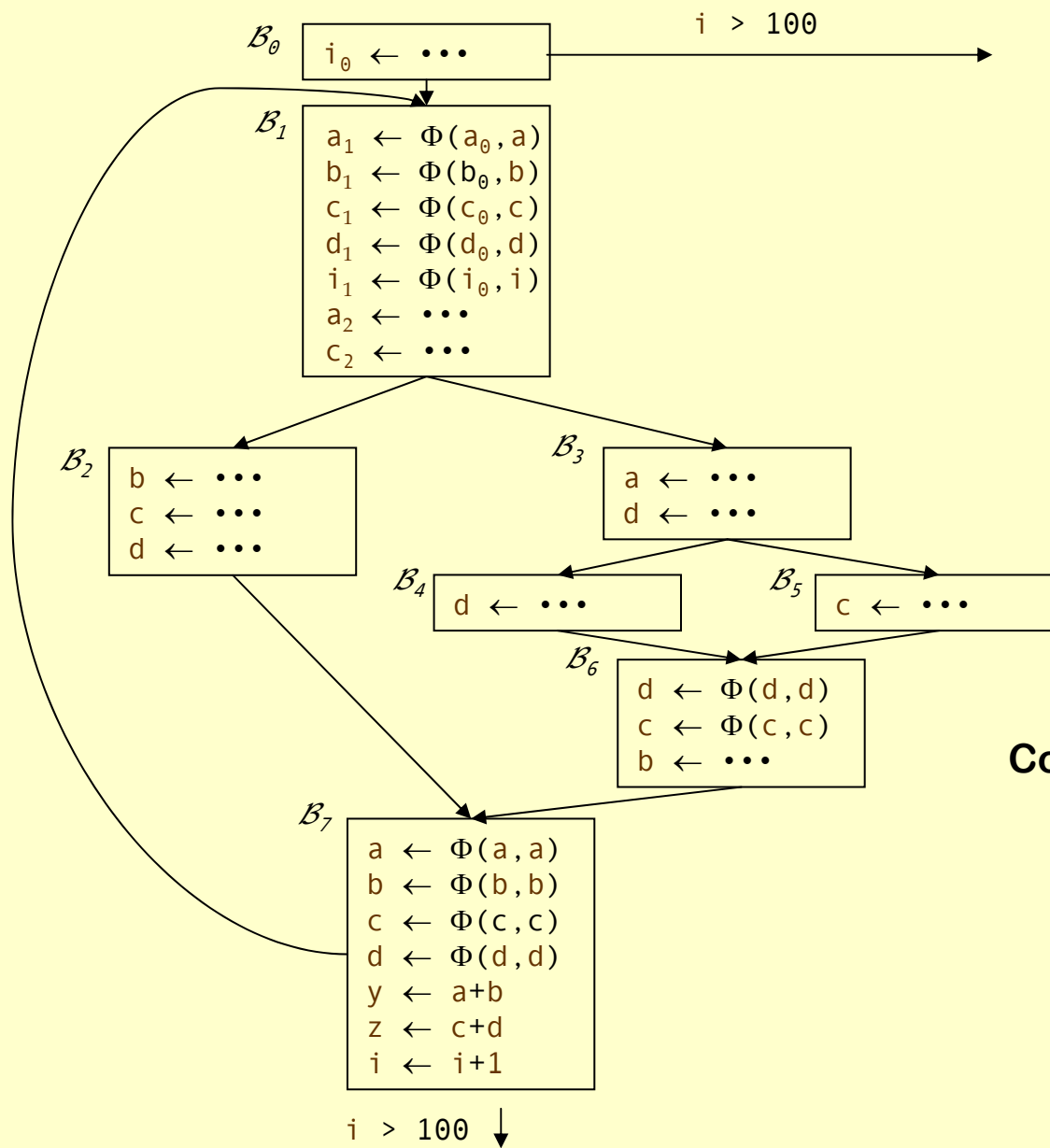


Counters
Stacks

<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>i</i>
1	1	1	1	1
a_0	b_0	c_0	d_0	i_0

Example

End of B_1

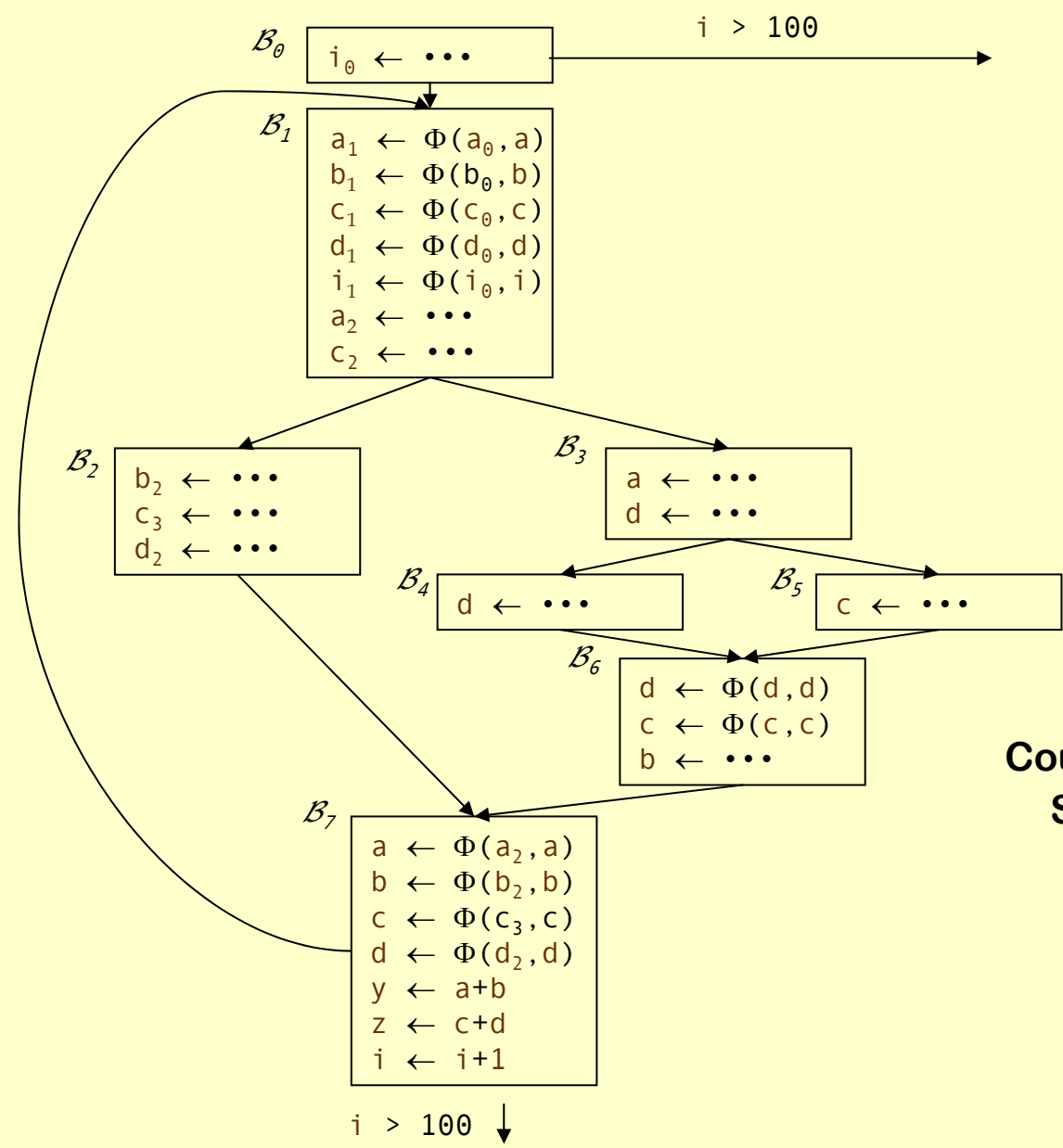


Counters
Stacks

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>i</i>
Counters	3	2	3	2	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2		

Example

End of B_2

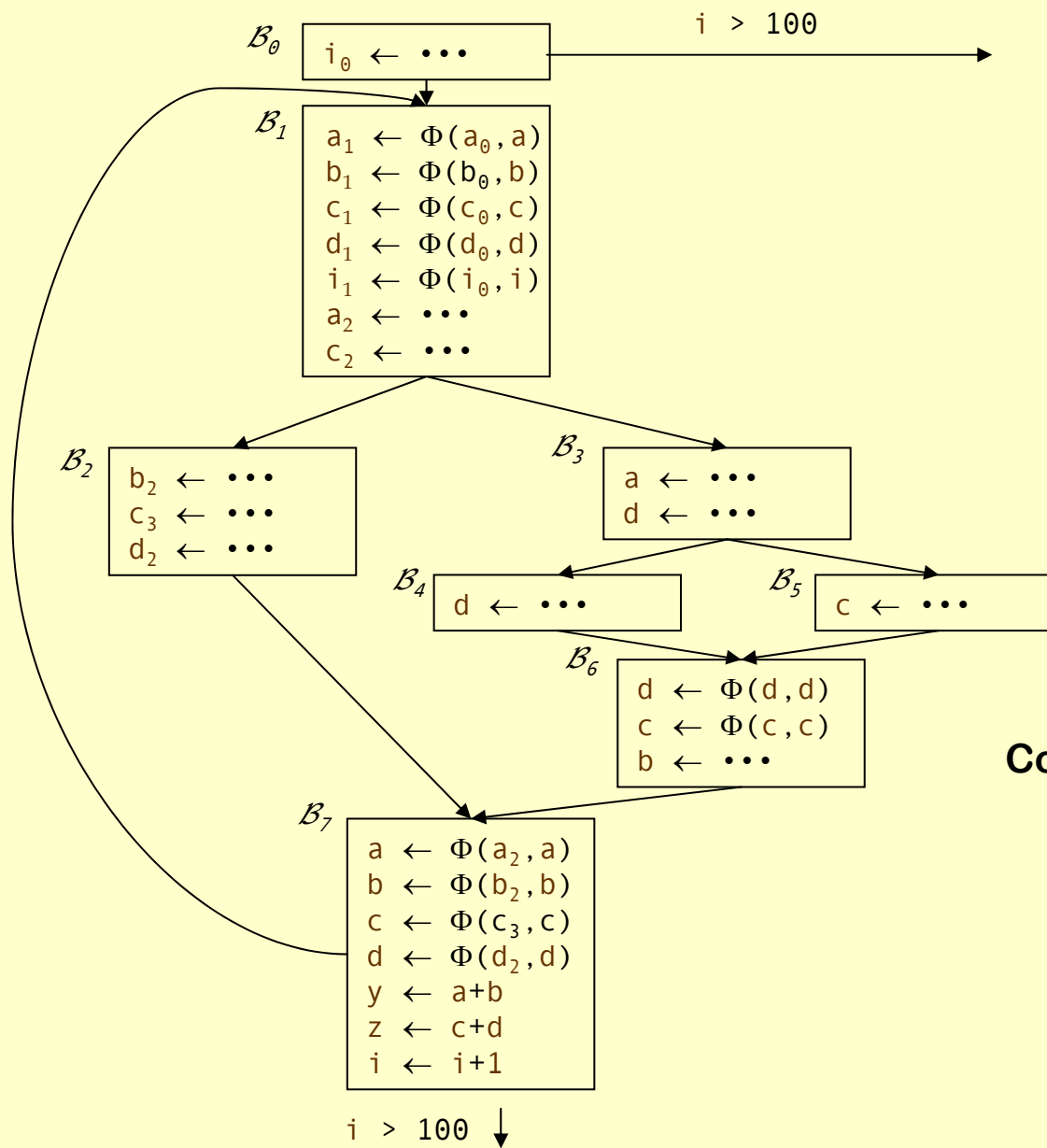


Counters
Stacks

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>i</i>
Counters	3	3	4	3	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2	b_2	c_2	d_2	
			c_3		

Example

Before starting B_3

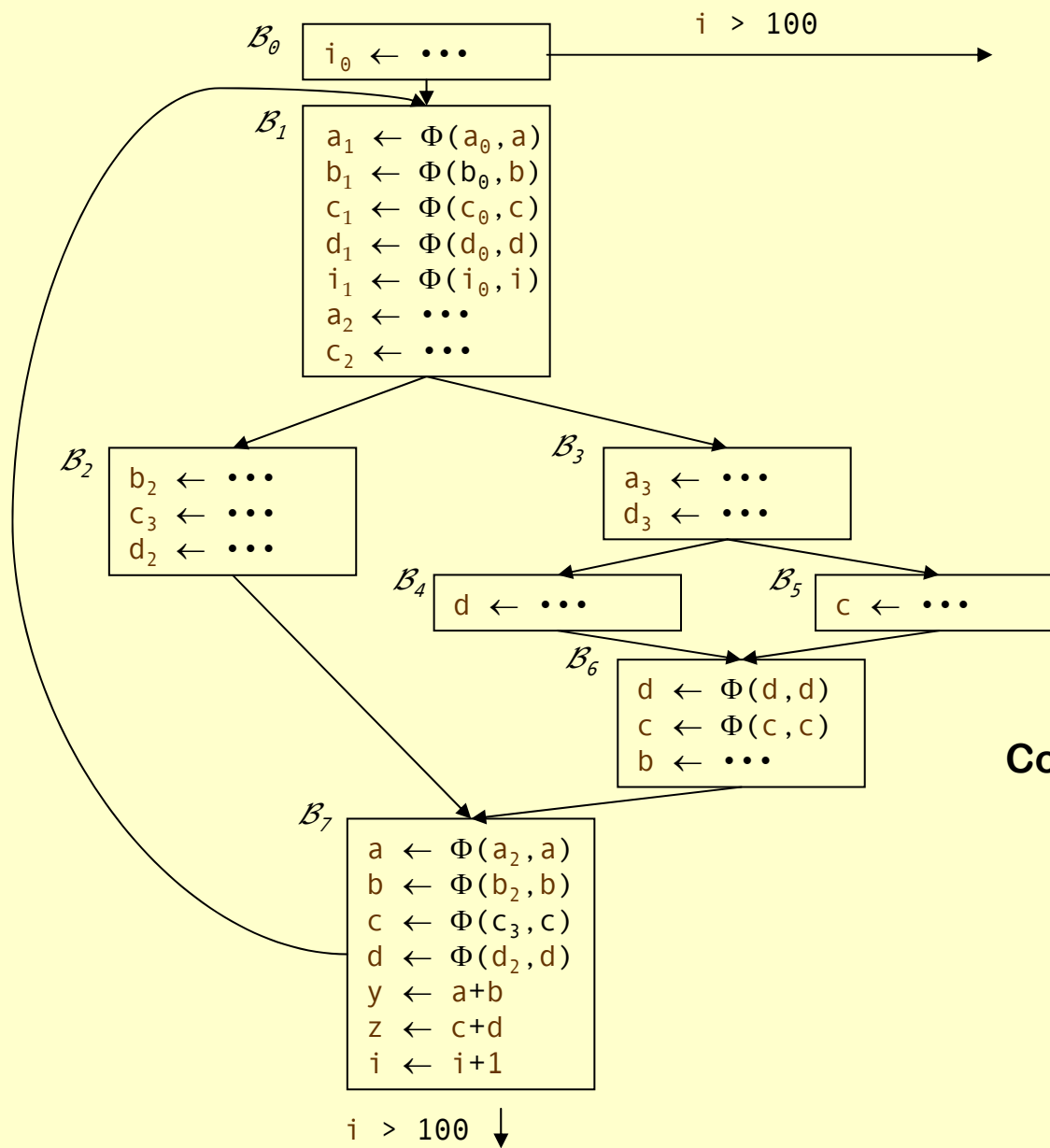


Counters
Stacks

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>i</i>
Counters	3	3	4	3	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2		

Example

End of B_3

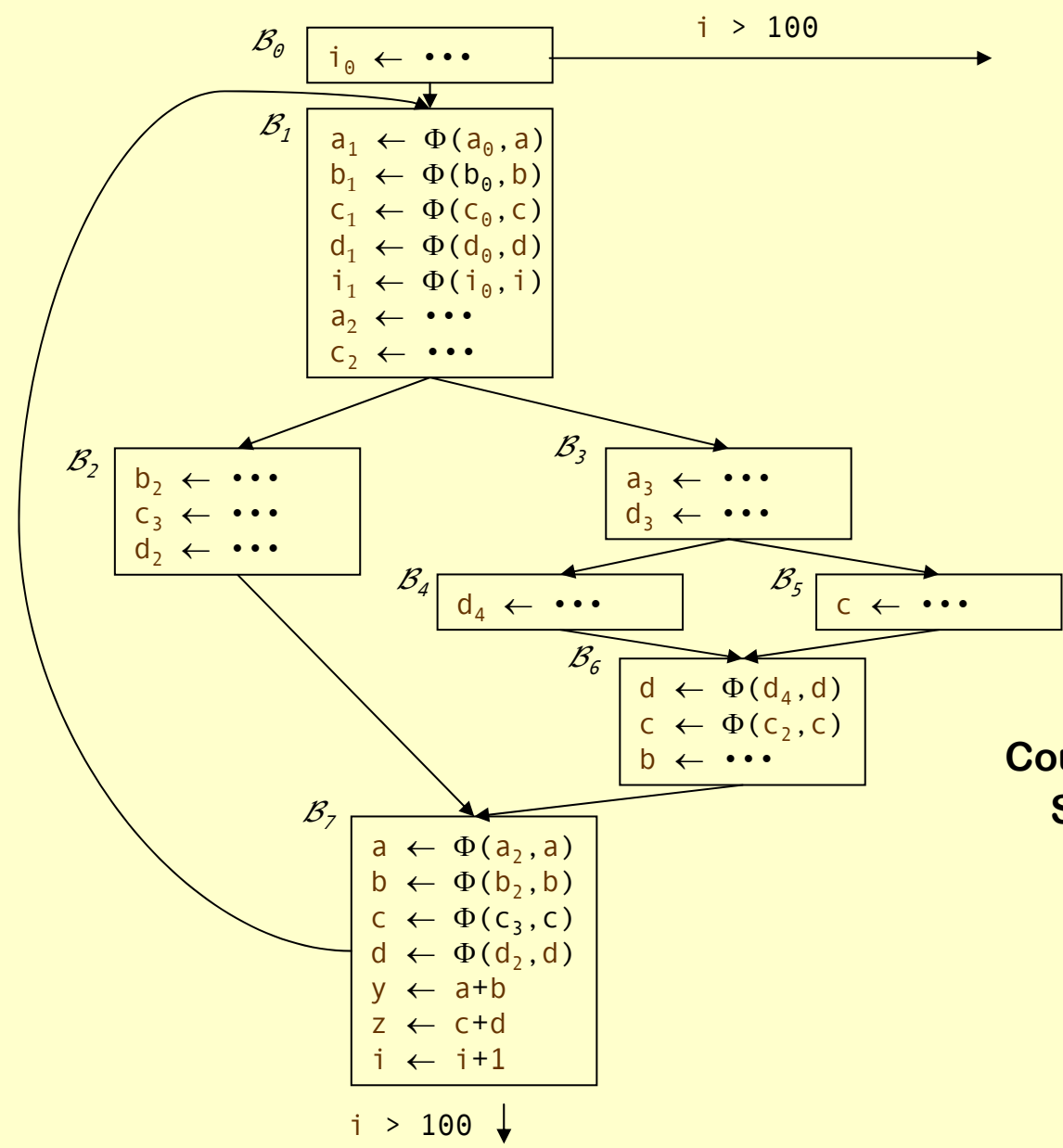


Counters
Stacks

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>i</i>
Counters	4	3	4	4	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2	d_3	
	a_3				

Example

End of B_4

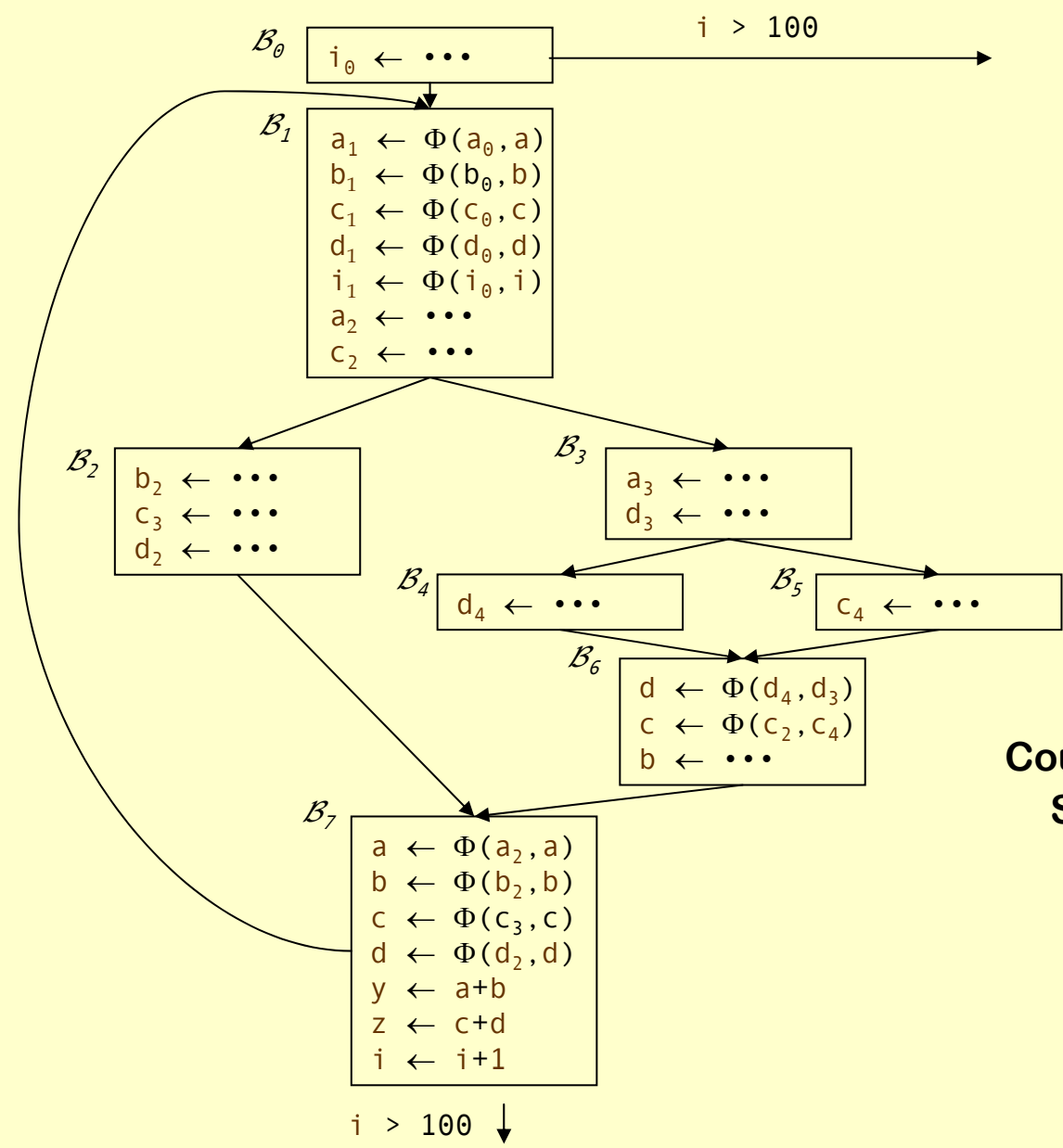


Counters
Stacks

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>i</i>
4	3	4	5	2	
a_0	b_0	c_0	d_0	i_0	
a_1	b_1	c_1	d_1	i_1	
a_2		c_2	d_3		
a_3			d_4		

Example

End of B_5

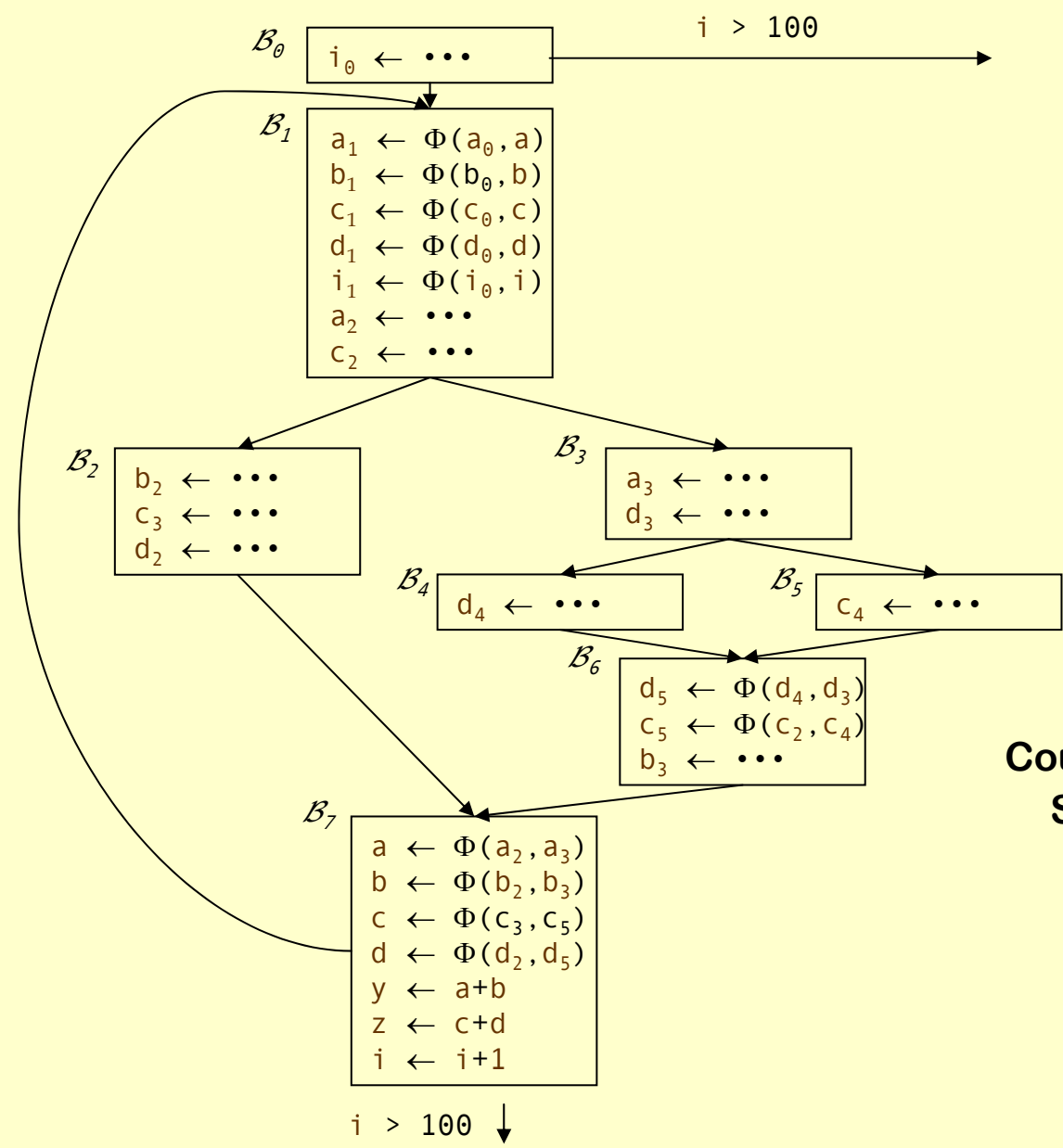


Counters
Stacks

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>i</i>
4					2
a_0	b_0	c_0	d_0	i_0	
a_1	b_1	c_1	d_1	i_1	
a_2		c_2	d_3		
a_3		c_4			

Example

End of B_6

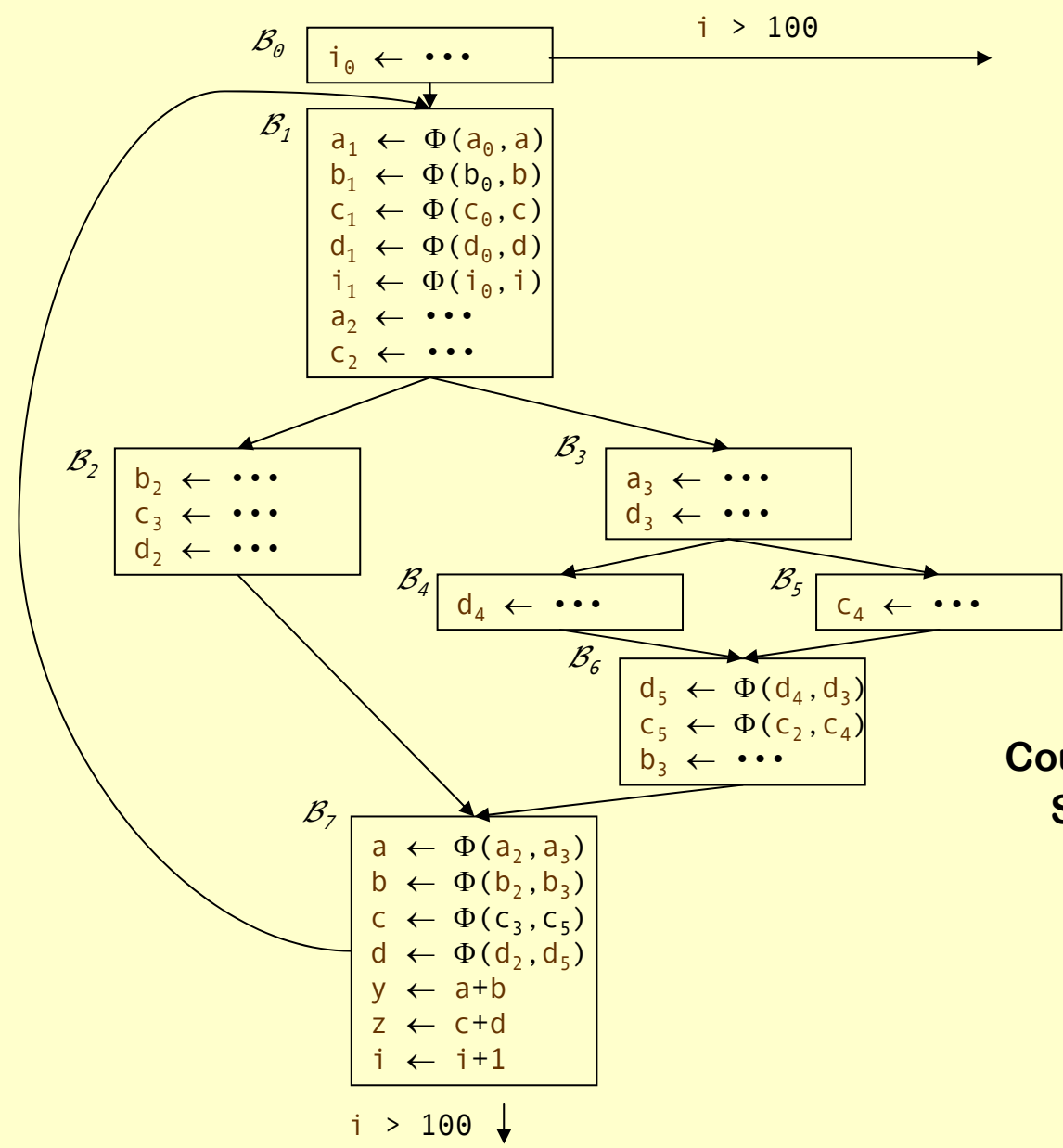


Counters
Stacks

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>i</i>
Counters	4	4	6	6	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2	b_3	c_2	d_3	
	a_3		c_5	d_5	

Example

Before B_7

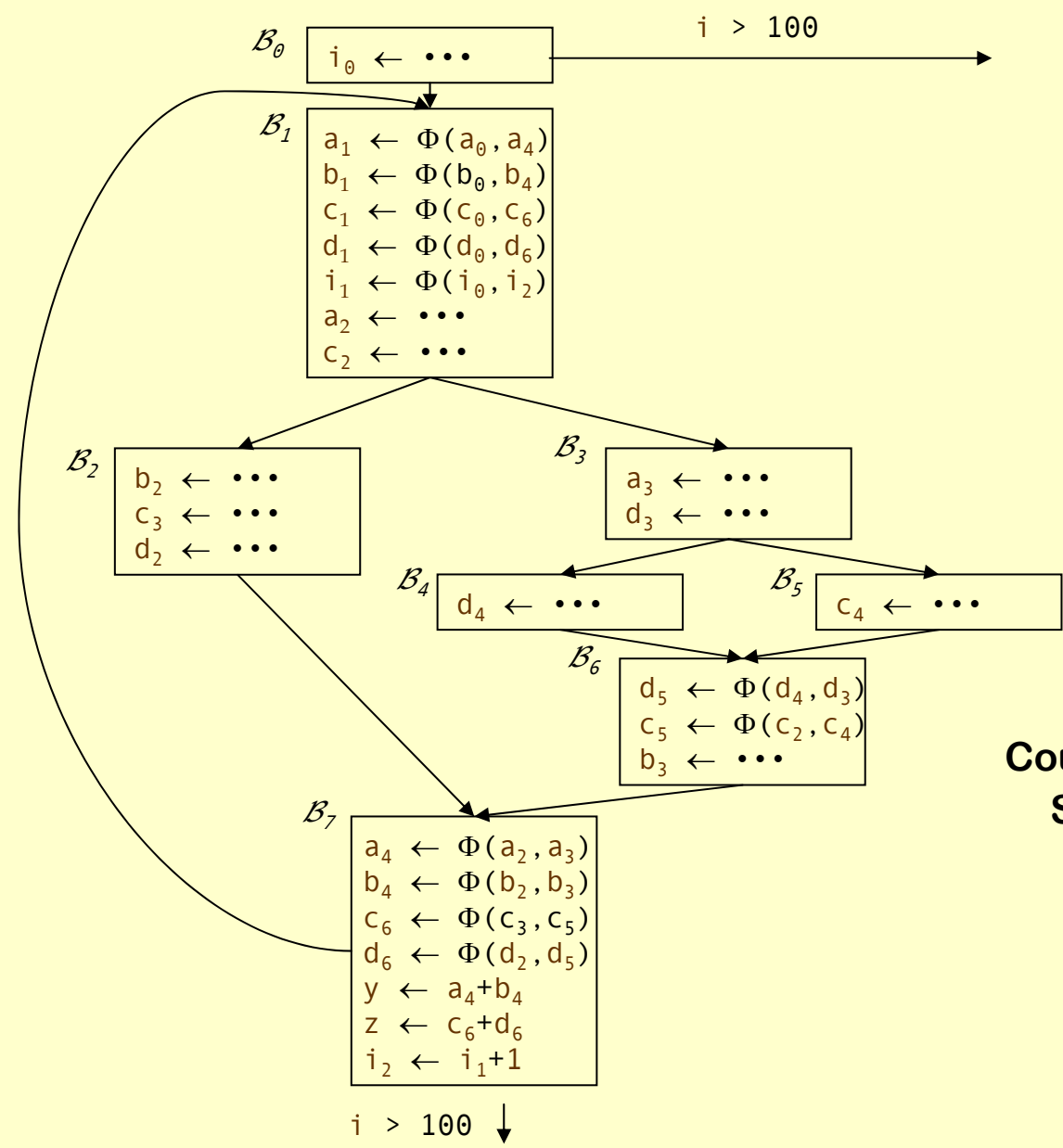


Counters
Stacks

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>i</i>
Counters	4	4	6	6	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2		

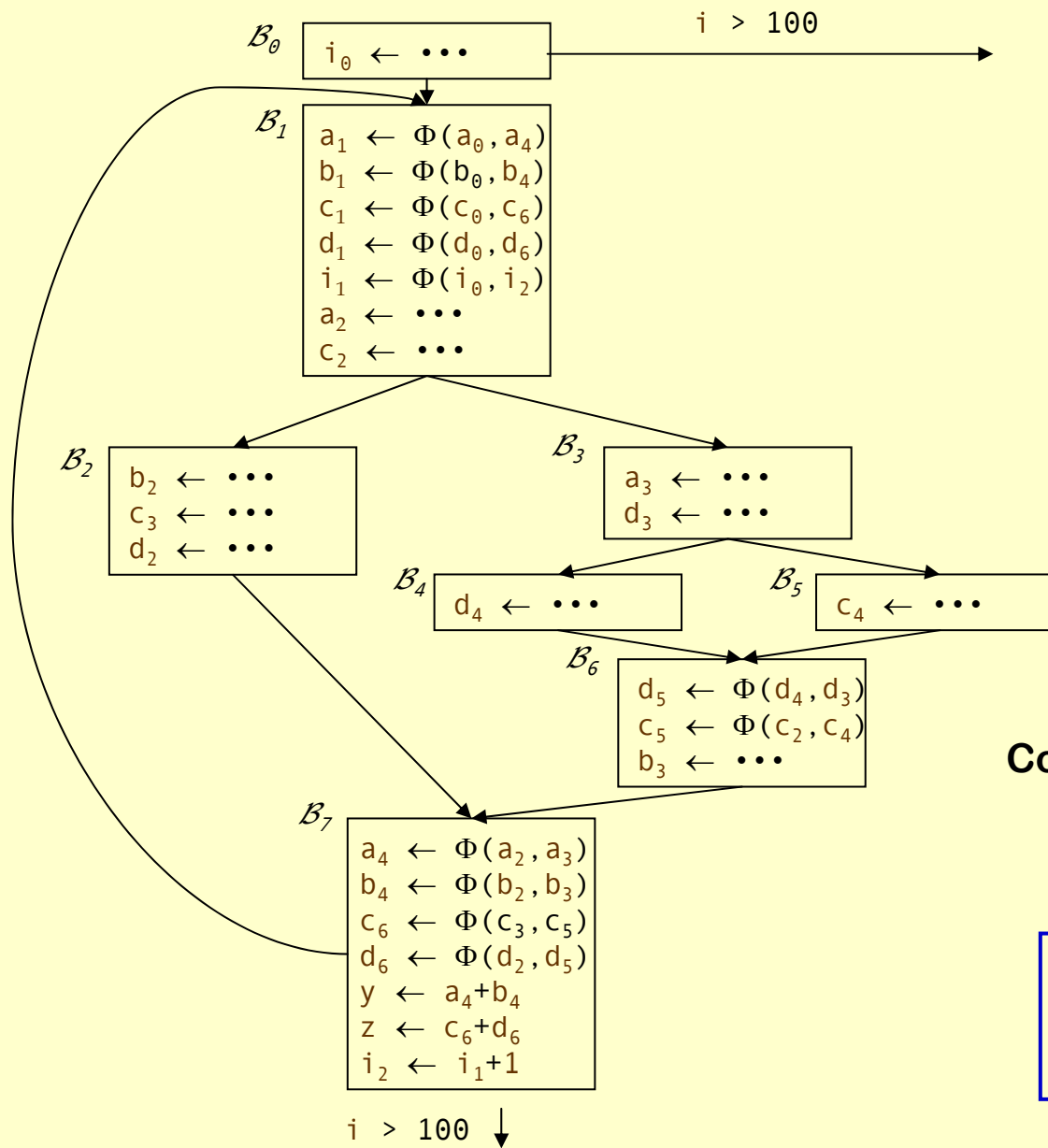
Example

End of B_7



Counters
Stacks

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>i</i>
5	5	7	7	3	
a_0	b_0	c_0	d_0	i_0	
a_1	b_1	c_1	d_1	i_1	
a_2	b_4	c_2	d_6	i_2	
a_4		c_6			



Example

After renaming

- Semi-pruned SSA form
- We're done ...

Counters
Stacks

Semi-pruned \Rightarrow only names live in 2 or more blocks are "global names".

SSA Construction Algorithm (Pruned SSA)

What's this “pruned SSA” stuff?

- ◆ Minimal SSA still contains extraneous Φ -functions.
- ◆ Inserts some Φ -functions where they are dead.
- ◆ Would like to avoid inserting them.

Two ideas

- ◆ *Semi-pruned SSA*: discard names used in only one block.
 - ◆ Significant reduction in total number of Φ -functions.
 - ◆ Needs only local liveness information. *(cheap to compute)*
- ◆ *Pruned SSA*: only insert Φ -functions where their value is live.
 - ◆ Inserts even fewer Φ -functions, but costs more to do.
 - ◆ Requires global live variable analysis. *(more expensive)*

In practice, both are simple modifications to step 1.

SSA Construction Algorithm

We can improve the stack management.

- ◆ Push at most one name per stack per block. (save push & pop)
- ◆ Thread names together by block.
- ◆ To pop names for block B , use B 's thread.

This is a good use for a scoped hash table.

- ◆ Significant reductions in pops and pushes.
- ◆ Makes a minor difference in SSA construction time.
- ◆ Scoped table is a clean, clear way to handle the problem.

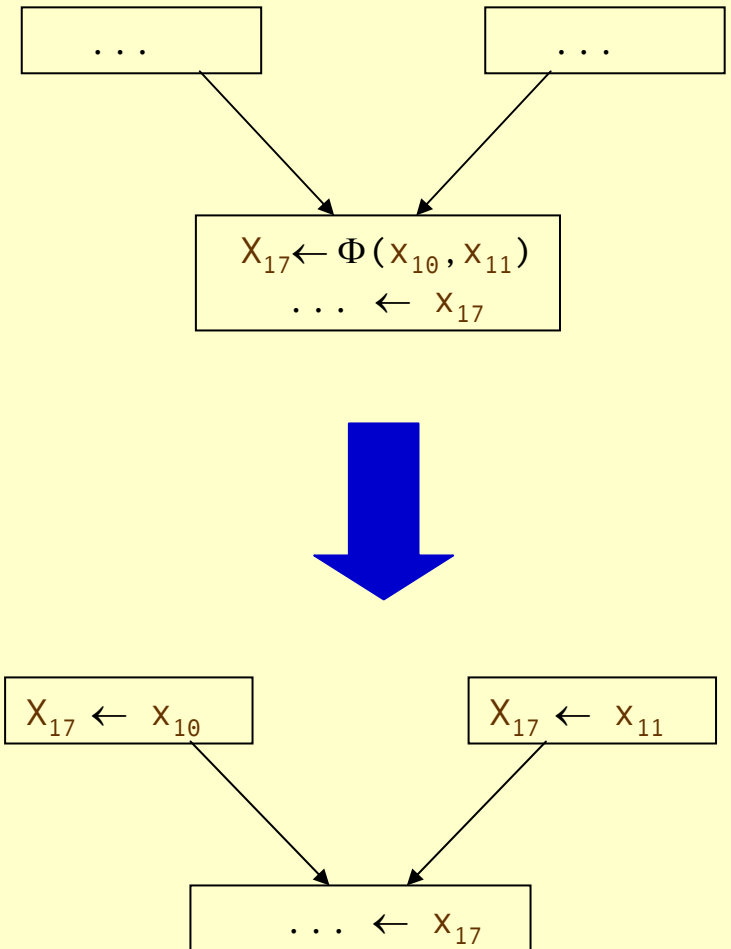
SSA Deconstruction

At some point, we need executable code.

- ◆ Few machines implement Φ operations.
- ◆ Need to fix up the flow of values.

Basic idea.

- ◆ Insert copies Φ -function pred's.
- ◆ Simple algorithm.
 - ◆ Works in most cases.
- ◆ Adds lots of copies.
 - ◆ Most of them coalesce away.



Dead Code Elimination & Constant Propagation on SSA form

This lecture is primarily based on Konstantinos Sagonas set of slides
(Advanced Compiler Techniques, (2AD518)
at Uppsala University, January-February 2004).

Used with kind permission.
(In turn based on Keith Cooper's slides)

Dead Code Elimination Using SSA

Dead code elimination

- ◆ Conceptually similar to mark-sweep garbage collection:
 - ◆ Mark *useful* operations.
 - ◆ Everything not marked is useless.
- ◆ Need an efficient way to find and to mark useful operations.
 - ◆ Start with critical operations.
 - ◆ Work back up SSA edges to find their antecedents.
- ◆ Operations defined as critical:
 - ◆ I/O statements,
 - ◆ linkage code (*entry & exit blocks*),
 - ◆ return values,
 - ◆ calls to other procedures.

Algorithm will use post-dominators & reverse dominance frontiers.

Dead Code Elimination Using SSA

Mark

```

for each op i
  clear i's mark
  if i is critical then
    mark i
    add i to WorkList

while (Worklist  $\neq \emptyset$ )
  remove i from WorkList
  (i has form "x ← y op z")
  if def(y) is not marked then
    mark def(y)
    add def(y) to WorkList
  if def(z) is not marked then
    mark def(z)
    add def(z) to WorkList

for each b  $\in$  RDF(block(i))
  mark the block-ending
  branch in b
  add it to WorkList

```

Sweep

```

for each op i
  if i is not marked then
    if i is a branch then
      rewrite with a jump to
      i's nearest useful
      post-dominator
    if i is not a jump then
      delete i

```

Notes:

- Eliminates some branches.
- Reconnects dead branches to the remaining live code.
- Find useful post-dominator by walking post-dominator tree.
 - > Entry & exit nodes are useful

Dead Code Elimination Using SSA

Handling Branches

- ◆ When is a branch useful?
 - ◆ When another useful operation depends on its existence

In the CFG, j is control dependent on i if

1. \exists a non-null path ρ from i to j such that j post-dominates every node on ρ after i
2. j does not strictly post-dominate i

- ◆ j control dependent on $i \Rightarrow$ one path from i leads to j , one doesn't
- ◆ This is the reverse dominance frontier of j ($\text{RDF}(j)$)

Algorithm uses $\text{RDF}(n)$ to mark branches as live

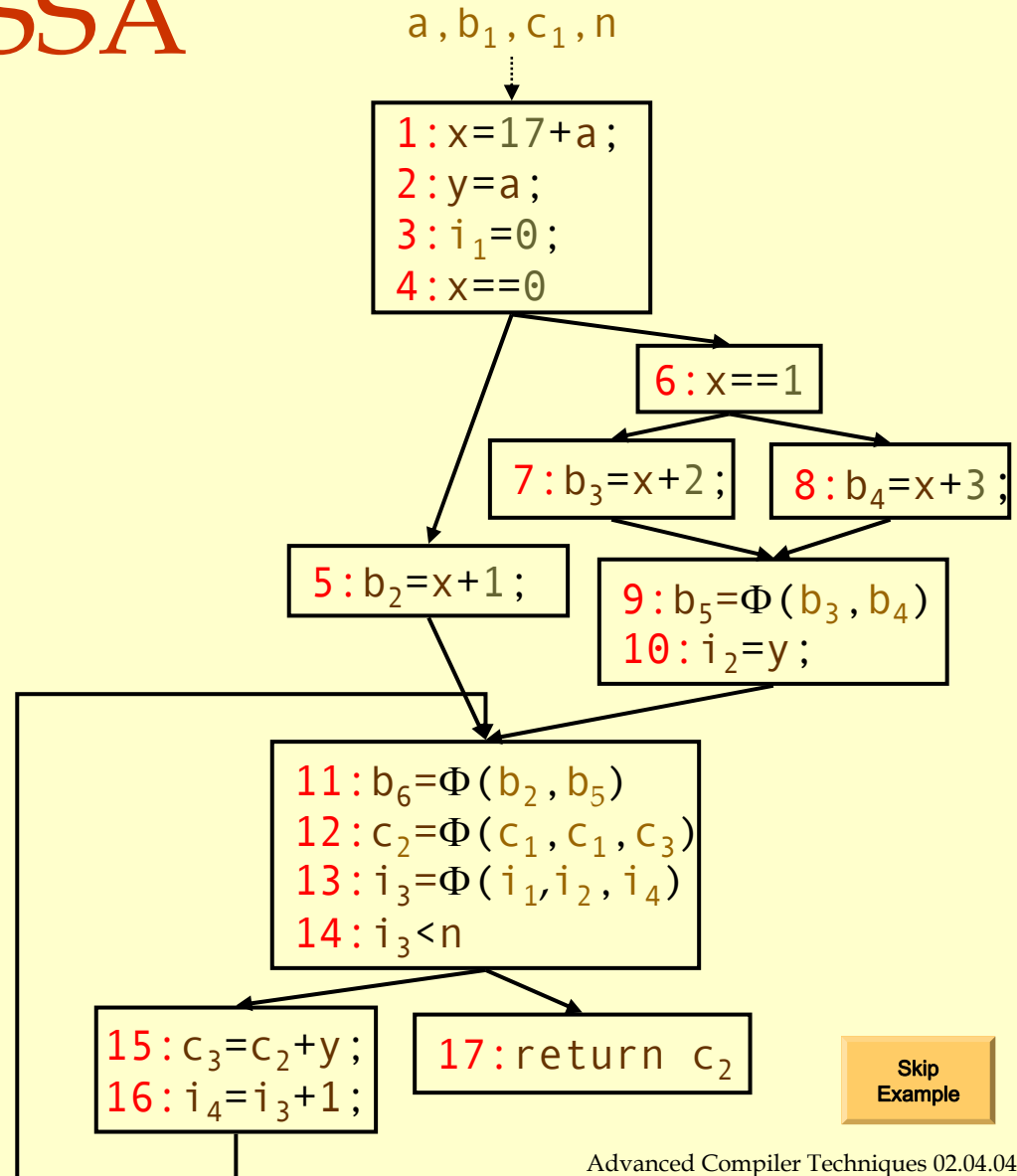
Dead Code Elimination Using SSA

Mark

for each op i
clear i 's mark
if i is critical then
mark i
add i to **WorkList**

while (**Worklist** $\neq \emptyset$)
remove i from **WorkList**
(i has form " $x \leftarrow y \text{ op } z$ ")
if $\text{def}(y)$ is not marked then
mark $\text{def}(y)$
add $\text{def}(y)$ to **WorkList**
if $\text{def}(z)$ is not marked then
mark $\text{def}(z)$
add $\text{def}(z)$ to **WorkList**

for each $b \in \text{RDF}(\text{block}(i))$
mark the block-ending
branch in b
add it to **WorkList**



Skip Example

Dead Code Elimination Using SSA

Mark

```

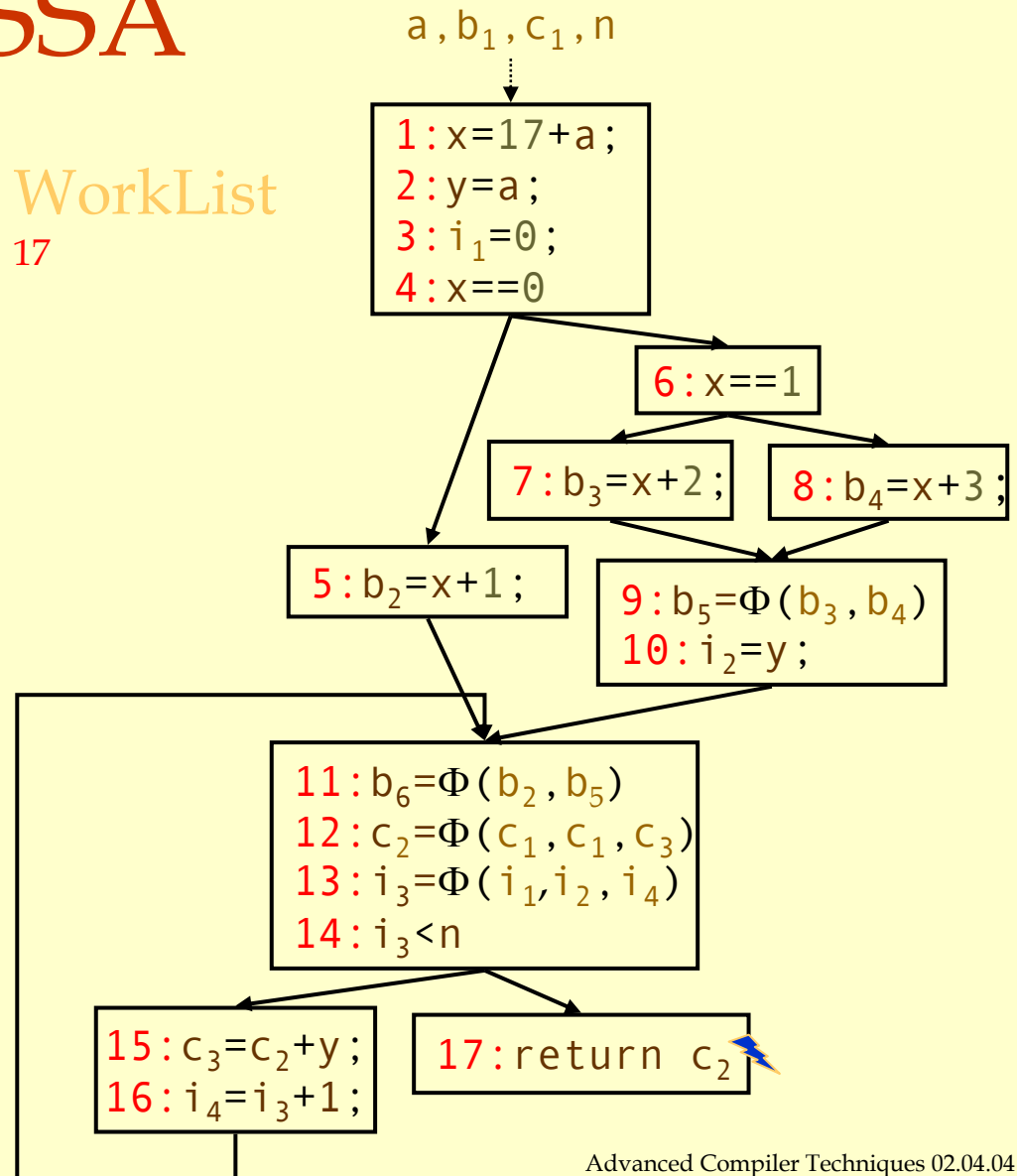
for each op i
  clear i's mark
  if i is critical then
    mark i
    add i to WorkList
  
```

```

while (Worklist  $\neq \emptyset$ )
  remove i from WorkList
  (i has form " $x \leftarrow y \text{ op } z$ ")
  if def(y) is not marked then
    mark def(y)
    add def(y) to WorkList
  if def(z) is not marked then
    mark def(z)
    add def(z) to WorkList

for each b  $\in$  RDF(block(i))
  mark the block-ending
  branch in b
  add it to WorkList
  
```

WorkList
17



Dead Code Elimination Using SSA

Mark

for each op i
 clear i 's mark
 if i is critical then
 mark i
 add i to WorkList

while (Worklist $\neq \emptyset$)

remove i from WorkList
 (i has form "op z ")

if $\text{def}(y)$ is not marked then
 mark $\text{def}(y)$
 add $\text{def}(y)$ to WorkList

if $\text{def}(z)$ is not marked then
 mark $\text{def}(z)$
 add $\text{def}(z)$ to WorkList

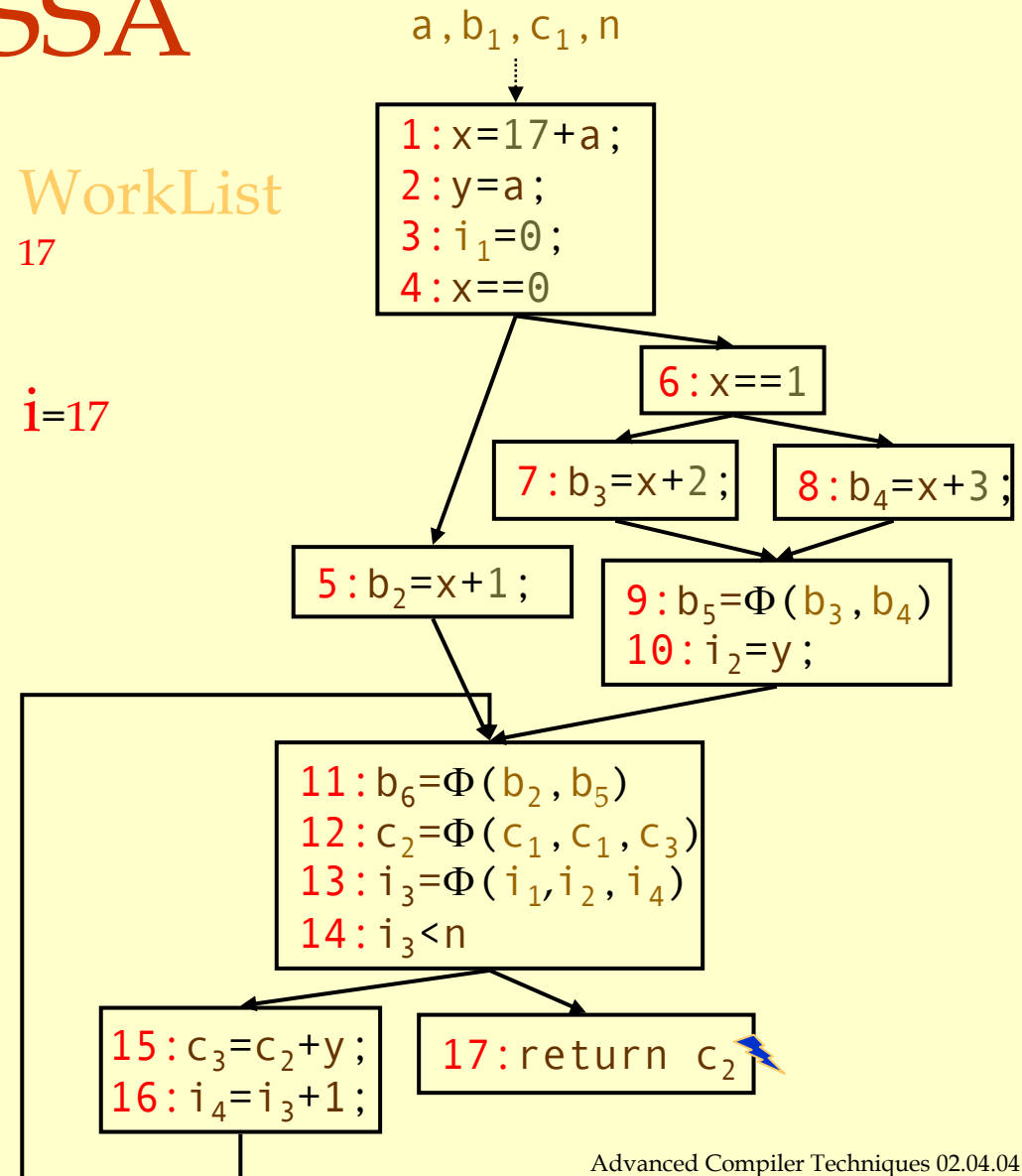
for each $b \in \text{RDF}(\text{block}(i))$
 mark the block-ending
 branch in b
 add it to WorkList

SSA

WorkList

17

$i=17$



Dead Code Elimination Using SSA

Mark

for each op i
 clear i 's mark
 if i is critical then
 mark i
 add i to **WorkList**

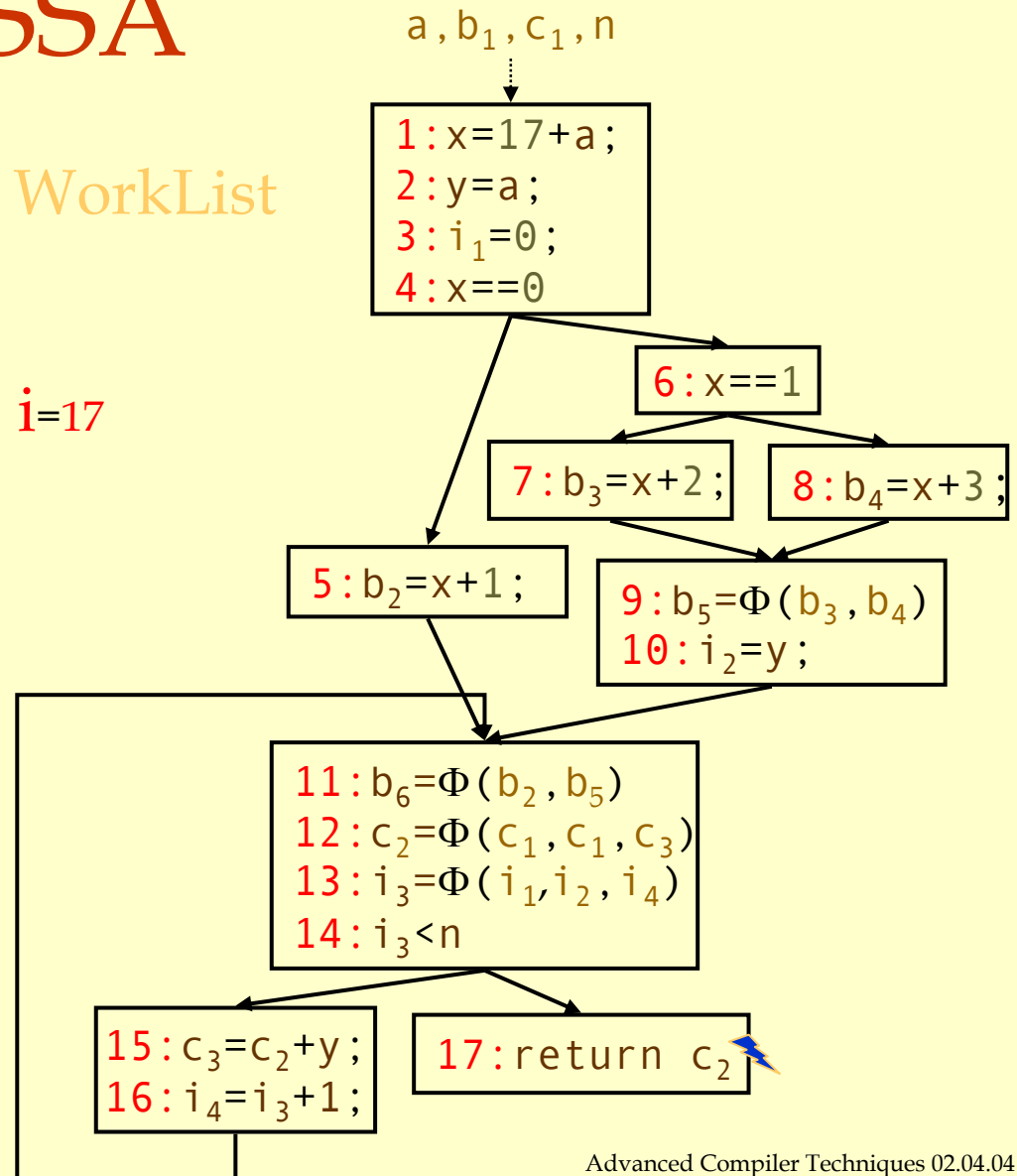
while (**Worklist** $\neq \emptyset$)
 remove i from **WorkList**
 (i has form " $op\ z$ ")
 if $def(y)$ is not marked then
 mark $def(y)$
 add $def(y)$ to **WorkList**

if $def(z)$ is not marked then
 mark $def(z)$
 add $def(z)$ to **WorkList**

for each $b \in RDF(block(i))$
 mark the block-ending
 branch in b
 add it to **WorkList**

WorkList

$i=17$



Dead Code Elimination Using SSA

Mark

for each op i
 clear i 's mark
 if i is critical then
 mark i
 add i to **WorkList**

while (**Worklist** $\neq \emptyset$)
 remove i from **WorkList**
 (i has form " $op\ z$ ")
 if $def(y)$ is not marked then
 mark $def(y)$
 add $def(y)$ to **WorkList**

if $def(z)$ is not marked then
 mark $def(z)$
 add $def(z)$ to **WorkList**

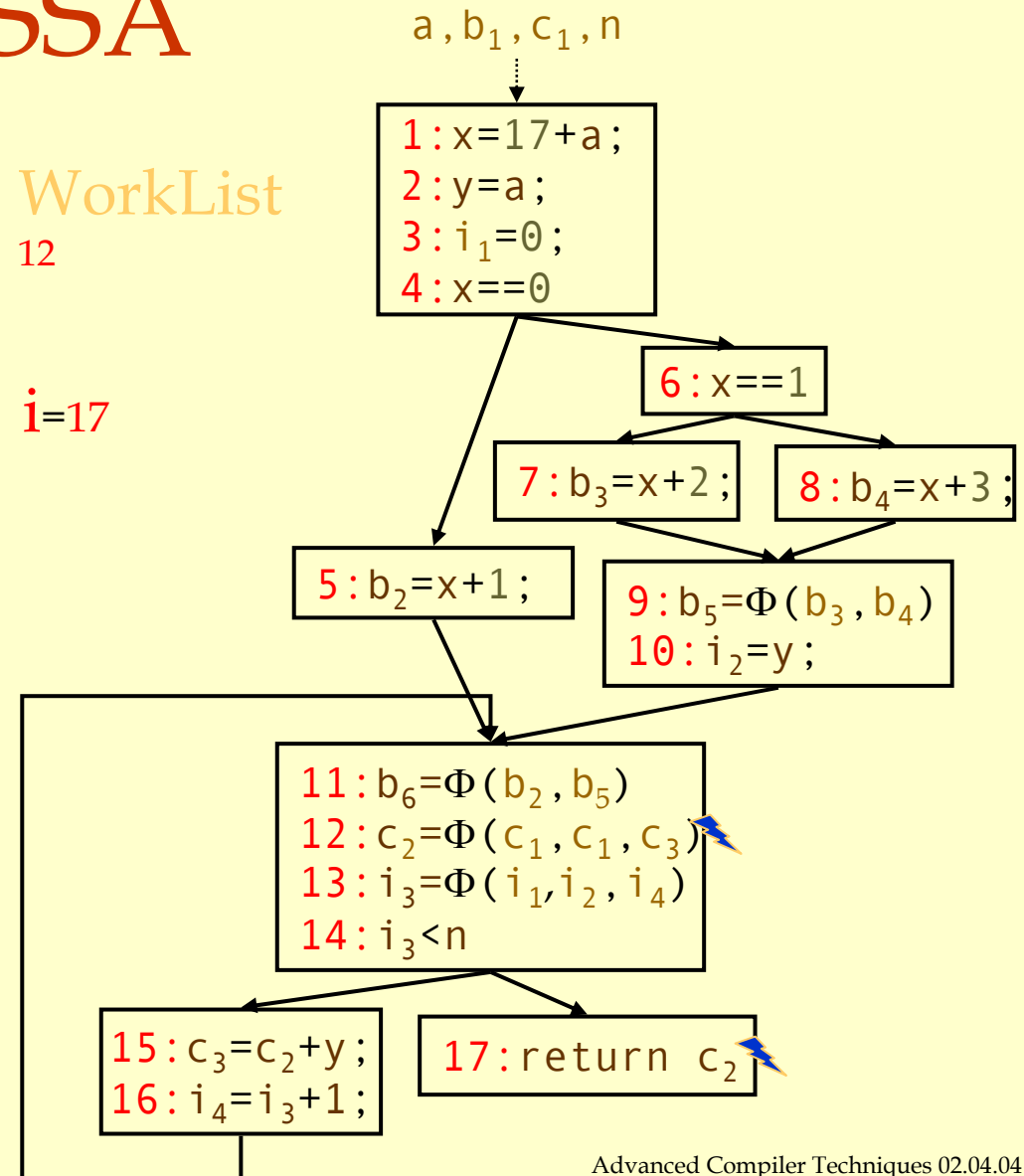
for each $b \in \mathbf{RDF}(block(i))$
 mark the block-ending
 branch in b
 add it to **WorkList**

SSA

WorkList

12

$i=17$



Dead Code Elimination Using SSA

Mark

for each op i
 clear i 's mark
 if i is critical then
 mark i
 add i to WorkList

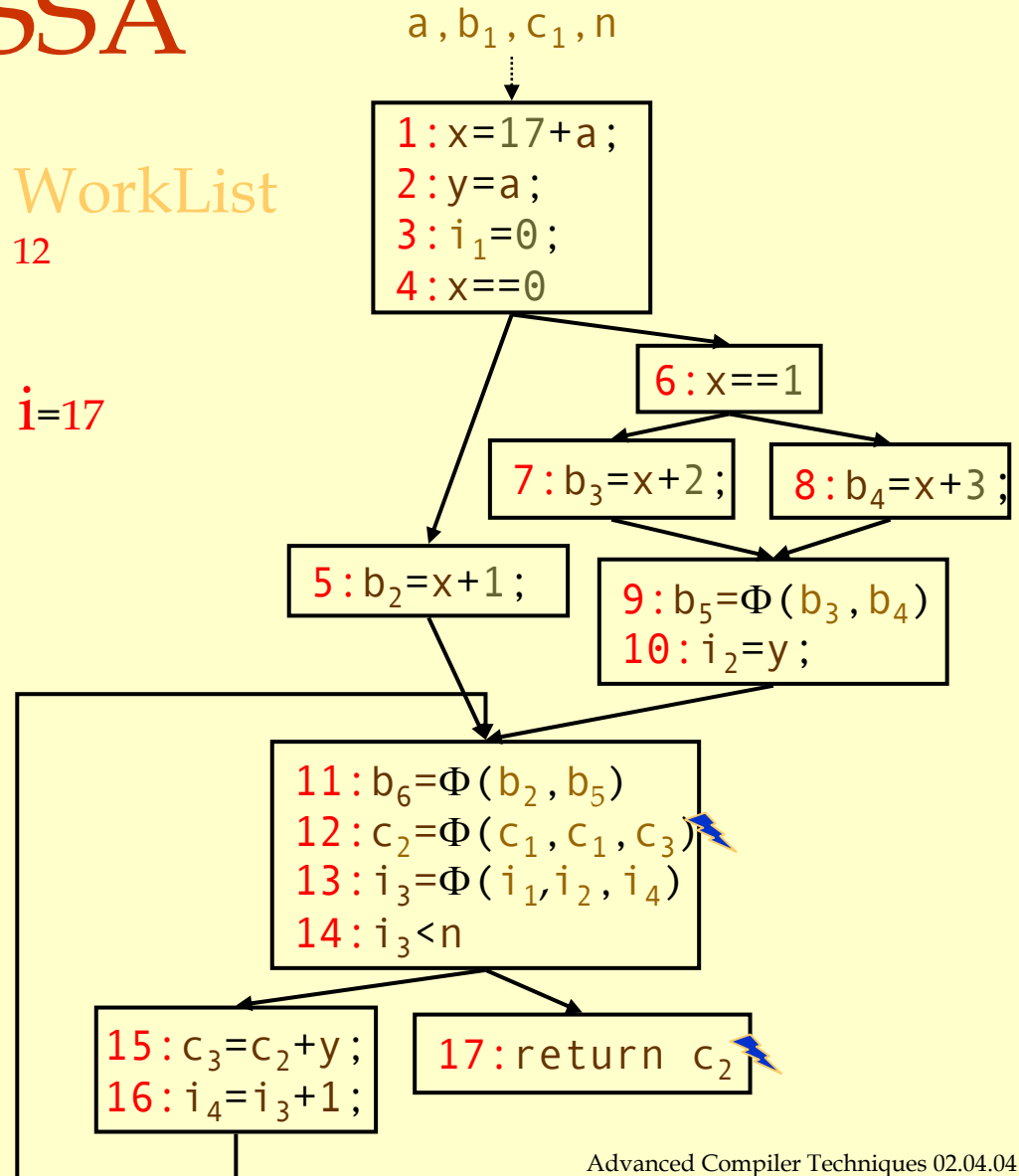
while (Worklist $\neq \emptyset$)
 remove i from WorkList
 (i has form "op z ")
 if $\text{def}(y)$ is not marked then
 mark $\text{def}(y)$
 add $\text{def}(y)$ to WorkList
 if $\text{def}(z)$ is not marked then
 mark $\text{def}(z)$
 add $\text{def}(z)$ to WorkList

for each $b \in \text{RDF}(\text{block}(i))$
 mark the block-ending
 branch in b
 add it to WorkList

SSA

WorkList
 12

$i=17$



Dead Code Elimination Using SSA

Mark

for each op i
 clear i 's mark
 if i is critical then
 mark i
 add i to **WorkList**

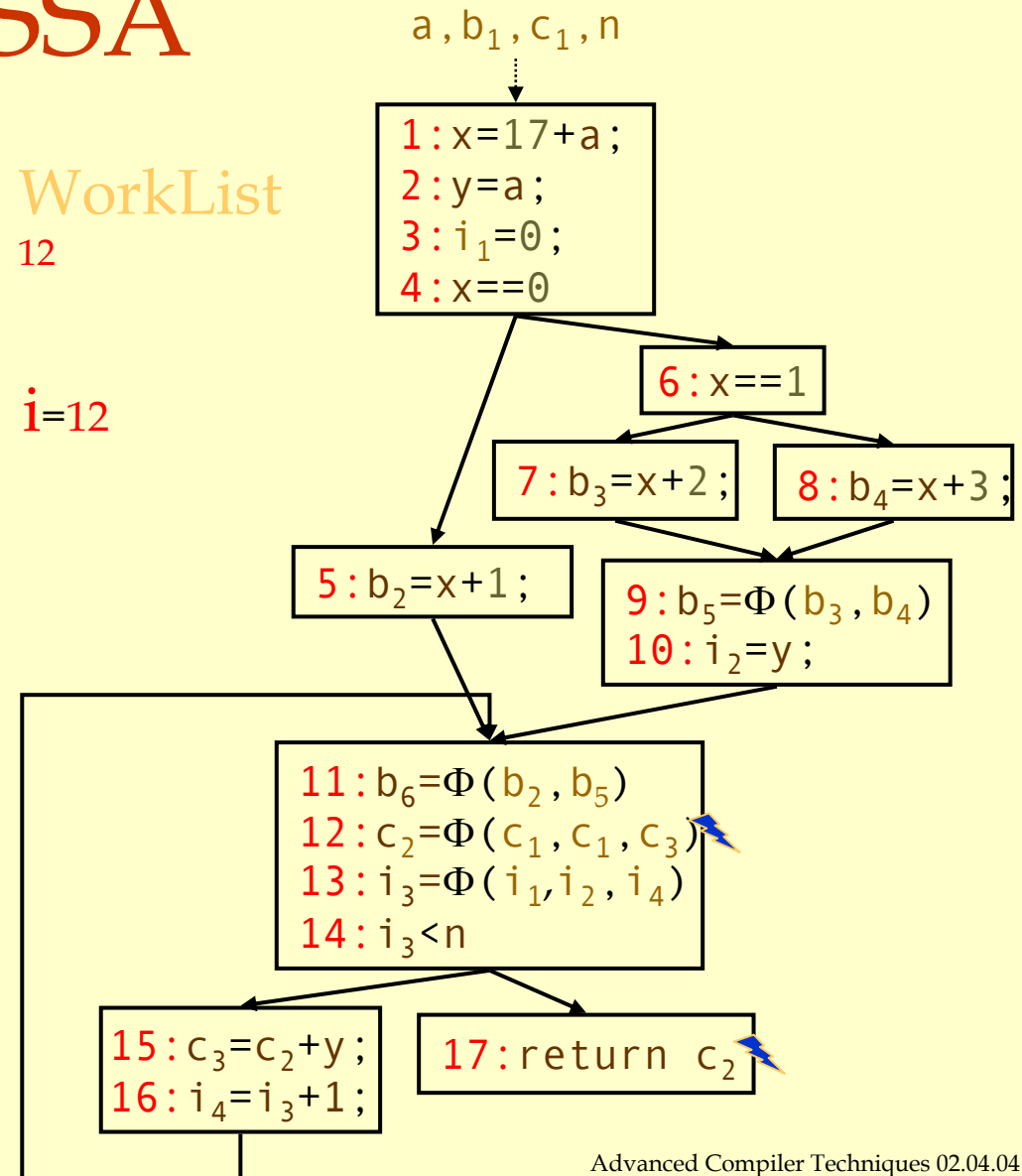
while (**Worklist** $\neq \emptyset$)
 remove i from **WorkList**
 (i has form " $x \leftarrow y \text{ op } z$ ")
 if $\text{def}(y)$ is not marked then
 mark $\text{def}(y)$
 add $\text{def}(y)$ to **WorkList**
 if $\text{def}(z)$ is not marked then
 mark $\text{def}(z)$
 add $\text{def}(z)$ to **WorkList**
 for each $b \in \text{RDF}(\text{block}(i))$
 mark the block-ending
 branch in b
 add it to **WorkList**

SSA

WorkList

12

$i=12$



Dead Code Elimination Using SSA

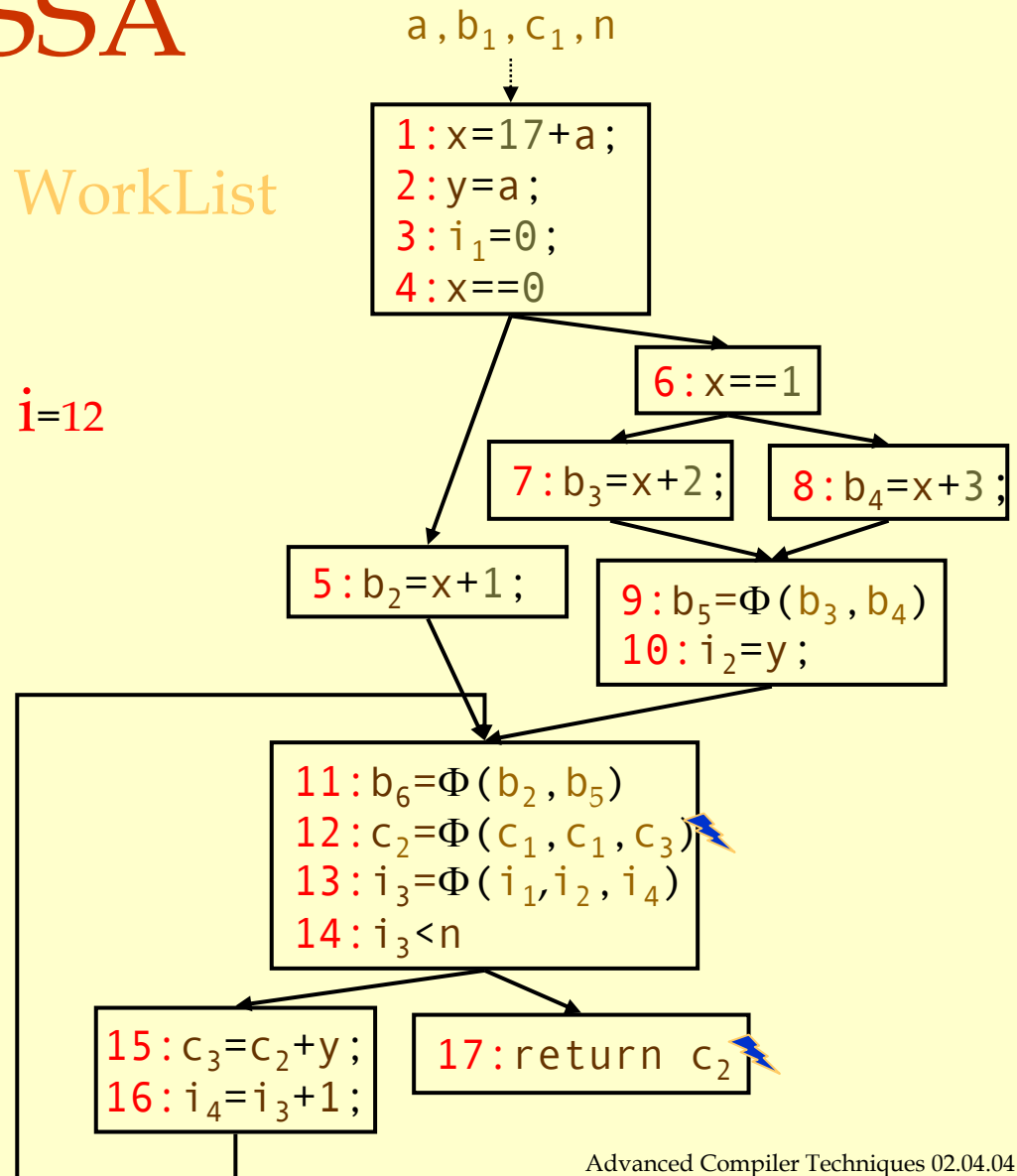
Mark

for each op i
 clear i 's mark
 if i is critical then
 mark i
 add i to **WorkList**

while (**Worklist** $\neq \emptyset$)
 remove i from **WorkList**
 (i has form " $x \leftarrow y \text{ op } z$ ")
 if $\text{def}(y)$ is not marked then
 mark $\text{def}(y)$
 add $\text{def}(y)$ to **WorkList**
 if $\text{def}(z)$ is not marked then
 mark $\text{def}(z)$
 add $\text{def}(z)$ to **WorkList**
 for each $b \in \text{RDF}(\text{block}(i))$
 mark the block-ending
 branch in b
 add it to **WorkList**

WorkList

$i=12$



Dead Code Elimination Using SSA

Mark

for each op i
 clear i 's mark
 if i is critical then
 mark i
 add i to **WorkList**

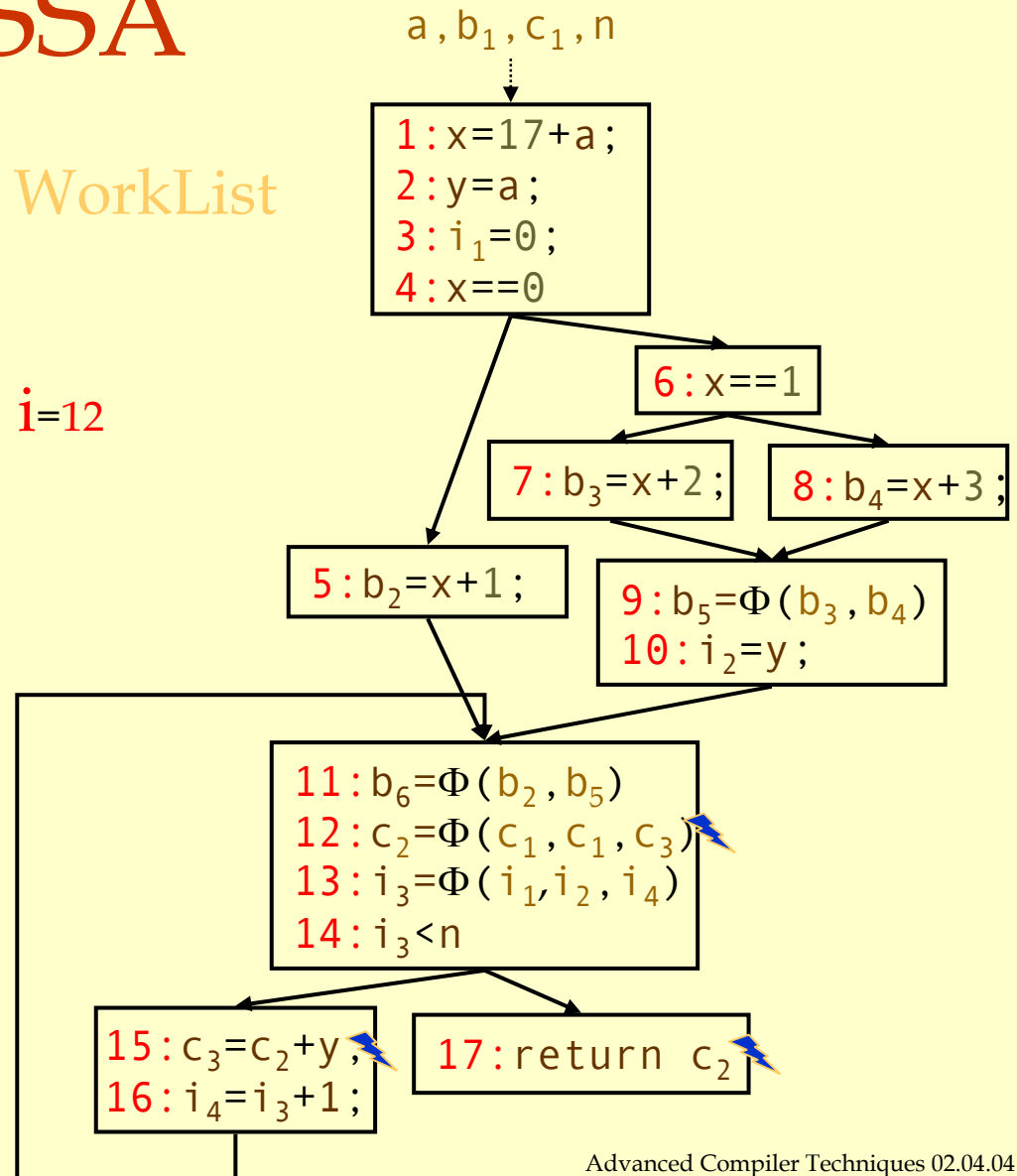
while (**Worklist** $\neq \emptyset$)
 remove i from **WorkList**
 (i has form " $x \leftarrow y \text{ op } z$ ")
 if $\text{def}(y)$ is not marked then
 mark $\text{def}(y)$
 add $\text{def}(y)$ to **WorkList**

if $\text{def}(z)$ is not marked then
 mark $\text{def}(z)$
 add $\text{def}(z)$ to **WorkList**

for each $b \in \text{RDF}(\text{block}(i))$
 mark the block-ending
 branch in b
 add it to **WorkList**

WorkList

$i=12$



Dead Code Elimination Using SSA

Mark

for each op i
 clear i 's mark
 if i is critical then
 mark i
 add i to WorkList

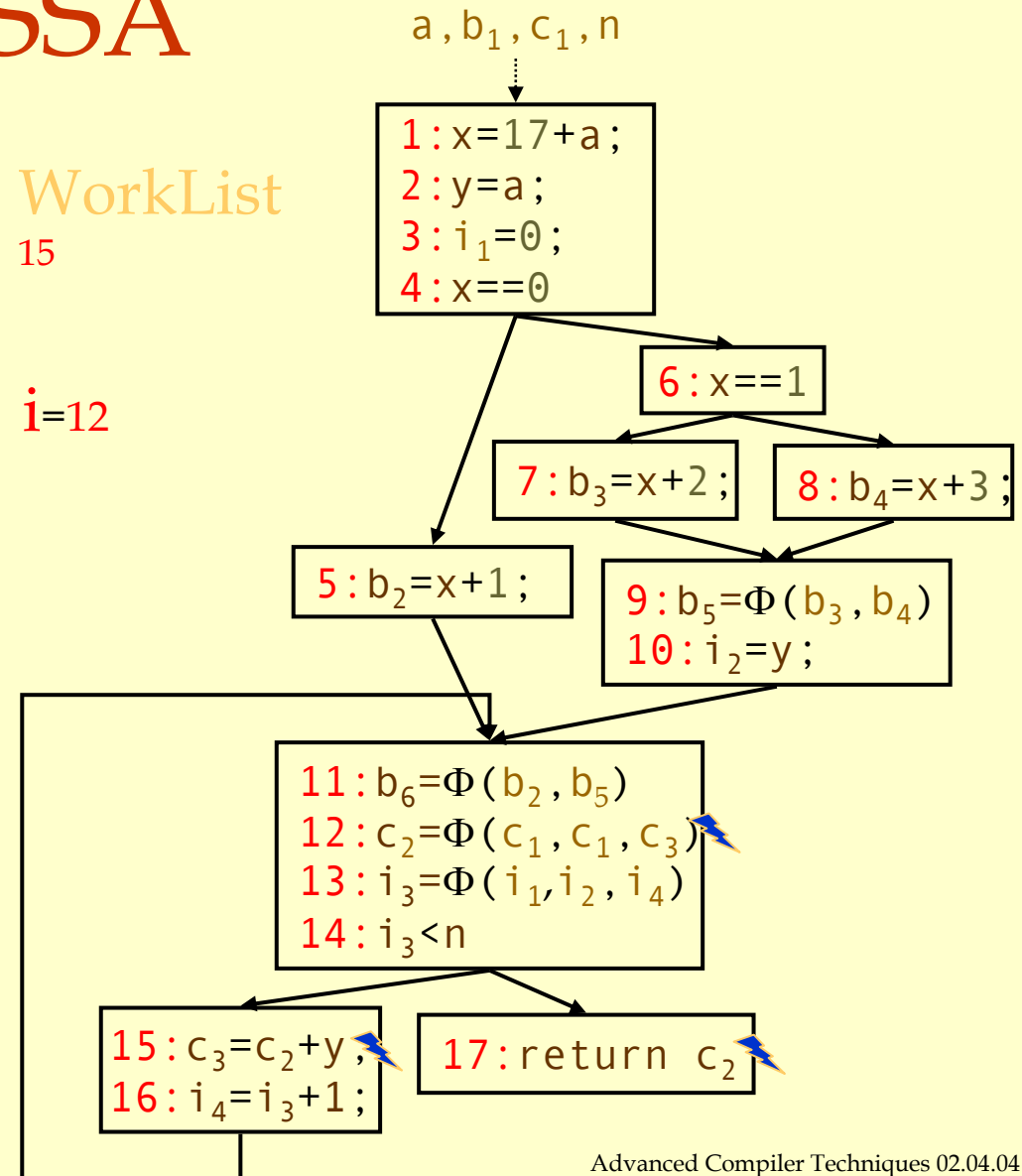
while (Worklist $\neq \emptyset$)
 remove i from WorkList
 (i has form " $x \leftarrow y \text{ op } z$ ")
 if $\text{def}(y)$ is not marked then
 mark $\text{def}(y)$
 add $\text{def}(y)$ to WorkList
 if $\text{def}(z)$ is not marked then
 mark $\text{def}(z)$
 add $\text{def}(z)$ to WorkList

for each $b \in \text{RDF}(\text{block}(i))$
 mark the block-ending
 branch in b
 add it to WorkList

WorkList

15

$i=12$



Dead Code Elimination Using SSA

Mark

for each op i
 clear i 's mark
 if i is critical then
 mark i
 add i to WorkList

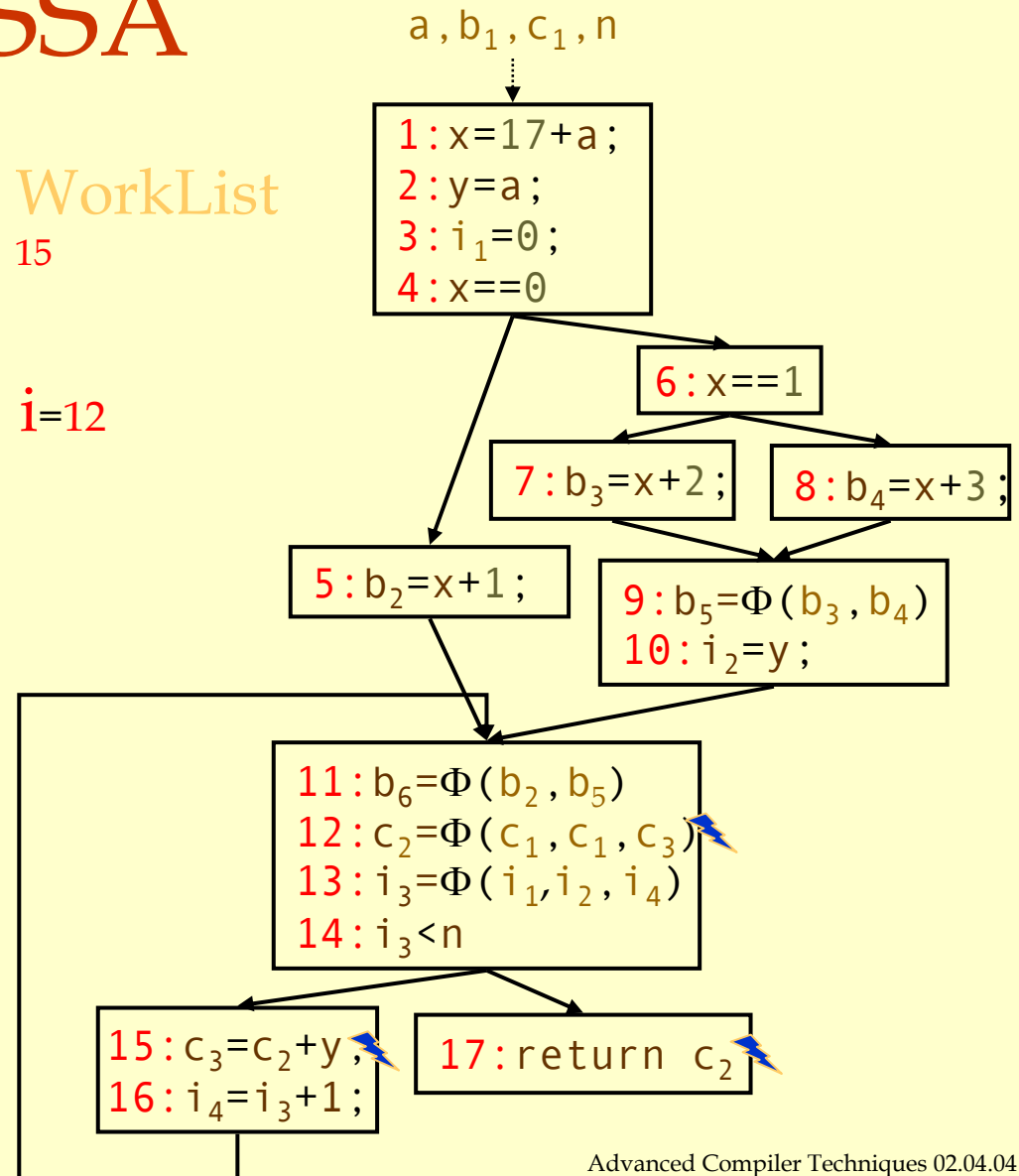
while (Worklist $\neq \emptyset$)
 remove i from WorkList
 (i has form " $x \leftarrow y \text{ op } z$ ")
 if $\text{def}(y)$ is not marked then
 mark $\text{def}(y)$
 add $\text{def}(y)$ to WorkList
 if $\text{def}(z)$ is not marked then
 mark $\text{def}(z)$
 add $\text{def}(z)$ to WorkList

for each $b \in \text{RDF}(\text{block}(i))$
 mark the block-ending
 branch in b
 add it to WorkList

WorkList

15

$i=12$



Dead Code Elimination Using SSA

Mark

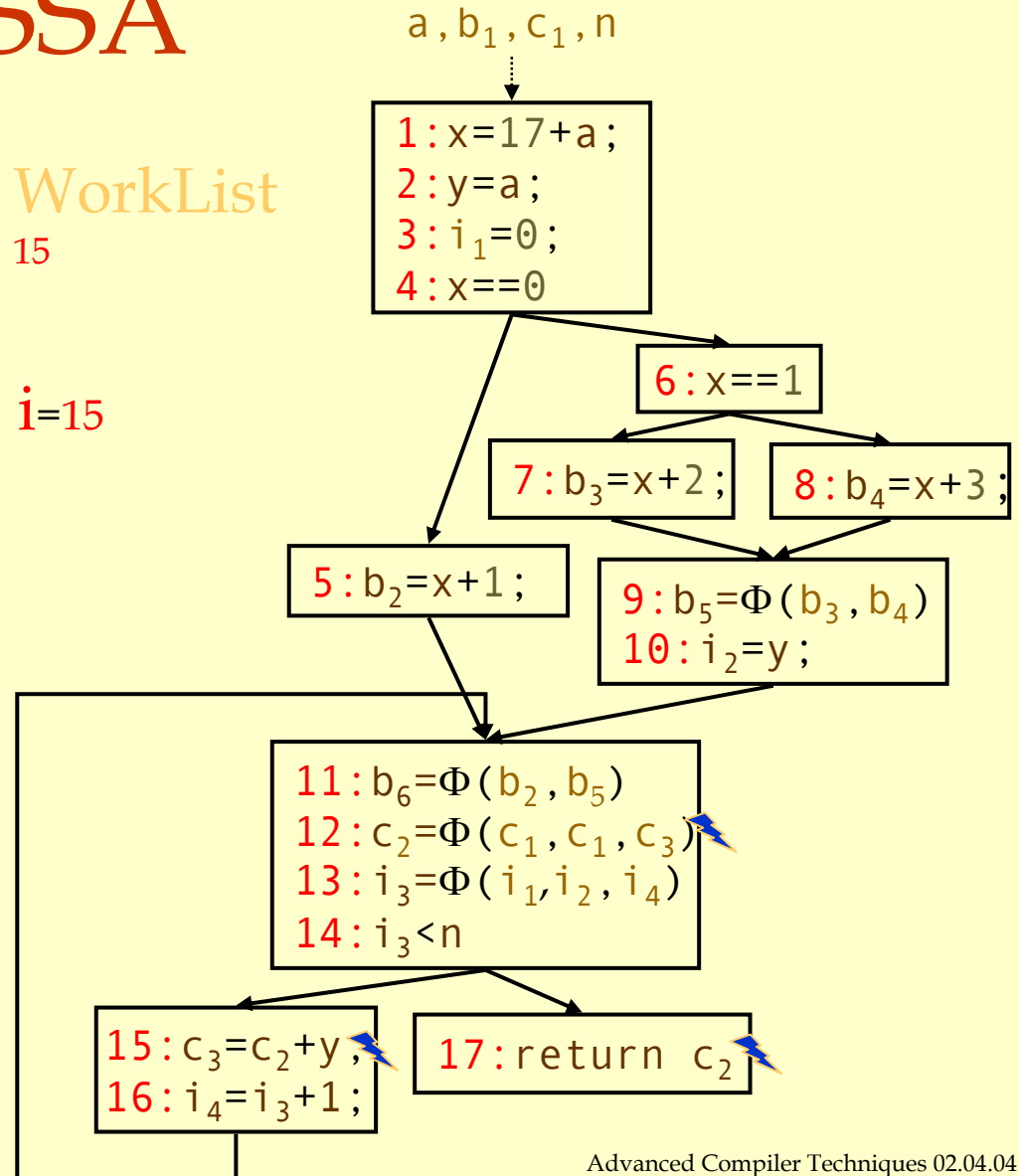
for each op i
 clear i 's mark
 if i is critical then
 mark i
 add i to WorkList

```
while (Worklist  $\neq \emptyset$ )
  remove  $i$  from WorkList
  ( $i$  has form " $x \leftarrow y \text{ op } z$ ")
  if  $\text{def}(y)$  is not marked then
    mark  $\text{def}(y)$ 
    add  $\text{def}(y)$  to WorkList
  if  $\text{def}(z)$  is not marked then
    mark  $\text{def}(z)$ 
    add  $\text{def}(z)$  to WorkList
  for each  $b \in \text{RDF}(\text{block}(i))$ 
    mark the block-ending
    branch in  $b$ 
    add it to WorkList
```

SSA

WorkList
15

$i=15$



Dead Code Elimination Using SSA

Mark

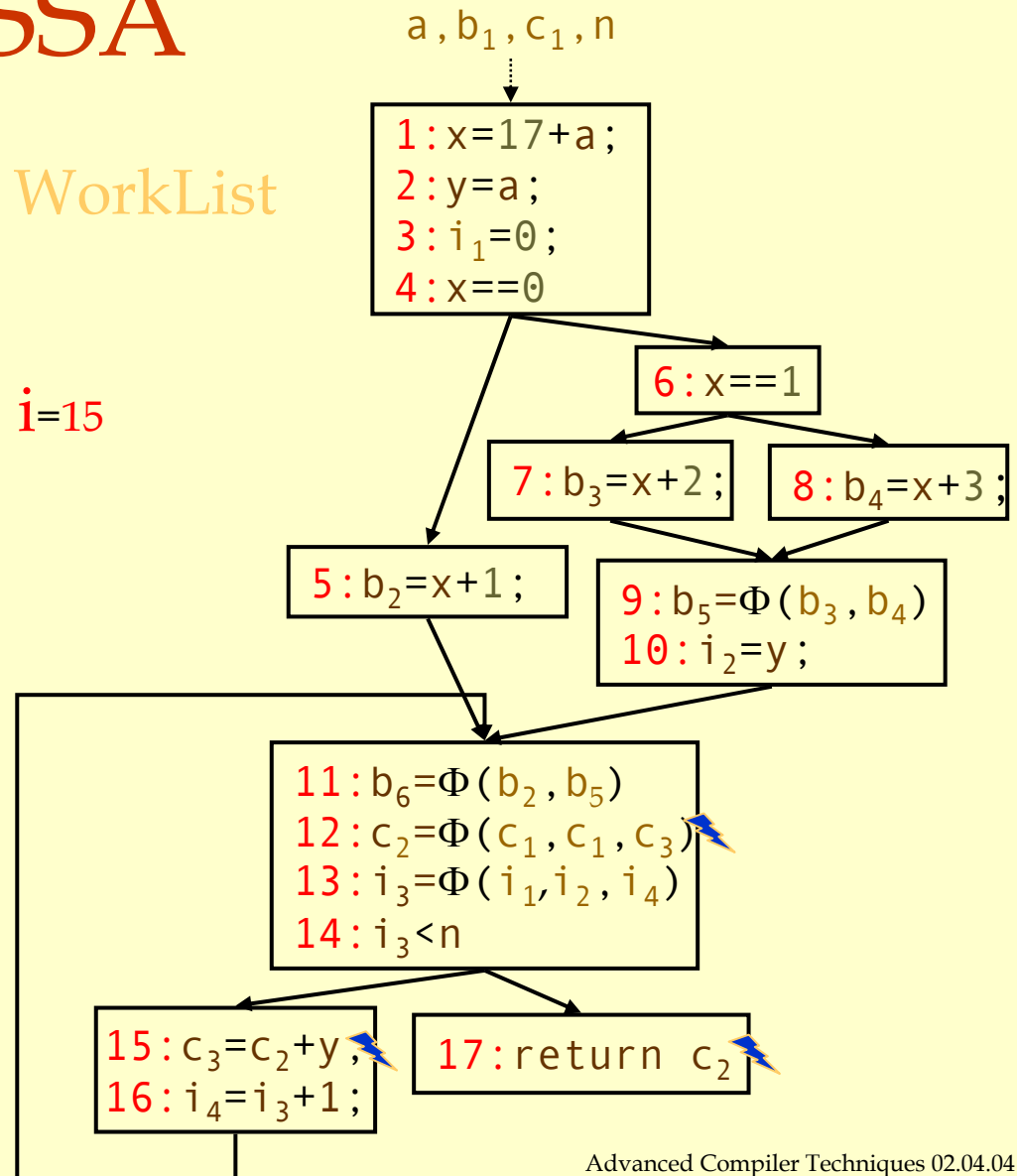
for each op i
 clear i 's mark
 if i is critical then
 mark i
 add i to **WorkList**

while (**Worklist** $\neq \emptyset$)
 remove i from **WorkList**
 (i has form " $x \leftarrow y \text{ op } z$ ")
 if $\text{def}(y)$ is not marked then
 mark $\text{def}(y)$
 add $\text{def}(y)$ to **WorkList**
 if $\text{def}(z)$ is not marked then
 mark $\text{def}(z)$
 add $\text{def}(z)$ to **WorkList**
 for each $b \in \text{RDF}(\text{block}(i))$
 mark the block-ending
 branch in b
 add it to **WorkList**

SSA

WorkList

$i=15$



Dead Code Elimination Using SSA

Mark

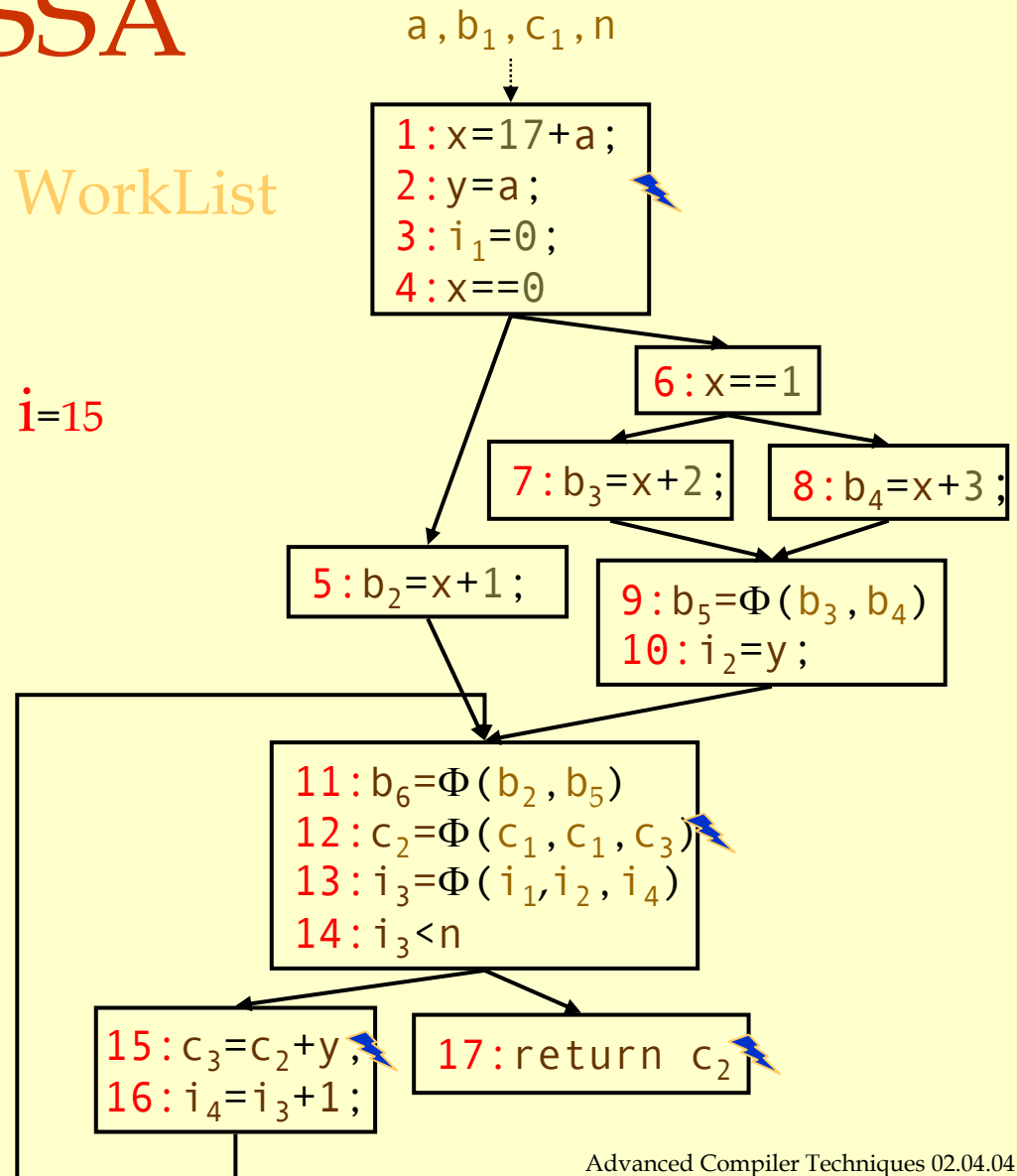
for each op i
 clear i 's mark
 if i is critical then
 mark i
 add i to WorkList

while (Worklist $\neq \emptyset$)
 remove i from WorkList
 (i has form " $x \leftarrow y \text{ op } z$ ")
 if $\text{def}(y)$ is not marked then
 mark $\text{def}(y)$
 add $\text{def}(y)$ to WorkList
 if $\text{def}(z)$ is not marked then
 mark $\text{def}(z)$
 add $\text{def}(z)$ to WorkList

for each $b \in \text{RDF}(\text{block}(i))$
 mark the block-ending
 branch in b
 add it to WorkList

WorkList

$i=15$



Dead Code Elimination Using SSA

Mark

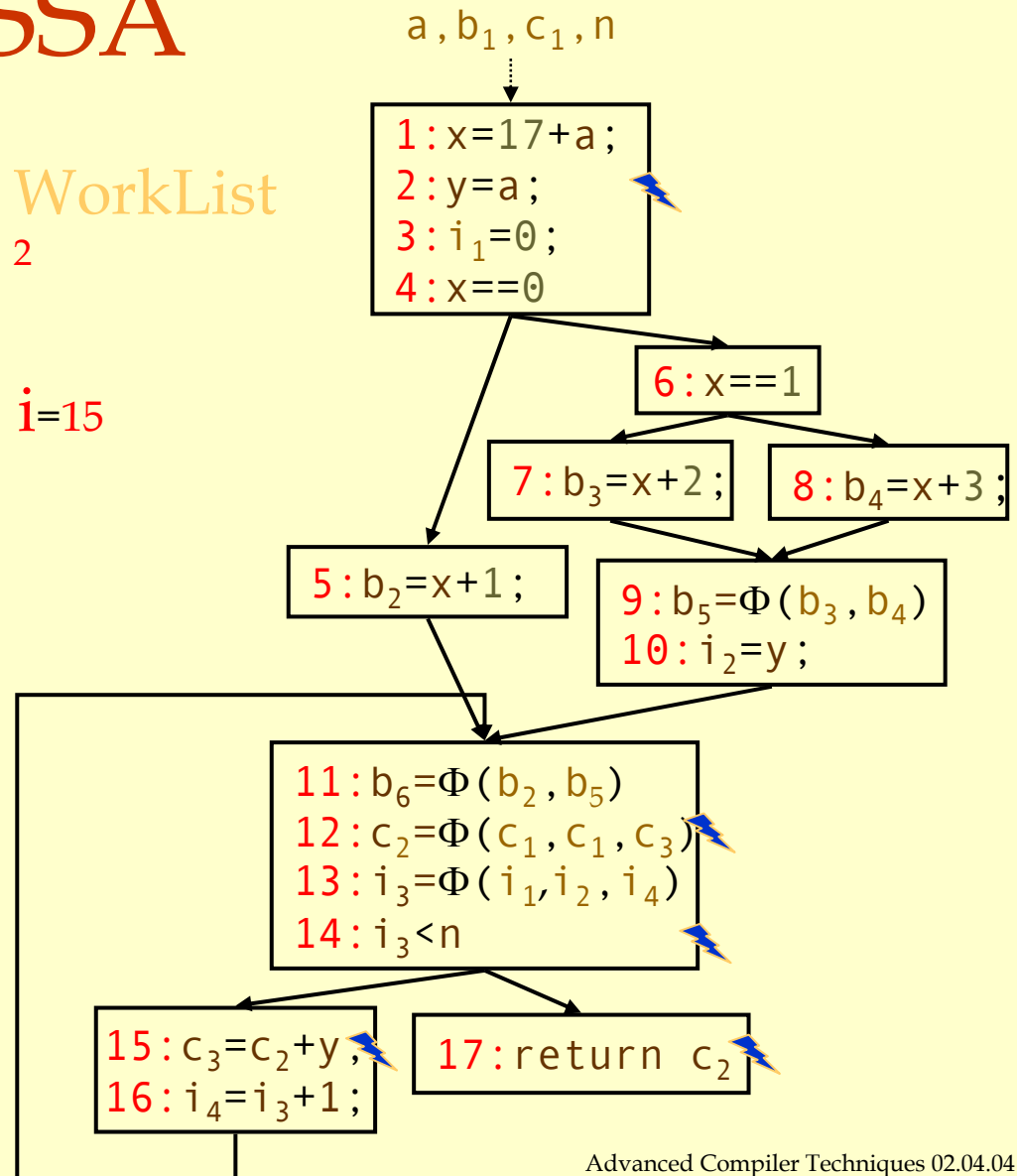
for each op i
 clear i 's mark
 if i is critical then
 mark i
 add i to WorkList

while (Worklist $\neq \emptyset$)
 remove i from WorkList
 (i has form " $x \leftarrow y \text{ op } z$ ")
 if $\text{def}(y)$ is not marked then
 mark $\text{def}(y)$
 add $\text{def}(y)$ to WorkList
 if $\text{def}(z)$ is not marked then
 mark $\text{def}(z)$
 add $\text{def}(z)$ to WorkList

for each $b \in \text{RDF}(\text{block}(i))$
 mark the block-ending
 branch in b
 add it to WorkList

WorkList
2

$i=15$



Dead Code Elimination Using SSA

Mark

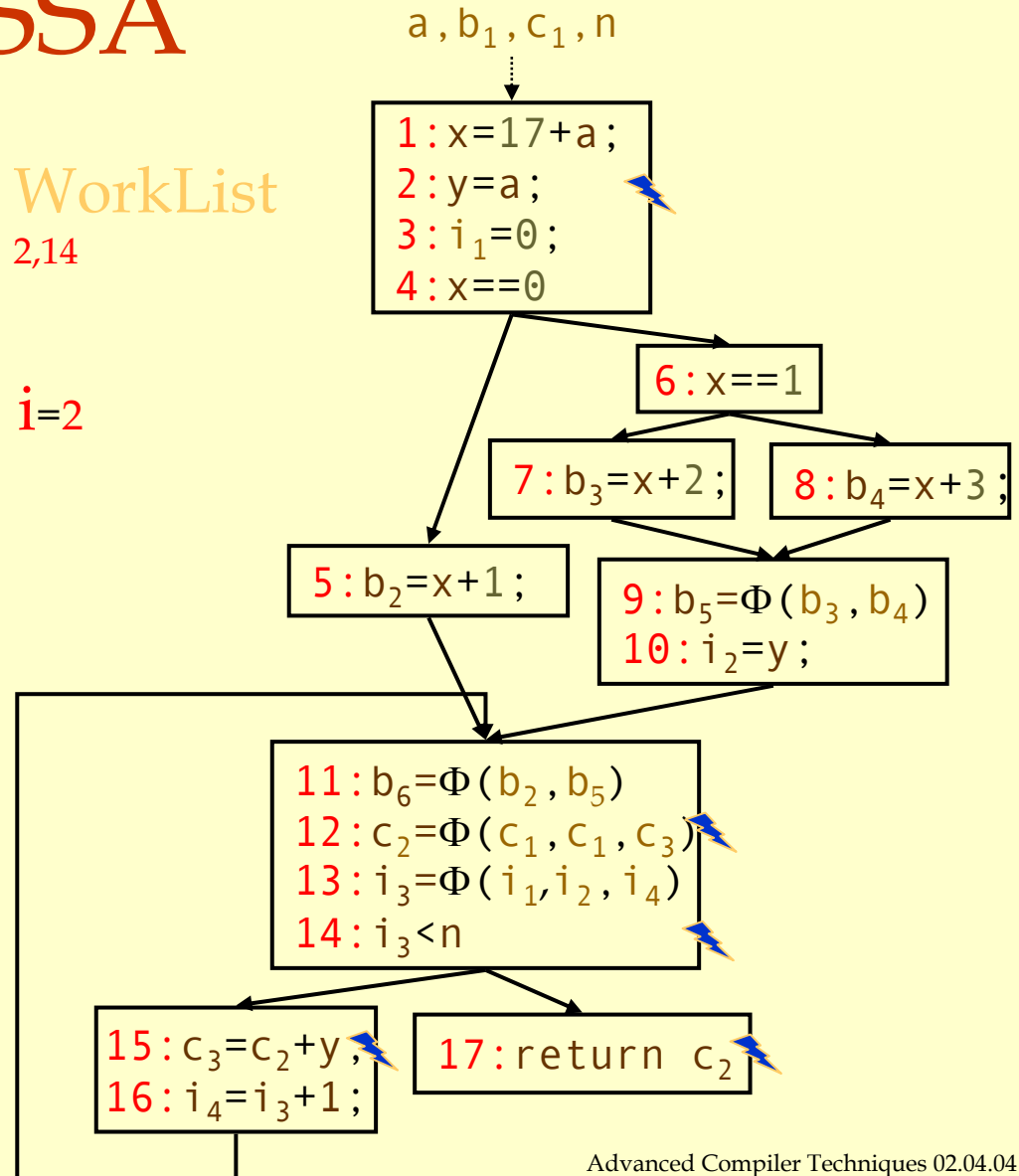
for each op i
 clear i 's mark
 if i is critical then
 mark i
 add i to **WorkList**

while (**Worklist** $\neq \emptyset$)
 remove i from **WorkList**
 (i has form " $x \leftarrow y \text{ op } z$ ")
 if $\text{def}(y)$ is not marked then
 mark $\text{def}(y)$
 add $\text{def}(y)$ to **WorkList**
 if $\text{def}(z)$ is not marked then
 mark $\text{def}(z)$
 add $\text{def}(z)$ to **WorkList**
 for each $b \in \text{RDF}(\text{block}(i))$
 mark the block-ending
 branch in b
 add it to **WorkList**

SSA

WorkList
 2,14

$i=2$



Dead Code Elimination Using SSA

Mark

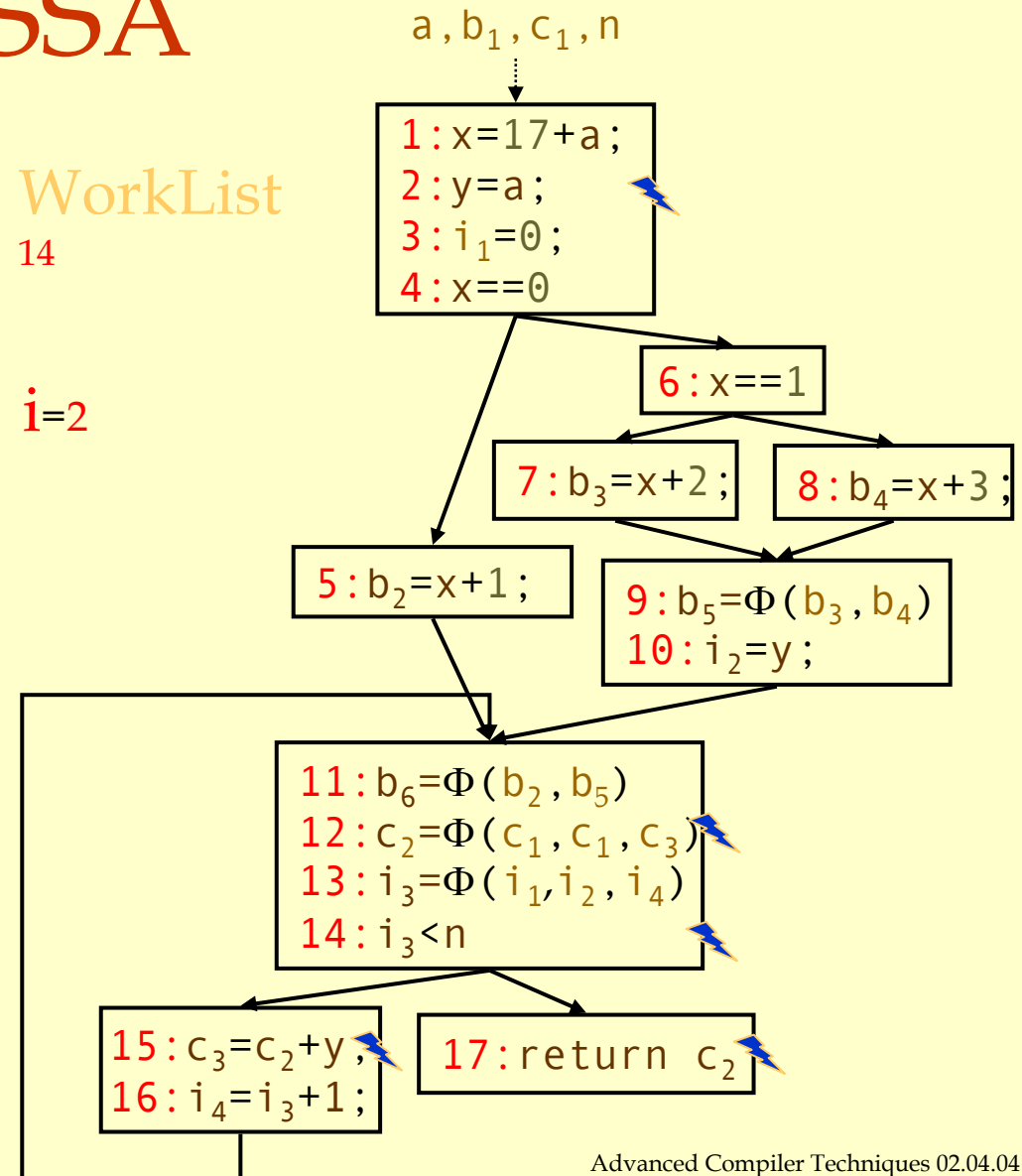
for each op i
 clear i 's mark
 if i is critical then
 mark i
 add i to WorkList

while (Worklist $\neq \emptyset$)
 remove i from WorkList
 (i has form " $x \leftarrow y \text{ op } z$ ")
 if $\text{def}(y)$ is not marked then
 mark $\text{def}(y)$
 add $\text{def}(y)$ to WorkList
 if $\text{def}(z)$ is not marked then
 mark $\text{def}(z)$
 add $\text{def}(z)$ to WorkList
 for each $b \in \text{RDF}(\text{block}(i))$
 mark the block-ending
 branch in b
 add it to WorkList

SSA

WorkList

14

 $i=2$ 

Dead Code Elimination Using SSA

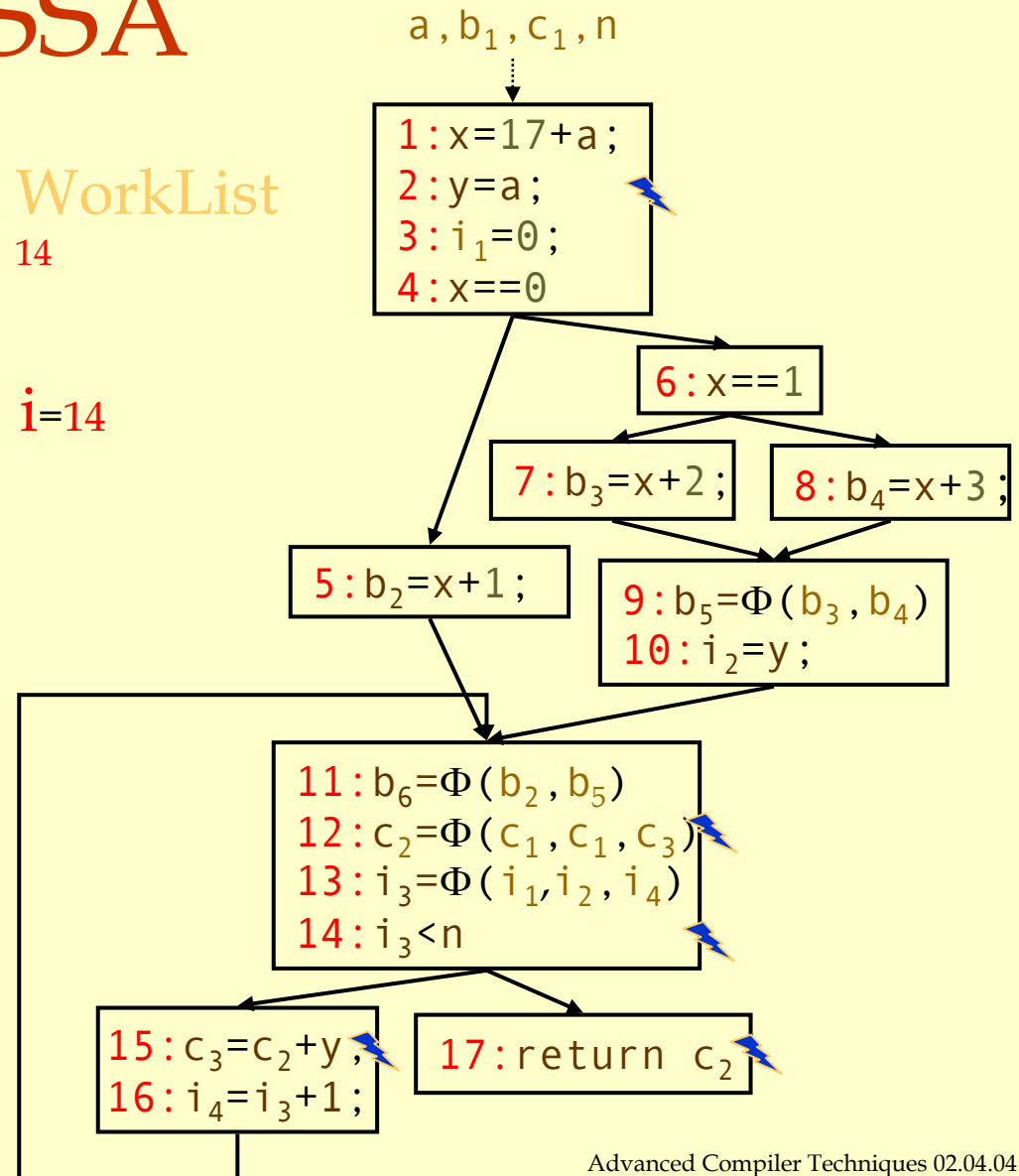
Mark

for each op i
 clear i 's mark
 if i is critical then
 mark i
 add i to WorkList

while (Worklist $\neq \emptyset$)
 remove i from WorkList
 (i has form " $x \leftarrow y \text{ op } z$ ")
 if $\text{def}(y)$ is not marked then
 mark $\text{def}(y)$
 add $\text{def}(y)$ to WorkList
 if $\text{def}(z)$ is not marked then
 mark $\text{def}(z)$
 add $\text{def}(z)$ to WorkList
 for each $b \in \text{RDF}(\text{block}(i))$
 mark the block-ending
 branch in b
 add it to WorkList

WorkList

14

 $i=14$ 

Dead Code Elimination Using SSA

Mark

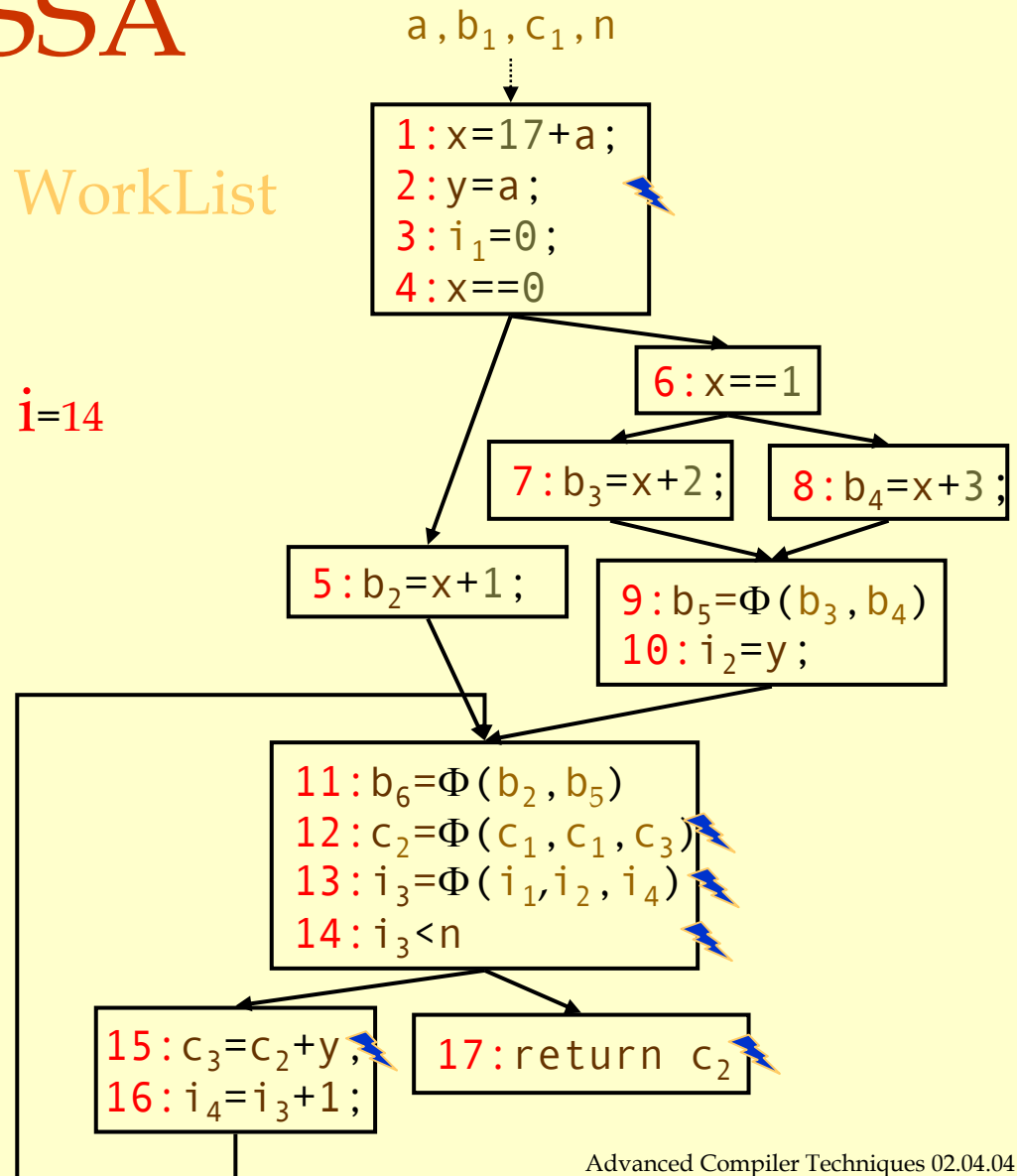
for each op i
 clear i 's mark
 if i is critical then
 mark i
 add i to **WorkList**

while (**Worklist** $\neq \emptyset$)
 remove i from **WorkList**
 (i has form " $x \leftarrow y \text{ op } z$ ")
 if $\text{def}(y)$ is not marked then
 mark $\text{def}(y)$
 add $\text{def}(y)$ to **WorkList**
 if $\text{def}(z)$ is not marked then
 mark $\text{def}(z)$
 add $\text{def}(z)$ to **WorkList**
 for each $b \in \text{RDF}(\text{block}(i))$
 mark the block-ending
 branch in b
 add it to **WorkList**

SSA

WorkList

$i=14$



Dead Code Elimination Using SSA

Mark

for each op i
 clear i 's mark
 if i is critical then
 mark i
 add i to WorkList

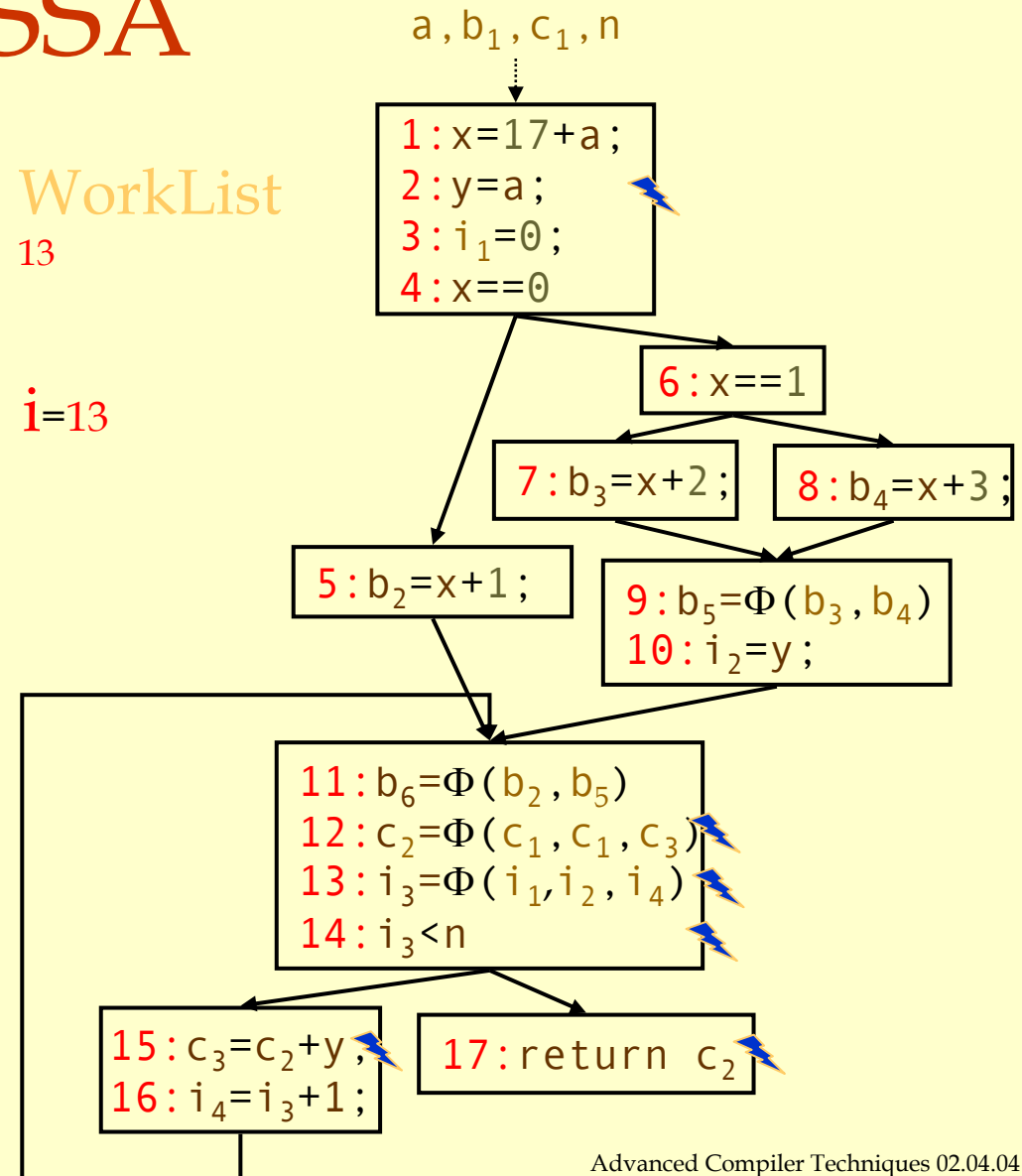
while (Worklist $\neq \emptyset$)
 remove i from WorkList
 (i has form " $x \leftarrow y \text{ op } z$ ")
 if $\text{def}(y)$ is not marked then
 mark $\text{def}(y)$
 add $\text{def}(y)$ to WorkList
 if $\text{def}(z)$ is not marked then
 mark $\text{def}(z)$
 add $\text{def}(z)$ to WorkList
 for each $b \in \text{RDF}(\text{block}(i))$
 mark the block-ending
 branch in b
 add it to WorkList

SSA

WorkList

13

$i=13$



Dead Code Elimination Using SSA

Mark

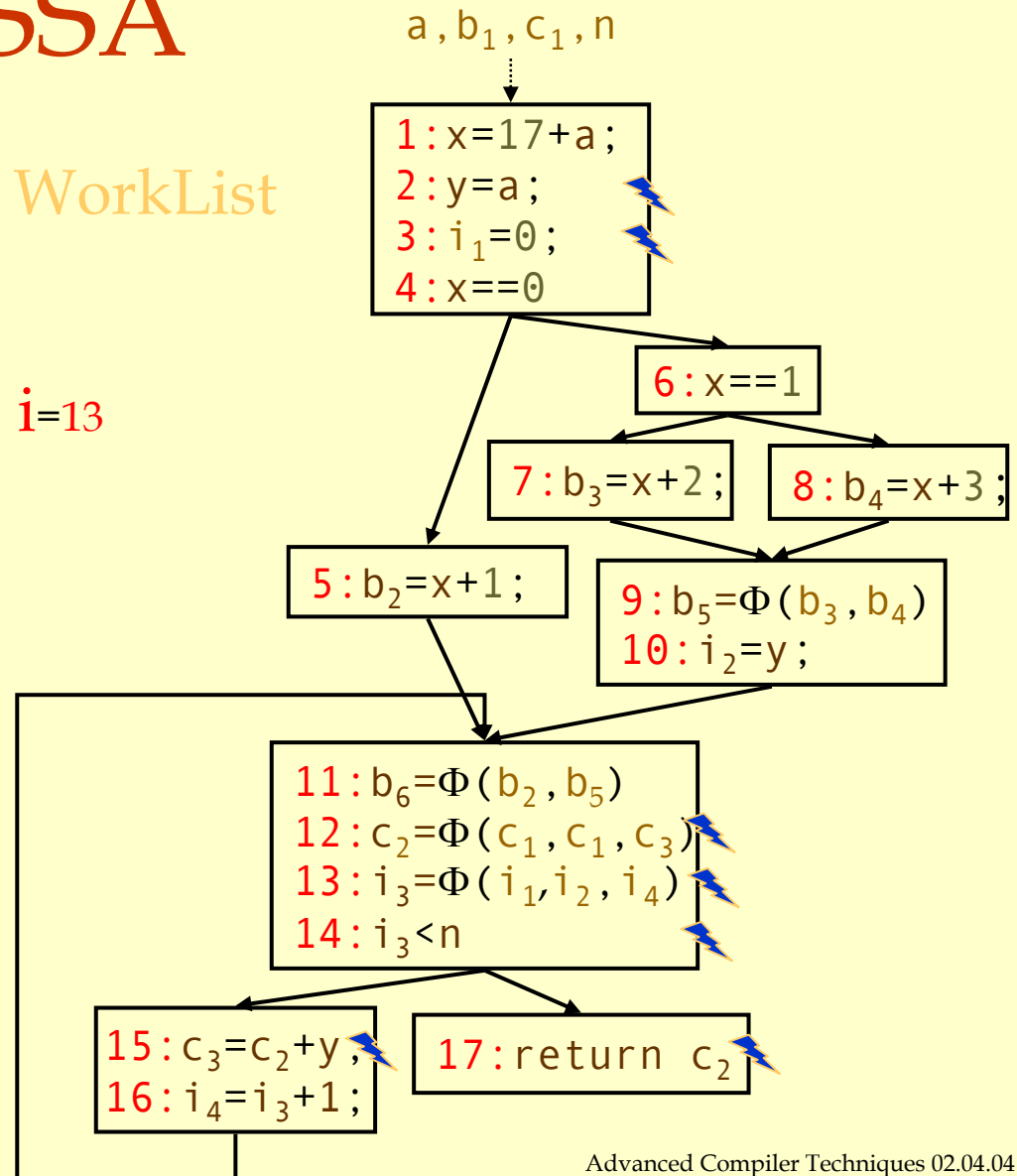
for each op i
 clear i 's mark
 if i is critical then
 mark i
 add i to **WorkList**

while (**Worklist** $\neq \emptyset$)
 remove i from **WorkList**
 (i has form " $x \leftarrow y \text{ op } z$ ")
 if $\text{def}(y)$ is not marked then
 mark $\text{def}(y)$
 add $\text{def}(y)$ to **WorkList**
 if $\text{def}(z)$ is not marked then
 mark $\text{def}(z)$
 add $\text{def}(z)$ to **WorkList**
 for each $b \in \text{RDF}(\text{block}(i))$
 mark the block-ending
 branch in b
 add it to **WorkList**

SSA

WorkList

$i=13$



Dead Code Elimination Using SSA

Mark

for each op i
 clear i 's mark
 if i is critical then
 mark i
 add i to **WorkList**

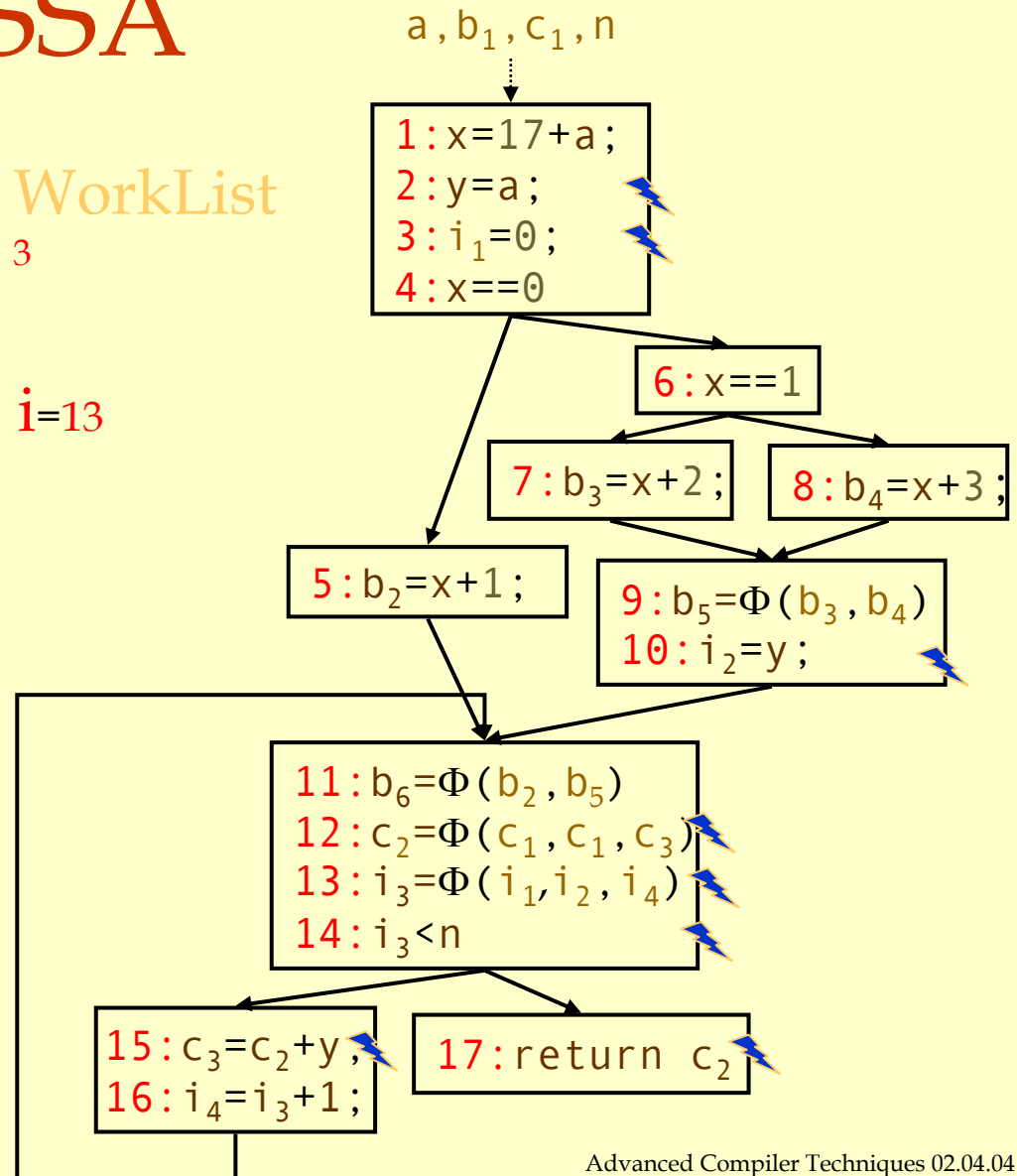
while (**Worklist** $\neq \emptyset$)
 remove i from **WorkList**
 (i has form " $x \leftarrow y \text{ op } z$ ")
 if $\text{def}(y)$ is not marked then
 mark $\text{def}(y)$
 add $\text{def}(y)$ to **WorkList**

if $\text{def}(z)$ is not marked then
 mark $\text{def}(z)$
 add $\text{def}(z)$ to **WorkList**

for each $b \in \text{RDF}(\text{block}(i))$
 mark the block-ending
 branch in b
 add it to **WorkList**

WorkList

3

 $i=13$ 

Dead Code Elimination Using SSA

Mark

for each op i
 clear i 's mark
 if i is critical then
 mark i
 add i to **WorkList**

while (**Worklist** $\neq \emptyset$)
 remove i from **WorkList**
 (i has form " $x \leftarrow y \text{ op } z$ ")
 if $\text{def}(y)$ is not marked then
 mark $\text{def}(y)$
 add $\text{def}(y)$ to **WorkList**

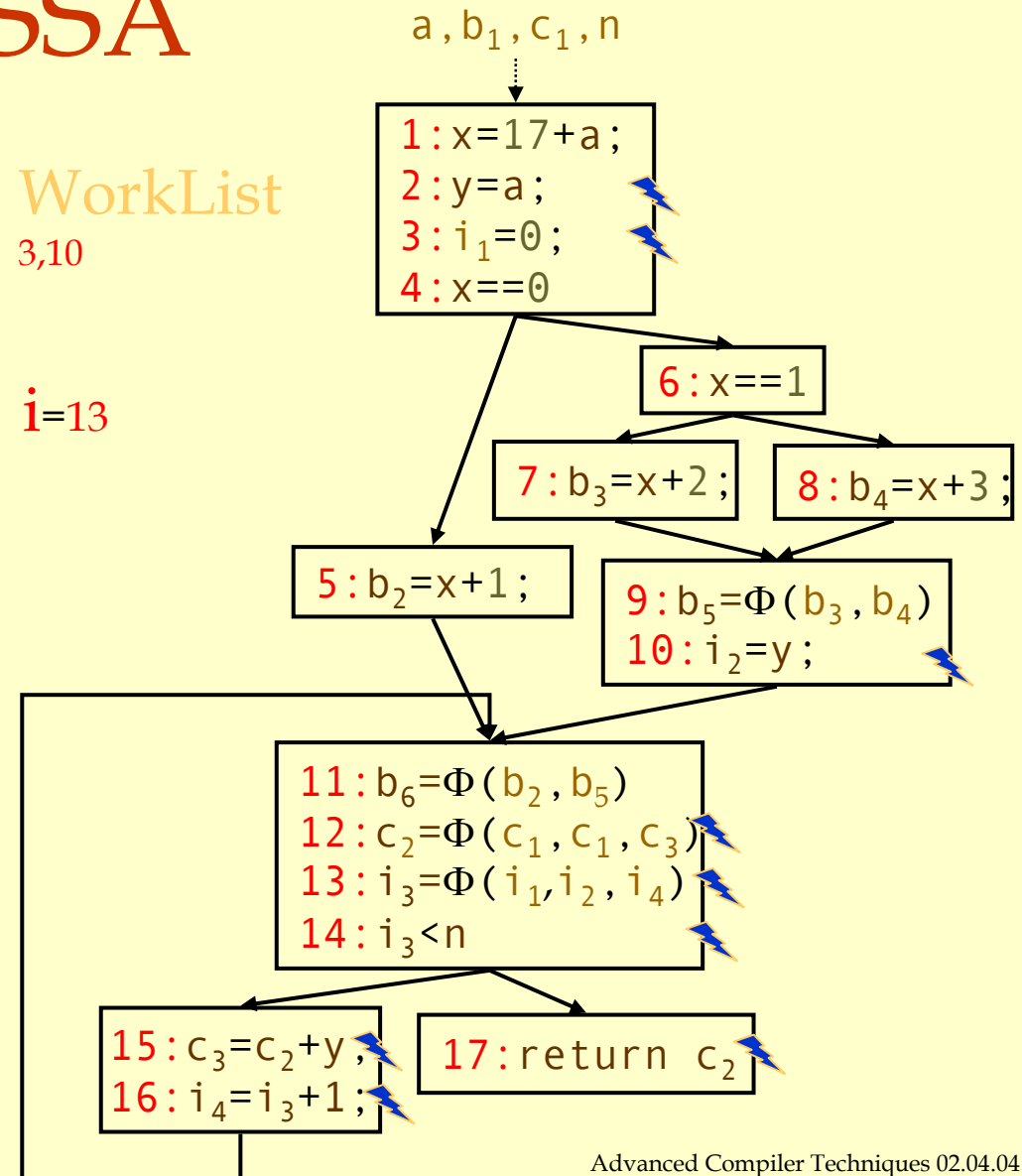
if $\text{def}(z)$ is not marked then
 mark $\text{def}(z)$
 add $\text{def}(z)$ to **WorkList**

for each $b \in \text{RDF}(\text{block}(i))$
 mark the block-ending
 branch in b
 add it to **WorkList**

WorkList

3,10

$i=13$



Dead Code Elimination Using SSA

Mark

for each op i
 clear i 's mark
 if i is critical then
 mark i
 add i to **WorkList**

while (**Worklist** $\neq \emptyset$)
 remove i from **WorkList**
 (i has form " $x \leftarrow y \text{ op } z$ ")
 if $\text{def}(y)$ is not marked then
 mark $\text{def}(y)$
 add $\text{def}(y)$ to **WorkList**

if $\text{def}(z)$ is not marked then
 mark $\text{def}(z)$
 add $\text{def}(z)$ to **WorkList**

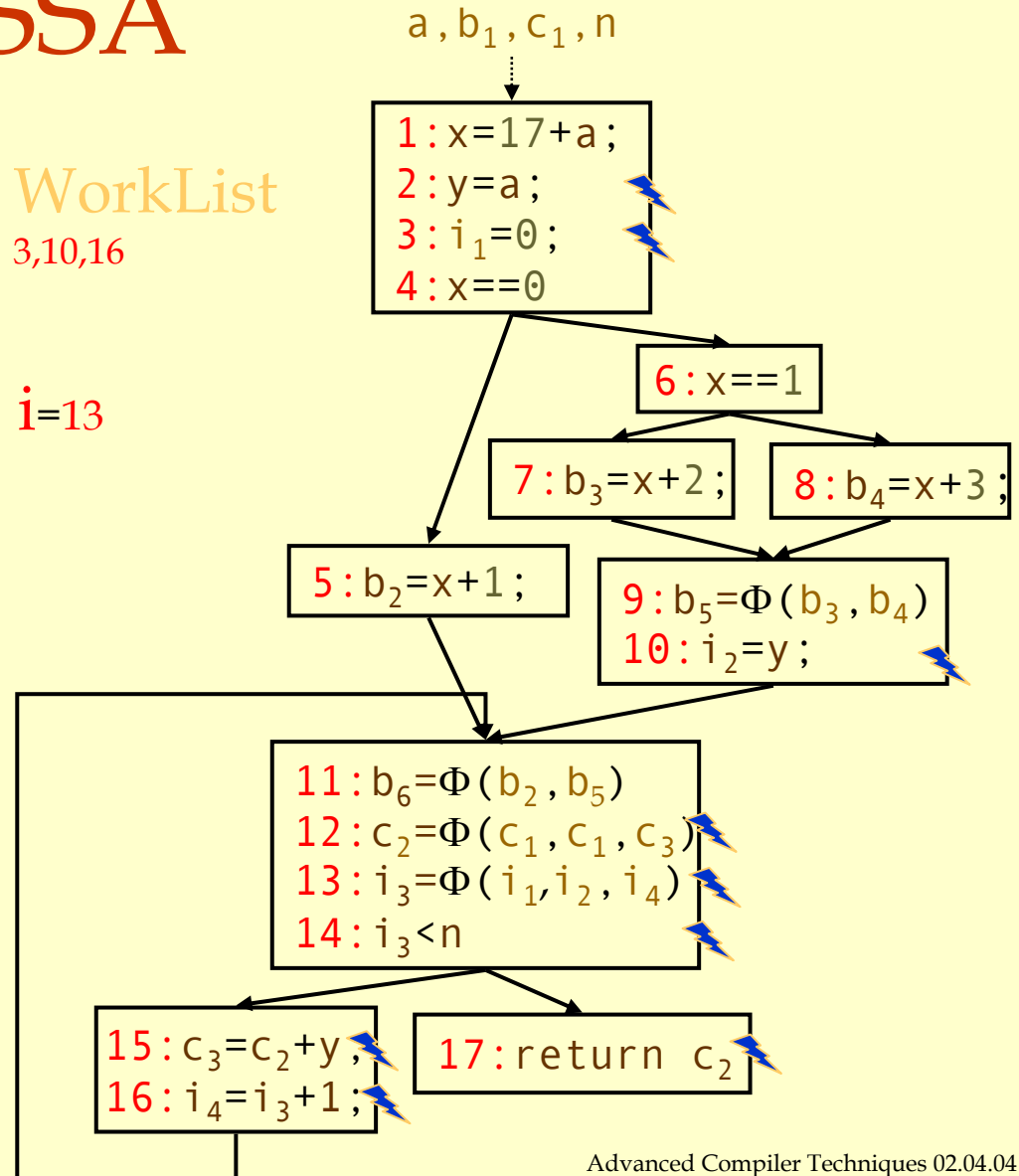
for each $b \in \text{RDF}(\text{block}(i))$
 mark the block-ending
 branch in b
 add it to **WorkList**

SSA

WorkList

3,10,16

$i=13$



Dead Code Elimination Using SSA

Mark

for each op i
 clear i 's mark
 if i is critical then
 mark i
 add i to **WorkList**

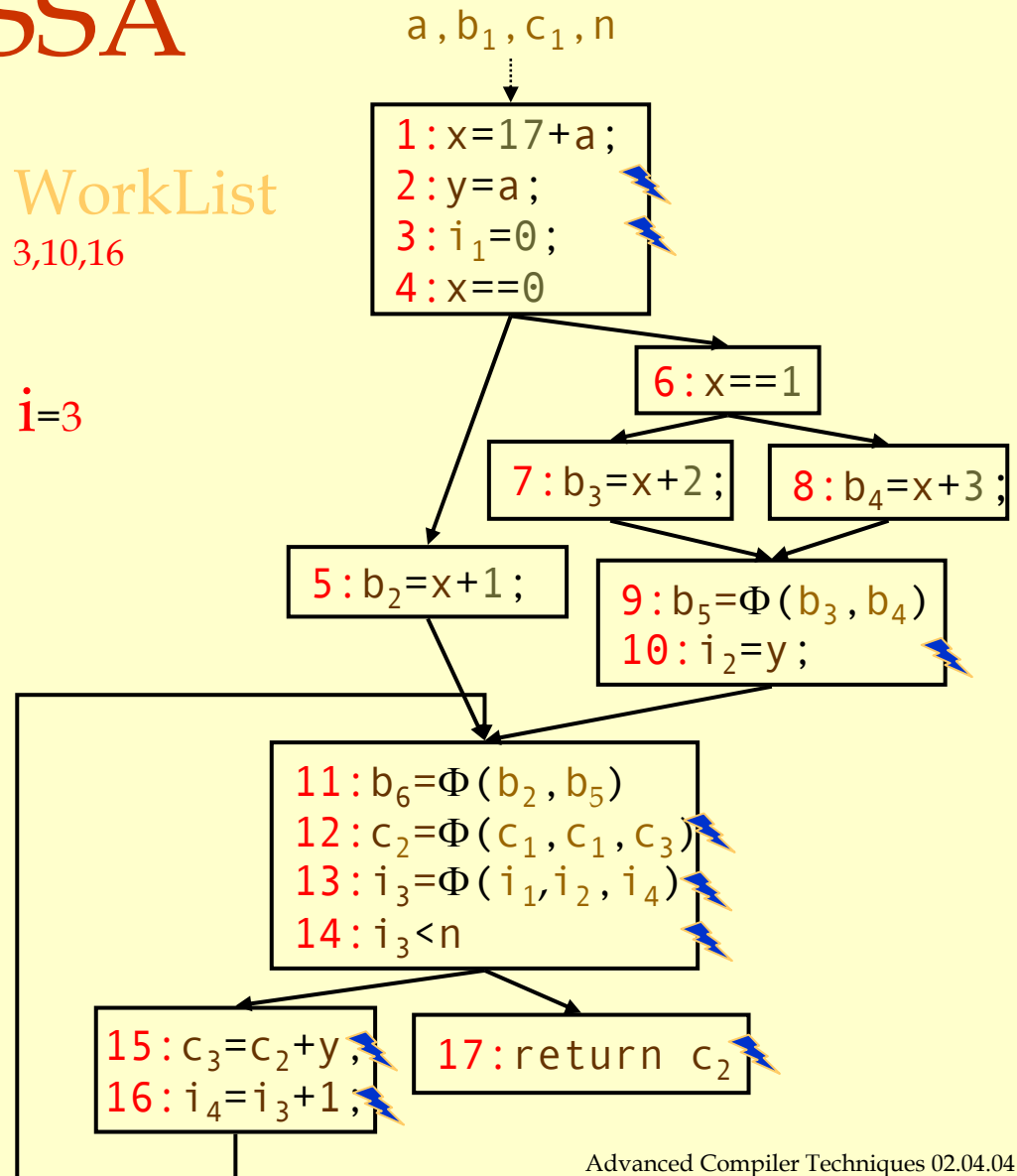
while (**Worklist** $\neq \emptyset$)
 remove i from **WorkList**
 (i has form " $x \leftarrow y \text{ op } z$ ")
 if $\text{def}(y)$ is not marked then
 mark $\text{def}(y)$
 add $\text{def}(y)$ to **WorkList**
 if $\text{def}(z)$ is not marked then
 mark $\text{def}(z)$
 add $\text{def}(z)$ to **WorkList**
 for each $b \in \text{RDF}(\text{block}(i))$
 mark the block-ending
 branch in b
 add it to **WorkList**

SSA

WorkList

3,10,16

$i=3$



Dead Code Elimination Using SSA

Mark

for each op i
 clear i 's mark
 if i is critical then
 mark i
 add i to **WorkList**

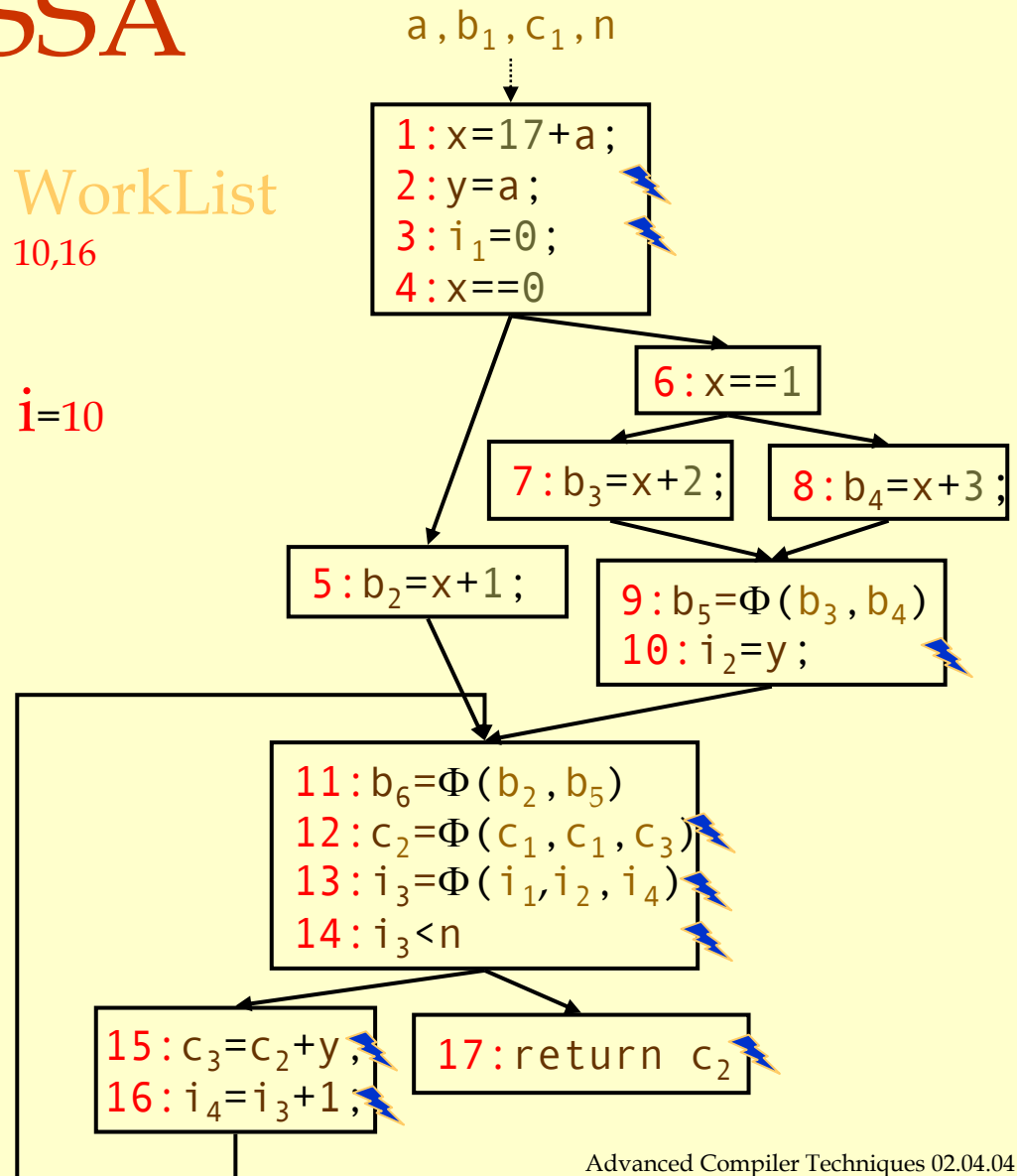
```
while (Worklist  $\neq \emptyset$ )
  remove  $i$  from WorkList
  ( $i$  has form " $x \leftarrow y \text{ op } z$ ")
  if  $\text{def}(y)$  is not marked then
    mark  $\text{def}(y)$ 
    add  $\text{def}(y)$  to WorkList
  if  $\text{def}(z)$  is not marked then
    mark  $\text{def}(z)$ 
    add  $\text{def}(z)$  to WorkList
  for each  $b \in \text{RDF}(\text{block}(i))$ 
    mark the block-ending
    branch in  $b$ 
    add it to WorkList
```

SSA

WorkList

10,16

$i=10$



Dead Code Elimination Using SSA

Mark

for each op i
 clear i 's mark
 if i is critical then
 mark i
 add i to **WorkList**

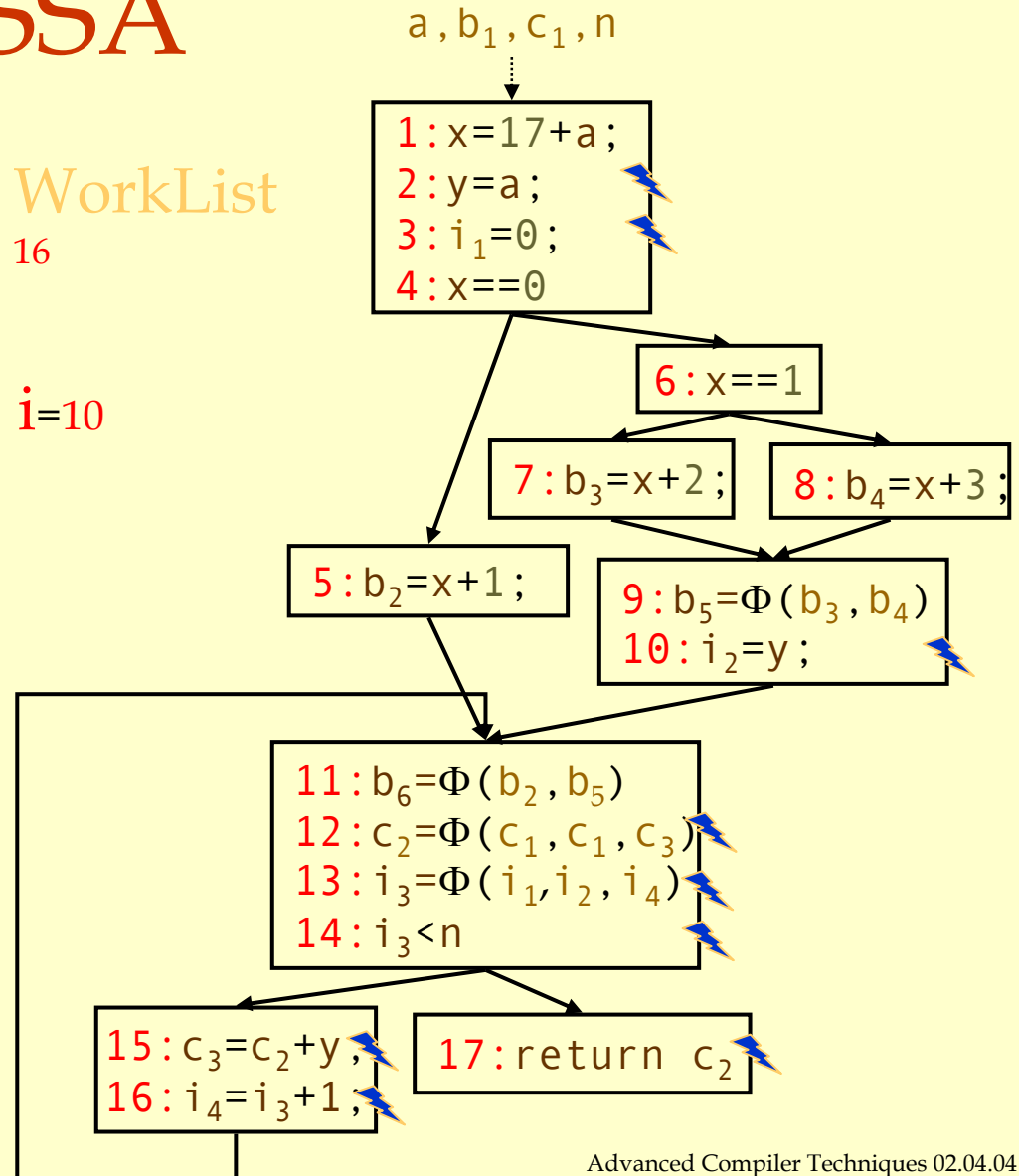
while (**Worklist** $\neq \emptyset$)
 remove i from **WorkList**
 (i has form " $x \leftarrow y \text{ op } z$ ")
 if $\text{def}(y)$ is not marked then
 mark $\text{def}(y)$
 add $\text{def}(y)$ to **WorkList**
 if $\text{def}(z)$ is not marked then
 mark $\text{def}(z)$
 add $\text{def}(z)$ to **WorkList**

for each $b \in \text{RDF}(\text{block}(i))$
 mark the block-ending
 branch in b
 add it to **WorkList**

SSA

WorkList
 16

$i=10$



Dead Code Elimination Using SSA

Mark

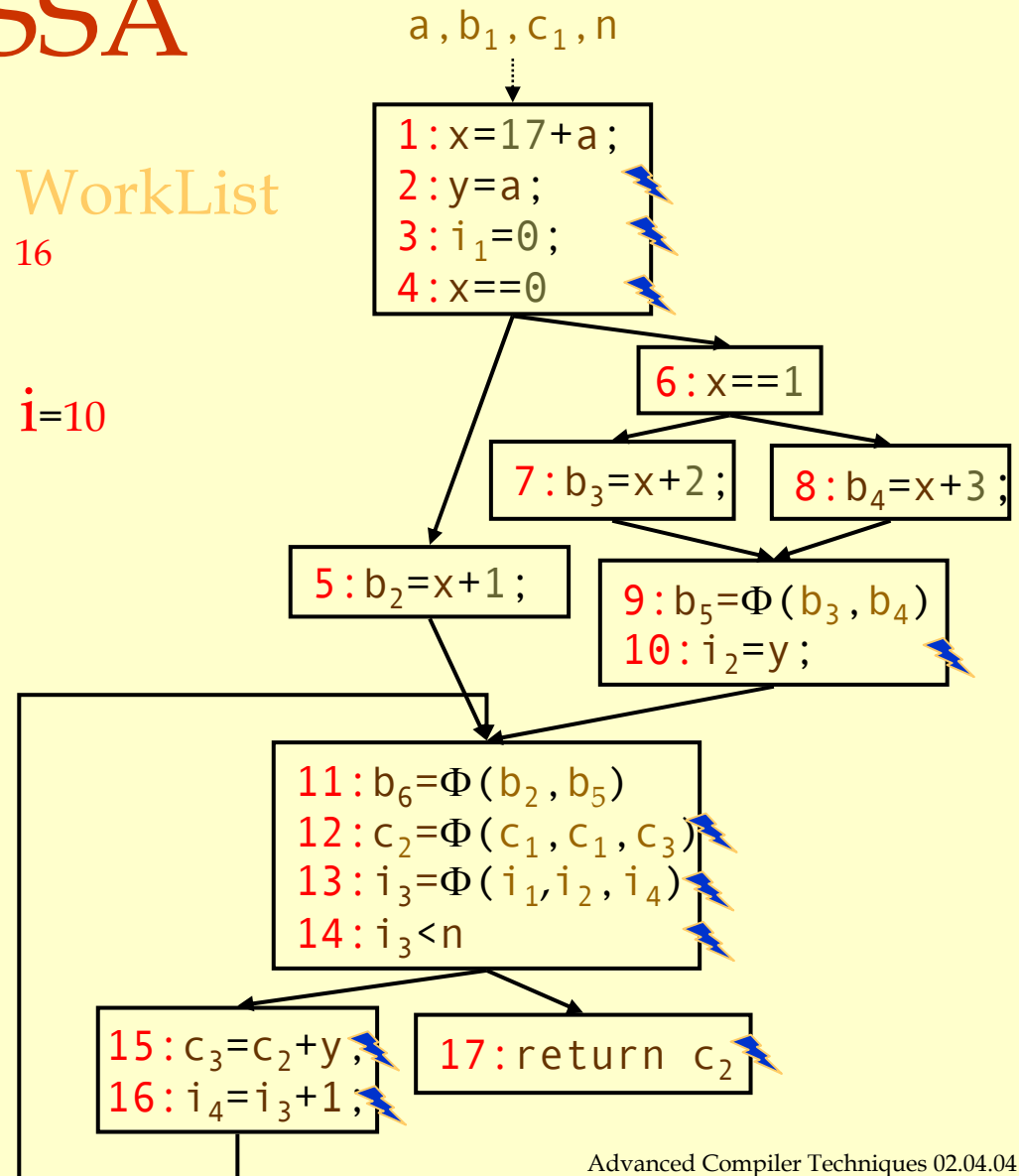
for each op i
 clear i 's mark
 if i is critical then
 mark i
 add i to WorkList

while (Worklist $\neq \emptyset$)
 remove i from WorkList
 (i has form " $x \leftarrow y \text{ op } z$ ")
 if $\text{def}(y)$ is not marked then
 mark $\text{def}(y)$
 add $\text{def}(y)$ to WorkList
 if $\text{def}(z)$ is not marked then
 mark $\text{def}(z)$
 add $\text{def}(z)$ to WorkList

for each $b \in \text{RDF}(\text{block}(i))$
 mark the block-ending
 branch in b
 add it to WorkList

WorkList

16

 $i=10$ 

Dead Code Elimination Using SSA

Mark

for each op i
 clear i 's mark
 if i is critical then
 mark i
 add i to **WorkList**

while (**Worklist** $\neq \emptyset$)

remove i from **WorkList**
 (i has form " $x \leftarrow y \text{ op } z$ ")

if $\text{def}(y)$ is not marked then
 mark $\text{def}(y)$
 add $\text{def}(y)$ to **WorkList**
 if $\text{def}(z)$ is not marked then
 mark $\text{def}(z)$
 add $\text{def}(z)$ to **WorkList**

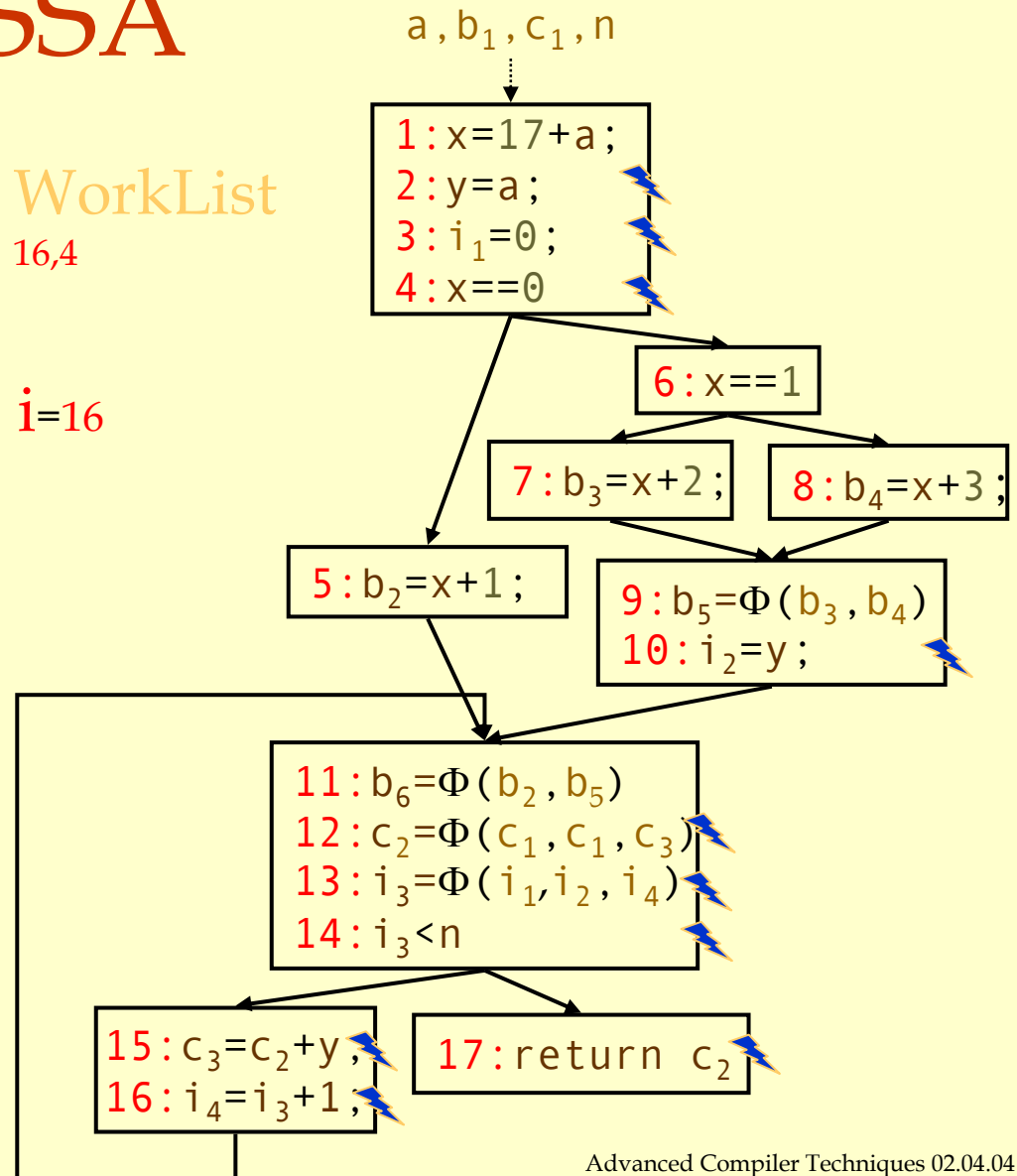
for each $b \in \text{RDF}(\text{block}(i))$
 mark the block-ending
 branch in b
 add it to **WorkList**

SSA

WorkList

16,4

$i=16$



Dead Code Elimination Using SSA

Mark

for each op i
 clear i 's mark
 if i is critical then
 mark i
 add i to **WorkList**

while (**Worklist** $\neq \emptyset$)

remove i from **WorkList**
 (i has form " $x \leftarrow y \text{ op } z$ ")

if $\text{def}(y)$ is not marked then
 mark $\text{def}(y)$

add $\text{def}(y)$ to **WorkList**

if $\text{def}(z)$ is not marked then
 mark $\text{def}(z)$

add $\text{def}(z)$ to **WorkList**

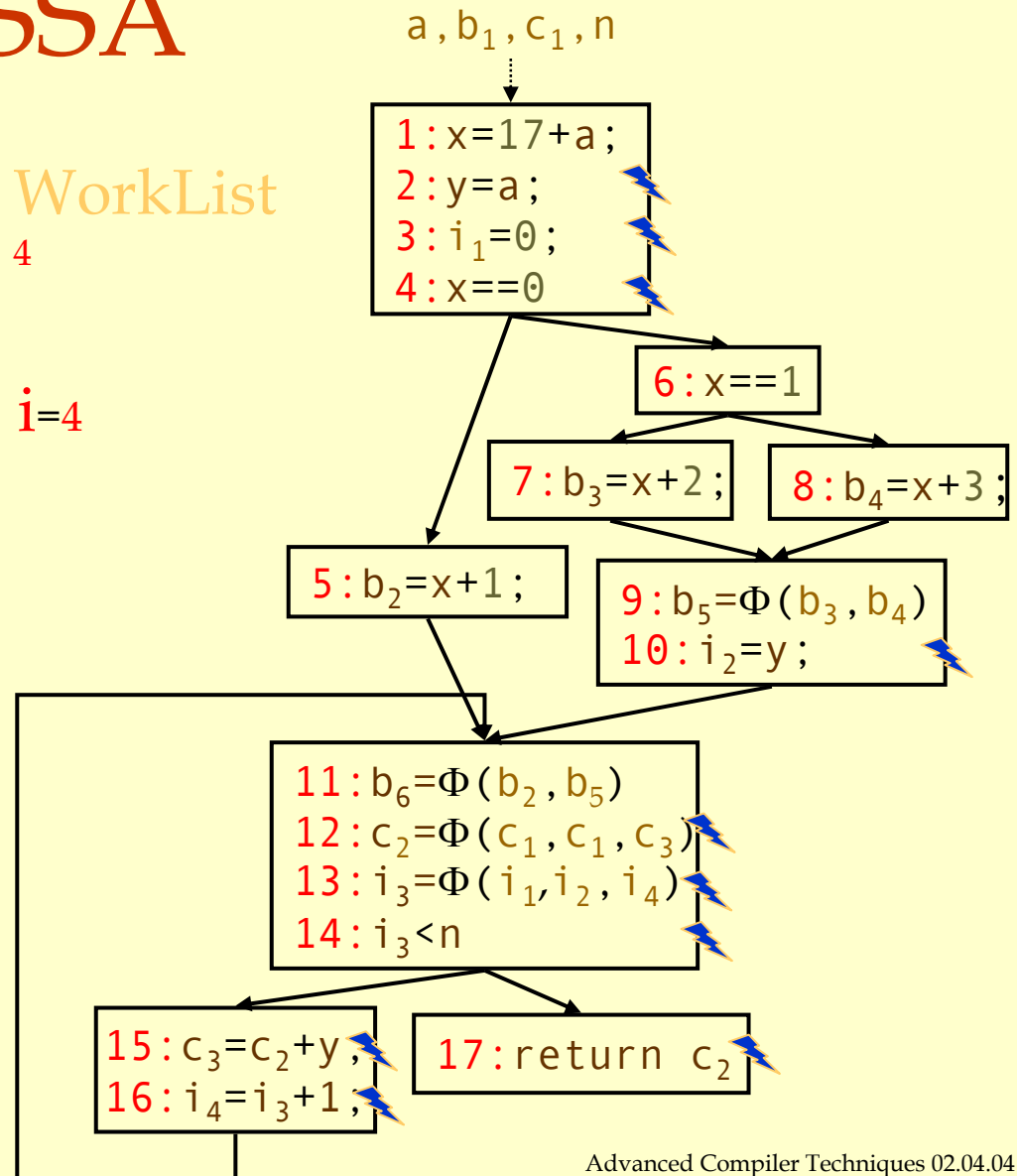
for each $b \in \text{RDF}(\text{block}(i))$
 mark the block-ending
 branch in b
 add it to **WorkList**

SSA

WorkList

4

$i=4$



Dead Code Elimination Using SSA

Mark

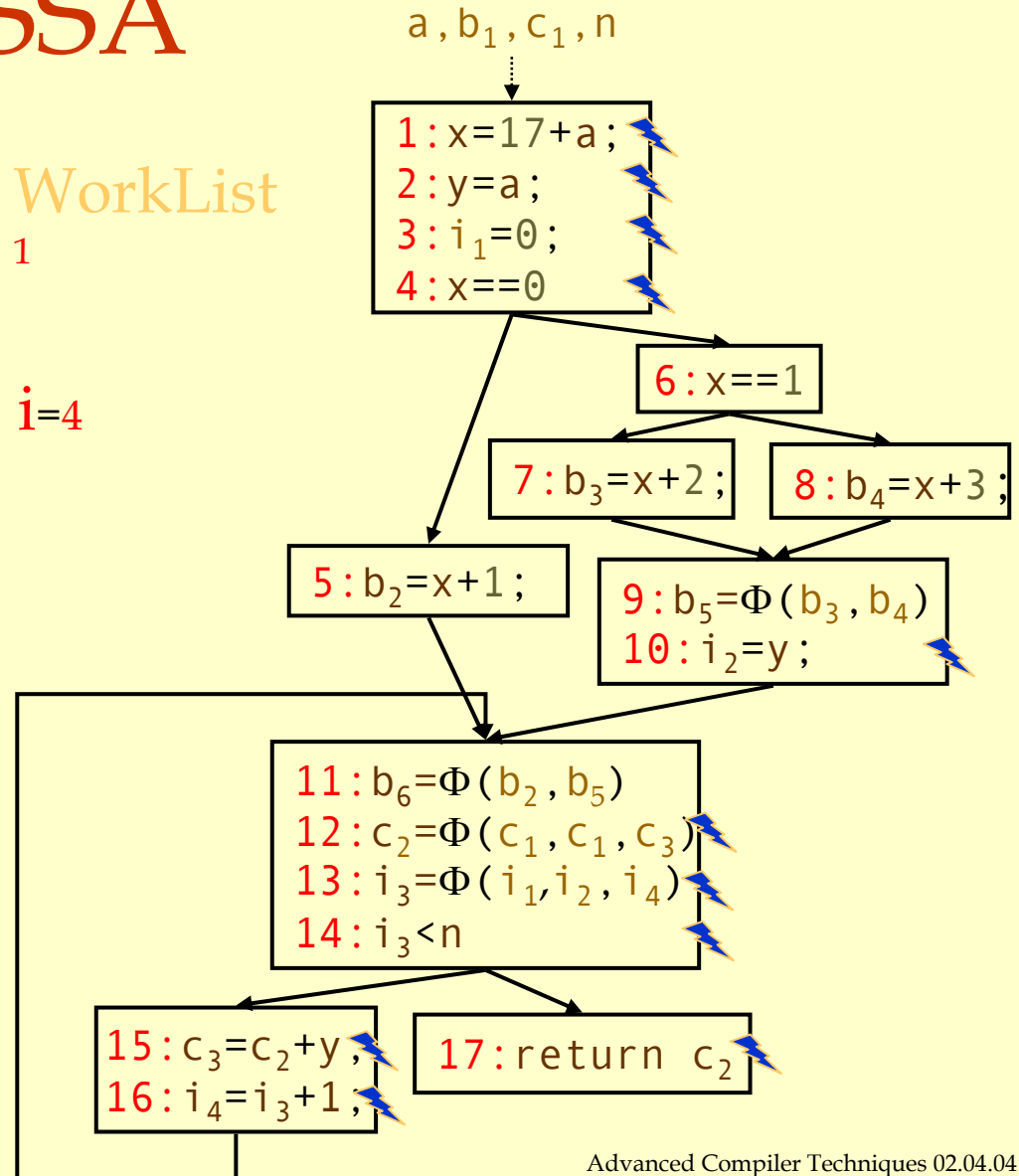
for each op i
 clear i 's mark
 if i is critical then
 mark i
 add i to **WorkList**

while (**Worklist** $\neq \emptyset$)
 remove i from **WorkList**
 (i has form " $x \leftarrow y \text{ op } z$ ")
 if $\text{def}(y)$ is not marked then
 mark $\text{def}(y)$
 add $\text{def}(y)$ to **WorkList**
 if $\text{def}(z)$ is not marked then
 mark $\text{def}(z)$
 add $\text{def}(z)$ to **WorkList**
 for each $b \in \text{RDF}(\text{block}(i))$
 mark the block-ending
 branch in b
 add it to **WorkList**

SSA

WorkList
1

$i=4$



Dead Code Elimination Using SSA

Mark

for each op i
 clear i 's mark
 if i is critical then
 mark i
 add i to WorkList

while (Worklist $\neq \emptyset$)

remove i from WorkList
 (i has form " $x \leftarrow y \text{ op } z$ ")

if $\text{def}(y)$ is not marked then
 mark $\text{def}(y)$

add $\text{def}(y)$ to WorkList

if $\text{def}(z)$ is not marked then
 mark $\text{def}(z)$

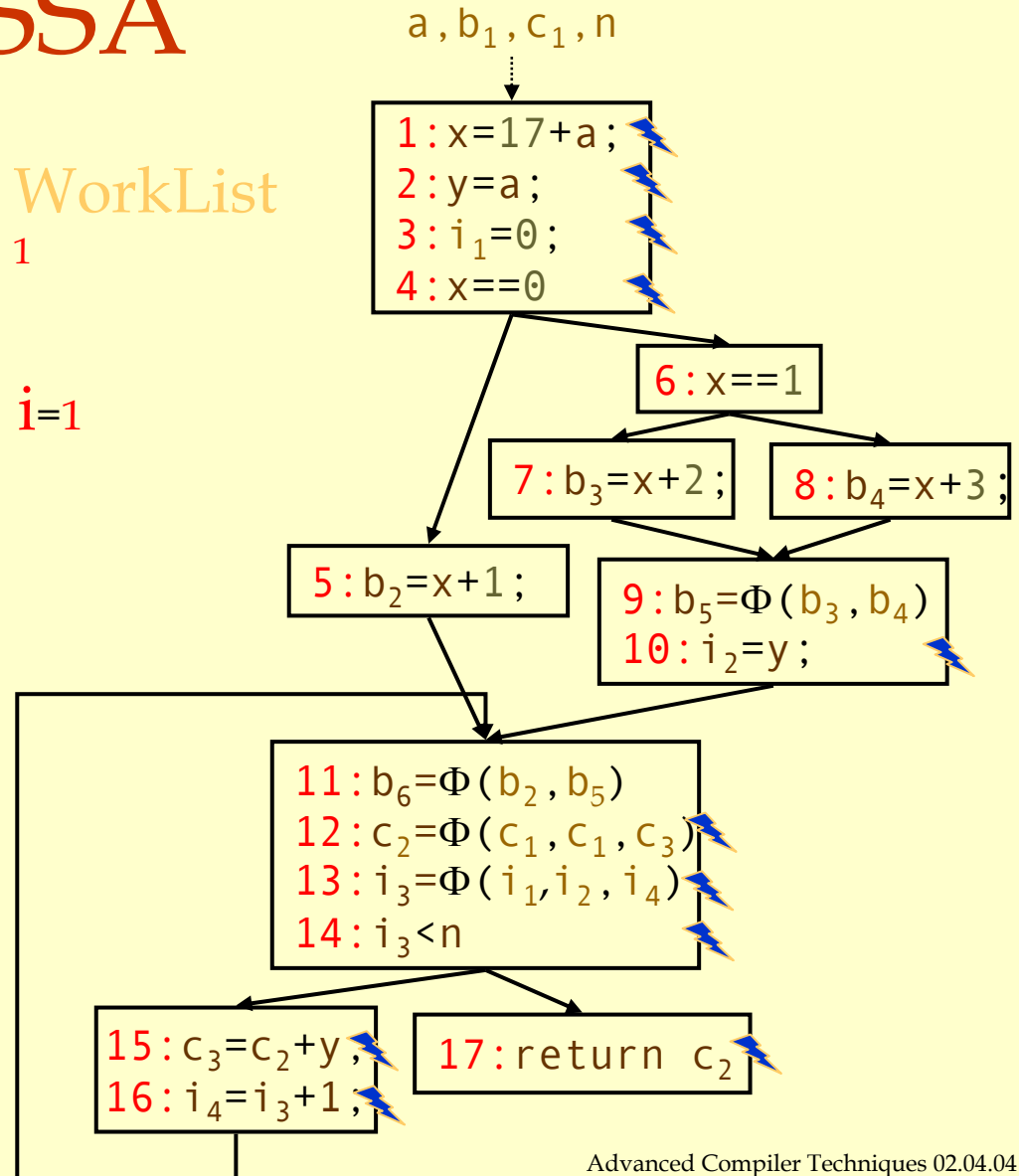
add $\text{def}(z)$ to WorkList

for each $b \in \text{RDF}(\text{block}(i))$
 mark the block-ending
 branch in b
 add it to WorkList

SSA

WorkList
1

$i=1$



Dead Code Elimination Using SSA

Mark

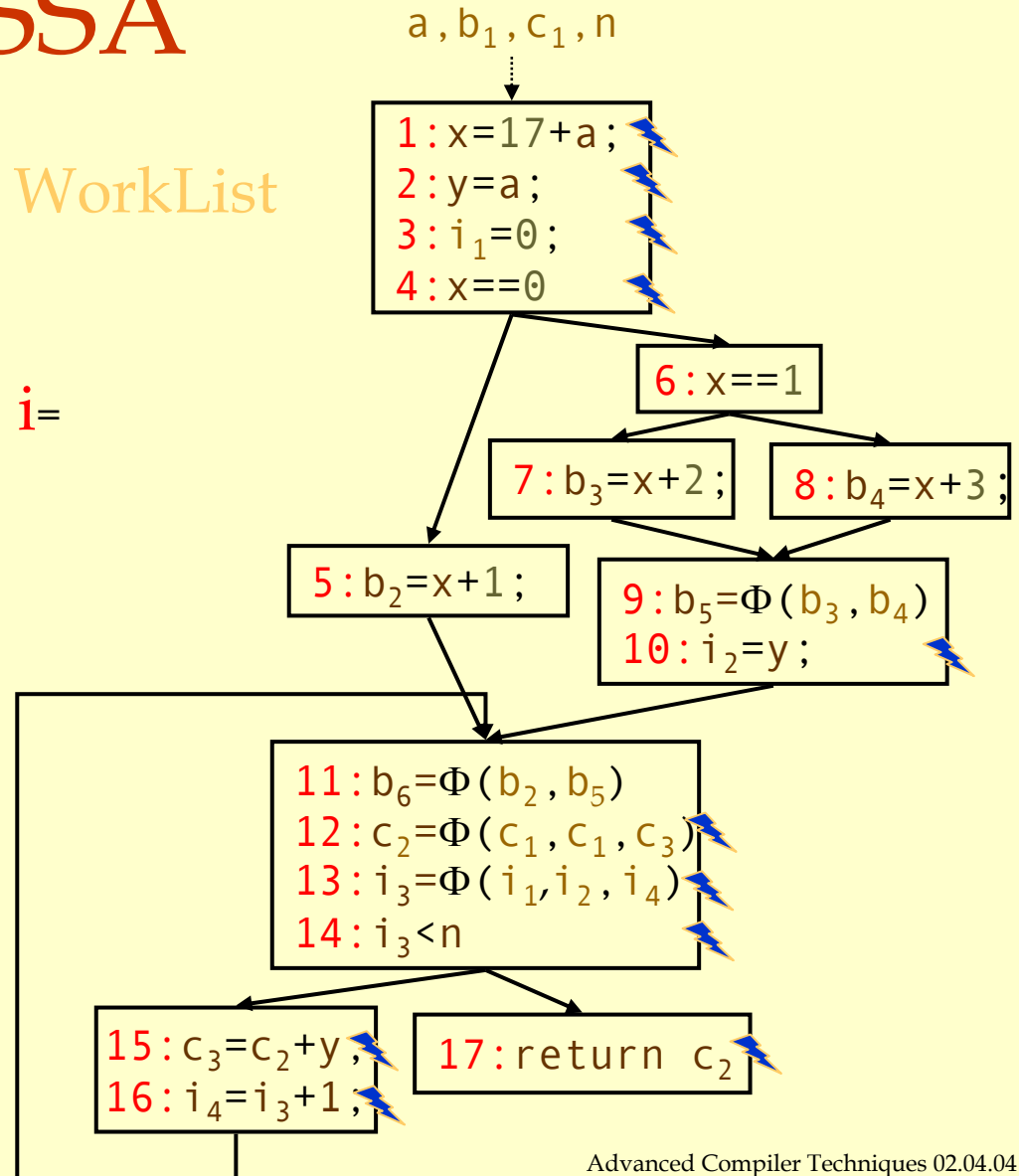
for each op i
clear i 's mark
if i is critical then
mark i
add i to **WorkList**

while (**Worklist** $\neq \emptyset$)
remove i from **WorkList**
(i has form " $x \leftarrow y \text{ op } z$ ")
if $\text{def}(y)$ is not marked then
mark $\text{def}(y)$
add $\text{def}(y)$ to **WorkList**
if $\text{def}(z)$ is not marked then
mark $\text{def}(z)$
add $\text{def}(z)$ to **WorkList**
for each $b \in \text{RDF}(\text{block}(i))$
mark the block-ending
branch in b
add it to **WorkList**

SSA

WorkList

$i =$

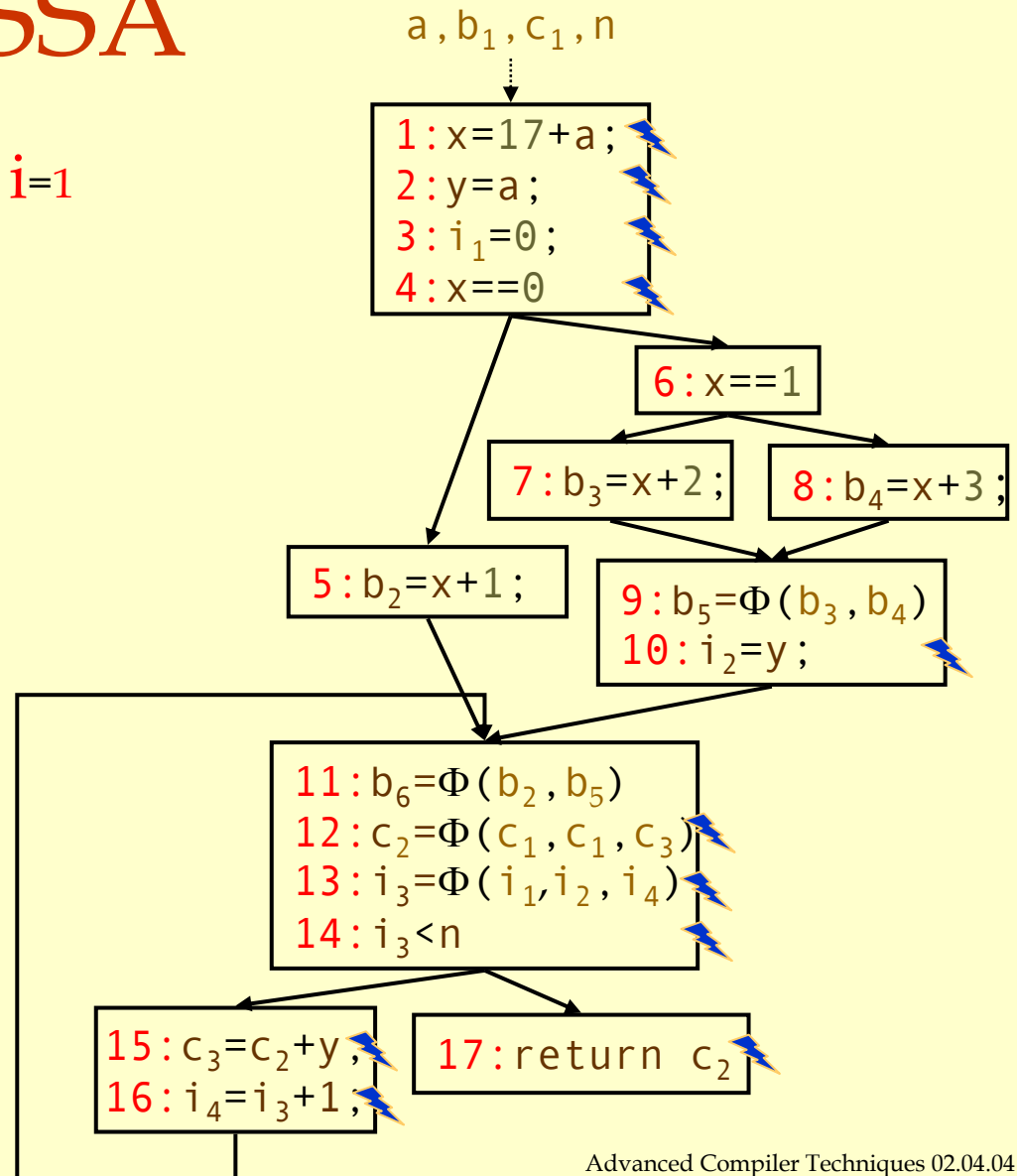


Dead Code Elimination Using SSA

Sweep

for each op i
if i is not marked then
if i is a branch then
rewrite with a jump to
 i 's nearest useful
post-dominator
if i is not a jump then
delete i

$i=1$



Dead Code Elimination Using SSA

Sweep

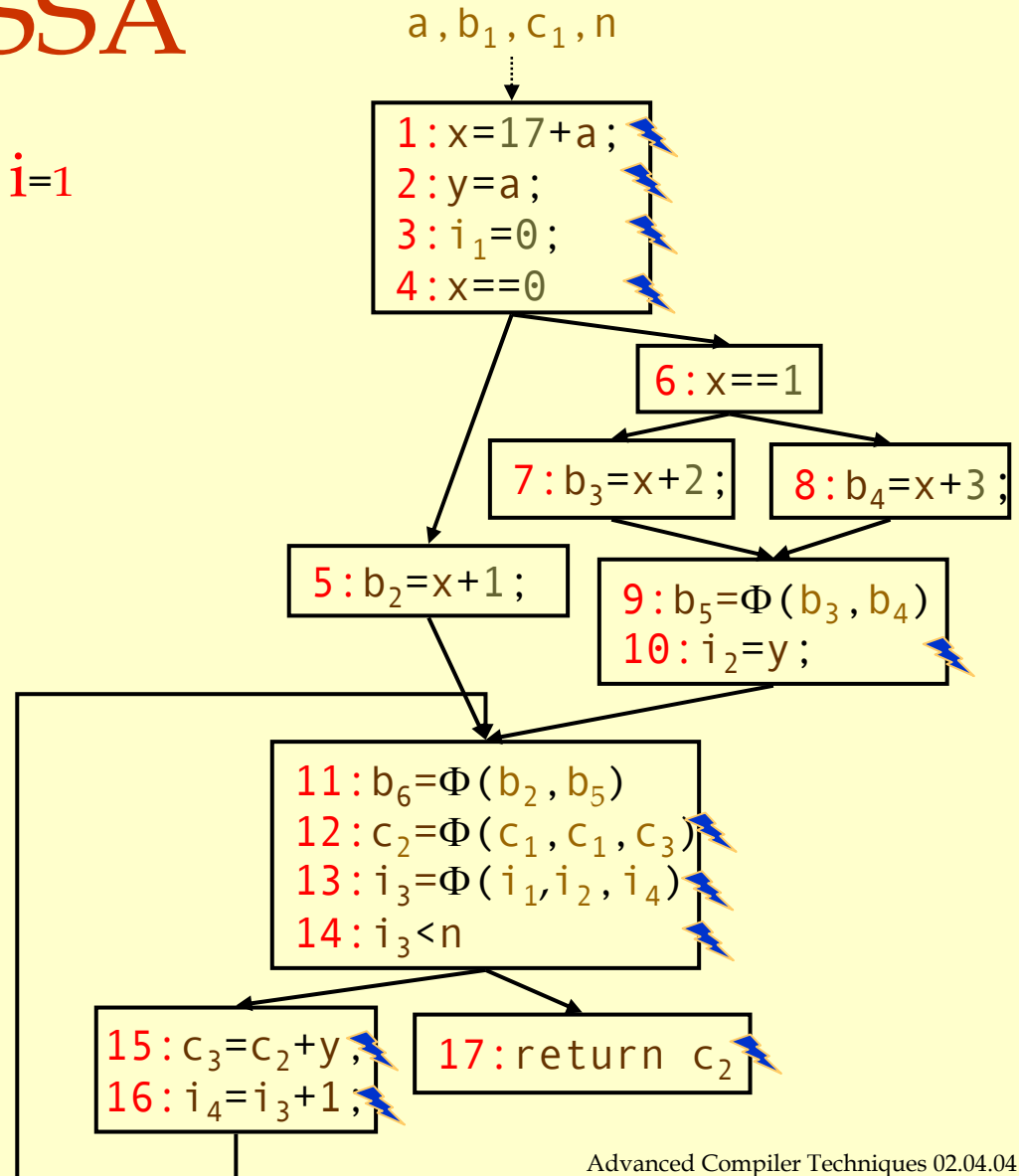
for each op i

if i is not marked then

if i is a branch then
rewrite with a jump to
 i 's nearest useful
post-dominator

if i is not a jump then
delete i

$i=1$



Dead Code Elimination Using SSA

Sweep

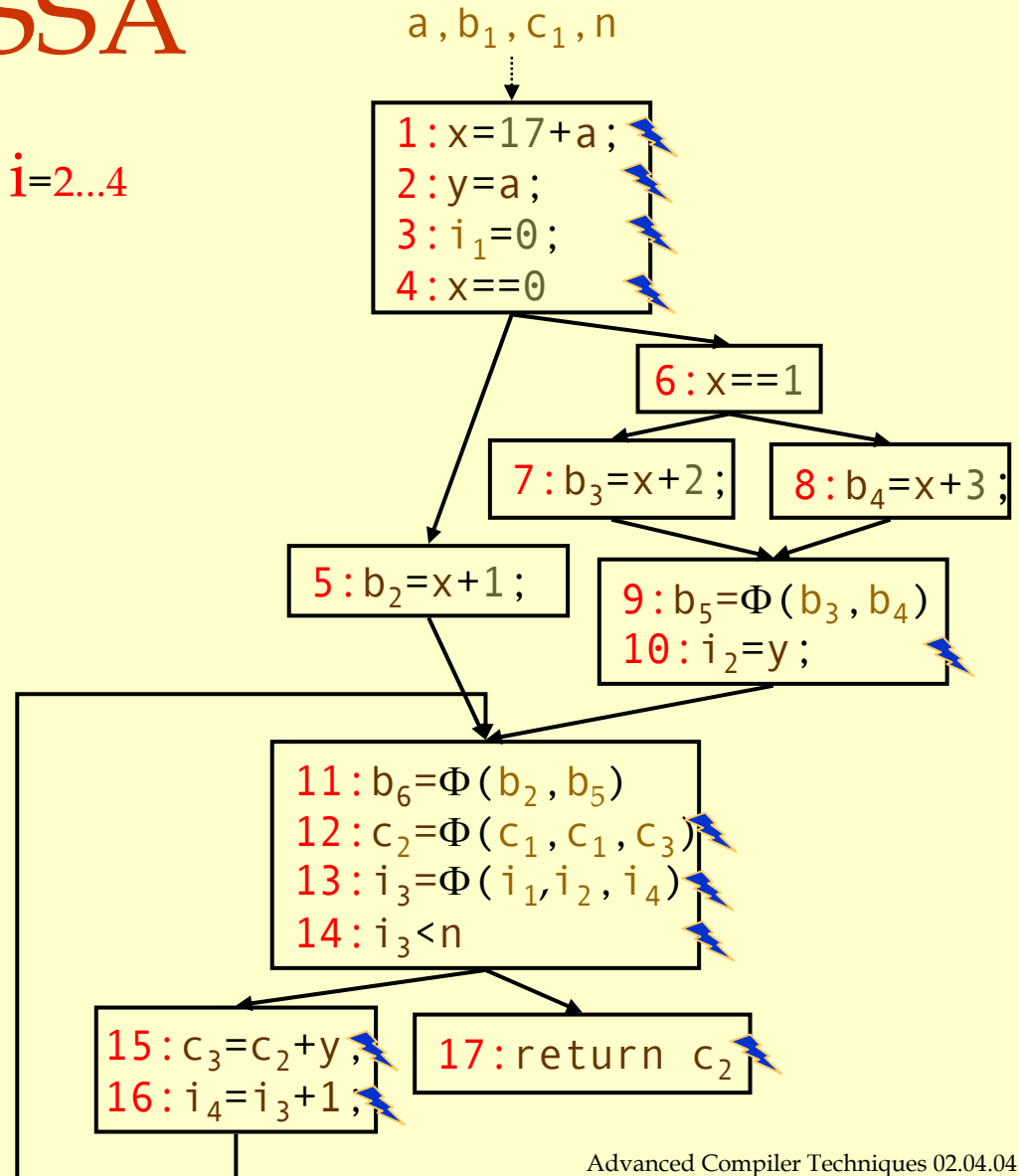
for each op i

if i is not marked then

if i is a branch then
rewrite with a jump to
 i 's nearest useful
post-dominator

if i is not a jump then
delete i

$i=2..4$



Dead Code Elimination Using SSA

Sweep

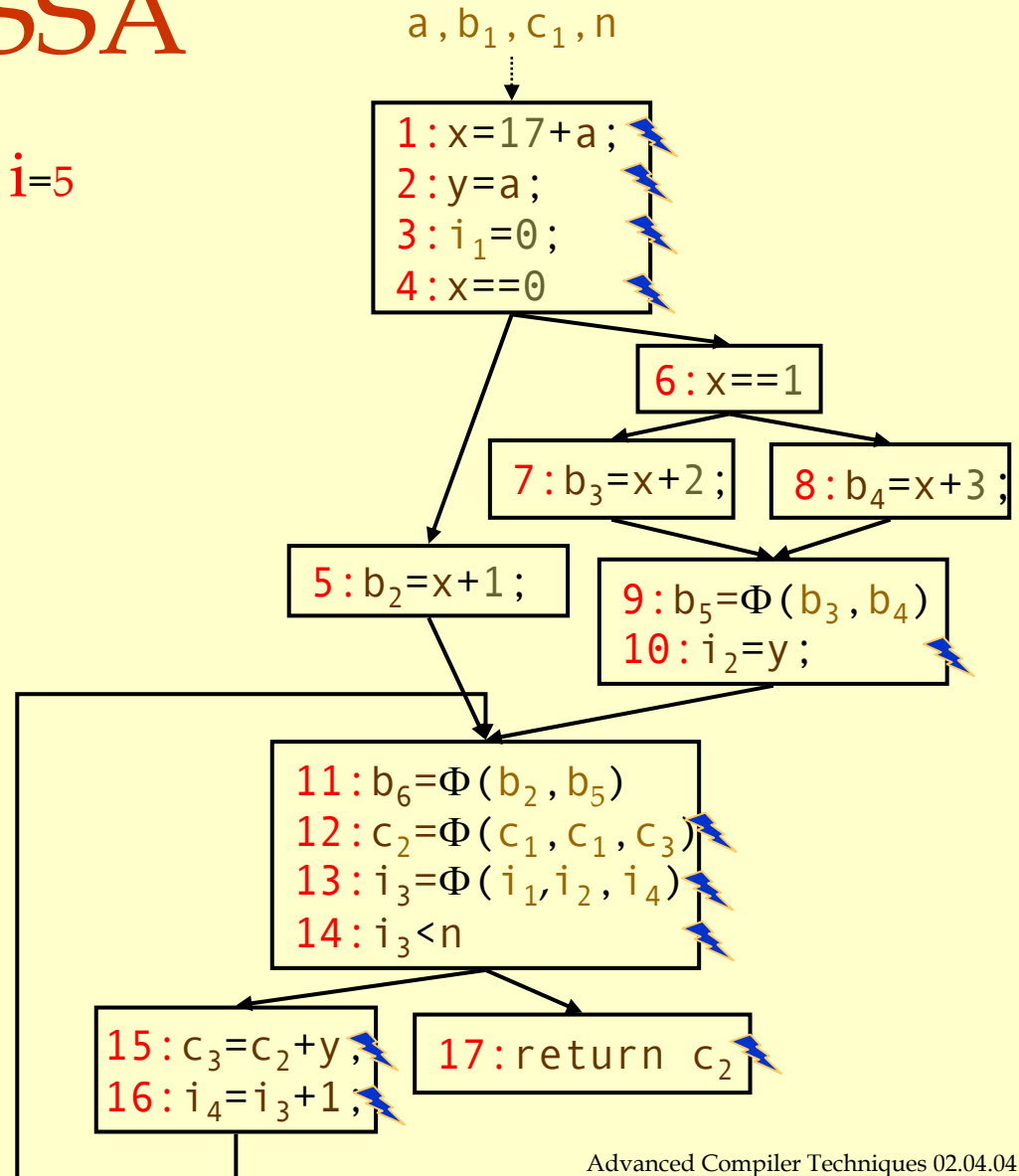
for each op i

if i is not marked then

if i is a branch then
rewrite with a jump to
 i 's nearest useful
post-dominator

if i is not a jump then
delete i

$i=5$



Dead Code Elimination Using SSA

Sweep

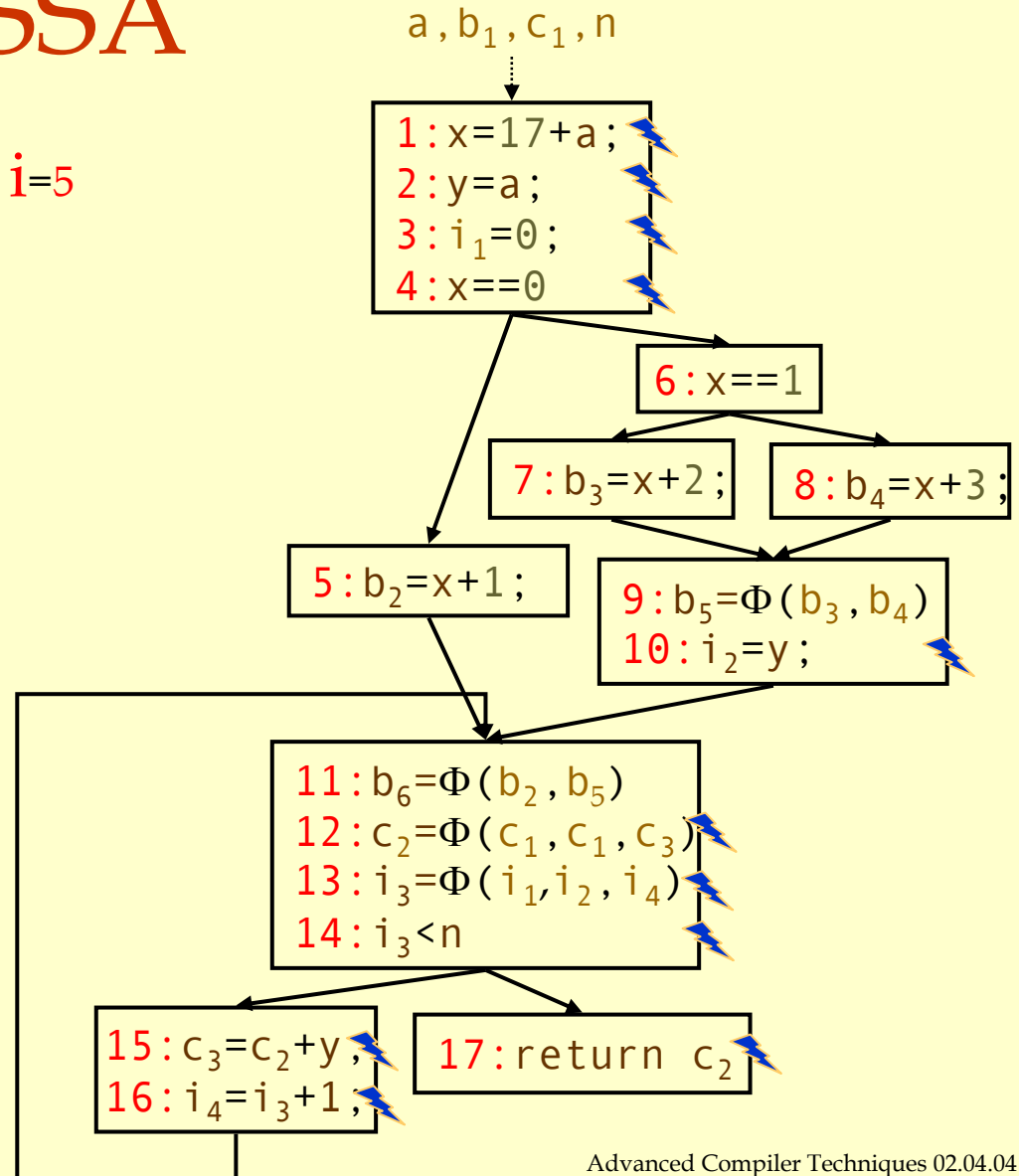
for each op i
if i is not marked then

if i is a branch then

rewrite with a jump to
 i 's nearest useful
post-dominator

if i is not a jump then
delete i

$i=5$



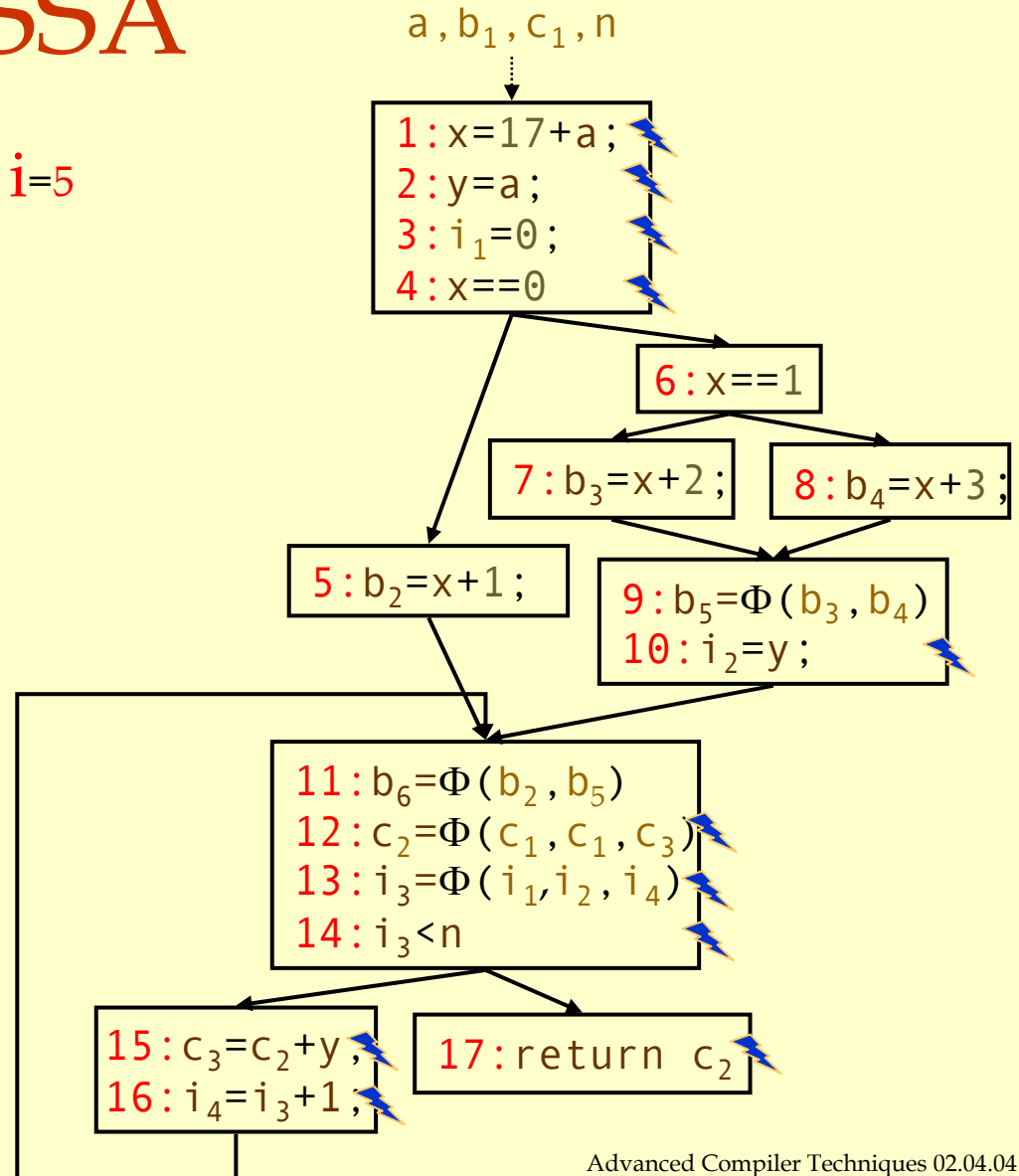
Dead Code Elimination Using SSA

Sweep

for each op i
if i is not marked then
if i is a branch then
rewrite with a jump to
 i 's nearest useful
post-dominator

if i is not a jump then
delete i

$i=5$



Dead Code Elimination Using SSA

Sweep

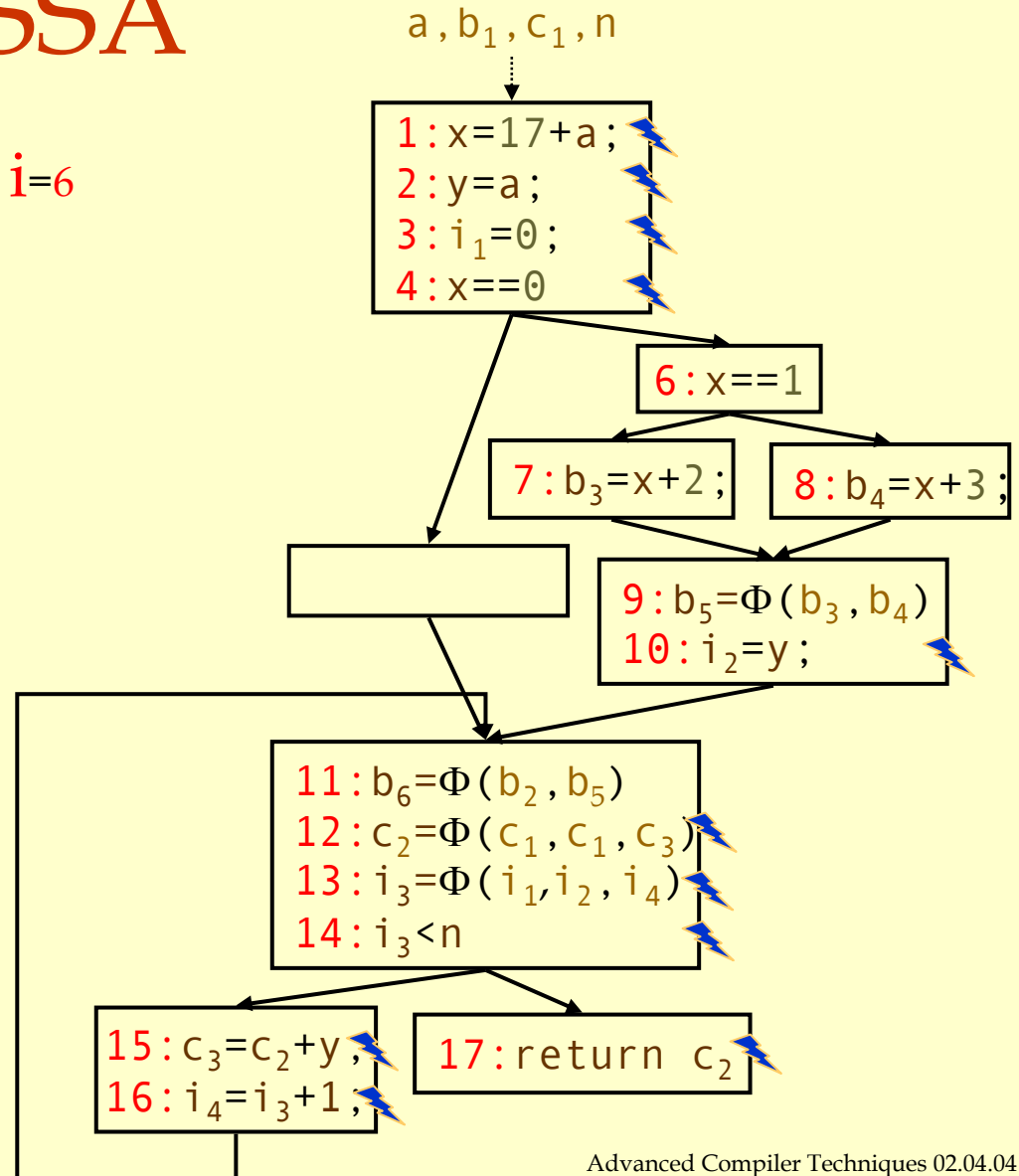
for each op i

if i is not marked then

if i is a branch then
rewrite with a jump to
 i 's nearest useful
post-dominator

if i is not a jump then
delete i

$i=6$

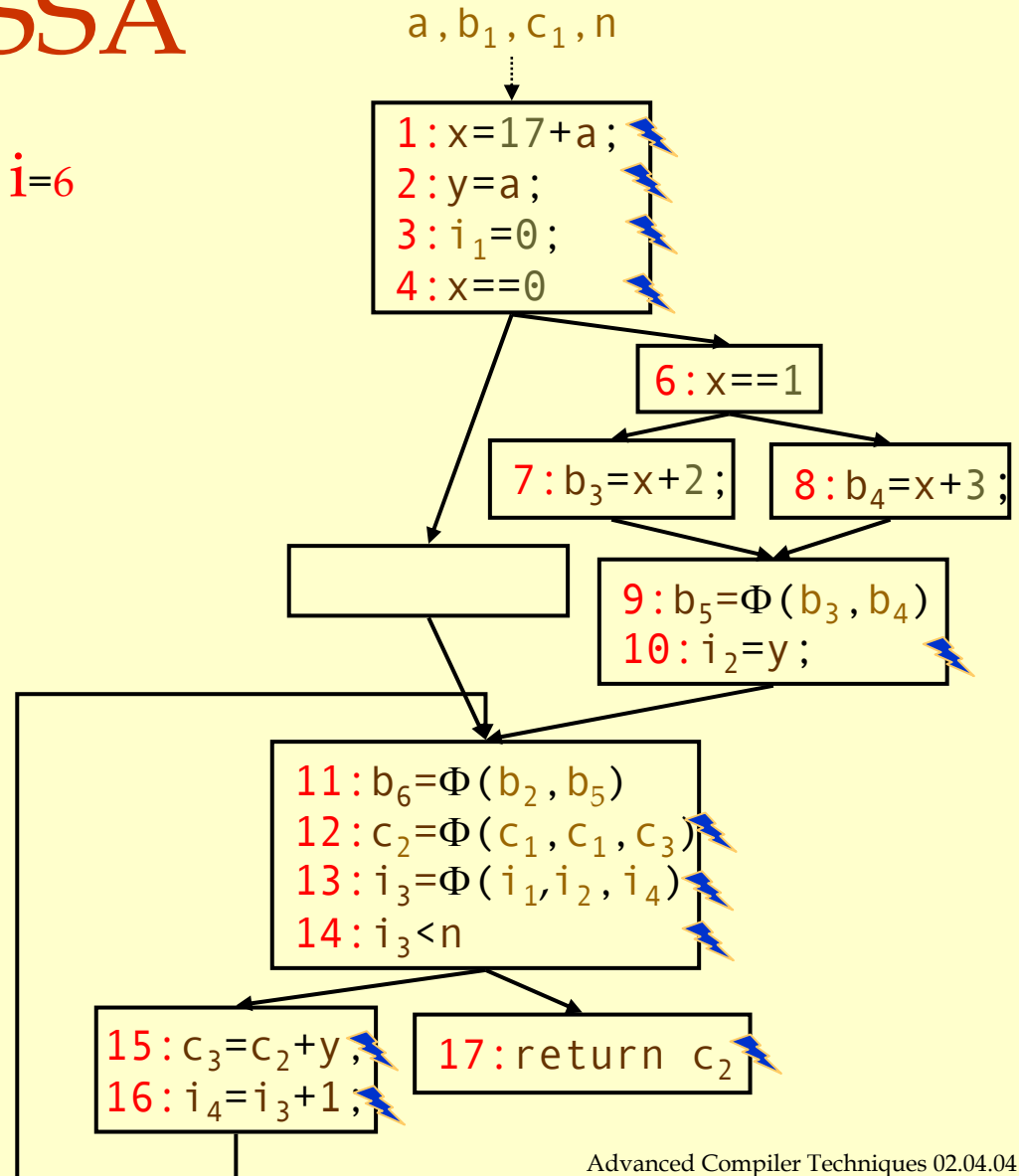


Dead Code Elimination Using SSA

Sweep

for each op i
 if i is not marked then
 if i is a branch then
 rewrite with a jump to
 i 's nearest useful
 post-dominator
 if i is not a jump then
 delete i

$i=6$



Dead Code Elimination Using SSA

Sweep

for each op i

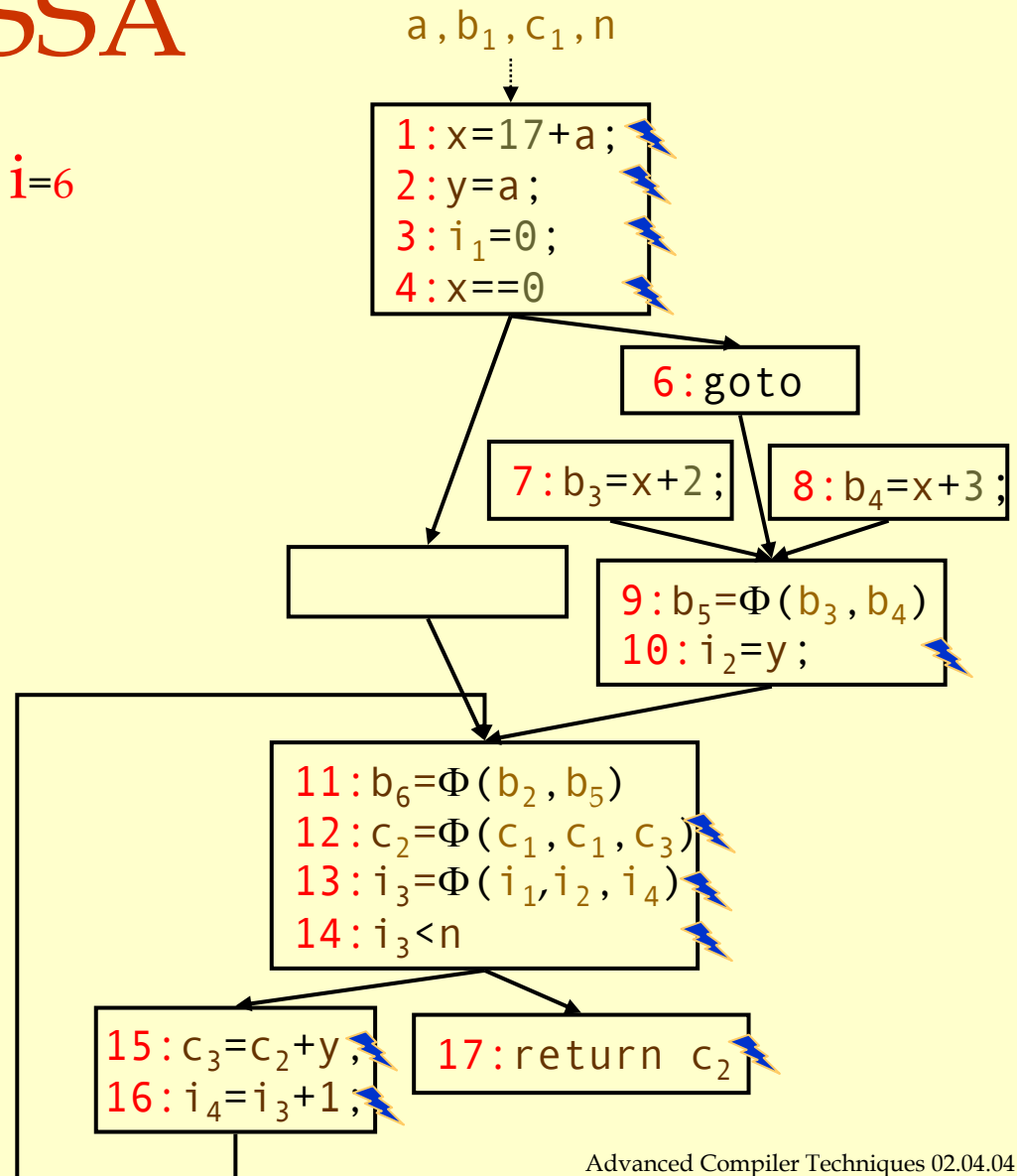
if i is not marked then

if i is a branch then

rewrite with a jump to
 i 's nearest useful
post-dominator

if i is not a jump then
delete i

$i=6$



Dead Code Elimination Using SSA

Sweep

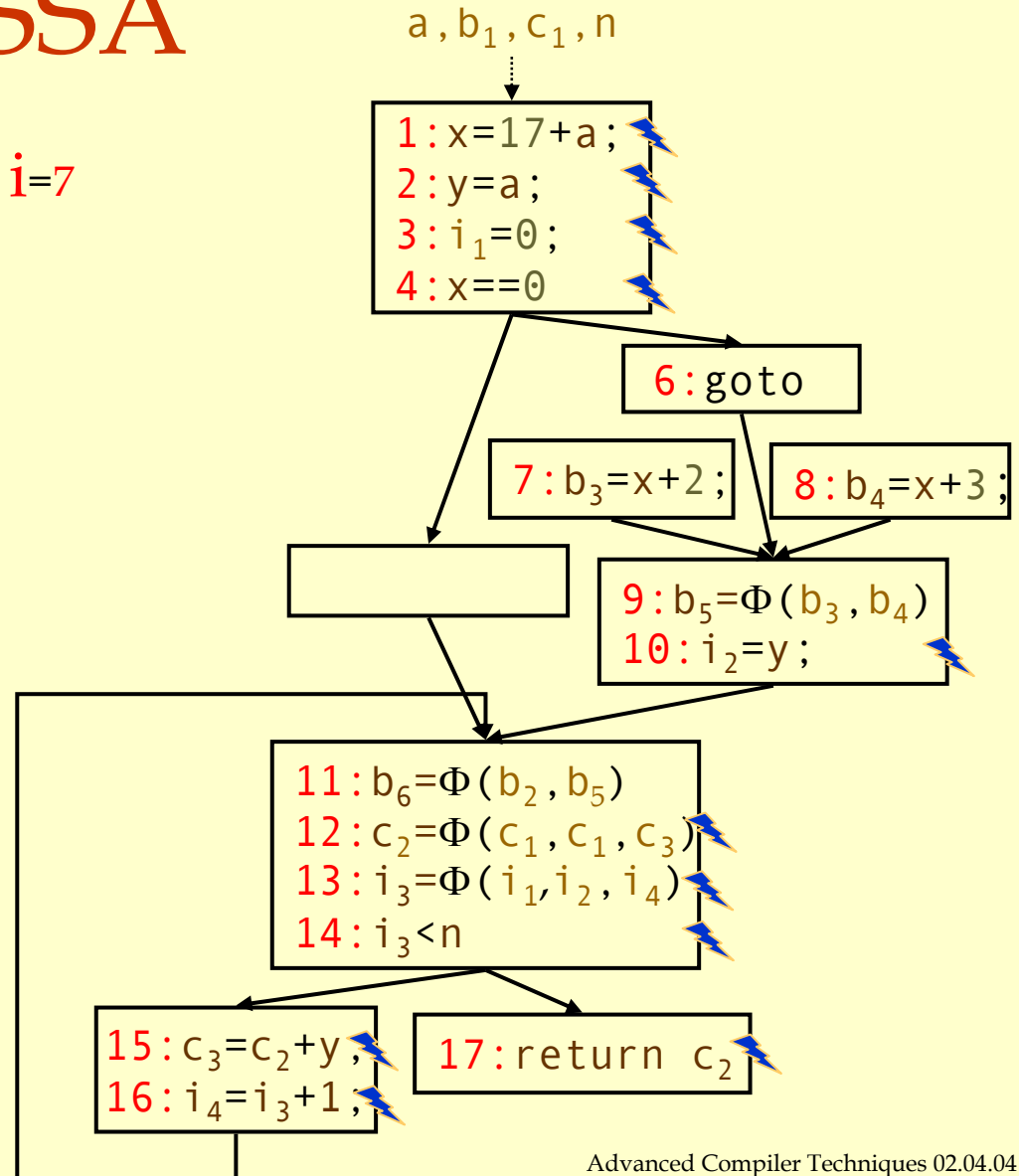
for each op i

if i is not marked then

if i is a branch then
rewrite with a jump to
 i 's nearest useful
post-dominator

if i is not a jump then
delete i

$i=7$



Dead Code Elimination Using SSA

Sweep

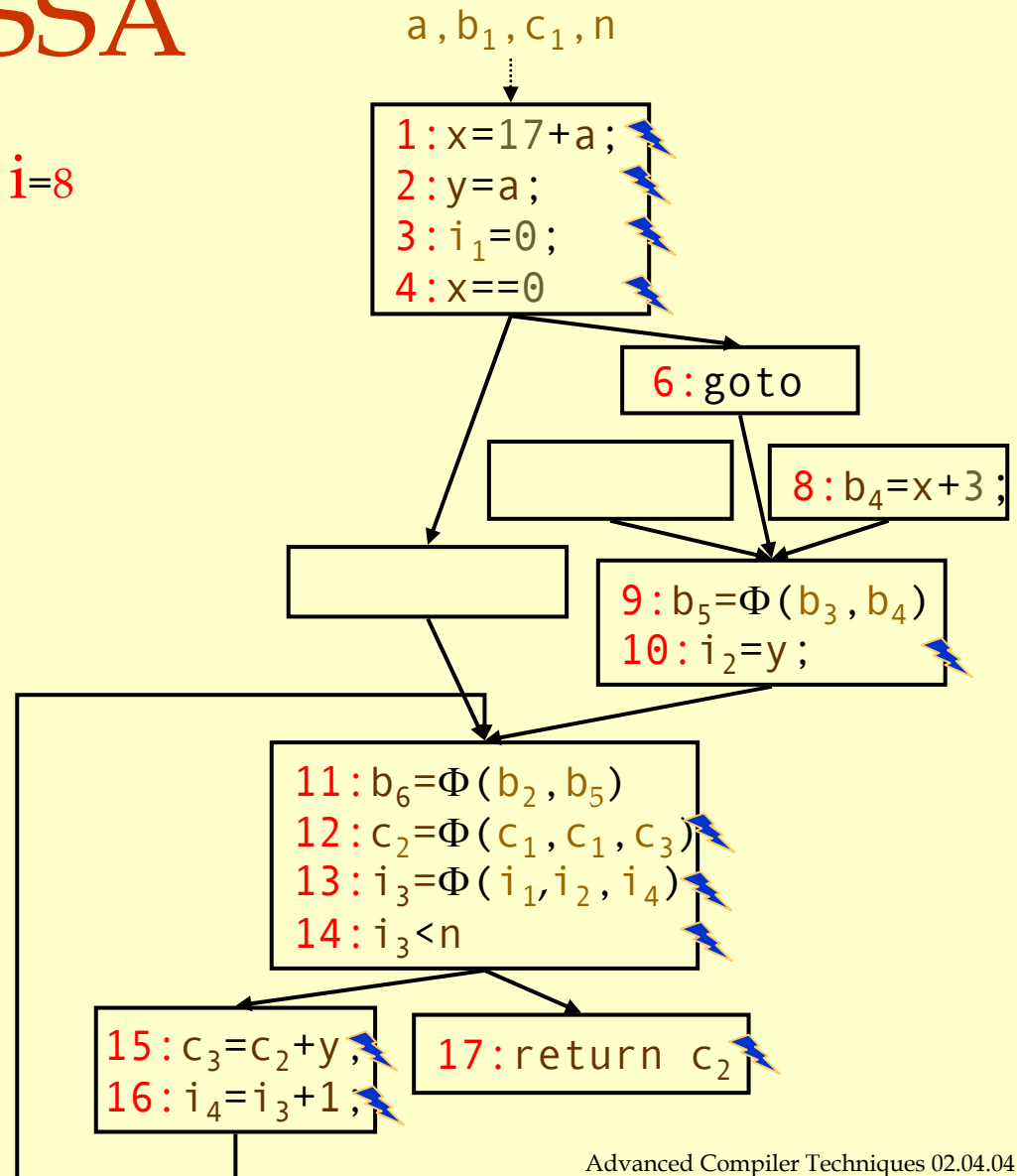
for each op i

if i is not marked then

if i is a branch then
rewrite with a jump to
 i 's nearest useful
post-dominator

if i is not a jump then
delete i

$i=8$



Dead Code Elimination Using SSA

Sweep

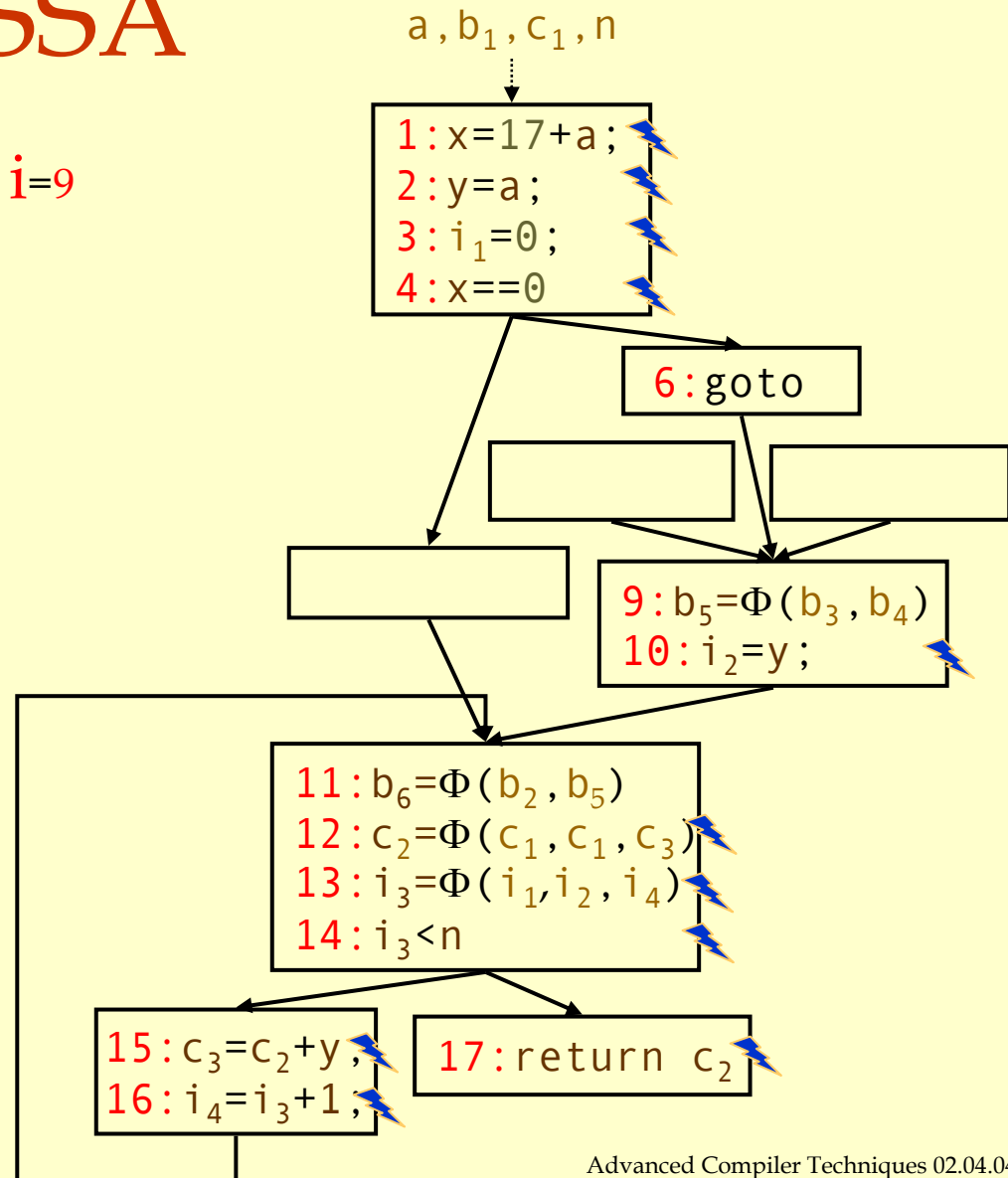
for each op i

if i is not marked then

if i is a branch then
rewrite with a jump to
 i 's nearest useful
post-dominator

if i is not a jump then
delete i

$i=9$



Dead Code Elimination Using SSA

Sweep

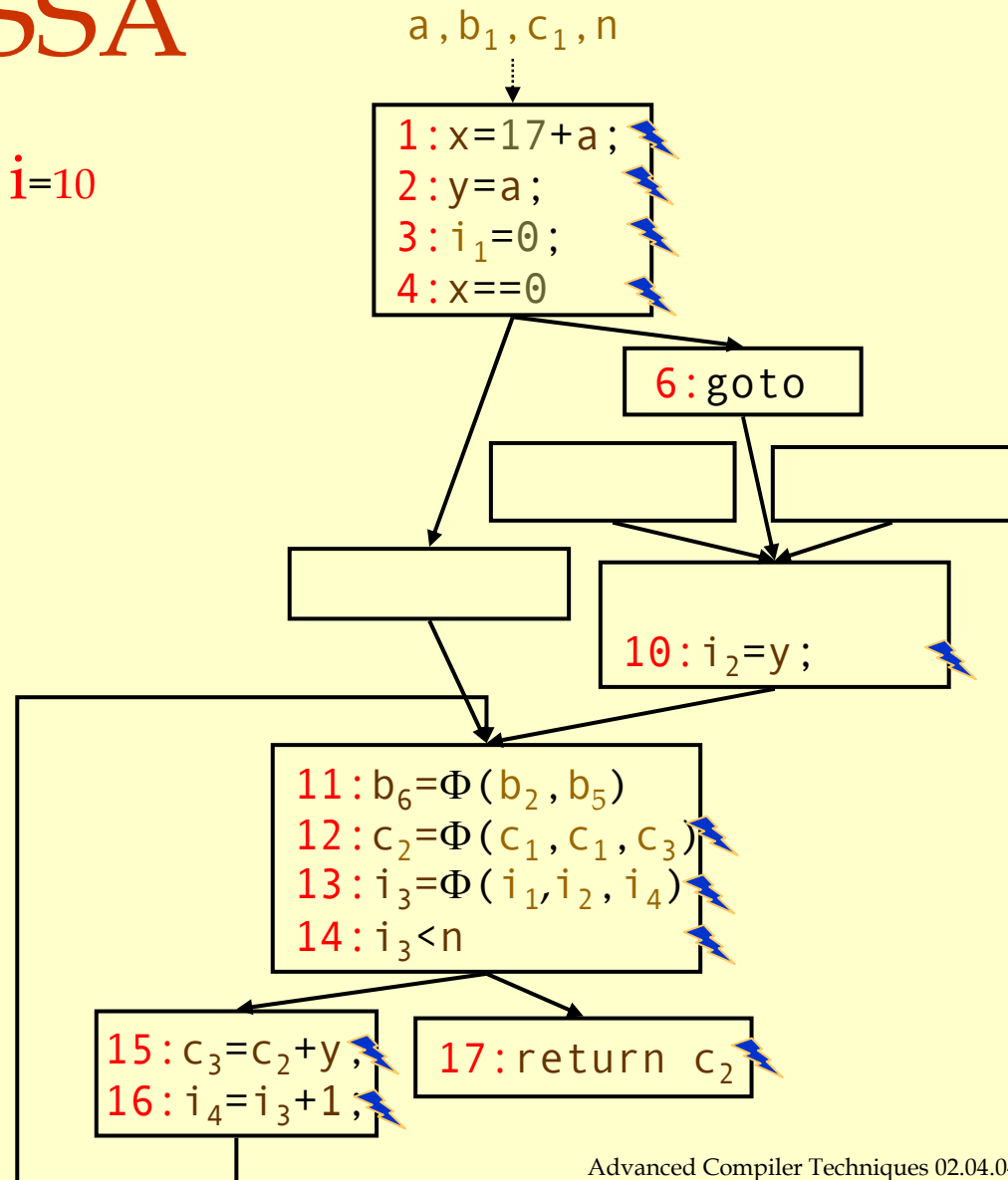
for each op i

if i is not marked then

if i is a branch then
rewrite with a jump to
 i 's nearest useful
post-dominator

if i is not a jump then
delete i

$i=10$



Dead Code Elimination Using SSA

Sweep

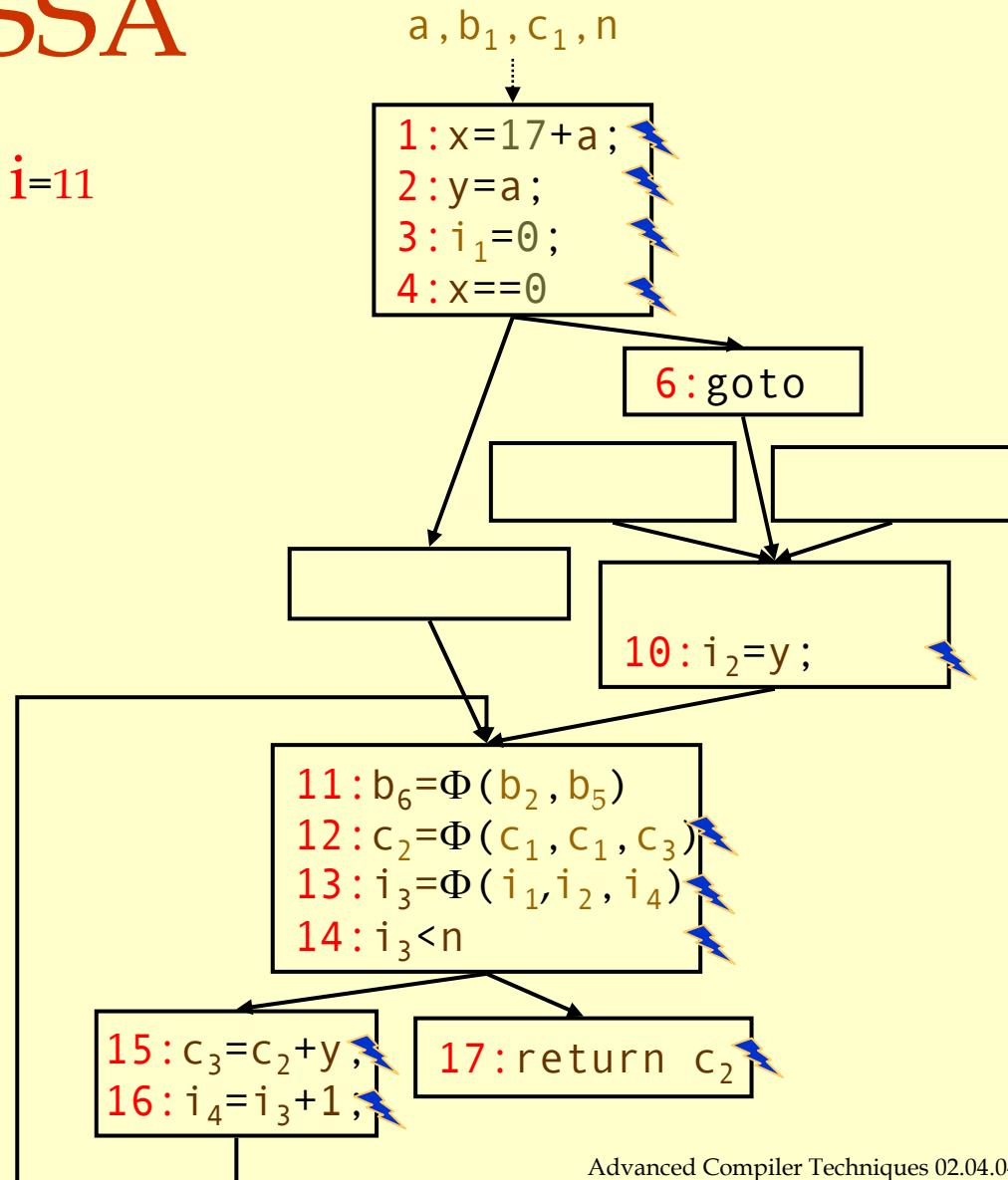
for each op i

if i is not marked then

if i is a branch then
rewrite with a jump to
 i 's nearest useful
post-dominator

if i is not a jump then
delete i

$i=11$



Dead Code Elimination Using SSA

Sweep

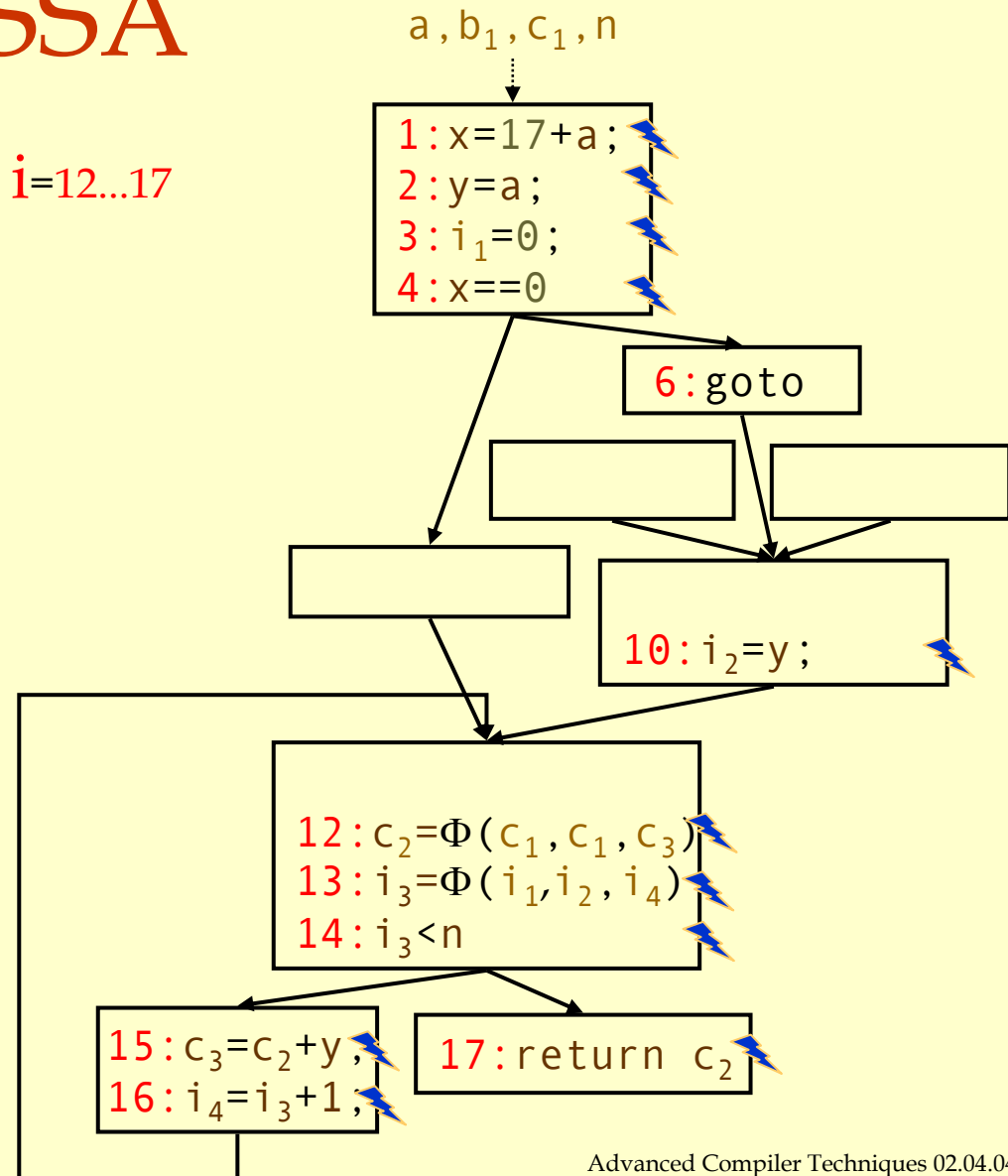
for each op i

if i is not marked then

if i is a branch then
rewrite with a jump to
 i 's nearest useful
post-dominator

if i is not a jump then
delete i

$i=12..17$



Dead Code Elimination Using SSA

What's left?

- ◆ Algorithm eliminates useless definitions & some useless branches
- ◆ Algorithm leaves behind empty blocks & extraneous control-flow

Algorithm from: Cytron, Ferrante, Rosen, Wegman, & Zadeck, *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*, ACM TOPLAS 13(4), October 1991

with a correction due to Rob Shillner

Two more issues

- ◆ Simplifying control-flow
- ◆ Eliminating unreachable blocks

Both are CFG transformations (no need for SSA)

Constant Propagation

Safety

- ◆ Proves that name always has known value
- ◆ Specializes code around that value
 - ◆ Moves some computations to compile time (\Rightarrow *code motion*)
 - ◆ Exposes some unreachable blocks (\Rightarrow *dead code*)

Opportunity

- ◆ Value $\neq \perp$ signifies an opportunity

Profitability

- ◆ Compile-time evaluation is cheaper than run-time evaluation
- ◆ Branch removal may lead to block coalescing
 - ◆ If not, it still avoids the test & makes branch predictable

Sparse Constant Propagation Using SSA

\forall expression, e

$\text{Value}(e) \leftarrow$	}	TOP if its value is unknown
$\text{WorkList} \leftarrow \emptyset$		c_i if its value is known (the constant c_i)
		BOT if its value is known to vary

\forall SSA edge $s = \langle u, v \rangle$
 if $\text{Value}(u) \neq \text{TOP}$ then
 add s to **WorkList**

i.e., o is “ $a \leftarrow b \text{ op } v$ ” or “ $a \leftarrow v \text{ op } b$ ”

while (**WorkList** $\neq \emptyset$)
 remove $s = \langle u, v \rangle$ from **WorkList**,
 let o be the operation that uses v
 if $\text{Value}(o) \neq \text{BOT}$ then
 $t \leftarrow$ result of evaluating o
 if $t \neq \text{Value}(o)$ then
 \forall SSA edge $\langle o, x \rangle$
 add $\langle o, x \rangle$ to **WorkList**

Same result, fewer \wedge operations
 Performs \wedge only at Φ nodes

Evaluating a Φ -node:

$\Phi(x_1, x_2, x_3, \dots, x_n)$ is
 $\text{Value}(x_1) \wedge \text{Value}(x_2) \wedge \text{Value}(x_3)$
 $\wedge \dots \wedge \text{Value}(x_n)$

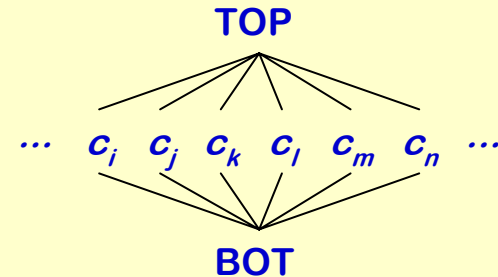
Where

TOP $\wedge x = x$	$\forall x$
$c_i \wedge c_j = c_j$	if $c_i = c_j$
$c_i \wedge c_j = \text{BOT}$	if $c_i \neq c_j$
BOT $\wedge x = \text{BOT}$	$\forall x$

Sparse Constant Propagation Using SSA

How long does this algorithm take to halt?

- ◆ Initialization is two passes
 - ◆ $|ops| + 2 \times |ops|$ edges
- ◆ Value(x) can take on 3 values
 - ◆ TOP, c_i , BOT
 - ◆ Each use can be on **WorkList** twice
 - ◆ $2 \times |args| = 4 \times |ops|$ evaluations, **WorkList** pushes & pops



This is an optimistic algorithm:

- ◆ Initialize all values to TOP, unless they are known constants
- ◆ Every value becomes BOT or c_i , unless its use is uninitialized

Sparse Conditional Constant Propagation

Optimism

```
 $i_0 \leftarrow 12$   
while ( ... )  
   $i_1 \leftarrow \Phi(i_0, i_3)$   
   $x \leftarrow i_1 * 17$   
   $j \leftarrow i_1$   
   $i_2 \leftarrow \dots$   
  ...  
   $i_3 \leftarrow j$ 
```

Optimism

- This version of the algorithm is an optimistic formulation
- Initializes values to **TOP**
- Prior version used \perp (*implicit*)

Sparse Conditional Constant Propagation

Optimism

```

 $i_0 \leftarrow 12$ 
while ( ... )
   $i_1 \leftarrow \Phi(i_0, i_3)$ 
   $x \leftarrow i_1 * 17$ 
   $j \leftarrow i_1$ 
   $i_2 \leftarrow \dots$ 
  ...
   $i_3 \leftarrow j$ 

```

Optimism

- This version of the algorithm is an *optimistic* formulation
- Initializes values to **TOP**
- Prior version used \perp (*implicit*)

Sparse Conditional Constant Propagation

Optimism

```

 $i_0 \leftarrow 12$ 
while ( ... )
   $i_1 \leftarrow \Phi(i_0, i_3)$ 
   $x \leftarrow i_1 * 17$ 
   $j \leftarrow i_1$ 
   $i_2 \leftarrow \dots$ 
  ...
   $i_3 \leftarrow j$ 

```

Clear
that i is
always
12 at
def of x

Optimism

- This version of the algorithm is an optimistic formulation
- Initializes values to **TOP**
- Prior version used \perp (*implicit*)

Sparse Conditional Constant Propagation

Optimism

```

12   $i_\theta \leftarrow 12$ 
    while ( ... )
       $\perp i_1 \leftarrow \Phi(i_\theta, i_3)$ 
       $\perp x \leftarrow i_1 * 17$ 
       $\perp j \leftarrow i_1$ 
       $\perp i_2 \leftarrow \dots$ 
      ...
       $\perp i_3 \leftarrow j$ 

```

**Pessimistic
initializations**

Leads to:

$$\begin{aligned}
 i_1 &\equiv 12 \wedge \perp \equiv \perp \\
 x &\equiv \perp * 17 \equiv \perp \\
 j &\equiv \perp \\
 i_3 &\equiv \perp
 \end{aligned}$$

Optimism

- This version of the algorithm is an *optimistic* formulation
- Initializes values to **TOP**
- Prior version used \perp (*implicit*)

Sparse Conditional Constant Propagation

Optimism

```

12   $i_0 \leftarrow 12$ 
    while ( ... )
      TOP  $i_1 \leftarrow \Phi(i_0, i_3)$ 
      TOP  $x \leftarrow i_1 * 17$ 
      TOP  $j \leftarrow i_1$ 
      TOP  $i_2 \leftarrow \dots$ 
      ...
      TOP  $i_3 \leftarrow j$ 

```

Optimistic initializations

Leads to:

$$i_1 \equiv 12 \wedge \text{TOP} \equiv 12$$

$$x \equiv 12 * 17 \equiv 204$$

$$j \equiv 12$$

$$i_3 \equiv 12$$

$$i_1 \equiv 12 \wedge 12 \equiv 12$$

Optimism

- This version of the algorithm is an *optimistic* formulation
- Initializes values to **TOP**
- Prior version used \perp (*implicit*)

In general, optimism helps inside loops.

M.N. Wegman & F.K. Zadeck, Constant propagation with conditional branches, ACM TOPLAS, 13(2), April 1991, pages 181–210.

Sparse Conditional Constant Propagation

What happens when it propagates a value into a branch?

- ◆ **TOP** \Rightarrow we gain no knowledge.
- ◆ **BOT** \Rightarrow either path can execute.
- ◆ **TRUE** or **FALSE** \Rightarrow only one path can execute.



But, the algorithm does not use this ...

Working this into the algorithm.

- ◆ Use two worklists: **SSAWorkList** & **CFGWorkList**:
 - ◆ **SSAWorkList** determines values.
 - ◆ **CFGWorkList** governs reachability.
- ◆ Don't propagate into operation until its block is reachable.

Sparse Conditional Constant Propagation

SSAWorkList $\leftarrow \emptyset$

CFGWorkList $\leftarrow n_0$

\forall block **b**

clear **b**'s mark

\forall expression **e** in **b**

Value(e) \leftarrow TOP

Initialization Step

To evaluate a branch

if arg is **BOT** then

put both targets on **CFGWorklist**

else if arg is **TRUE** then

put TRUE target on **CFGWorkList**

else if arg is **FALSE** then

put FALSE target on **CFGWorkList**

To evaluate a jump

place its target on **CFGWorkList**

while (**CFGWorkList** \cup **SSAWorkList**) $\neq \emptyset$)

while(**CFGWorkList** $\neq \emptyset$)

remove **b** from **CFGWorkList**

mark **b**

evaluate each Φ -function in **b**

evaluate each op in **b**, *in order*

while(**SSAWorkList** $\neq \emptyset$)

remove **s** = **<u,v>** from **SSAWorkList**

let **o** be the operation that contains **v**

t \leftarrow result of evaluating **o**

if **t** \neq **Value(o)** then

Value(o) \leftarrow **t**

\forall SSA edge **<o,x>**

if **x** is marked, then

add **<o,x>** to **SSAWorkList**

Propagation Step

Sparse Conditional Constant Propagation

There are some subtle points:

- ◆ Branch conditions should not be **TOP** when evaluated.
 - ◆ Indicates an upwards-exposed use. (*no initial value - undefined*)
 - ◆ Hard to envision compiler producing such code.
- ◆ Initialize all operations to **TOP**.
 - ◆ Block processing will fill in the non-top initial values.
 - ◆ Unreachable paths contribute **TOP** to Φ -functions.
- ◆ Code shows CFG edges first, then SSA edges.
 - ◆ Can intermix them in arbitrary order. (*correctness*)
 - ◆ Taking CFG edges first may help with speed. (*minor effect*)

Sparse Conditional Constant Propagation

More subtle points:

- ◆ $TOP * BOT \rightarrow TOP$
 - ◆ If TOP becomes 0 , then $0 * BOT \rightarrow 0$.
 - ◆ This prevents non-monotonic behavior for the result value.
 - ◆ Uses of the result value might go irretrievably to 0 .
 - ◆ Similar effects with any operation that has a “zero”.

- ◆ Some values reveal simplifications, rather than constants
 - ◆ $BOT * c_i \rightarrow BOT$, but might turn into shifts & adds ($c_i = 2, BOT \geq 0$)
 - ◆ Removes commutativity. *(reassociation)*
 - ◆ $BOT^{**}2 \rightarrow BOT * BOT$. *(vs. series or call to library)*

- ◆ $cbr \ TRUE \rightarrow L_1, L_2$ becomes $br \rightarrow L_1$
 - ◆ Method discovers this; it must rewrite the code, too!

Sparse Conditional Constant Propagation

Unreachable Code

```
i ← 17
if (i > 0) then
  j1 ← 10
else
  j2 ← 20
j3 ← Φ(j1, j2)
k ← j3 * 17
```

Optimism

- Initialization to **TOP** is still important.
- Unreachable code keeps **TOP**.
- \wedge with **TOP** has desired result.

Sparse Conditional Constant Propagation

Unreachable Code

```

17  i ← 17
    if (i > 0) then
10  j1 ← 10
    else
20  j2 ← 20
⊥  j3 ← Φ(j1, j2)
⊥  k ← j3 * 17
  
```

All paths execute

Optimism

- Initialization to **TOP** is still important.
- Unreachable code keeps **TOP**.
- \wedge with **TOP** has desired result.

Sparse Conditional Constant Propagation

Unreachable Code

```

17  i ← 17
    if (i > 0) then
TOP  j1 ← 10
    else
TOP  j2 ← 20
TOP  j3 ← Φ(j1, j2)
170 k ← j3 * 17

```

With SCC
marking
blocks

Optimism

- Initialization to **TOP** is still important.
- Unreachable code keeps **TOP**.
- \wedge with **TOP** has desired result.

Sparse Conditional Constant Propagation

Unreachable Code

```

17  i ← 17
    if (i > 0) then
10  j1 ← 10
    else
TOP  j2 ← 20
10  j3 ← Φ(j1, j2)
170 k ← j3 * 17

```

With SCC
marking
blocks

Optimism

- Initialization to **TOP** is still important.
- Unreachable code keeps **TOP**.
- \wedge with **TOP** has desired result.

Cannot get this any other way:

- DEAD code cannot test ($i > 0$).
- DEAD marks j_2 as useful.

Sparse Conditional Constant Propagation

Unreachable Code

```

17  i ← 17
    if (i > 0) then
10  j1 ← 10
    else
TOP  j2 ← 20
10  j3 ← Φ(j1, j2)
170 k ← j3 * 17
  
```

With SCC marking blocks

Optimism

- Initialization to **TOP** is still important.
- Unreachable code keeps **TOP**.
- \wedge with **TOP** has desired result.

In general, combining two optimizations can lead to answers that cannot be produced by any combination of running them separately.

This algorithm is one example of that general principle.

Combining register allocation & instruction scheduling is another ...

Using SSA Form for Optimizations

In general, using SSA conversion leads to:

- ◆ Cleaner formulations.
- ◆ Better results.
- ◆ Faster algorithms.

We've seen two SSA-based algorithms.

- ◆ Dead-code elimination.
- ◆ Sparse conditional constant propagation.

Partial Redundancy Elimination

This lecture is primarily based on Konstantinos Sagonas set of slides
(Advanced Compiler Techniques, (2AD518)
at Uppsala University, January-February 2004).
Used with kind permission.
(In turn based on Keith Cooper's slides)

Common-Subexpression Elimination

An occurrence of an expression in a program is a common subexpression if there is another occurrence of the expression whose evaluation always precedes this one in execution order and if the operands of the expression remain unchanged between the two evaluations.

Local Common Subexpression Elimination (CSE) keeps track of the set of *available expressions* within a basic block and replaces instances of them by references to new temporaries that keep their value.

```
...  
a = (x + y) + z ;  
b = a - 1 ;  
c = x + y ;  
...
```

Before CSE

```
...  
t = x + y ;  
a = t + z ;  
b = a - 1 ;  
c = t ;  
...
```

After CSE

Available Expressions

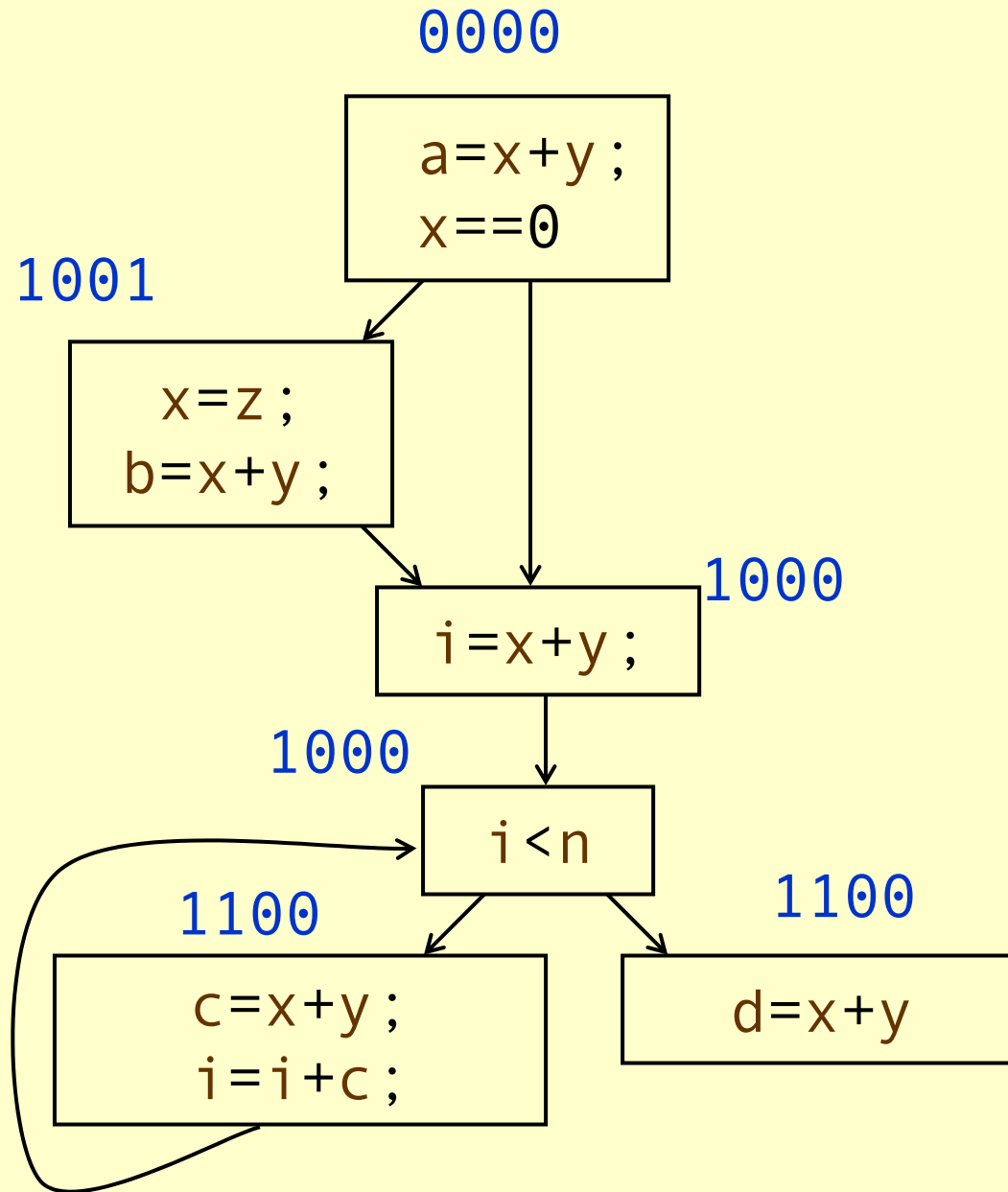
- ◆ An expression $x+y$ is available at a program point p if
 - ◆ every path from the initial node to p evaluates $x+y$ before reaching p ,
 - ◆ and there are no assignments to x or y after the evaluation but before p .
- ◆ Available Expression information can be used to do global (across basic blocks) CSE.
- ◆ If an expression is available at the point of its use, there is no need to re-evaluate it.

Computing Available Expressions

- ◆ Represent sets of expressions using bit vectors
- ◆ Each expression corresponds to a bit
- ◆ Run dataflow algorithm similar to reaching definitions
- ◆ Notice that:
 - ◆ A definition reaches a basic block if it comes from ANY predecessor in CFG.
 - ◆ An expression is available at a basic block only if it is available from ALL block's predecessors in the CFG.

Expressions

- 1: $x+y$
- 2: $i < n$
- 3: $i+c$
- 4: $x == 0$

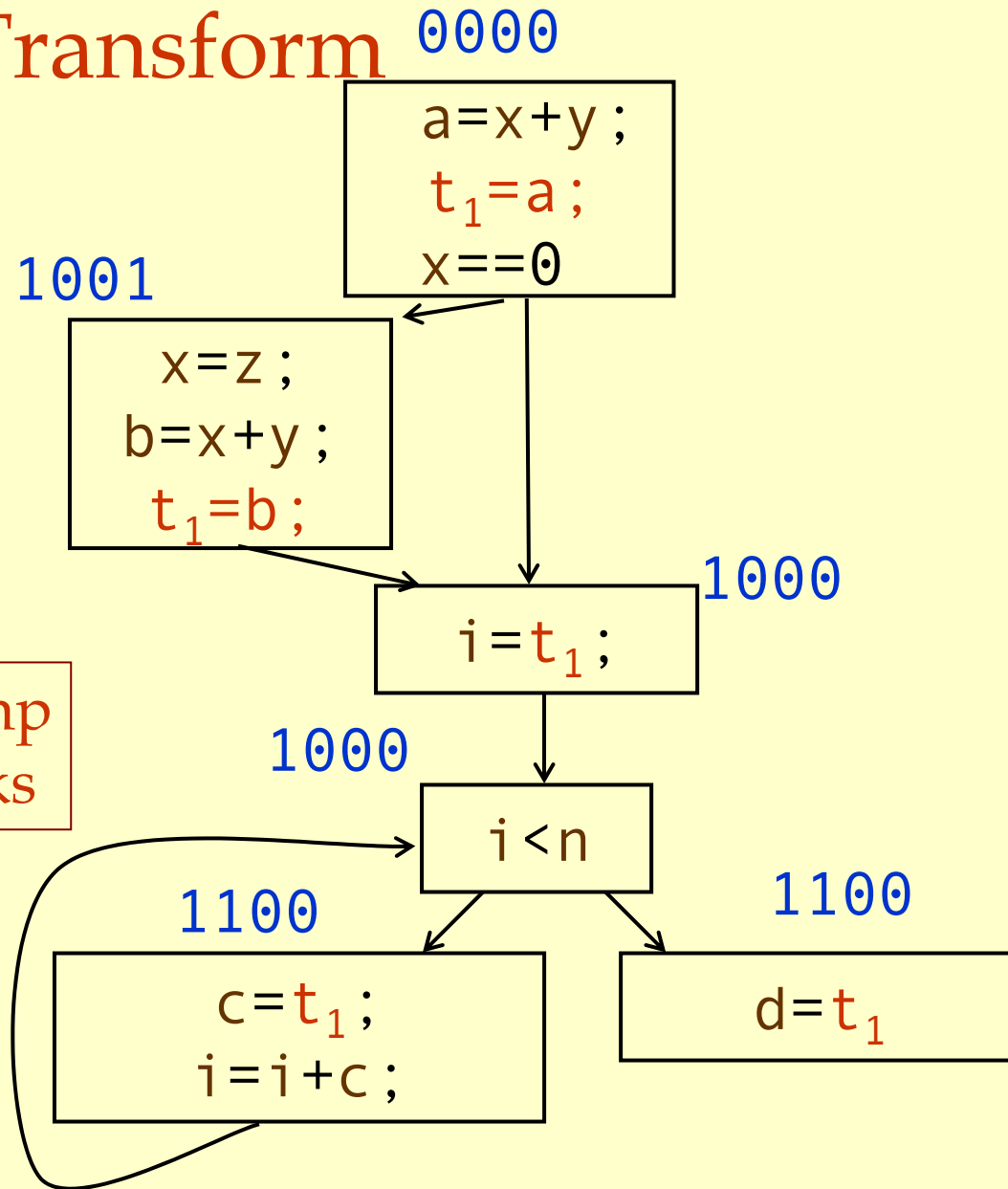


Global CSE Transform

Expressions

- 1: $x+y$
- 2: $i < n$
- 3: $i+c$
- 4: $x == 0$

Must use same temp for CSE in all blocks

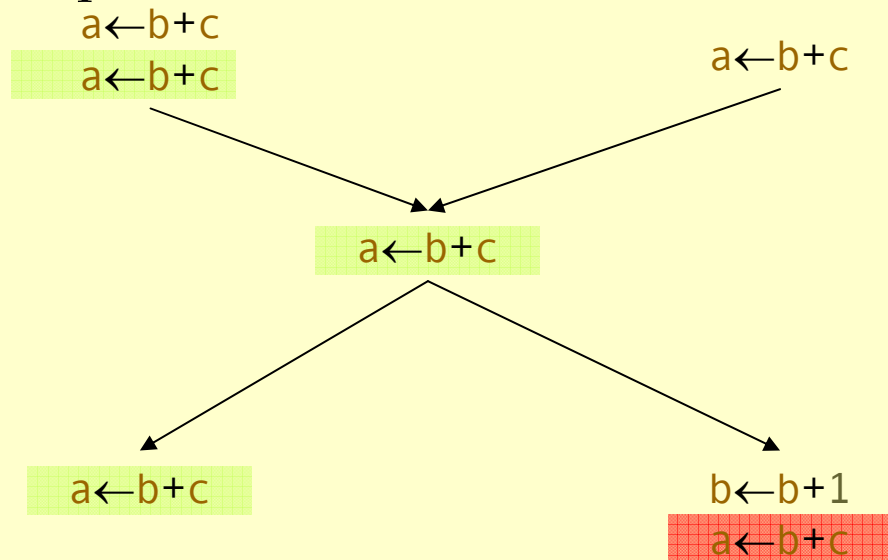


Redundant Expressions

An expression is redundant at a point p if, on every path to p

1. It is evaluated before reaching p , and
2. None of its constituent values is redefined before p

Example



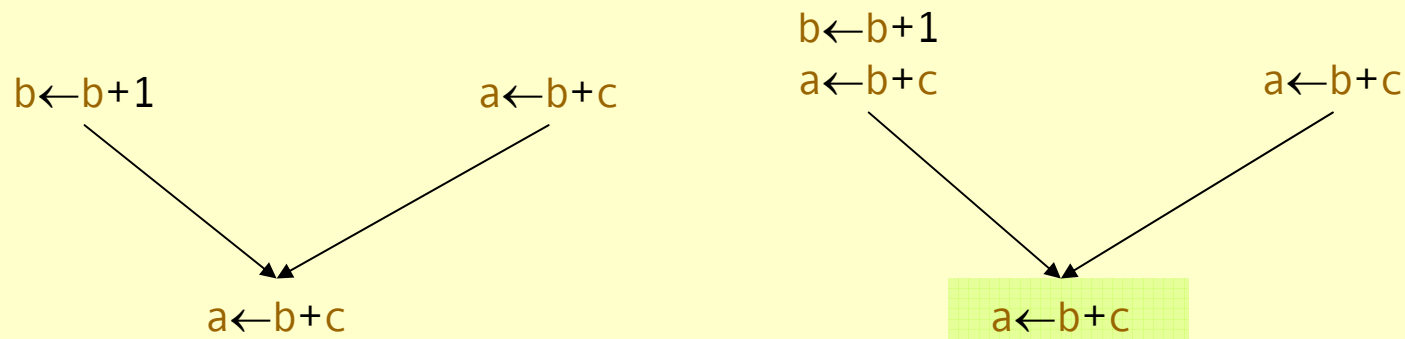
Some occurrences of $b+c$ are redundant

Not all occurrences of $b+c$ are redundant!

Partially Redundant Expressions

An expression is partially redundant at p if it is redundant along some, but not all, paths reaching p .

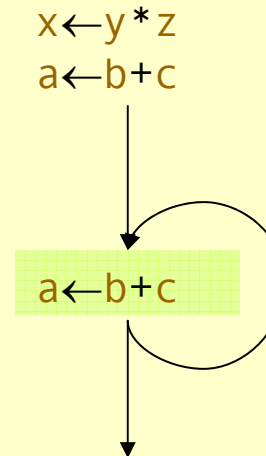
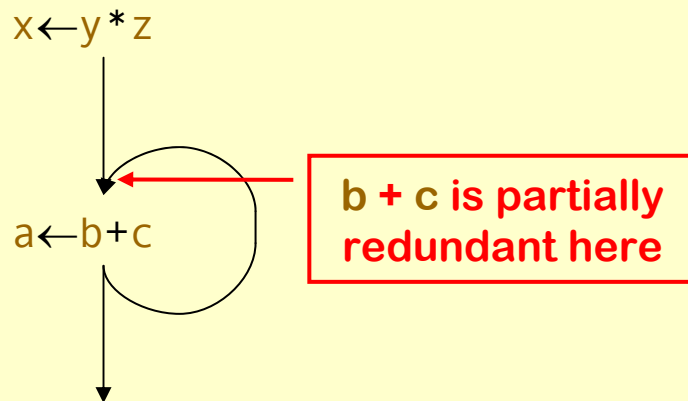
Example



Inserting a copy of “ $a \leftarrow b+c$ ” after the definition of b can make it redundant.

Loop Invariant Expressions

Another example:



Loop invariant expressions are partially redundant.

- ◆ Partial redundancy elimination performs code motion.
- ◆ Major part of the work is figuring out where to insert operations.

Memory Management

Advanced Compiler Techniques

2004

Erik Stenman

EPFL

Memory Management

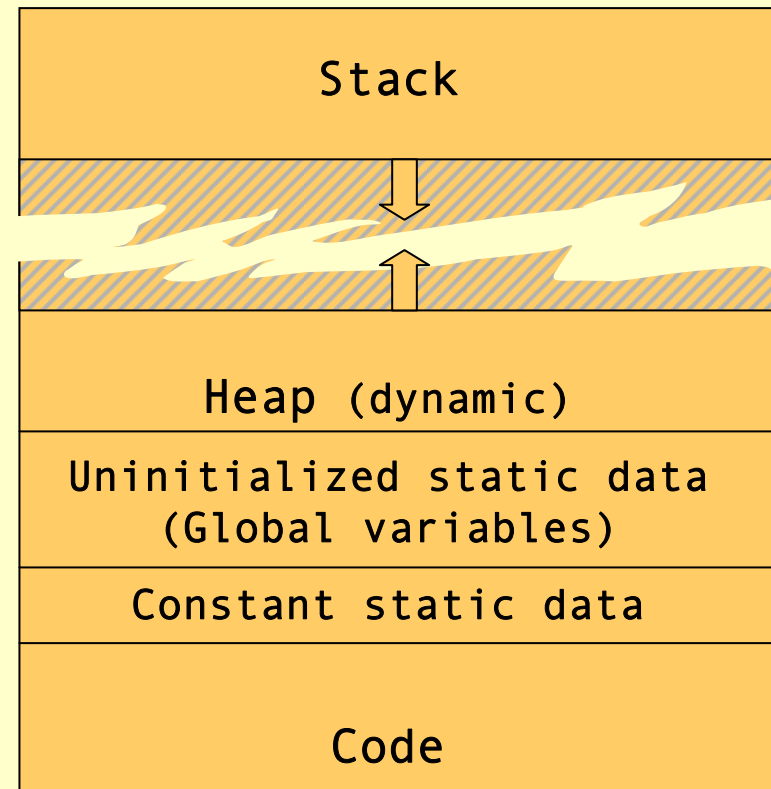
- ◆ The computer memory is a limited resource so the memory use of programs has to be managed in some way.
- ◆ The memory management is usually performed by a *runtime system* with help from the compiler.
 - ◆ The runtime system is a set of system procedures linked to the program.
 - ◆ For C programs it can be as simple as a small library for interacting with the operating system.
 - ◆ For Erlang programs the runtime system implements almost all the functionality normally provided by the OS.

Memory Management

- ◆ In a language such as C there are three ways to allocate memory:
 1. Static allocation. The size of memory needed by global variables (and code) is decided at compile time.
 2. Stack allocation. Activation records are allocated on the stack at function calls.
 3. Heap allocation. Dynamically allocated by the programmer by the use of `malloc`.

Memory Organization

- ◆ A typical layout of the memory of a C program looks like:



Dynamic Memory Management

- ◆ Heap allocation is necessary for data that lives longer than the function which created it, and which is passed by reference, e.g., lists in misc.
- ◆ Two design questions for the heap:
 - ◆ How is space for data allocated on the heap?
 - ◆ How and when is the space deallocated?
- ◆ Considerations in memory management design:
 - ◆ Space leaks & dangling pointers.
 - ◆ The cost for allocation and deallocation.
 - ◆ Space overhead of the memory manager.
 - ◆ Fragmentation.

Fragmentation

- ◆ The memory management system should try to avoid *fragmentation*, i.e. when the free memory is broken up into several small blocks instead of few large blocks.
- ◆ In a fragmented system memory allocation may fail because there is no free block that is large enough even though the total free memory would be large enough.
- ◆ We distinguish between:
 - ◆ Internal fragmentation – the allocated block is larger than the requested size (the waste is in the allocated data).
 - ◆ External fragmentation – all free blocks are too small (the waste is in the layout of the free data).

Memory Allocation

- ◆ The use of a free-list is a common scheme.
- ◆ The system keeps a list of unused memory blocks.
- ◆ To allocate memory the free-list is searched to find a block which is large enough.
- ◆ The block is removed from the free-list and used to store the data. If the block is larger than the need, it is split and the unused part is returned to the free-list (to avoid internal fragmentation).
- ◆ When the memory is freed it is returned to the free-list. Adjacent memory blocks can be merged (or coalesced) into larger blocks (to avoid external fragmentation).

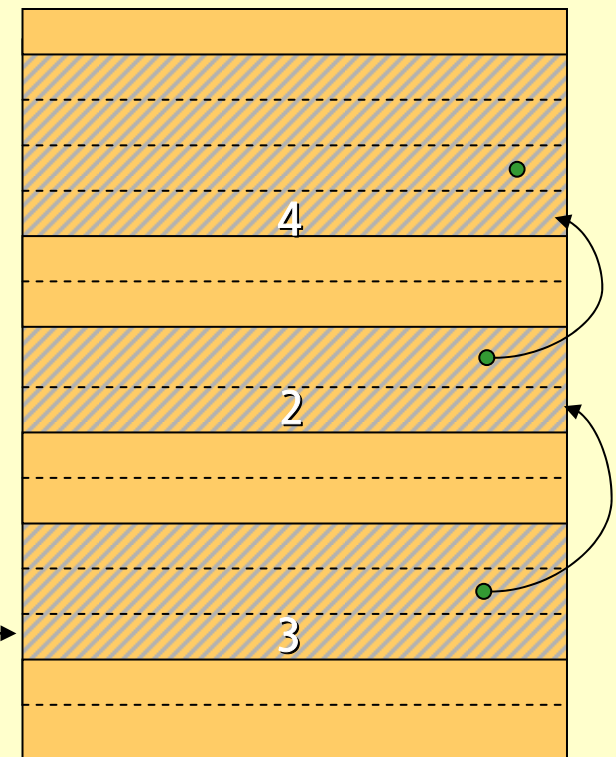
Free-list

- ◆ The free-list can be stored in the free memory since it is not used for anything else. (We assume, or ensure, that each memory block is at least two words).

Free list: ●

This can be stored as a static global variable.

In use 
Free 



Free-list

- ◆ Note that we **need to know the size** of a block when it is deallocated. This means that even allocated blocks need to have **a size field** in them.
- ◆ Thus the space overhead will be at least **one word per allocated data object**. (It might also be advantageous to keep the link.)
- ◆ The cost (time) of allocation/deallocation is proportional to the search through the free-list.

Free-list

- ◆ There are many different ways to implement the details of the free-list algorithm:
 - ◆ Search method: first-fit, best-fit, next-fit.
 - ◆ Links: single, double.
 - ◆ Layout: one list, one list per block size, tree, buddy.

Deallocation

- ◆ Deallocation can either be *explicit* or *implicit*.
- ◆ Explicit deallocation is used in e.g., Pascal (new/dispose), C (malloc/free), and C++ (new/delete).
- ◆ Implicit deallocation is used in e.g., Lisp, Prolog, Erlang, ML, and Java.

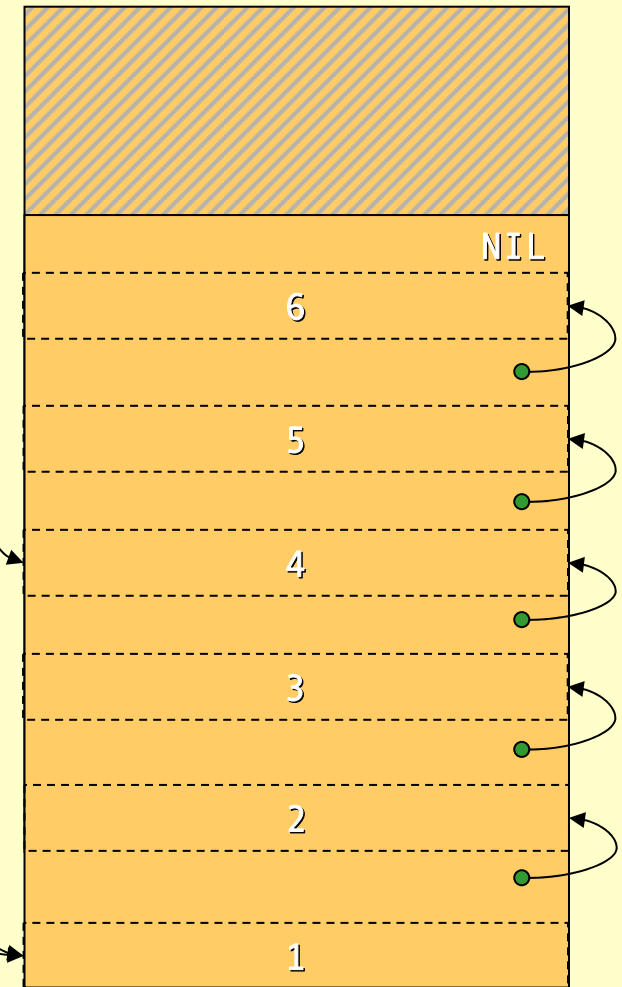
Explicit Deallocation

- ◆ Explicit deallocation has a number of problems:
 - ◆ If done too soon it leads to dangling pointers.
 - ◆ If done too late (or not at all) it leads to space leaks.
 - ◆ In some cases it is almost impossible to do it at the right time. Consider a library routine to append two mutable lists:

```
c = append(a, b);
```

Explicit Deallocation

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
printList(b);
```



Explicit Deallocation

```
list a = new List(1,2,3);  
list b = new List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
printList(b);  
free(c);
```

- ◆ The programmer now has to ensure that *a*, *b*, and *c* are all deallocated at the same time. A mistake would lead to dangling pointers.
- ◆ If *b* is in use long after *a*, and *c*, then we will keep *a* live too long. A space leak.

Implicit Deallocation

- ◆ With *implicit deallocation* the programmer does not have to worry about when to deallocate memory.
- ◆ The runtime system will *dynamically* decide when it is **safe** to do this.
- ◆ In some cases, and systems, the compiler can also add static deallocations to the program.
- ◆ The most commonly used automatic deallocation method is called *garbage collection* (GC).
- ◆ There are other methods such as *region based* allocation and deallocation.

Garbage Collection (GC)

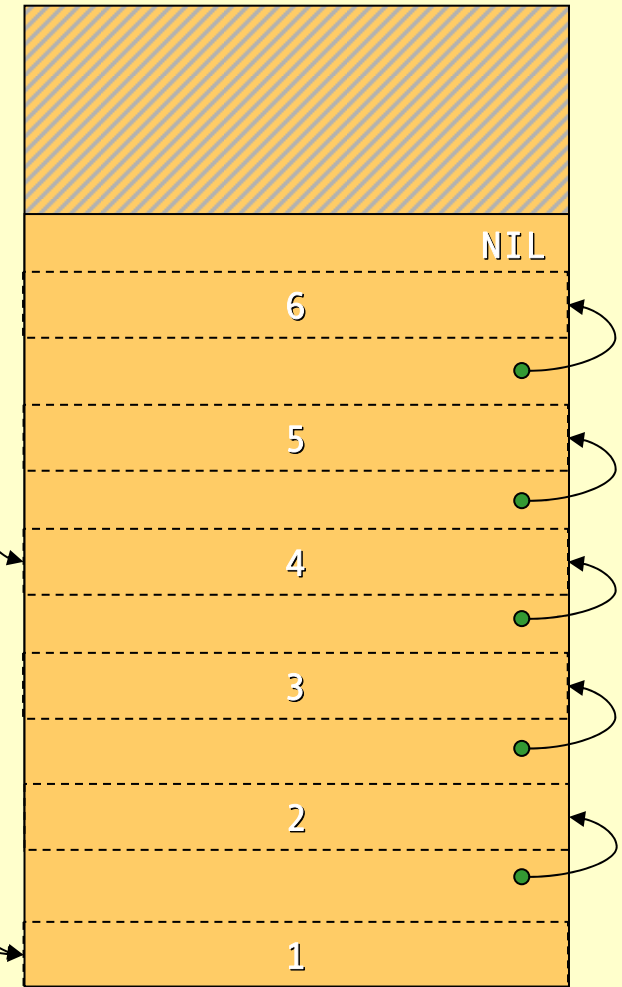
- ◆ *Garbage collection* is a common name for a set of techniques to deallocate heap memory that is unreachable by the program.
- ◆ There are several different base algorithms: *reference counting, mark & sweep, copying.*
- ◆ We can also distinguish between how the GC interferes or interacts with the program: *disruptive, incremental, real-time, concurrent.*

The Reachability Graph

- ◆ The data reachable by the program form a directed graph, where the edges are pointers.
- ◆ The *roots* of this graph can be in:
 1. global variables,
 2. registers,
 3. local variables & formal parameters on the stack.
- ◆ Objects are *reachable* iff there is a path of edges that leads to them from some root. Hence, the compiler must tell the GC where the roots are.

The Reachability Graph

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```



The Reachability Graph

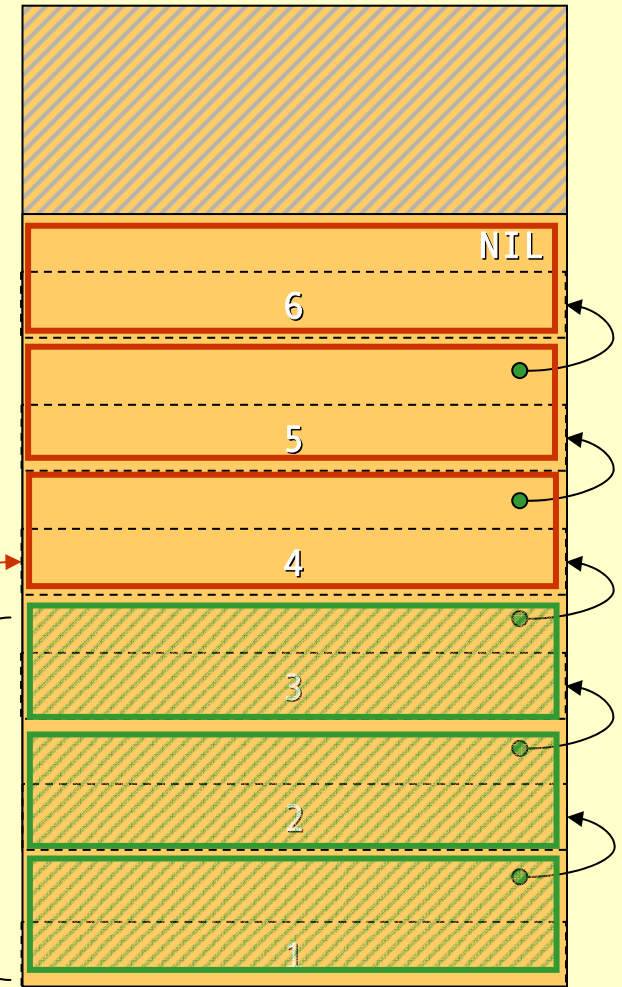
```

list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
doLotsOfStuff();
return b;

```

roots: b

The goal with the GC is to deallocate these:

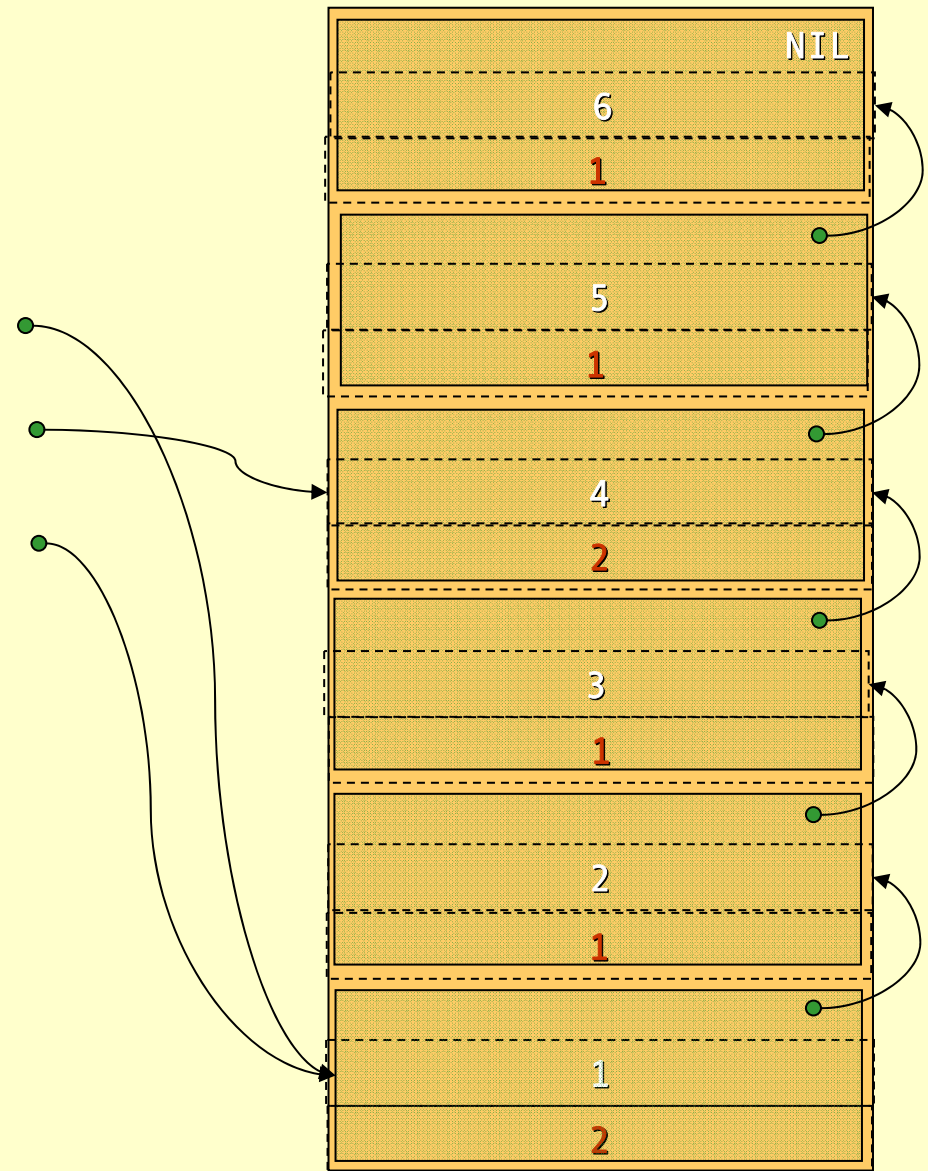


Reference Counting

- ◆ Idea: Keep track of how many references there are to each object.
- ◆ If there are 0 references deallocate the object.
- ◆ The compiler must add code to maintain the reference count (refcount).
 - ◆ Set the count to 1 when created.
 - ◆ For an assignment $x = y$:
 - ◆ if ($x \neq \text{null}$) $x.\text{refcount} -$;
 - ◆ if ($y \neq \text{null}$) $y.\text{refcount}++$;
 - ◆ When a stack frame is deallocated decrease the refcount of each object pointed to from the frame.
 - ◆ When refcount reaches 0 deallocate the object and decrease refcount of each child.

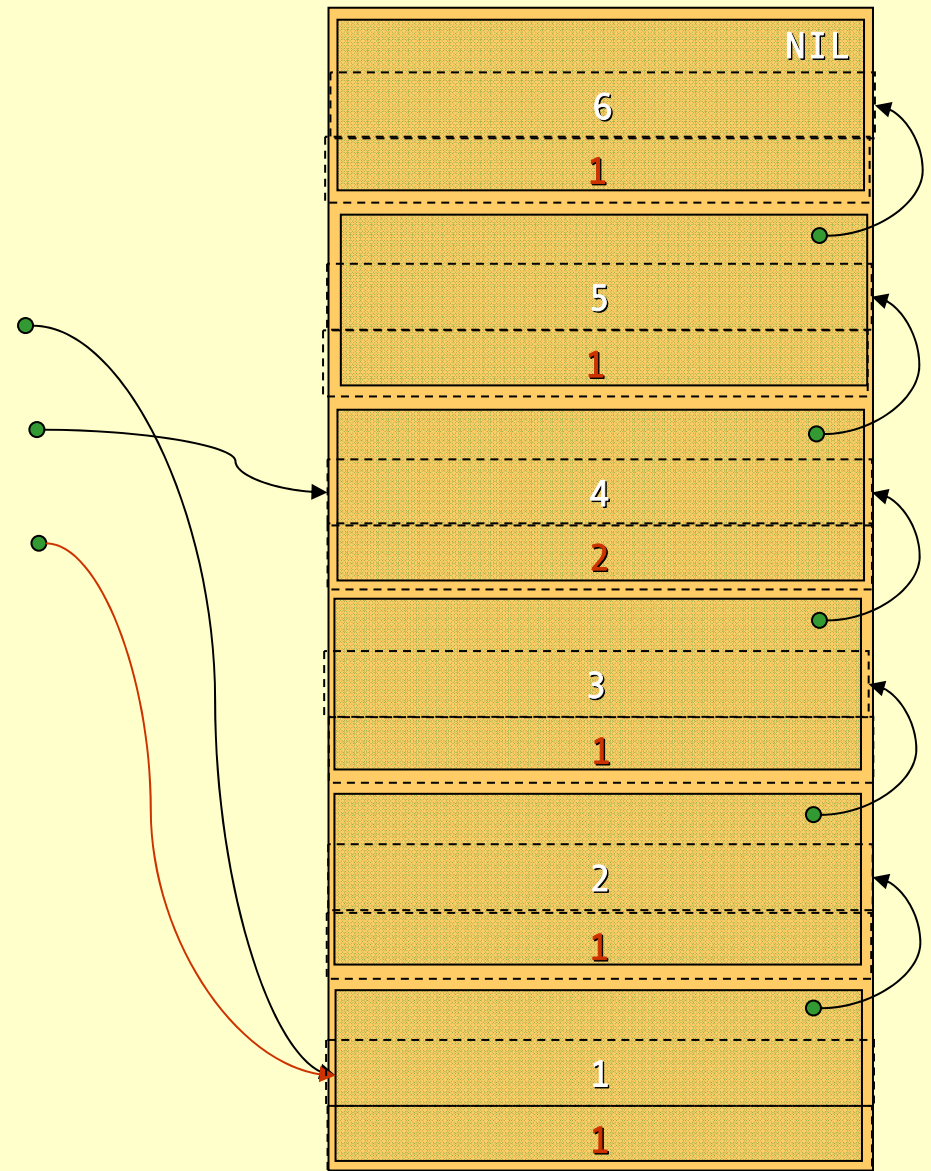
```

list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
decRefCount(c);
decRefCount(a);
doLotsOfStuff();
return b;
    
```



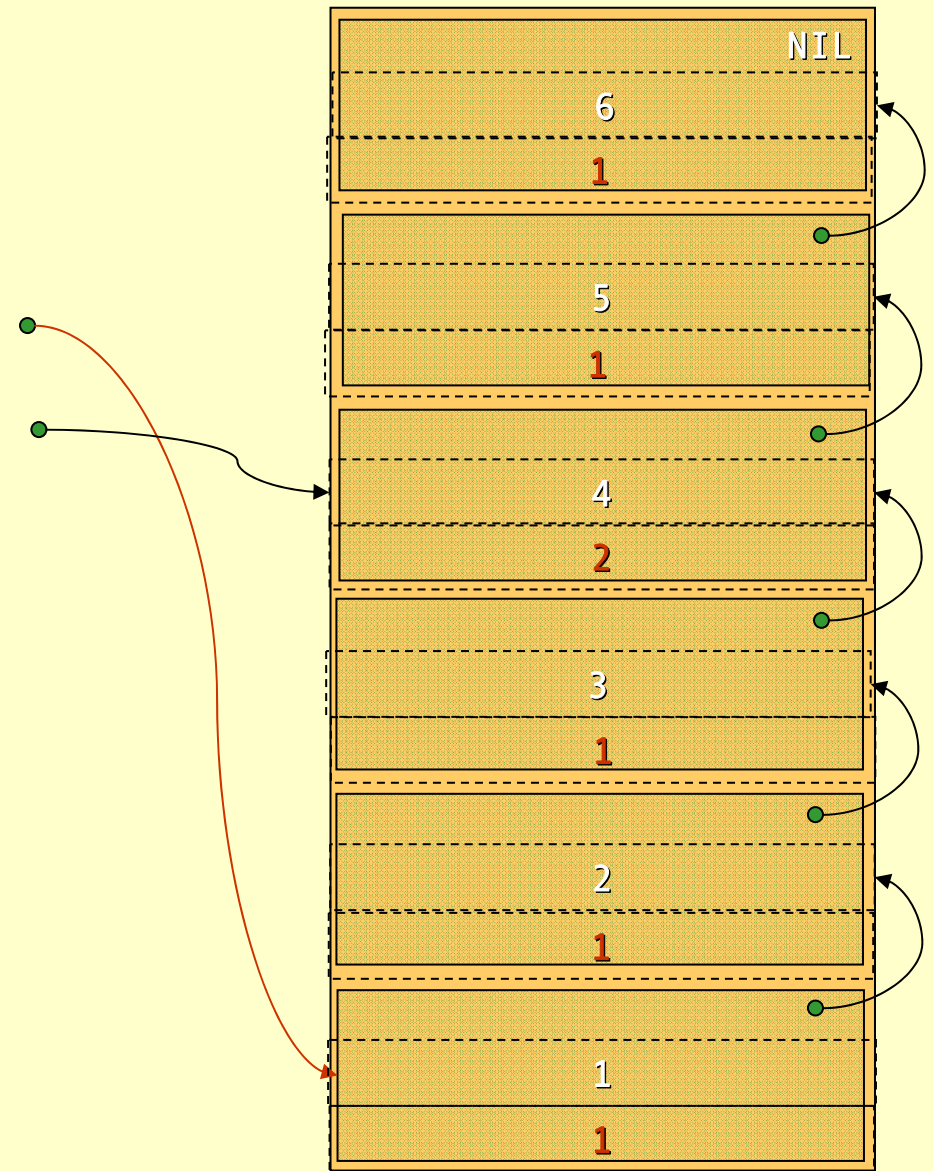
```

list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
decRefCount(c);
decRefCount(a);
doLotsOfStuff();
return b;
    
```



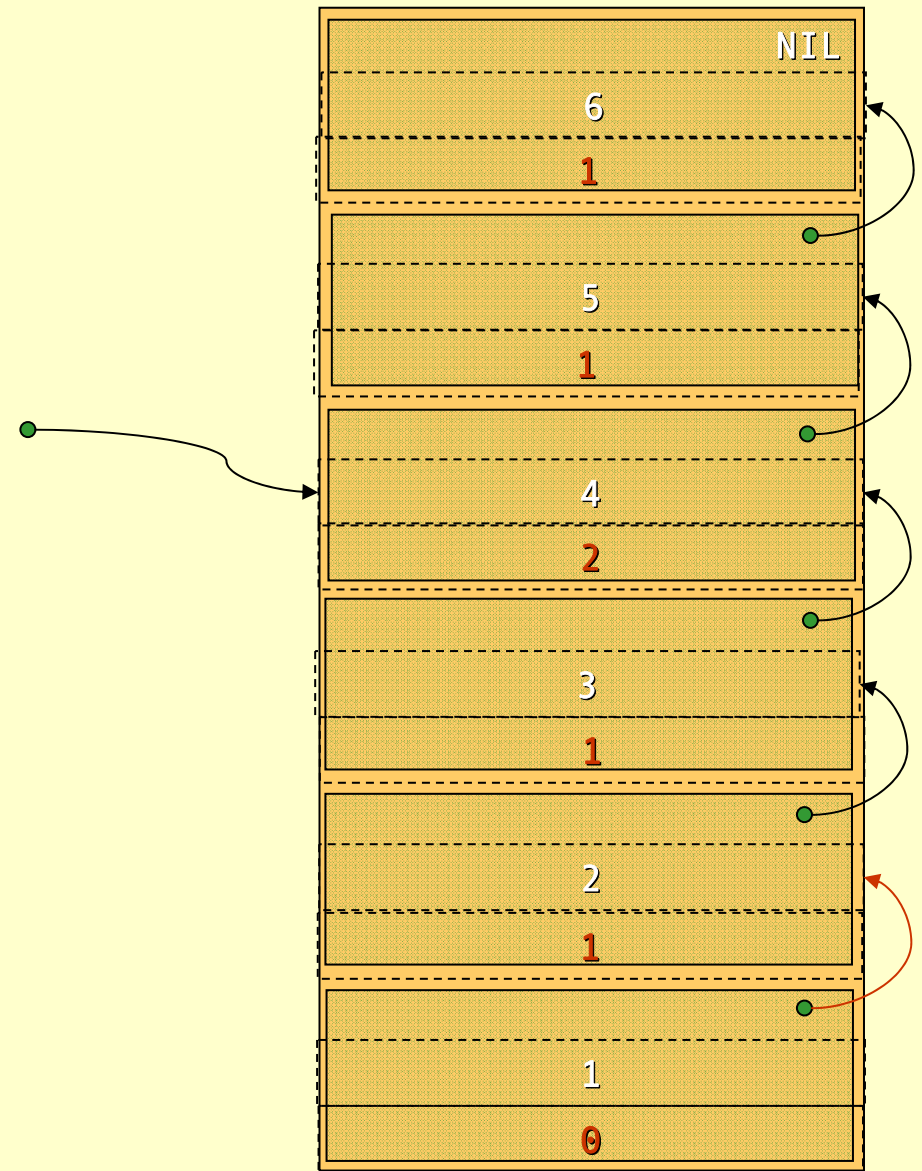
```

list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
decRefCount(c);
➔ decRefCount(a);
doLotsOfStuff();
return b;
    
```



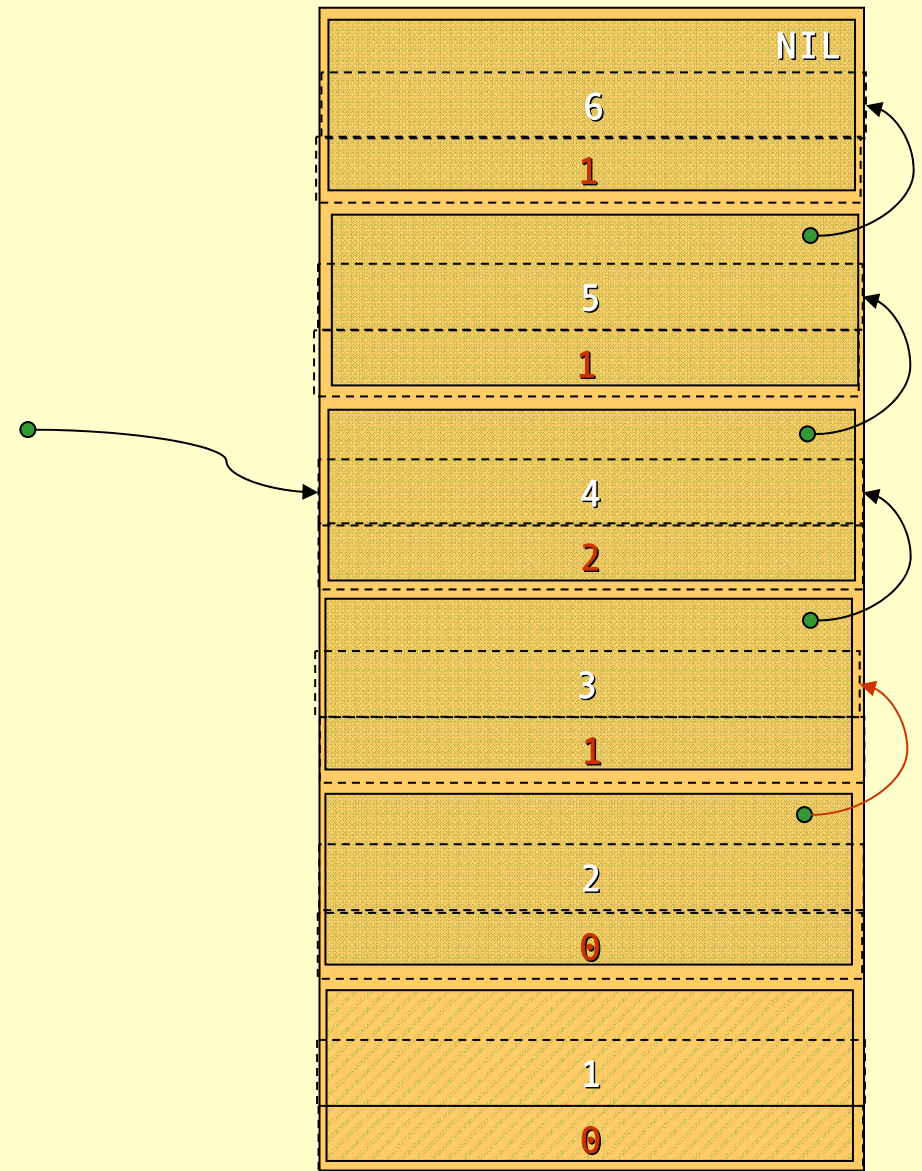

```

list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
decRefCount(c);
➔ decRefCount(a);
doLotsOfStuff();
return b;
    
```



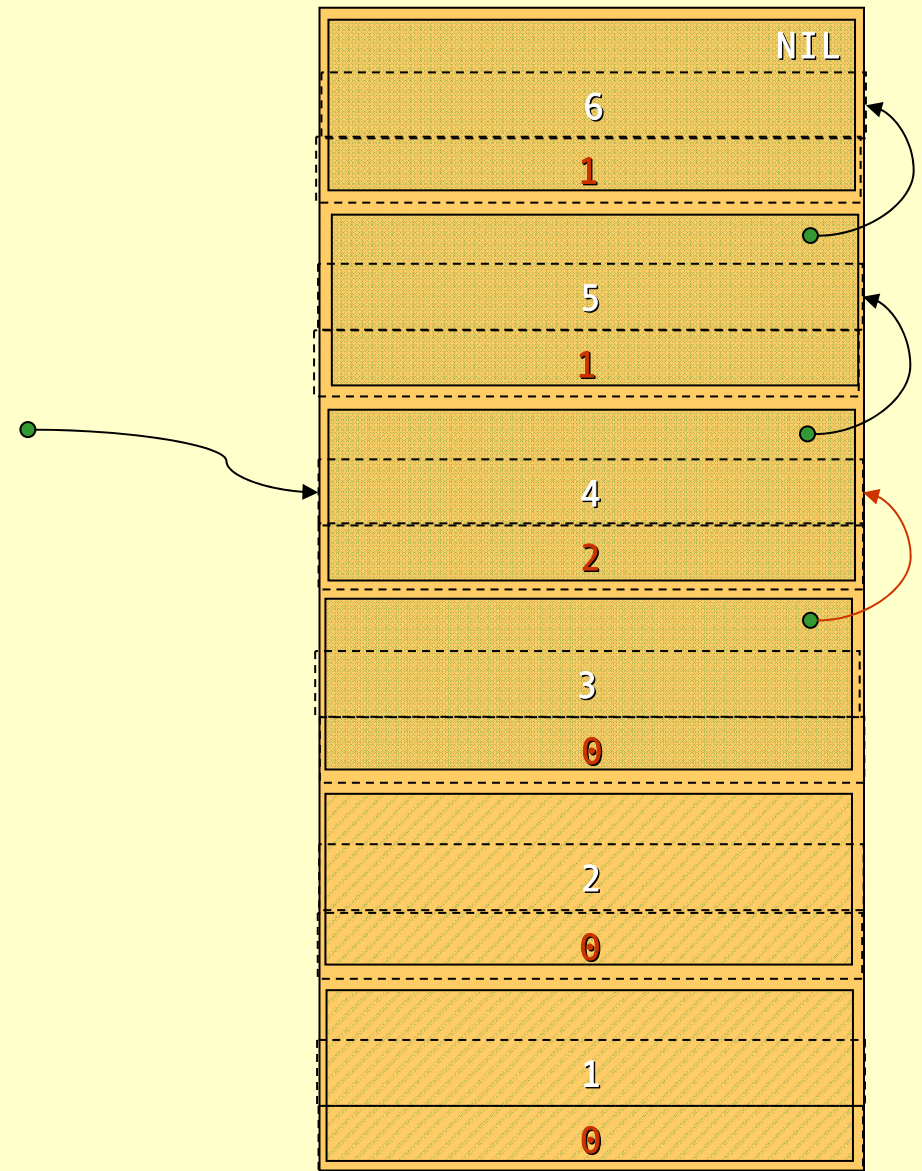
```

list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
decRefCount(c);
➔ decRefCount(a);
doLotsOfStuff();
return b;
    
```



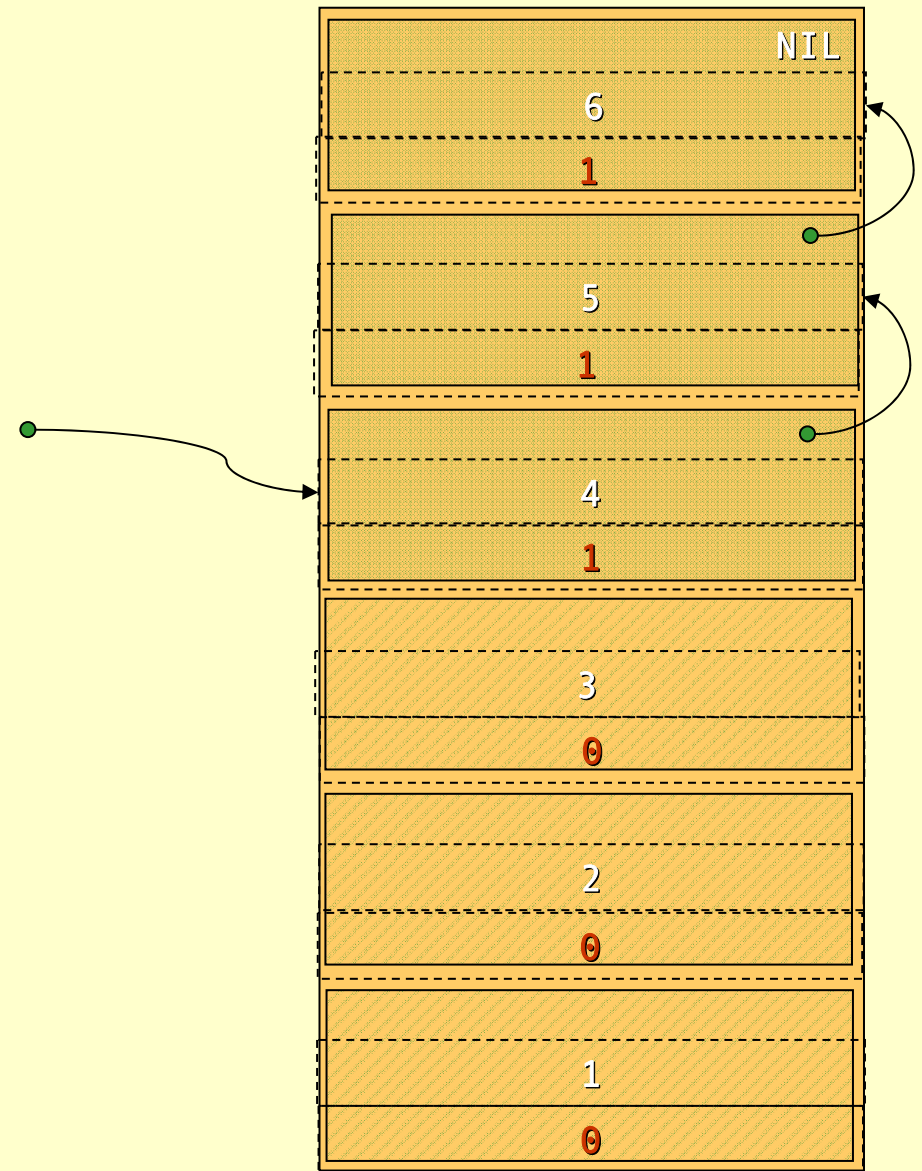
```

list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
decRefCount(c);
➔ decRefCount(a);
doLotsOfStuff();
return b;
    
```



```

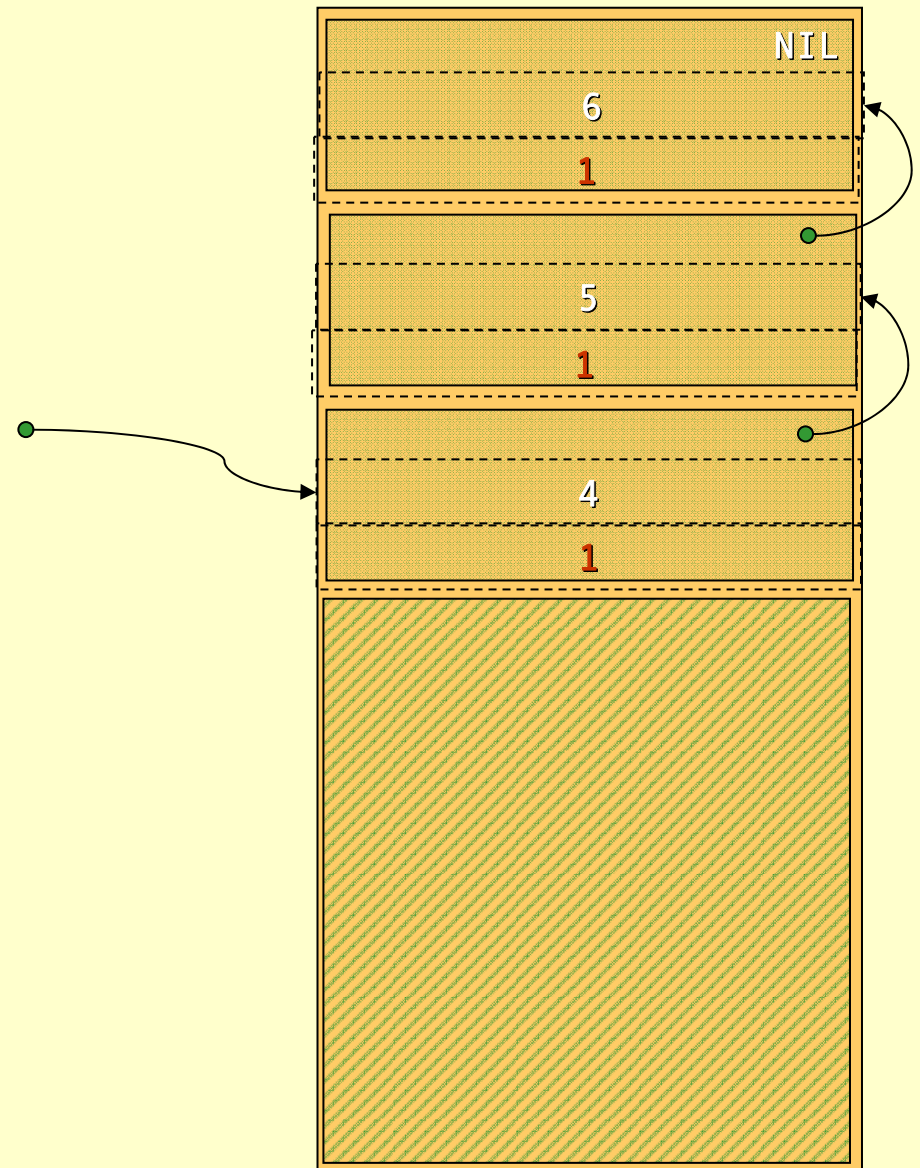
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
decRefCount(c);
➔ decRefCount(a);
doLotsOfStuff();
return b;
    
```




```

list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
decRefCount(c);
decRefCount(a);
doLotsOfStuff();
return b;

```



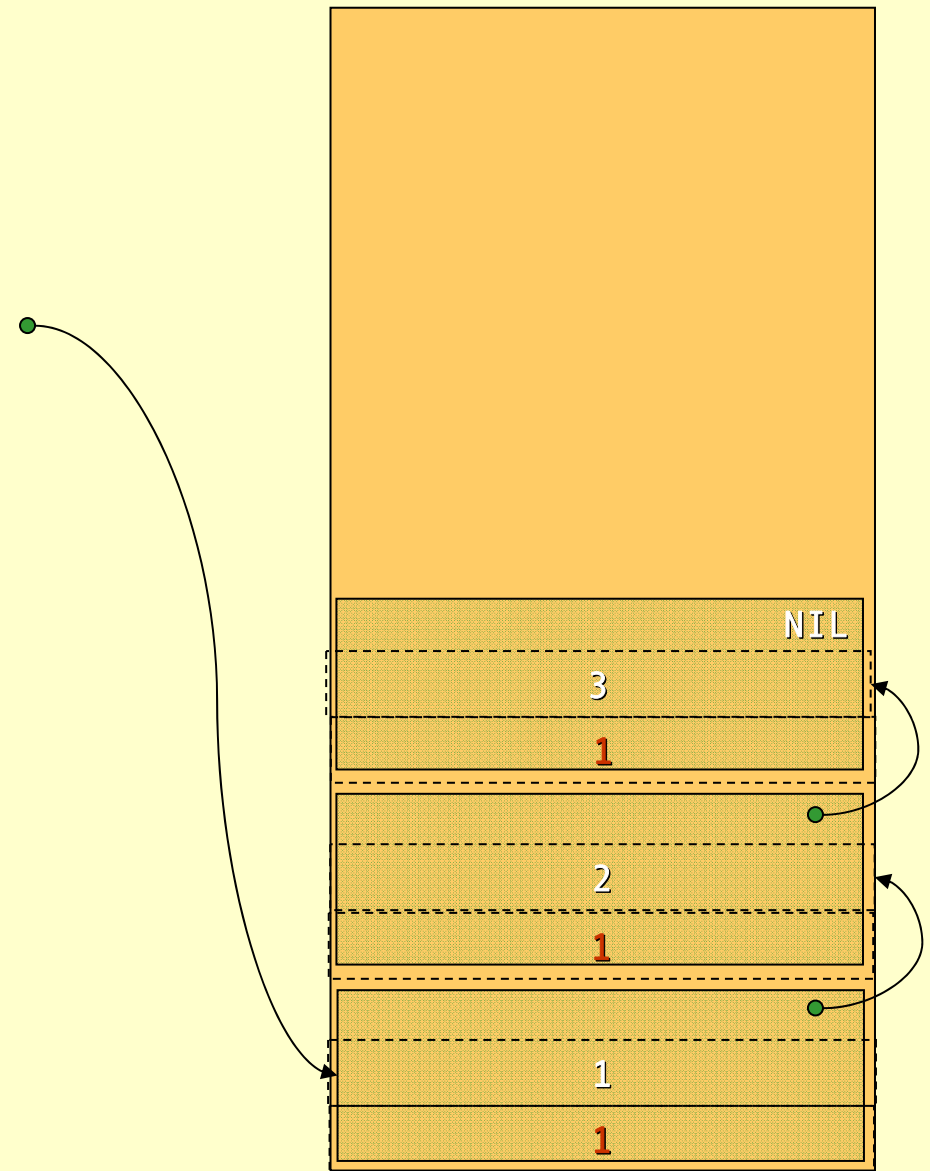
Reference Count

- ◆ Advantages of reference count:
 - ◆ Rather easy to implement.
 - ◆ Storage reclaimed **immediately**.
- ◆ Disadvantages of reference count:
 - ◆ **Space overhead**: 1 word per object.
 - ◆ Keeping track of the reference counts is **very expensive**. (Each simple pointer copy becomes several instructions.)
 - ◆ There is one more problem...

```

➔ list a = List(1,2,3);
  list b = NIL;
  list c = append(a,a);
  printList(c);
  decRefCount(c);
  decRefCount(a);
  doLotsOfStuff();
  return b;

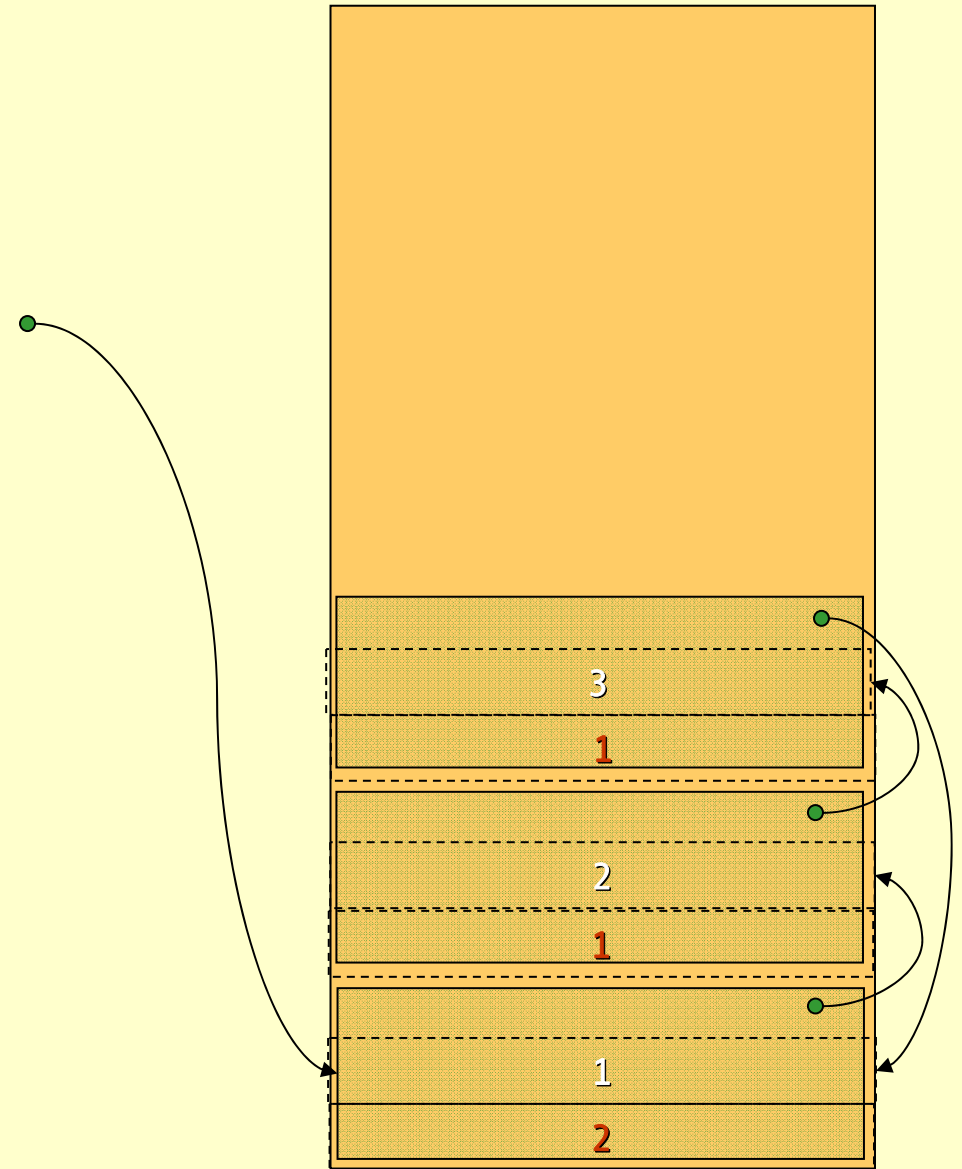
```




```

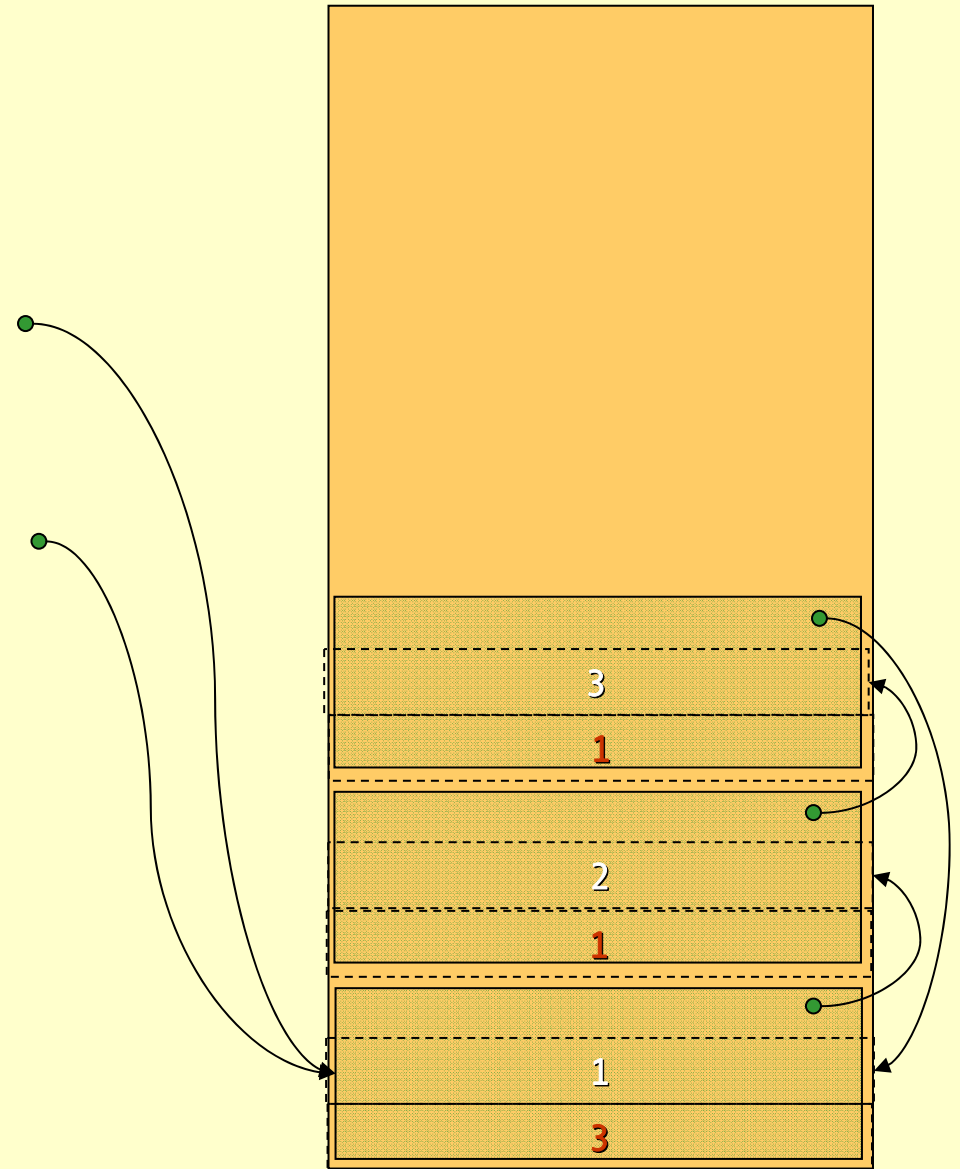
list a = List(1,2,3);
list b = NIL;
→ list c = append(a,a);
printList(c);
decRefCount(c);
decRefCount(a);
doLotsOfStuff();
return b;

```





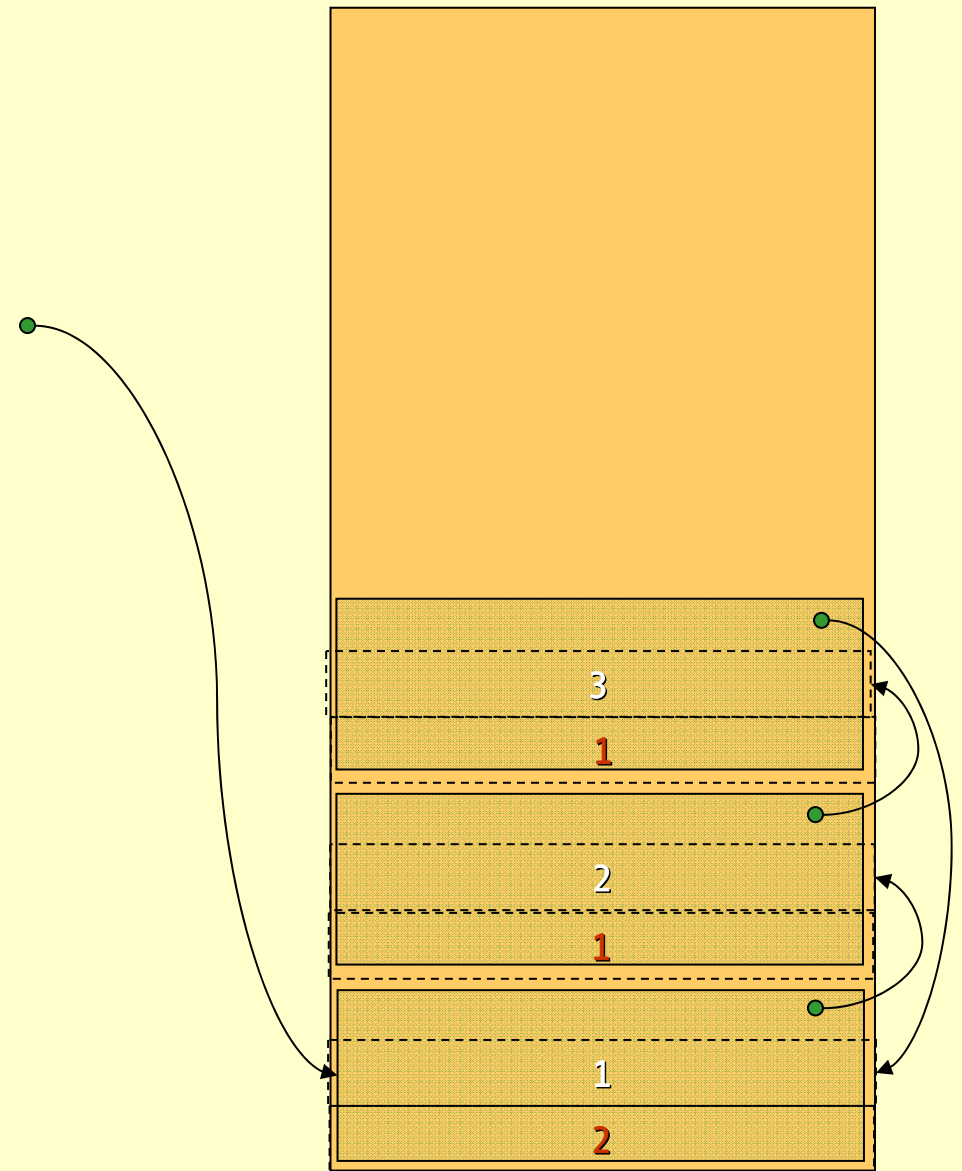
```
list a = List(1,2,3);
list b = NIL;
list c = append(a,a);
printList(c);
decRefCount(c);
decRefCount(a);
doLotsOfStuff();
return b;
```



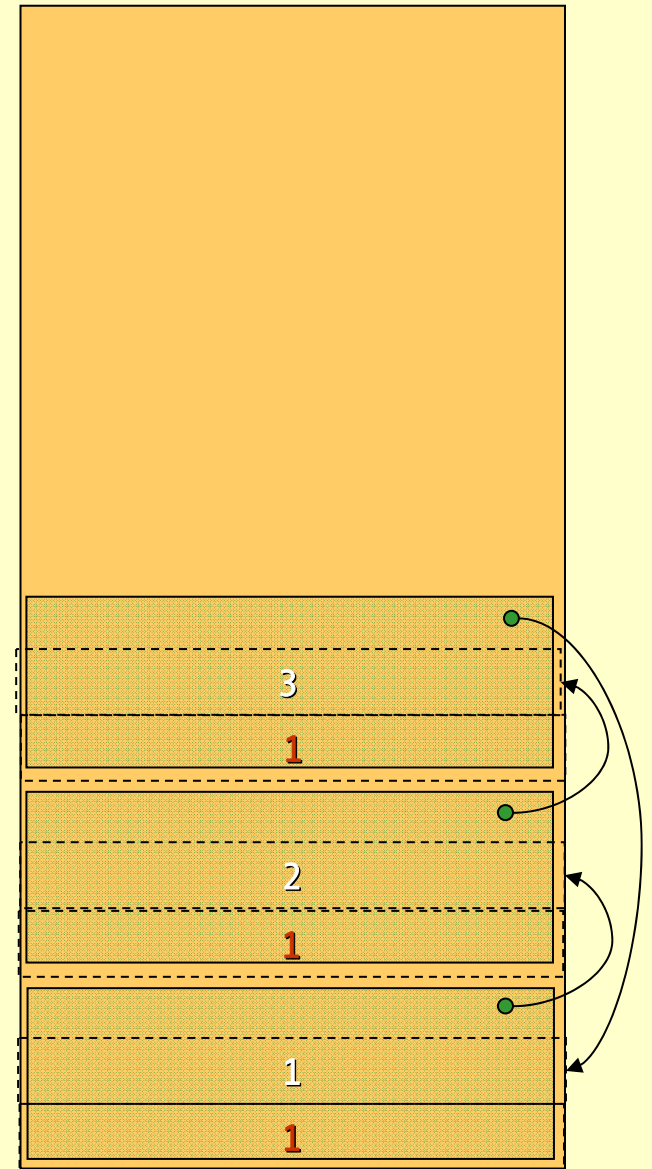
```

list a = List(1,2,3);
list b = NIL;
list c = append(a,a);
printList(c);
decRefCount(c);
→ decRefCount(a);
doLotsOfStuff();
return b;

```



```
list a = List(1,2,3);  
list b = NIL;  
list c = append(a,a);  
printList(c);  
decRefCount(c);  
decRefCount(a);  
doLotsOfStuff();  
return b;
```



Reference Count

- ◆ Big disadvantage with reference count:
 - ◆ The refcount of *cyclic structures* never reaches zero!
- ◆ There are ways to solve this, but they are very complicated.
- ◆ Due to this fact reference count is *very seldom* used in practice. There is one nice use, as we shall see later...
- ◆ In a pure language or a language without destructive updates there are no cyclic structures, making reference counting a viable option.

Mark & Sweep

- ◆ A *mark & sweep* GC is made up of two *phases*:
 1. First all reachable objects are *marked*.
 2. Then the heap is *swept* clean of dead objects.
- ◆ The mark phase is done by a *depth first search* through the reachability graph starting from the roots.

Depth First Mark Algorithm

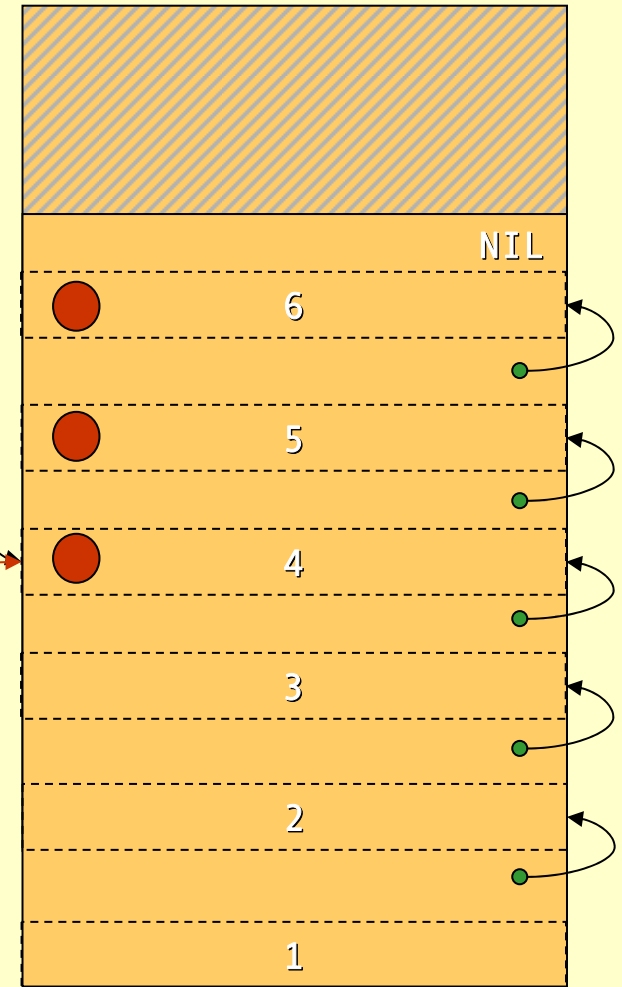
```
mark(x) {  
    if(! marked(x)) {  
        setMark(x);  
        for each field f of x  
            mark(*f)  
    }  
}
```

Example: Mark

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```



mark(b)



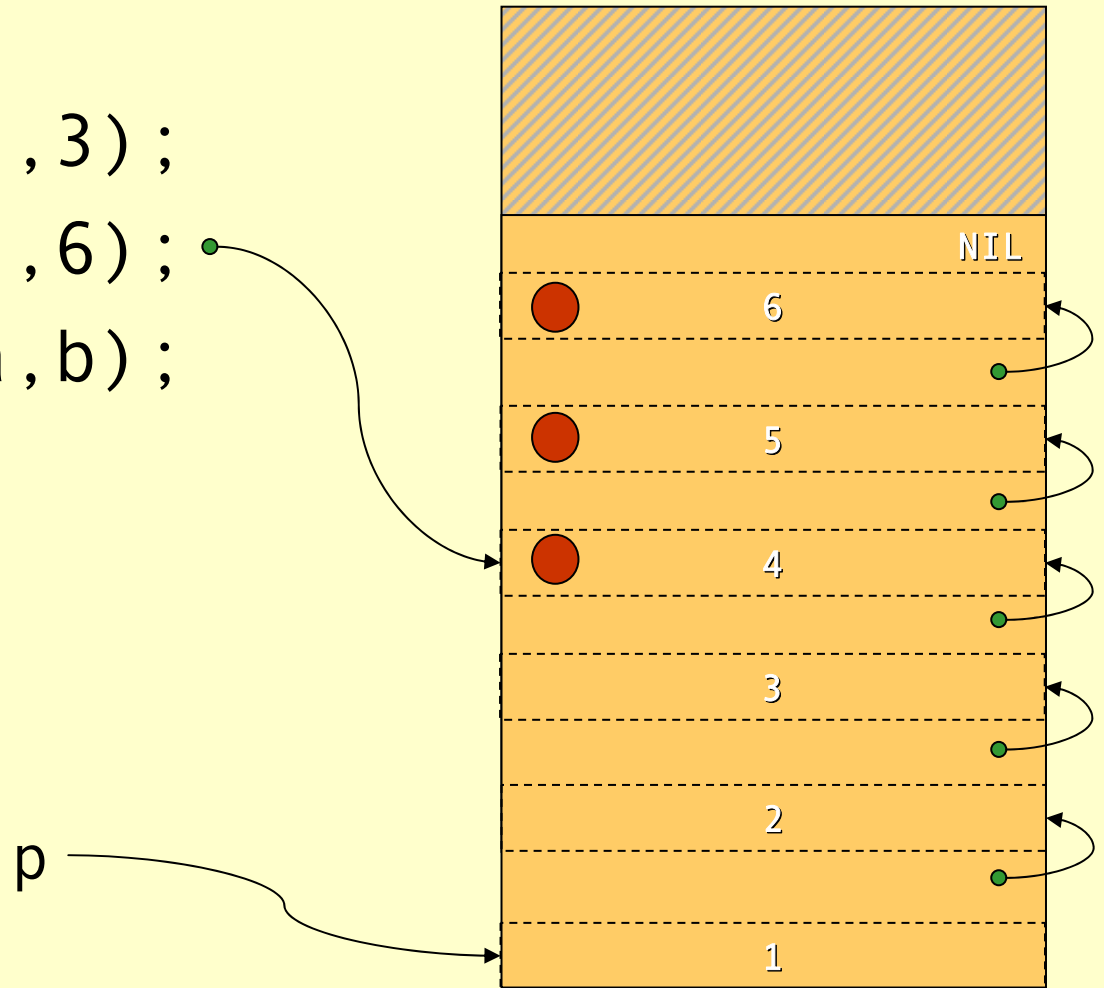
The Sweep

- ◆ The Sweep phase goes through the whole heap from start to finish and adds unmarked objects to the free-list.

```
p = heapStart;
while (p < heapEnd) {
    if (marked(*p)) clearMark(*p);
    else free(p);
    p += size(*p);
}
```

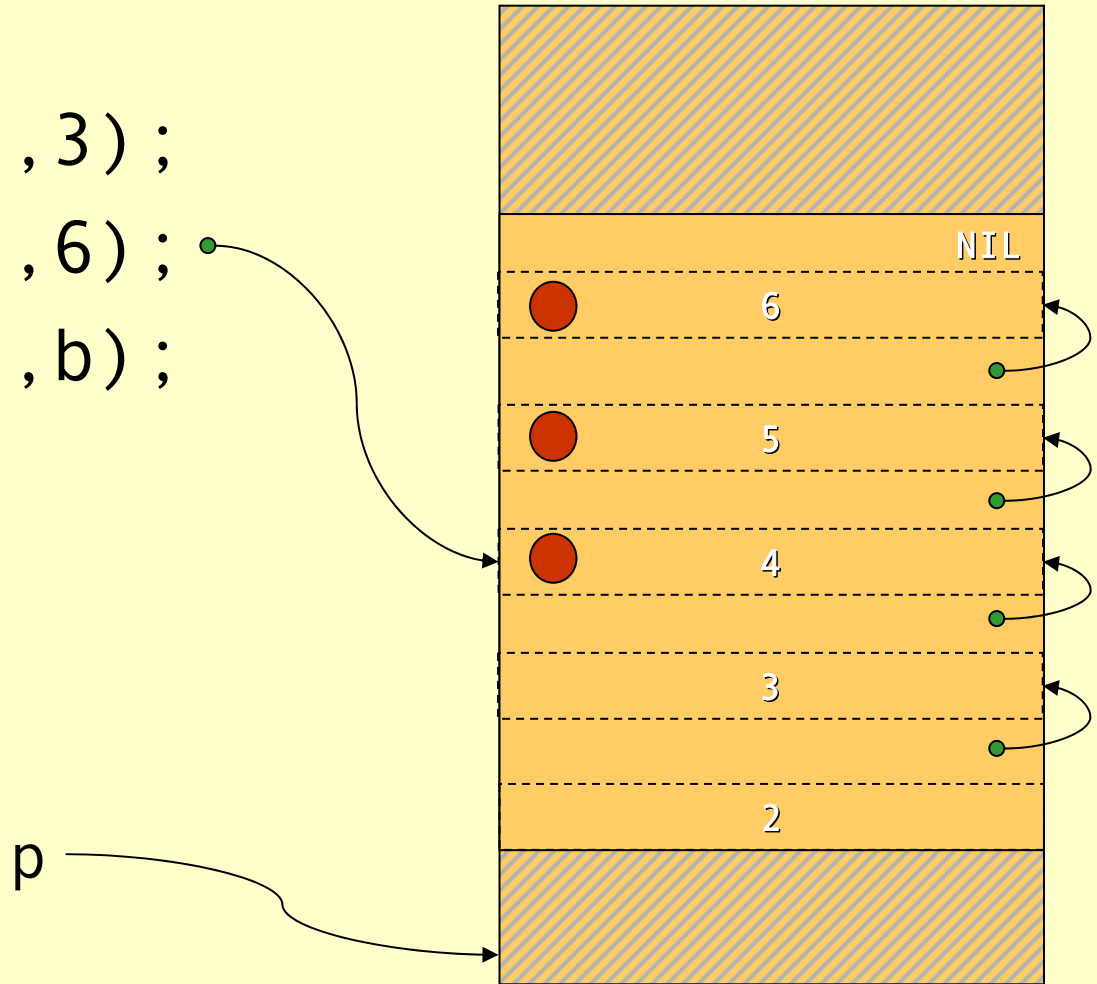
Example: Sweep

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```



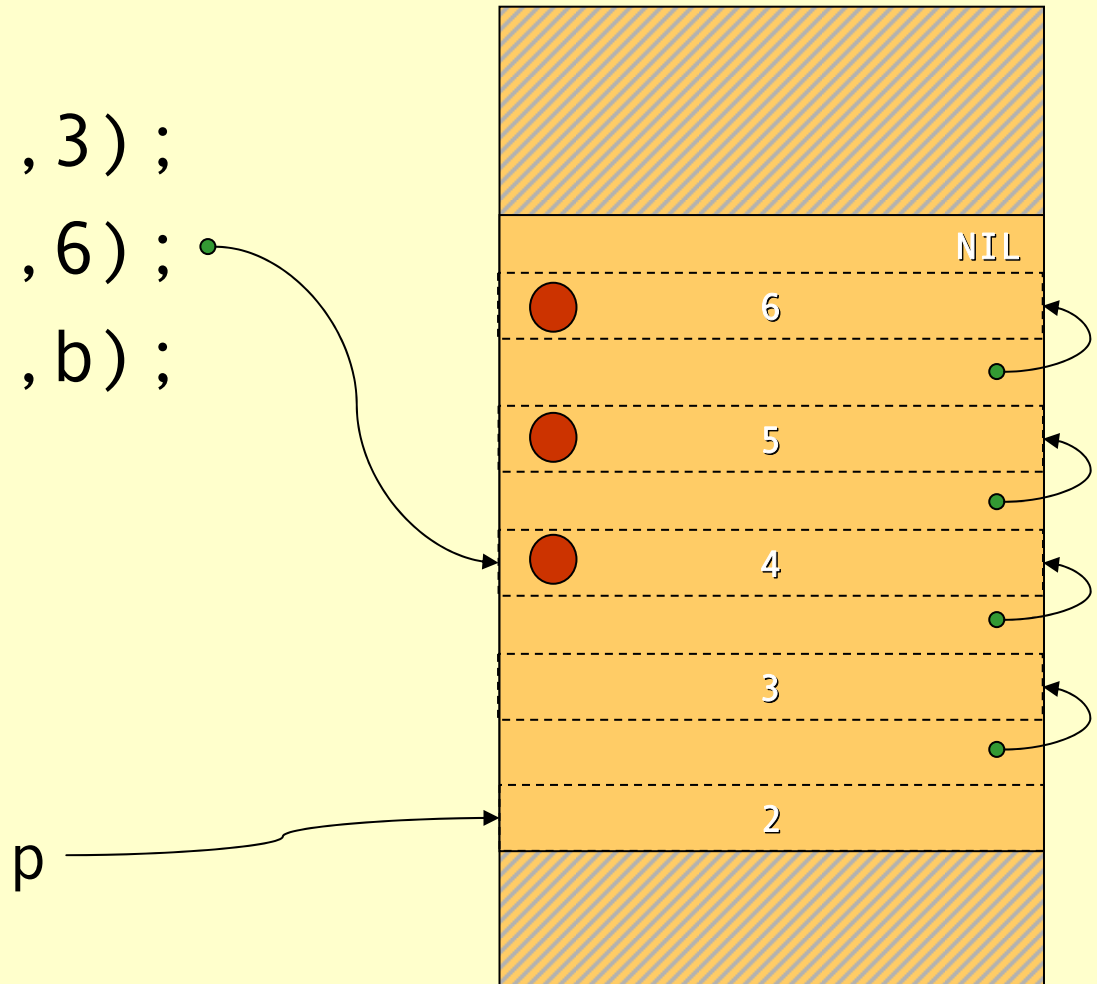
Example: Sweep

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```



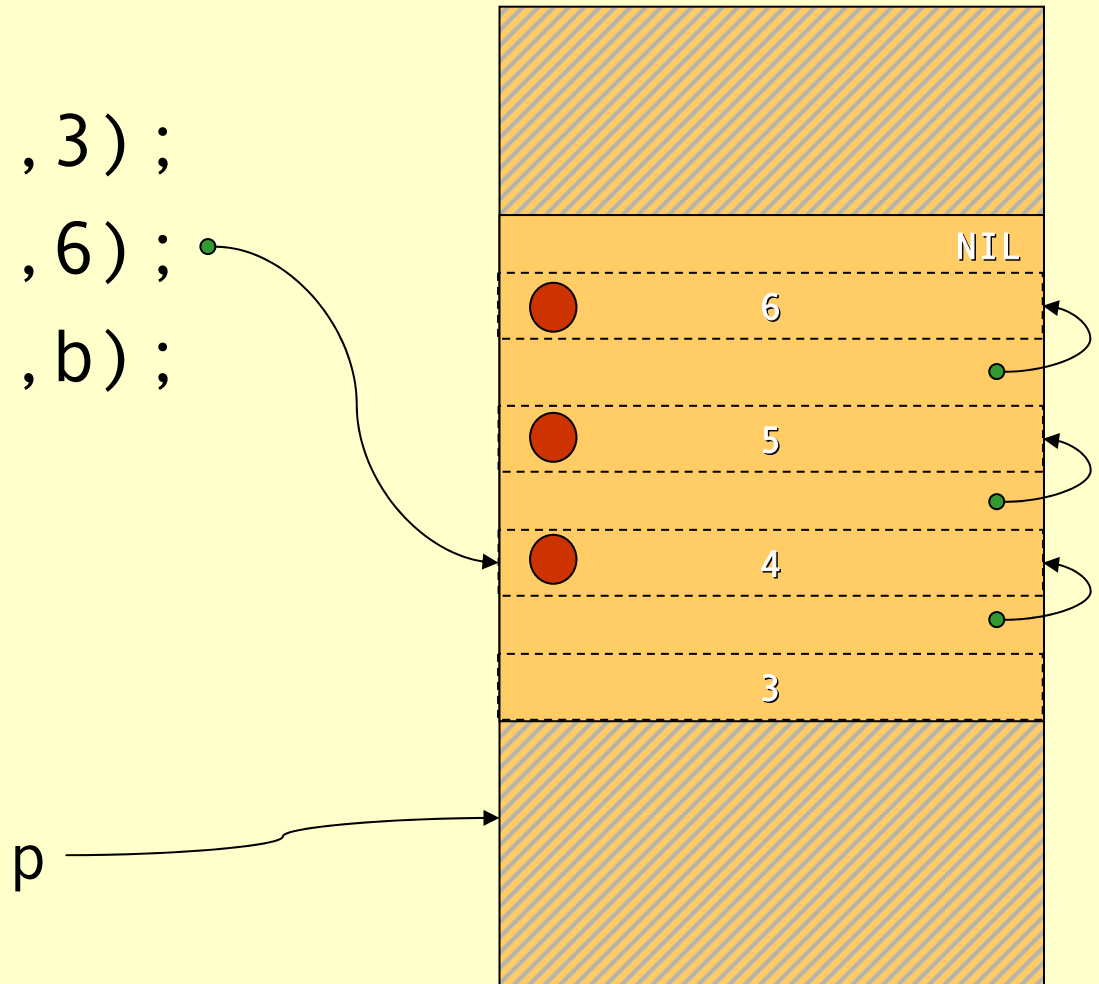
Example: Sweep

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```



Example: Sweep

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```

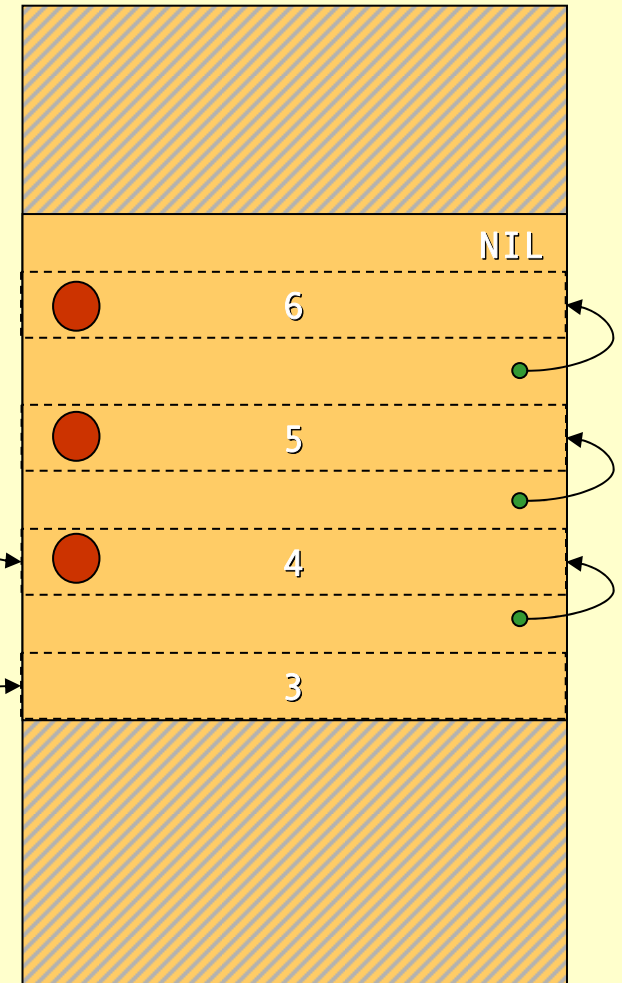


Example: Sweep

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```



p

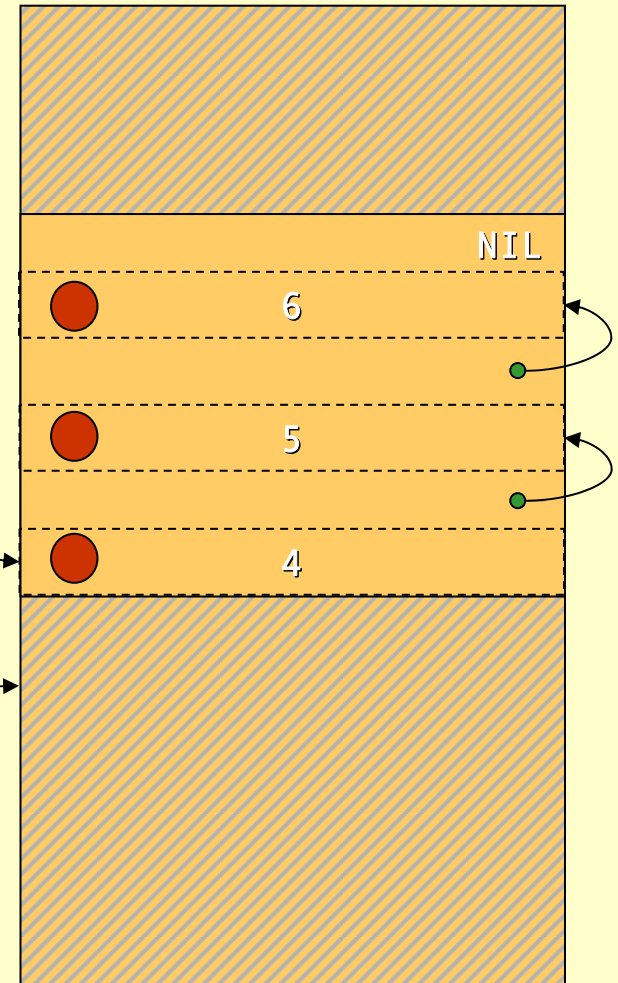


Example: Sweep

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```



p

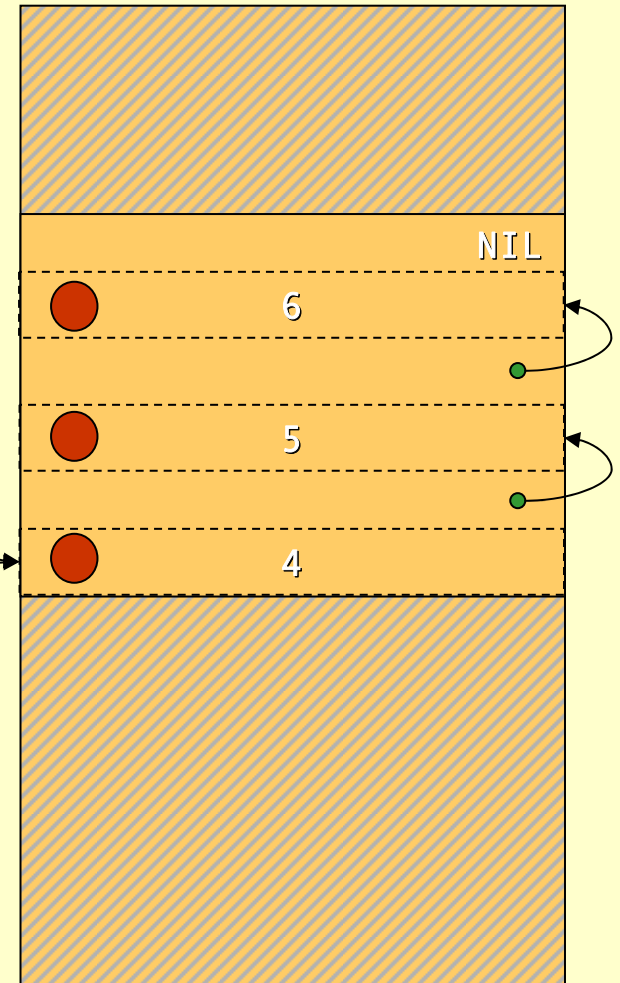


Example: Sweep

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```



p

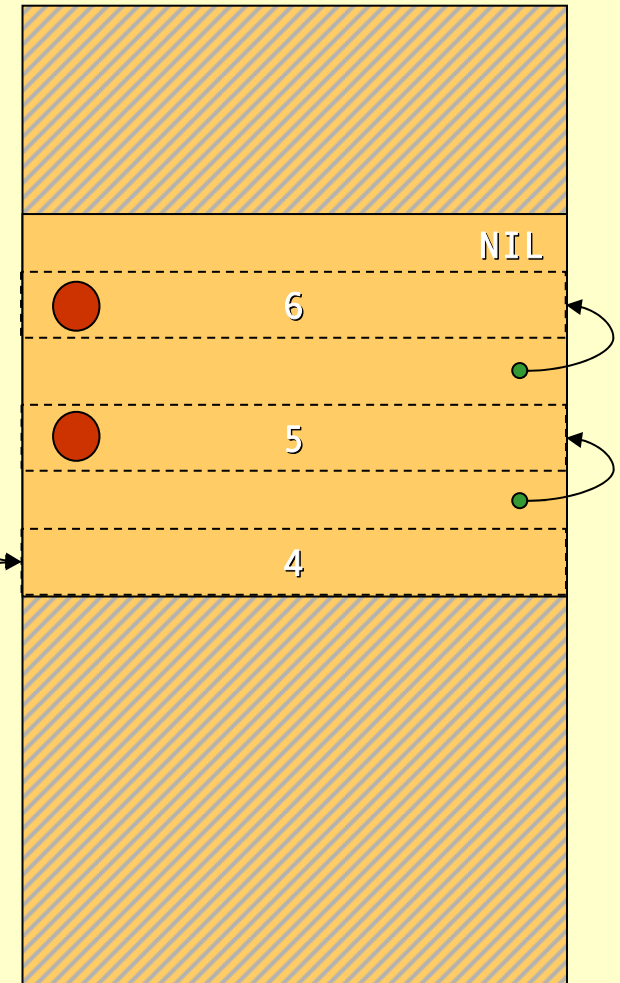


Example: Sweep

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```



p



Cost of Mark & Sweep

- ◆ The mark phase takes time proportional to the amount of reachable data (R).
- ◆ The sweep phase takes time proportional to the size of the heap (H).
- ◆ The work done by the GC is to recover $H-R$ words of memory.
- ◆ Their *amortized cost* of GC (overhead/allocated word) is:

$$\frac{c_1 R + c_2 H}{H - R}$$

- ◆ If $R \approx H$ the cost is very high. The cost goes down as the number of dead words increases.

Mark & Sweep

- ◆ Where do we store the mark bits?
 - ◆ We will discuss data representation a bit more at the end of the lecture. With some representations there will always be a tag or a header word in each heap object where the mark bit can be stored.
- ◆ They can be stored in a separate bitmap table:
 - ◆ If we have a **32-bit architecture** and the smallest heap object is **2 words**. (The three least significant bits == 0)
 - ◆ Then we can have **536,870,911** objects and need **67,108,863** bytes to store these bits.
 - ◆ This might seem to be a lot, but it is *only* **1.562%** of the total heap.

Tuning Mark & Sweep

- ◆ There is one problem with the mark phase:
 - ◆ While doing the depth first search we need to keep track of other paths to search.
 - ◆ If this is done with recursive calls we will need one **allocation record for each** level we descend in the reachability graph.
 - ◆ **Solutions:** Explicit stack or pointer reversal.

Mark & Sweep

- ◆ Advantages with mark & sweep:
 - ◆ Can reclaim cyclic structures.
 - ◆ Standard version is easy to implement.
 - ◆ Can have relatively low space overhead.
- ◆ Disadvantages:
 - ◆ Fragmentation can become a problem.
 - ◆ Allocation from a free-list can be costly.

Copying Collector

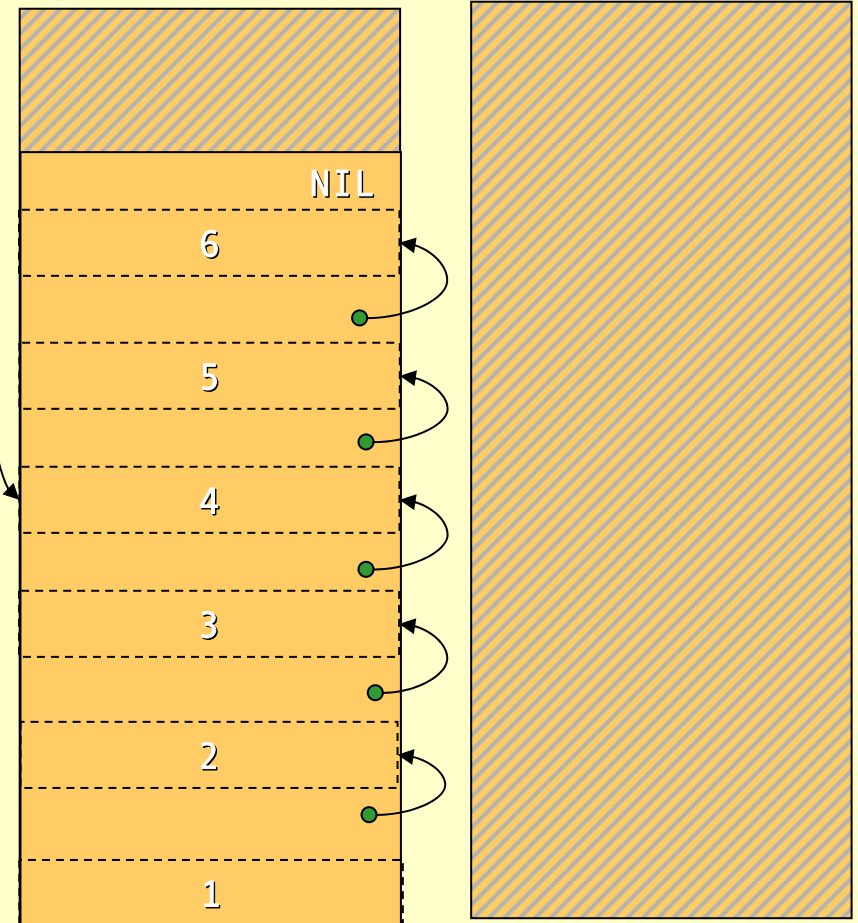
- ◆ The idea of a copying garbage collector is to divide the memory space in two parts.
- ◆ Allocation is done linearly in one part (*from-space*).
- ◆ When that part is full all reachable objects are copied to the other part (*to-space*).

Before GC

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```

from-space

to-space

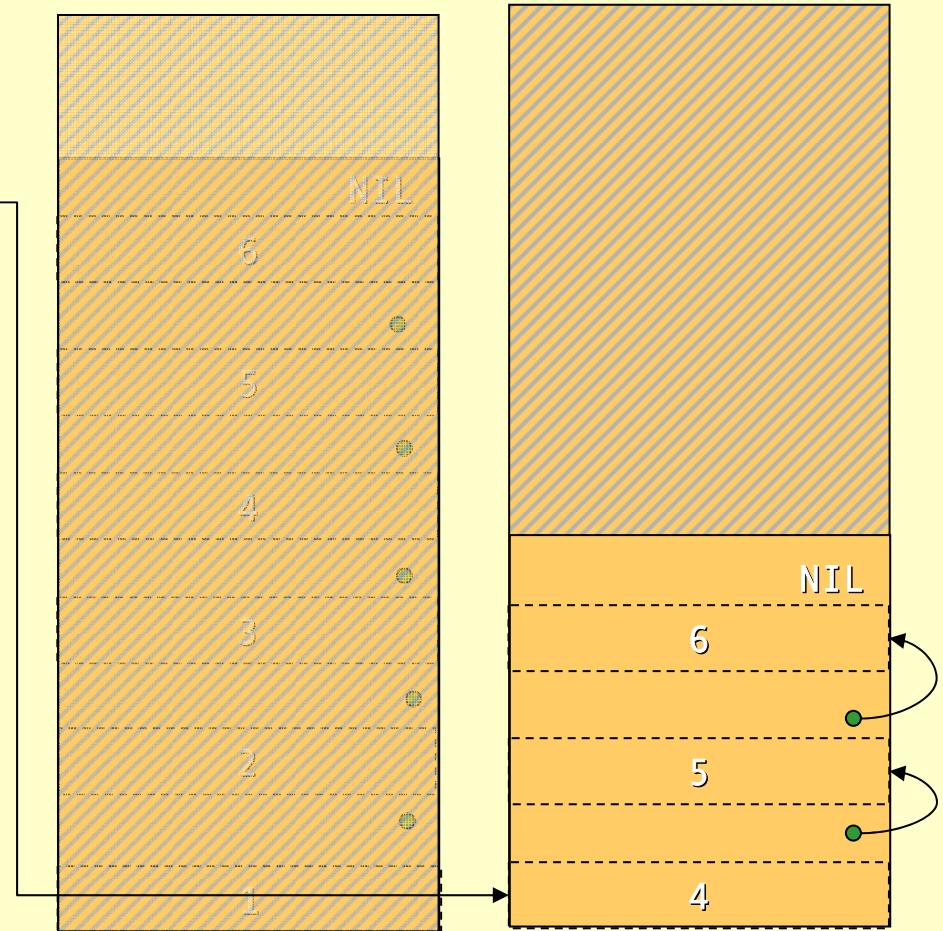


After GC

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```

to-space

from-space



Forwarding Pointers

- ◆ Given a pointer p that point to **from-space** make it point to **to-space**:
 - ◆ If p points to a from-space record that contains a pointer to to-space, then $*p$ is a forwarding-pointer that indicates where the copy is. Set $p = *p$.
 - ◆ If $*p$ has not been copied, copy $*p$ to location $next$, $*p = next$, $p = next$, $next += size(*p)$.

Cheney's Copying Collector

- ◆ Cheney's algorithm uses breadth-first to traverse the live data.
- ◆ The algorithm is non-recursive, requires no extra space or time consuming tricks (such as pointer reversal), and it is very simple to implement.
- ◆ The disadvantage is that breadth-first does not give as good locality of references as depth-first.

Cheney's Copying Collector

- ◆ The algorithm:
 1. Forward all roots.
 2. Use the area between `scan` and `next` as a queue for copied records whose children has yet not been forwarded.

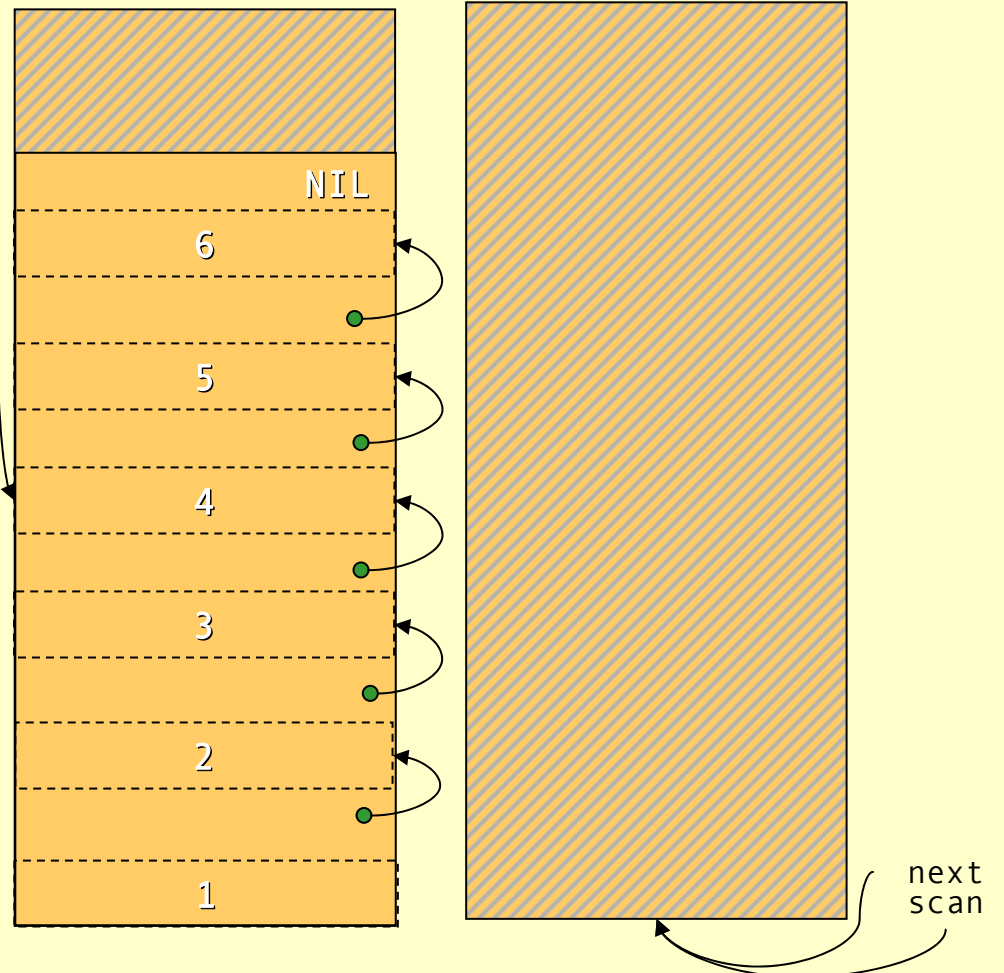
```
scan = next = start of to-space
for each root r { r = forward(r); }
while scan < next {
  for each field f of *scan
    scan->f = forward(scan->f)
  scan += size(*scan)
}
```

Before GC

```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
doLotsOfStuff();
return b;
```

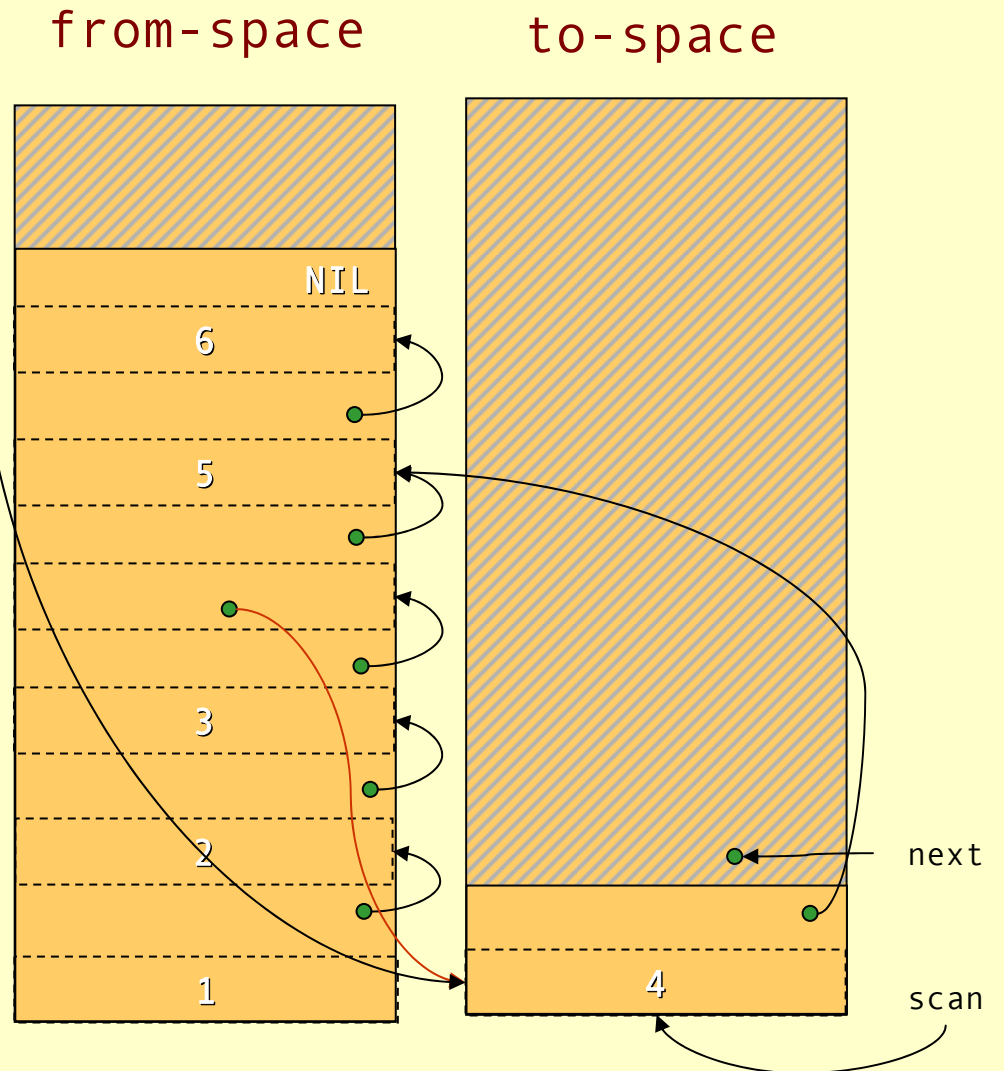
from-space

to-space



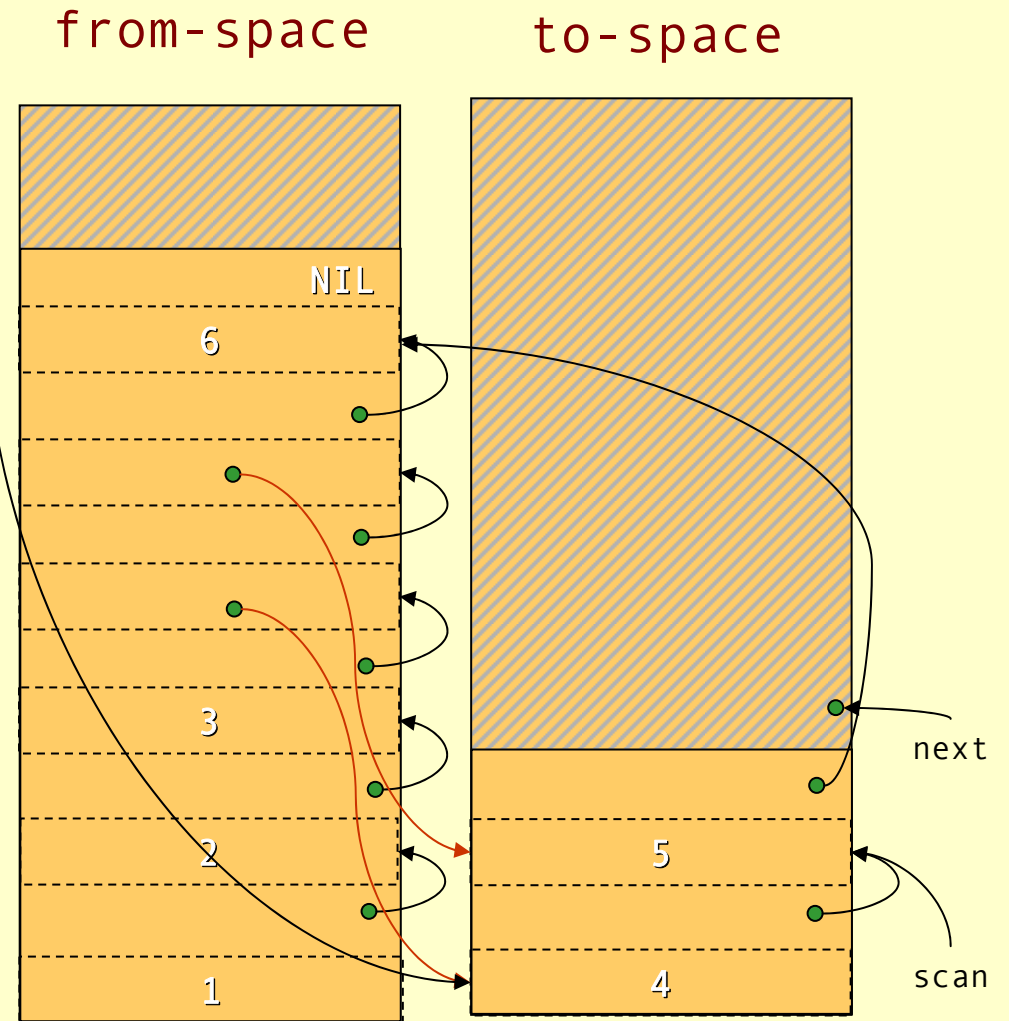
Forward Roots

```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
doLotsOfStuff();
return b;
```



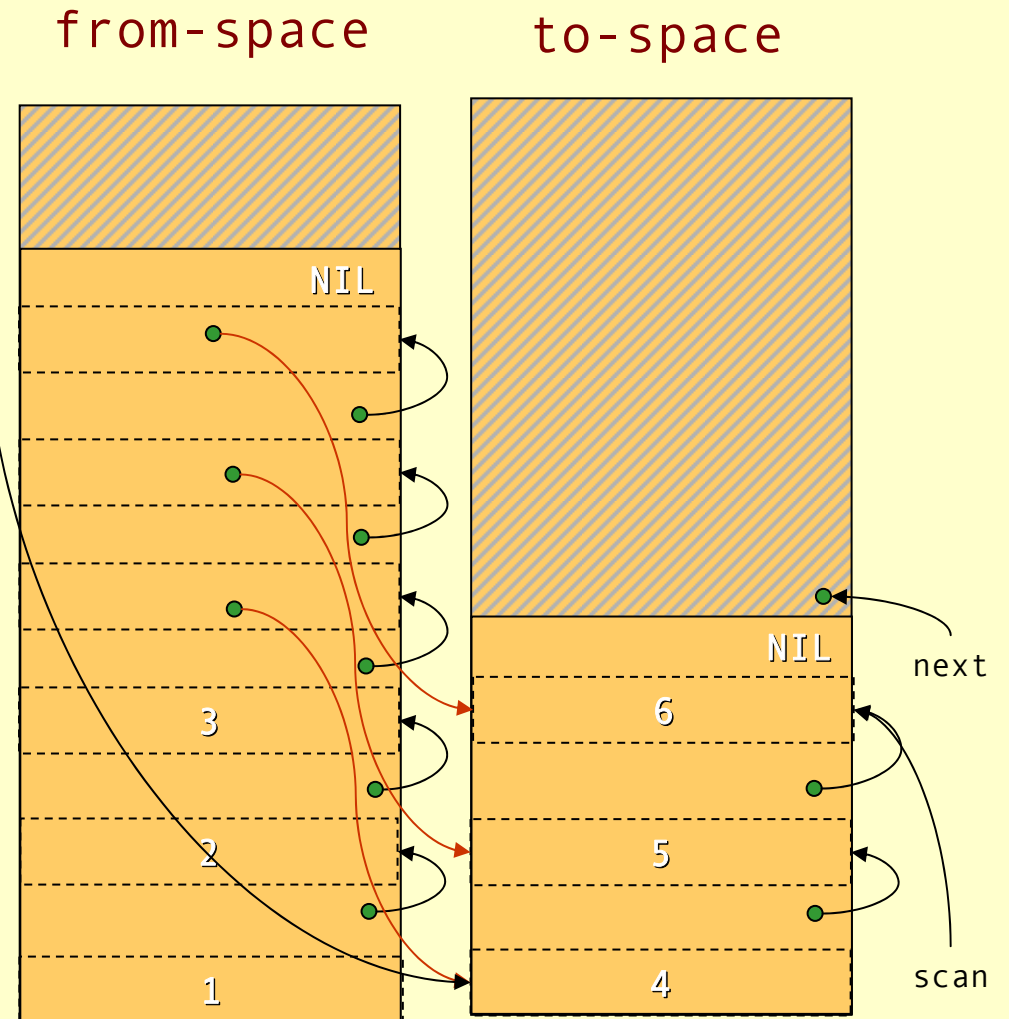
Scanning

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```



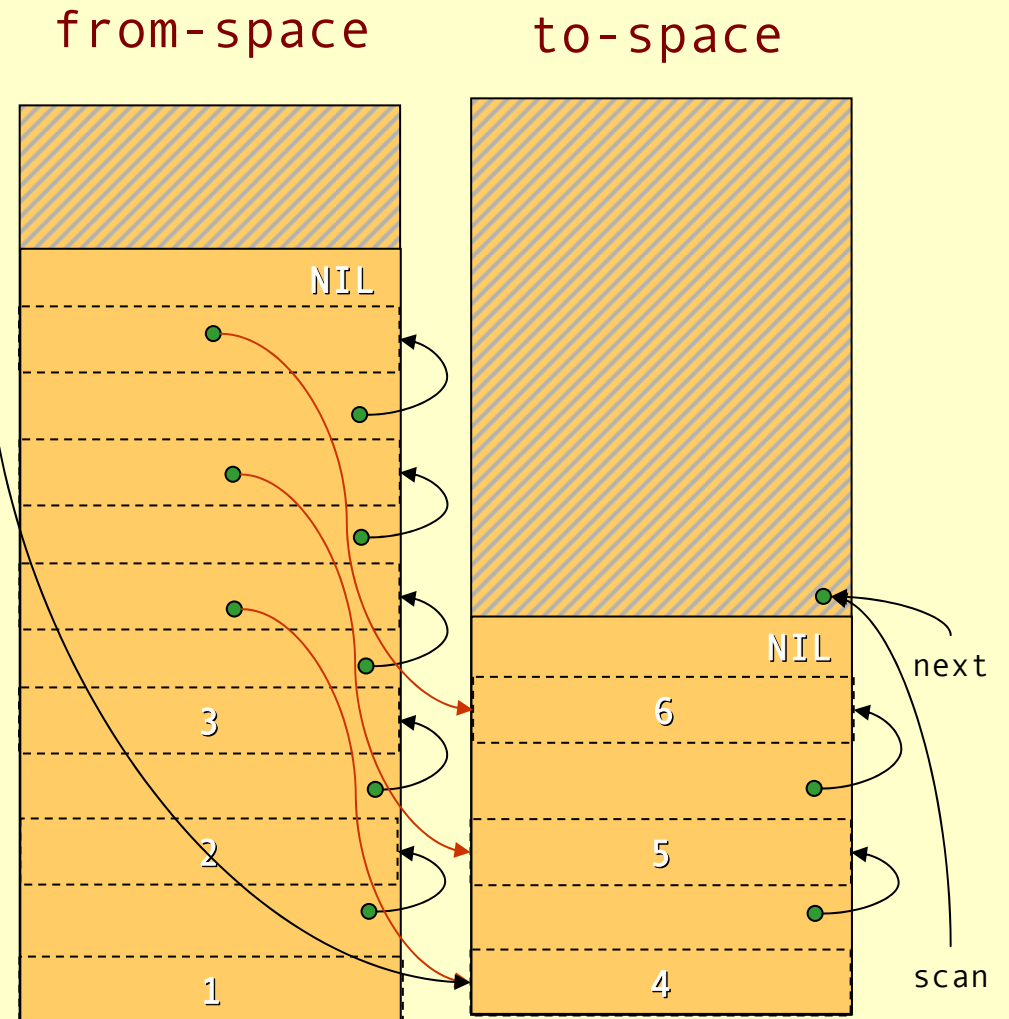
Scanning

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```



Scanning

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```

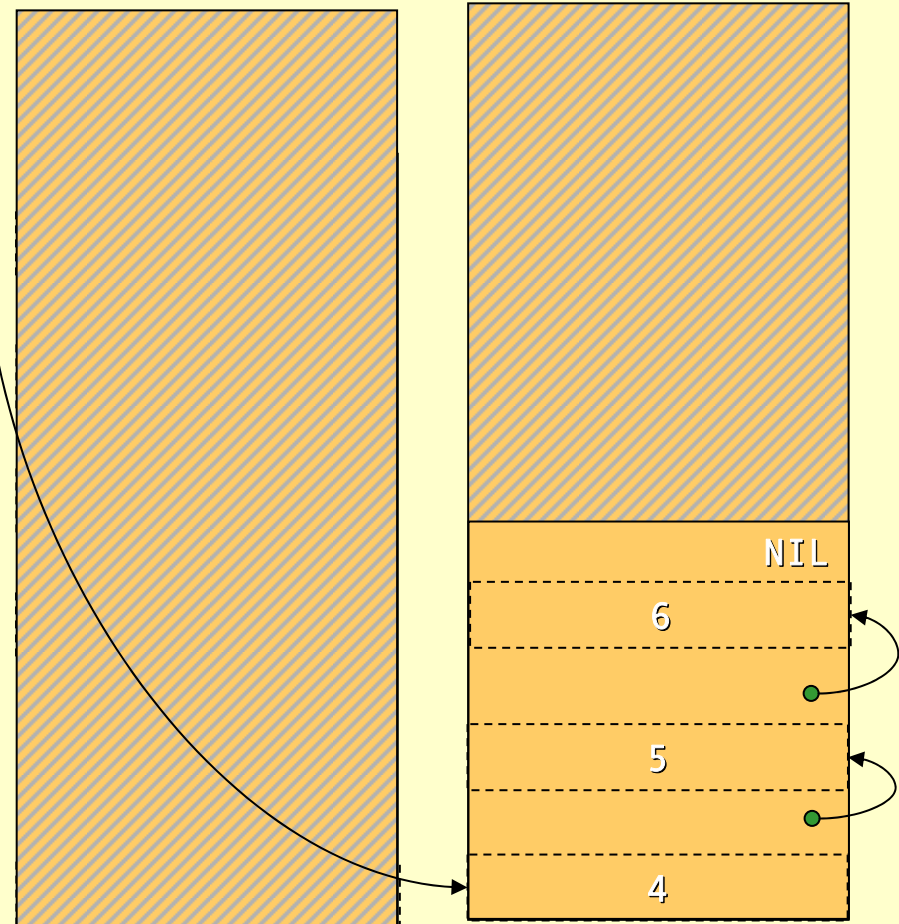


Scanning

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```

from-space

to-space



Cost of Copying GC

- ◆ The GC takes time proportional to the amount of reachable data (\mathbf{R}).
- ◆ The work done by the GC is to recover $\mathbf{H}/2 - \mathbf{R}$ words of memory.
- ◆ The *amortized cost* of GC (overhead/allocated word) is:

$$\frac{c_1 \mathbf{R}}{(\mathbf{H}/2) - \mathbf{R}}$$

- ◆ If \mathbf{H} is much larger than \mathbf{R} then the cost approaches zero.
- ◆ The GC is often self-tuning so that $\mathbf{H} = 4\mathbf{R}$ giving a GC cost of c_1 per allocated word.

Copying GC

- ◆ Advantages of copying GC:
 - ◆ Can handle cyclic structures.
 - ◆ Very easy to implement.
 - ◆ Extremely fast allocation (no free-list) just a check and heap pointer increment.
 - ◆ Automatic compaction: no fragmentation.
 - ◆ Only visits live data – time only proportional to live data.
- ◆ Disadvantages of copying GC:
 - ◆ Double the space overhead since two heaps are needed.
 - ◆ Long lived live data might be copied several times.
 - ◆ Copying all the live data might lead to long stop times.

Generational GC

- ◆ **Empirical observation:** most objects die young. The longer an object lives the higher the probability it will survive the next GC.
- ◆ The benefit of GC is highest for young objects.
- ◆ **Idea:** Keep young objects in a small space which is GC more often than the whole heap.
- ◆ With such a *generational GC* each collection takes less time and yields proportionally more space.

Generational GC

- ◆ In a generational GC we want to collect the younger generation without having to look at older generations.
- ◆ But we have to consider all pointers from older generations to younger generations as roots.
 - ◆ (In a language without destructive updates this is not a problem, since there are no such pointers.)
- ◆ These inter-generational references must be remembered (e.g., by keeping a *remembered set*). The compiler has to ensure that all store operations in an older generation are checked.

Cost of Generational GC

- ◆ It is common for the youngest generation to have **less than 10%** live data.
- ◆ With a copying collector $H/R = 10$ in this generation.
- ◆ The *amortized cost* of a *minor* collection is:

$$\frac{c_1 R}{(10 R) - R}$$

- ◆ Performing a major collection can be very expensive.
- ◆ Maintaining the remembered set also takes time. If a program does many updates of old objects with pointers to new objects a generational GC can be more expensive than a non-generational GC.

Incremental GC

- ◆ An *incremental* (or *concurrent*) GC keeps the stop-times down by interleaving GC with program execution.
 - ◆ The *collector* tries to free memory while the program, called the *mutator* changes the reachability graph.
- ◆ An incremental GC only operates at request from the mutator.
- ◆ A concurrent GC can operate in between any two mutator instructions.

Data Layout

- ◆ The compiler and the runtime system have to agree on a *data layout*. The GC needs to know the size of records, and which fields of a record contains pointers to other records.
- ◆ In statically typed or OO languages, each record can start with a *header word* that points to a description of the type or class.
- ◆ In many functional languages the set of data types can not be extended; for such languages one can use a *tagging scheme* where unused bits in a pointer indicate what data type it points to.
- ◆ Another approach is to not give any information to the collector about which fields are pointers. The collector must then make a *conservative guess*, and treat all words that looks like pointers to the heap as such. Since it is unsafe to change such pointers a *conservative collector* has to be non-moving.

The Root Set

- ◆ The set of registers and stack slots that contain live data can be described by a *pointer map* (*stack map*).
- ◆ For each pointer that is live after a function call the pointer map identifies its register or stack slot.
- ◆ The *return address* can be used as a key in a hash map to find the pointer map.
- ◆ To mark/forward the roots the GC starts at the top of the stack and scans downwards frame by frame. (In a generational collector the stack scan can also be made generational.)

Finalizers

- ◆ Some languages (notably OO) have *finalizers*, that is, some code that should be executed before some data is deallocated.
- ◆ This is, e.g., useful to make sure that an object frees all resources (open files, locks, etc) before dying.
- ◆ With a **copying collector** the handling of finalizers becomes more difficult. Such a GC does not normally visit the dead data. So all finalizers have to be remembered and after GC a check has to be done to see if any freed data triggers a finalizer.
- ◆ A **mark & sweep** collector does not have this problem, but just as with a copying collector it might take a long time after the last use before garbage is actually collected.
- ◆ If one wants to ensure that a finalizer is executed as soon as the object dies then one has to use **reference counting**.

Summary

- ◆ Manual allocation is unsafe and should not be used. (It also comes at a cost, maintaining a free-list is not for free.)
- ◆ Garbage collection solves the problem of automatic memory management.
- ◆ In most cases a generational copying collector will be the most efficient solution.

Lazy Code Motion

This lecture is primarily based on Konstantinos Sagonas set of slides
(Advanced Compiler Techniques, (2AD518)
at Uppsala University, January-February 2004).
Used with kind permission.
(In turn based on Keith Cooper's slides)

Lazy Code Motion

The concept

- ◆ Solve data-flow problems that reveal limits of code motion
- ◆ Compute **INSERT** & **DELETE** sets from solutions
- ◆ Linear pass over the code to rewrite it (using **INSERT** & **DELETE**)

The history

- ◆ Partial redundancy elimination (Morel & Renvoise, CACM, 1979)
- ◆ Improvements by Drechsler & Stadel, Joshi & Dhamdhere, Chow, Knoop, Ruthing & Steffen, Dhamdhere, Sorkin, ...
- ◆ All versions of PRE optimize placement
 - ◆ Guarantee that no path is lengthened
- ◆ LCM was invented by Knoop et al. in PLDI, 1992
- ◆ We will look at a variation by Drechsler & Stadel

SIGPLAN Notices,
28(5), May, 1993

Lazy Code Motion

The intuitions

- ◆ Compute *available expressions*
- ◆ Compute *anticipable expressions*
- ◆ These lead to an earliest placement for each expression
- ◆ Push expressions down the CFG until it changes behavior

Assumptions

- ◆ Uses a lexical notion of identity (*not value identity*)
- ◆ Code is in an Intermediate Representation with unlimited name space
- ◆ Consistent, disciplined use of names
 - ◆ Identical expressions define the same name
 - ◆ No other expression defines that name

} Avoids copies
} Result serves as proxy

Lazy Code Motion

The Name Space

- ◆ $r_i + r_j \rightarrow r_k$, always *(hash to find k)*
- ◆ We can refer to $r_i + r_j$ by r_k *(bit-vector sets)*
- ◆ Variables must be set by copies
 - ◆ No consistent definition for a variable
 - ◆ Break the rule for this case, but require $r_{source} < r_{destination}$
 - ◆ To achieve this, assign register names to variables first

Without this name space

- ◆ LCM must insert copies to preserve redundant values
- ◆ LCM must compute its own map of expressions to unique ids

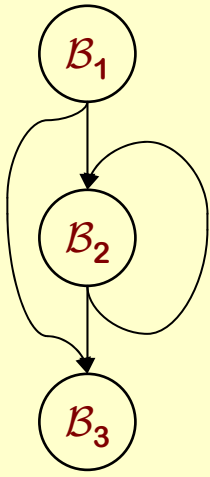
Lazy Code Motion: Running Example

```

B1:
  r1 ← 1
  r2 ← r1
  r3 ← r0 + @m
  r4 ← r3
  r5 ← (r1 < r2)
  if r5 then B2 else B3
B2:
  r20 ← r17 * r18
  r21 ← r19 + r20
  r8 ← r21
  r6 ← r2 + 1
  r2 ← r6
  r7 ← (r2 > r4)
  if r7 then B3 else B2
B3: ...
    
```

Variables:
 r_2, r_4, r_8

Expressions:
 $r_1, r_3, r_5, r_6, r_7, r_{20}, r_{21}$



Lazy Code Motion

Predicates (computed by Local Analysis)

- ◆ **DEEXPR**(b) contains expressions defined in b that survive to the end of b .
 $e \in \text{DEEXPR}(b) \Rightarrow$ evaluating e at the end of b produces the same value for e as evaluating it in its original position.
- ◆ **UEEXPR**(b) contains expressions defined in b that have upward exposed arguments (both args).
 $e \in \text{UEEXPR}(b) \Rightarrow$ evaluating e at the start of b produces the same value for e as evaluating it in its original position.
- ◆ **KILLEDEXPR**(b) contains those expressions whose arguments are (re)defined in b .
 $e \in \text{KILLEDEXPR}(b) \Rightarrow$ evaluating e at the start of b does not produce the same result as evaluating it at its end.

Lazy Code Motion: Running Example

```

B1:
  r1 ← 1
  r2 ← r1
  r3 ← r0 + @m
  r4 ← r3
  r5 ← (r1 < r2)
  if r5 then B2 else B3

B2:
  r20 ← r17 * r18
  r21 ← r19 + r20
  r8 ← r21
  r6 ← r2 + 1
  r2 ← r6
  r7 ← (r2 > r4)
  if r7 then B3 else B2

B3: ...
    
```

Variables:
 r₂, r₄, r₈

Expressions:
 r₁, r₃, r₅, r₆, r₇, r₂₀, r₂₁

	B ₁	B ₂	B ₃
DEEXPR	r ₁ , r ₃ , r ₅	r ₇ , r ₂₀ , r ₂₁	
UEEXPR	r ₁ , r ₃	r ₆ , r ₂₀	
KILLEDEXPR	r ₅ , r ₆ , r ₇	r ₅ , r ₆ , r ₇ , r ₂₁	

Lazy Code Motion

Availability

$$\text{AVAILIN}(n) = \bigcap_{m \in \text{preds}(n)} \text{AVAILOUT}(m), \quad n \neq n_0$$

$$\text{AVAILOUT}(m) = \text{DEEXPR}(m) \cup (\text{AVAILIN}(m) \cap \overline{\text{KILLEDEXPR}(m)})$$

Initialize **AVAILIN**(*n*) to the set of all names, except at *n*₀

Set **AVAILIN**(*n*₀) to ∅

Interpreting **AVAIL**

- ◆ $e \in \text{AVAILOUT}(b) \Leftrightarrow$ evaluating *e* at end of *b* produces the same value for *e*. **AVAILOUT** tells the compiler how far forward *e* can move the evaluation of *e*, ignoring any uses of *e*.
- ◆ This differs from the way we talk about **AVAIL** in global redundancy elimination.

Lazy Code Motion

Anticipability

$$\mathbf{ANTOUT}(n) = \bigcap_{m \in \text{succs}(n)} \mathbf{ANTIN}(m), \quad n \text{ not an exit block}$$

$$\mathbf{ANTIN}(m) = \mathbf{UEEXPR}(m) \cup (\mathbf{ANTOUT}(m) \cap \overline{\mathbf{KILLEDEXPR}(m)})$$

Initialize $\mathbf{ANTOUT}(n)$ to the set of all names, except at exit blocks

Set $\mathbf{ANTOUT}(n)$ to \emptyset , for each exit block n

Interpreting \mathbf{ANTOUT}

- ◆ $e \in \mathbf{ANTIN}(b) \Leftrightarrow$ evaluating e at start of b produces the same value for e . \mathbf{ANTIN} tells the compiler how far backward e can move
- ◆ This view shows that anticipability is, in some sense, the inverse of availability (& explains the new interpretation of \mathbf{AVAIL}).

Lazy Code Motion

Earliest placement

$$\text{EARLIEST}(i,j) = \text{ANTIN}(j) \cap \overline{\text{AVAILOUT}(i)} \cap (\text{KILLEDEXPR}(i) \cup \overline{\text{ANTOUT}(i)})$$

$$\text{EARLIEST}(n_0,j) = \text{ANTIN}(j) \cap \overline{\text{AVAILOUT}(n_0)}$$

EARLIEST is a predicate

- ◆ Computed for edges rather than nodes (*placement*)
- ◆ $e \in \text{EARLIEST}(i,j)$ if
 - ◆ It can move to head of j ,
 - ◆ It is not available at the end of i , and
 - ◆ either it cannot move to the head of i ($\text{KILLEDEXPR}(i)$)
 - ◆ or another edge leaving i prevents its placement in i ($\overline{\text{ANTOUT}(i)}$)

Lazy Code Motion

Later (than earliest) placement

$$\text{LATERIN}(j) = \bigcap_{i \in \text{preds}(j)} \text{LATER}(i,j), \quad j \neq n_0$$

$$\text{LATER}(i,j) = \text{EARLIEST}(i,j) \cup (\text{LATERIN}(i) \cap \overline{\text{UEEXPR}(i)})$$

Initialize $\text{LATERIN}(n_0)$ to \emptyset

$x \in \text{LATERIN}(k) \Leftrightarrow$ every path that reaches k has $x \in \text{EARLIEST}(m)$ for some block m , and the path from m to k is x -clear & does not evaluate x .

\Rightarrow the compiler can move x through k without losing any benefit.

$x \in \text{LATER}(i,j) \Leftrightarrow \langle i,j \rangle$ is its earliest placement, or it can be moved forward from i ($\text{LATER}(i)$) and placement at entry to i does not anticipate a use in i (*moving it across the edge exposes that use*).

Lazy Code Motion

Rewriting the code

$$\text{INSERT}(i,j) = \text{LATER}(i,j) \cap \overline{\text{LATERIN}(j)}$$

$$\text{DELETE}(k) = \text{UEEXPR}(k) \cap \overline{\text{LATERIN}(k)}, k \neq n_0$$

INSERT & **DELETE** are predicates

Compiler uses them to guide the rewrite step

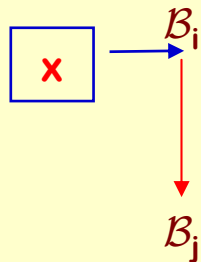
- ◆ $x \in \text{INSERT}(i,j) \Rightarrow$ insert x at start of i , end of j , or new block
- ◆ $x \in \text{DELETE}(k) \Rightarrow$ delete first evaluation of x in k

If local redundancy elimination has already been performed, only one copy of x exists. Otherwise, remove all upward exposed copies of x .

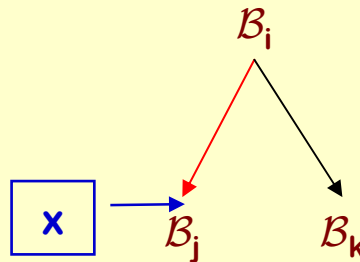
Lazy Code Motion

Edge placement

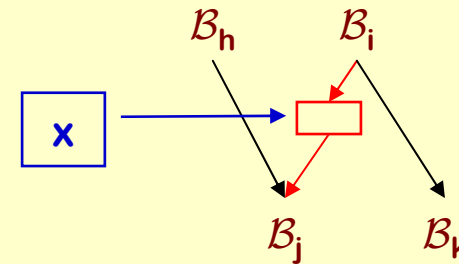
- ◆ $x \in \text{INSERT}(i,j)$



$$|\text{succs}(i)| = 1$$



$$|\text{preds}(j)| = 1$$



$$|\text{succs}(i)| > 1$$

$$|\text{preds}(j)| > 1$$

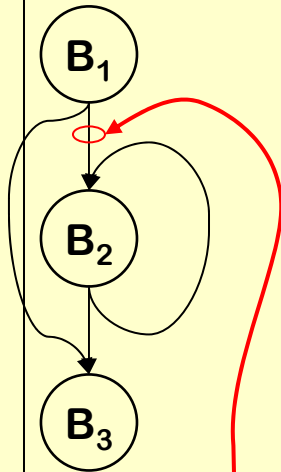
Three cases

- ◆ $|\text{succs}(i)| = 1 \Rightarrow$ insert x at end of i .
- ◆ $|\text{succs}(i)| > 1$ but $|\text{preds}(j)| = 1 \Rightarrow$ insert x at start of j .
- ◆ $|\text{succs}(i)| > 1$ and $|\text{preds}(j)| > 1 \Rightarrow$ create new block in $\langle i,j \rangle$ for x .

Lazy Code Motion Example

```

B1: r1 ← 1
      r2 ← r1
      r3 ← r0 + @m
      r4 ← r3
      r5 ← (r1 < r2)
      if r5 then B2 else B3
B2: r20 ← r17 * r18
      r21 ← r19 + r20
      r8 ← r21
      r6 ← r2 + 1
      r2 ← r6
      r7 ← (r2 > r4)
      if r7 then B3 else B2
B3: ...
    
```



	B1	B2	B3
DEEXPR	r1, r3, r5	r7, r20, r21	
UEEXPR	r1, r3	r6, r20	
KILLEEXPR	r5, r6, r7	r5, r6, r7, r21	

	B1	B2	B3
AVAILIN	{}	r1, r3	r1, r3
AVAILOUT	r1, r3, r5	r1, r3, r7, r20, r21	...
ANTIN	r1, r3	r6, r20	{}
ANTOUT	{}	{}	{}

	1,2	1,3	2,2	2,3
EARLIEST	r20, r21	{}	{}	{}

Example is too small to show off **LATER**

INSERT(1,2) = { r₂₀, r₂₁ }

DELETE(2) = { r₂₀, r₂₁ }

Register Allocation

This lecture is primarily based on Konstantinos Sagonas set of slides
(Advanced Compiler Techniques, (2AD518)
at Uppsala University, January-February 2004).
Used with kind permission.

Register Allocation

- ◆ What is register allocation?
- ◆ Different types of register allocators.
- ◆ Webs.
- ◆ Interference Graphs.
- ◆ Graph coloring.
- ◆ Spilling.
- ◆ Live-Range Splitting.
- ◆ More optimizations.

Storing values between defs and uses

- ◆ Program computes with values
 - ◆ value definitions (where computed)
 - ◆ value uses (where read to compute new values)
- ◆ Values must be stored between def and use

First Option:

- ◆ store each value in memory at definition
- ◆ retrieve from memory at each use

Second Option:

- ◆ store each value in register at definition
- ◆ retrieve value from register at each use

Issues

- ◆ On a typical RISC architecture:
 - ◆ All computation takes place in registers.
 - ◆ Load instructions and store instructions transfer values between memory and registers.
- ◆ Add two numbers; values in memory:
 - load r1, 4(sp)
 - load r2, 8(sp)
 - add r3,r1,r2
 - store r3, 12(sp)

Issues

- ◆ On a typical RISC architecture
 - ◆ All computation takes place in registers
 - ◆ Load instructions and store instructions transfer values between memory and registers
- ◆ Add two numbers; values in registers:

```
add r3,r1,r2
```

Issues

- ◆ Fewer instructions when using registers.
 - ◆ Most instructions are register-to-register.
 - ◆ Additional instructions for memory accesses.
- ◆ Registers are faster than memory.
 - ◆ Wider gap in faster, newer processors.
 - ◆ Factor of about 4 bandwidth, factor of about 3 latency.
 - ◆ Could be bigger depending on program characteristics.
- ◆ But only a small number of registers available.
 - ◆ Usually 32 integer and 32 floating-point registers.
 - ◆ Some of those registers have fixed users (r0, ra, sp, fp).

Register Allocation

- ◆ Deciding which values to store in a limited number of registers.
- ◆ Register allocation has a direct impact on performance.
 - ◆ Affects almost every statement of the program.
 - ◆ Eliminates expensive memory instructions.
 - ◆ # of instructions goes down due to direct manipulation of registers (no need for load and store instructions).
 - ◆ This is probably the optimization with the most impact!

What can be put in a register?

- ◆ Values stored in compiler-generated temps.
- ◆ Language-level values:
 - ◆ Values stored in local scalar variables.
 - ◆ Big constants.
 - ◆ Values stored in array elements and object fields
 - ◆ Issue: **alias analysis**
- ◆ Register set depends on the data-type:
 - ◆ floating-point values in floating point registers.
 - ◆ integer and pointer values in integer registers.

Allocation vs Assignment?

- ◆ We sometimes distinguish between register allocation and register assignment.
- ◆ Register allocation deals with the problem to decide which values to store in registers and which to spill to memory.
- ◆ Register assignment decides which value goes into which register.

Different Types of Register Allocation

- ◆ Local Register allocation.
 - ◆ Tree-based approaches:
 - ◆ Sethi-Ullman numbering.
 - ◆ Basic Block.
- ◆ Global Register allocation.
 - ◆ Linear Scan.
 - ◆ Graph Coloring.
- ◆ Inter-procedural allocation.

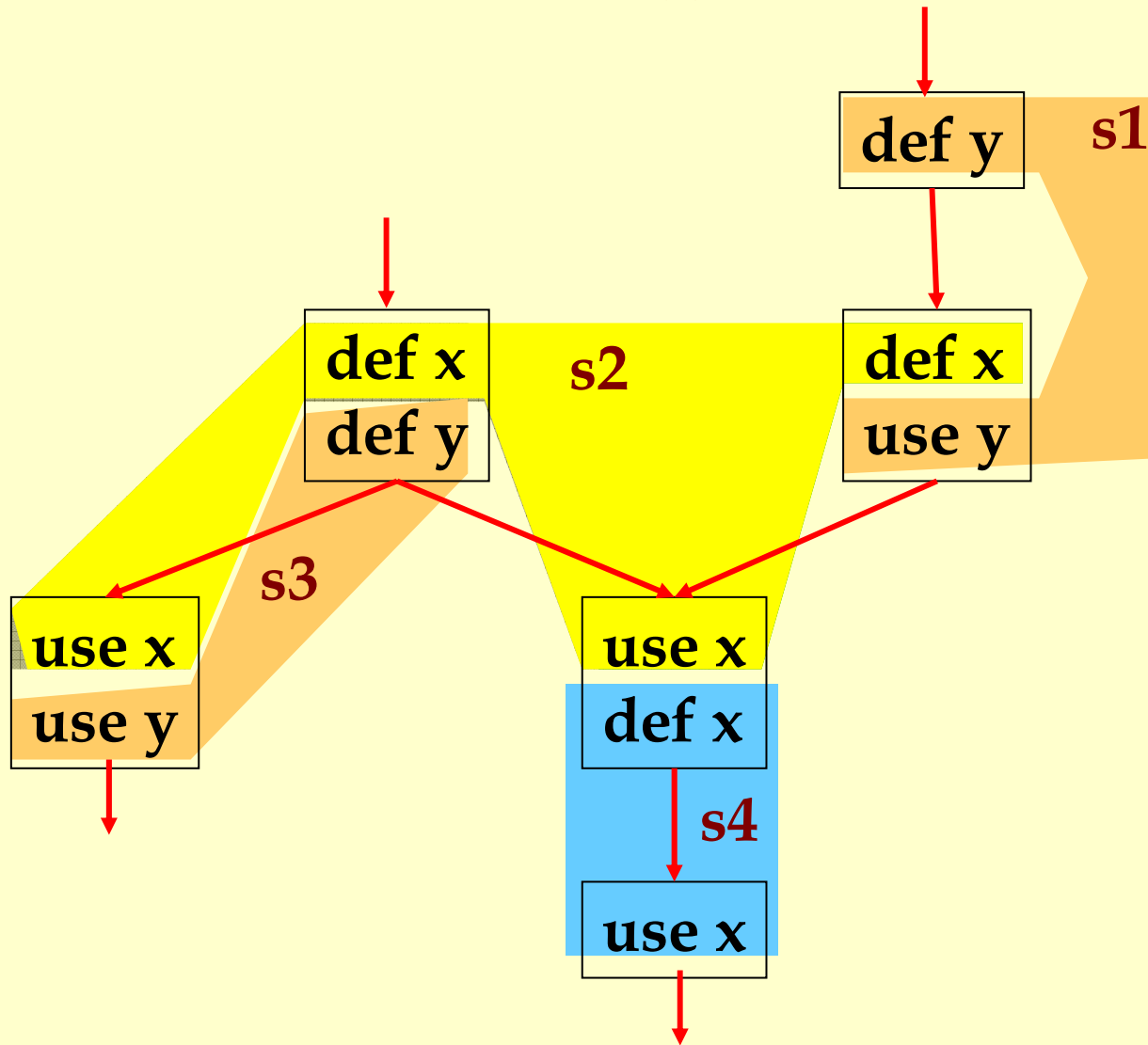
Web-Based Register Allocation

- ◆ Determine live ranges for each value (*web*).
- ◆ Determine overlapping ranges (interference).
- ◆ Compute the benefit of keeping each web in a register (spill cost).
- ◆ Decide which webs get a register (allocation).
- ◆ Split webs if needed (spilling and splitting).
- ◆ Assign hard registers to webs (assignment).
- ◆ Generate code including spills (code gen.).

Webs

- ◆ Starting Point: def-use chains (DU chains).
 - ◆ Connects definition to all reachable uses.
- ◆ Conditions for putting defs and uses into same web:
 - ◆ Def and all reachable uses must be in same web.
 - ◆ All defs that reach same use must be in same web.
- ◆ Use a union-find algorithm.

Example



Webs

- ◆ Web is unit of register allocation.
- ◆ If web allocated to a given register R:
 - ◆ All definitions computed into R.
 - ◆ All uses read from R.
- ◆ If web allocated to a memory location M:
 - ◆ All definitions computed into M.
 - ◆ All uses read from M.
- ◆ Issue: instructions compute only from registers.
- ◆ Reserve some registers to hold memory values.

Convex Sets and Live Ranges

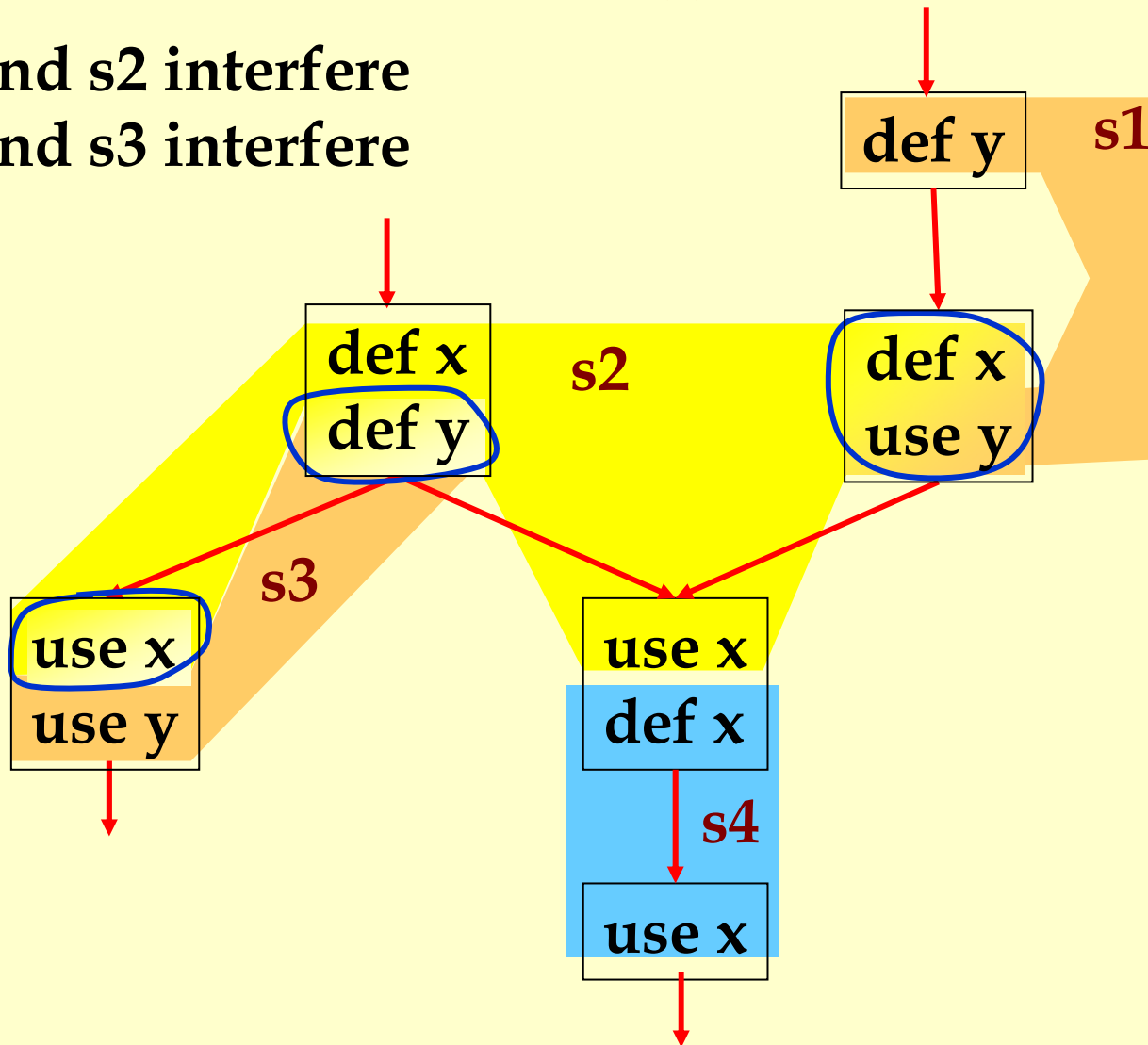
- ◆ Concept of convex set.
- ◆ A set S is **convex** if
 - ◆ $a, b \in S$ and c is on a path from a to b implies $c \in S$
- ◆ Concept of **live range** of a web.
 - ◆ Minimal convex set of instructions that includes all defs and uses in web.
 - ◆ Intuitively, region in which web's value is live.

Interference

- ◆ Two webs **interfere** if their live ranges overlap (have a nonempty intersection).
- ◆ If two webs interfere, values must be stored in different registers or memory locations.
- ◆ If two webs do not interfere, can store values in same register or memory location.

Example

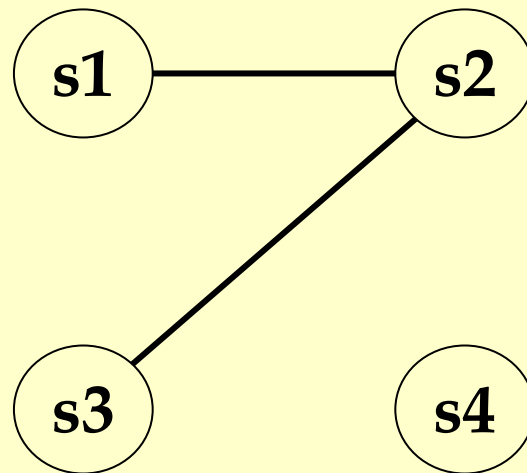
Webs s1 and s2 interfere
 Webs s2 and s3 interfere



Interference Graph

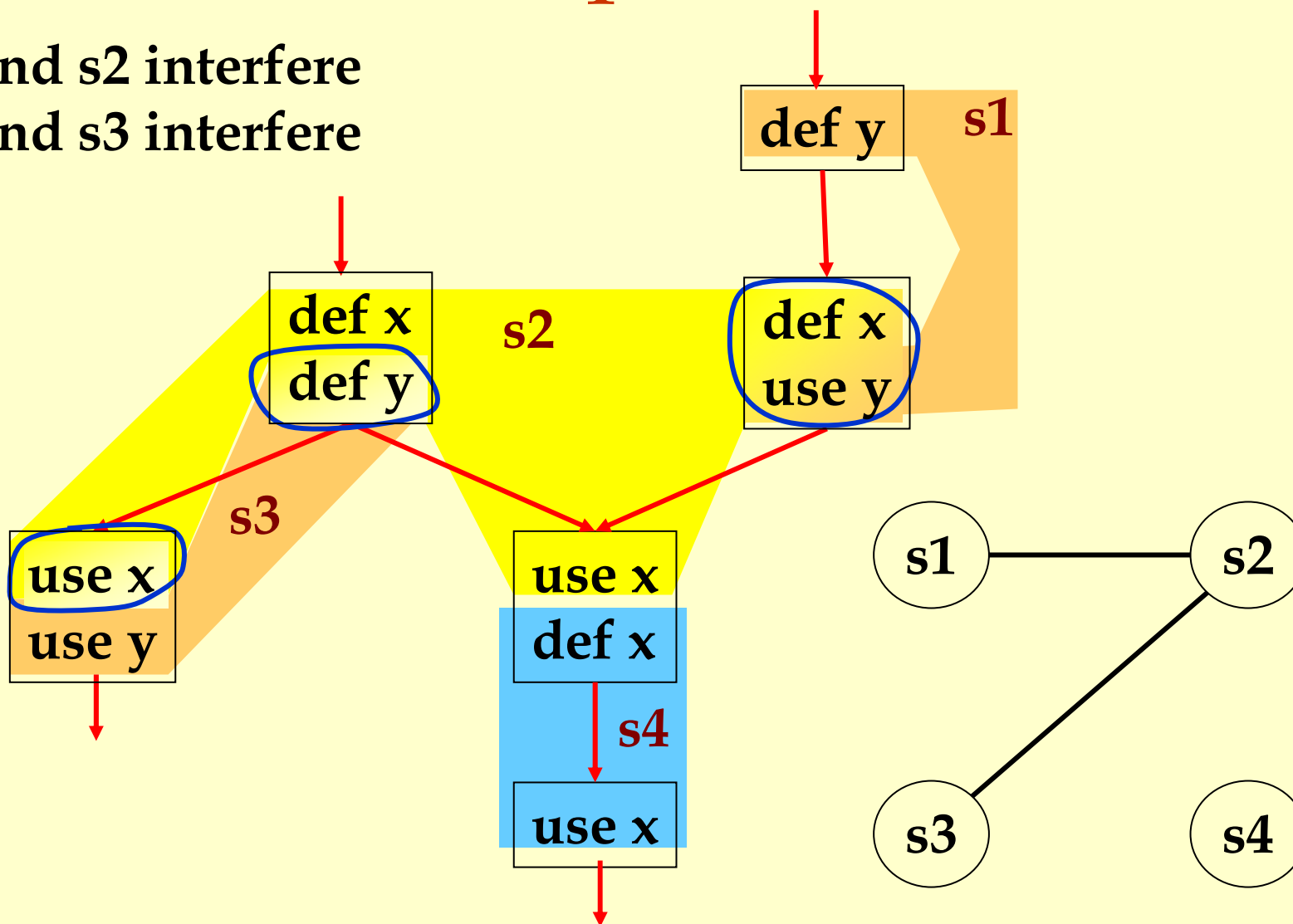
Representation of webs and their interference:

- ◆ Nodes are the webs
- ◆ An edge exists between two nodes if they interfere:



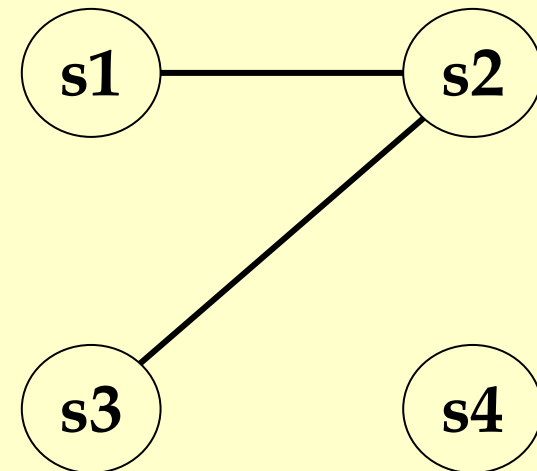
Example

Webs s1 and s2 interfere
 Webs s2 and s3 interfere



Register Allocation Using Graph Coloring

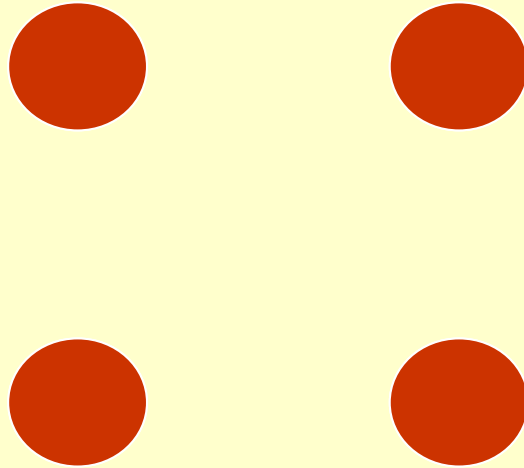
- ◆ Each web is allocated to a register.
 - ◆ Each node gets a register (color).
- ◆ If two webs interfere they cannot use the same register.
 - ◆ If two nodes have an edge between them, they cannot have the same color.



Graph Coloring

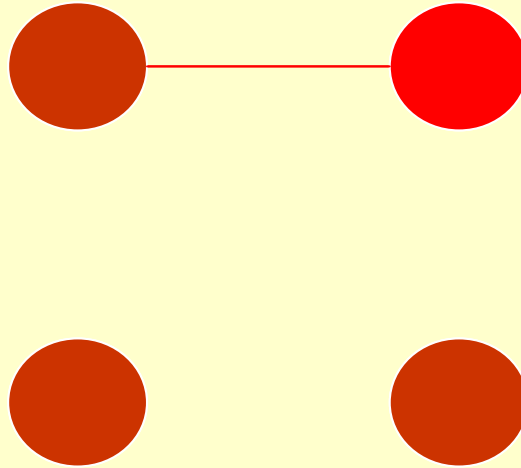
- ◆ Assign a color to each node in the graph.
- ◆ Two nodes connected to same edge must have different colors.
- ◆ Classic problem in graph theory.
- ◆ NP complete.
 - ◆ But good heuristics exist for register allocation.

Graph Coloring Example



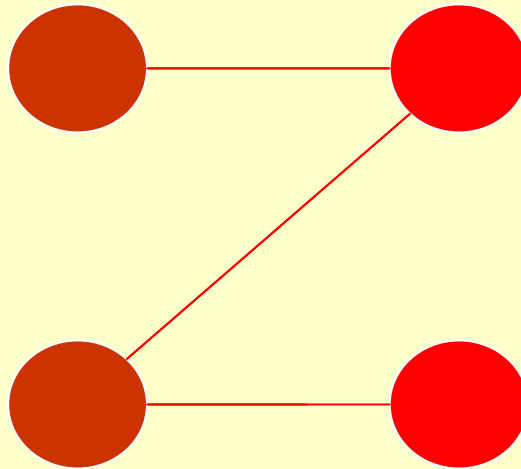
1 Color

Graph Coloring Example



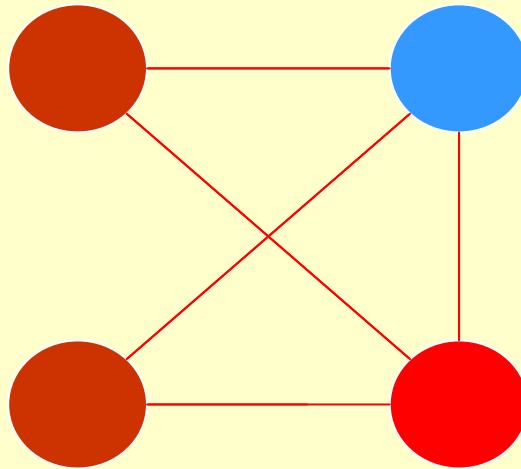
2 Colors

Graph Coloring Example



Still 2 Colors

Graph Coloring Example



3 Colors

Heuristics for Register Coloring

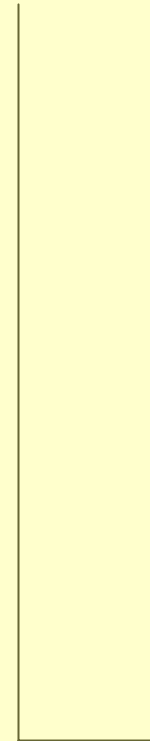
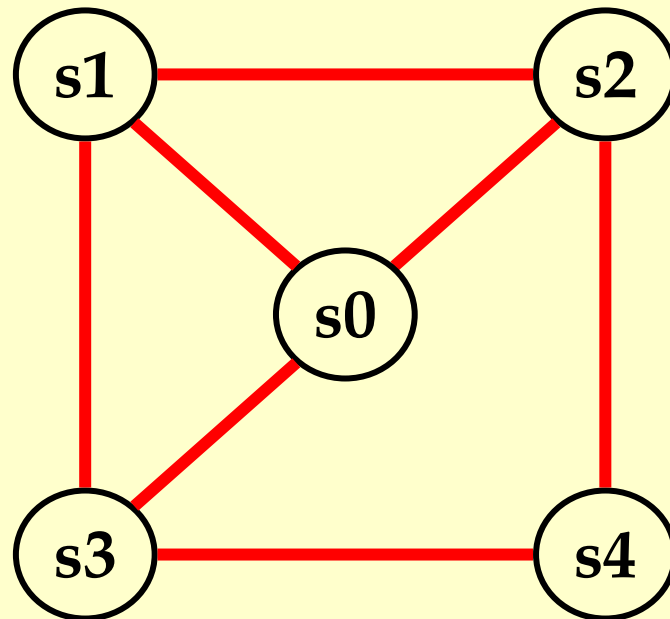
- ◆ Coloring a graph with N colors.
- ◆ If $\text{degree} < N$ (degree of a node = # of edges):
 - ◆ Node can always be colored.
 - ◆ After coloring the rest of the nodes, there is at least one color left to color the current node.
- ◆ If $\text{degree} \geq N$:
 - ◆ Still may be colorable with N colors. (If some neighbors are colored with the same color.)

Heuristics for Register Coloring

- ◆ Remove nodes that have degree $< N$.
 - ◆ Push the removed nodes onto a stack.
- ◆ When all the nodes have degree $\geq N$:
 - ◆ Find a node to spill (no color for that node).
 - ◆ Push that node into the stack.
- ◆ When empty, start to color:
 - ◆ Pop a node from stack back.
 - ◆ Assign it a color that is different from its connected nodes (if possible).

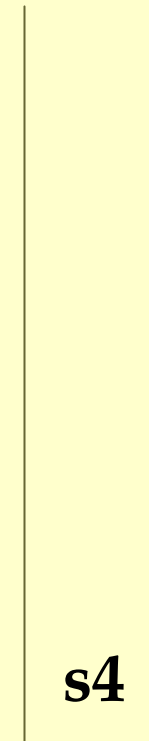
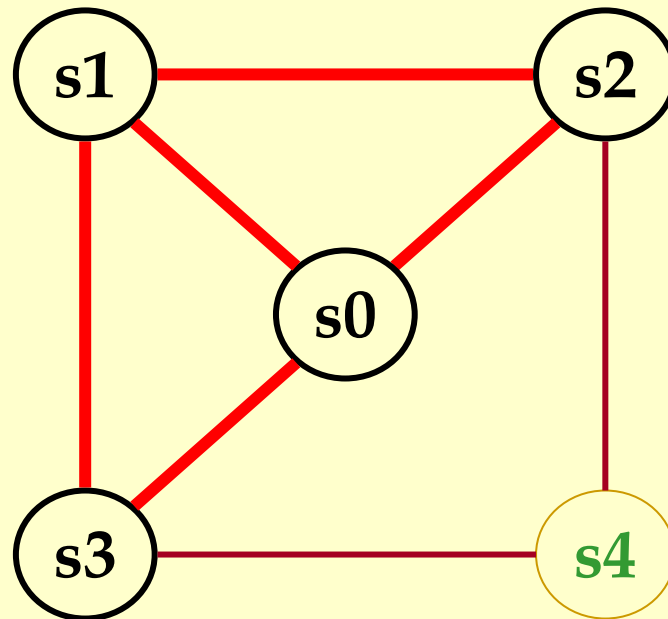
Coloring Example

$N = 3$



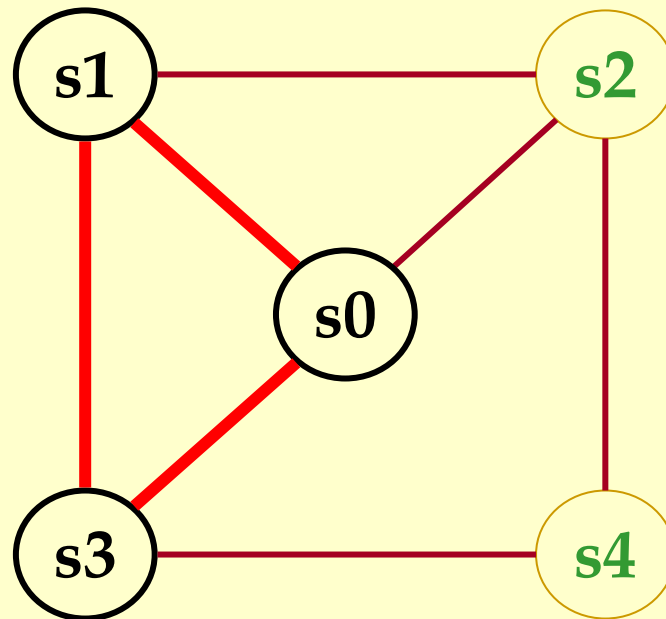
Coloring Example

$N = 3$



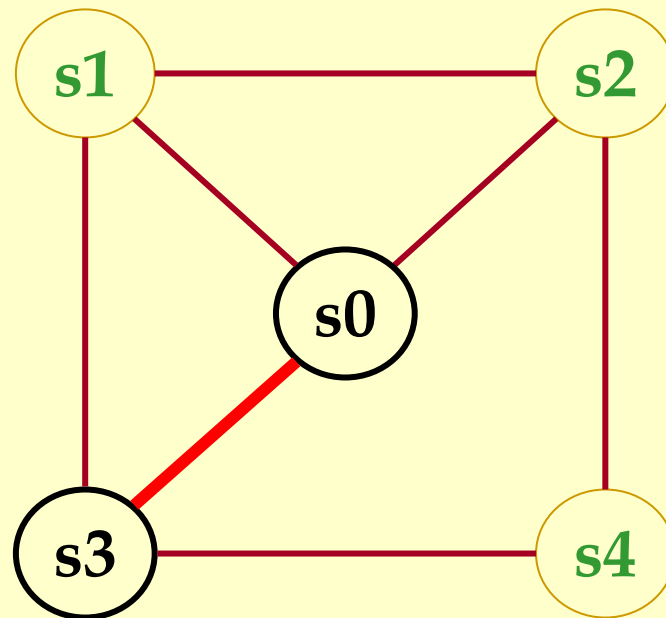
Coloring Example

$N = 3$



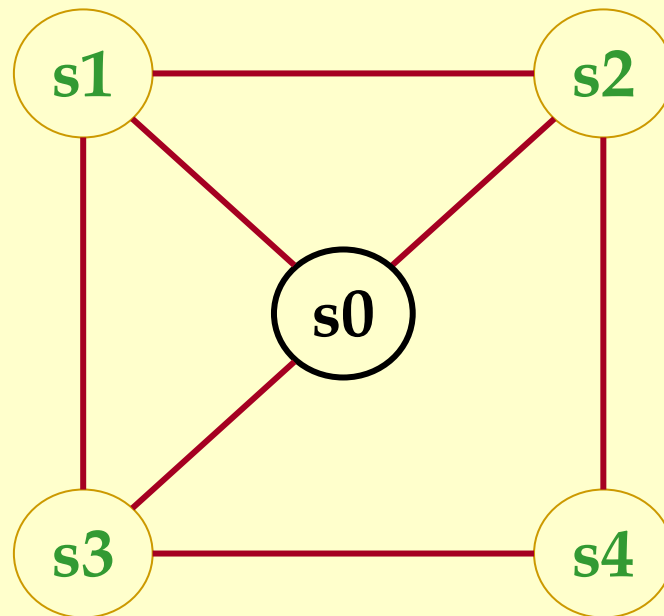
Coloring Example

$N = 3$



Coloring Example

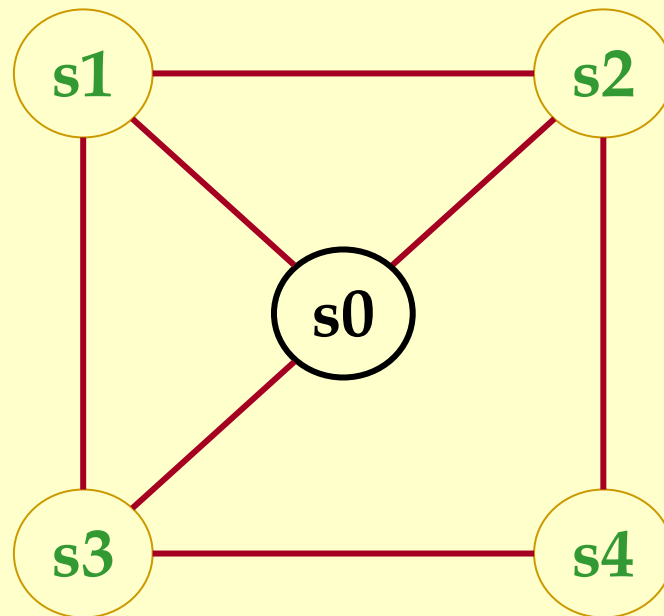
$N = 3$



- s3
- s1
- s2
- s4

Coloring Example

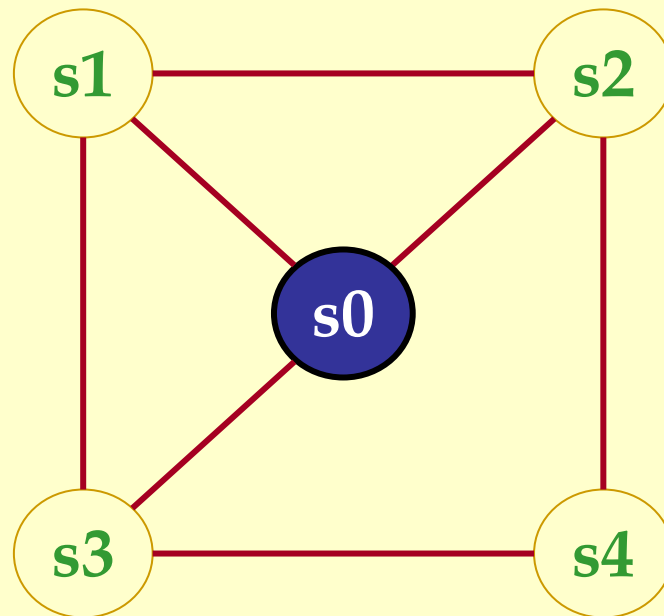
$N = 3$   



s3
s1
s2
s4

Coloring Example

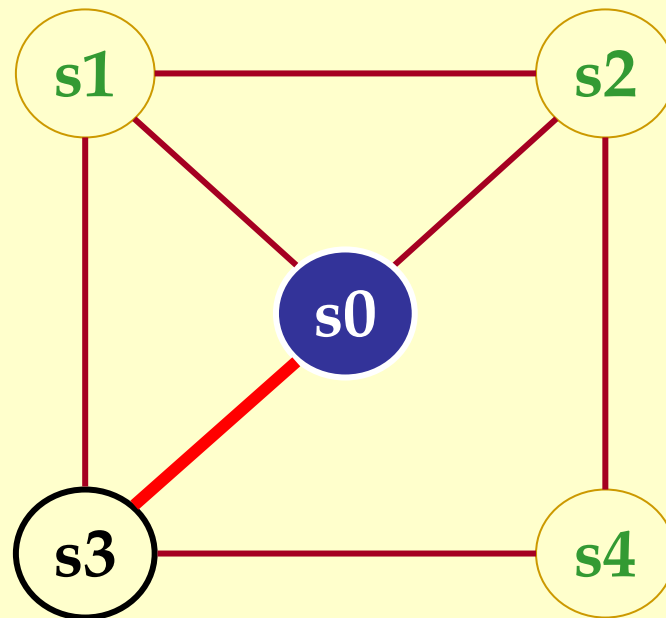
$N = 3$   



s3
s1
s2
s4

Coloring Example

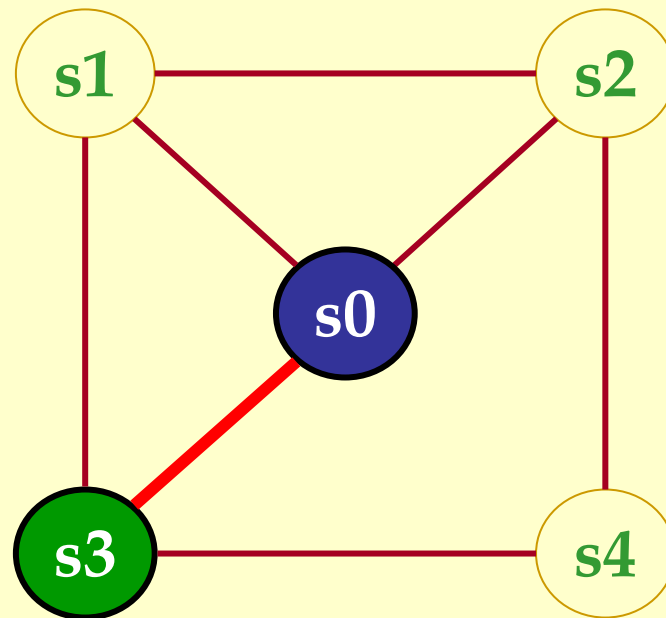
$N = 3$   



s1
s2
s4

Coloring Example

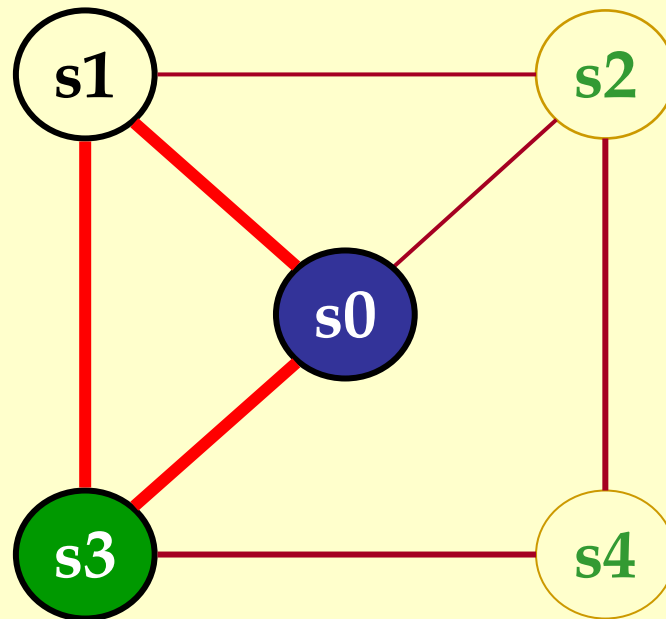
$N = 3$



s1
s2
s4

Coloring Example

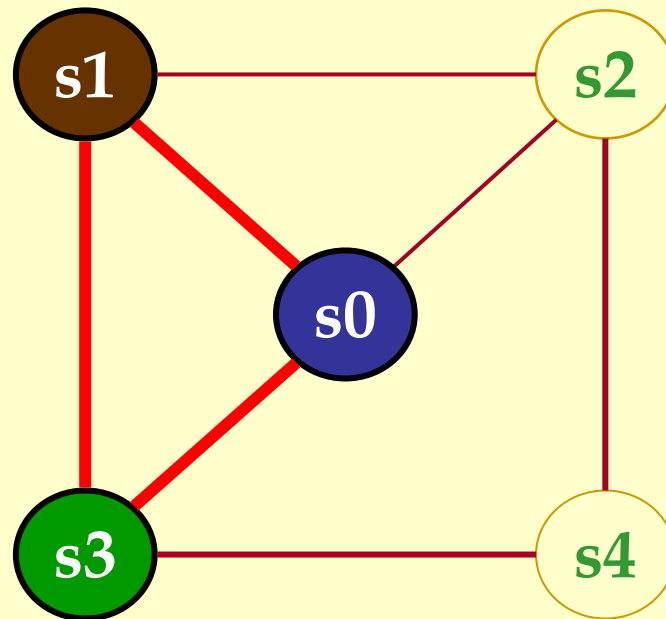
$N = 3$



s2
s4

Coloring Example

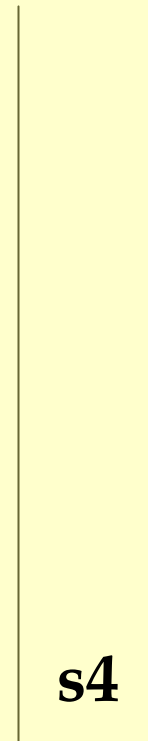
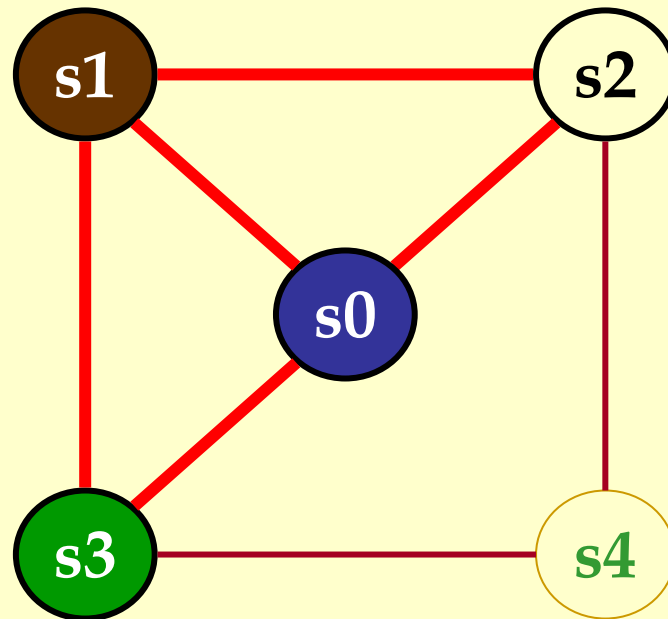
$N = 3$



s2
s4

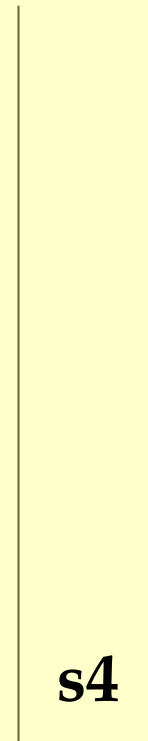
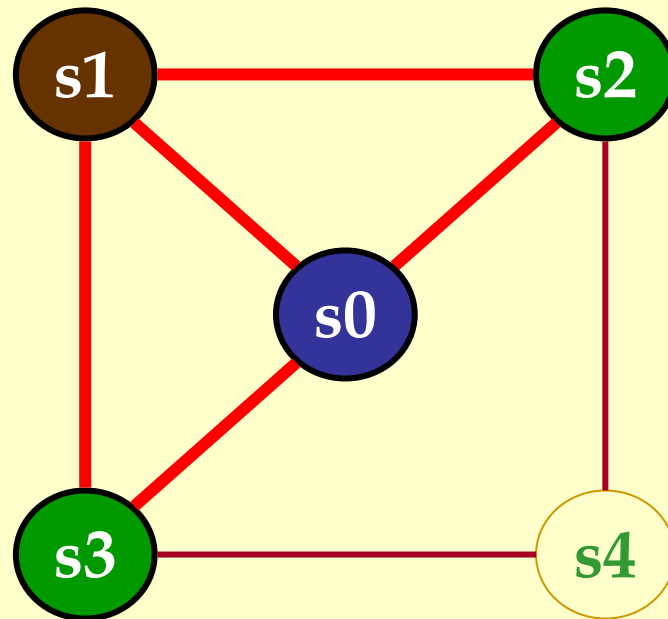
Coloring Example

$N = 3$



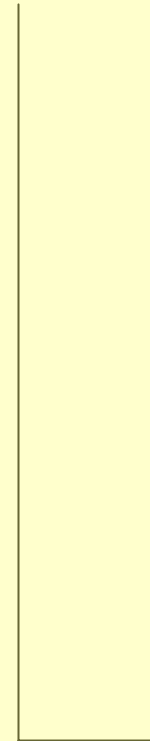
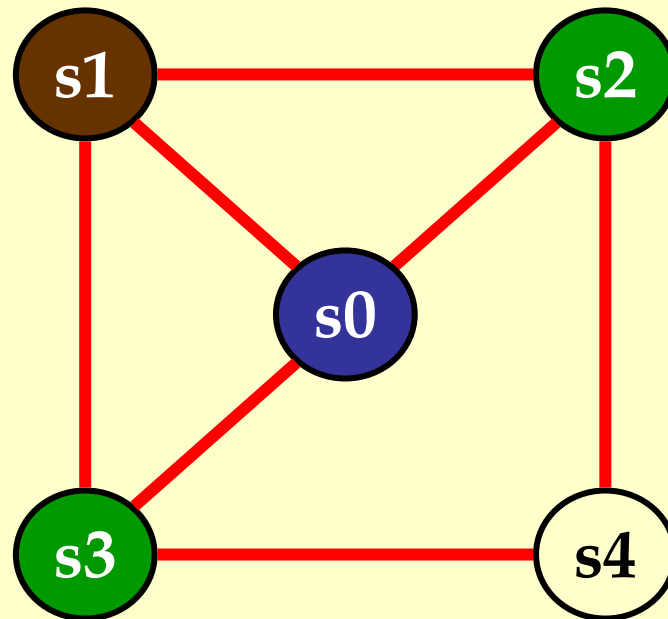
Coloring Example

$N = 3$



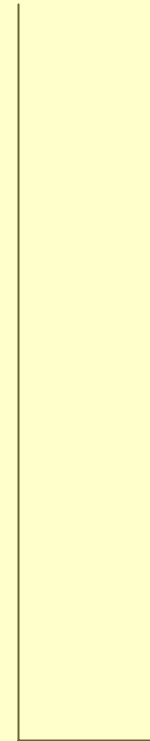
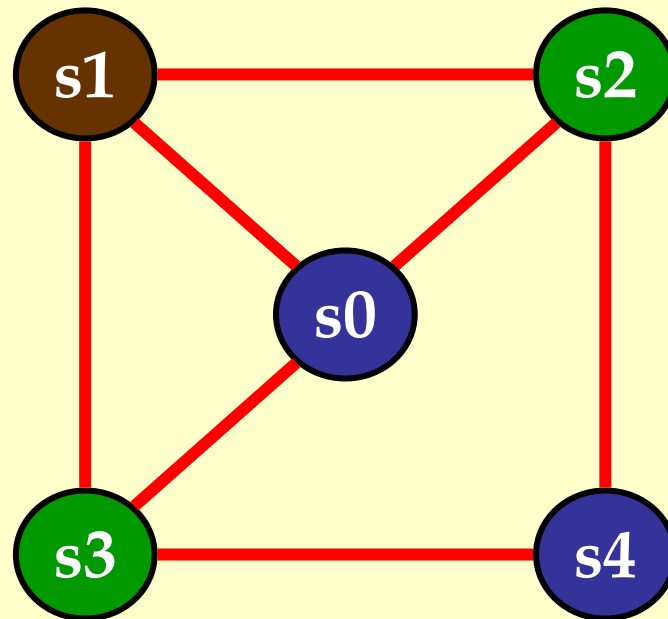
Coloring Example

$N = 3$   



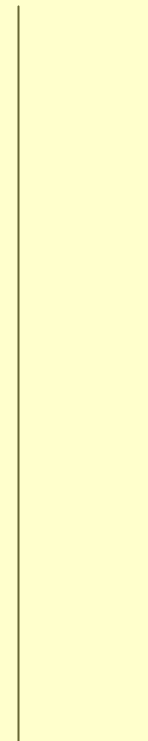
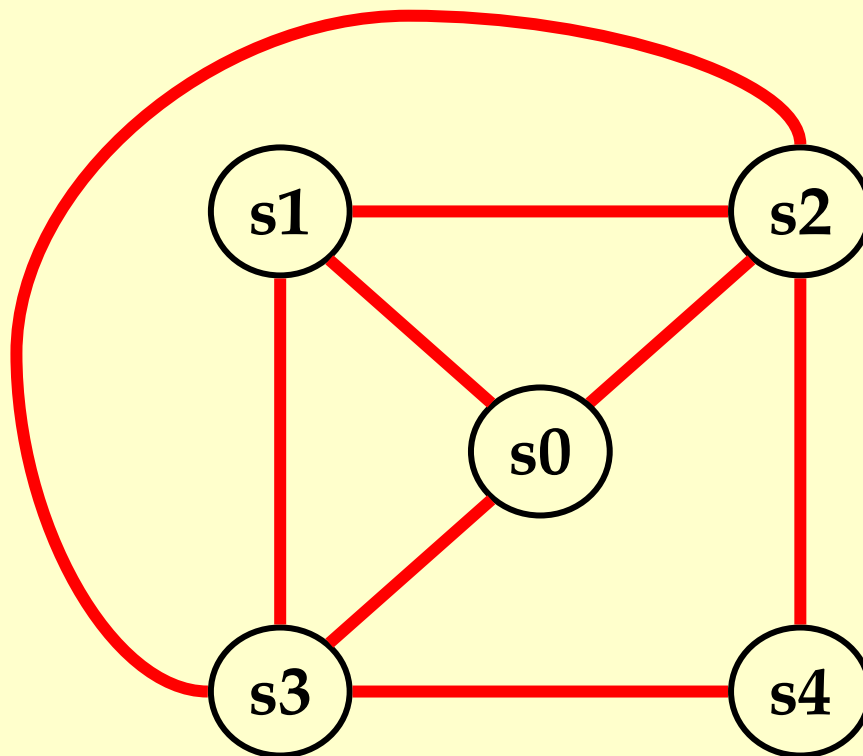
Coloring Example

$N = 3$   



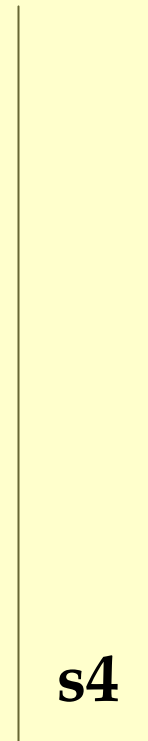
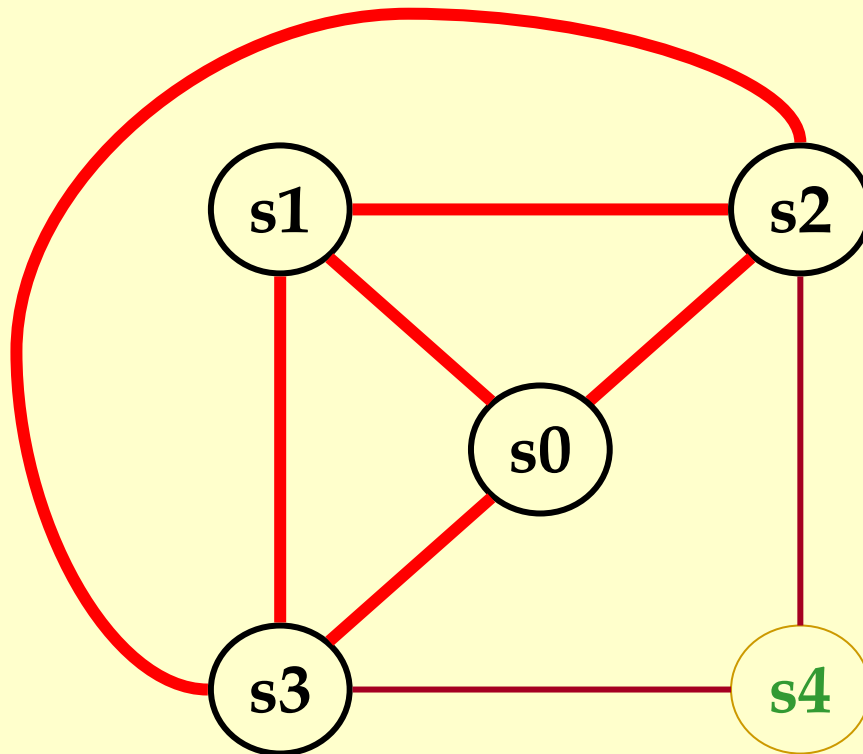
Another Coloring Example

$N = 3$



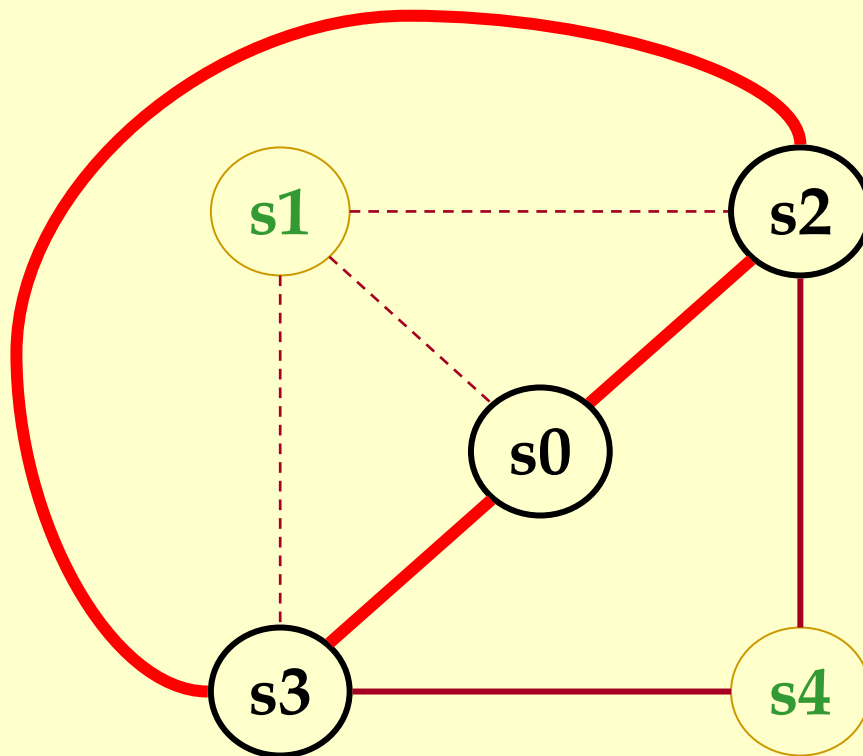
Another Coloring Example

$N = 3$



Another Coloring Example

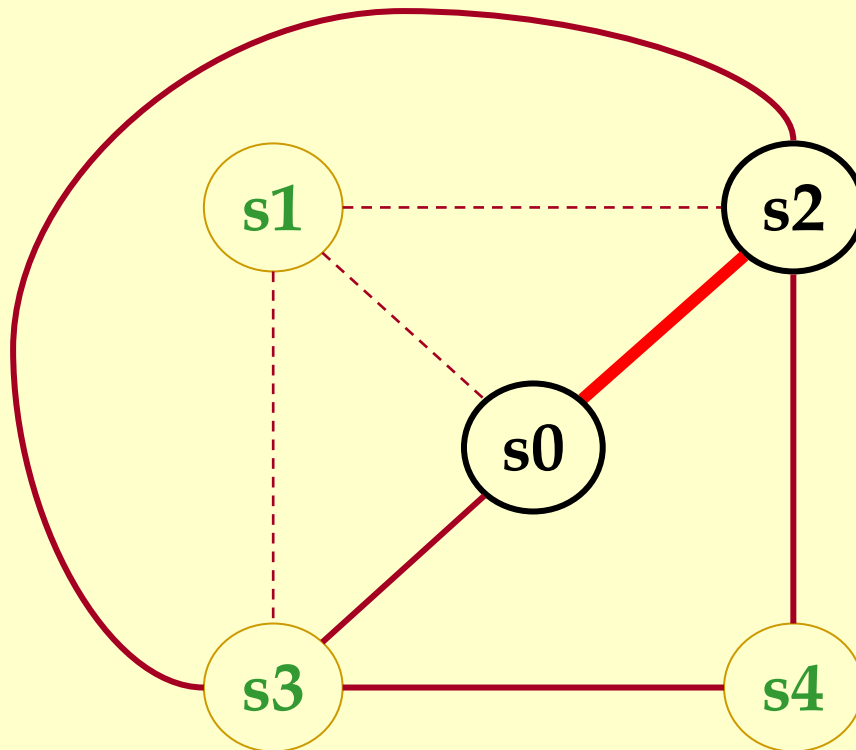
$N = 3$



s1: Possible Spill

Another Coloring Example

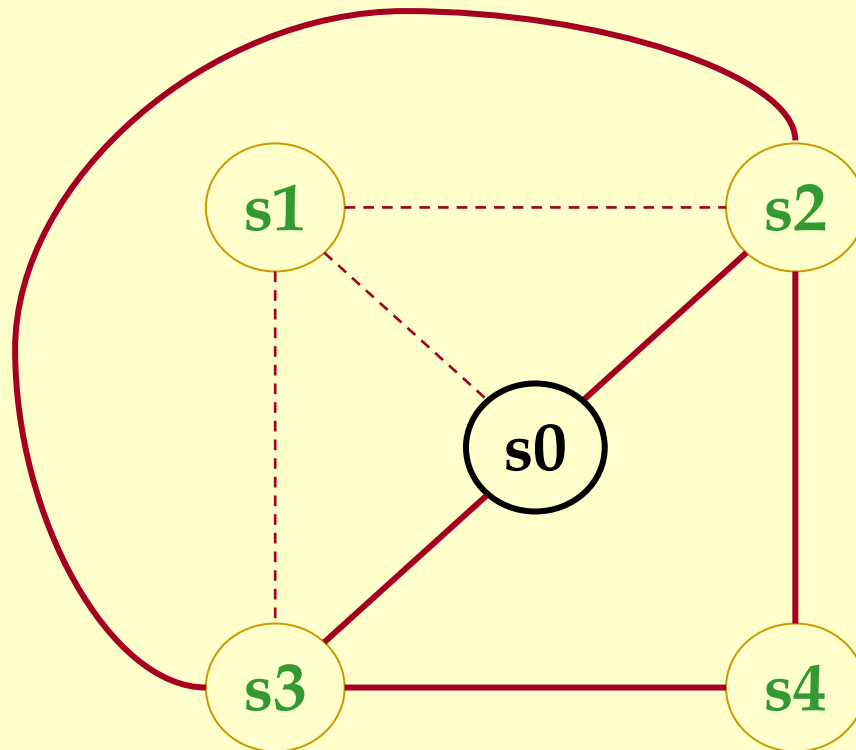
$N = 3$



- s3
- s1**
- s4

Another Coloring Example

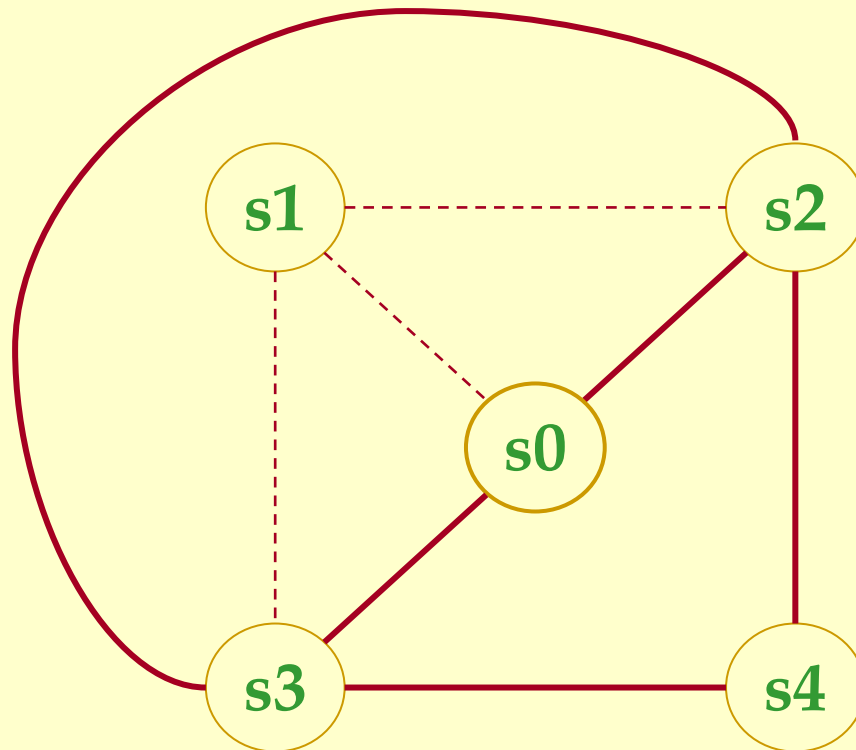
$N = 3$



- s2
- s3
- s1**
- s4

Another Coloring Example

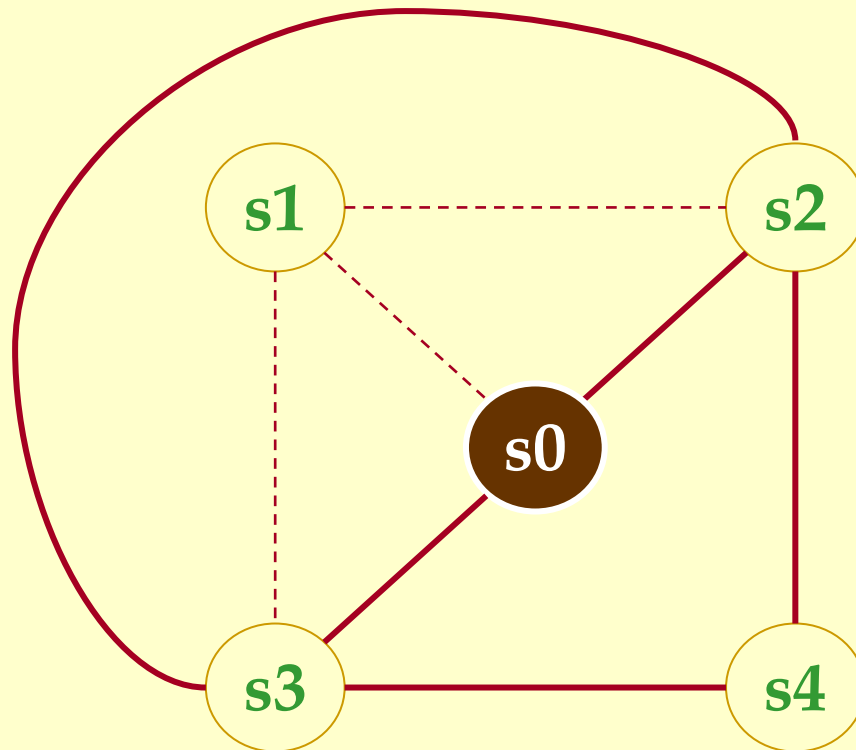
$N = 3$



- s0
- s2
- s3
- s1**
- s4

Another Coloring Example

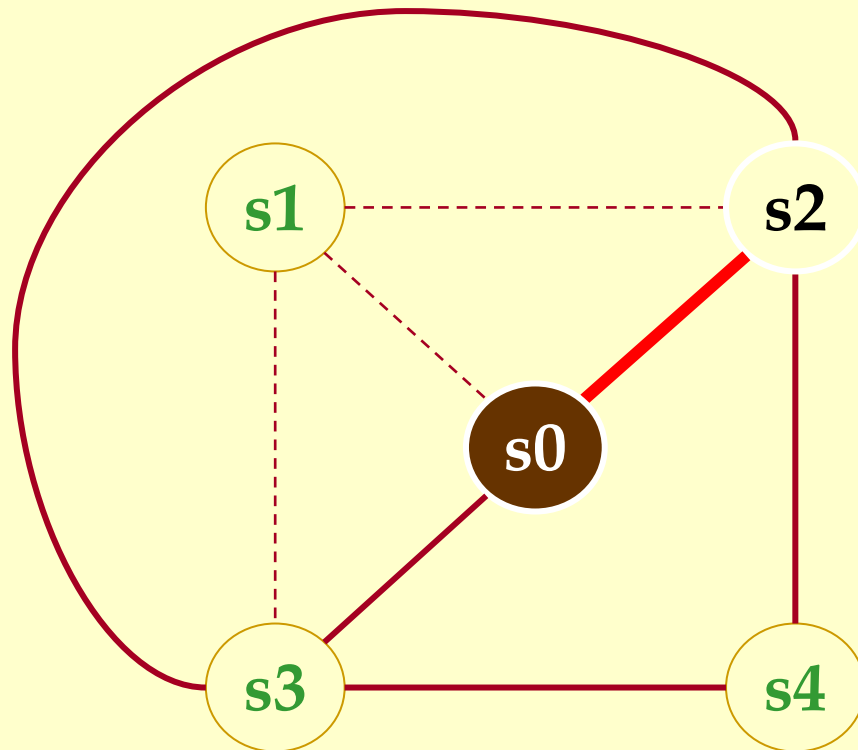
$N = 3$



- s2
- s3
- s1**
- s4

Another Coloring Example

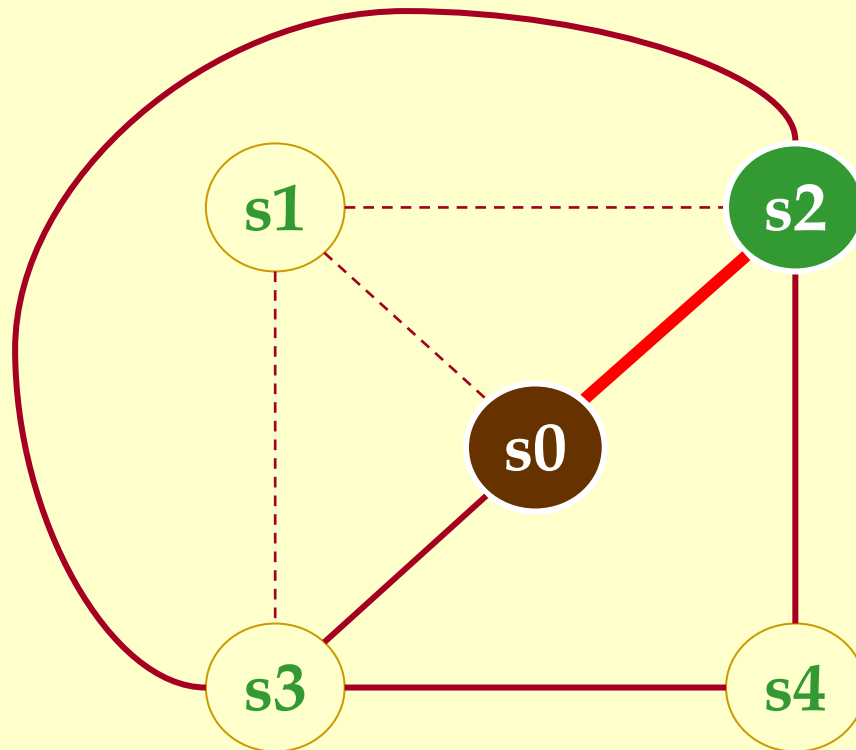
$N = 3$



- s3
- s1**
- s4

Another Coloring Example

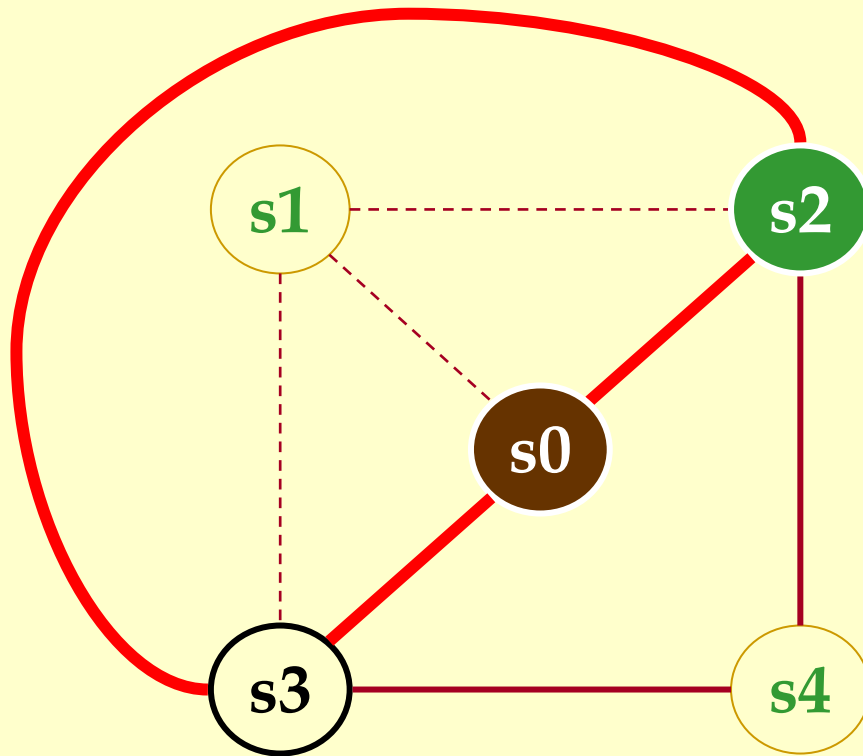
$N = 3$



- s3
- s1**
- s4

Another Coloring Example

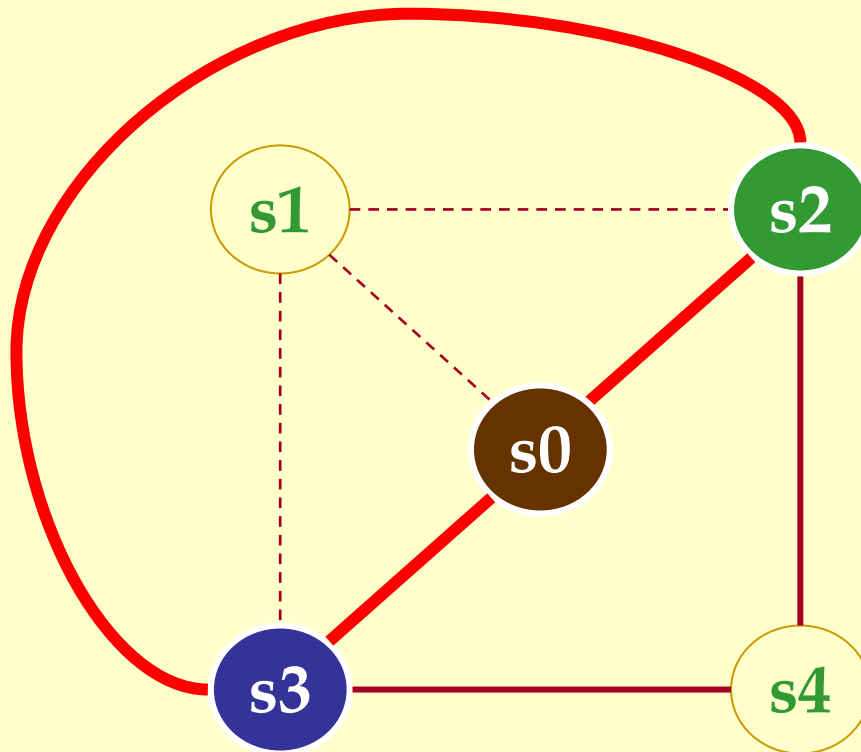
$N = 3$



s1
s4

Another Coloring Example

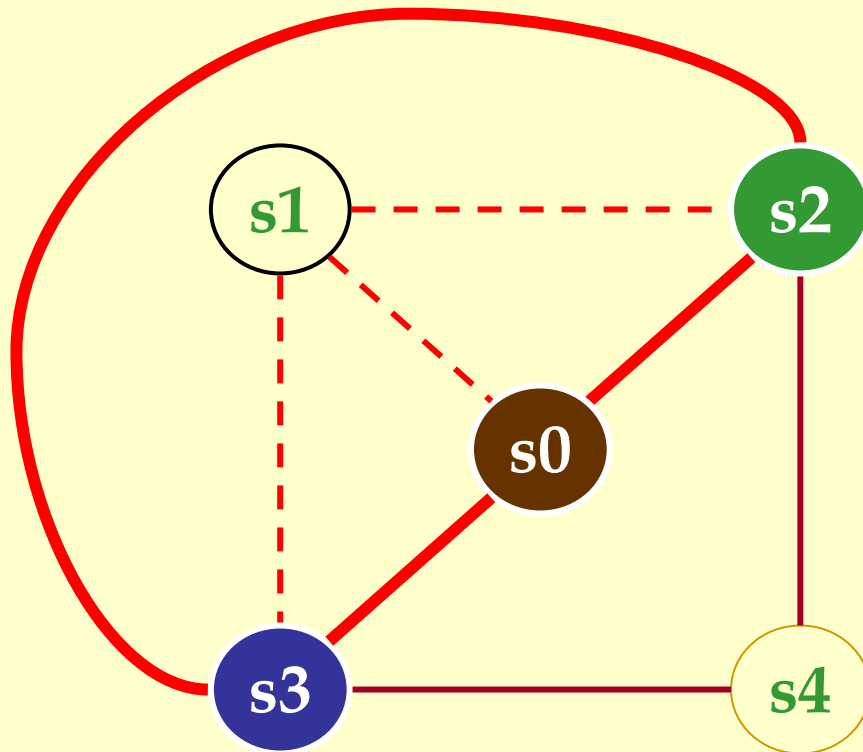
$N = 3$



s1
s4

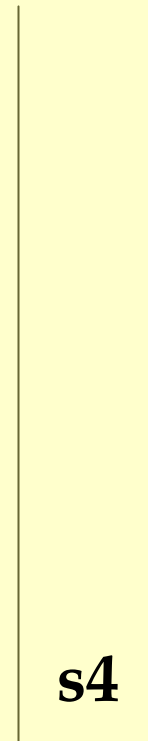
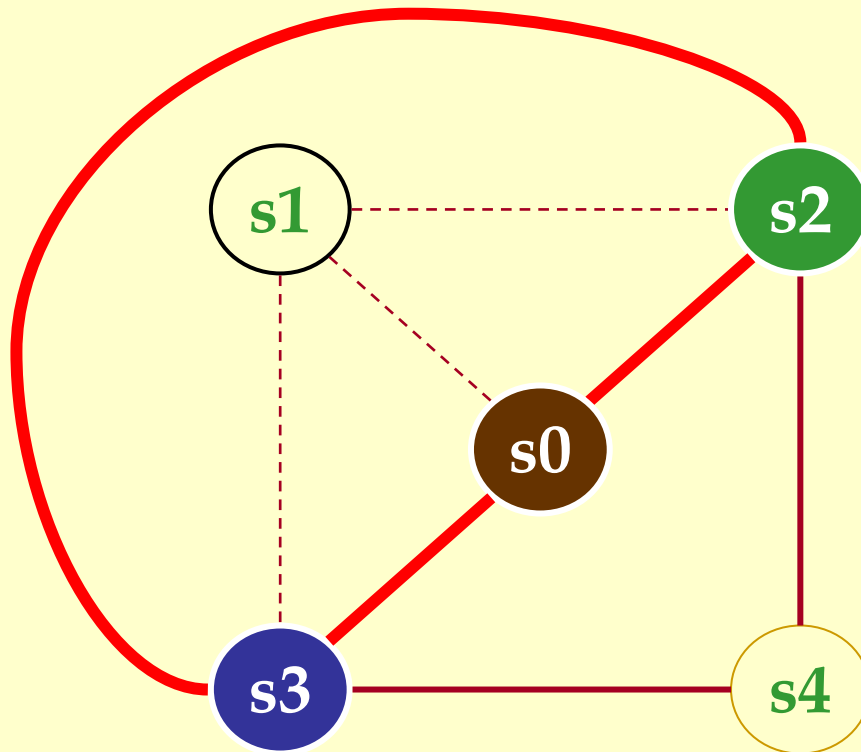
Another Coloring Example

$N = 3$



Another Coloring Example

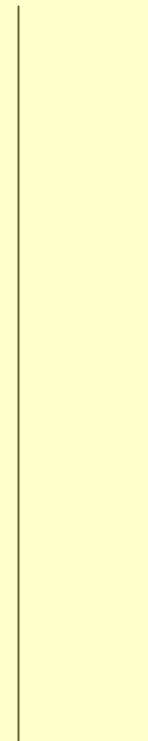
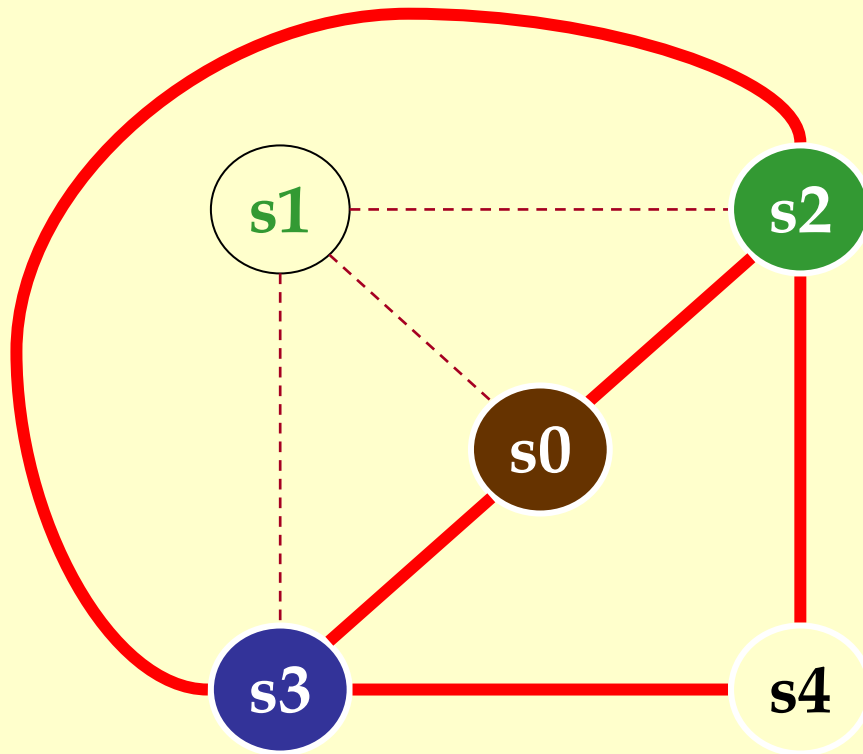
$N = 3$



s1: Actual Spill

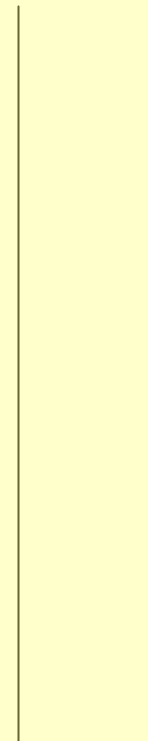
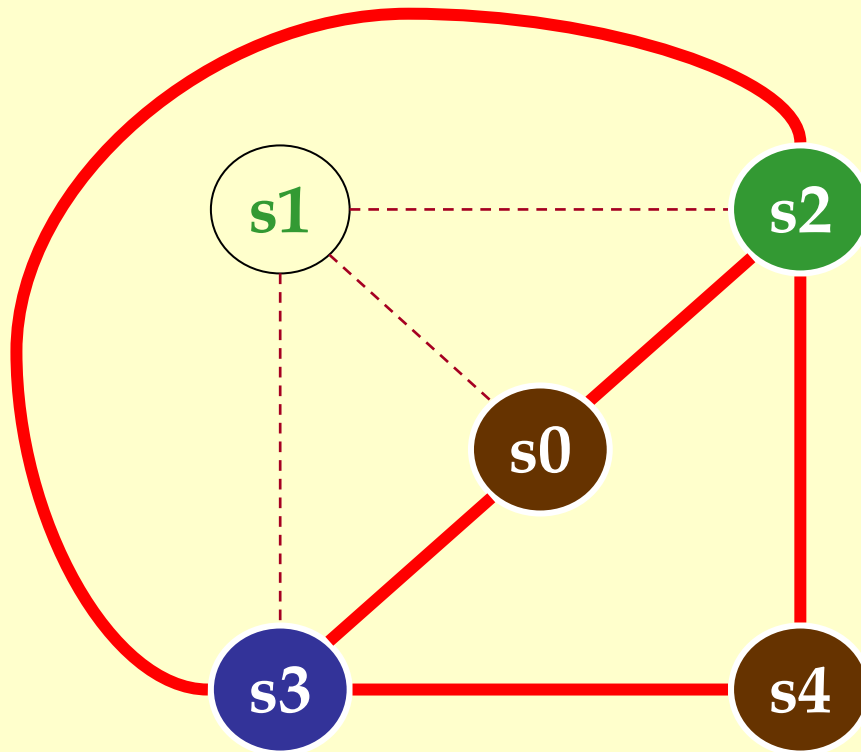
Another Coloring Example

$N = 3$



Another Coloring Example

$N = 3$



When Coloring Heuristics Fail...

Option 1:

- ◆ Pick a web and allocate value in memory.
- ◆ All defs go to memory, all uses come from memory.

Option 2:

- ◆ Split the web into multiple webs.

- ◆ In either case, will retry the coloring.

Which web to spill?

- ◆ One with interference degree $\geq N$.
- ◆ One with minimal **spill cost** (cost of placing value in memory rather than in register).
- ◆ What is spill cost?
 - ◆ Cost of extra load and store instructions.

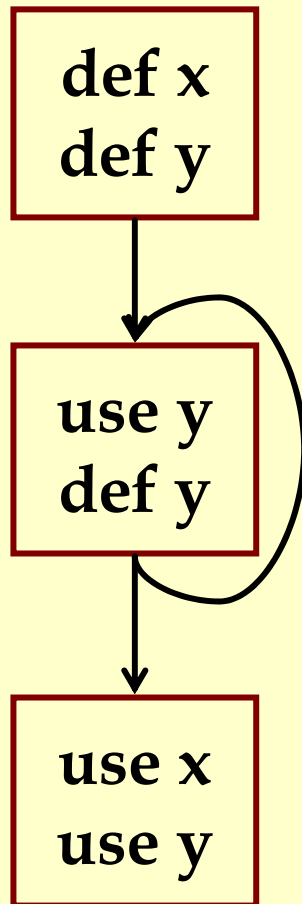
Ideal and Useful Spill Costs

- ◆ Ideal spill cost - dynamic cost of extra load and store instructions. Can't expect to compute this.
 - ◆ Don't know which way branches resolve.
 - ◆ Don't know how many times loops execute.
 - ◆ Actual cost may be different for different executions.
- ◆ Solution: Use a static approximation.
 - ◆ profiling can give instruction execution frequencies.
 - ◆ or use heuristics based on structure of control flow graph.

One Way to Compute Spill Cost

- ◆ Goal: give priority to values used in loops.
- ◆ So assume loops execute 10 (or 8) times.
- ◆ Spill cost =
 - ◆ sum over all def sites of cost of a store instruction times 8 to the loop nesting depth power, plus
 - ◆ sum over all use sites of cost of a load instruction times 8 to the loop nesting depth power.
- ◆ Choose the web with the lowest spill cost.

Spill Cost Example



Spill Cost For x
storeCost+loadCost

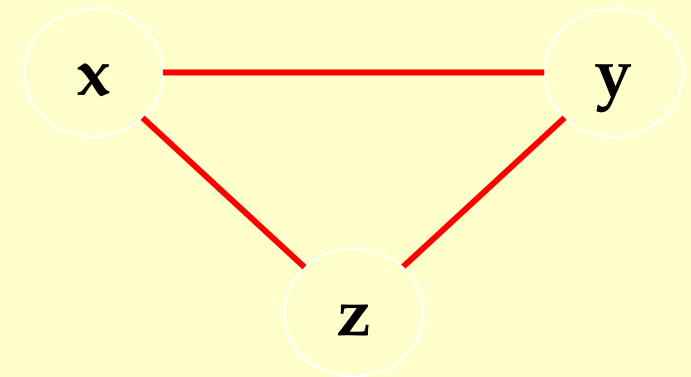
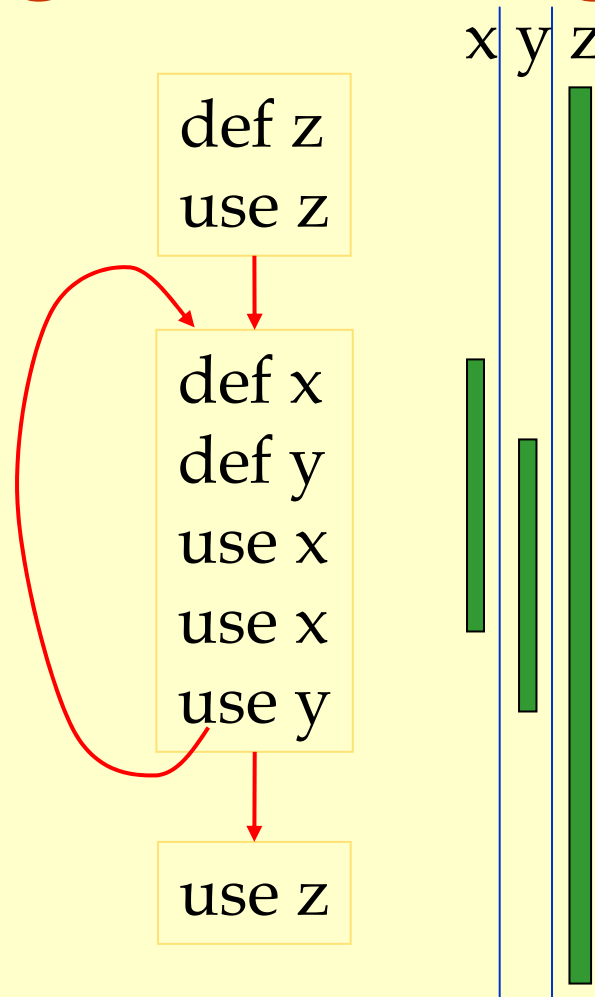
Spill Cost For y
9*storeCost+9*loadCost

**With 1 Register, Which
Variable Gets Spilled?**

Splitting Rather Than Spilling

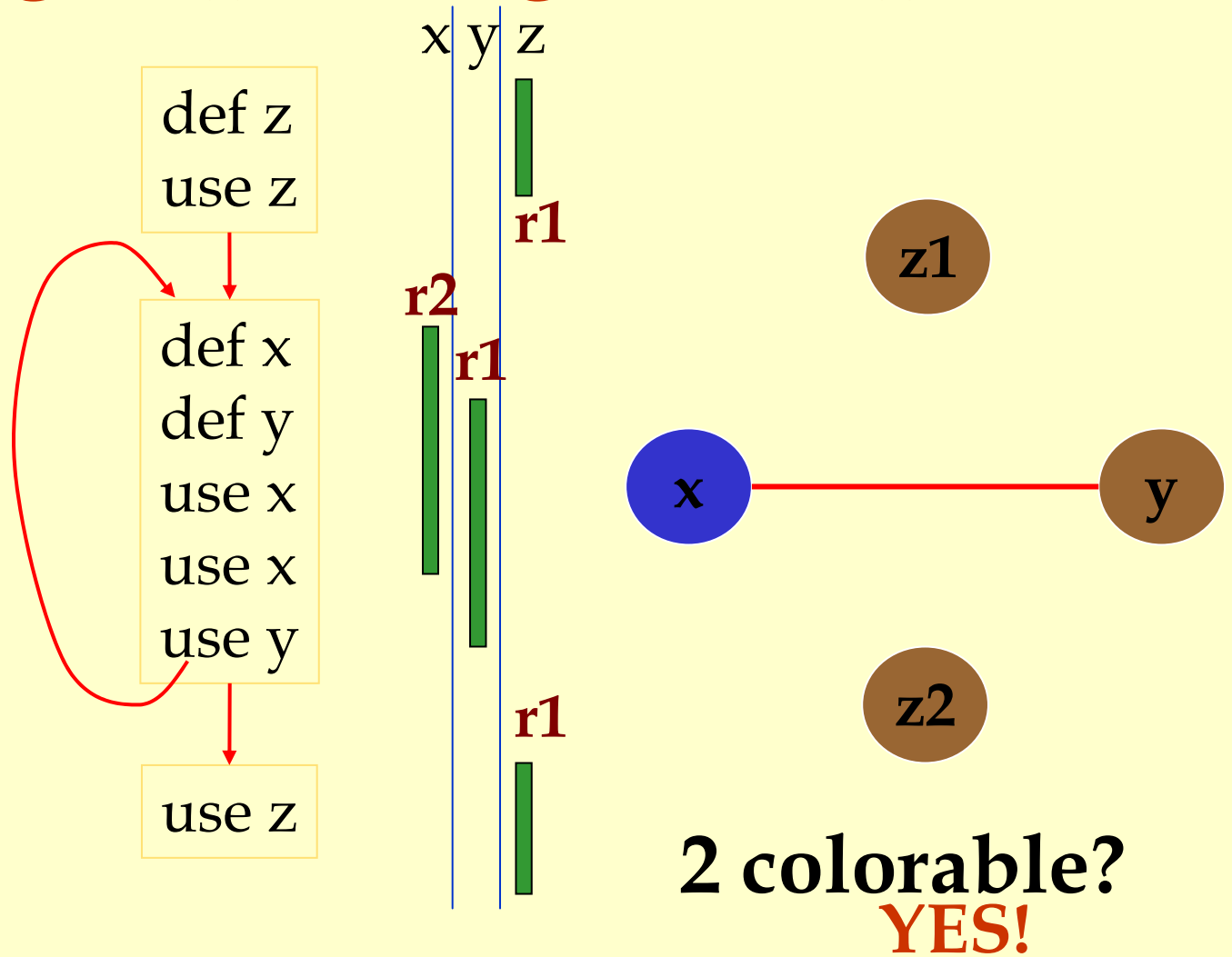
- ◆ Split the web:
 - ◆ Split a web into multiple webs so that there will be less interference in the interference graph making it N -colorable.
 - ◆ Spill the value to memory and load it back at the points where the web is split.

Live-Range Splitting Example

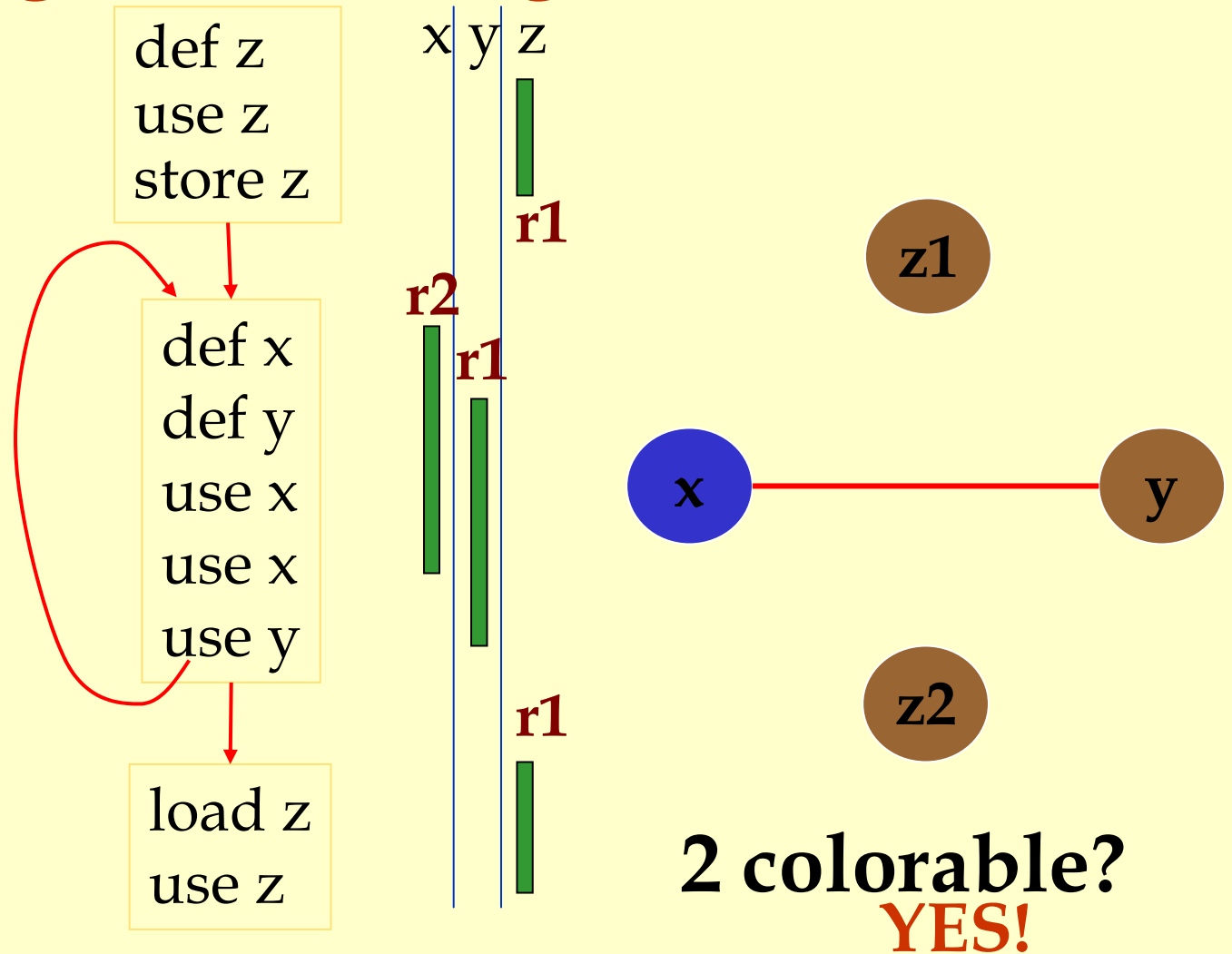


2 colorable?
NO!

Live-Range Splitting Example



Live-Range Splitting Example



Live-Range Splitting Heuristic

- ◆ Identify a program point where the graph is not N -colorable (point where # of webs $> N$).
 - ◆ Pick a web that is not used for the largest enclosing block around that point of the program.
 - ◆ Split that web at the corresponding edge.
 - ◆ Redo the interference graph.
 - ◆ Try to re-color the graph.

Cost and Benefit of Splitting

- ◆ Cost of splitting a node:
 - ◆ Proportional to number of times split edge has to be crossed dynamically.
 - ◆ Estimate by its loop nesting.
- ◆ Benefit:
 - ◆ Increase colorability of the nodes the split web interferes with.
 - ◆ Can be approximate by its degree in the interference graph.
- ◆ Greedy heuristic:
 - ◆ Pick the live-range with the highest benefit-to-cost ration to spill.

Further Optimizations

- ◆ Register coalescing.
- ◆ Register targeting (pre-coloring).
- ◆ Pre-splitting of webs.
- ◆ Interprocedural register allocation.

Register Coalescing

- ◆ Find register copy instructions $s_j = s_i$.
- ◆ If s_j and s_i do not interfere, combine their webs.
- ◆ Pros:
 - ◆ Similar to copy propagation.
 - ◆ Reduce the number of instructions.
- ◆ Cons:
 - ◆ May increase the degree of the combined node.
 - ◆ A colorable graph may become non-colorable.

Register Targeting (pre-coloring)

- ◆ Some variables need to be in special registers at a given time:
 - ◆ First n arguments to a function.
 - ◆ The return value.
- ◆ Pre-color those nodes and bind them to the right register.
- ◆ Will eliminate unnecessary copy instructions.

Pre-splitting of the webs

- ◆ Some live ranges have very large “dead” regions.
 - ◆ Large region where the variable is unused.
- ◆ Break-up the live ranges:
 - ◆ Need to pay a small cost in spilling.
 - ◆ But the graph will be very easy to color.
- ◆ Can find strategic locations to break-up:
 - ◆ At a call site (need to spill anyway).
 - ◆ Around a large loop nest (reserve registers for values used in the loop).

Interprocedural Register Allocation

- ◆ Saving registers across procedure boundaries is expensive.
 - ◆ especially for programs with many small functions.
- ◆ Calling convention is too general and inefficient.
- ◆ Customize calling convention per function by doing interprocedural register allocation.

Summary

- ◆ The goal of register allocation is to speed up the program by keeping values in registers.
- ◆ Usually gives a big impact on performance.
- ◆ The most commonly used method is some form of heuristic graph coloring.
- ◆ There exists many other methods.

Instruction Scheduling

This lecture is primarily based on Konstantinos Sagonas set of slides
(Advanced Compiler Techniques, (2AD518)
at Uppsala University, January-February 2004).
Used with kind permission.

Outline

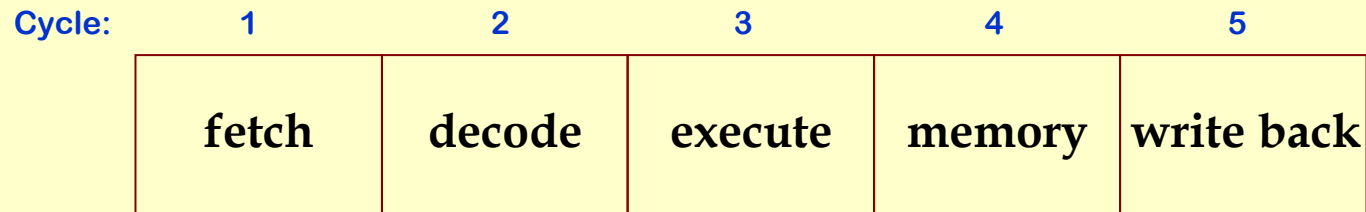
- ◆ Modern architectures.
- ◆ Delay slots.
- ◆ Introduction to instruction scheduling.
- ◆ List scheduling.
- ◆ Resource constraints.
- ◆ Interaction with register allocation.
- ◆ Scheduling across basic blocks.
- ◆ Trace scheduling.
- ◆ Scheduling for loops.
- ◆ Loop unrolling.
- ◆ Software pipelining.

Simple Machine Model

- ◆ Instructions are executed in sequence.
 - ◆ Fetch, decode, execute, store results.
 - ◆ One instruction at a time.
- ◆ For branch instructions, start fetching from a different location if needed.
 - ◆ Check branch condition.
 - ◆ Next instruction may come from a new location given by the branch instruction.

Simple Execution Model

5 Stage pipe-line:



Fetch: get the next instruction.

Decode: figure out what that instruction is.

Execute: perform ALU operation.

address calculation in a memory op

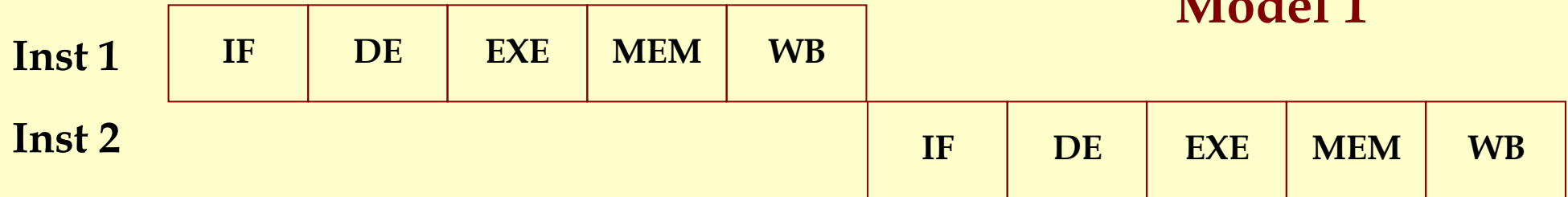
Memory: do the memory access in a mem. op.

Write Back: write the results back.

Execution Models

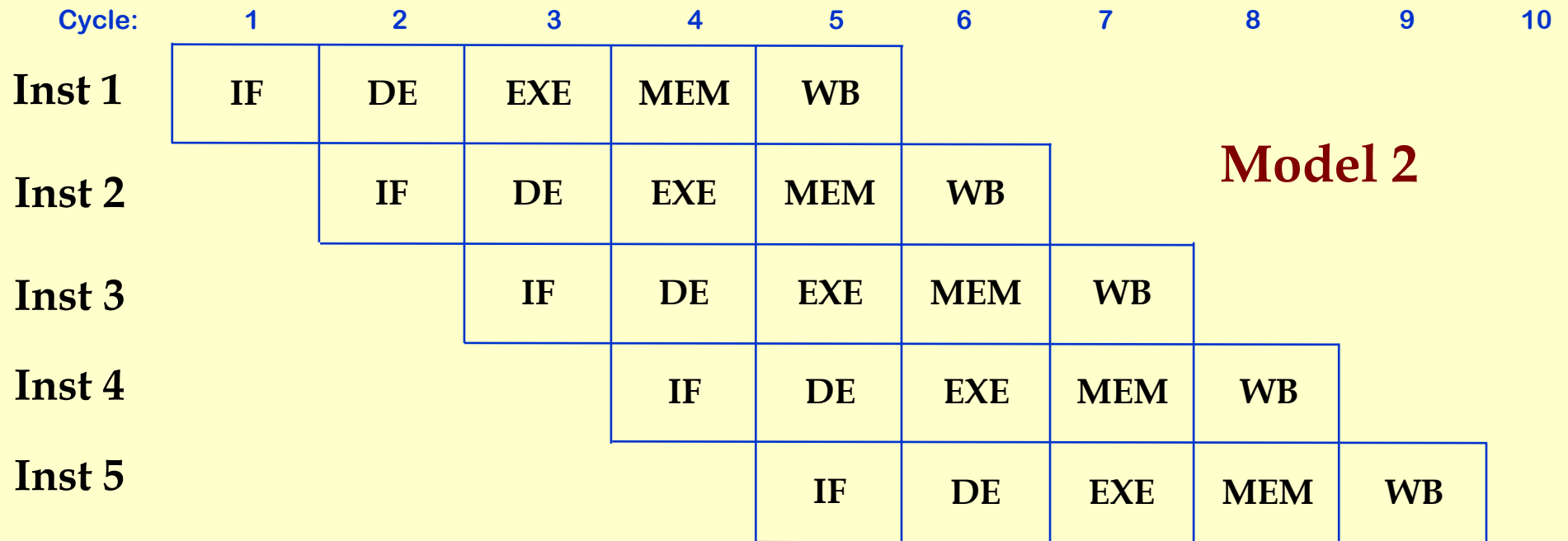
time (cycles) →

Model 1



One instruction finish every 5 cycles.

Model 2

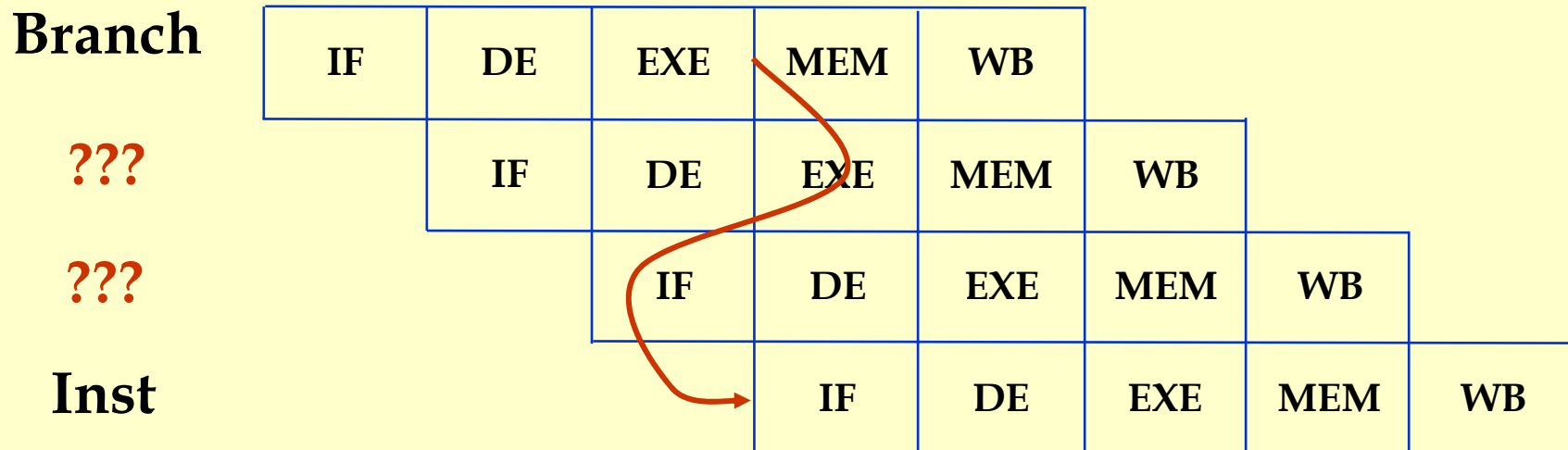


One instruction finish every cycle.

Handling Branch Instructions

Problem: We do not know the location of the next instruction until later.

- ◆ after DE in jump instructions
- ◆ after EXE in conditional branch instructions

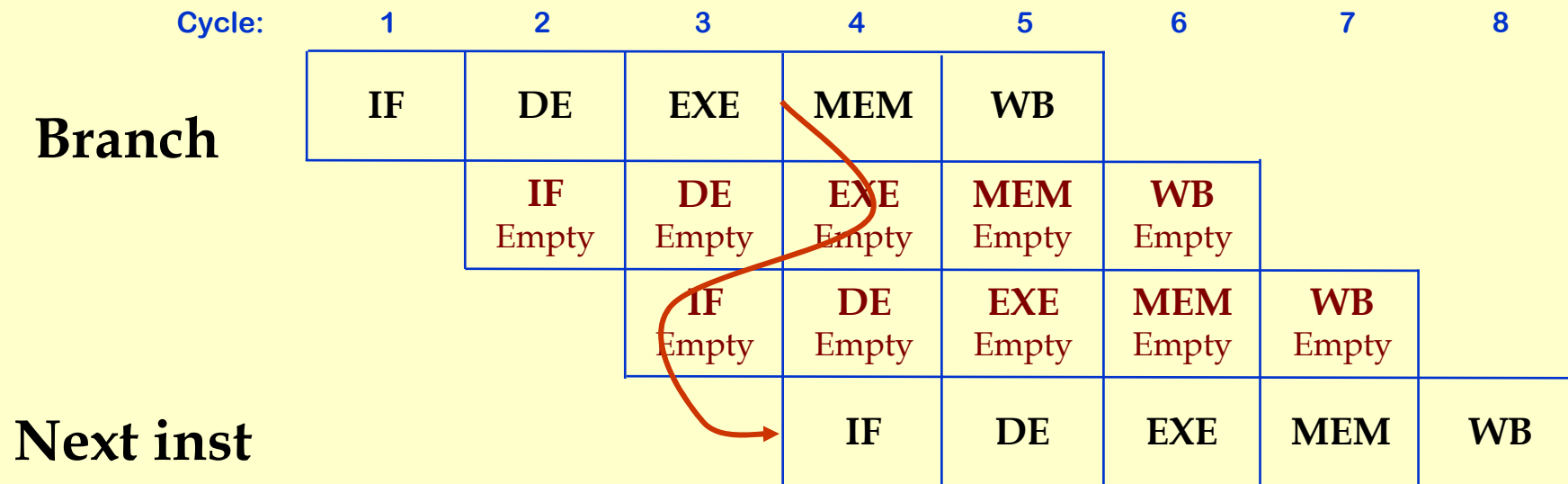


What to do with the middle 2 instructions?

Handling Branch Instructions

What to do with the middle 2 instructions?

1. Stall the pipeline in case of a branch until we know the address of the next instruction:
 - ◆ wasted cycles

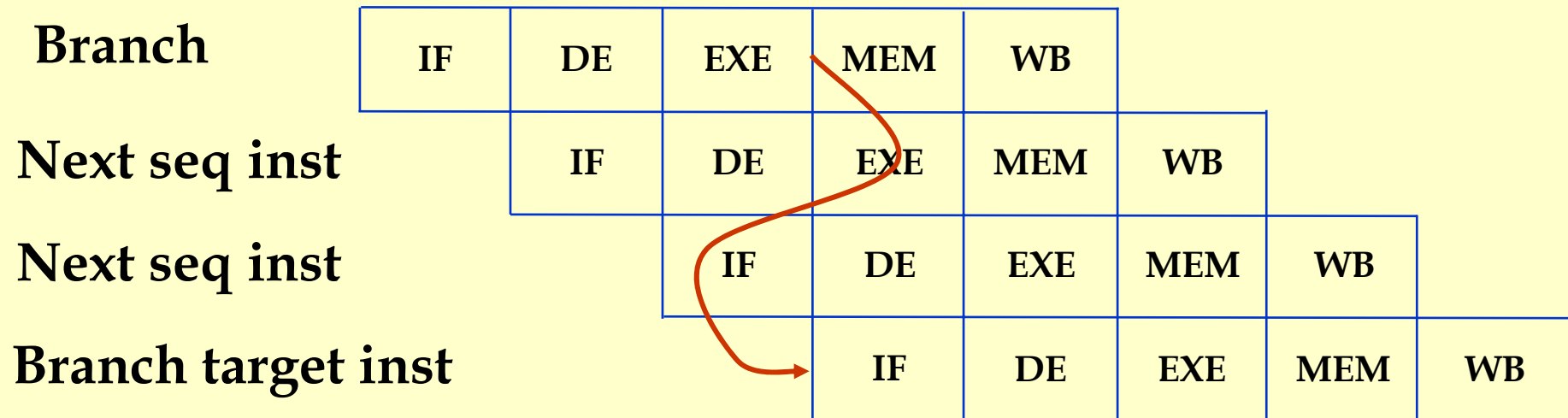


Handling Branch Instructions

What to do with the middle 2 instructions?

2. Delay the action of the branch

- ◆ Make branch affect only after two instructions
- ◆ Following two instructions after the branch get executed regardless of the branch

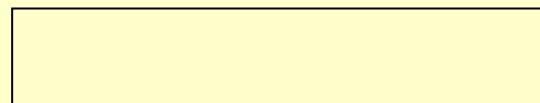


Branch Delay Slot(s)

MIPS has a branch delay slot

- ◆ The instruction after a conditional branch gets executed even if the code branches to target
- ◆ Fetching from the branch target takes place only after that

```
ble r3, foo
```



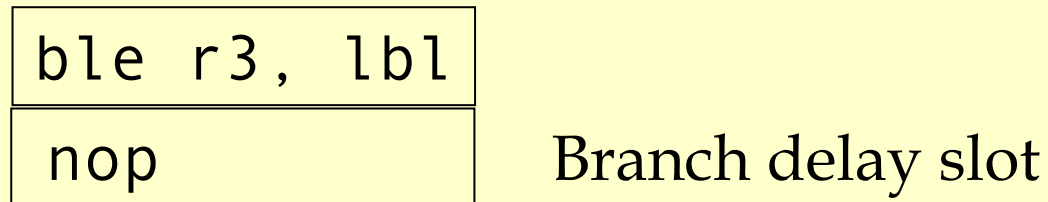
Branch delay slot

What instruction to put in the branch delay slot?

Filling the Branch Delay Slot

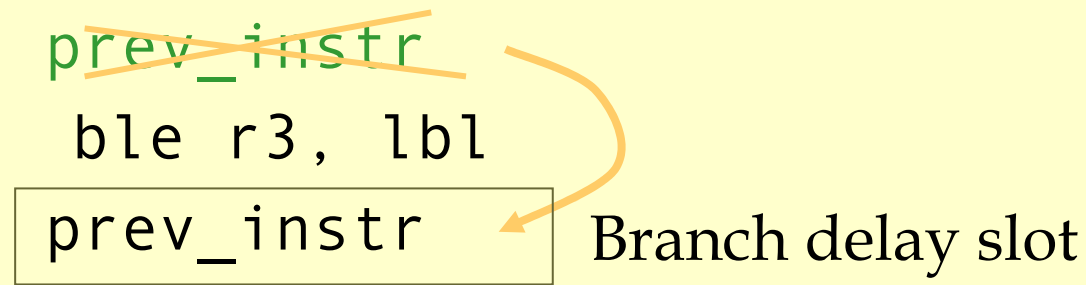
Simple Solution: Put a no-op.

Wasted instruction, just like a stall.



Filling the Branch Delay Slot

Move an instruction from above the branch.



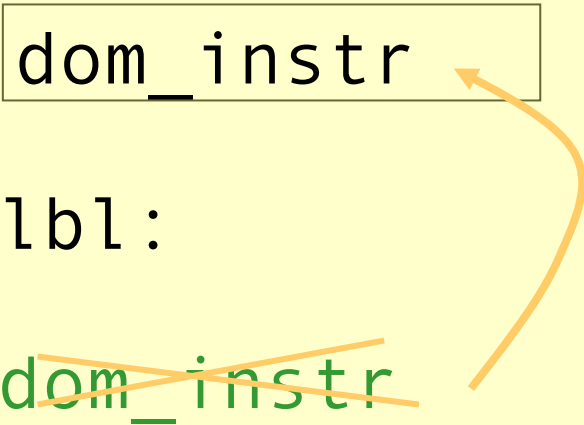
- ◆ Moved instruction executes iff branch executes.
 - ◆ Get the instruction from the same basic block as the branch.
 - ◆ Don't move a branch instruction!
- ◆ Instruction need to be moved over the branch.
 - ◆ Branch does not depend on the result of the inst.

Filling the Branch Delay Slot

Move an instruction dominated by the branch instruction.

```
ble r3, lbl
```

```
dom_instr
```



Branch delay slot

```
lbl:
```

```
dom_instr
```

Filling the Branch Delay Slot

Move an instruction from the branch target.

- ◆ Instruction dominated by target.
- ◆ No other ways to reach target (if so, take care of them).
- ◆ If conditional branch, instruction should not have a lasting effect if the branch is not taken.

```
ble r3, lbl
```

```
instr
```



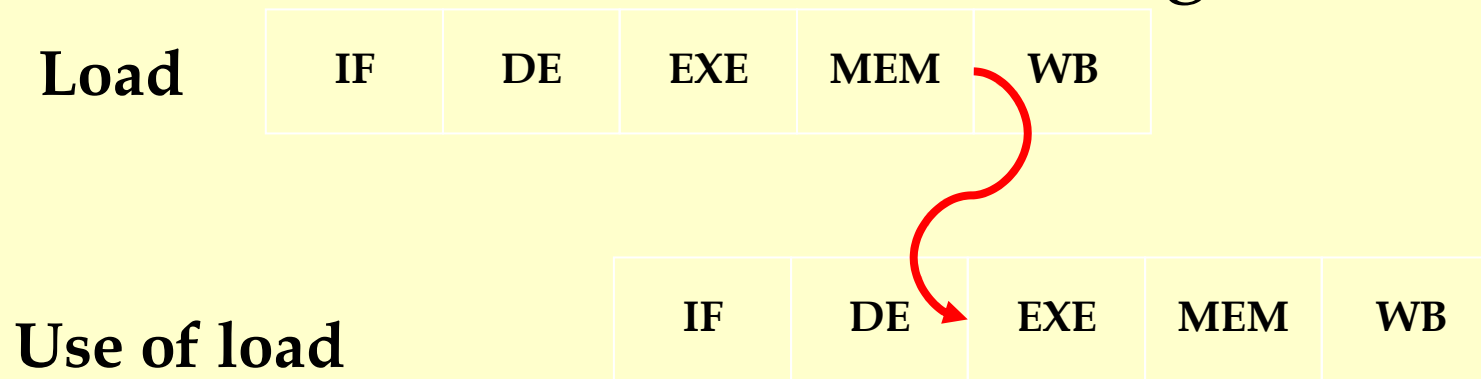
Branch delay slot

```
lbl:
```

```
instr
```

Load Delay Slots

Problem: Results of the loads are not available until end of MEM stage



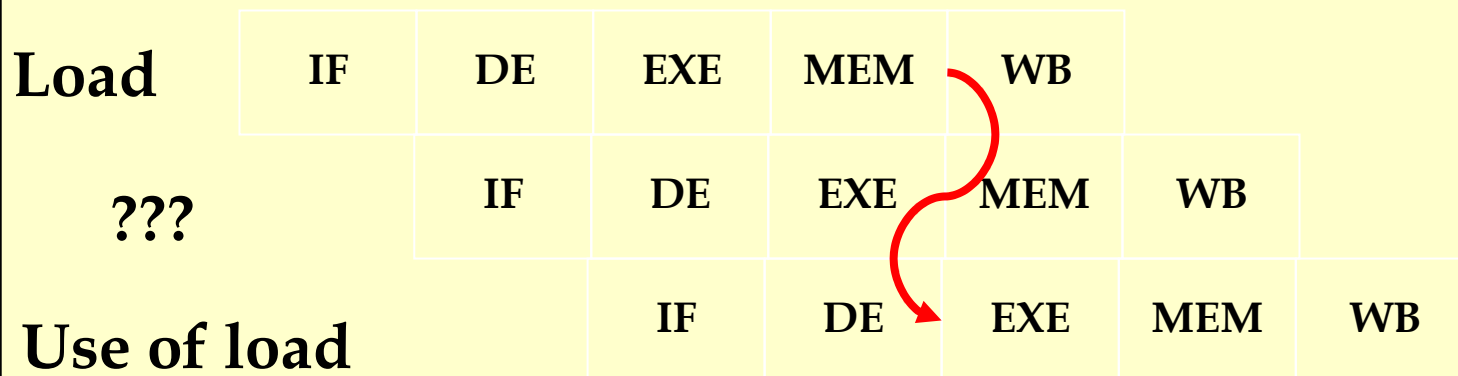
If the value of the load is used...what to do??

Load Delay Slots

If the value of the load is used...what to do?

Always stall one cycle.

- ◆ Stall one cycle if next instruction uses the value.
 - ◆ Need hardware to do this.
- ◆ Have a delay slot for load.
 - ◆ The new value is only available after two instructions.
 - ◆ If next inst. uses the register, it will get the old value.



Example

`r2 = *(r1 + 4)`

`r3 = *(r1 + 8)`

`r4 = r2 + r3`

`r5 = r2 - 1`

`goto L1`

Example

r2 = *(r1 + 4)

r3 = *(r1 + 8)

noop

r4 = r2 + r3

r5 = r2 - 1

goto L1

noop

Assume 1 cycle delay on branches
and 1 cycle latency for loads

Example

r2 = *(r1 + 4)

r3 = *(r1 + 8)

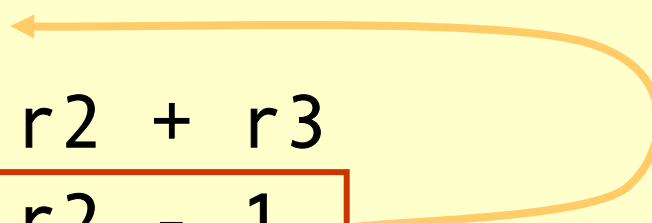
noop

r4 = r2 + r3

r5 = r2 - 1

goto L1

noop



Example

`r2 = *(r1 + 4)`

`r3 = *(r1 + 8)`

`r5 = r2 - 1`

`r4 = r2 + r3`

`goto L1`

`noop`



Example

`r2 = *(r1 + 4)`

`r3 = *(r1 + 8)`

`r5 = r2 - 1`

`goto L1`

`r4 = r2 + r3`

Example

```
r2 = *(r1 + 4)
```

```
r3 = *(r1 + 8)
```

```
r5 = r2 - 1
```

```
goto L1
```

```
r4 = r2 + r3
```

Final code after delay slot filling

From a Simple Machine Model to a Real Machine Model

- ◆ Many pipeline stages.
 - ◆ MIPS R4000 has 8 stages.
- ◆ Different instructions take different amount of time to execute.
 - ◆ mult 10 cycles
 - ◆ div 69 cycles
 - ◆ ddiv 133 cycles
- ◆ Hardware to stall the pipeline if an instruction uses a result that is not ready.

Real Machine Model cont.

- ◆ Most modern processors have multiple execution units (**superscalar**).
 - ◆ If the instruction sequence is correct, multiple operations will take place in the same cycles.
 - ◆ Even more important to have the right instruction sequence.

Instruction Scheduling

Goal: Reorder instructions so that pipeline stalls are minimized.

Constraints on Instruction Scheduling:

- ◆ Data dependencies.
- ◆ Control dependencies .
- ◆ Resource constraints.

Data Dependencies

- ◆ If two instructions access the same variable, they can be dependent.
- ◆ Kinds of dependencies:
 - ◆ True: **write** → **read**. (Read After Write, RAW)
 - ◆ Anti: **read** → **write**. (Write After Read, WAR)
 - ◆ Anti (Output): **write** → **write**. (Write After Write, WAW)
- ◆ What to do if two instructions are dependent?
 - ◆ The order of execution cannot be reversed.
 - ◆ Reduce the possibilities for scheduling.

Computing Data Dependencies

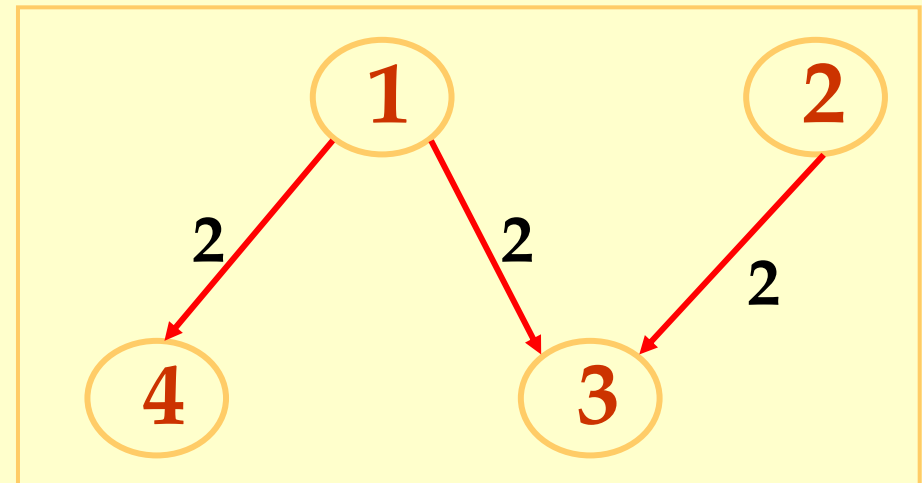
- ◆ For basic blocks, compute dependencies by walking through the instructions.
- ◆ Identifying register dependencies is simple.
 - ◆ is it the same register?
- ◆ For memory accesses.
 - ◆ simple: $\text{base} + \text{offset1} \neq \text{base} + \text{offset2}$
 - ◆ data dependence analysis: $a[2i] \neq a[2i+1]$
 - ◆ interprocedural analysis: $\text{global} \neq \text{parameter}$
 - ◆ pointer alias analysis: $p1 \neq p$

Representing Dependencies

- ◆ Using a dependence DAG, one per basic block.
- ◆ Nodes are instructions, edges represent dependencies.

```

1: r2 = *(r1 + 4)
2: r3 = *(r1 + 8)
3: r4 = r2 + r3
4: r5 = r2 - 1
  
```



Edge is labeled with latency:

$v(i \rightarrow j)$ = delay required between initiation times of i and j minus the execution time required by i .

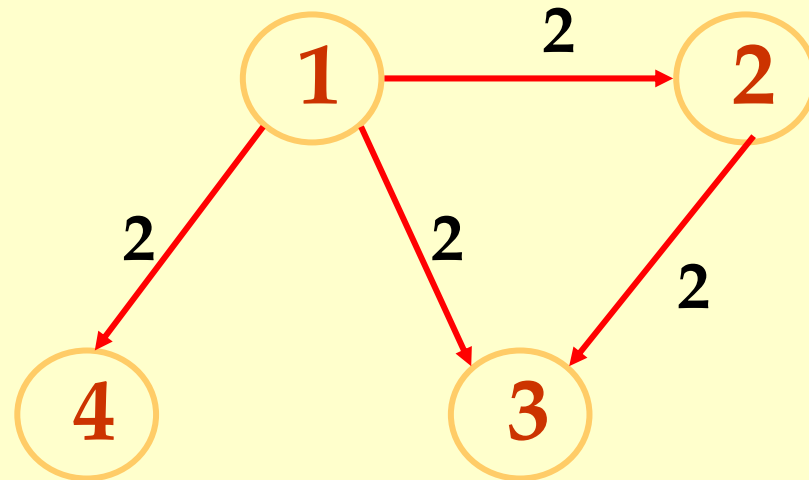
Example

$$1: r2 = *(r1 + 4)$$

$$2: r3 = *(r2 + 4)$$

$$3: r4 = r2 + r3$$

$$4: r5 = r2 - 1$$



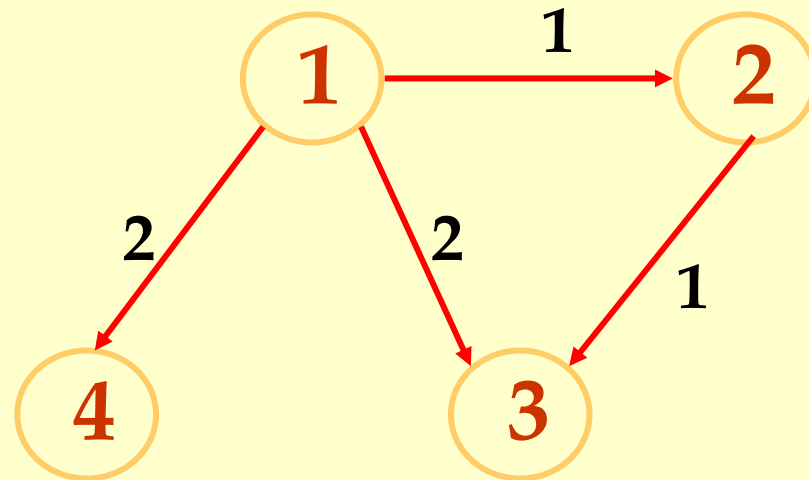
Another Example

1: $r2 = *(r1 + 4)$

2: $*(r1 + 4) = r3$

3: $r3 = r2 + r3$

4: $r5 = r2 - 1$



Control Dependencies and Resource Constraints

- ◆ For now, let's only worry about basic blocks.
- ◆ For now, let's look at simple pipelines.

Example

		Results available in
1:	LA r1, array	1 cycle
2:	LD r2, 4(r1)	1 cycle
3:	AND r3, r3, 0x00FF	1 cycle
4:	MULC r6, r6, 100	3 cycles
5:	ST r7, 4(r6)	
6:	DIVC r5, r5, 100	4 cycles
7:	ADD r4, r2, r5	1 cycle
8:	MUL r5, r2, r4	3 cycles
9:	ST r4, 0(r1)	

14 cycles!

1	2	3	4	st	st	5	6	st	st	st	7	8	9
---	---	---	---	----	----	---	---	----	----	----	---	---	---

List Scheduling Algorithm

- ◆ Idea:
 - ◆ Do a topological sort of the dependence DAG.
 - ◆ Consider when an instruction can be scheduled without causing a stall.
 - ◆ Schedule the instruction if it causes no stall and all its predecessors are already scheduled.
- ◆ Optimal list scheduling is NP-complete.
 - ◆ Use heuristics when necessary.

List Scheduling Algorithm

- ◆ Create a dependence DAG of a basic block.
- ◆ Topological Sort.

READY = nodes with no predecessors.

Loop until **READY** is empty.

Schedule each node in **READY** when no stalling

READY += nodes whose predecessors have all been scheduled.

Heuristics for selection

Heuristics for selecting from the READY list:

1. pick the node with the longest path to a leaf in the dependence graph.
2. pick a node with the most immediate successors.
3. pick a node that can go to a less busy pipeline (in a superscalar implementation).

Heuristics for selection

Pick the node with the longest path to a leaf in the dependence graph

Algorithm (for node x)

- ◆ If x has no successors $d_x = 0$
- ◆ $d_x = \text{MAX}(d_y + c_{xy})$ for all successors y of x .

Use reverse breadth-first visiting order

Heuristics for selection

Pick a node with the most immediate successors.

Algorithm (for node x):

- ◆ f_x = number of successors of x

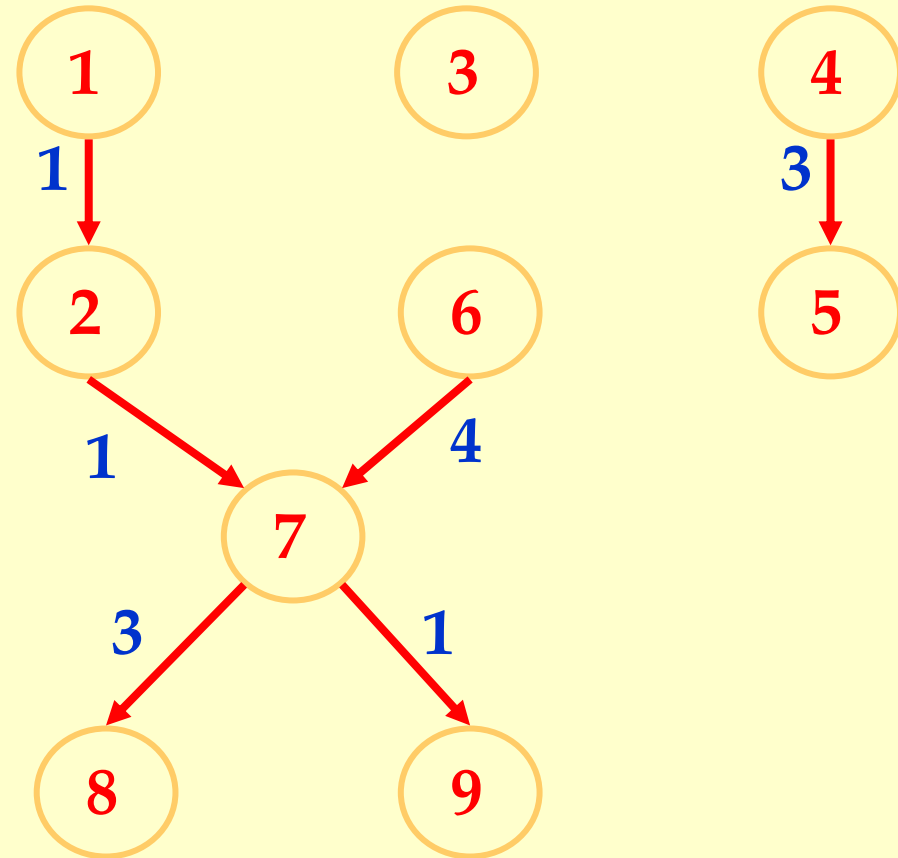
Example

Results available in

1:	LA	r1, array	1 cycle
2:	LD	r2, 4(r1)	1 cycle
3:	AND	r3, r3, 0x00FF	1 cycle
4:	MULC	r6, r6, 100	3 cycles
5:	ST	r7, 4(r6)	
6:	DIVC	r5, r5, 100	4 cycles
7:	ADD	r4, r2, r5	1 cycle
8:	MUL	r5, r2, r4	3 cycles
9:	ST	r4, 0(r1)	

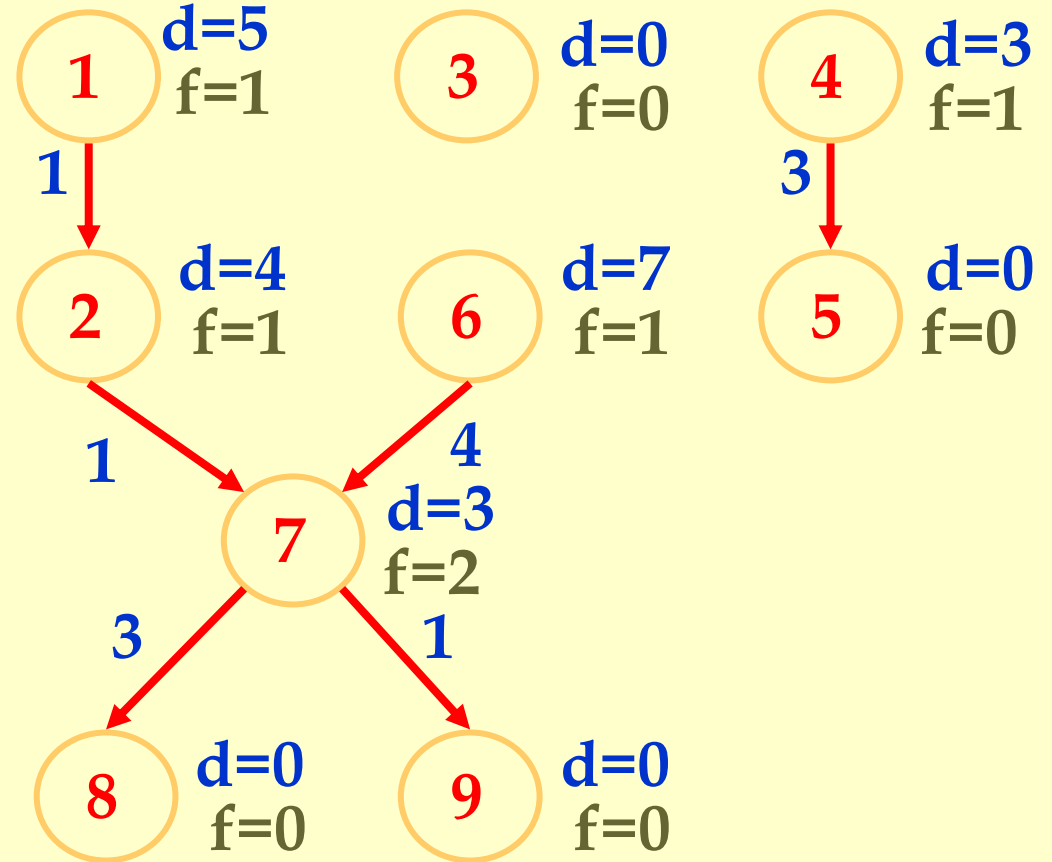
Example

- 1: LA r1, array
- 2: LD r2, 4(r1)
- 3: AND r3, r3, 0x00FF
- 4: MULC r6, r6, 100
- 5: ST r7, 4(r6)
- 6: DIVC r5, r5, 100
- 7: ADD r4, r2, r5
- 8: MUL r5, r2, r4
- 9: ST r4, 0(r1)



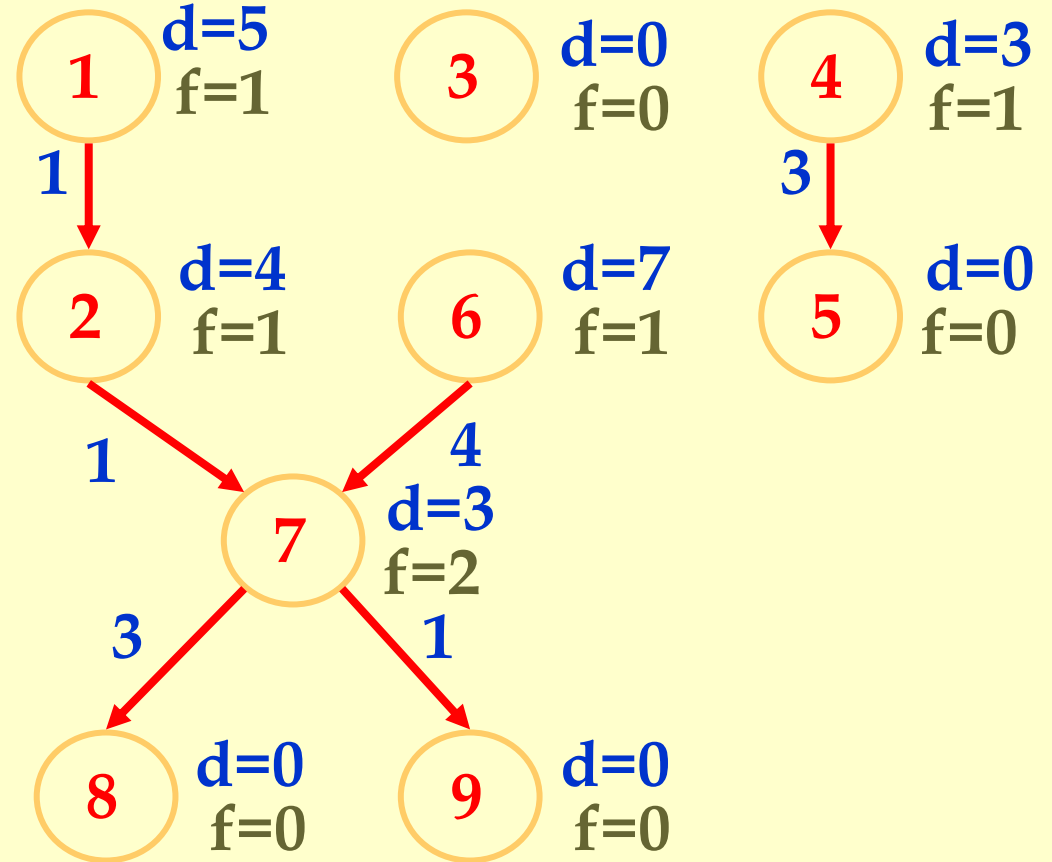
Example

READY = { }



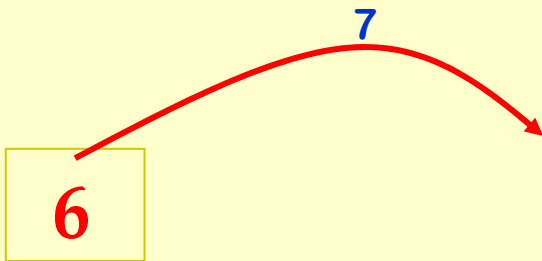
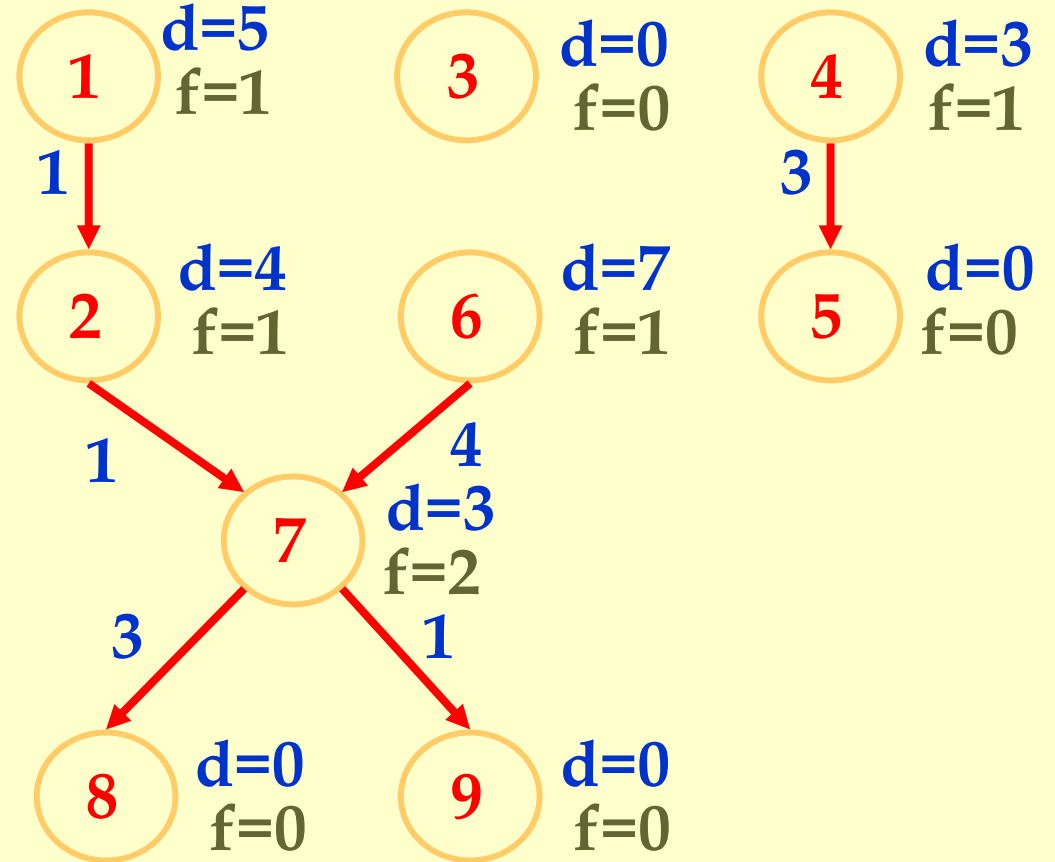
Example

1, 3, 4, 6
READY = { 6, 1, 4, 3 }



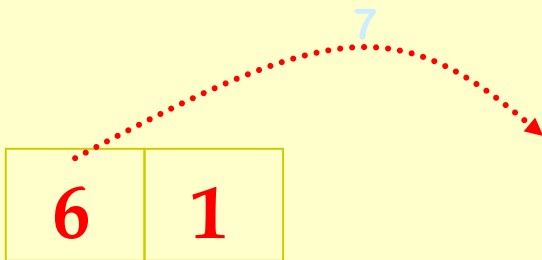
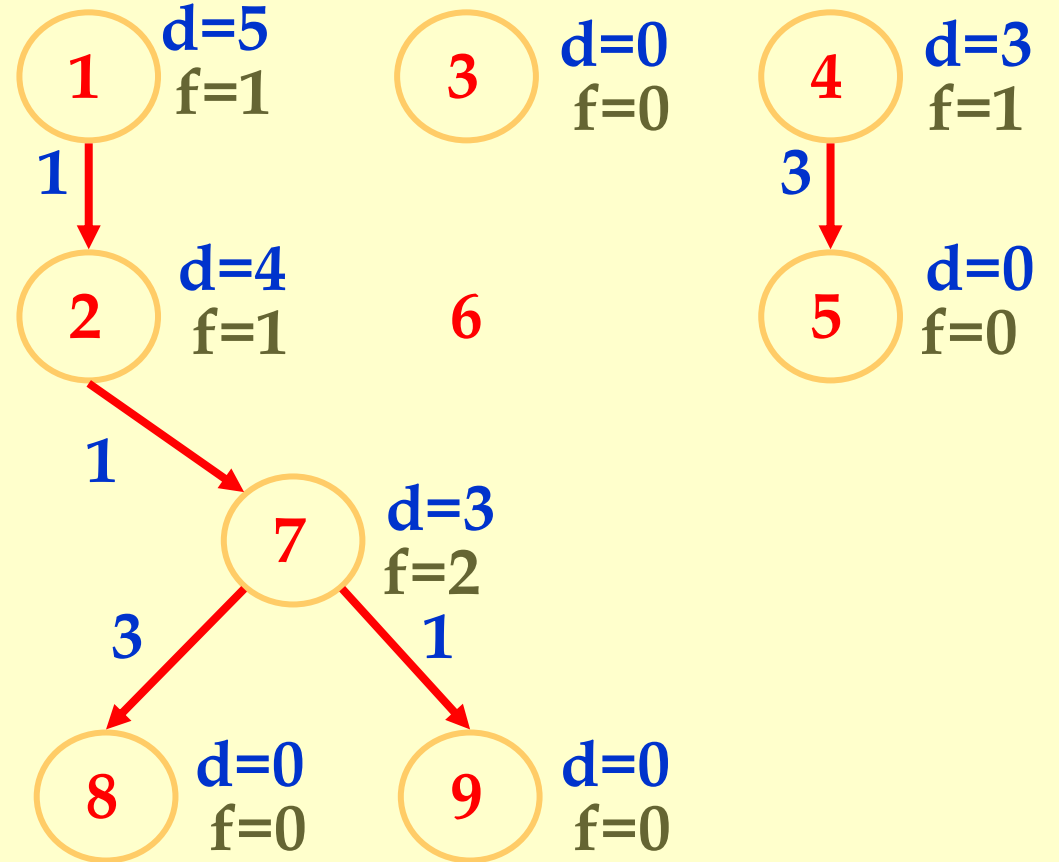
Example

READY = { 6, 1, 4, 3 }



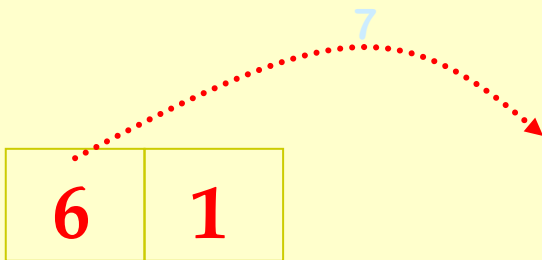
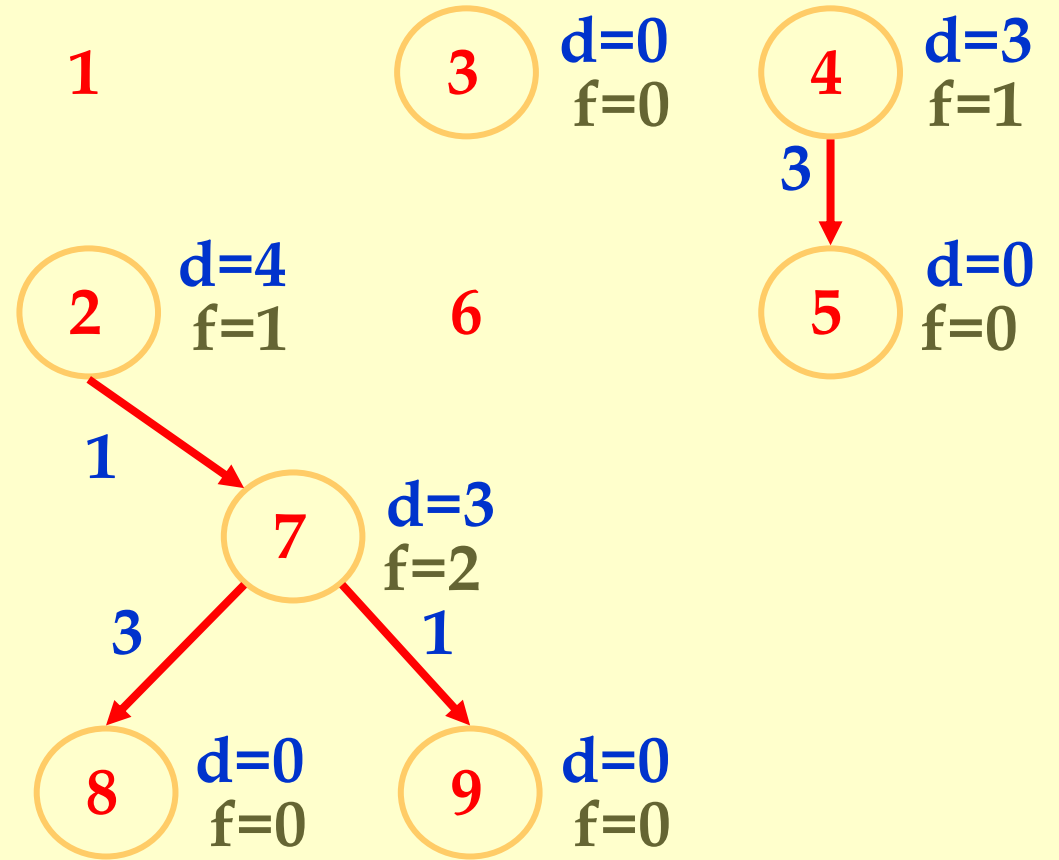
Example

READY = { 1, 4, 3 }



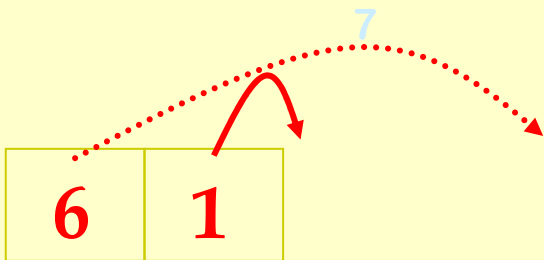
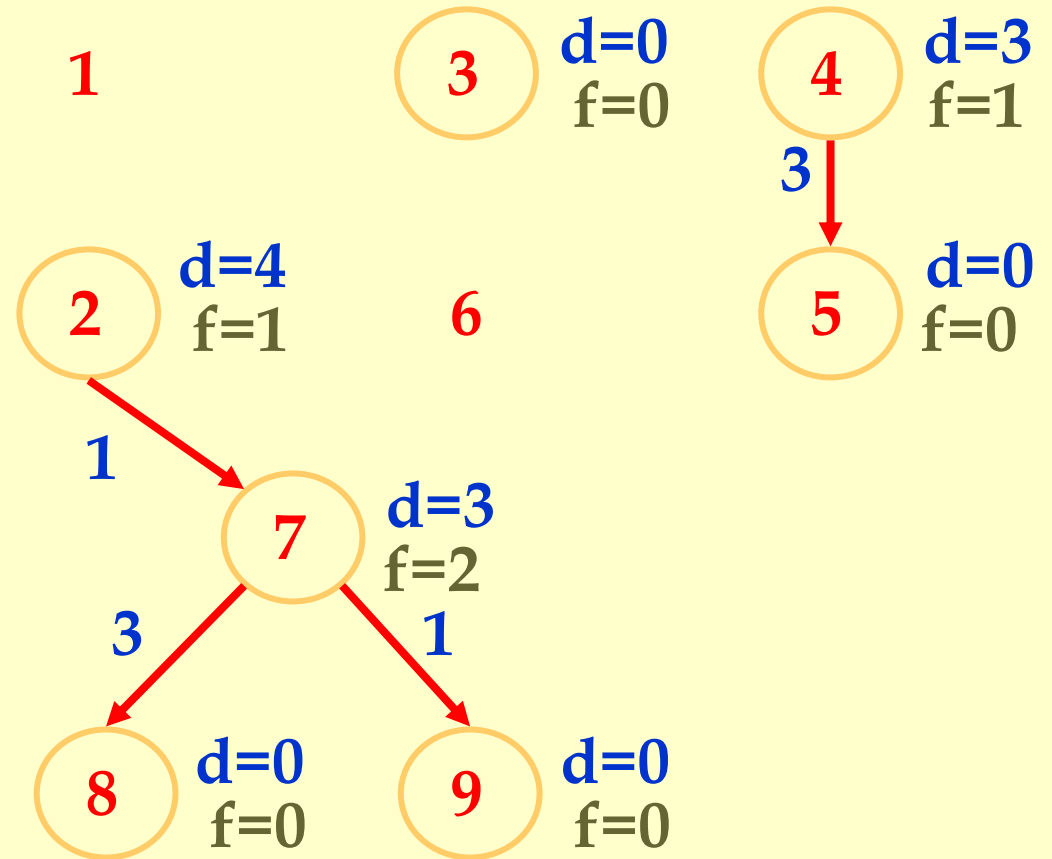
Example

2
READY = { 4, 3 }



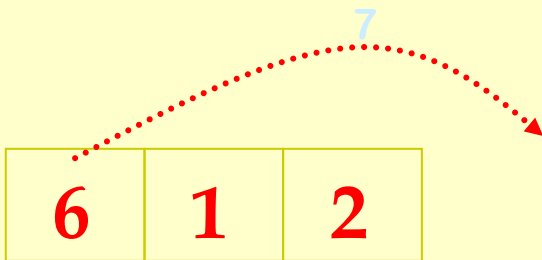
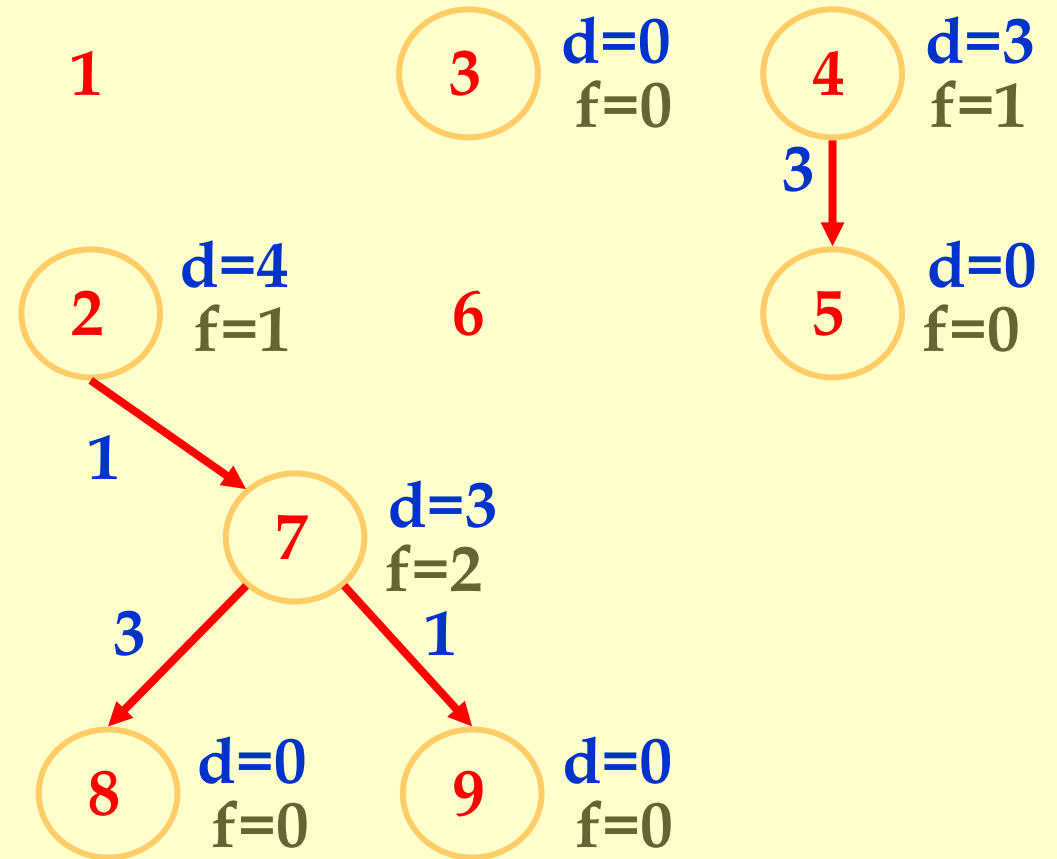
Example

READY = { 2, 4, 3 }



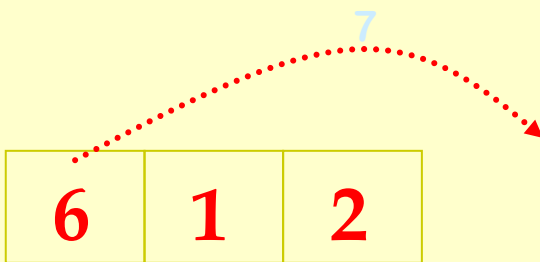
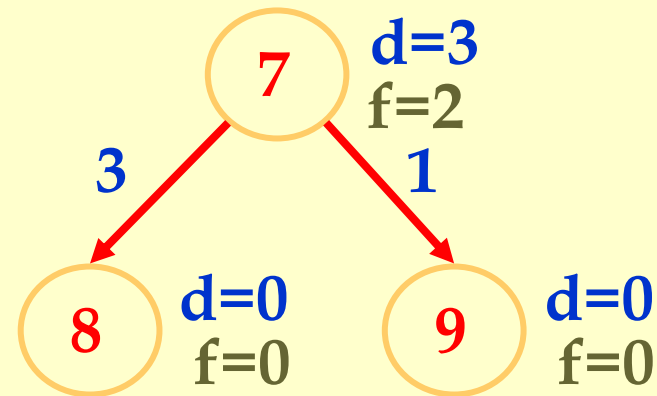
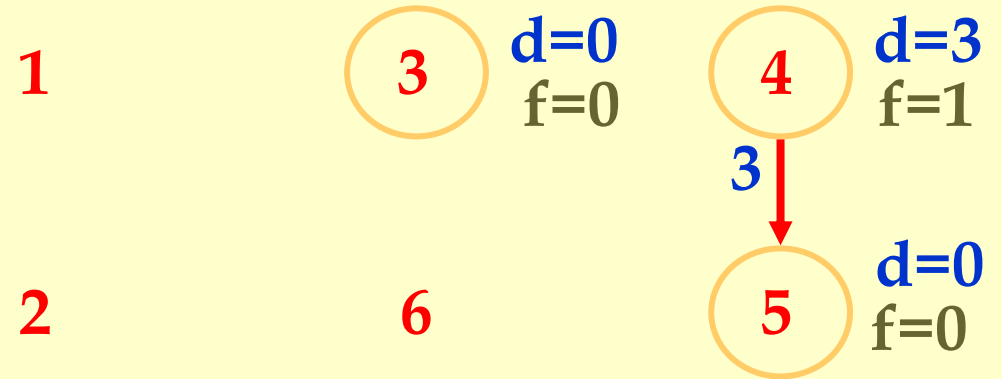
Example

READY = { 2, 4, 3 }



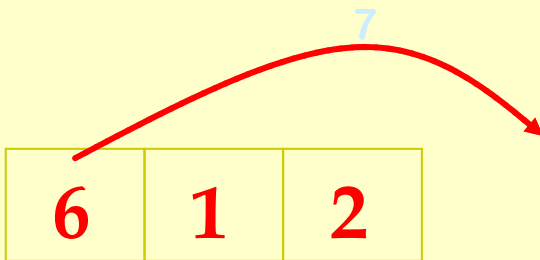
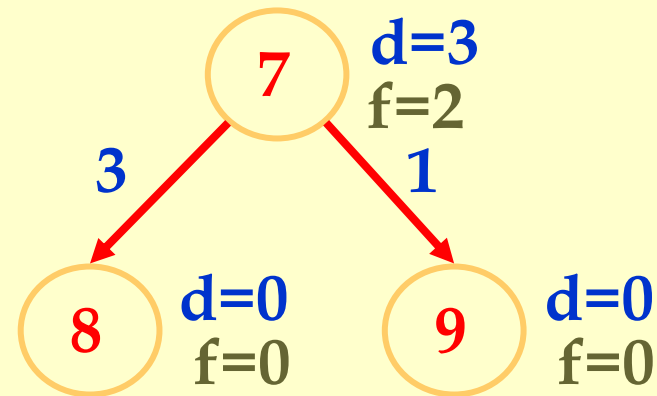
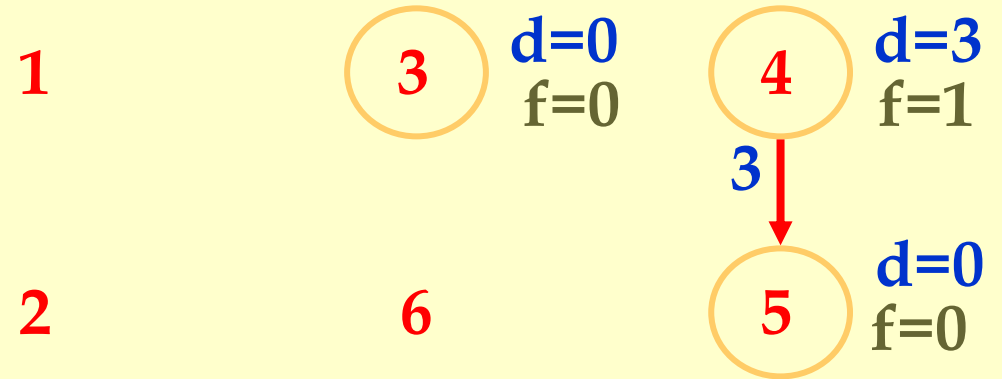
Example

⁷
READY = { 4, 3 }



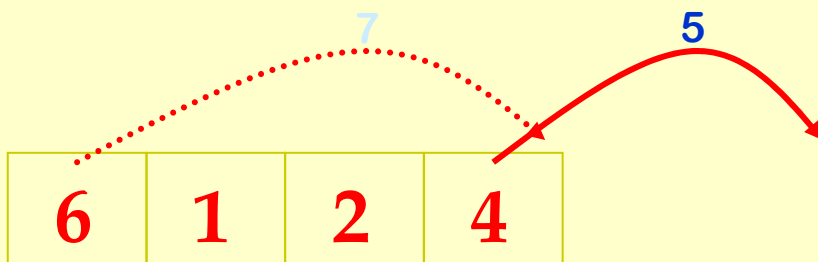
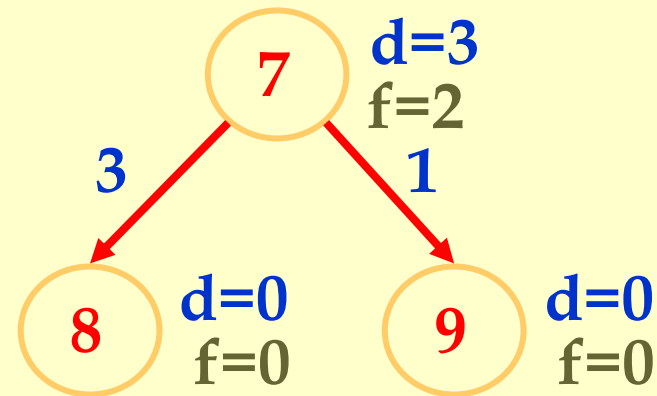
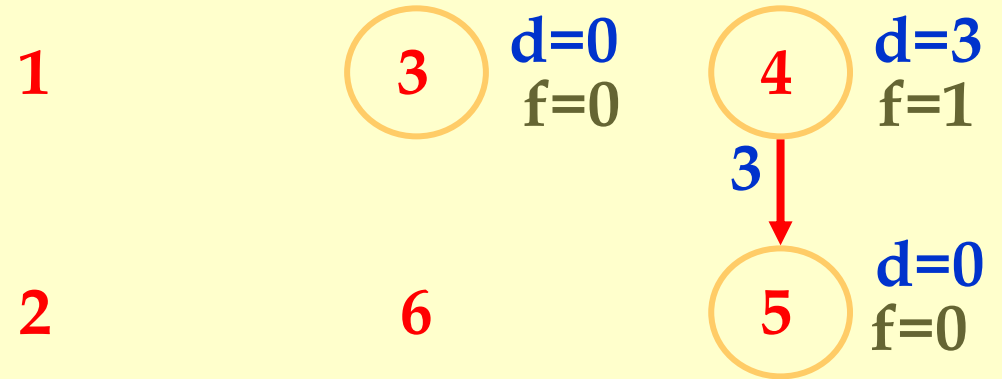
Example

READY = { 7, 4, 3 }



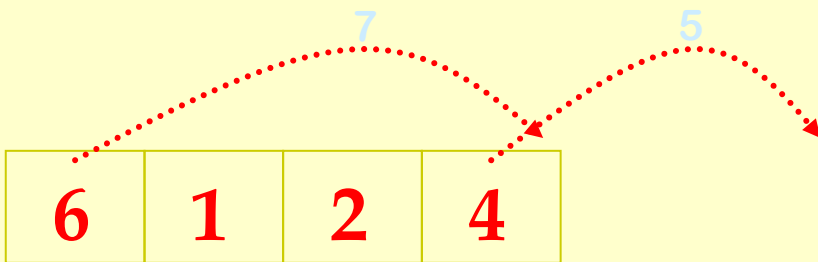
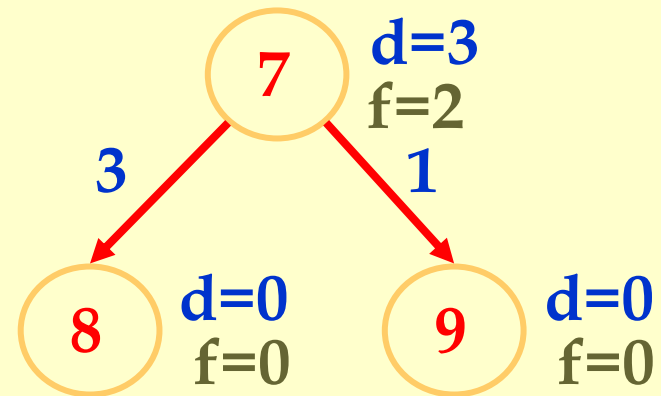
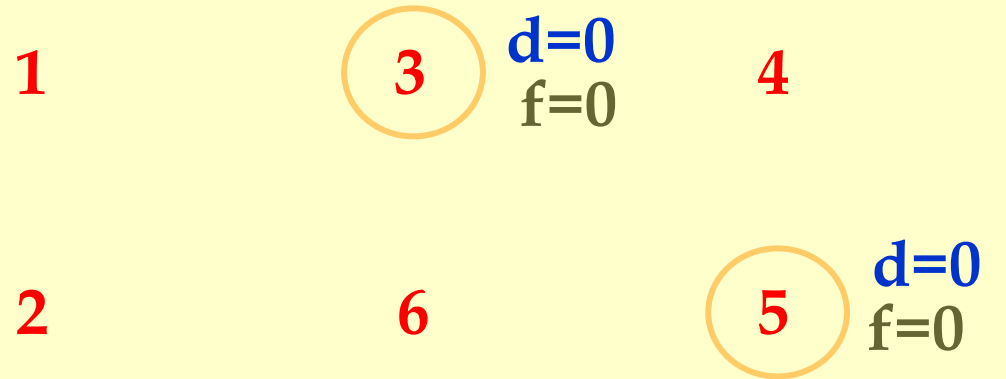
Example

READY = { 7, 4, 3 }



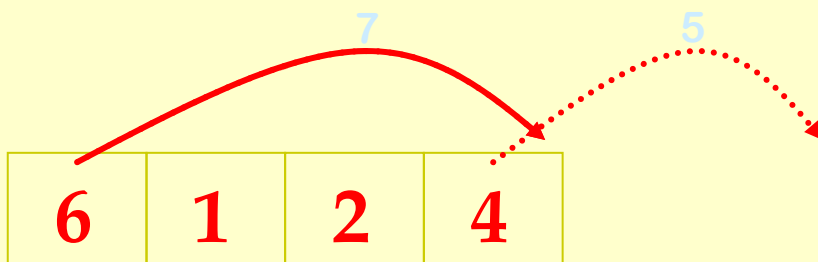
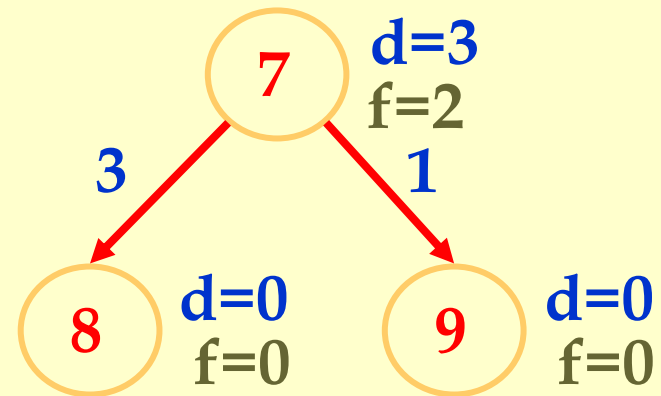
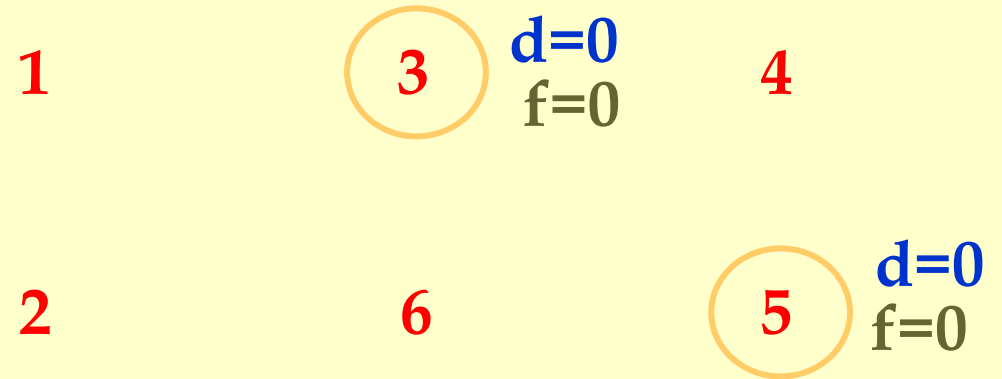
Example

⁵
READY = { 7, 3 }



Example

READY = { 7, 3, 5 }



Example

READY = { 3, 5^{8,9} }

1

3 $d=0$
 $f=0$

4

2

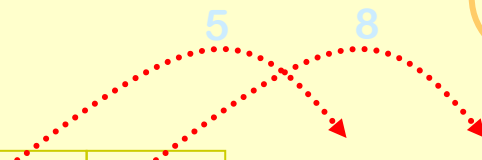
6

5 $d=0$
 $f=0$

7

8 $d=0$
 $f=0$

9 $d=0$
 $f=0$



Example

READY = { 3, 5, 8, 9 }

1

3 $d=0$
 $f=0$

4

2

6

5 $d=0$
 $f=0$

7

8 $d=0$
 $f=0$

9 $d=0$
 $f=0$



Example

READY = { 5, 8, 9 }

1

3

4

2

6

5 $d=0$
 $f=0$

7

8 $d=0$
 $f=0$

9 $d=0$
 $f=0$



Example

READY = { 5, 8, 9 }

1

3

4

2

6

5 $d=0$
 $f=0$

7

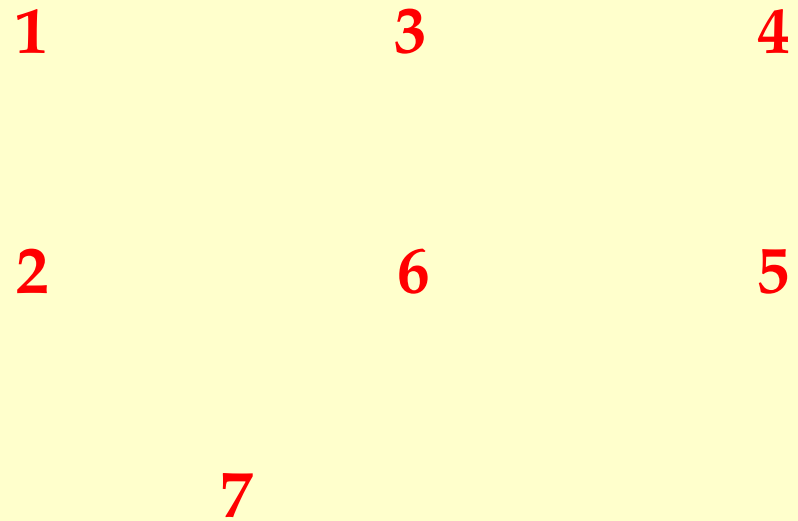
8 $d=0$
 $f=0$

9 $d=0$
 $f=0$



Example

READY = { 8, 9 }



Example

READY = { 8, 9 }

1

3

4

2

6

5

7

8 **d=0**
f=0

9 **d=0**
f=0

6	1	2	4	7	3	5	8
---	---	---	---	---	---	---	---

Example

READY = { 9 }

1

3

4

2

6

5

7

8

9 **d=0**
f=0

6	1	2	4	7	3	5	8	9
---	---	---	---	---	---	---	---	---

Example

READY = { }

1

3

4

2

6

5

7

8

9

6	1	2	4	7	3	5	8	9
---	---	---	---	---	---	---	---	---

Example

1: LA	r1, array	1 cycle
2: LD	r2, 4(r1)	1 cycle
3: AND	r3, r3, 0x00FF	1 cycle
4: MULC	r6, r6, 100	3 cycles
5: ST	r7, 4(r6)	
6: DIVC	r5, r5, 100	4 cycles
7: ADD	r4, r2, r5	1 cycle
8: MUL	r5, r2, r4	3 cycles
9: ST	r4, 0(r1)	

Results available in



14 cycles

*vs.
9 cycles*

Resource Constraints

- ◆ Modern machines have many resource constraints.
- ◆ Superscalar architectures:
 - ◆ can run few parallel operations.
 - ◆ but have constraints.

Resource Constraints of a Superscalar Processor

Example:

- ◆ 1 integer operation, e.g.,
ALUop dest, src1, src2 # in 1 clock cycle

In parallel with

- ◆ 1 memory operation, e.g.,
LD dst, addr # in 2 clock cycles
ST src, addr # in 1 clock cycle

List Scheduling Algorithm with Resource Constraints

- ◆ Represent the superscalar architecture as multiple pipelines.
 - ◆ Each pipeline represents some resource.

List Scheduling Algorithm with Resource Constraints

- ◆ Represent the superscalar architecture as multiple pipelines
 - ◆ Each pipeline represents some resource
- ◆ Example:
 - ◆ One single cycle ALU unit.
 - ◆ One two-cycle pipelined memory unit.

ALUop							
MEM 1							
MEM 2							

List Scheduling Algorithm with Resource Constraints

- ◆ Create a dependence DAG of a basic block.

- ◆ Topological Sort

READY = nodes with no predecessors

Loop until READY is empty

Let $n \in \text{READY}$ be the node with the highest priority

Schedule n in the earliest slot

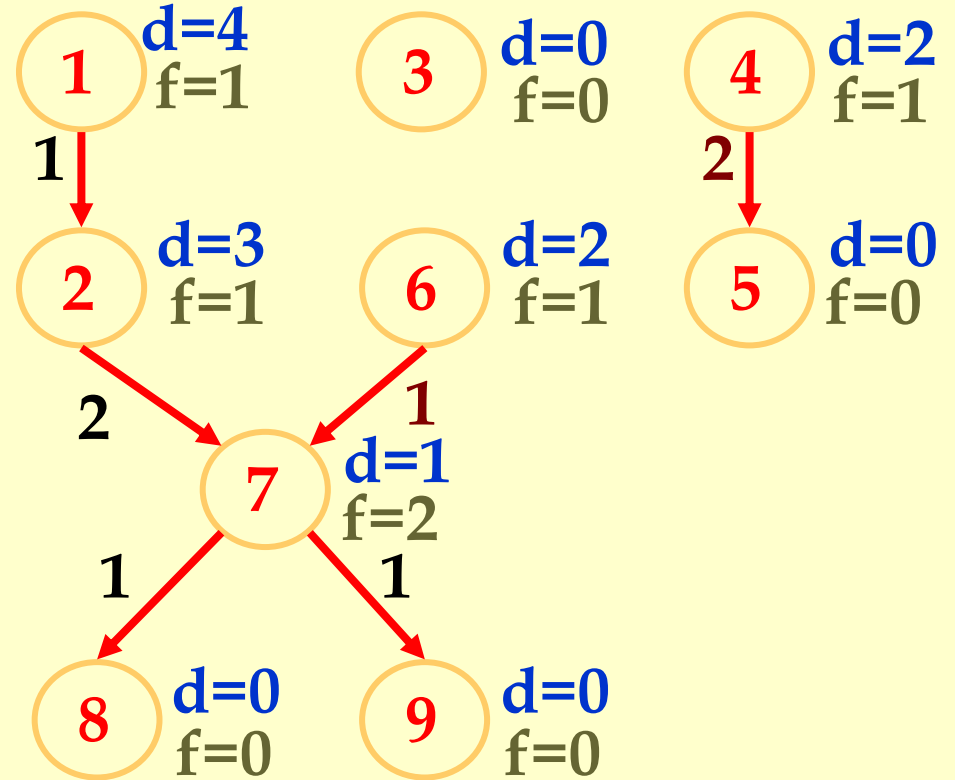
that satisfies precedence + resource constraints

Update READY

Example

(Slightly different from previous example.)

- 1: LA r1, array
- 2: LD r2, 4(r1)
- 3: AND r3, r3, 0x00FF
- 4: LD r6, 8(sp)
- 5: ST r7, 4(r6)
- 6: ADD r5, r5, 100
- 7: ADD r4, r2, r5
- 8: MUL r5, r2, r4
- 9: ST r4, 0(r1)



READY = { 1, 6, 4, 3 }

ALUop

1

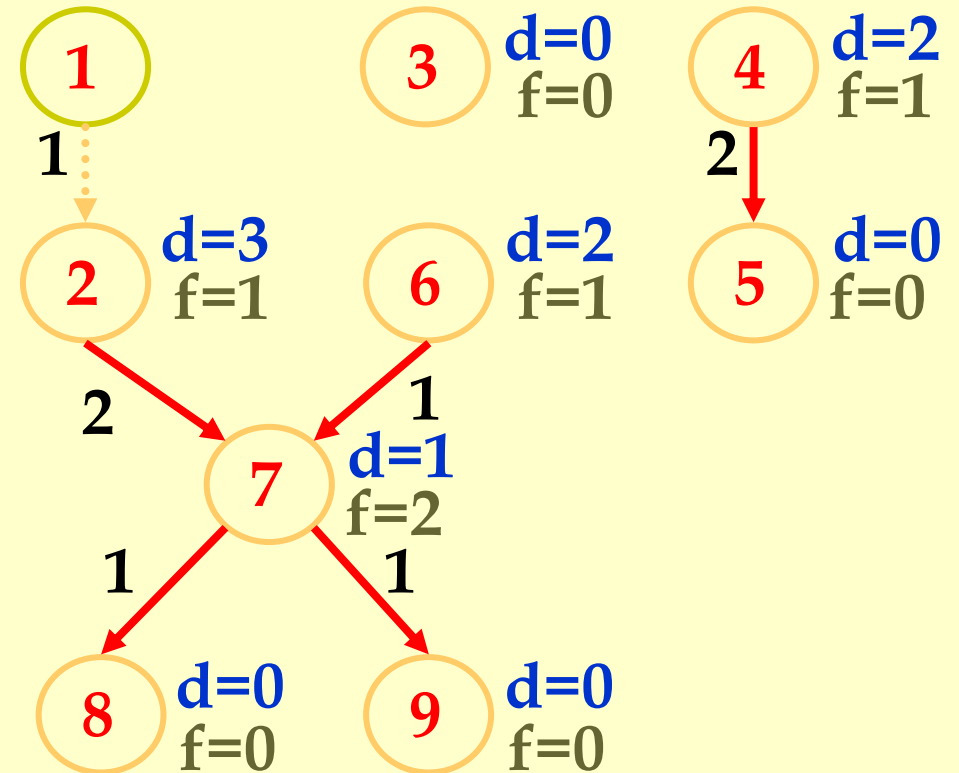
MEM 1

MEM 2

1						

Example

- 1: LA r1, array
- 2: LD r2, 4(r1)
- 3: AND r3, r3, 0x00FF
- 4: LD r6, 8(sp)
- 5: ST r7, 4(r6)
- 6: ADD r5, r5, 100
- 7: ADD r4, r2, r5
- 8: MUL r5, r2, r4
- 9: ST r4, 0(r1)



READY = { 6, 4, 3 } ← 2

ALUop

MEM 1

MEM 2

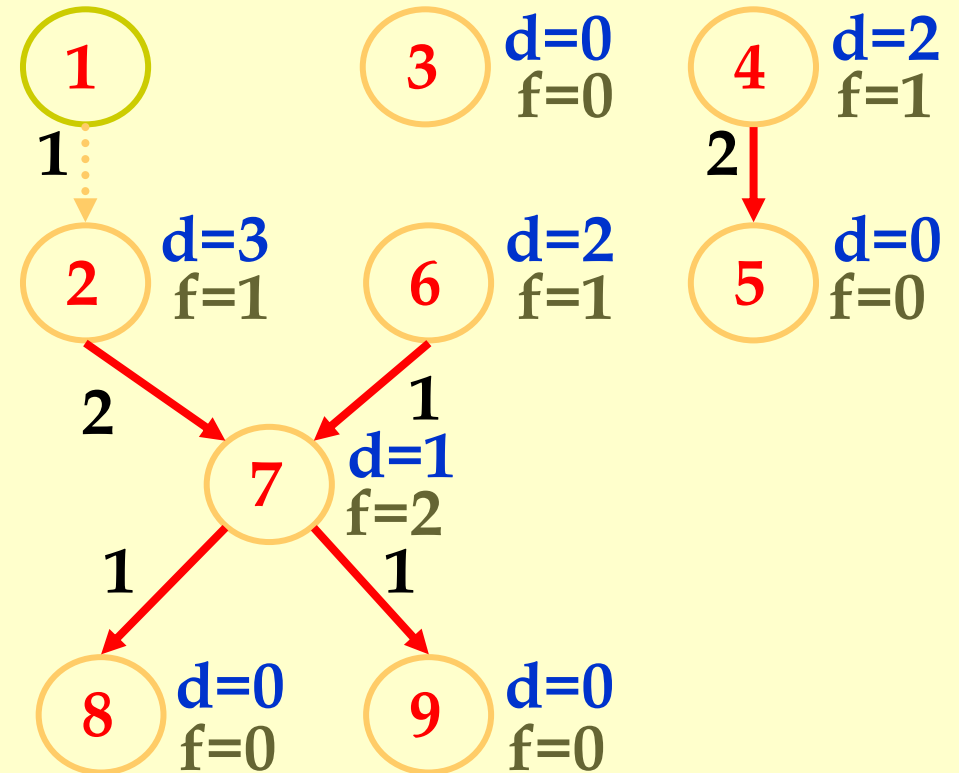
1						

Example

```

1: LA      r1, array
2: LD      r2, 4(r1)
3: AND     r3, r3, 0x00FF
4: LD      r6, 8(sp)
5: ST      r7, 4(r6)
6: ADD     r5, r5, 100
7: ADD     r4, r2, r5
8: MUL     r5, r2, r4
9: ST      r4, 0(r1)
    
```

READY = { 2, 6, 4, 3 }



ALUop

MEM 1

MEM 2

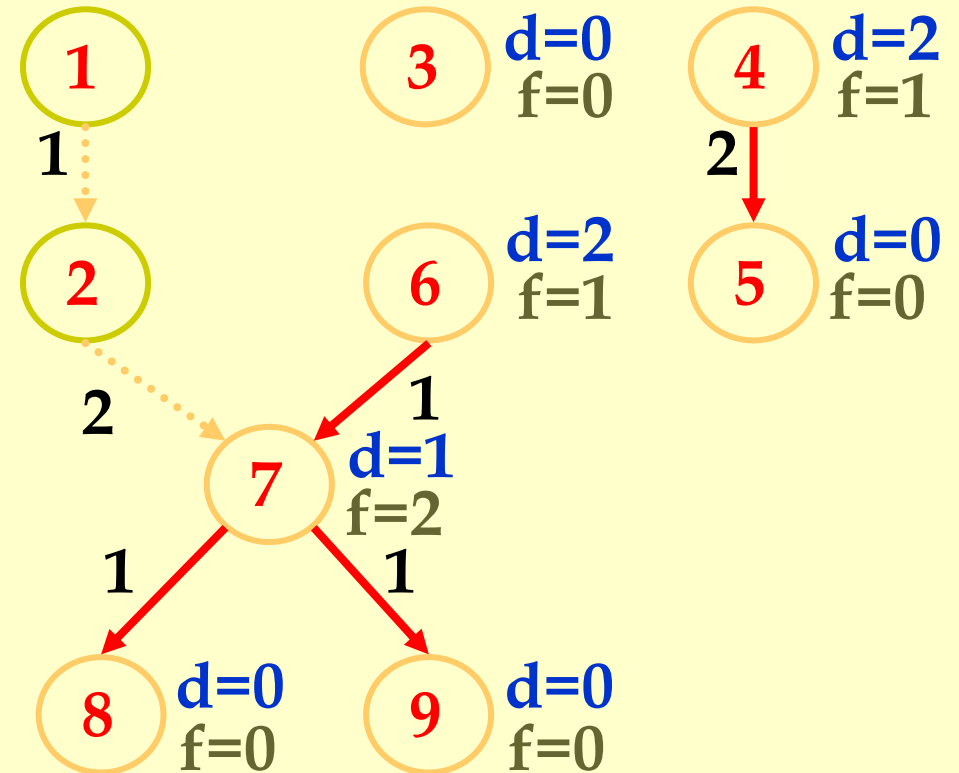
1						
	2					
		2				

Example

```

1: LA      r1, array
2: LD      r2, 4(r1)
3: AND     r3, r3, 0x00FF
4: LD      r6, 8(sp)
5: ST      r7, 4(r6)
6: ADD     r5, r5, 100
7: ADD     r4, r2, r5
8: MUL     r5, r2, r4
9: ST      r4, 0(r1)
    
```

READY = { 6, 4, 3 }



ALUop

1 6

MEM 1

2

MEM 2

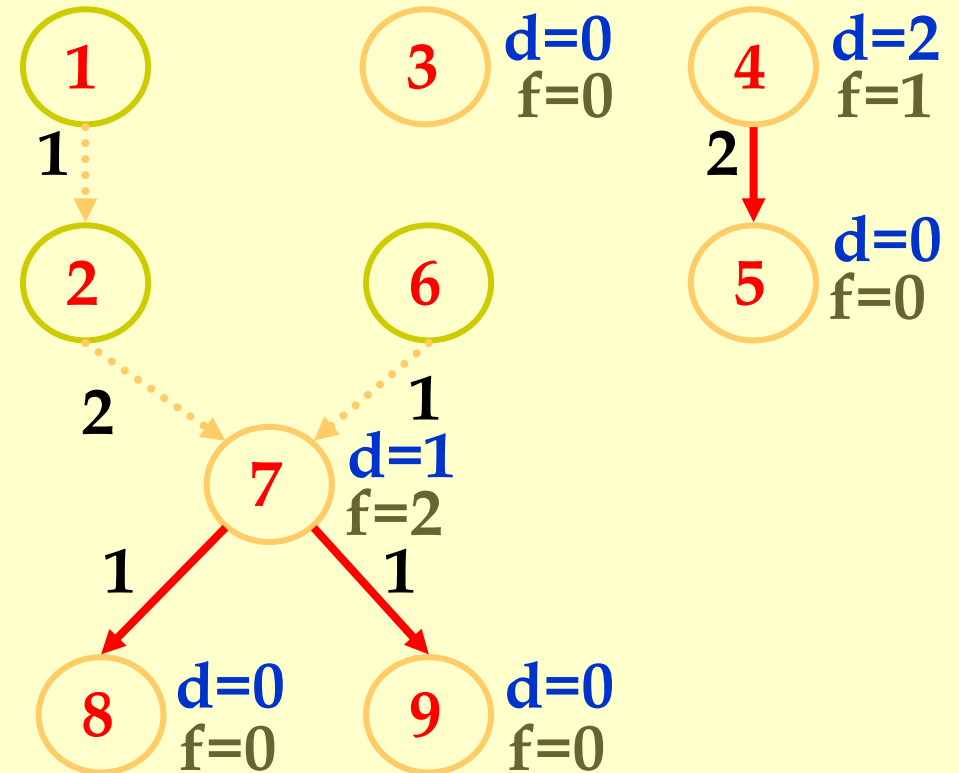
2

1	6					
	2					
		2				

Example

```

1: LA      r1, array
2: LD      r2, 4(r1)
3: AND     r3, r3, 0x00FF
4: LD      r6, 8(sp)
5: ST      r7, 4(r6)
6: ADD     r5, r5, 100
7: ADD     r4, r2, r5
8: MUL     r5, r2, r4
9: ST      r4, 0(r1)
    
```



READY = { 4, 3 } ← 7

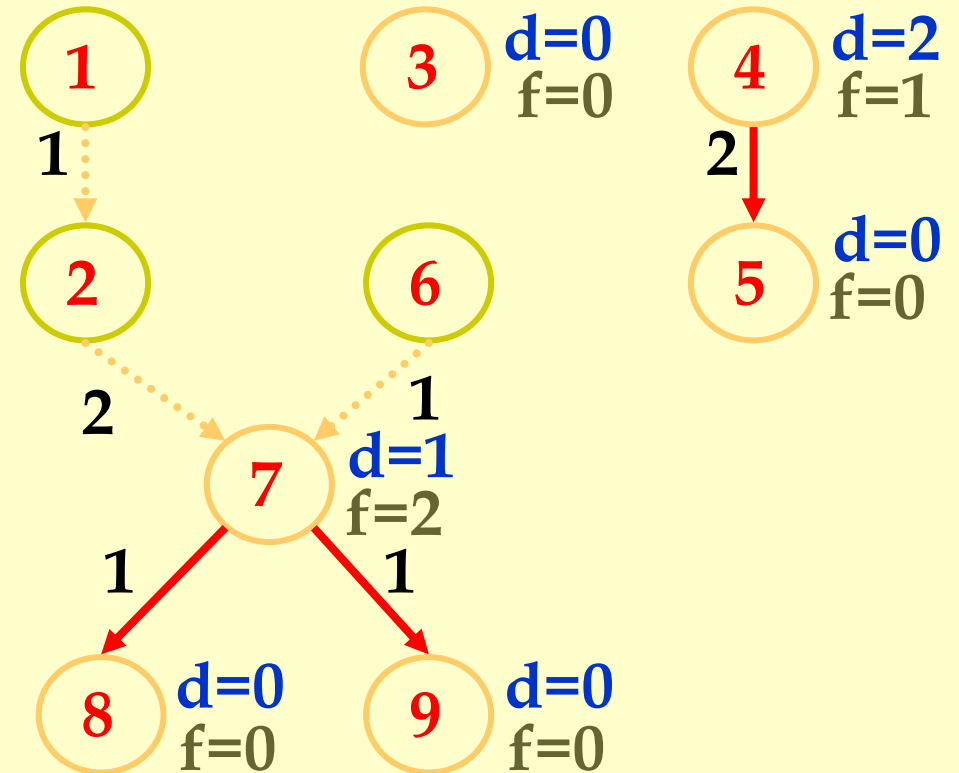
ALUop	1	6				
MEM 1		2				
MEM 2			2			

Example

```

1: LA      r1, array
2: LD      r2, 4(r1)
3: AND     r3, r3, 0x00FF
4: LD      r6, 8(sp)
5: ST      r7, 4(r6)
6: ADD     r5, r5, 100
7: ADD     r4, r2, r5
8: MUL     r5, r2, r4
9: ST      r4, 0(r1)
    
```

READY = { 4, 7, 3 }



ALUop

1 6

MEM 1

4 2

MEM 2

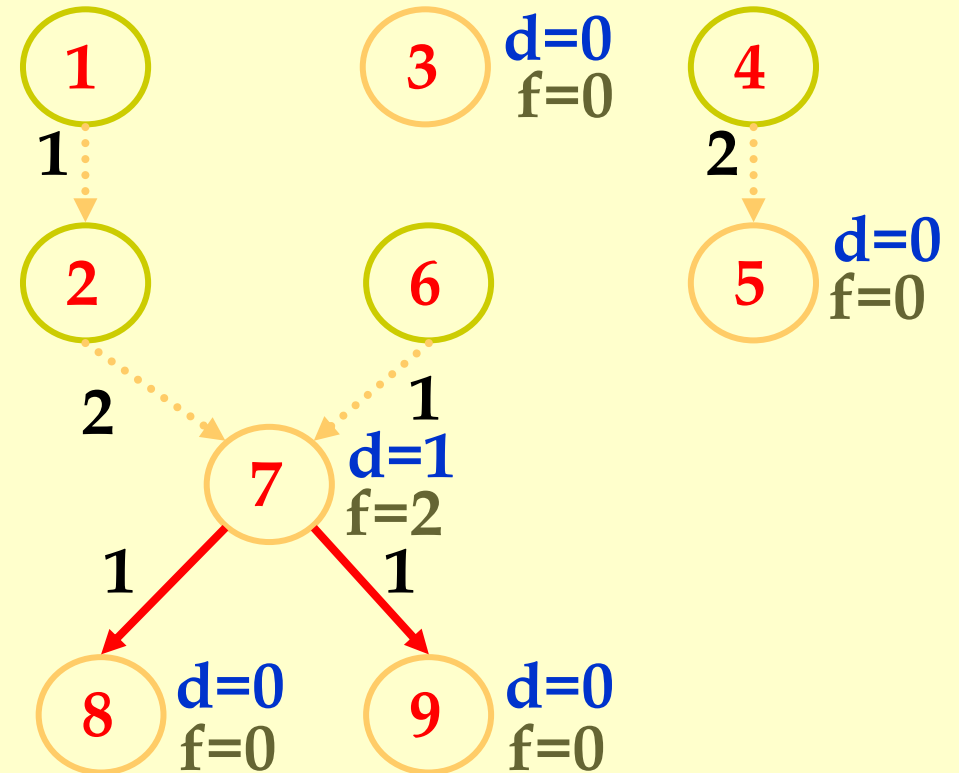
4 2

1	6					
4	2					
	4	2				

Example

```

1: LA      r1, array
2: LD      r2, 4(r1)
3: AND     r3, r3, 0x00FF
4: LD      r6, 8(sp)
5: ST      r7, 4(r6)
6: ADD     r5, r5, 100
7: ADD     r4, r2, r5
8: MUL     r5, r2, r4
9: ST      r4, 0(r1)
    
```



READY = { 7, 3 } ← 5

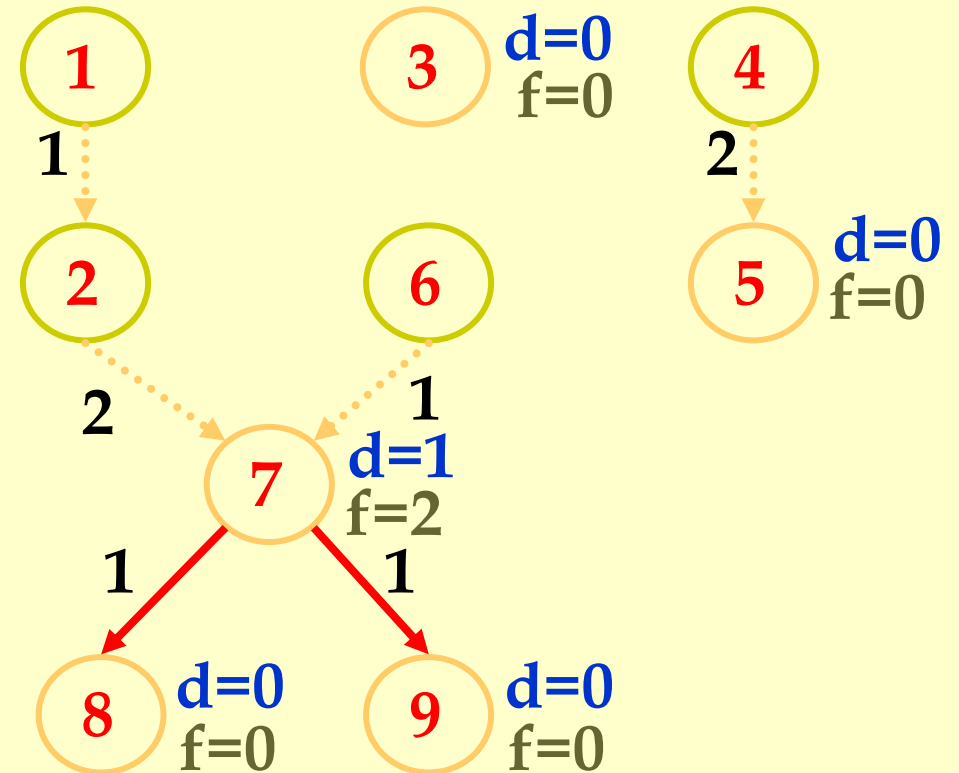
ALUop	1	6				
MEM 1	4	2				
MEM 2		4	2			

Example

```

1: LA      r1, array
2: LD      r2, 4(r1)
3: AND     r3, r3, 0x00FF
4: LD      r6, 8(sp)
5: ST      r7, 4(r6)
6: ADD     r5, r5, 100
7: ADD     r4, r2, r5
8: MUL     r5, r2, r4
9: ST      r4, 0(r1)
    
```

READY = { 7, 3, 5 }



ALUop

1

6

7

MEM 1

4

2

MEM 2

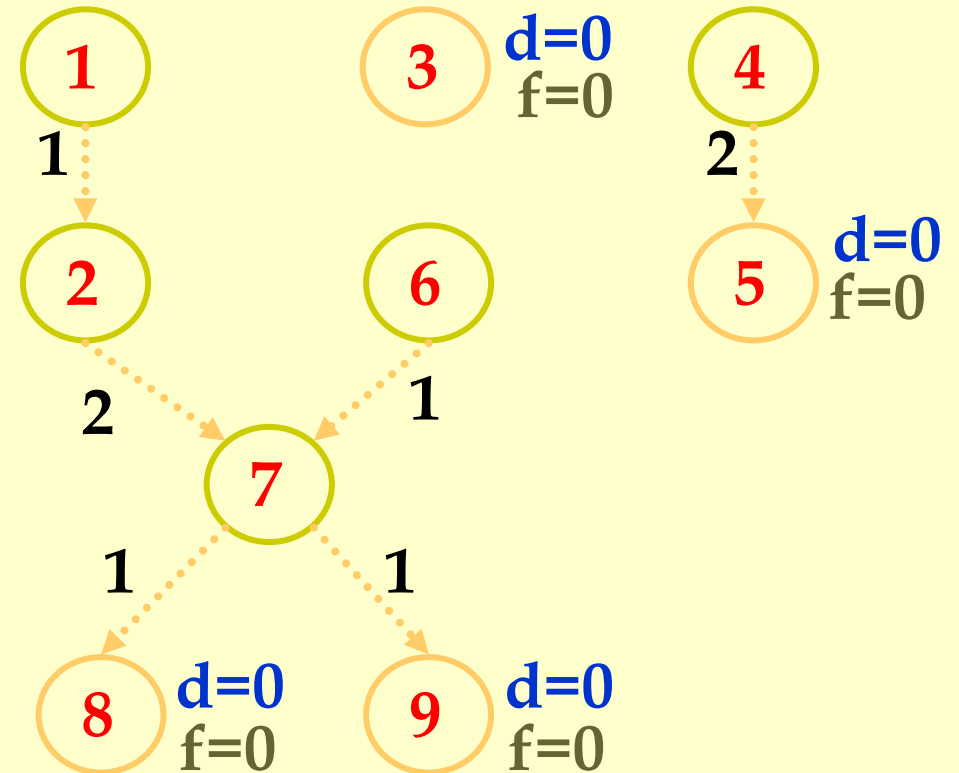
4

2

Example

```

1: LA      r1, array
2: LD      r2, 4(r1)
3: AND     r3, r3, 0x00FF
4: LD      r6, 8(sp)
5: ST      r7, 4(r6)
6: ADD     r5, r5, 100
7: ADD     r4, r2, r5
8: MUL     r5, r2, r4
9: ST      r4, 0(r1)
    
```



READY = { 3, 5 } ← 8, 9

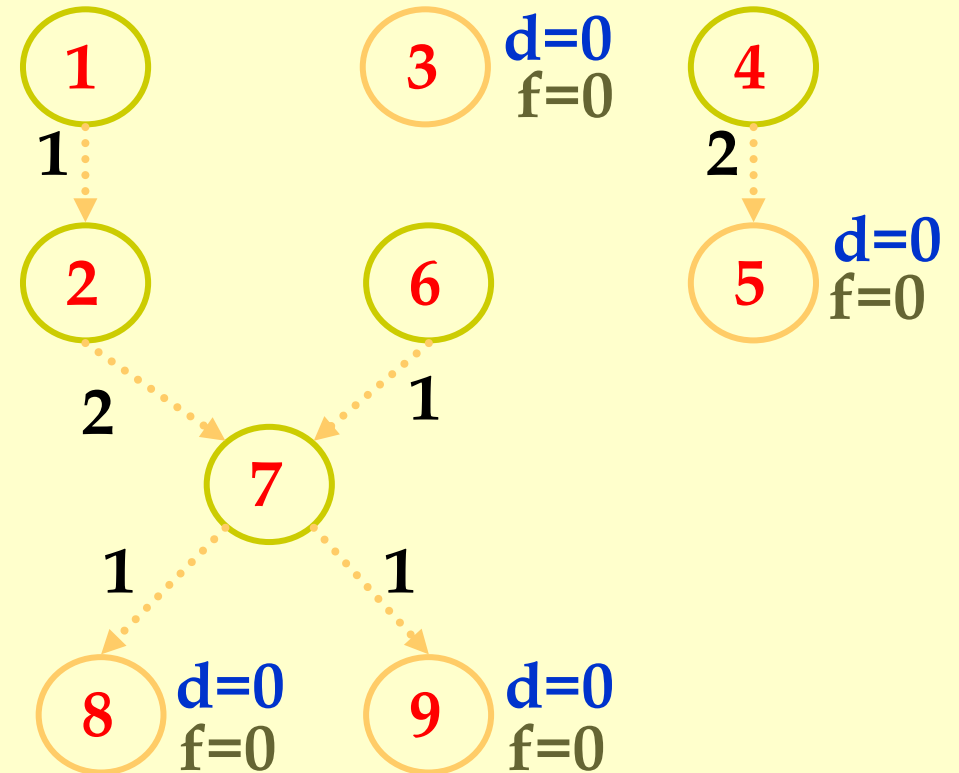
ALUop	1	6		7			
MEM 1	4	2					
MEM 2		4	2				

Example

```

1: LA      r1, array
2: LD      r2, 4(r1)
3: AND     r3, r3, 0x00FF
4: LD      r6, 8(sp)
5: ST      r7, 4(r6)
6: ADD     r5, r5, 100
7: ADD     r4, r2, r5
8: MUL     r5, r2, r4
9: ST      r4, 0(r1)
    
```

READY = { 3, 5, 8, 9 }



ALUop

1	6	3	7			
----------	----------	----------	----------	--	--	--

MEM 1

4	2					
----------	----------	--	--	--	--	--

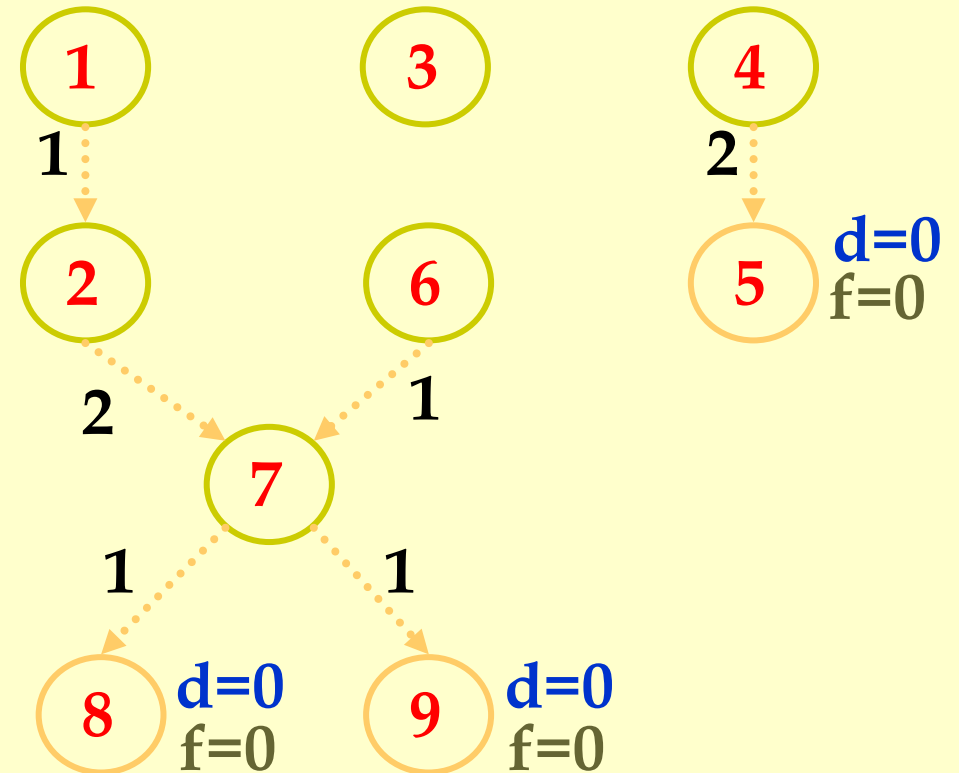
MEM 2

4	2					
----------	----------	--	--	--	--	--

Example

```

1: LA      r1, array
2: LD      r2, 4(r1)
3: AND     r3, r3, 0x00FF
4: LD      r6, 8(sp)
5: ST      r7, 4(r6)
6: ADD     r5, r5, 100
7: ADD     r4, r2, r5
8: MUL     r5, r2, r4
9: ST      r4, 0(r1)
    
```



READY = { 5, 8, 9 }

ALUop

MEM 1

MEM 2

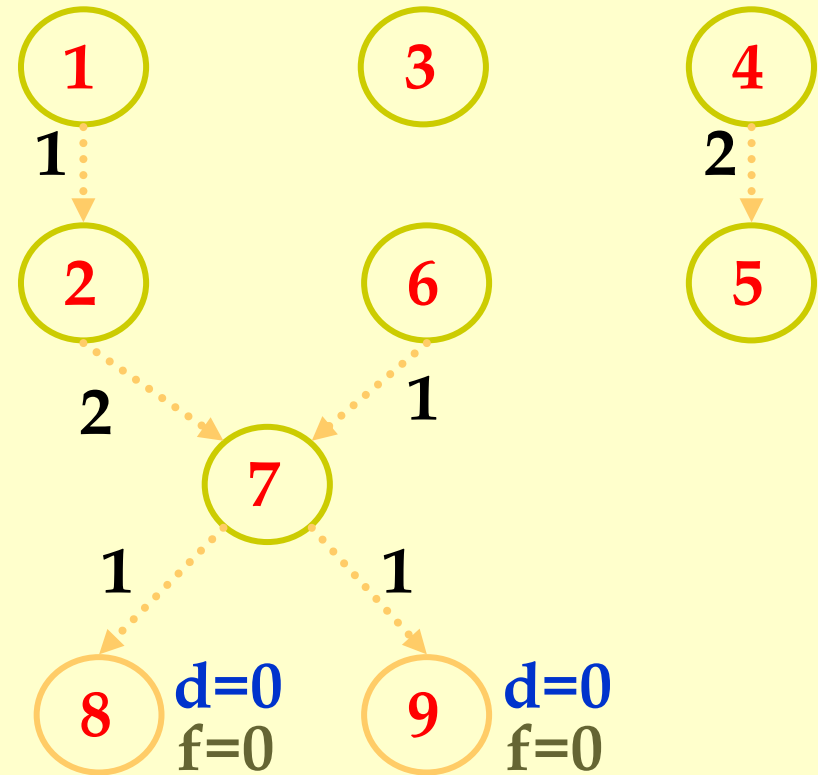
1	6	3	7			
4	2	5				
	4	2				

Example

```

1: LA      r1, array
2: LD      r2, 4(r1)
3: AND     r3, r3, 0x00FF
4: LD      r6, 8(sp)
5: ST      r7, 4(r6)
6: ADD     r5, r5, 100
7: ADD     r4, r2, r5
8: MUL     r5, r2, r4
9: ST      r4, 0(r1)
    
```

READY = { 8, 9 }



ALUOp

1	6	3	7	8		
---	---	---	---	---	--	--

MEM 1

4	2	5				
---	---	---	--	--	--	--

MEM 2

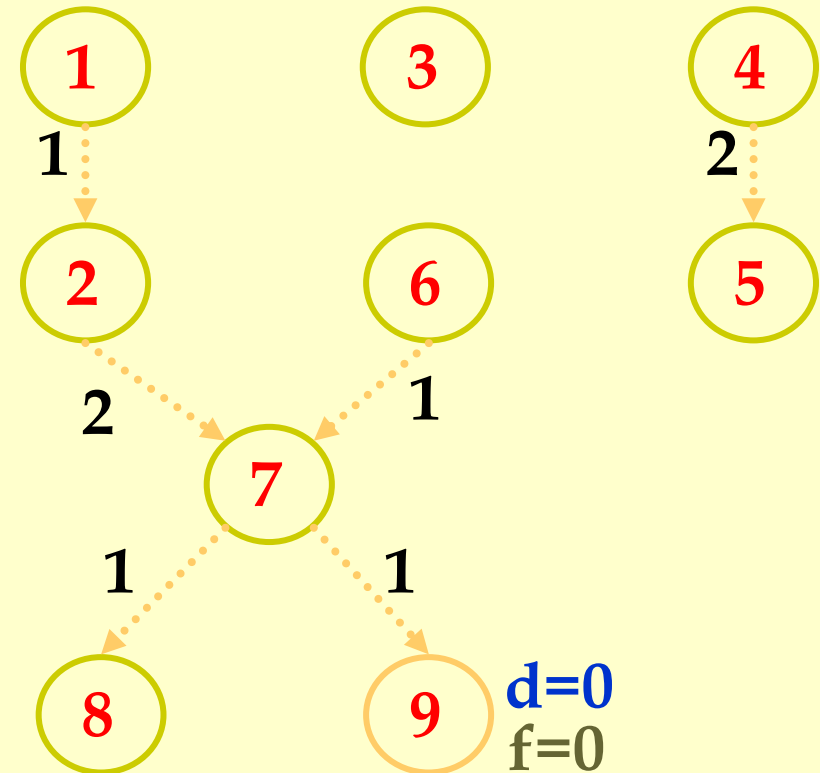
	4	2				
--	---	---	--	--	--	--

Example

```

1: LA      r1, array
2: LD      r2, 4(r1)
3: AND     r3, r3, 0x00FF
4: LD      r6, 8(sp)
5: ST      r7, 4(r6)
6: ADD     r5, r5, 100
7: ADD     r4, r2, r5
8: MUL     r5, r2, r4
9: ST      r4, 0(r1)
    
```

READY = { 9 }



ALUOp

1	6	3	7	8		
4	2	5		9		
	4	2				

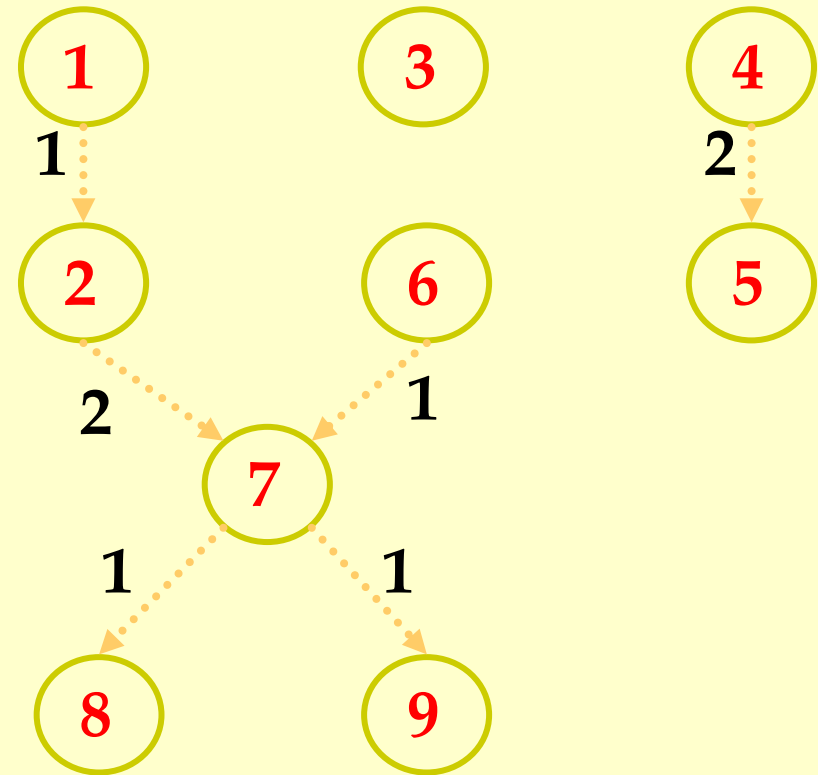
MEM 1

MEM 2

Example

```

1: LA      r1,array
2: LD      r2,4(r1)
3: AND     r3,r3,0x00FF
4: LD      r6,8(sp)
5: ST      r7,4(r6)
6: ADD     r5,r5,100
7: ADD     r4,r2,r5
8: MUL     r5,r2,r4
9: ST      r4,0(r1)
    
```



READY = { }

ALUop	1	6	3	7	8		
MEM 1	4	2	5		9		
MEM 2		4	2				

Register Allocation and Instruction Scheduling

- ◆ If register allocation is performed before instruction scheduling:
 - ◆ the choices for scheduling are restricted.

Example

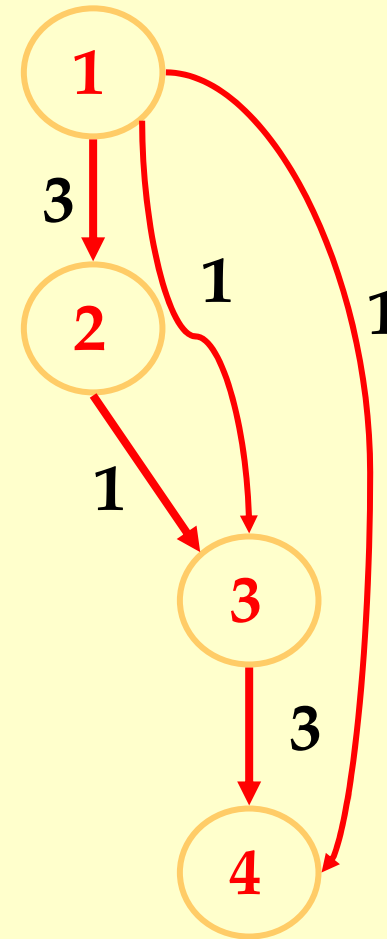
1: LD r2, 0(r1)
2: ADD r3, r3, r2
3: LD r2, 4(r5)
4: ADD r6, r6, r2

ALUop

MEM 1

MEM 2

		2			4
1			3		
	1			3	

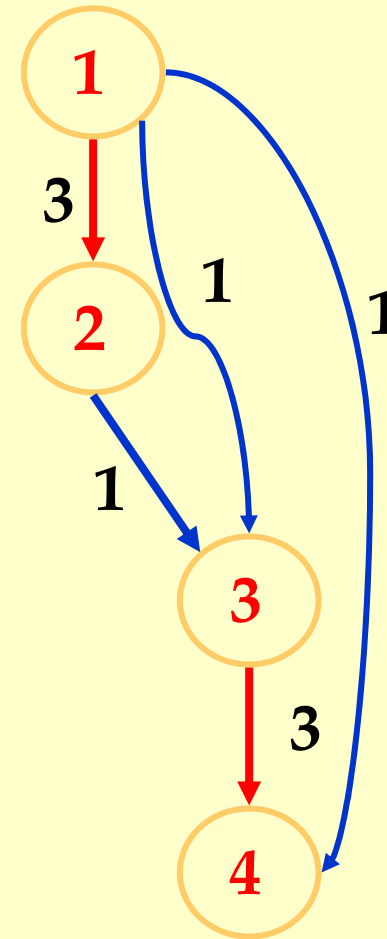


Example

```

1: LD      r2, 0(r1)
2: ADD    r3, r3, r2
3: LD      r2, 4(r5)
4: ADD    r6, r6, r2
    
```

False dependencies
(Anti-dependencies)

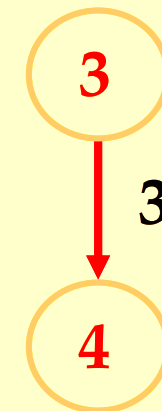
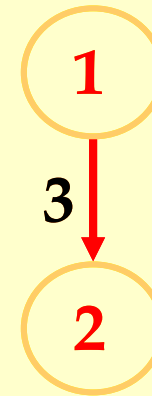


How about using a different register?

Example

```

1: LD      r2, 0(r1)
2: ADD    r3, r3, r2
3: LD     r4, 4(r5)
4: ADD    r6, r6, r4
    
```



ALUop

MEM 1

MEM 2

		2	4
1	3		
	1	3	

Register Allocation and Instruction Scheduling

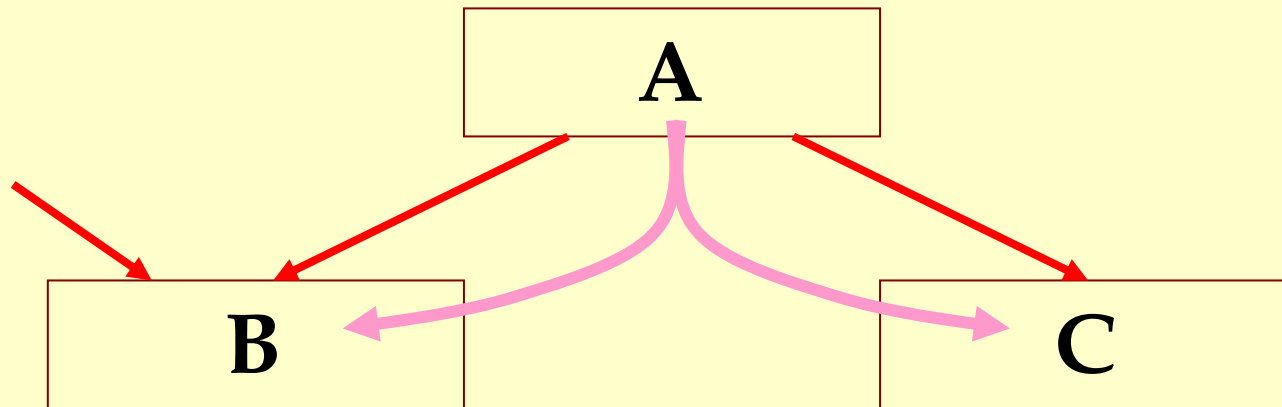
- ◆ If register allocation is performed before instruction scheduling:
 - ◆ the choices for scheduling are restricted.
- ◆ If instruction scheduling is performed before register allocation:
 - ◆ register allocation may spill registers.
 - ◆ will change the carefully done schedule!!!

Scheduling across basic blocks

- ◆ Number of instructions in a basic block is small.
 - ◆ Cannot keep a multiple units with long pipelines busy by just scheduling within a basic block.
- ◆ Need to handle control dependencies.
 - ◆ Scheduling constraints across basic blocks.
 - ◆ Scheduling policy.

Moving across basic blocks

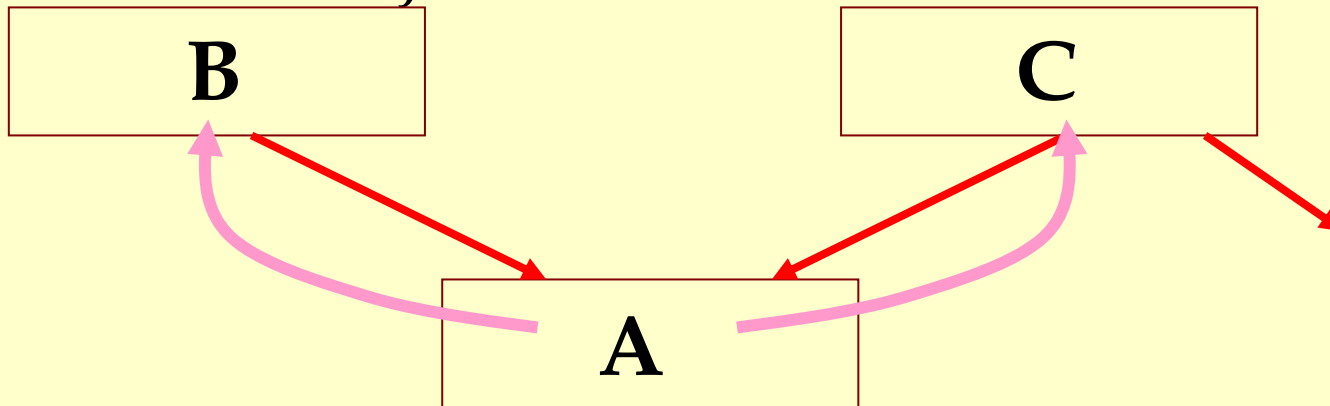
Downward to adjacent basic block



A path to **B** that does not execute **A**?

Moving across basic blocks

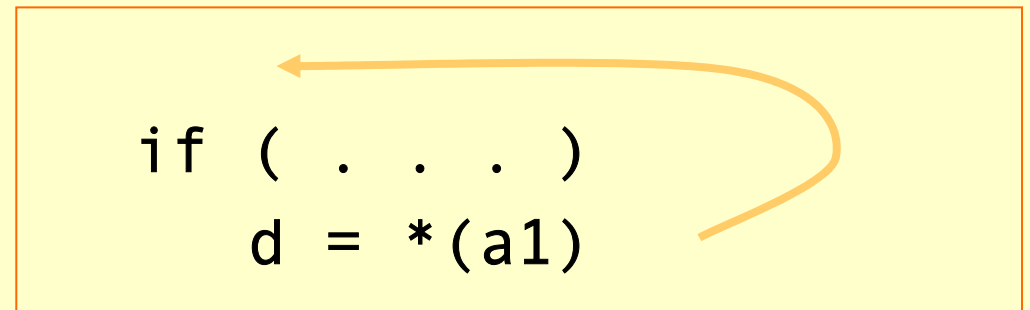
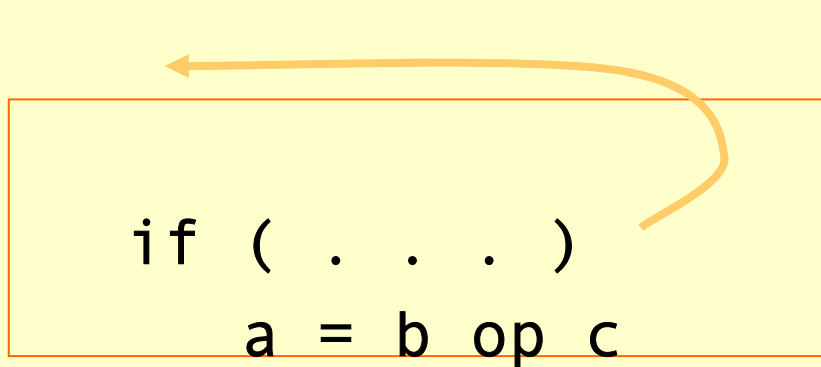
Upward to adjacent basic block



A path from C that does not reach A?

Control Dependencies

Constraints in moving instructions across basic blocks



Not allowed if e.g.

```

if ( c != 0 )
  a = b / c

```

Not allowed if e.g.

```

if ( valid_address(a1) )
  d = *(a1)

```

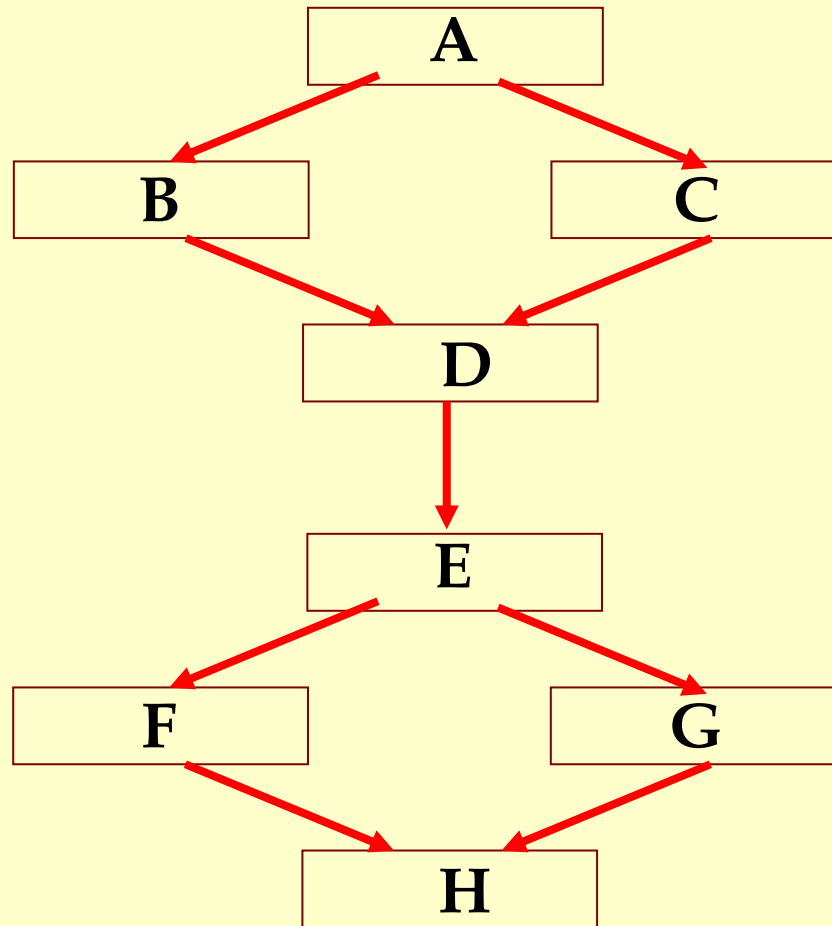
Outline

- ◆ Modern architectures
- ◆ Delay slots
- ◆ Introduction to instruction scheduling
- ◆ List scheduling
- ◆ Resource constraints
- ◆ Interaction with register allocation
- ◆ Scheduling across basic blocks
- ◆ Trace scheduling
- ◆ Scheduling for loops
- ◆ Loop unrolling
- ◆ Software pipelining

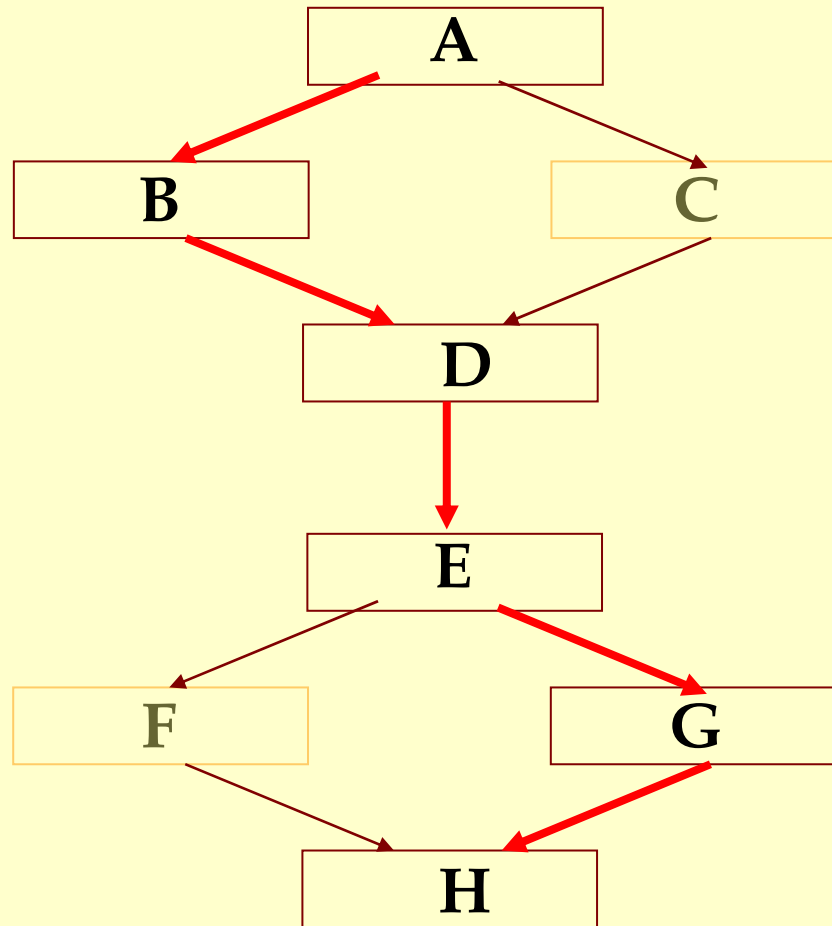
Trace Scheduling

- ◆ Find the most common trace of basic blocks.
 - ◆ Use profile information.
- ◆ Combine the basic blocks in the trace and schedule them as one block.
- ◆ Create compensating (clean-up) code if the execution goes off-trace.

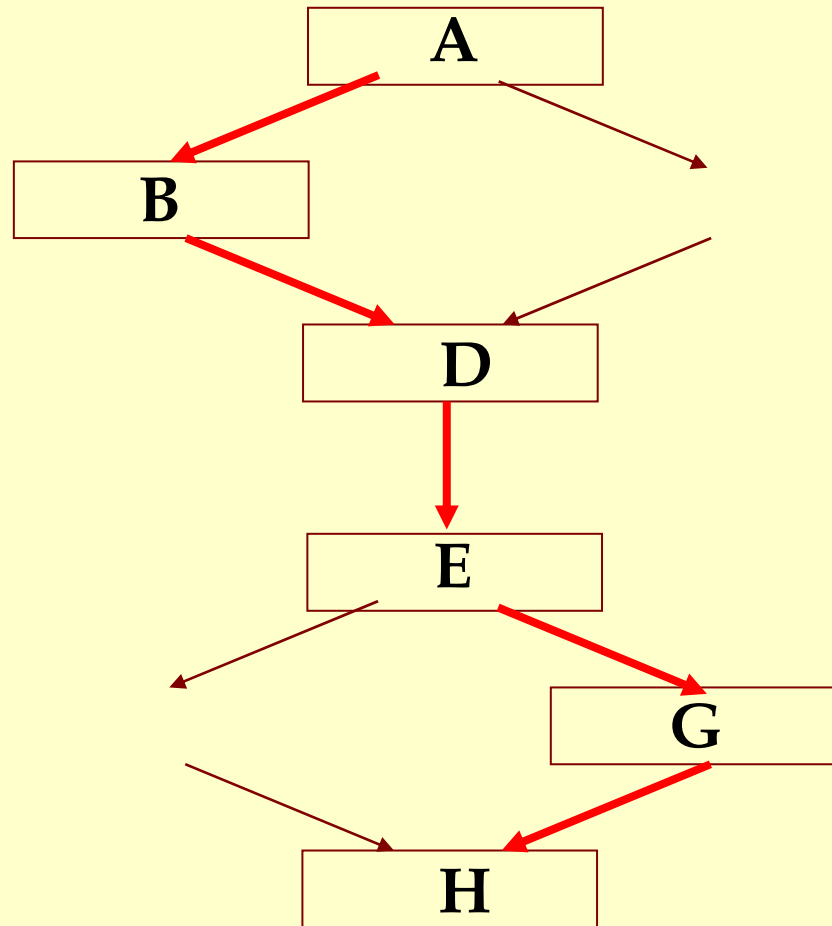
Trace Scheduling



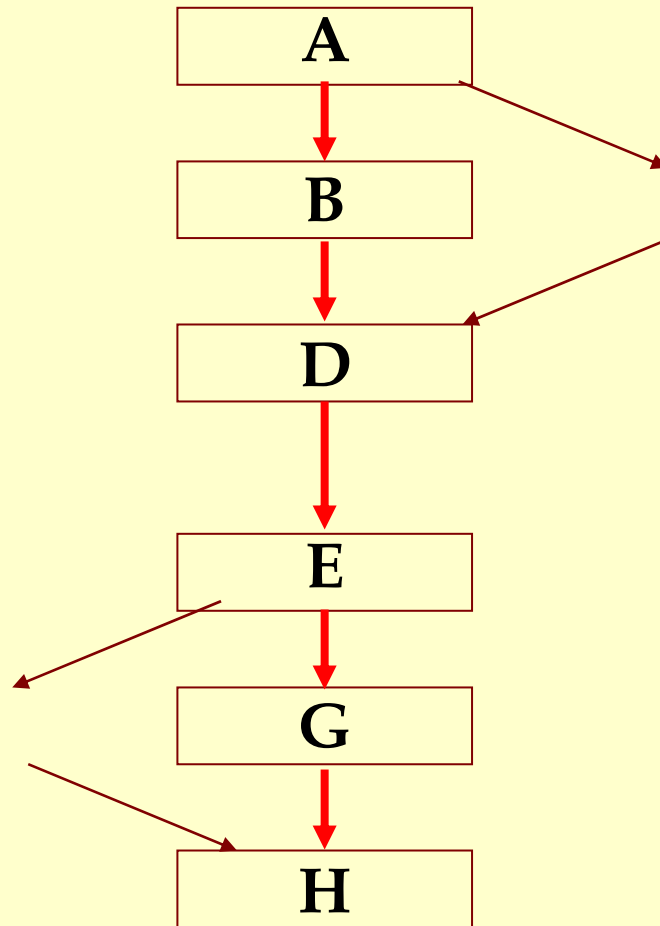
Trace Scheduling



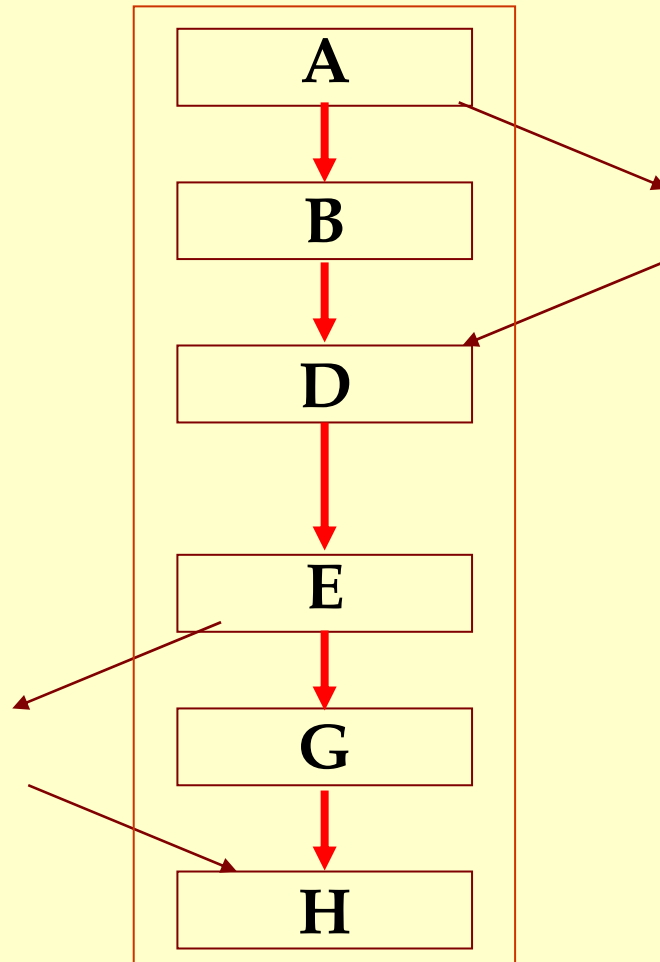
Trace Scheduling



Trace Scheduling

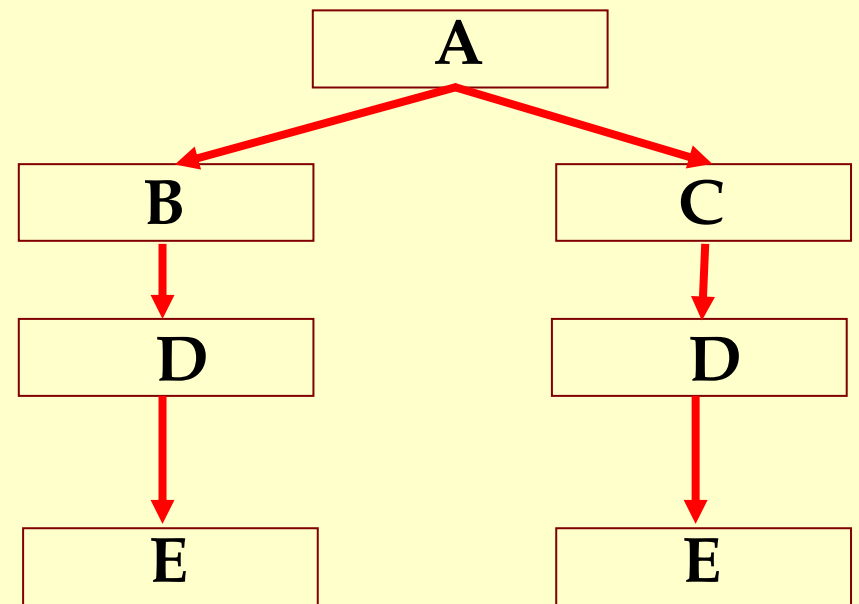
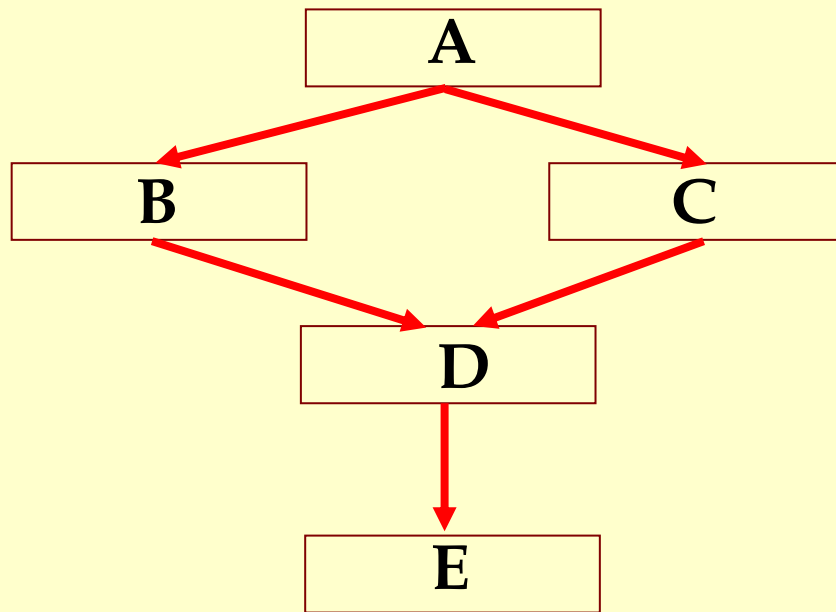


Trace Scheduling



Large Basic Blocks via Code Duplication

- ◆ Creating large extended basic blocks by duplication.
- ◆ Schedule the larger blocks.



Scheduling for Loops

- ◆ Loop bodies are typically small.
- ◆ But a lot of time is spend in loops due to their iterative nature.
- ◆ Need better ways to schedule loops.

Loop Example

Machine:

- ◆ One load/store unit
 - ◆ load 2 cycles
 - ◆ store 2 cycles
- ◆ Two arithmetic units
 - ◆ add 2 cycles
 - ◆ branch 2 cycles (no delay slot)
 - ◆ multiply 3 cycles
- ◆ Both units are pipelined (initiate one op each cycle)

Loop Example

Source Code

```
for i = 1 to N
    A[i] = A[i] * b
```

Assembly Code

```
loop:
    ld  r6, (r2)
    mul r6, r6, r3
    st  r6, (r2)
    add r2, r2, 4
    ble r2, r5, loop
```

Loop Example

Assembly Code

```
loop:  
  ld  r6, (r2)  
  mul r6, r6, r3  
  st  r6, (r2)  
  add r2, r2, 4  
  ble r2, r5, loop
```

Schedule (9 cycles per iteration)

Mem			st				
ALU1	mul				ble		
		mul				ble	
ALU2			add				
				add			

Loop Unrolling

Oldest compiler trick of the trade:

Unroll the loop body a few times

Pros:

- ◆ Creates a much larger basic block for the body.
- ◆ Eliminates few loop bounds checks.

Cons:

- ◆ Much larger program.
- ◆ Setup code (# of iterations < unroll factor).
- ◆ Beginning and end of the schedule can still have unused slots.

Loop Example

```

loop:
  ld  r6, (r2)
  mul r6, r6, r3
  st  r6, (r2)
  add r2, r2, 4
  ble r2, r5, loop
    
```

```

loop:
  ld  r6, (r2)
  mul r6, r6, r3
  st  r6, (r2)
  add r2, r2, 4
  ld  r6, (r2)
  mul r6, r6, r3
  st  r6, (r2)
  add r2, r2, 4
  ble r2, r5, loop
    
```

Schedule (8 cycles per iteration)

Mem	ld				st		ld				st			
		ld				st		ld				st		
ALU1			mul					mul					ble	
				mul					mul					ble
					mul					mul				
ALU2				add							add			
					add							add		
													add	

Loop Unrolling

- ◆ Rename registers.
 - ◆ Use different registers in different iterations.

Loop Example

```
loop:
  ld  r6, (r2)
  mul r6, r6, r3
  st  r6, (r2)
  add r2, r2, 4
  ld  r6, (r2)
  mul r6, r6, r3
  st  r6, (r2)
  add r2, r2, 4
  ble r2, r5, loop
```

```
loop:
  ld  r6, (r2)
  mul r6, r6, r3
  st  r6, (r2)
  add r2, r2, 4
  ld  r7, (r2)
  mul r7, r7, r3
  st  r7, (r2)
  add r2, r2, 4
  ble r2, r5, loop
```

Loop Unrolling

- ◆ Rename registers.
 - ◆ Use different registers in different iterations.
- ◆ Eliminate unnecessary dependencies.
 - ◆ again, use more registers to eliminate true, anti and output dependencies.
 - ◆ eliminate dependent-chains of calculations when possible.

Loop Example

```
loop:
  ld  r6, (r2)
  mul r6, r6, r3
  st  r6, (r2)
  add r2, r2, 4
  ld  r7, (r2)
  mul r7, r7, r3
  st  r7, (r2)
  add r2, r2, 4
  ble r2, r5, loop
```

```
loop:
  ld  r6, (r1)
  mul r6, r6, r3
  st  r6, (r1)
  add r2, r1, 4
  ld  r7, (r2)
  mul r7, r7, r3
  st  r7, (r2)
  add r1, r2, 4
  ble r1, r5, loop
```

Loop Example

```
loop:
  ld  r6, (r1)
  mul r6, r6, r3
  st  r6, (r1)
  add r2, r1, 4
  ld  r7, (r2)
  mul r7, r7, r3
  st  r7, (r2)
  add r1, r2, 4
  ble r1, r5, loop
```

```
loop:
  ld  r6, (r1)
  mul r6, r6, r3
  st  r6, (r1)
  add r2, r1, 4
  ld  r7, (r2)
  mul r7, r7, r3
  st  r7, (r2)
  add r1, r2, 4
  ble r1, r5, loop
```

Loop Example

```
loop:
  ld  r6, (r1)
  mul r6, r6, r3
  st  r6, (r1)
  add r2, r1, 4
  ld  r7, (r2)
  mul r7, r7, r3
  st  r7, (r2)
  add r1, r2, 4
  ble r1, r5, loop
```

```
loop:
  ld  r6, (r1)
  mul r6, r6, r3
  st  r6, (r1)
  add r2, r1, 4
  ld  r7, (r2)
  mul r7, r7, r3
  st  r7, (r2)
  add r1, r1, 8
  ble r1, r5, loop
```

Loop Example

```

loop:
  ld  r6, (r1)
  mul r6, r6, r3
  st  r6, (r1)
  add r2, r1, 4
  ld  r7, (r2)
  mul r7, r7, r3
  st  r7, (r2)
  add r1, r1, 8
  ble r1, r5, loop
    
```

Schedule (4.5 cycles per iteration)

Mem	ld		ld			st		st			
		ld		ld			st		st		
ALU1			mul		mul			ble			
				mul		mul			ble		
					mul		mul				
ALU2	add					add					
		add					add				

Software Pipelining

- ◆ Try to overlap multiple iterations so that the slots will be filled.
- ◆ Find the steady-state window so that:
 - ◆ all the instructions of the loop body is executed.
 - ◆ but from different iterations.

Loop Example

Assembly Code

```

loop:
  ld   r6, (r2)
  mul  r6, r6, r3
  st   r6, (r2)
  add  r2, r2, 4
  ble  r2, r5, loop
  
```

Schedule

ld		ld1		ld2	st	ld3	st1	ld4	st2	ld5	st3	ld6
	ld		ld1		ld2	st	ld3	st1	ld4	st2	ld5	st3
		mul		mul1		mul2	ble	mul3	ble1	mul4	ble2	mul5
			mul		mul1		mul2	ble	mul3	ble1	mul4	ble2
				mul		mul1		mul2		mul3		mul4
					add		add1		add2		add3	
						add		add1		add2		add3

Loop Example

Assembly Code

```
loop:  
  ld    r6, (r2)  
  mul   r6, r6, r3  
  st    r6, (r2)  
  add   r2, r2, 4  
  ble   r2, r5, loop
```

ld3	st1
st	ld3
mul2	ble
	mul2
mul1	
	add1
add	

Schedule (2 cycles per iteration)

Loop Example

4 iterations are overlapped.

- ◆ values of r3 and r5 don't change
- ◆ 4 regs for &A[i] (r2)
- ◆ each addr. incremented by 4*4
- ◆ 4 regs to keep value A[i] (r6)
- ◆ Same registers can be reused after 4 of these blocks generate code for 4 blocks, otherwise need to move .

ld3	st1
st	ld3
mul2	ble
	mul2
mul1	
	add1
add	

```
loop:
    ld    r6, (r2)
    mul   r6, r6, r3
    st    r6, (r2)
    add   r2, r2, 4
    ble   r2, r5, loop
```


Software Pipelining

- ◆ Optimal use of resources.
- ◆ Need a lot of registers.
 - ◆ Values in multiple iterations need to be kept.
- ◆ Issues in dependencies.
 - ◆ Executing a store instruction in an iteration before branch instruction is executed for a previous iteration (writing when it should not have).
 - ◆ Loads and stores are issued out-of-order (need to figure-out dependencies before doing this).
- ◆ Code generation issues.
 - ◆ Generate pre-amble and post-amble code.
 - ◆ Multiple blocks so no register copy is needed.

Optimization Techniques

Advanced Compiler Techniques

2004

Erik Stenman

EPFL

Optimization Techniques Summary

- ◆ The most important aspect of an optimization is that it is correct.
- ◆ The subject is confusing:
 - ◆ The notion of optimality.
 - ◆ Huge number of possible optimization.
 - ◆ Many intricate and NP-complete problems.
- ◆ In this course we have tried to give an overview of some common optimization techniques.

Optimization Techniques Summary

- ◆ Suggested method for compiler optimization:
 1. Look at the generated code – try to find sources of inefficient code. (Better yet profile.)
 2. Look in the literature for solutions to these inefficiencies. (Most likely someone has already solved the problem.)
 3. Implement the solution.
 4. Repeat from 1.

Optimization Techniques Summary

- ◆ Some techniques are useful for many different problems.
 - ◆ Dataflow analysis.
 - ◆ Dominators.
 - ◆ Liveness.
 - ◆ SSA form.
 - ◆ Reverse post order traversal.
 - ◆ Graph coloring.

Optimization Techniques Taxonomy

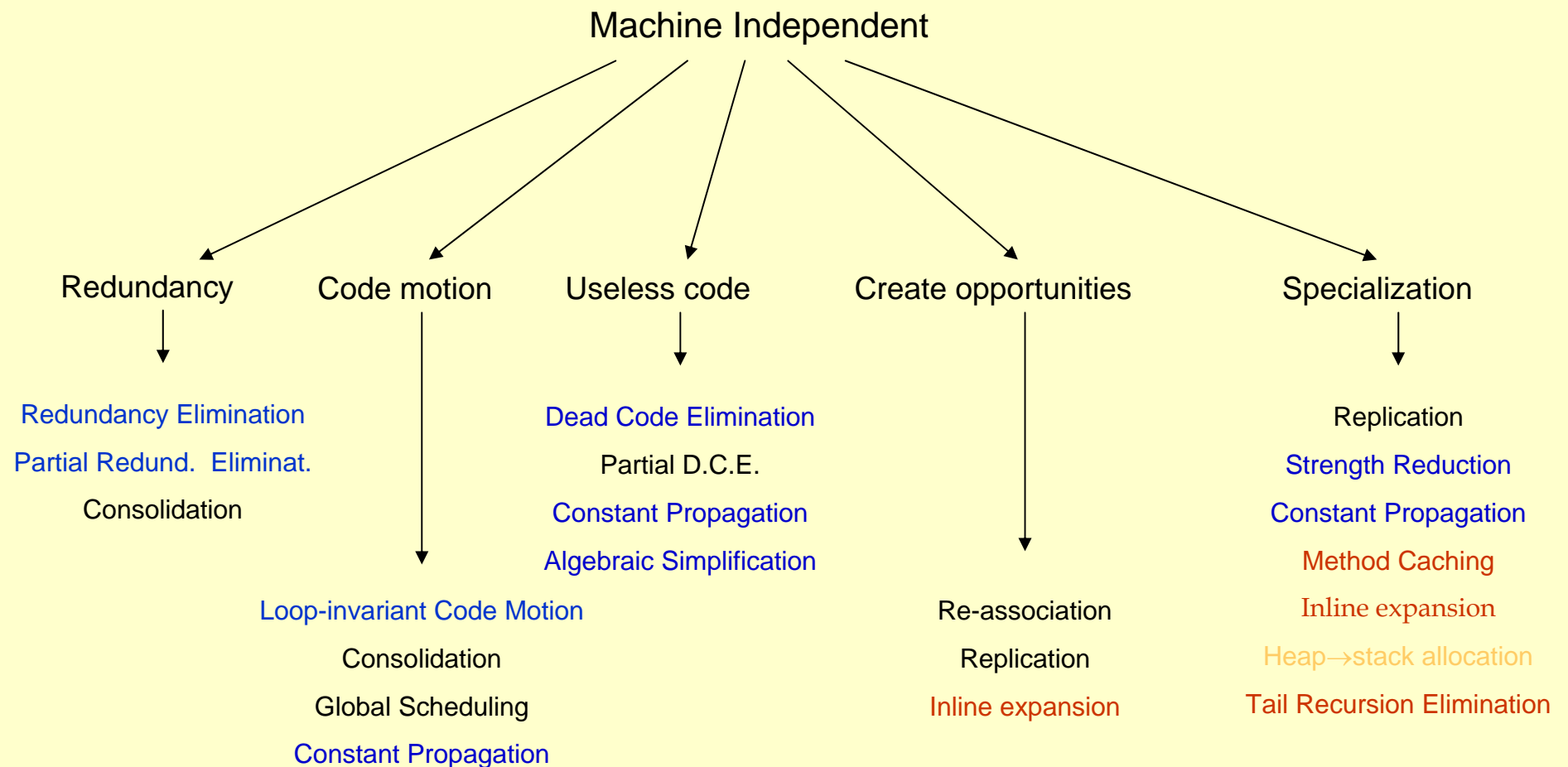
- ◆ We can divide optimizations into:
 - ◆ Machine independent optimizations.
 - ◆ Decrease ratio of overhead to real work.
 - ◆ Example: dead code elimination.
 - ◆ Machine dependent optimizations.
 - ◆ Take advantage of specific machine properties.
 - ◆ Work around limitations of a specific machine.
 - ◆ Example: instruction scheduling.

Optimization Techniques

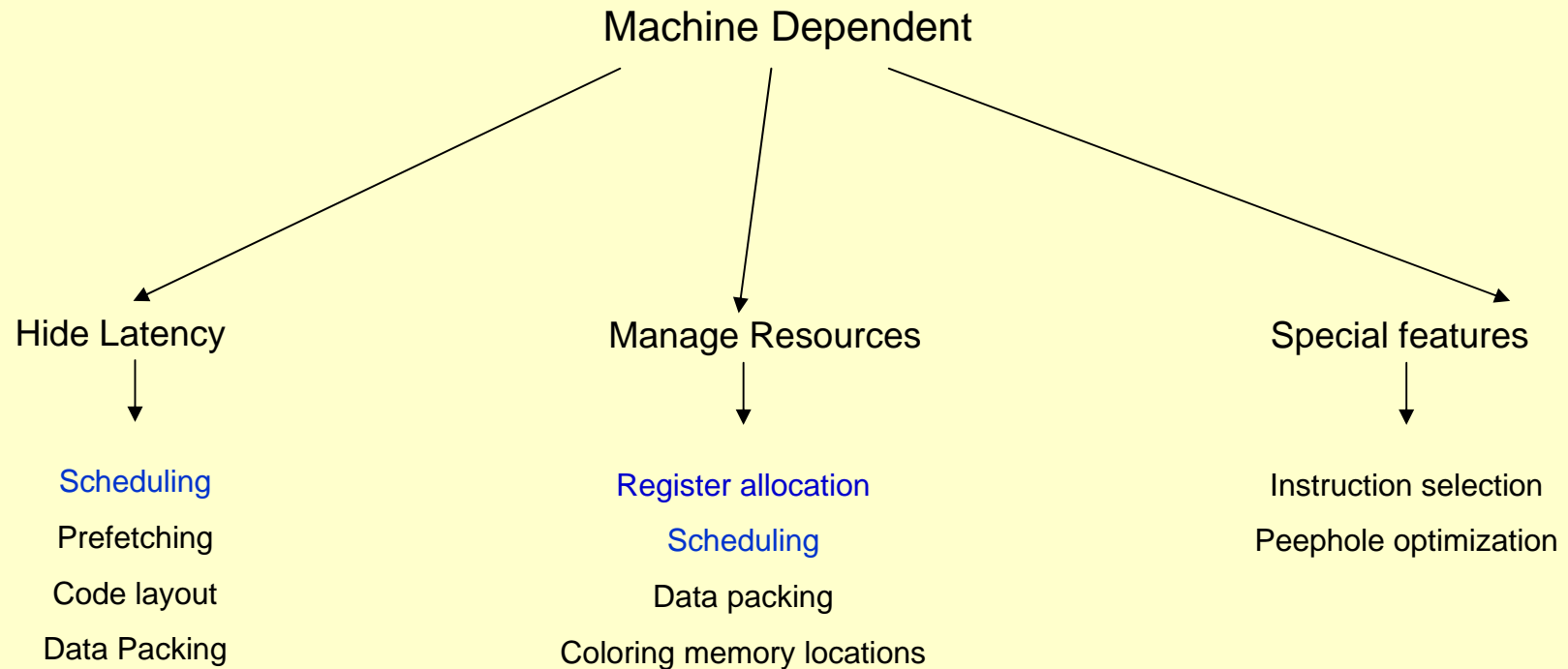
Taxonomy

- ◆ We can further divide the optimizations on their intended effect.
 - ◆ Machine independent optimizations.
 1. Eliminating redundant computations.
 2. Move code to execute it less.
 3. Eliminate dead code.
 4. Specialize on context.
 5. Enable other optimizations.
 - ◆ Machine dependent optimizations.
 1. Manage or hide latency.
 2. Take advantage of special hardware features.
 3. Manage finite resources.

Taxonomy of Global Compiler Optimizations



Taxonomy of Global Compiler Optimizations



Optimization Techniques Summary

- ◆ The aim of the lectures have been to give you an insight into and overview of some of the most important concepts in optimizing compilers.
- ◆ You might also have discovered that the topic is complex and often difficult.
- ◆ The project will probably really show you how difficult it is.
- ◆ Hopefully the project will also show you how fun it can be.

Implementation of High Level Languages

Advanced Compiler Techniques

2004

Erik Stenman

EPFL

Overview

- ◆ In this second part of the course we will talk about how to implement:
 - ◆ Objects and inheritance.
 - ◆ FPLs: higher order functions, laziness.
 - ◆ Concurrency: processes, message passing.
 - ◆ Automatic memory management. (GC)
 - ◆ Virtual Machines. (maybe also interpretation.)
 - ◆ Just in time compilation.

Implementation of High Level Languages

- ◆ We will look at some simple ways to implement concepts in HLL.
- ◆ We will look at some more complex and more efficient implementations of these concepts.
- ◆ We will also look at some general optimization techniques that can be used with great advantage in HLL.

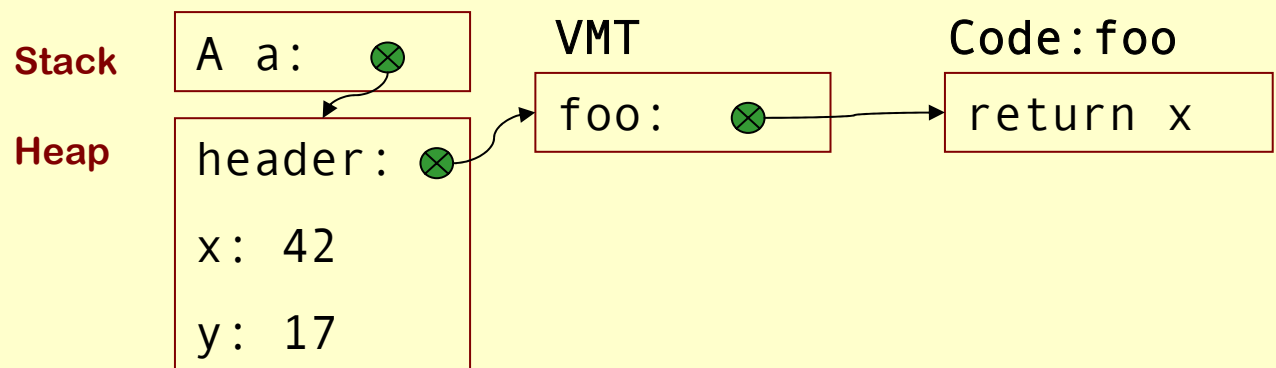
Implementation of Object Oriented Languages

- ◆ In class based OO languages each object belongs to a class that defines the fields, methods, and the type of the object.

```
class A {
  int x=42;
  int y=17;

  int foo() {
    return x;
  }
}
```

```
A a = new A();
a.foo();
```



Implementation of Object Oriented Languages

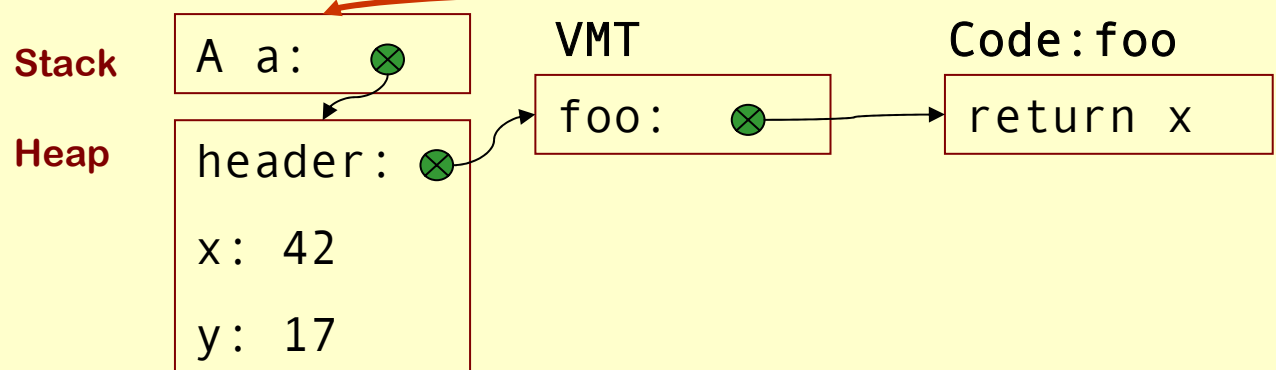
- ◆ In class based OO languages each object belongs to a class that defines the fields, methods, and the type of the object.

```
class A {
  int x=42;
  int y=17;

  int foo() {
    return x;
  }
}
```

```
A a = new A;
a.foo();
```

Reference to object:
many/object.



Implementation of Object Oriented Languages

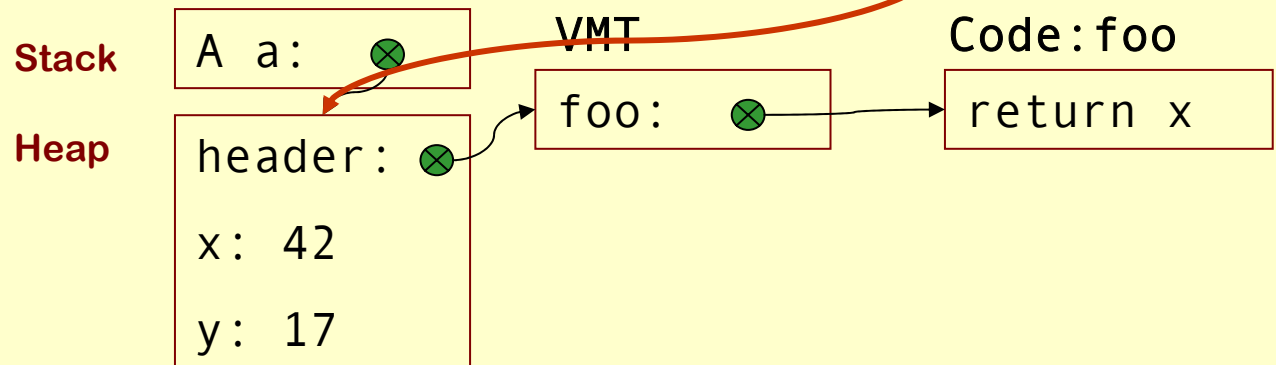
- ◆ In class based OO languages each object belongs to a class that defines the fields, methods, and the type of the object.

```
class A {
  int x=42;
  int y=17;

  int foo() {
    return x;
  }
}
```

```
A a = new A;
a.foo();
```

Representation of object:
1/object.



Implementation of Object Oriented Languages

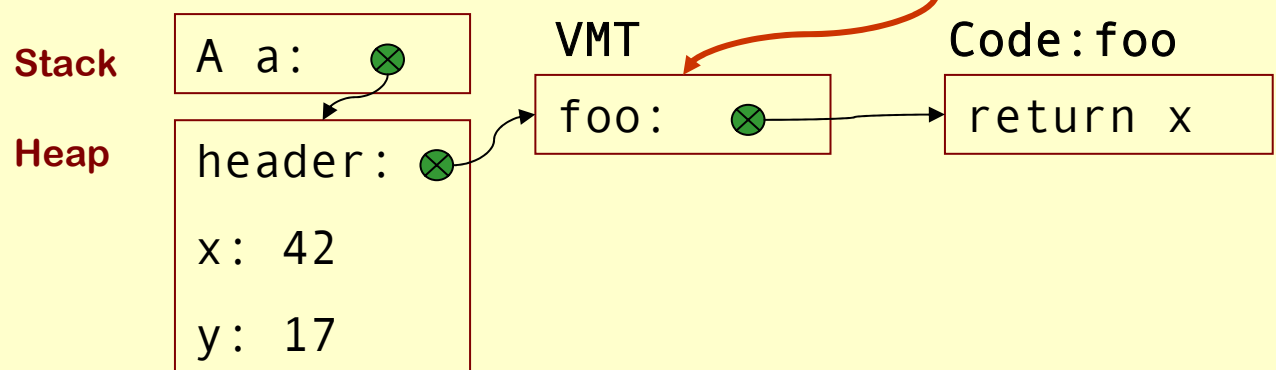
- ◆ In class based OO languages each object belongs to a class that defines the fields, methods, and the type of the object.

```
class A {
  int x=42;
  int y=17;

  int foo() {
    return x;
  }
}
```

```
A a = new A;
a.foo();
```

Virtual Method Table:
1/class.



Implementation of Object Oriented Languages

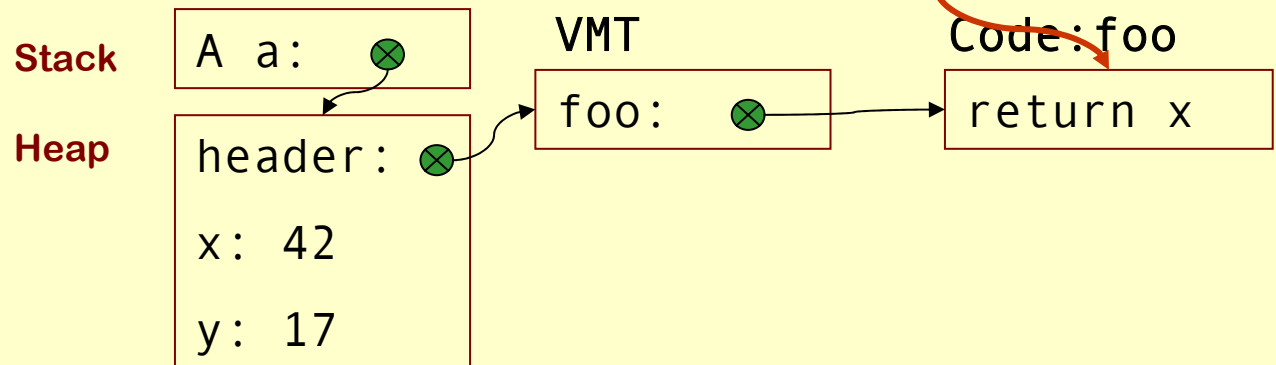
- ◆ In class based OO languages each object belongs to a class that defines the fields, methods, and the type of the object.

```
class A {
  int x=42;
  int y=17;

  int foo() {
    return x;
  }
}
```

```
A a = new A();
a.foo();
```

Code for functions (foo):
max 1/class.



Implementation of Object Oriented Languages

- ◆ Object Oriented languages support inheritance.
- ◆ Inheritance complicates the answer to some questions:
 - ◆ Where is the value of a field stored?
 - ◆ Where is the code for a certain method?
 - ◆ What type will a value have at runtime?

Single Inheritance: Fields

- ◆ With single inheritance we can order the fields in such a way that all fields of a class are stored after fields of the superclass.
- ◆ This way we know at compile time the offset of each field.

Single Inheritance: Fields

◆ Example:

```
class A          { int x = 0; }  
class B extends A { int y = 0;  
                  int z = 0; }  
class C extends A { int r = 0; }  
class D extends A { int s = 0; }
```

Single Inheritance: Fields

```

class A      {int x = 0;}
class B extends A {int y = 0;
                  int z = 0;}
class C extends A {int r = 0;}
class D extends B {int s = 0;}
  
```

Offsets:

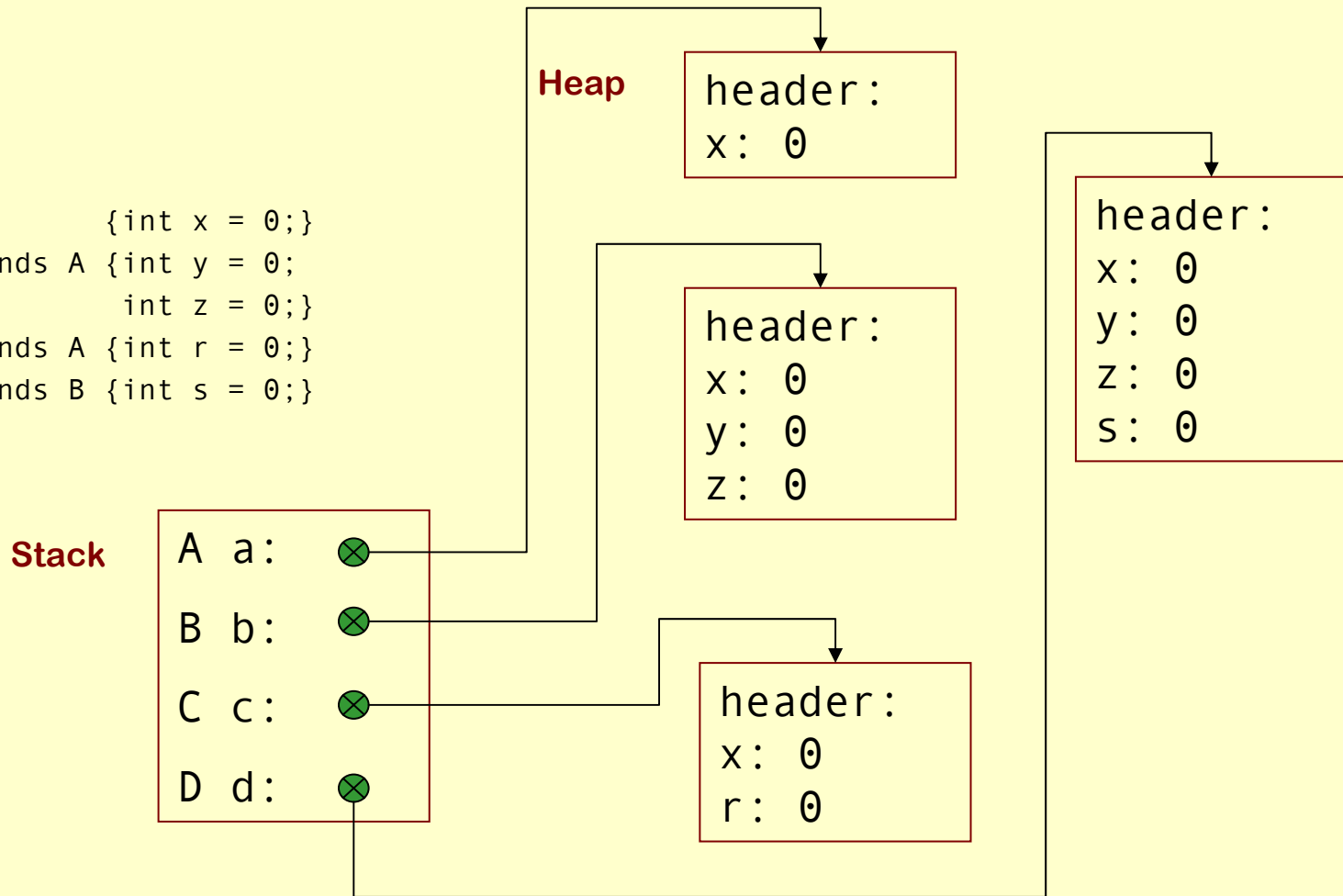
(A,B,C,D).x: 1

(B,D).y: 2

(B,D).z: 3

(C).r: 2

(D).s: 4



Single Inheritance: Methods

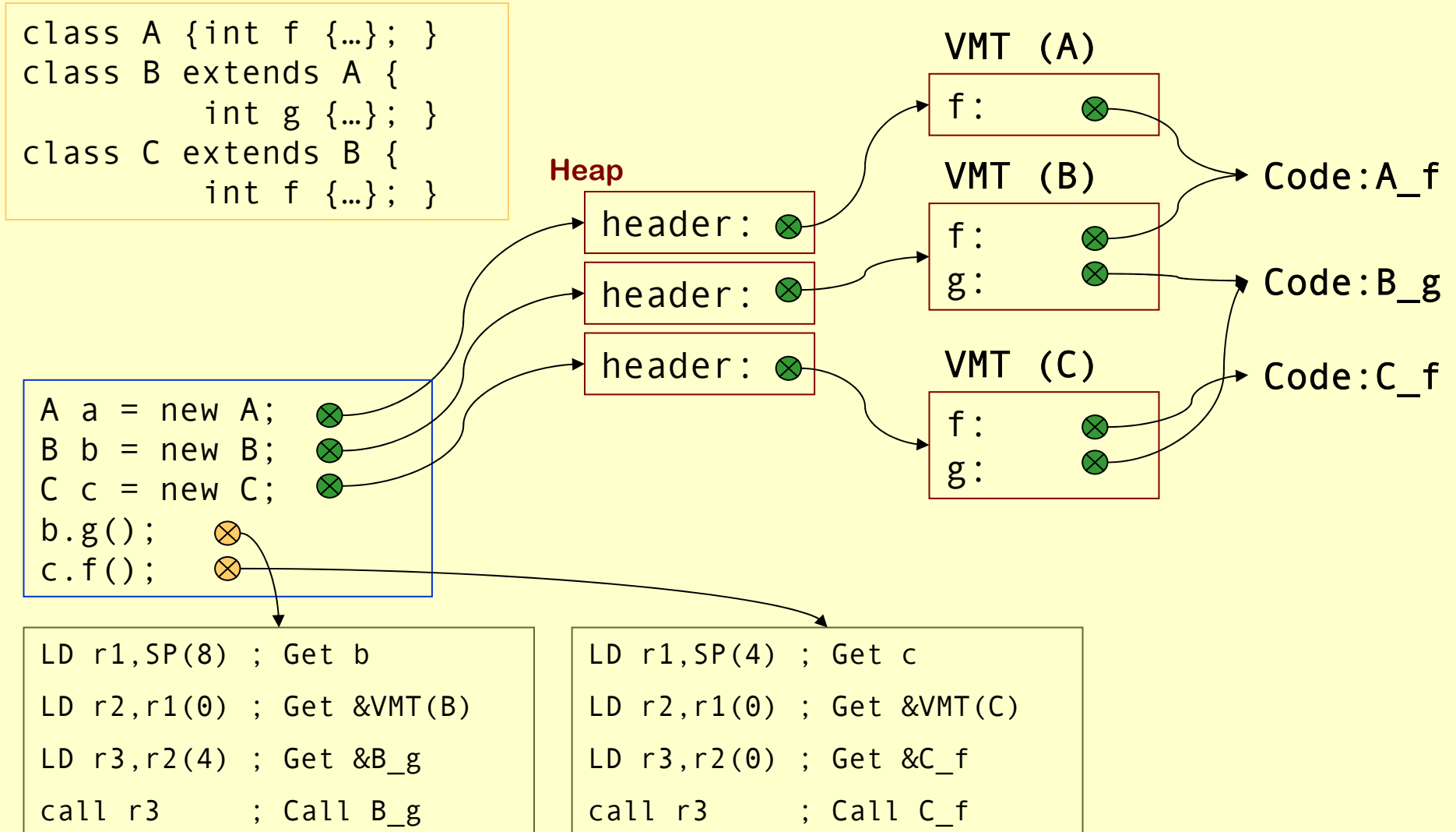
- ◆ If we only have single inheritance we can handle methods in much the same way as fields.
- ◆ We store addresses to methods in the VMT instead of in the object.
- ◆ We copy all the addresses of the super classes to the VMT of the subclasses.
- ◆ If a method is overridden we use the address of the new definition instead of the definition in the superclass.

Single Inheritance: Methods

◆ Example:

```
class A          { int f {...}; }  
class B extends A { int g {...}; }  
class C extends B { int f {...}; }
```


Single Inheritance: Methods



Single Inheritance: Testing Class Membership

- ◆ Many OO languages allow you to test class membership of an object.
- ◆ In Java there is “`o instanceof C`”.
- ◆ An object is a member of all its superclasses.
- ◆ We need to be able to find the superclass of a class. Let us extend our implementation with class descriptors.

Single Inheritance: Class Membership

```
class A {int f {...}; }
class B extends A {
    int g {...}; }
class C extends B {
    int f {...}; }
```

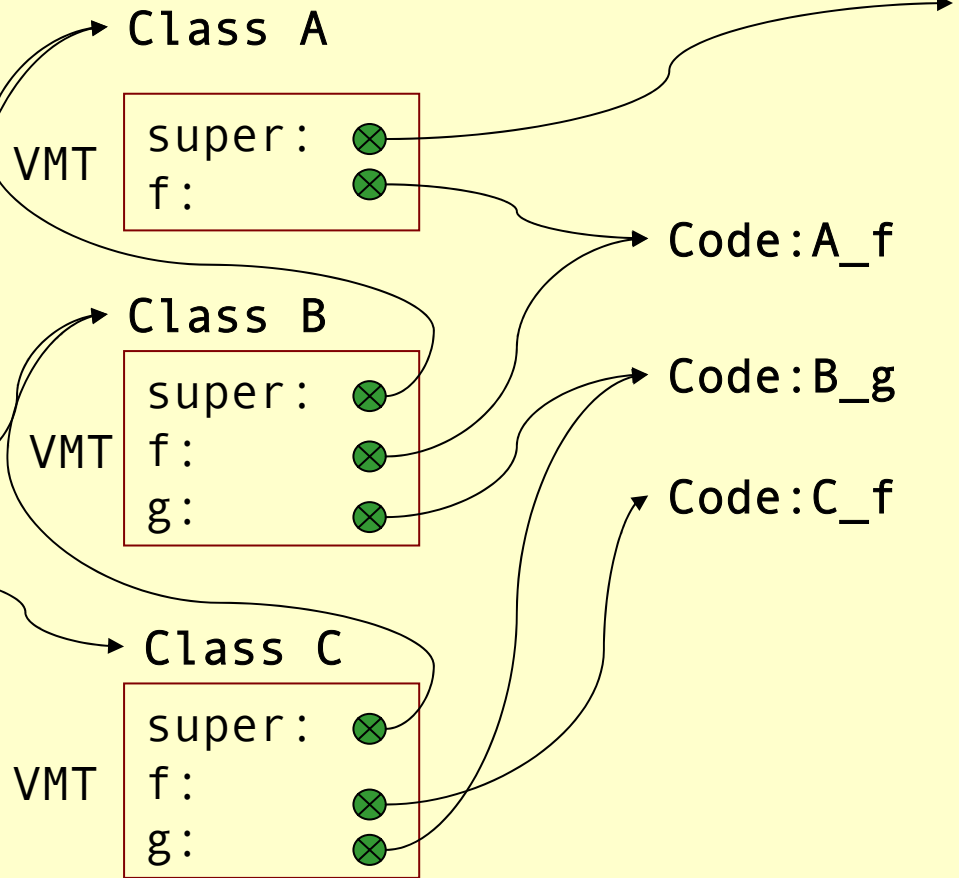
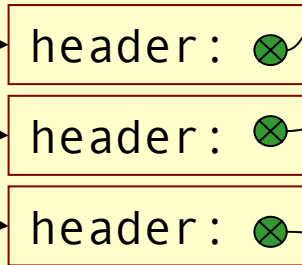
```
A a = new A;
B b = new B;
C c = new C;
c instance of A;
```

Now we can do
c instance of A as:

```

    t = c.header
L:   if t == A goto True
    t = t.super
    if t != nil goto L
    res = false
    goto End
True: res = true
End:
```

Heap



Single Inheritance: Testing Class Membership

- ◆ Searching through the class hierarchy is inefficient.
- ◆ We can trade space for speed.
- ◆ Let each class descriptor have a *display* of all superclasses. I.E., a direct link to each superclass.

Single Inheritance: Class Membership

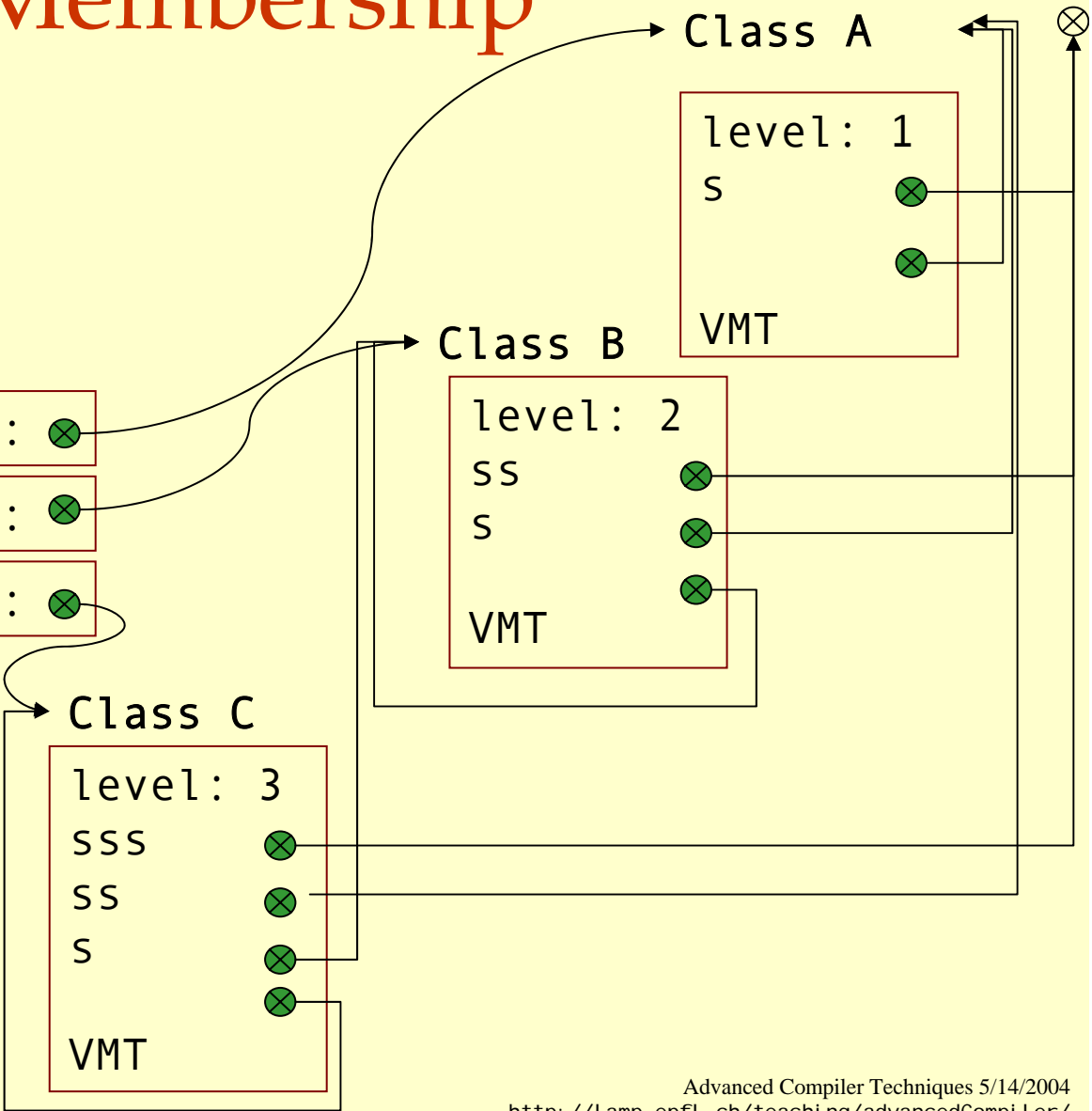
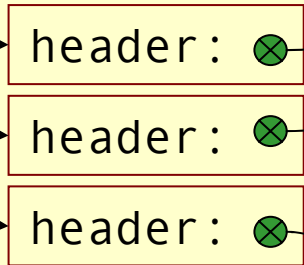
```
class A {}
class B extends A { }
class C extends B { }
```

```
A a = new A;
B b = new B;
C c = new C;
c instance of A;
```

Now we can do
c instance of A as:

```
t1 = c.header
res = t1[0] >= 1 \ \ A_level
if !res goto End
t2 = t1[2] \ \ 2<-A_level+1
res = (t2 == A)
End:
```

Heap



Multiple Inheritance

- ◆ In languages with multiple inheritance, i.e., where it is possible to extend several parent classes with a class, all the operations we have seen become more difficult.
- ◆ Java's hybrid approach with interfaces complicates these issues in the same way as multiple inheritance.

Multiple Inheritance: Graph Coloring

- ◆ One way to handle the layout of fields would be to use graph coloring. (This can also be used for methods.)
- ◆ All identical fields would have to occupy the same offset in the object.
- ◆ For some objects there would be holes in the array of fields. To reduce the wasted space the fields can be compacted in the object by storing the offsets in the class descriptor.

Multiple Inheritance: Graph Coloring

```

class A      {int x = 0;}
class B      {int y = 0;
              int z = 0;}
class C extends A,B {int r = 0;}
  
```

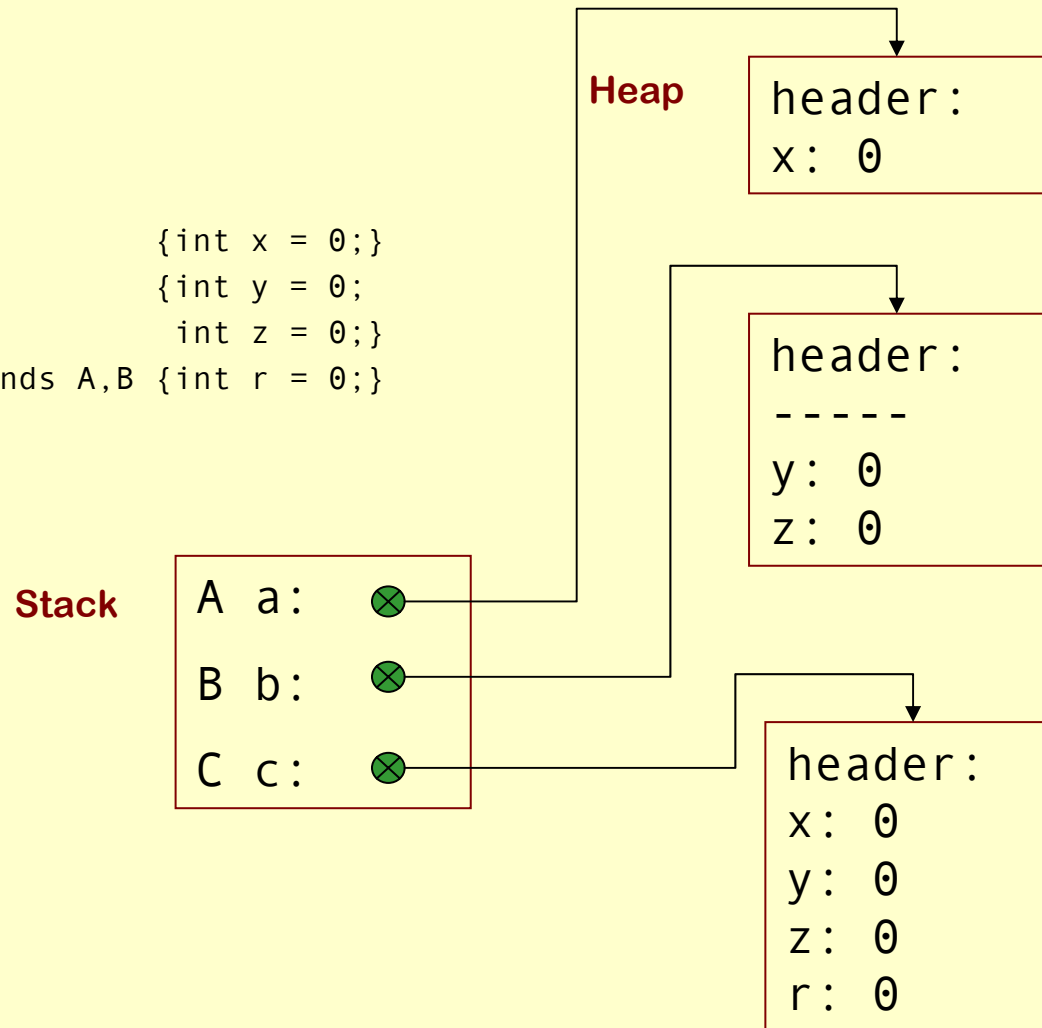
Offsets:

(A,C).x: 1

(B,C).y: 2

(B,C).z: 3

(C).r: 4



Multiple Inheritance: Graph Coloring

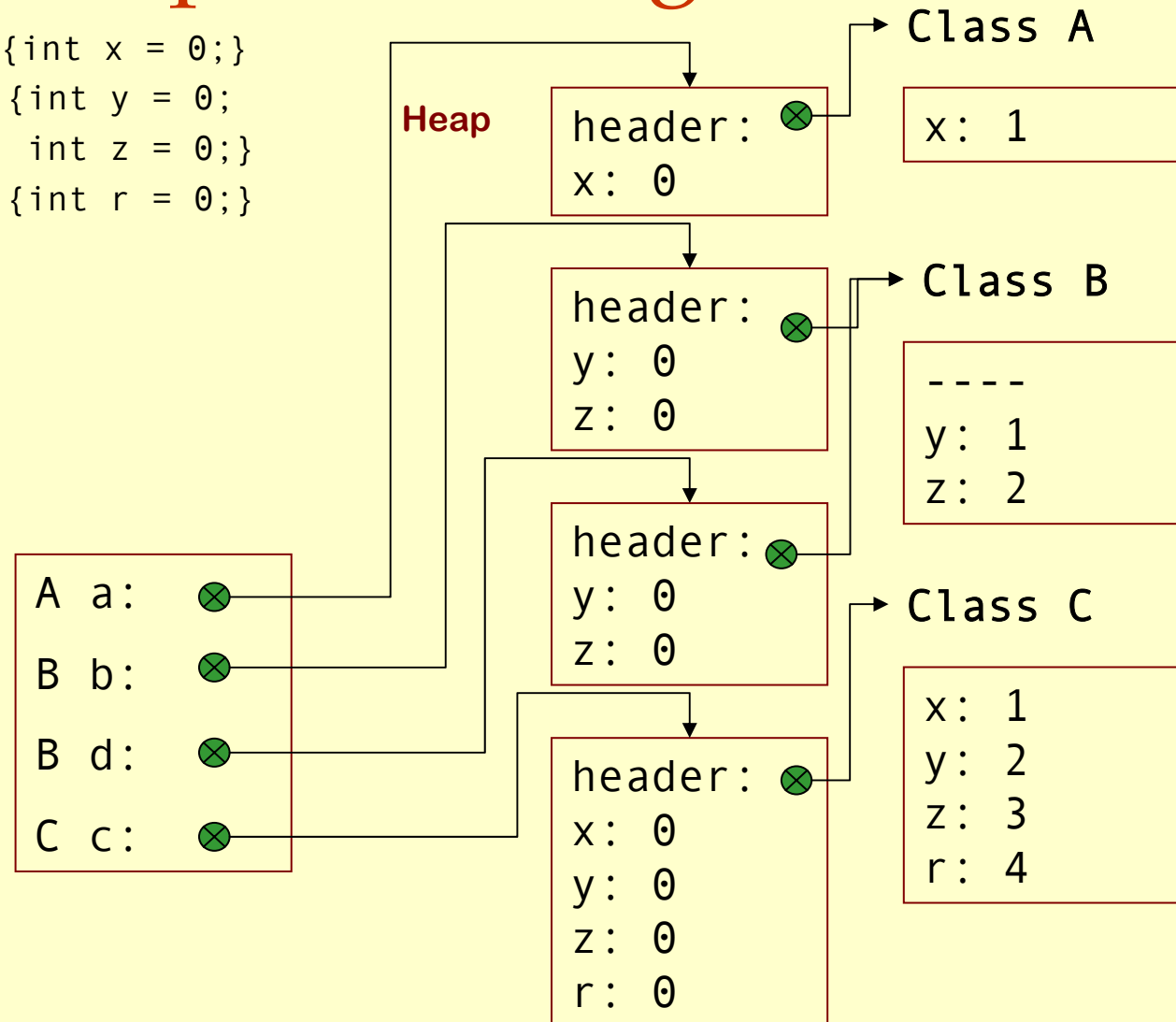
```

class A          {int x = 0;}
class B          {int y = 0;
                  int z = 0;}
class C extends A,B {int r = 0;}
A a = new A;
B b = new B;
B d = new B;
C c = new C;
  
```

Offsets:

Stack

- (A,C).x: header[0]
- (B,C).y: header[1]
- (B,C).z: header[2]
- (C).r: header[3]



Multiple Inheritance: Graph Coloring

- ◆ One problem with global graph coloring is that it is global: you need the whole program – must be done at link time.
- ◆ If dynamic linking is possible this approach becomes even harder.

Multiple Inheritance: Hashing

- ◆ Second approach: Hashing.
- ◆ Instead of a global compile- or link time solution we can calculate a hash value for each name at compile time.
- ◆ At runtime we use the hash value as an offset into a hash table in the class descriptor.
- ◆ This hash table contains the offset to fields in the object. (This also works for method addresses.)
- ◆ This can be costly if there are many collisions in the hash table.

Multiple Inheritance: Trampolines

- ◆ Third approach: Trampoline functions.
- ◆ We give each object several headers, one for each extended class.
- ◆ We add trampoline functions that changes the view of the object from one class to another in an efficient way.


Multiple Inheritance: Trampolines

```

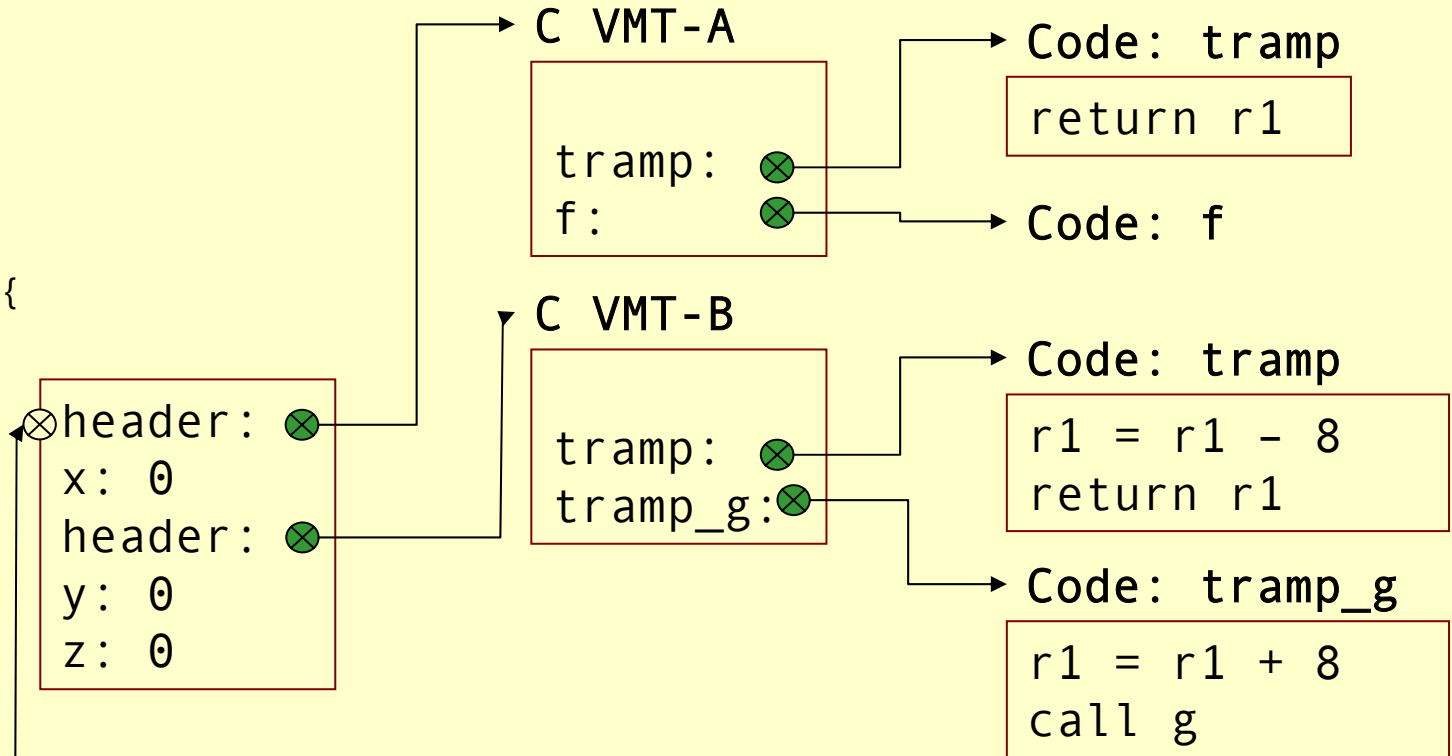
class A {
  int x = 0;
  int f() {...}
}
class B {
  int y = 0;
  int g() {... y ...}
}
class C extends A,B {
  int z = 0;
}
C c1 = new C();
A a = (A) c1;
C c2 = (C) a;
B b = (B) c2;
C c3 = (C) b;

```

```

c1 = 
a = c1;
c2 = a.tramp(); /* = a */
b = c2+8;
c3 = b.tramp(); /* = b-8 */

```

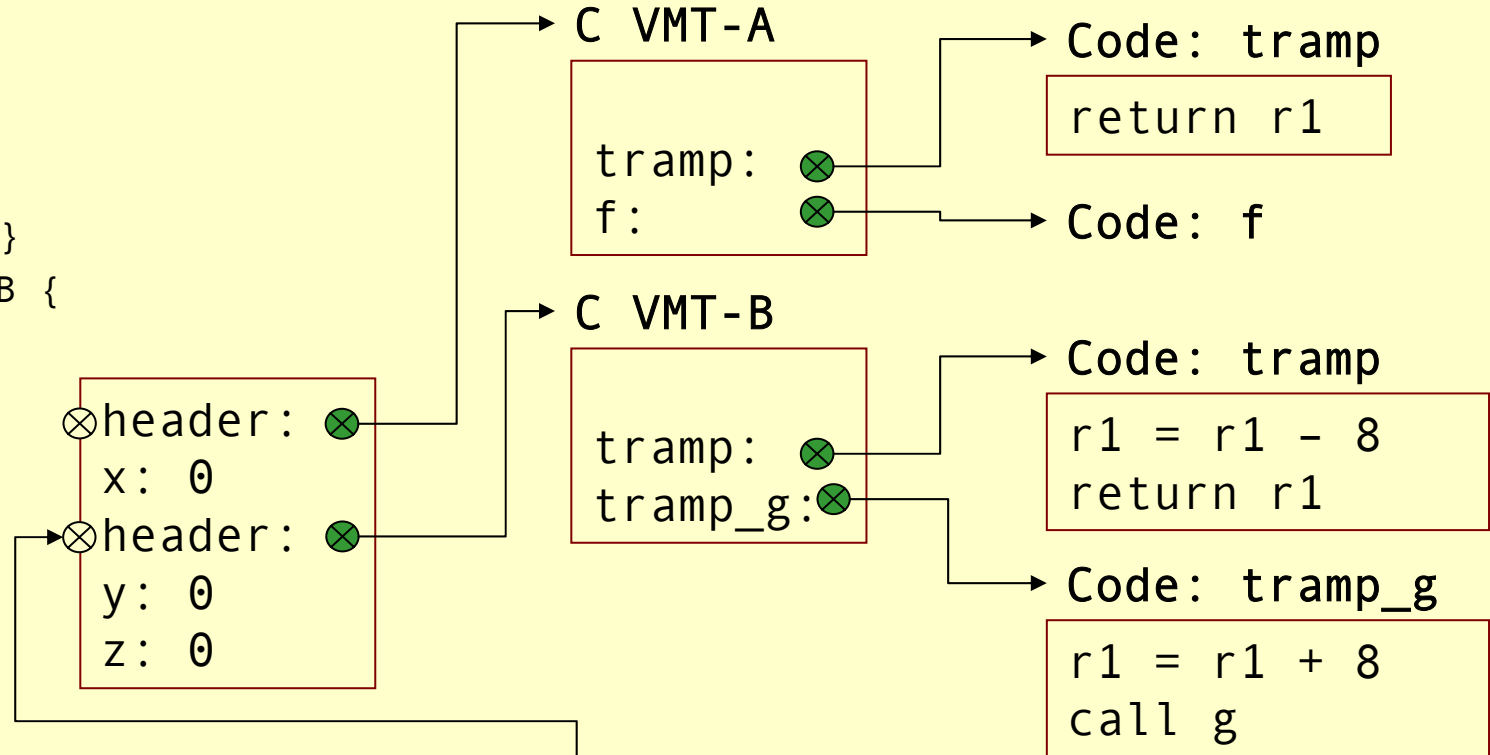


Code: g

Multiple Inheritance: Trampolines

```

class A {
    int x = 0;
    int f() {...}
}
class B {
    int y = 0;
    int g() {... y ...}
}
class C extends A,B {
    int z = 0;
}
C c1 = new C();
A a = (A) c1;
C c2 = (C) a;
B b = (B) c2;
C c3 = (C) b;
    
```



```

c1 = ●
a = c1;
c2 = a.tramp(); /* = a */
b = c2+8; ●
c3 = b.tramp(); /* = b-8 */
    
```

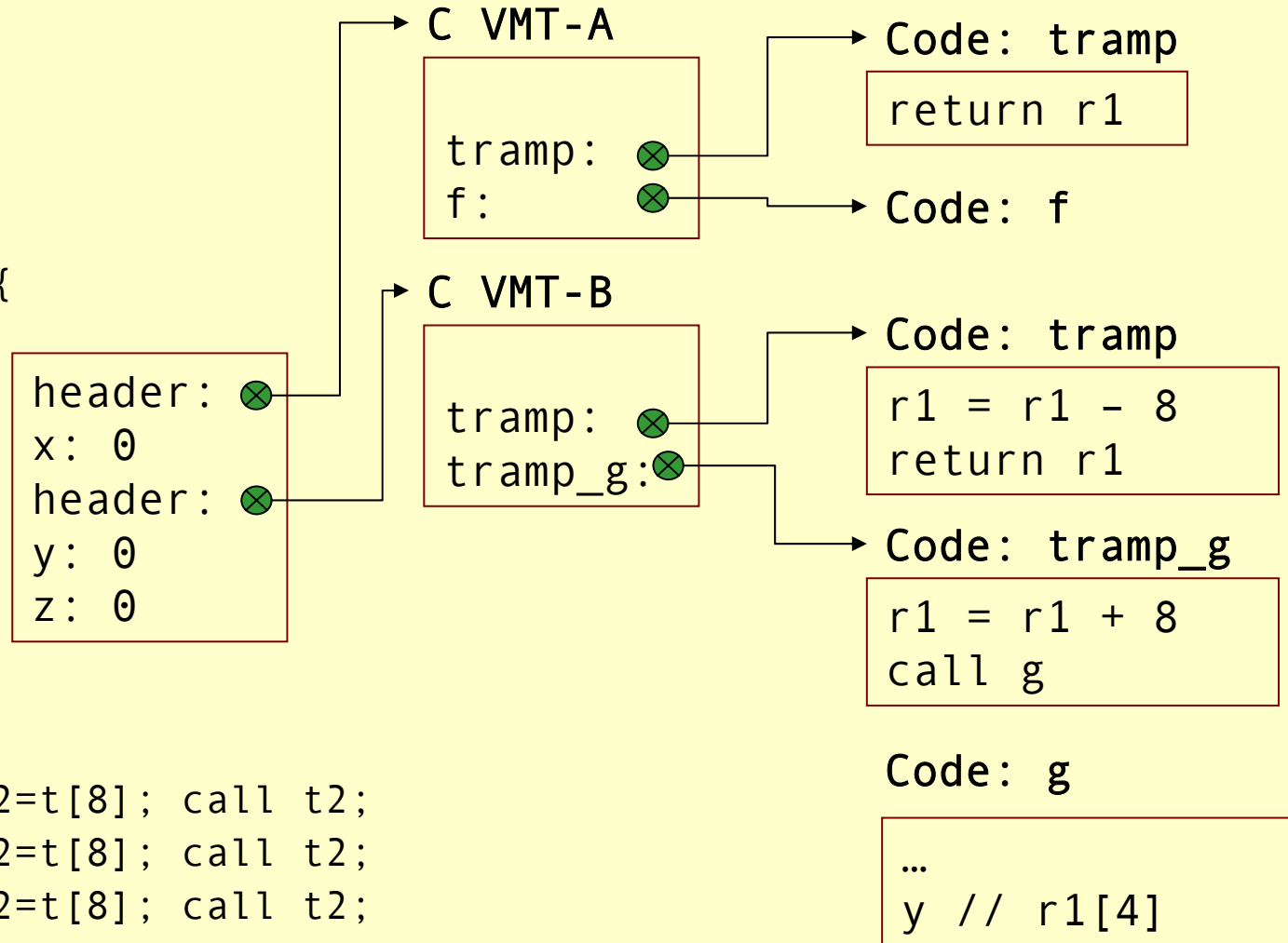
Code: g

Multiple Inheritance: Trampolines

```

class A {
  int x = 0;
  int f() {...}
}
class B {
  int y = 0;
  int g() {... y ...}
}
class C extends A,B {
  int z = 0;
}
C c1 = new C();
A a = (A) c1;
C c2 = (C) a;
B b = (B) c2;
C c3 = (C) b;
c1.z; // c1[16]
c1.x; // c1[4]
c1.z; // c1[12]
c1.g(); // t=c[8]; t2=t[8]; call t2;
a.f(); // t=a[0]; t2=t[8]; call t2;
b.g(); // t=b[0]; t2=t[8]; call t2;

```



Optimizing OO-Programs

- ◆ In modern machines a jump to a known address is much faster than a jump to an address fetched from a table.
- ◆ Dynamic dispatch also makes inlining and interprocedural analysis harder.
- ◆ Possible solutions: Whole program optimization, link time optimization, JIT compilation, or runtime optimizations.
- ◆ When we have the whole program we can turn many dynamic properties into static properties.

Inline caching

- ◆ Many dynamic calls actually go to the same class all the time.
- ◆ For each call site remember the actual target of the last call.
- ◆ Next time jump directly to this location, and check if we end up in the right place.

Polymorphic Inline Caching

- ◆ If a call site is polymorphic inline caching can lead to degraded performance.
- ◆ Solution: Polymorphic inline caching, remember more than one target address.

Polymorphic Inline Caching

- ◆ Polymorphic inline caching can be implemented with an if then else search tree:

v.f()



```
if c.header < C {  
    if c.header < B A.f() else B.f()  
} else {  
    if c.header < D C.f() else D.f()  
}
```

OO: Summary

- ◆ Implementing OO efficiently means implementing inheritance efficiently.
- ◆ There are several possible solutions available and there is still research going on in this area.
- ◆ One of the most successful techniques for optimizing OO is to do it at runtime using JIT compilation – something we will look closer at later in the course.

Implementation of Functional Programming Languages

- ◆ There is no common agreement on exactly what a functional programming language is. But usually such a language should have at least one of the following concepts:
 - ◆ No statements – only functions (or expressions).
 - ◆ Higher order functions.
 - ◆ Pureness (no side effects).
 - ◆ Laziness.
 - ◆ Automatic memory management (Garbage collection.)

Higher Order Functions

- ◆ In Misc (and in C) you have “second”-order functions.
 - ◆ That is, functions are also values in the language: you can take their addresses and pass them around and apply them.

```
def apply(f: (Int) => Int, x: Int): Int = f(x);
```
 - ◆ These functions can be represented with just a function pointer, i.e., the address of the function.
- ◆ Functions that take functions as arguments are called *higher order functions*.
- ◆ For a language to have interesting higher order functions you need to be able to create new functions at runtime. E.g., in Scala you can write:

```
val f:(Int => Int) = x => x + 1;
```

Higher Order Functions

- ◆ To get **really** interesting functions at runtime you need to be able to capture the *free variables* of the function.
 - ◆ A free variable is a variable that is not bound by the definition of the function. (y is free in $x \Rightarrow x+y$.)

```
def f(y: Int): (Int => Int) = x => x+y;
```
- ◆ In order to do this we need *closures*.
- ◆ A closure is a data structure that contains a function pointer and a way to access all free variables of the body of the function.

Higher Order Functions

- ◆ In an OO language a closure can be implemented as an object with a single method and several instance variables.

```
def f(y:Int):(Int => Int) = x=>x+y;  
f(42)(17)
```



```
class F {  
    int y;  
    public F(int y) { this.y = y; }  
    public int apply(int x) {  
        return x+y;  
    }  
}
```

```
public F f(int y) = new F(y);  
f(42).apply(17);
```


Higher Order Functions

- ◆ This is more or less the way Scala implements functions.
- ◆ To make it more general we can make all closures implement the `Function` interface:

```
public interface Function1 {  
    public abstract java.lang.Object apply(java.lang.Object a0);  
}
```

- ◆ We also need to take care of local (mutable) variables that are captured by the function. This can be done by turning them into references.

Higher Order Functions

```
def f(y:Int):(Int => Int) = {
  var z = y*2;
  val f = x=>x+z;
  z = z +1;
  f;
}
```



```
class F {
  IntRef y;
  public F(IntRef y) {
    this.y = y; }
  public int apply(int x) {
    return x+y.v;
  }
}
```

```
class IntRef {
  int v;
  public IntRef(int i) {v=i;}
  public set(int i) {v=i;}
}
```

```
public F f(int y) = {
  IntRef z = new IntRef(y*2);
  F f = new F(z);
  z.set(z.v + 1);
  return f;
}
```

Pure Functional Languages

- ◆ In a *pure functional language* there are no *side effects*.
- ◆ This includes no updates of variables. That is, variables are immutable.
 - ◆ Variables are, like variables in mathematics, just names for values.
 - ◆ If we say $x = 42$; then we give the value 42 a new name: x , from now on x and 42 are interchangeable.
- ◆ With a pure functional language it is possible to do *equational reasoning*.

Lazy Evaluation

- ◆ With *lazy evaluation*, an expression is not evaluated unless its value is demanded by some other part of the computation.
- ◆ In contrast, strict languages (Java, ML, C, Erlang) evaluate each expression as the control flow reaches it.

Call-by-Name Evaluation

- ◆ Most languages pass function arguments using call-by-value:
 - ◆ i.e. all arguments are evaluated before a function is called.
 - ◆ e.g. in the expression $f(g(x+y))$, first $(x+y)$ is evaluated then the function g is called before the function f is called.
 - ◆ If the function f doesn't use its argument then the evaluation of g and of $x+y$ is done in vane.

Call-by-Name Evaluation

- ◆ *Call-by-name* evaluation avoids this problem by not evaluating the arguments, instead a *thunk* is created for each argument.
- ◆ A *thunk* is a function that can be called to compute the value on demand.
 $f(g(x+y))$ is translated to
 $f(() \Rightarrow g(() \Rightarrow x+y))$
- ◆ Any use of the argument in f is replaced by an application of the function:
 $f(x) = x$; is translated to
 $f(x) = x()$;

Call-by-Name Evaluation

- ◆ Scala provides call-by-name with explicit `def` parameters.
- ◆ A problem with call-by-name is that a thunk may be executed many times.

`f(x) = x+x;` is translated to

`f(x) = x()+x();`

Call-by-Need

- ◆ With *call-by-need* each thunk is only evaluated once.
- ◆ This is implemented by giving each thunk a *memo slot* that stores the evaluated value; each evaluation of the thunk first checks the memo slot: if it is empty the expression is evaluated and stored in the slot, otherwise the value in the slot is returned.

Call-by-Need

Conceptually a thunk for $x+y$ can be implemented as:

```
class Thunk {  
    res = null;  
    apply() = {  
        if res == null then res = x+y  
        else res  
    }  
}
```

Call-by-need

- ◆ A thunk can also be implemented just as two words `<thunk_function, memo_slot>`
- ◆ When the thunk is evaluated both fields are updated: the memo slot with the value and the function with a new function that returns the value.

Optimization of FP

- ◆ Functional programs tend to use many small functions. Modern hardware is optimized for imperative programs with few large functions, i.e., function calls are relatively expensive.
- ◆ Hence it can be profitable to reduce the number of function calls and increase the size of functions.
- ◆ This can be done by *inline expansion*.

Inline Expansion

- ◆ Inline expansion or *inlining* is an optimization where a function call is replaced by the body of the function.
- ◆ If this is done in a stage in the compiler where all independent names are replaced by unique symbols then the process is quite straightforward. Otherwise the formal parameters need to be renamed (α -converted).

Inline Expansion

- ◆ If inline expansion is applied indiscriminately, the size of the program explodes.
- ◆ To limit the code growth we can:
 1. Expand only frequent call sites.
 2. Expand only small functions.
 3. Expand functions called only once, and perform *dead function elimination*.

Inline Expansion

- ◆ If we inline a recursive function just as any other function we would probably end up with a call to the original function. Either directly after the first iteration or after a while.

Inline Expansion

```
def loop(int x, int max, int y) =  
  if (x > max) y else loop(x+1, y*y);  
def f(int z) = loop(1,10,z);  
---  
def f(int z) = {  
  val x=1; val max=10; val y=z;  
  if (x>max) y  
  else loop(x+1,max,y*y);  
}
```

Inline Expansion

- ◆ To remedy this we can bring the definition of the recursion with us in the inlining by splitting the function into a *prelude* and a *loop header*.

Inline Expansion

```
def loop(int x, int max, int y) =  
  if (x > max) y else loop(x+1, y*y);  
def f(int z) = loop(1,10,z);
```

```
def f(int z) = {  
  val x=1; val max=10; val y=z;  
  val loop= (int xX, int maxX, int yX) =>  
    if (xX > maxX) yX else loop(xX+1,maxX, yX*yX);  
  if (x>max) y else loop(x+1,max,y*y);  
}
```

Loop-Invariant Hoisting

- ◆ We can avoid passing around values that are the same in each recursive call by using *loop-invariant hoisting*.
- ◆ Just let the constant value become a free variable.
- ◆ In our example lift `max` from an argument to a free variable.

Loop-Invariant Hoisting

```
def loop(int x, int max, int y) =  
  if (x > max) y else loop(x+1, y*y);  
def f(int z) = loop(1,10,z);
```

```
def f(int z) = {  
  val x=1; val max=10; val y=z;  
  val loop= (int xX, int yX) =>  
    if (xX > max) yX else loop(xX+1, yX*yX);  
  if (x>max) y else loop(x+1,y*y);  
}
```

Inline Expansion

- ◆ Inline expansion in itself can be useful since the overhead for a function call and return is removed, but the real benefit comes from applying standard optimizations on the inline expanded program.
- ◆ Constant propagation and folding, dead code and unreachable code elimination all work better when the scope of a function is increased.

Inline Expansion after constant prop

```
def loop(int x, int max, int y) =  
  if (x > max) y else loop(x+1, y*y);  
def f(int z) = loop(1,10,z);
```

```
def f(int z) = {  
  val x=1; val max=10; val y=z;  
  val loop= (int xX, int yX) =>  
    if (xX > 10) yX else loop(xX+1, yX*yX);  
  if (1>10) z else loop(1+1,z*z);  
}
```

Inline Expansion after constant folding

```
def loop(int x, int max, int y) =  
  if (x > max) y else loop(x+1, y*y);  
def f(int z) = loop(1,10,z);
```

```
def f(int z) = {  
  val loop= (int xX, int yX) =>  
    if (xX > 10) yX else loop(xX+1, yX*yX);  
  loop(2, z*z);  
}
```

Efficient Tail Calls

- ◆ A function call $f(x)$ within a body of a function g is in a *tail position* if calling f is the last thing g will do before returning.
- ◆ We can save stack space and execution time by turning the call to f into a jump to f .
- ◆ For some languages, like Erlang and Scheme, proper tail calls is not an optimization but a **requirement**.

Tail Calls

- ◆ A tail call can be transformed from a call to a jump as follows:
 1. Move actual parameters into argument registers (and stack positions).
 2. Restore callee-save registers.
 3. Pop the stack frame of the calling function.
 4. Jump to the callee.
- ◆ If both the caller and the callee have few arguments so that they all fit in argument registers then step 1 might be eliminated by a coalescing register allocator, and step 2 and 3 might also be unnecessary: the tail call becomes just a jump.

Equational Reasoning

- ◆ In a pure language we can perform β -substitution.
 - ◆ That is, replacing a call to a function with a version of the body of the function where each occurrence of the formal parameter is replaced by the argument.
 - ◆ $((x) \Rightarrow x + x)(42) \beta \rightarrow 42 + 42$
- ◆ Basically: we can perform function calls at compile time.

Optimization of Lazy FP

- ◆ A lazy language allows us to do some optimizations that would not be safe in a strict language:
 - ◆ Invariant hoisting.
 - ◆ Dead code removal (of function calls).
 - ◆ Strictness Analysis.

Optimization of Lazy FP

- ◆ Invariant hoisting:

```
def f(i) = {  
    def g(j) = h(i) * j;  
    g  
}
```

```
def f(i) = {  
    val h = h(i);  
    def g(j) = h * j;  
    g  
}
```

- ◆ If $h(n)$ loops infinitely but the result of $f(n)$ is never called a strict language would loop in the call to $f(n)$.

Optimization of Lazy FP

- ◆ Dead code removal:

```
def f(i:int): int = {  
  var d = g(x);  
  i + 2;  
}
```

- ◆ In an imperative language $g(x)$ can not be removed, there might be side effects.
- ◆ In a strict pure language removing $g(x)$ might turn a non-terminating computation into a terminating one.

Optimization of Lazy FP

- ◆ The overhead of thunk creation and evaluation is quite high, so they should only be used when needed.
- ◆ If a function $f(x)$ is certain to evaluate its argument x , there is no need to create a thunk for x .
- ◆ We can use a *strictness analysis* to find out which arguments should be evaluated at the call site and which should be passed as thunks.
- ◆ In general exact strictness analysis is **not computable** – a conservative approximation must be used, i.e., assume that arguments who can not be proved strict are non-strict.

Implementation of FPL & Concurrency

Advanced Compiler Techniques

2004

Erik Stenman

EPFL

Implementation of FPL

(Repetition)

- ◆ Possible properties of functional languages:
 - ◆ No statements.
 - ◆ Higher order functions.
 - ◆ Pureness.
 - ◆ Laziness.
 - ◆ Automatic memory management.
- ◆ A *declarative language* is a language where the program declares **what** to calculate.
- ◆ In an *imperative language* the program states **how** to calculate.

Higher Order Functions

(Repetition)

- ◆ A function that takes a function as an argument is called a higher order function.
- ◆ E.g.

```
f(x:int, g:int=>int) = x + g(x);
```


Tail calls (Repetition)

- ◆ A function call $f(x)$ within a body of a function g is in a *tail position* if calling f is the last thing g will do before returning.
- ◆ We can save stack space and execution time by turning the call to f into a jump to f .

Continuations

- ◆ We can combine higher order functions with tail calls to get *continuations*.

- ◆ Normally each function returns a value:

```
def f(x:int) = foo(x) + 1;
```

- ◆ We can instead let each function take a *continuation* that tells where the execution is to continue:

```
def f(x:int, c:int=>int) = c(foo(x)+1);
```

Continuation Passing Style (CPS)

- ◆ Continuations are the basis for a compilation technique called *continuation passing style (CPS)*.
- ◆ In CPS all functions are transformed to take one extra argument, the continuation, and the bodies are transformed to call the continuation instead of returning.
- ◆ Also, all nested expressions of the function body are transformed into continuations. (Primitive operations such as $+$ also takes a continuation.)

CPS Transformation

```
def f(x:int) = foo(x) + 1;
```

```
def f(x:int, c:int=>int) =  
  foo(x,  
    (v:int) => +(v, 1, c)  
  )
```

CPS Transformation

- ◆ CPS transformation is used in many compilers for functional languages such as Scheme and ML.
- ◆ CPS was studied extensively by e.g. Steele in the Rabbit Scheme compiler, and Appel in the SML/NJ compiler.
- ◆ A disadvantage with CPS is that it introduces many closures, and hence the compiler have to optimize as many of them away as possible in order to get good performance.
- ◆ An advantage is that, if closures are your only control structure and you have optimized them to the max, then you have optimized all control structures.

Call with Current Continuation

`call/cc`

- ◆ If we have a language compiled with CPS we can easily implement a very powerful construct called *call/cc* or *call with current continuation*.

```
def call_cc(f, c) = f(c, c)
```

- ◆ That is, we call the function **f** with the current continuation **c** as an argument, and also as the continuation of **f**.
- ◆ With `call/cc` you can “easily” implement backtracking, exceptions, coroutines, and concurrency.

Implementation of Concurrency

- ◆ What is concurrency?
- ◆ Some communication methods.
- ◆ Erlang - a concurrent language.
- ◆ Implementation of Erlang.

Concurrency vs. Parallelism

- ◆ Concurrency:
 - ◆ If two events are *concurrent* then they **conceptually** take place at the same time. That is, different schedulings of two events are indistinguishable or irrelevant.
 - ◆ A **language** can be concurrent.
- ◆ Parallelism:
 - ◆ If two events occur in parallel then they actually occur at the same time.
 - ◆ An **implementation** can be parallel.

Concurrency vs. Parallelism

- ◆ A concurrent language can be implemented either in parallel or sequentially.
- ◆ Some sequential languages can also be implemented either in parallel or sequentially.
 - ◆ Declarative languages are usually easier to make parallel than imperative ones.

Message Passing vs. Shared Memory

- ◆ In a concurrent system with *message passing* each message has to be copied from the sender to the receiver. (Like when sending a mail to someone.)
- ◆ In a *shared memory* system the participating processes can all updated the shared memory, and the new state is “immediately” visible to all. (Like when two people are writing on and looking at the same blackboard.)

Message Passing vs. Shared Memory

- ◆ Shared memory:
 - ◆ Pros:
 1. Performance.
 - ◆ Cons:
 1. The programmer has to ensure consistency.
 2. Can not (practically) be implemented in a distributed system.
- ◆ Message passing:
 - ◆ Pros:
 1. Processes are decoupled (errors don't propagate as easily).
 2. The programmer can reason about the process interaction on a higher level.
 3. Can easily be extended to a distributed system.
 - ◆ Cons:
 1. (Perceived) loss of performance.

Message Passing vs. Shared Memory

- ◆ The distinction between shared memory and message passing is done on the level that the programmer has to deal with.
- ◆ On a lower level message passing can be implemented with shared memory (and often is, at least to some extent).
- ◆ In a network the shared memory model has to be implemented with some form of message passing.

Synchronous vs. Asynchronous

- ◆ In a *synchronous system* both the sender and the receiver have to be in special states (ready to send and ready to receive).
 - ◆ If either of the processes reaches this state before the other it will block and wait until both are in the right state.
- ◆ In an *asynchronous system* the sender does not have to wait for the receiver to be ready in order to send its message.

Synchronous vs. Asynchronous

- ◆ Only one type of primitives is necessary since each can be implemented by the other.
- ◆ To implement synchronization in an asynchronous environment you only need a loop and a protocol where an acknowledgement is sent back upon receive.
- ◆ To implement asynchronous messages in a synchronous environment you need a relaying process.

Processes vs. Threads

- ◆ **In this presentation** *processes* do not refer to OS processes but processes implemented by a programming language.
 - ◆ Such processes can be assumed to be lightweight, not to share memory, and execute concurrently.
- ◆ A *thread* is slightly more heavyweight, share memory and can execute in parallel on a parallel machine.

Concurrency in Programming Languages

- ◆ Concurrency in programming languages can be implemented by utilizing processes or threads from the operating system.
 - ◆ Either directly like in C or with a thin abstraction layer like in Java.
 - ◆ Further abstractions can be built into libraries.
- ◆ Another approach is to build concurrency into the language as such.

Implementation of Concurrency

Example: Erlang

- ◆ Erlang is a concurrent programming language, i.e., concurrency is built into the language from the beginning.
- ◆ Erlang was developed by the Ericsson to be used in large telecom application such as telephone exchanges. (Used in e.g. Ericsson's ATM switch and their GPRS systems.)
- ◆ We will present some details of how to implement a concurrent language by studying how Erlang is implemented.

Erlang

- ◆ The sequential part of Erlang is a small higher order functional language with no mutable data structures.
- ◆ Data in Erlang is represented by a *term*, a term can be a *list* of terms, a *tuple* of terms or ground (*atoms, numbers, PIDs, ...*).
- ◆ Erlang uses *pattern matching* to decompose and switch on the structure of Erlang terms.
- ◆ Erlang **requires** proper tail-calls.

Erlang

- ◆ The concurrent part of Erlang (processes that communicate through message passing) provides the following constructs:
 - ◆ Asynchronous send.
`Receiver ! Message`
 - ◆ Blocking, selective receive with timeouts.
`receive PATTERN -> ... ; after T -> ... end.`
 - ◆ A method to dynamically spawn new processes.
`spawn(Closure) .`
 - ◆ For error correction processes can be linked in order to receive signals when a linked process dies:
`link(Process) .`
or
`spawn_link(Closure).`

A Simple Generic Server

```

loop(State, Handler) ->
  receive
    {From, Request} ->
      {Res, NewState} = Handler(State, Request),
      From ! {self(), Res},
      loop(NewState, Handler);
    {swap_code, NewHandler} ->
      loop(State, NewHandler);
    quit -> ok
  end.

```

```

> Server = spawn(fun() -> loop(0,
                                fun(S, inc) -> {ok, S+1};
                                (S, get) -> {S, S} end)
                                end),
Server ! {self(), inc}, receive {_, _} -> ok end,
Server ! {self(), get}, receive {_, Val} -> Val end.
1
>

```

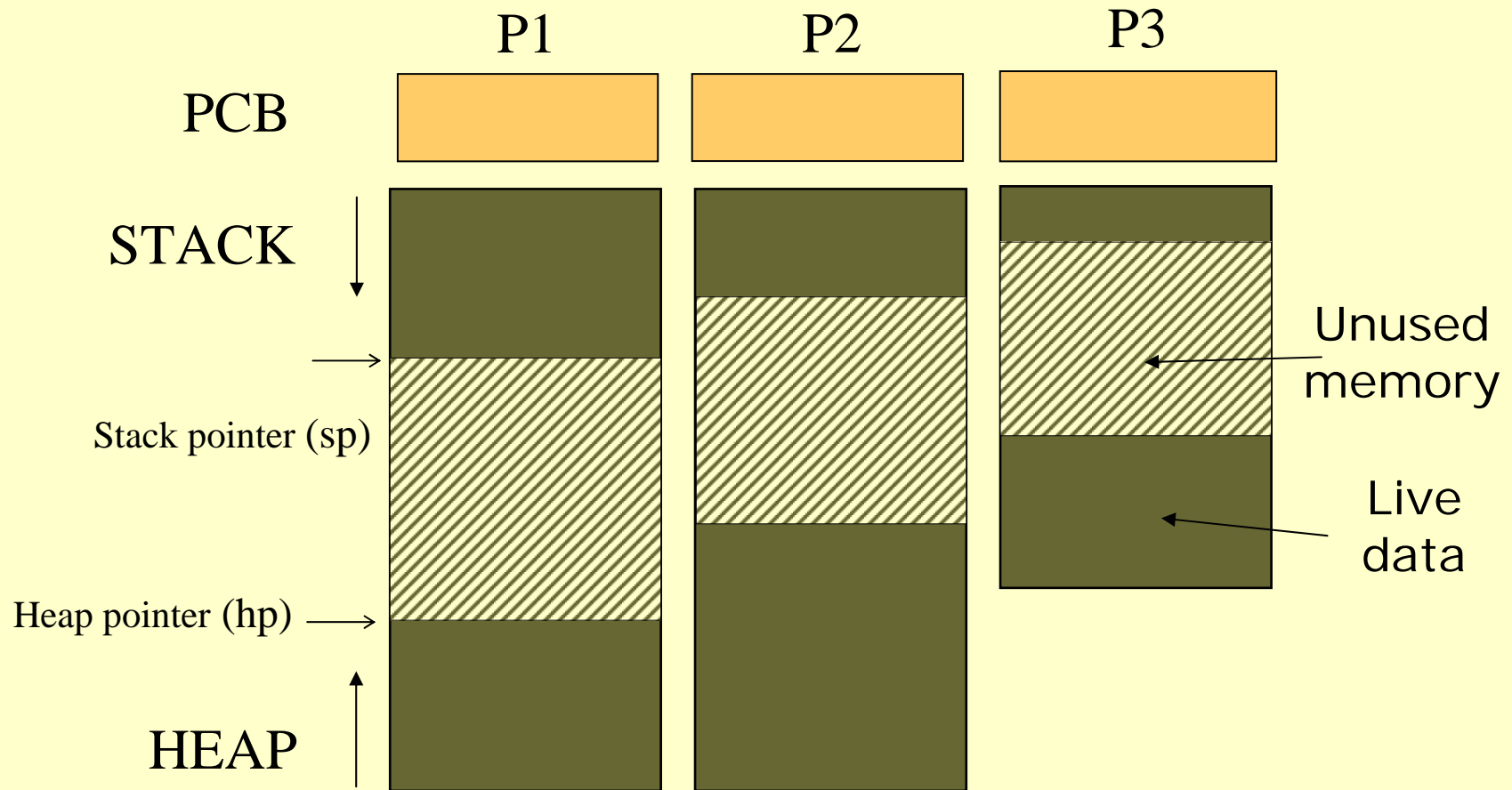
Concurrency in Erlang

- ◆ Erlang is *concurrent*.
 - ◆ The standard implementation is **not** *parallel*, but multi-tasking.
- ◆ Erlang processes are conceptually scheduled with *pre-emptive multitasking* – the programmer does not have to worry about the scheduling.
 - ◆ The standard implementation uses *cooperative multitasking* enforced by the compiler.
 - ◆ Each function call is counted as a *reduction*, when the number of reductions allocated to a process reaches 0 the process is suspended.
 - ◆ Since there are no loop constructs in Erlang other than tail calls, this is sufficient to ensure cooperation.

Implementation of Processes in Erlang

- ◆ Each process has its own stack, heap, message queue, and process control block (PCB).
- ◆ The PCB is relatively small ~70 words.
- ◆ The mailbox is a linked list of pointers to the heap containing only unprocessed messages.
- ◆ The heap and the stack are collocated in one memory area with a default initial size of 233 words. (233=fibonacci(12)).
- ◆ The heap and stack grow (and shrink) as needed.

Processes in Erlang



Process Communication in Erlang

- ◆ All communication between processes in Erlang is done by *message passing*.
- ◆ In the standard implementation this means that all messages are copied between the heap of the sender and the heap of the receiver.
- ◆ This copying is done by first calculating the size of the message, then allocating the right amount on the receivers heap, finally the message is copied.
- ◆ Since the receiver is guaranteed to be suspended, no locking is needed.

Some “Optimizations”

- ◆ Large chunks of immutable data can be stored in *binaries*.
 - ◆ Binaries larger than 64 words are not stored on a process heap and not copied when sent as messages.
 - ◆ Binaries are managed by reference counting.
- ◆ Larger sets of *shared, mutable* data are handled by ETS-tables.
 - ◆ ETS stands for *Erlang Term storage*.
 - ◆ Conceptually an ETS table could be implemented as a process mapping keys to values.
 - ◆ In reality ETS tables are implemented in C.

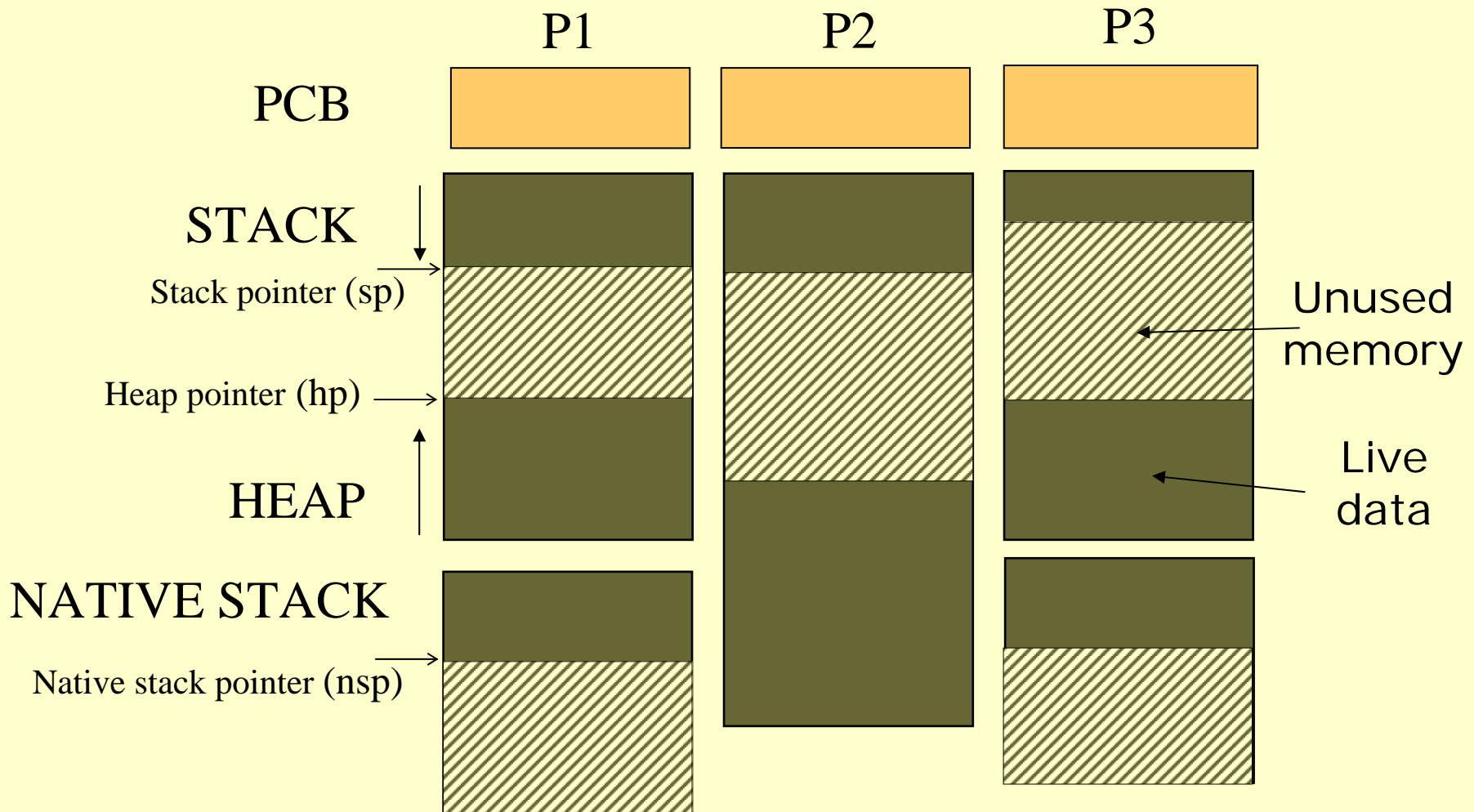
Implementing Erlang in Native Code

- ◆ The standard implementation of Erlang uses a virtual machine (VM). We will discuss how to implement VMs in a later lecture.
- ◆ It is also possible to compile Erlang to native code, here we will present some implementation details for such an implementation.

Implementing Erlang in Native Code

- ◆ In order to enable easy integration with the VM the native implementation uses the same data representation, GC, and runtime system as the VM.
- ◆ The only major difference is that each process that calls native code also get a native stack.

Processes in Erlang



Implementation Details

- ◆ In order to handle scheduling and stack resizing some bookkeeping code is added to the beginning of each function:

```
reductions = reductions - 1;
if (reductions == 0) suspend(p); // p is the current process pointer
checkstack:
if (nsp - STACKNEED < stackEnd) {
    resizeStack();
    goto checkstack;
}
```

Implementation Details

- ◆ The stack need can be calculated at compile time:
number of spills + $\max(\forall \text{ calls: } \text{argsOnStack} + \text{callerSaves}) + \text{buffer}$.
- ◆ By ensuring that there is a buffer of free words on the stack we do not need the bookkeeping code for leaf-functions that uses less than that many words.

Implementation Details

- ◆ The function `suspend` has to be implemented in machine code in order to get access to the return address.

```
suspend: // p (the current process) is passed as the argument.  
p->pc = <RETADDRESS> // From the stack on x86 from a register on SPARC  
p->status = READY;  
SAVE(p); // Save the process sp, switch to C stack.  
add(p, readyQueue);  
p = schedule();  
RESTORE(p); // Restore the process sp, switch from C stack.  
jmp p->pc;
```

The Scheduler

- ◆ Since Erlang does not use OS processes or threads, the Erlang runtime system has to implement its own scheduler. (In, e.g., C)

```
pid schedule() {
    static int majorReductions = MREDS;
    majorReductions--;
    if(majorReductions == 0) { externalPoll();
        majorReductions = MREDS; }
    checkTimeouts();
    pid p = nextReady(readyQueue);
    p->reductions = REDS; p->status = RUNNING;
    return p;
}
```


Send

- ◆ A message send from p1 to p2 can be implemented as:

```
send(pid:p1, pid:p2, term:message) {  
    int s = size(message);  
    if(s > (p2->heapTop - p2->heapPointer)) gc(p2, s);  
    term mp = copy(message, p2->heapTop);  
    add(mp, p2->messageQueue);  
    if(p2->status == SUSPENDED) {  
        p2->status = READY;  
        add(p2, readyQueue);  
    }  
}
```

Receive

- ◆ A message receive is slightly more complicated.

```
messageLoop:
```

```
  m = nextMessage(p);
```

```
  if (m == NIL)
```

```
    sleep(p, timeout, &messageLoop, &handler);
```

```
  cont = MATCH(m, PATTERNS);
```

```
  if (cont == 0) goto messageLoop;
```

```
  unlink(p);
```

```
  jmp cont;
```

```
handler:
```

```
...
```

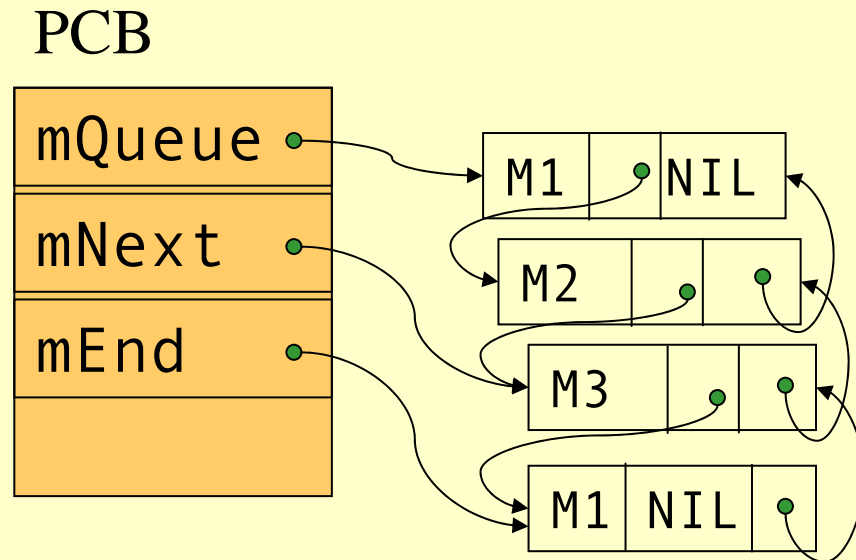
Receive

```

nextMessage(pid p) {
  term m = p->mNext;
  p->mNext = m->next;
  return m;
}

unlink(pid p) {
  term m = p->mNext;
  if(m->prev != NIL)
    p->prev->next = m->next;
  else
    p->mQueue = m->next;
  if(p->mEnd == m)
    p->mEnd = m->prev;
  p->mNext = p->mQueue;
}

```



Receive

```
sleep(p, timeout, messageLoop, handler) {  
    p->pc = messageLoop;  
    p->handler = handler;  
    add(p, now()+timeout, timeoutQueue)  
    p->status = SUSPENDED;  
    p = schedule();  
    (p->pc)();  
}
```

Receive

- ◆ The `checkTimeout` function in the scheduler will activate a process when the timeout has elapsed.
- ◆ While doing so `p->pc` will be updated with `p->handler` so that the process will start executing in the timeout handler when scheduled.

Spawn

- ◆ The spawn primitive creates a new process, i.e. allocates a new PCB, stack, and heap.
- ◆ Then the argument to spawn (the closure) is copied to the new heap.
- ◆ The new pid is added to the ready queue.
- ◆ Then execution continues in the old process with the instructions after spawn.

Summary

- ◆ Concurrency is an important concept that can be useful as an abstraction when decomposing a program, just as modules, objects, and functions.
- ◆ Concurrency can be implemented by either using primitives provided by the OS or by implementing a scheduler specifically for the language.

Memory Management

Advanced Compiler Techniques

2004

Erik Stenman

EPFL

Memory Management

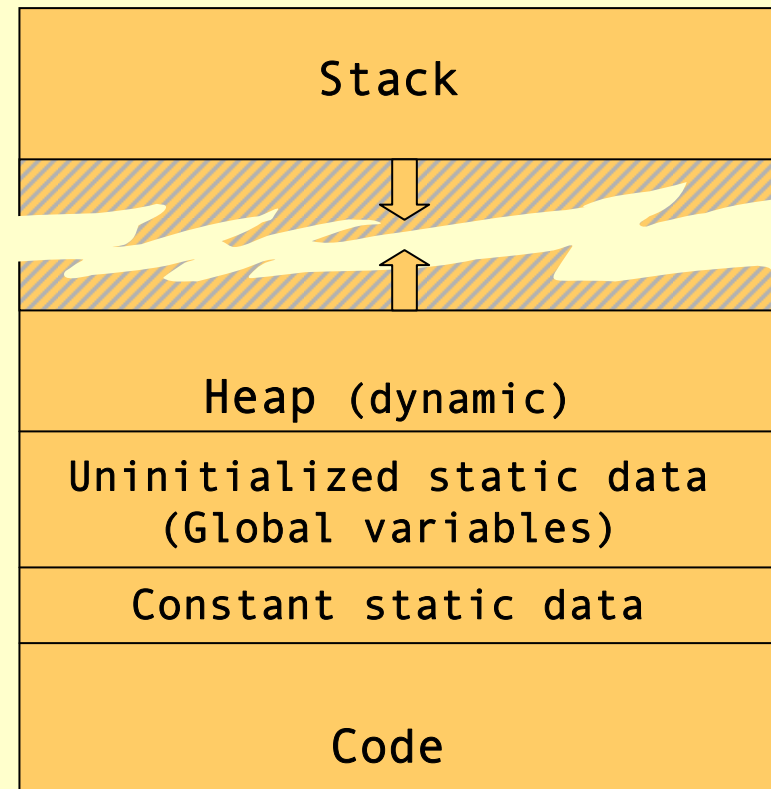
- ◆ The computer memory is a limited resource so the memory use of programs has to be managed in some way.
- ◆ The memory management is usually performed by a *runtime system* with help from the compiler.
 - ◆ The runtime system is a set of system procedures linked to the program.
 - ◆ For C programs it can be as simple as a small library for interacting with the operating system.
 - ◆ For Erlang programs the runtime system implements almost all the functionality normally provided by the OS.

Memory Management

- ◆ In a language such as C there are three ways to allocate memory:
 1. Static allocation. The memory needed by global variables (and code) is allocated at compile time.
 2. Stack allocation. Activation records are allocated on the stack at function calls.
 3. Heap allocation. Dynamically allocated by the programmer by the use of `malloc`.

Memory Organization

- ◆ A typical layout of the memory of a C program looks like:



Dynamic Memory Management

- ◆ Heap allocation is necessary for data that lives longer than the function which created it, and which is passed by reference, e.g., lists in misc.
- ◆ Two design questions for the heap:
 - ◆ How is space for data allocated on the heap?
 - ◆ How and when is the space deallocated?
- ◆ Considerations in memory management design:
 - ◆ Space leaks & dangling pointers.
 - ◆ The cost for allocation and deallocation.
 - ◆ Space overhead of the memory manager.
 - ◆ Fragmentation.

Fragmentation

- ◆ The memory management system should try to avoid *fragmentation*, i.e. when the free memory is broken up into several small blocks instead of few large blocks.
- ◆ In a fragmented system memory allocation may fail because there is no free block that is large enough even though the total free memory would be large enough.
- ◆ We distinguish between:
 - ◆ Internal fragmentation – the allocated block is larger than the requested size (the waste is in the allocated data).
 - ◆ External fragmentation – all free blocks are too small (the waste is in the layout of the free data).

Memory Allocation

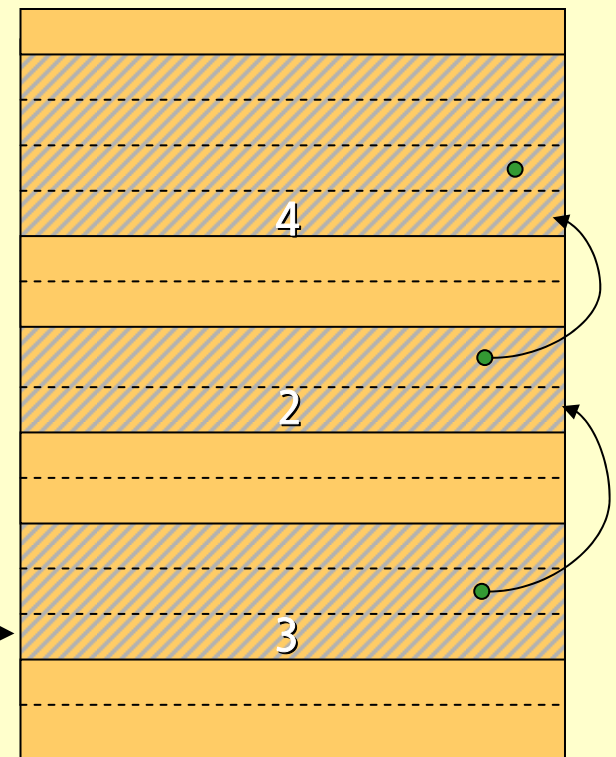
- ◆ The use of a free-list is a common scheme.
- ◆ The system keeps a list of unused memory blocks.
- ◆ To allocate memory the free-list is searched to find a block which is large enough.
- ◆ The block is removed from the free-list and used to store the data. If the block is larger than the need, it is split and the unused part is returned to the free-list (to avoid internal fragmentation).
- ◆ When the memory is freed it is returned to the free-list. Adjacent memory blocks can be merged (or coalesced) into larger blocks (to avoid external fragmentation).

Free-list

- ◆ The free-list can be stored in the free memory since it is not used for anything else. (We assume, or ensure, that each memory block is at least two words).

Free list: ●

This can be stored as a static global variable.



Free-list

- ◆ Note that we **need to know the size** of a block when it is deallocated. This means that even allocated blocks need to have **a size field** in them.
- ◆ Thus the space overhead will be at least **one word per allocated data object**. (It might also be advantageous to keep the link.)
- ◆ The cost (time) of allocation/deallocation is proportional to the search through the free-list.

Free-list

- ◆ There are many different ways to implement the details of the free-list algorithm:
 - ◆ Search method: first-fit, best-fit, next-fit.
 - ◆ Links: single, double.
 - ◆ Layout: one list, one list per block size, tree, buddy.

Deallocation

- ◆ Deallocation can either be *explicit* or *implicit*.
- ◆ Explicit deallocation is used in e.g., Pascal (new/dispose), C (malloc/free), and C++ (new/delete).
- ◆ Implicit deallocation is used in e.g., Lisp, Prolog, Erlang, ML, and Java.

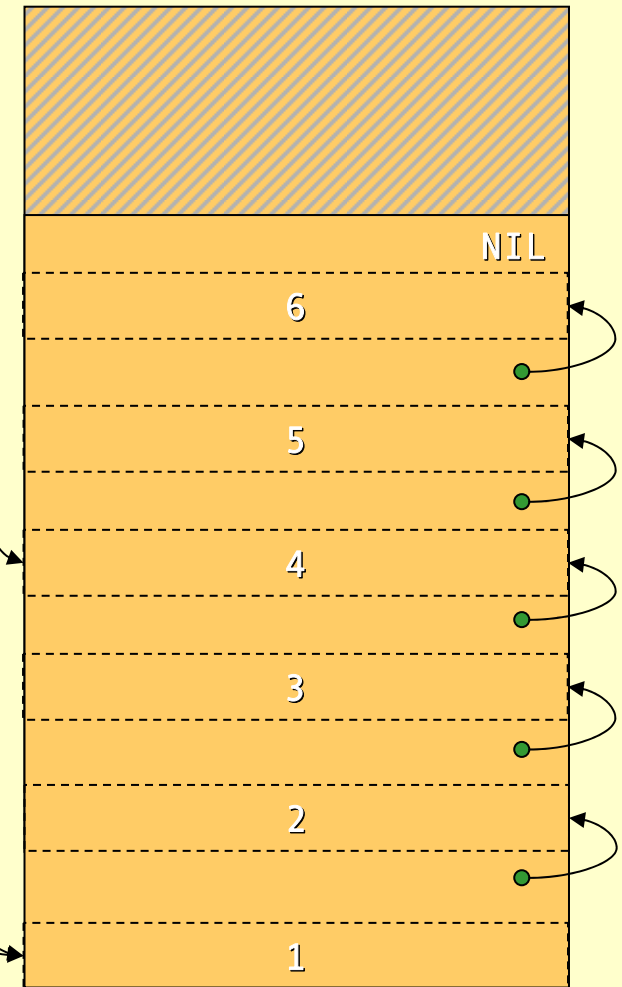
Explicit Deallocation

- ◆ Explicit deallocation has a number of problems:
 - ◆ If done too soon it leads to dangling pointers.
 - ◆ If done too late (or not at all) it leads to space leaks.
 - ◆ In some cases it is almost impossible to do it at the right time. Consider a library routine to append two destructive lists:

```
c = append(a, b);
```

Explicit Deallocation

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
printList(b);
```



Explicit Deallocation

```
list a = new List(1,2,3);  
list b = new List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
printList(b);  
free(c);
```

- ◆ The programmer now has to ensure that *a*, *b*, and *c* are all deallocated at the same time. A mistake would lead to dangling pointers.
- ◆ If *b* is in use long after *a*, and *c*, then we will keep *a* live too long. A space leak.

Implicit Deallocation

- ◆ With *implicit deallocation* the programmer does not have to worry about when to deallocate memory.
- ◆ The runtime system will *dynamically* decide when it is **safe** to do this.
- ◆ In some cases, and systems, the compiler can also add static deallocations to the program.
- ◆ The most commonly used automatic deallocation method is called *garbage collection* (GC).
- ◆ There are other methods such as *region based* allocation and deallocation.

Garbage Collection (GC)

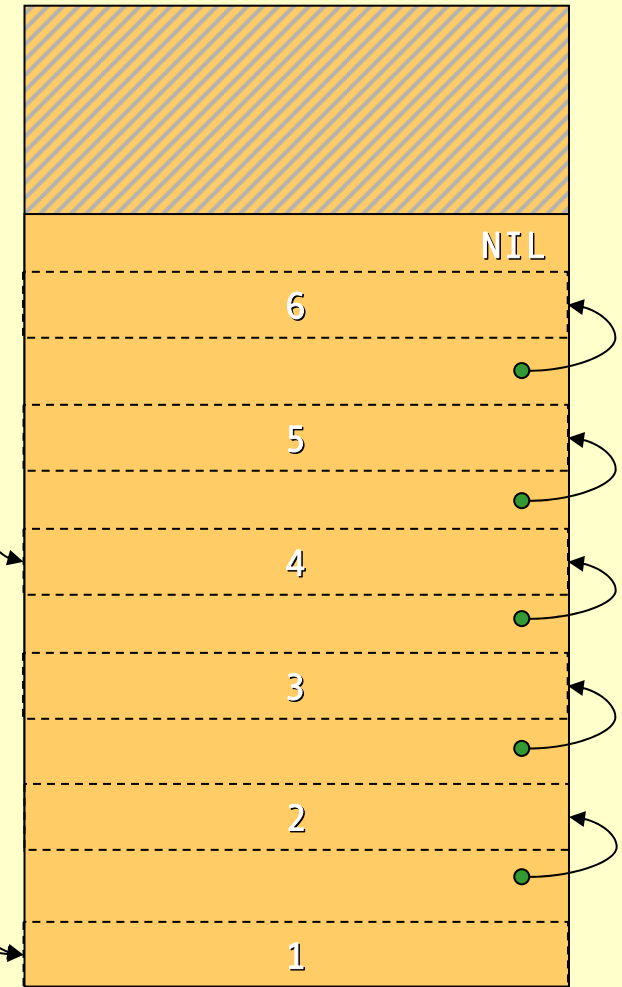
- ◆ *Garbage collection* is a common name for a set of techniques to deallocate heap memory that is unreachable by the program.
- ◆ There are several different base algorithms: *reference counting, mark & sweep, copying.*
- ◆ We can also distinguish between how the GC interferes or interacts with the program: *disruptive, incremental, real-time, concurrent.*

The Reachability Graph

- ◆ The data reachable by the program form a directed graph, where the edges are pointers.
- ◆ The *roots* of this graph can be in:
 1. global variables,
 2. registers,
 3. local variables & formal parameters on the stack.
- ◆ Objects are *reachable* iff there is a path of edges that leads to them from some root. Hence, the compiler must tell the GC where the roots are.

The Reachability Graph

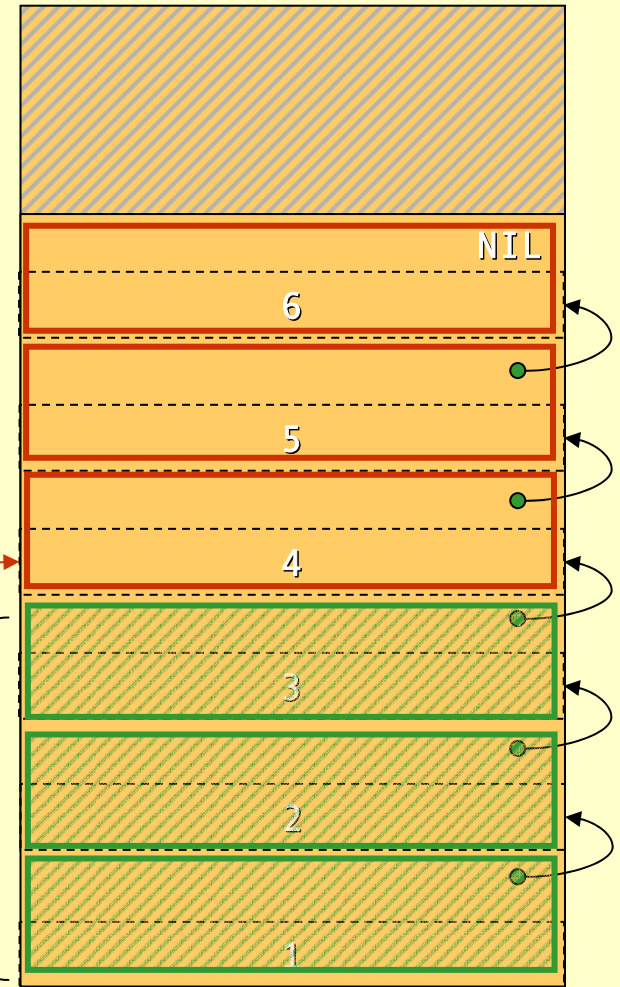
```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```



The Reachability Graph

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;          roots: b
```

The goal with the GC is to deallocate these:

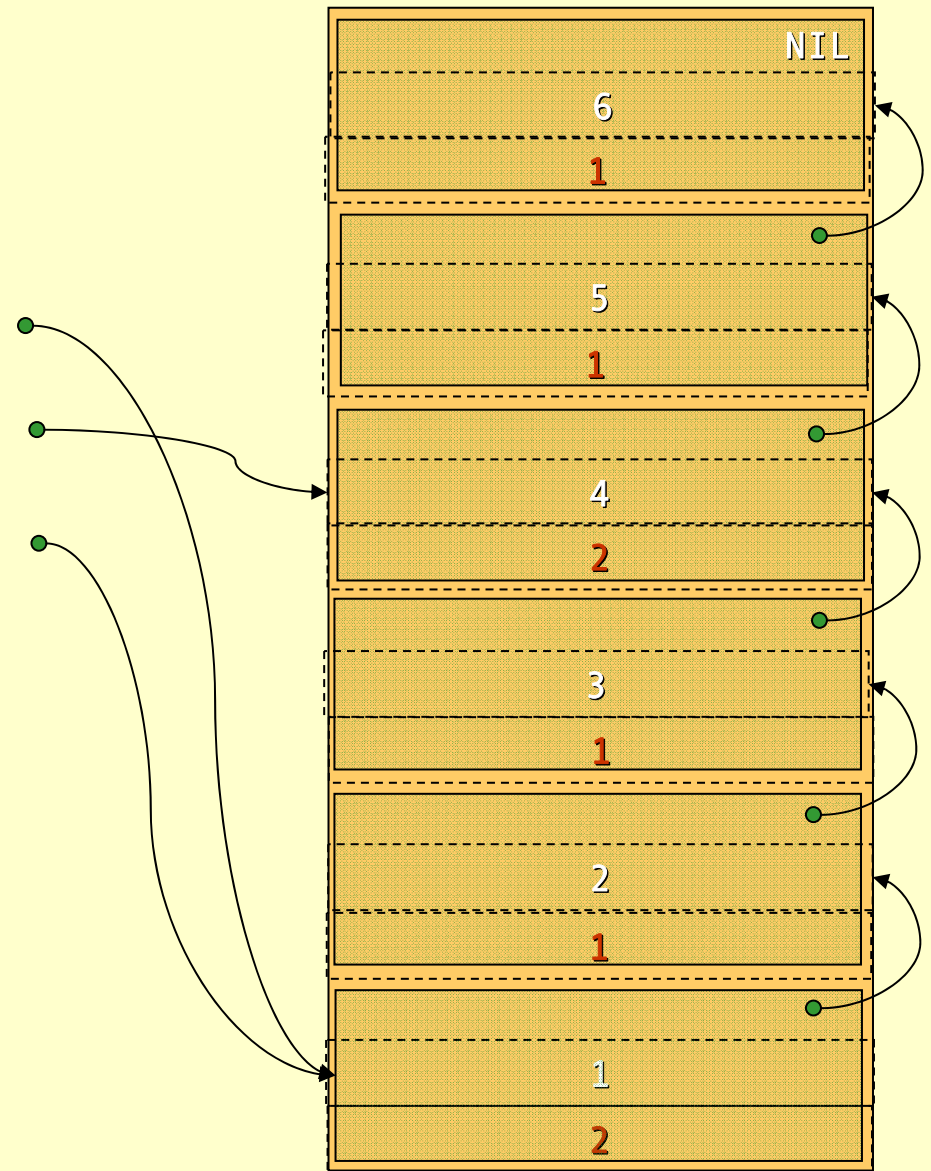


Reference Counting

- ◆ Idea: Keep track of how many references there are to each object.
- ◆ If there are 0 references deallocate the object.
- ◆ The compiler must add code to maintain the reference count (refcount).
 - ◆ Set the count to 1 when created.
 - ◆ For an assignment $x = y$:
 - ◆ if ($x \neq \text{null}$) $x.\text{refcount} -$;
 - ◆ if ($y \neq \text{null}$) $y.\text{refcount}++$;
 - ◆ When a stack frame is deallocated decrease the refcount of each object pointed to from the frame.
 - ◆ When refcount reaches 0 deallocate the object and decrease refcount of each child.

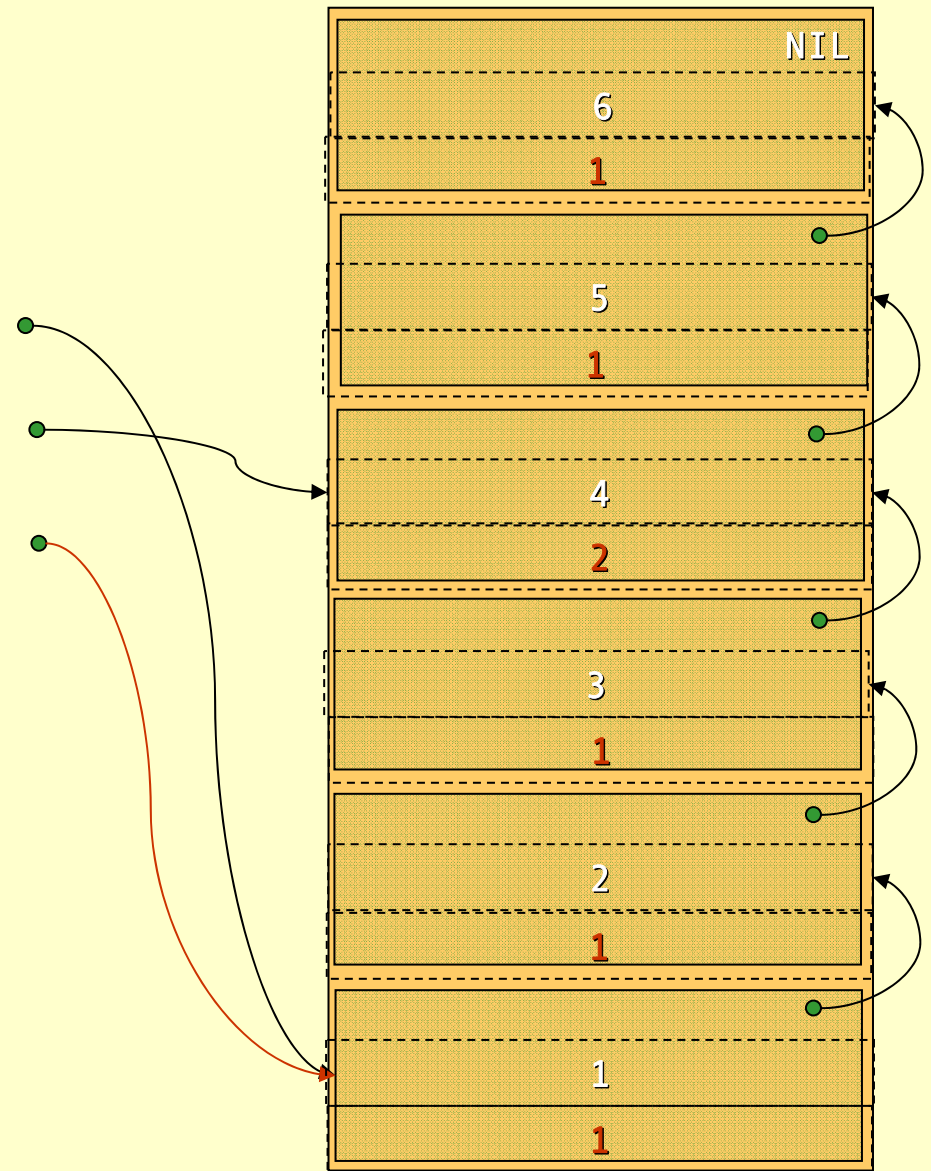
```

list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
decRefCount(c);
decRefCount(a);
doLotsOfStuff();
return b;
    
```



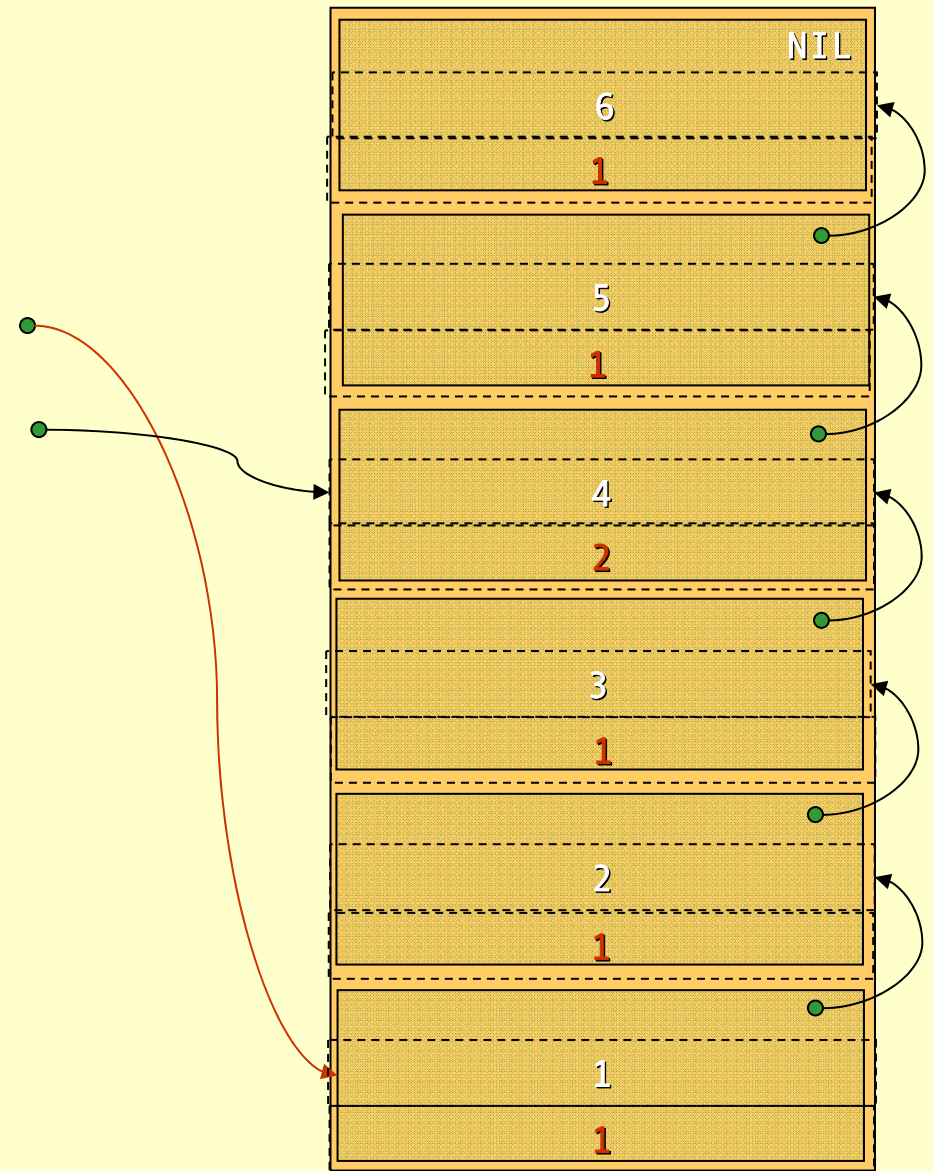
```

list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
decRefCount(c);
decRefCount(a);
doLotsOfStuff();
return b;
    
```



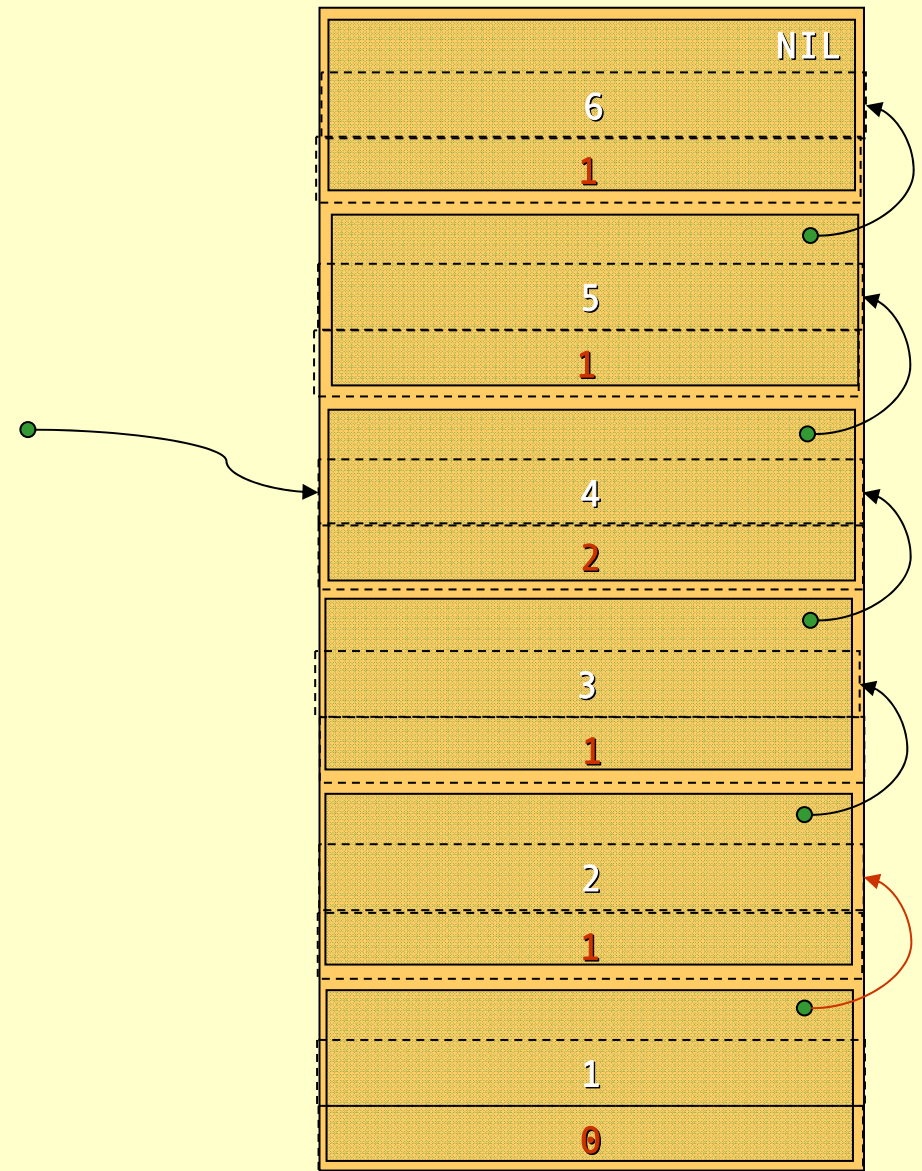
```

list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
decRefCount(c);
➔ decRefCount(a);
doLotsOfStuff();
return b;
    
```



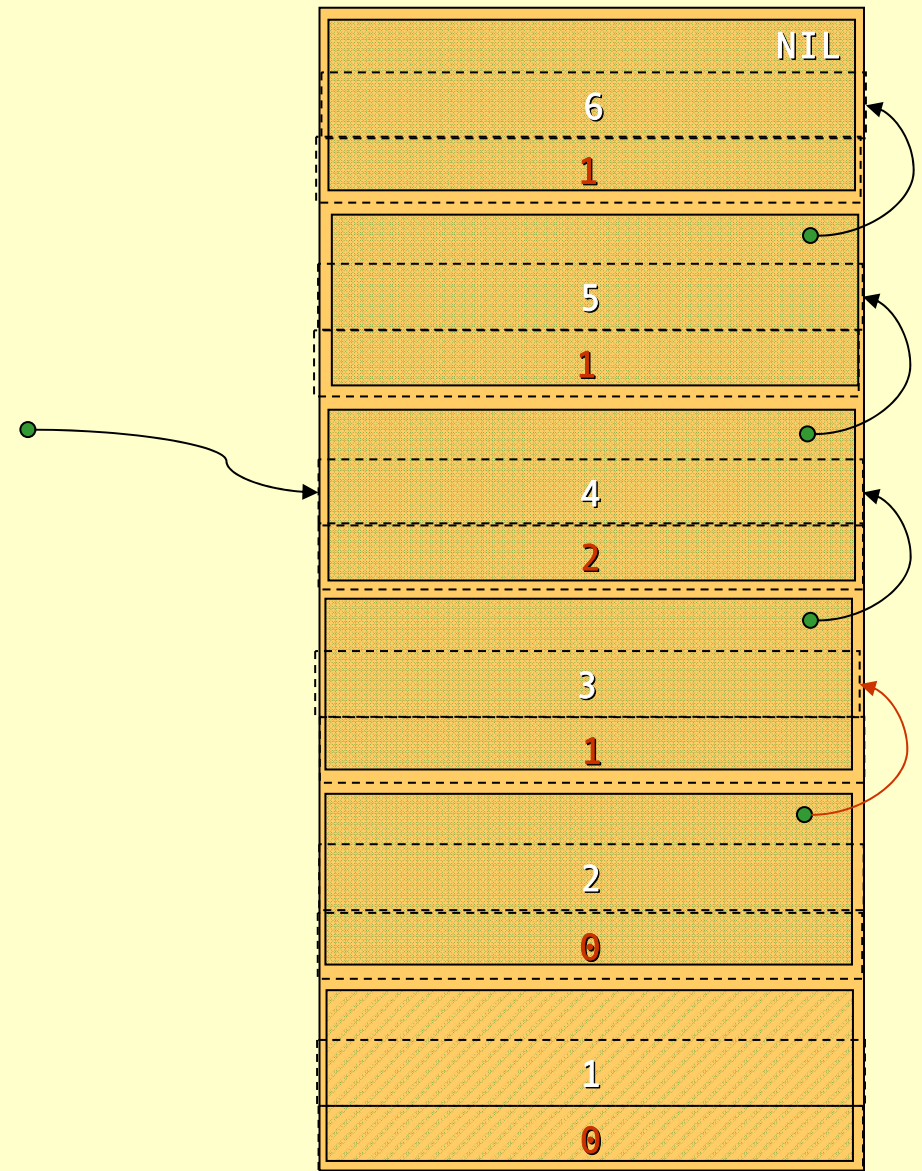
```

list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
decRefCount(c);
➔ decRefCount(a);
doLotsOfStuff();
return b;
    
```



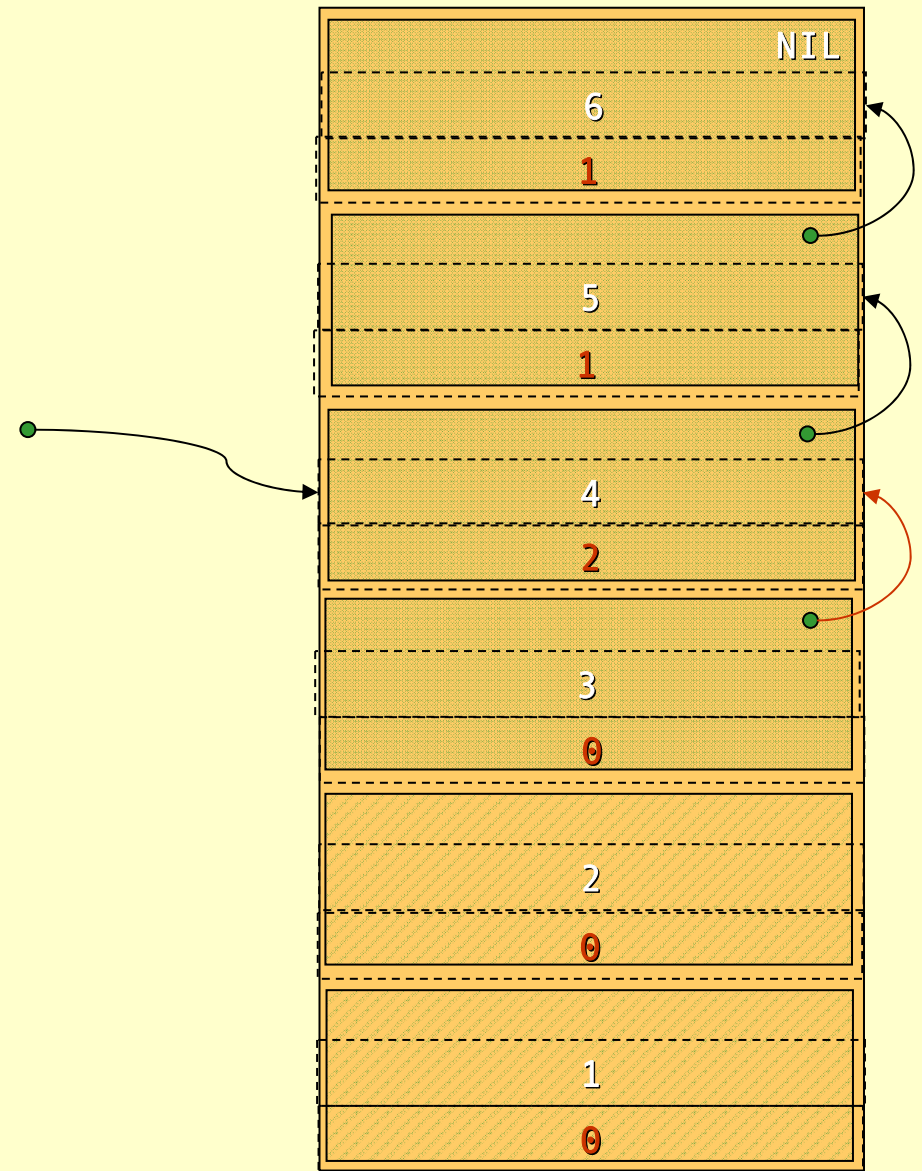
```

list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
decRefCount(c);
➔ decRefCount(a);
doLotsOfStuff();
return b;
    
```




```

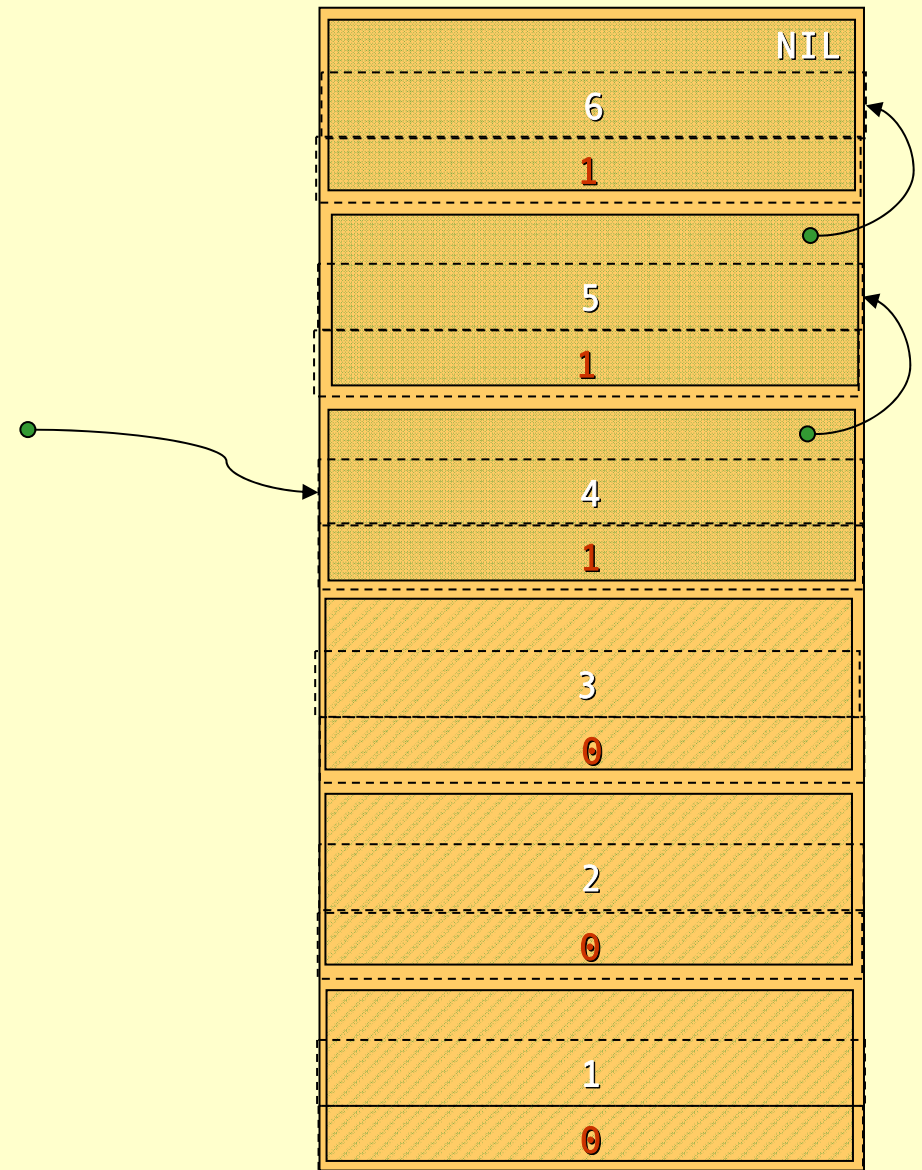
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
decRefCount(c);
➔ decRefCount(a);
doLotsOfStuff();
return b;
    
```



```

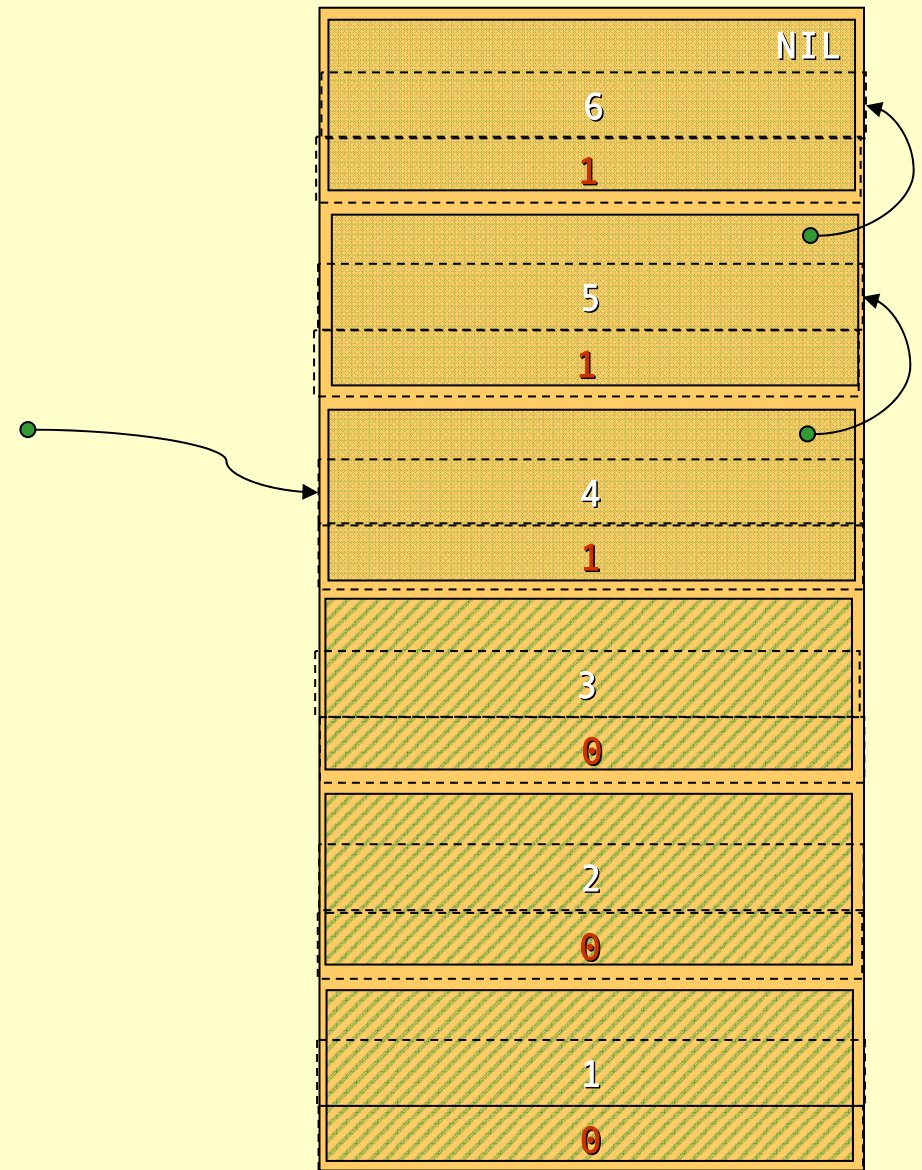
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
decRefCount(c);
➔ decRefCount(a);
doLotsOfStuff();
return b;

```



```

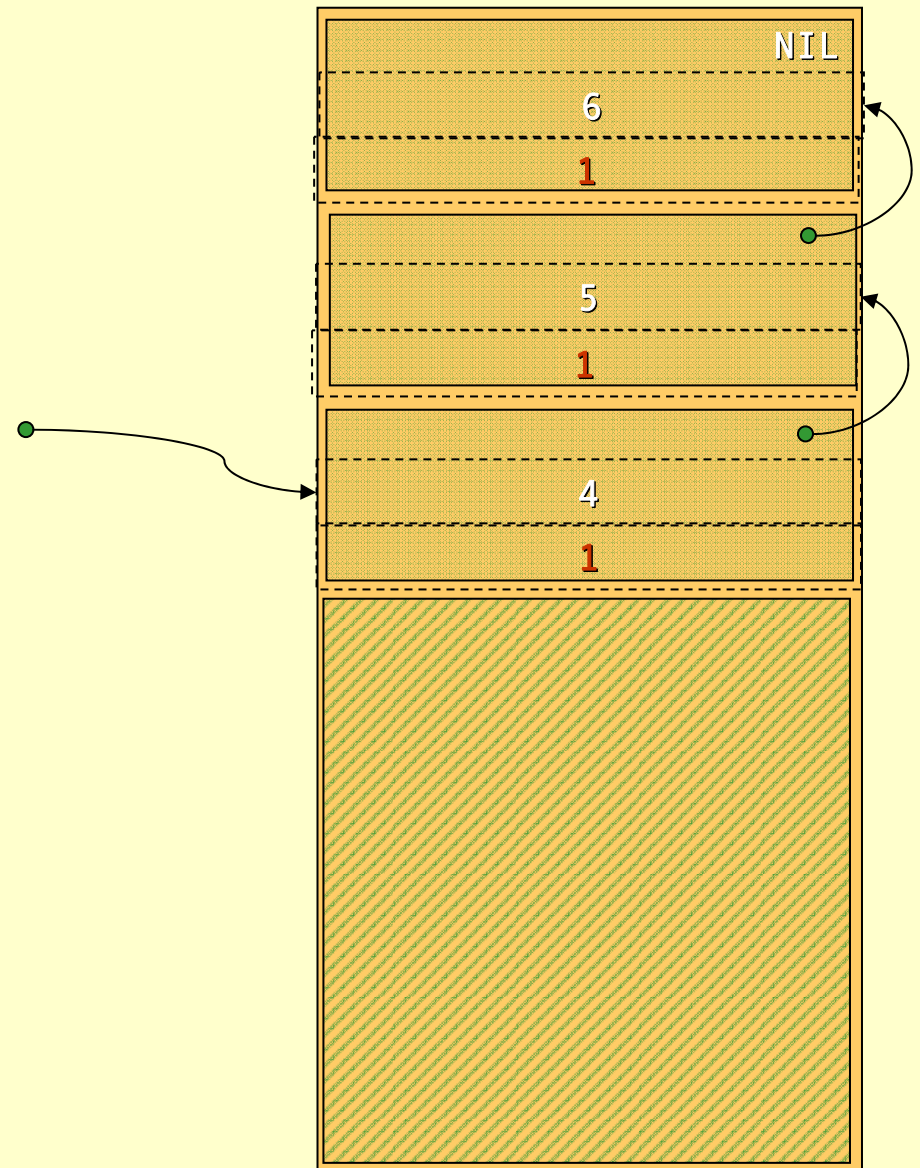
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
decRefCount(c);
➔ decRefCount(a);
doLotsOfStuff();
return b;
    
```



```

list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
decRefCount(c);
decRefCount(a);
doLotsOfStuff();
return b;

```



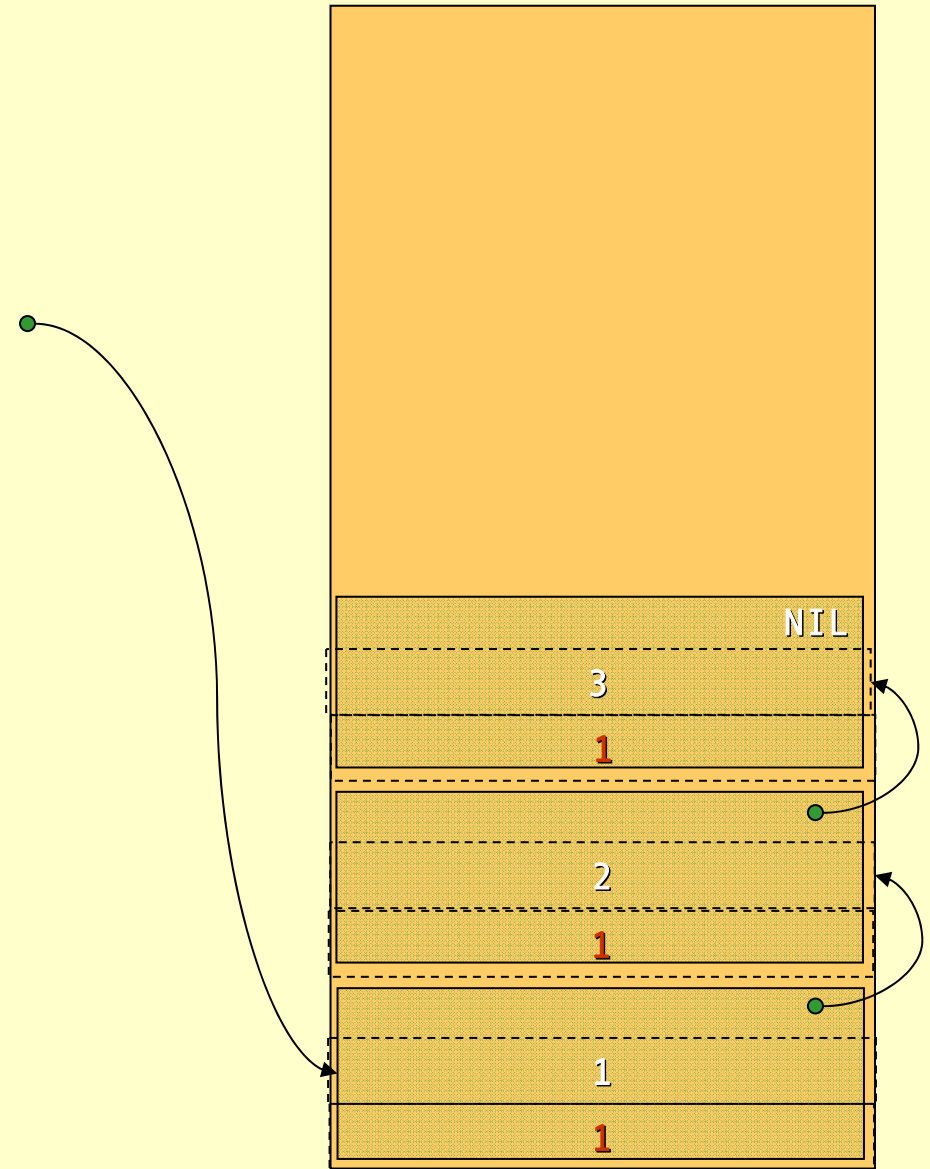
Reference Count

- ◆ Advantages of reference count:
 - ◆ Rather easy to implement.
 - ◆ Storage reclaimed **immediately**.
- ◆ Disadvantages of reference count:
 - ◆ **Space overhead**: 1 word per object.
 - ◆ Keeping track of the reference counts is **very expensive**. (Each simple pointer copy becomes several instructions.)
 - ◆ There is one more problem...

```

➔ list a = List(1,2,3);
  list b = NIL;
  list c = append(a,a);
  printList(c);
  decRefCount(c);
  decRefCount(a);
  doLotsOfStuff();
  return b;

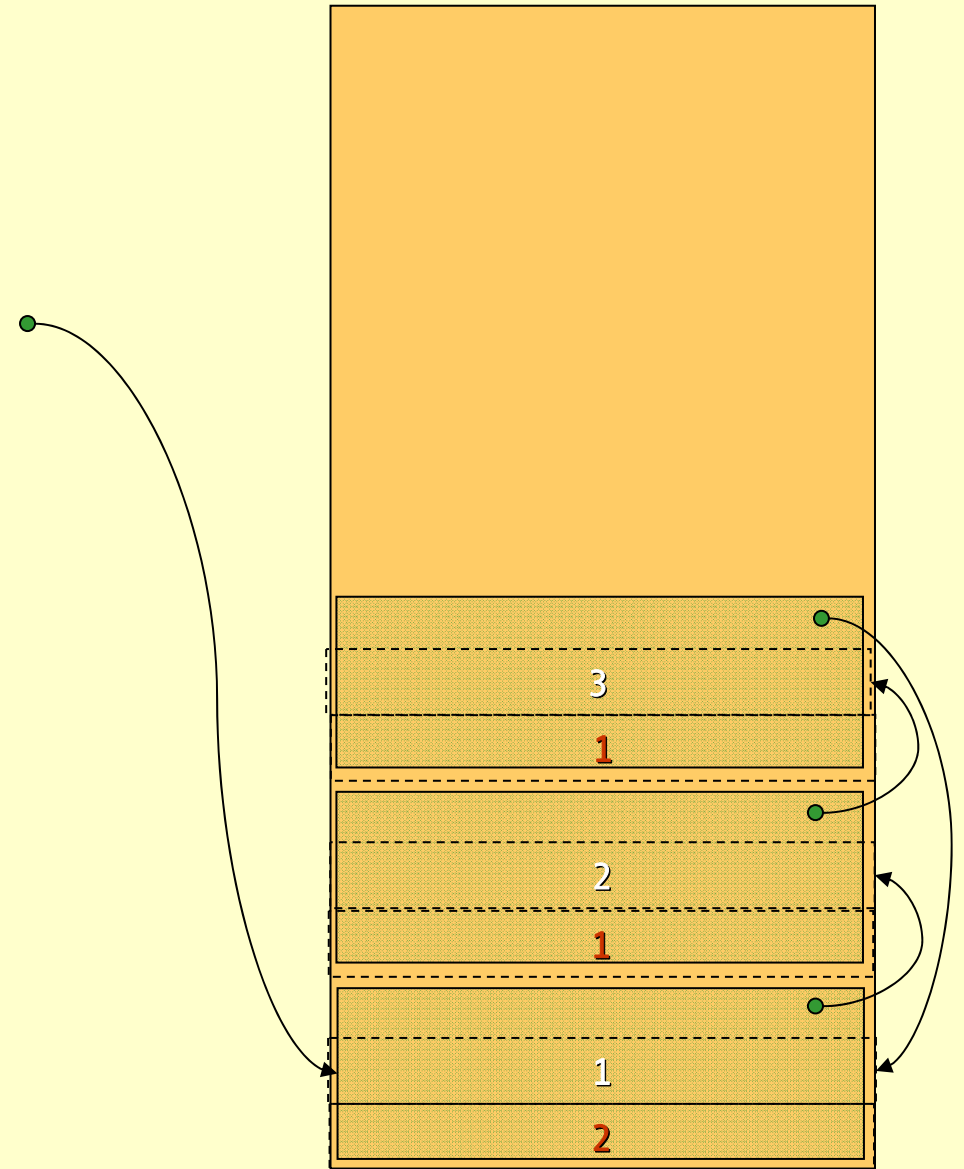
```



```

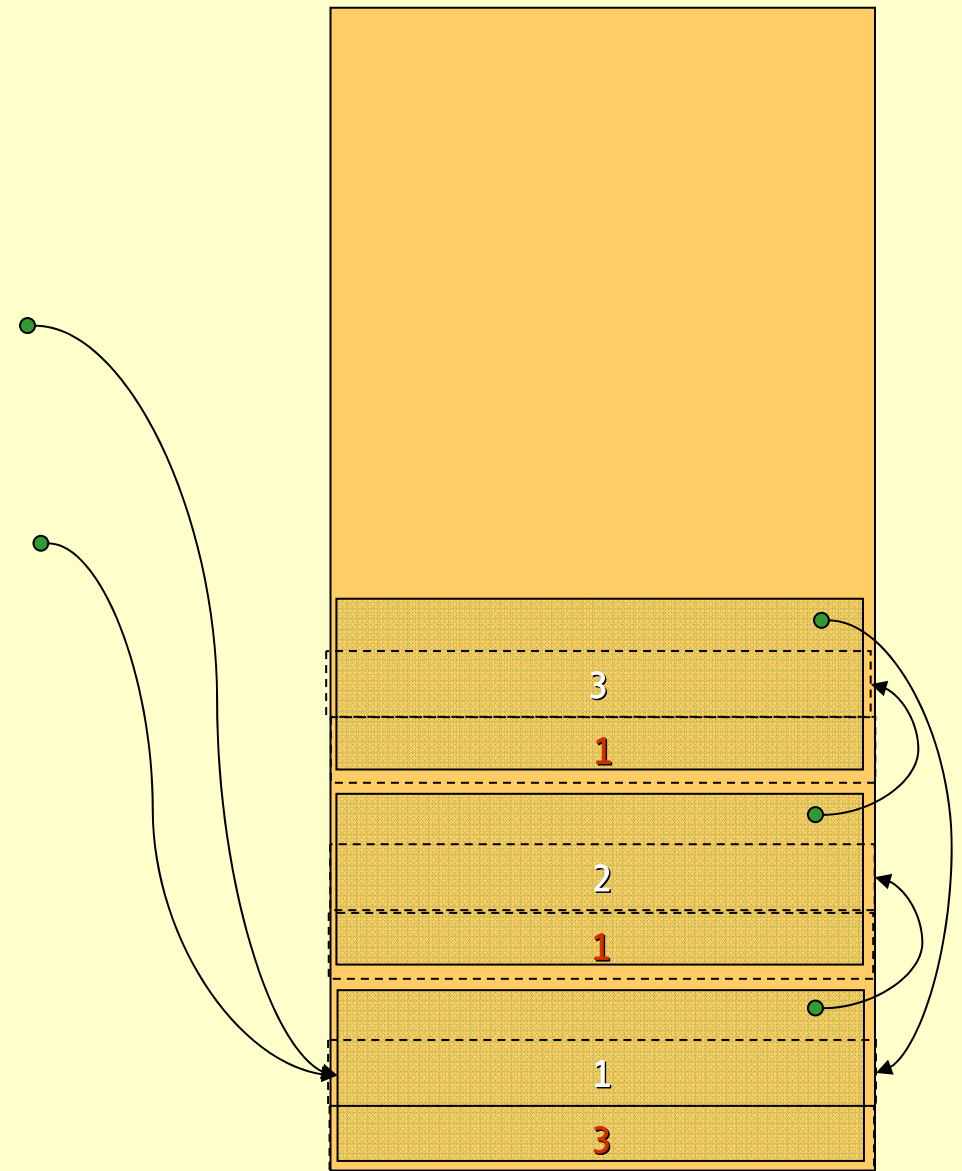
list a = List(1,2,3);
list b = NIL;
→ list c = append(a,a);
printList(c);
decRefCount(c);
decRefCount(a);
doLotsOfStuff();
return b;

```



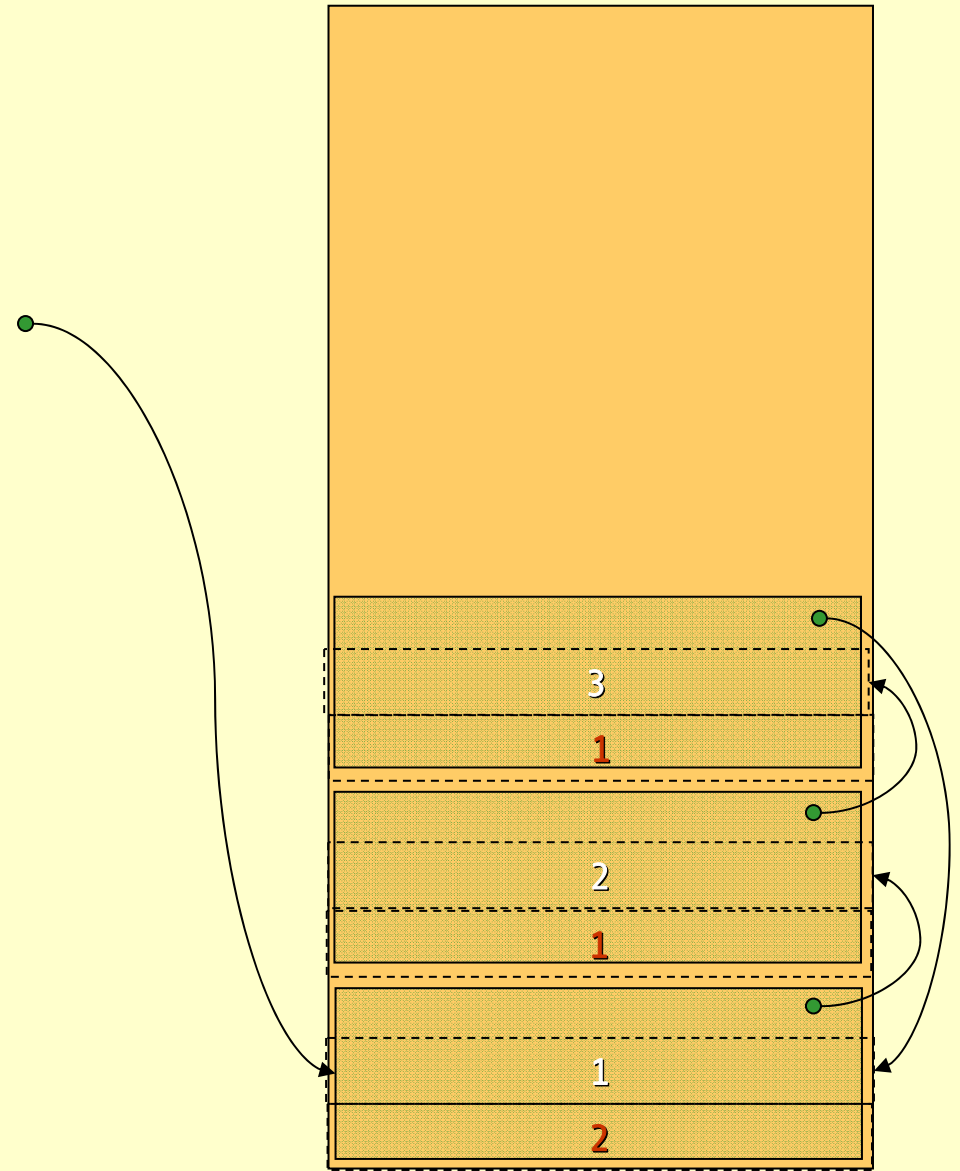
```

list a = List(1,2,3);
list b = NIL;
list c = append(a,a);
printList(c);
decRefCount(c);
decRefCount(a);
doLotsOfStuff();
return b;
    
```




```

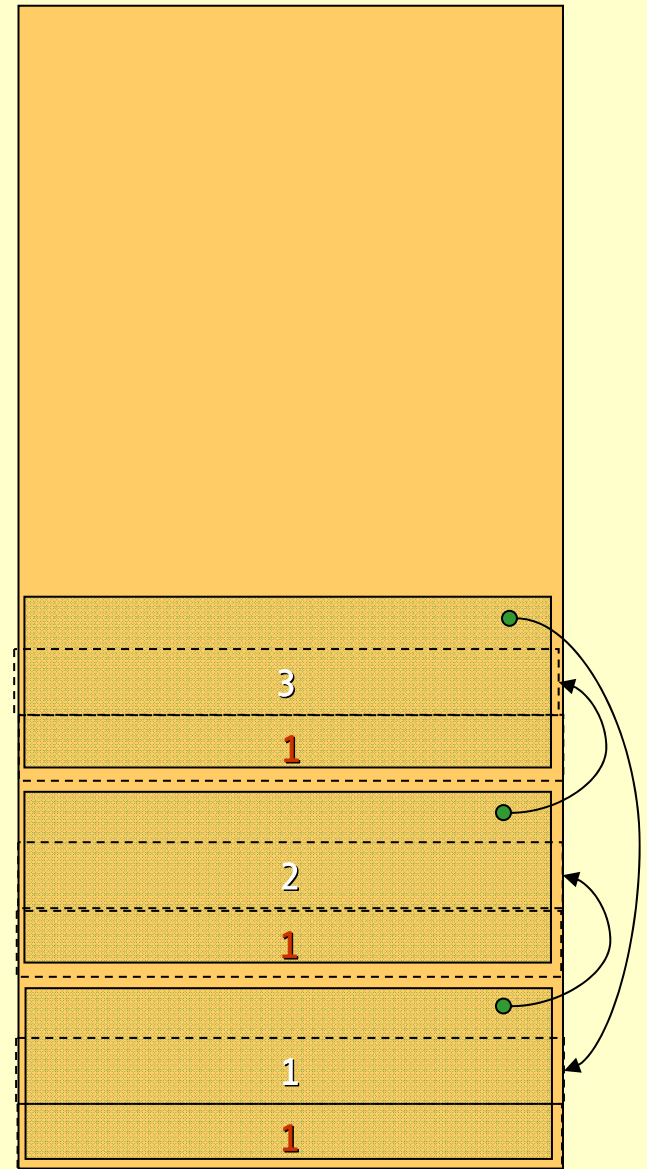
list a = List(1,2,3);
list b = NIL;
list c = append(a,a);
printList(c);
decRefCount(c);
→ decRefCount(a);
doLotsOfStuff();
return b;
    
```



```

list a = List(1,2,3);
list b = NIL;
list c = append(a,a);
printList(c);
decRefCount(c);
decRefCount(a);
doLotsOfStuff();
return b;

```



Reference Count

- ◆ Big disadvantage with reference count:
 - ◆ The refcount of *cyclic structures* never reaches zero!
- ◆ There are ways to solve this, but they are very complicated.
- ◆ Due to this fact reference count is *very seldom* used in practice. There is one nice use, as we shall see later...
- ◆ In a pure language or a language without destructive updates there are no cyclic structures, making reference counting a viable option.

Mark & Sweep

- ◆ A *mark & sweep* GC is made up of two *phases*:
 1. First all reachable objects are *marked*.
 2. Then the heap is *swept* clean of dead objects.
- ◆ The mark phase is done by a *depth first search* through the reachability graph starting from the roots.

Depth First Mark Algorithm

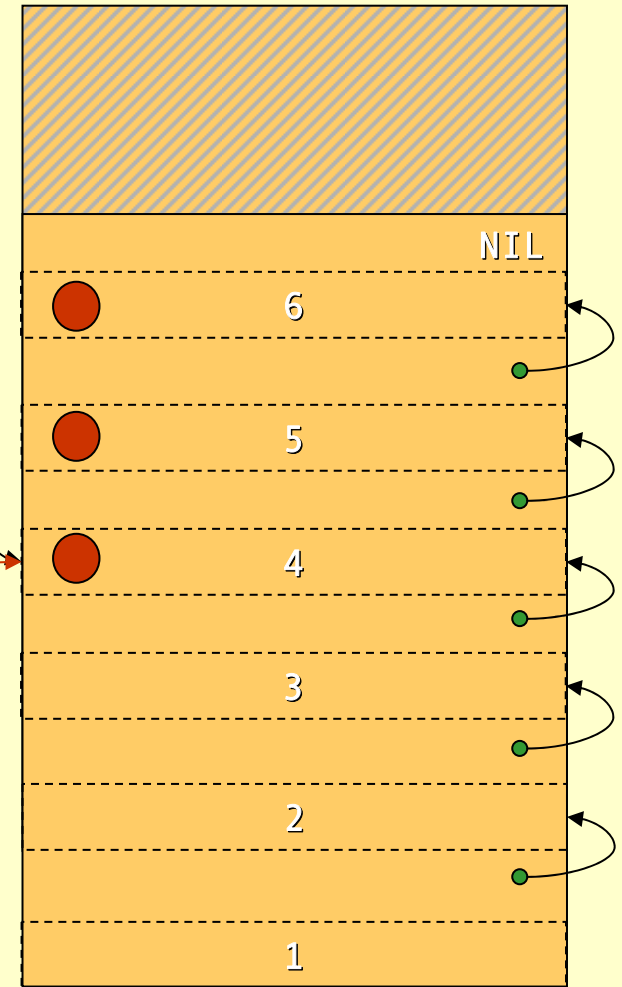
```
mark(x) {  
    if(! marked(x)) {  
        setMark(x);  
        for each field f of x  
            mark(*f)  
    }  
}
```

Example: Mark

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```



mark(b)



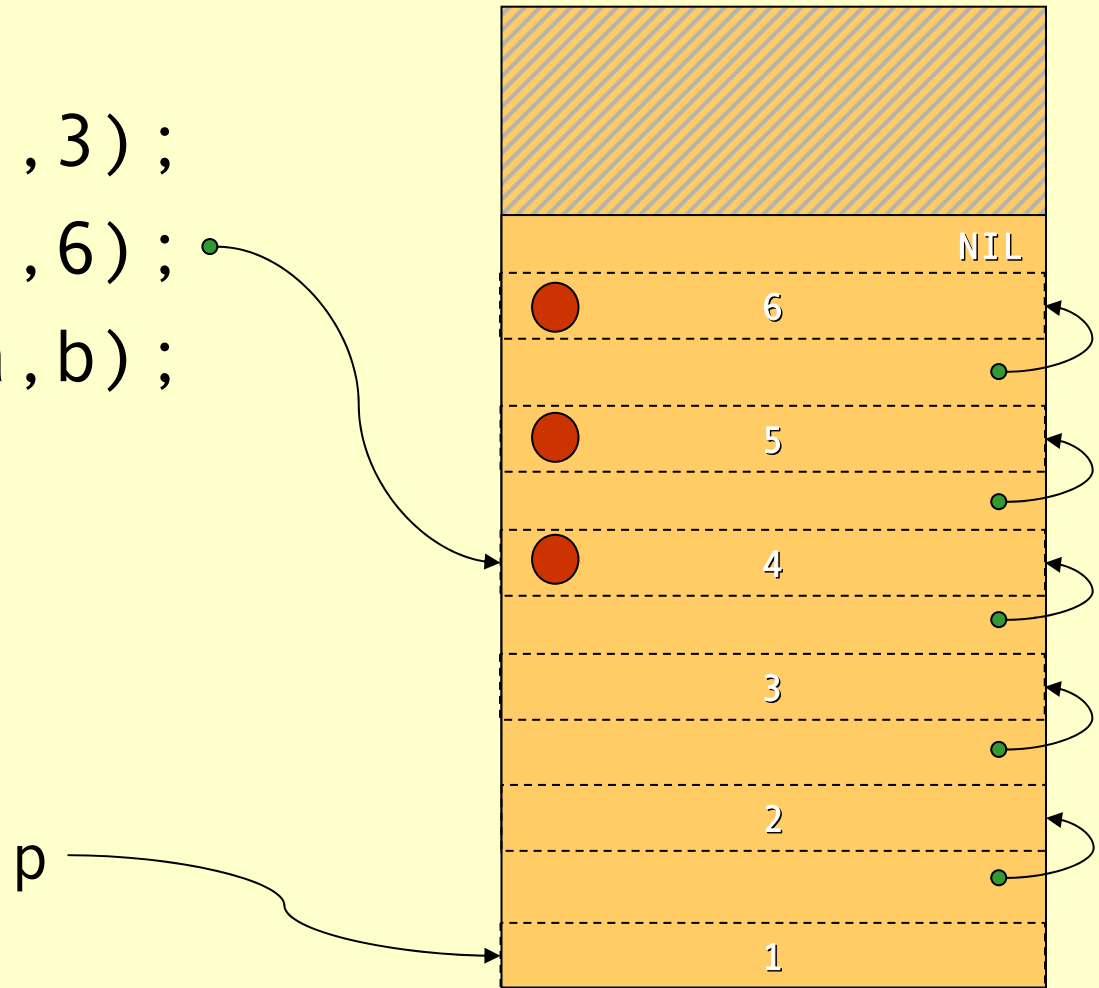
The Sweep

- ◆ The Sweep phase goes through the whole heap from start to finish and adds unmarked objects to the free-list.

```
p = heapStart;
while (p < heapEnd) {
    if (marked(*p)) clearMark(*p);
    else free(p);
    p += size(*p);
}
```

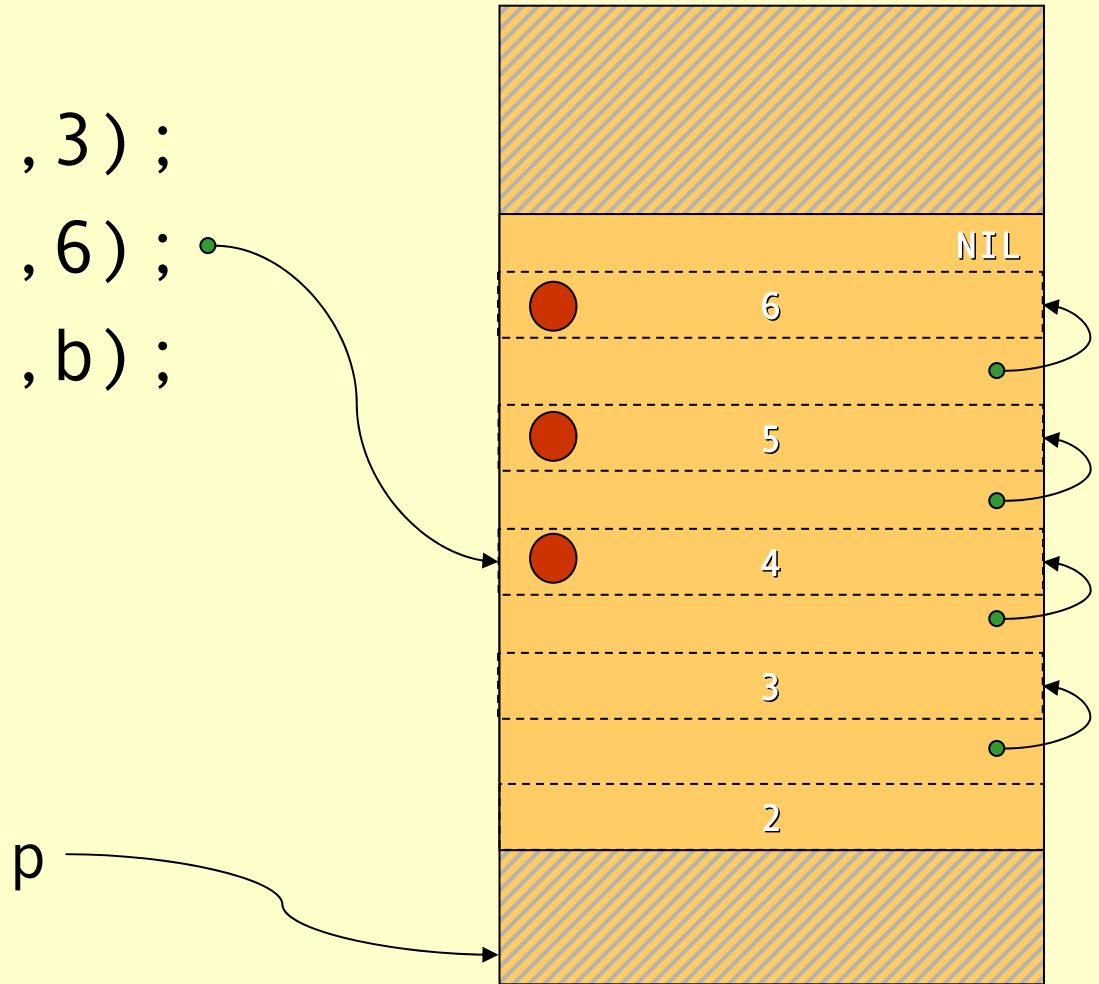
Example: Sweep

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```



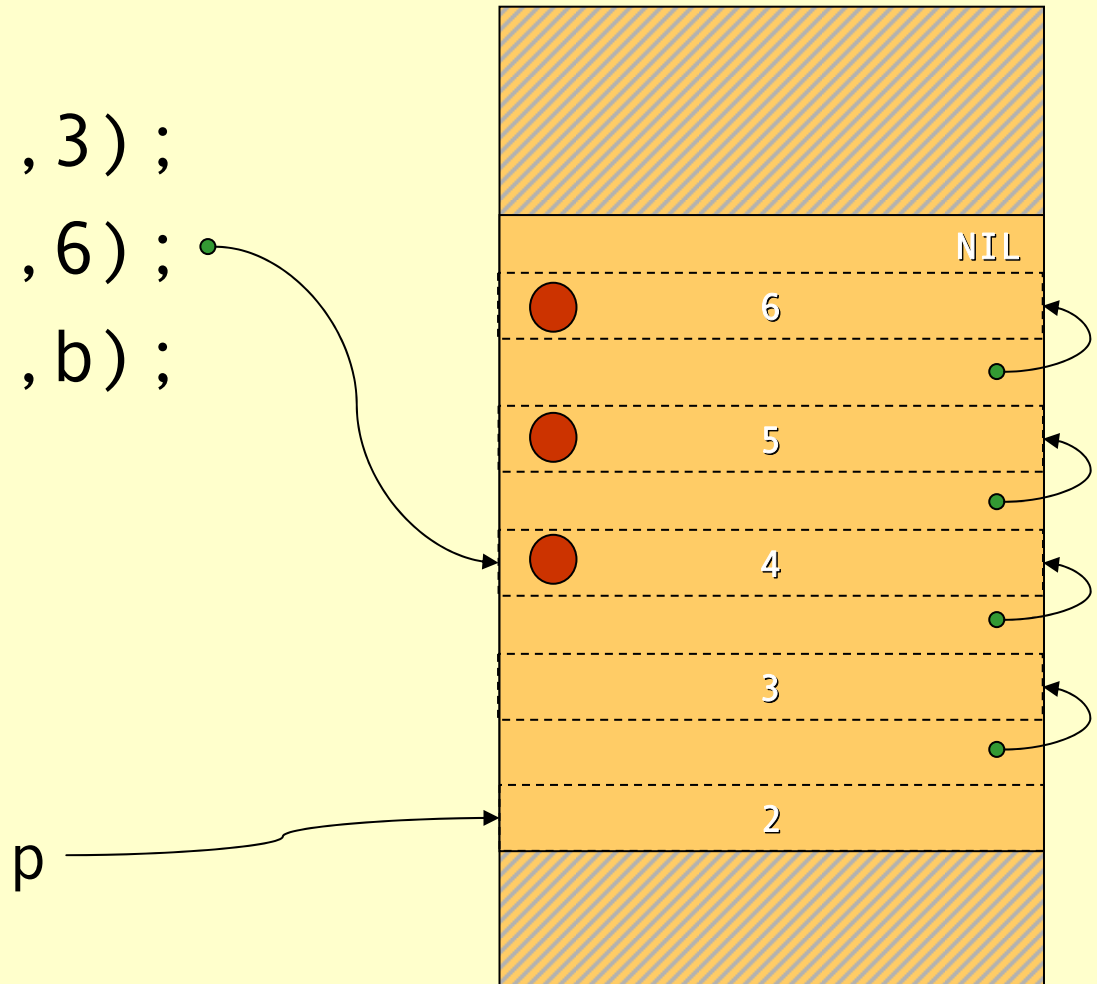
Example: Sweep

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```



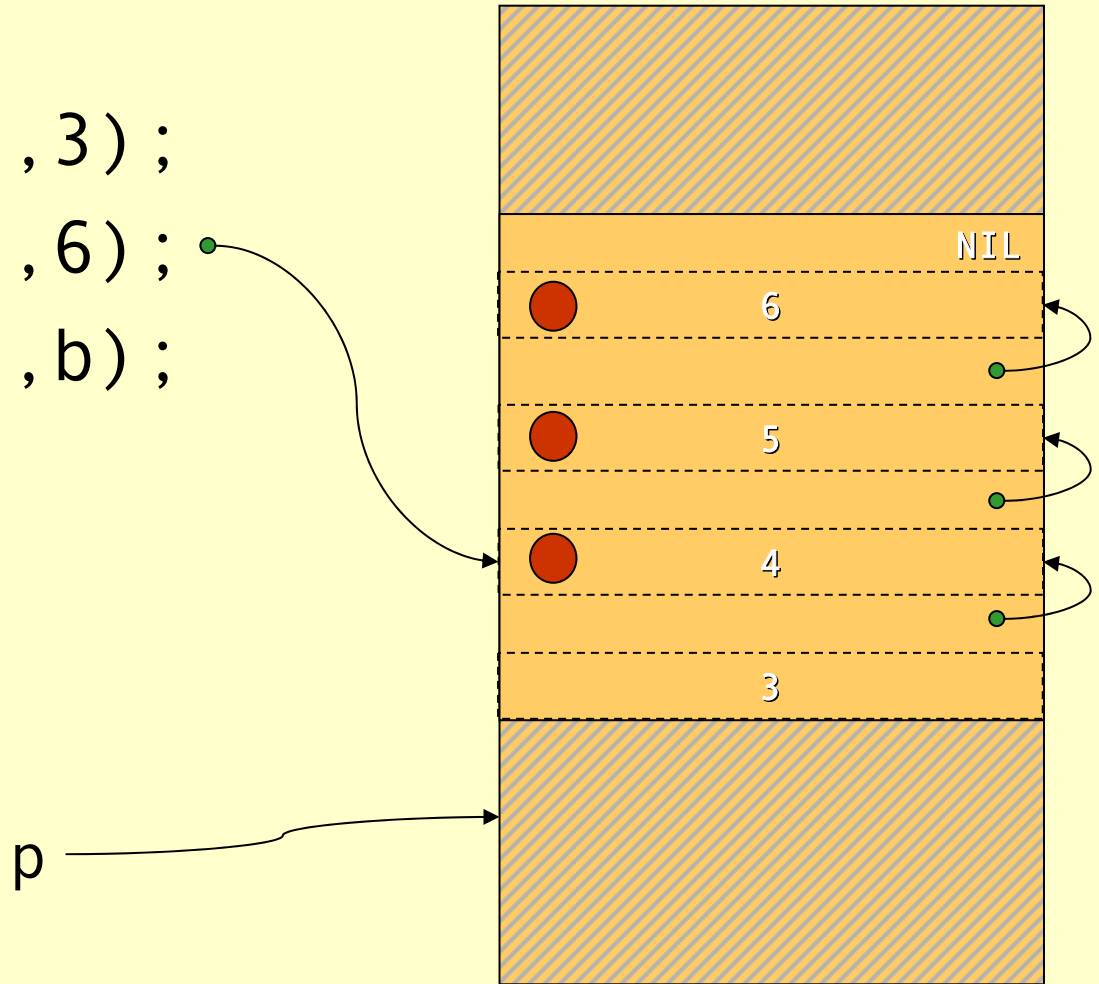
Example: Sweep

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```



Example: Sweep

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```

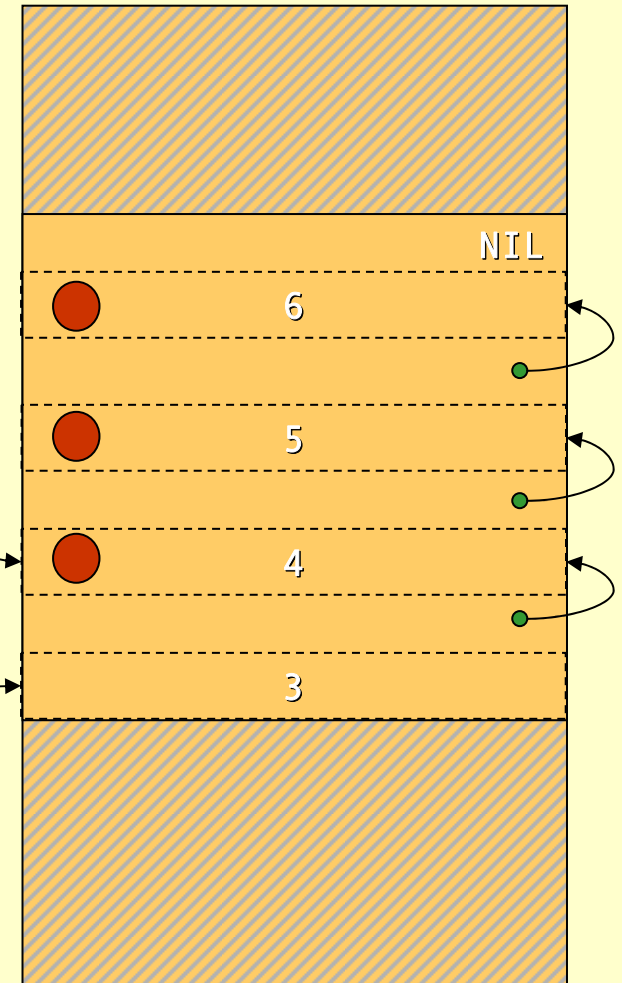


Example: Sweep

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```



p

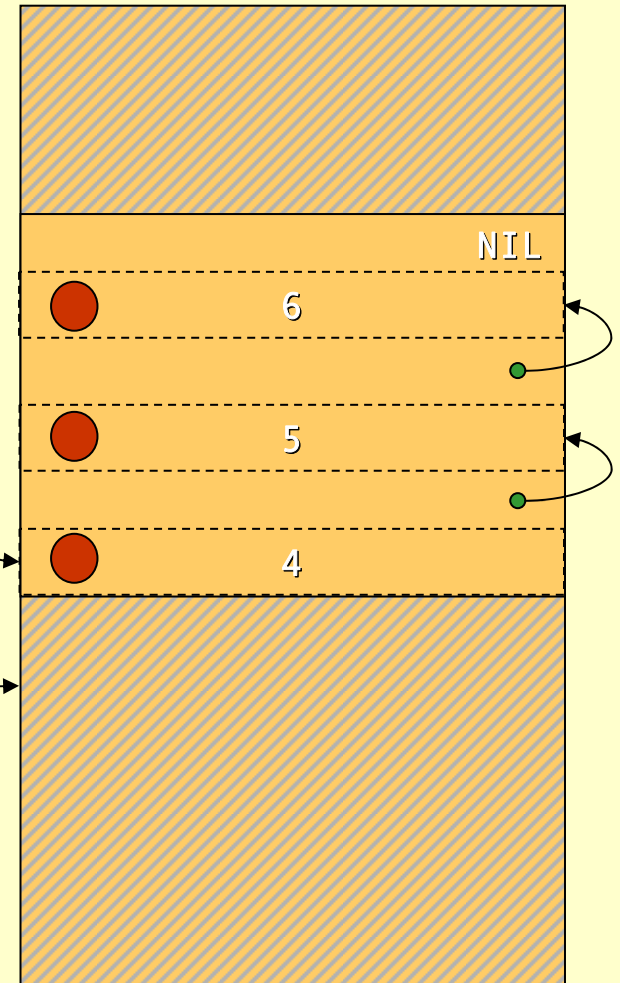


Example: Sweep

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```



p

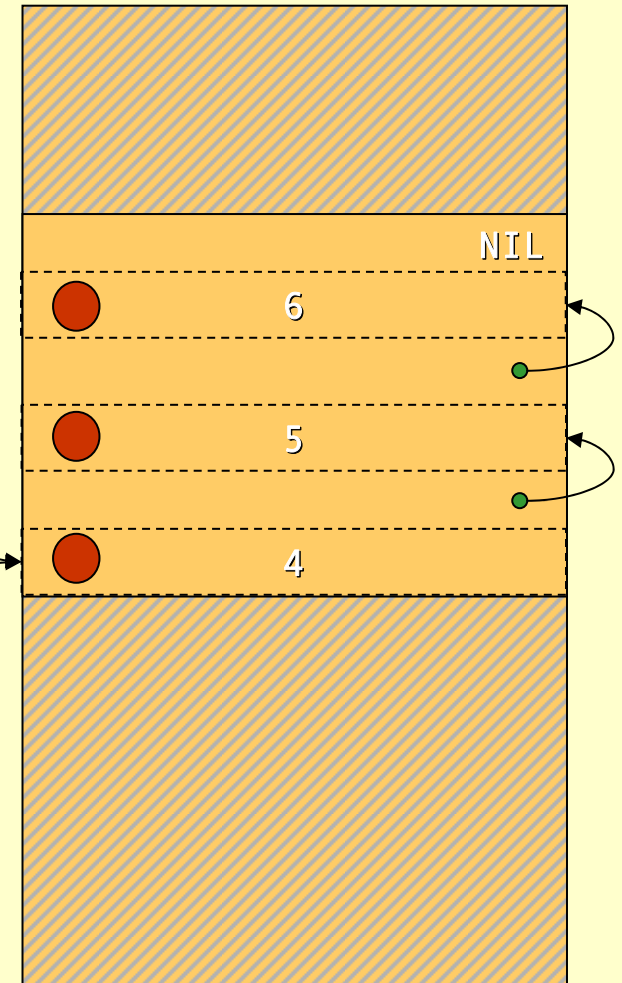


Example: Sweep

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```



p

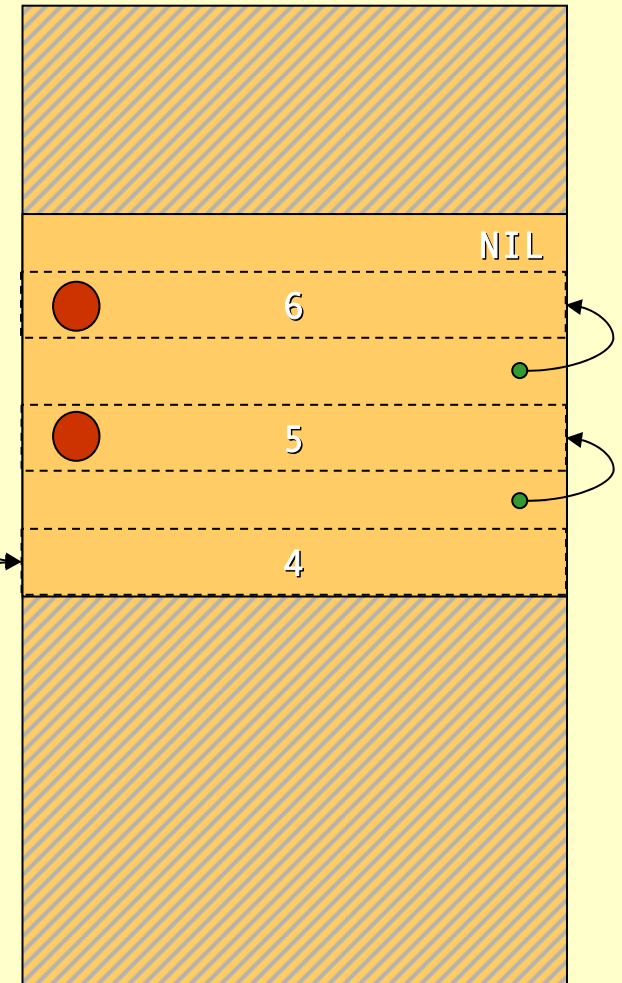


Example: Sweep

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```



p

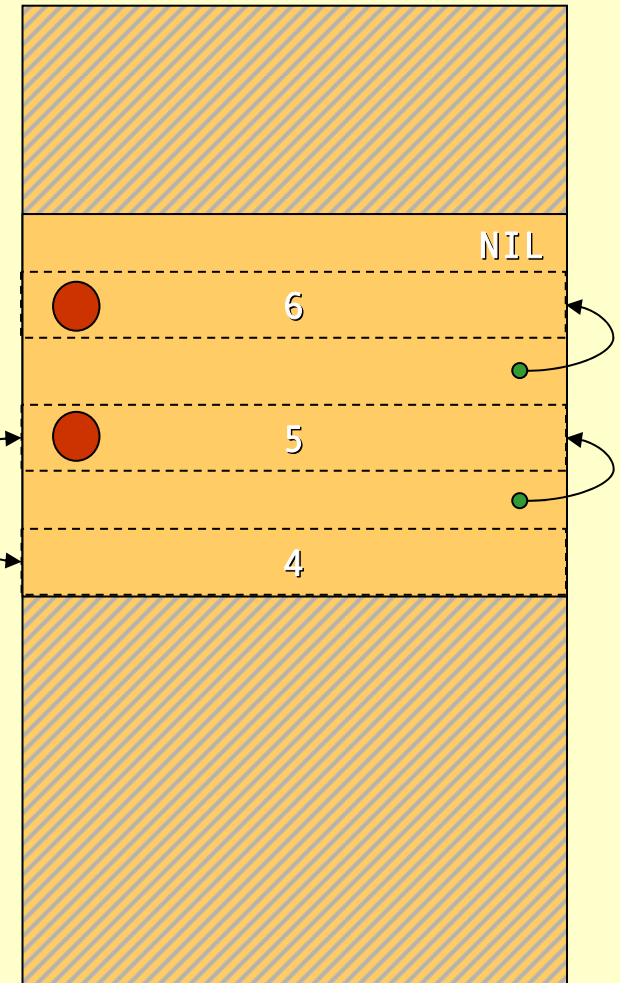


Example: Sweep

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```



p

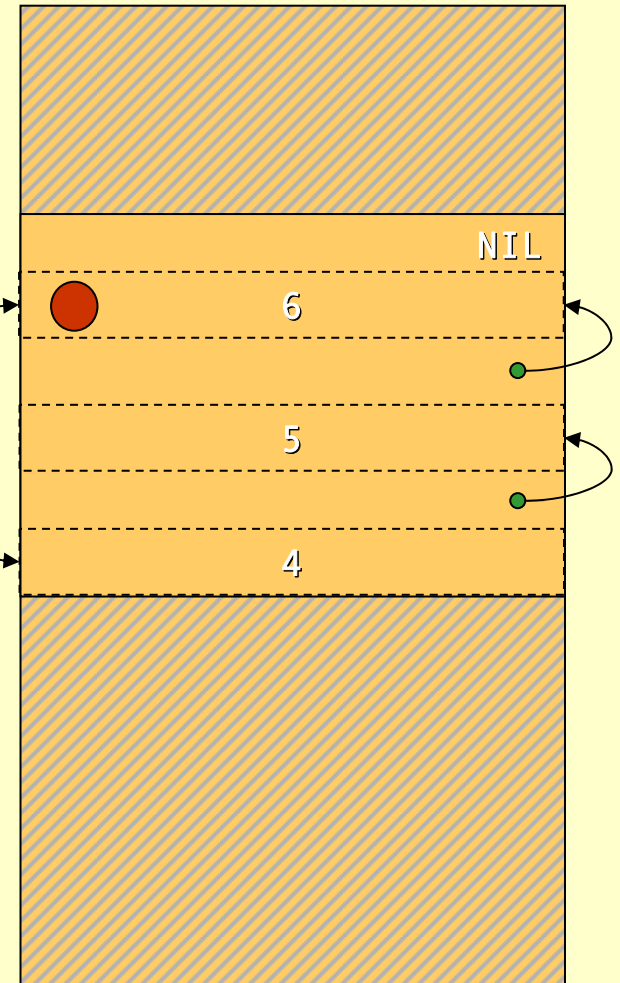


Example: Sweep

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```



p

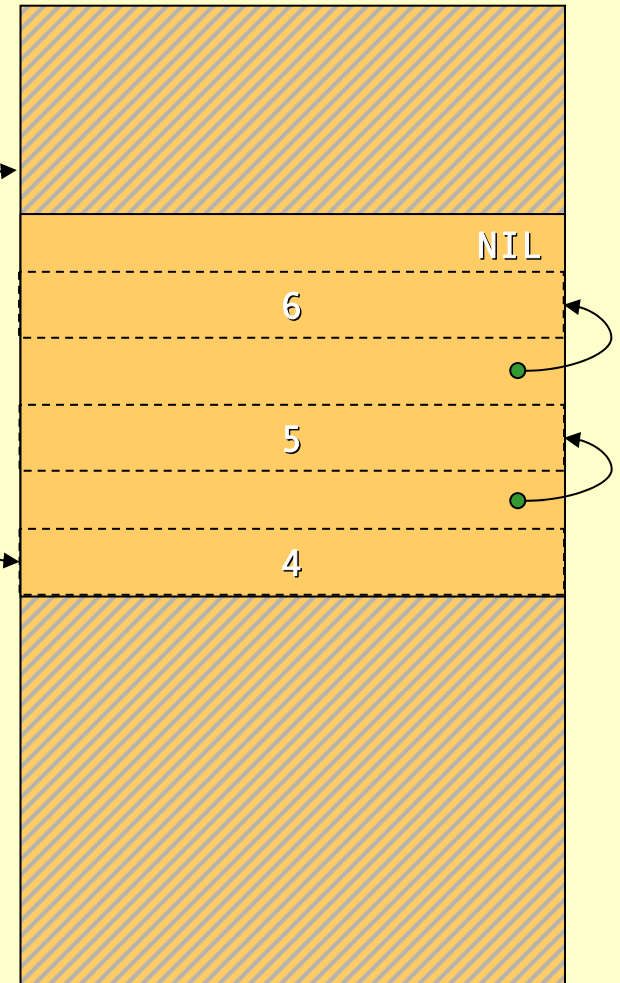


Example: Sweep

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```



p



Cost of Mark & Sweep

- ◆ The mark phase takes time proportional to the amount of reachable data (**R**).
- ◆ The sweep phase takes time proportional to the size of the heap (**H**).
- ◆ The work done by the GC is to recover **H-R** words of memory.
- ◆ Their *amortized cost* of GC (overhead/allocated word) is:

$$\frac{c_1R + c_2H}{H-R}$$

- ◆ If **R** \approx **H** the cost is very high. The cost goes down as the number of dead words increases.

Mark & Sweep

- ◆ Where do we store the mark bits?
 - ◆ We will discuss data representation a bit more at the end of the lecture. With some representations there will always be a tag or a header word in each heap object where the mark bit can be stored.
- ◆ They can be stored in a separate bitmap table:
 - ◆ If we have a **32-bit architecture** and the smallest heap object is **2 words**. (The three least significant bits == 0)
 - ◆ Then we can have **536,870,911** objects and need **67,108,863** bytes to store these bits.
 - ◆ This might seem to be a lot, but it is *only* **1.562%** of the total heap.

Tuning Mark & Sweep

- ◆ There is one problem with the mark phase:
 - ◆ While doing the depth first search we need to keep track of other paths to search.
 - ◆ If this is done with recursive calls we will need one **allocation record for each** level we descend in the reachability graph.
 - ◆ **Solutions:** Explicit stack or pointer reversal.

Mark & Sweep

- ◆ Advantages with mark & sweep:
 - ◆ Can reclaim cyclic structures.
 - ◆ Standard version is easy to implement.
 - ◆ Can have relatively low space overhead.
- ◆ Disadvantages:
 - ◆ Fragmentation can become a problem.
 - ◆ Allocation from a free-list can be costly.

Copying Collector

- ◆ The idea of a copying garbage collector is to divide the memory space in two parts.
- ◆ Allocation is done linearly in one part (*from-space*).
- ◆ When that part is full all reachable objects are copied to the other part (*to-space*).

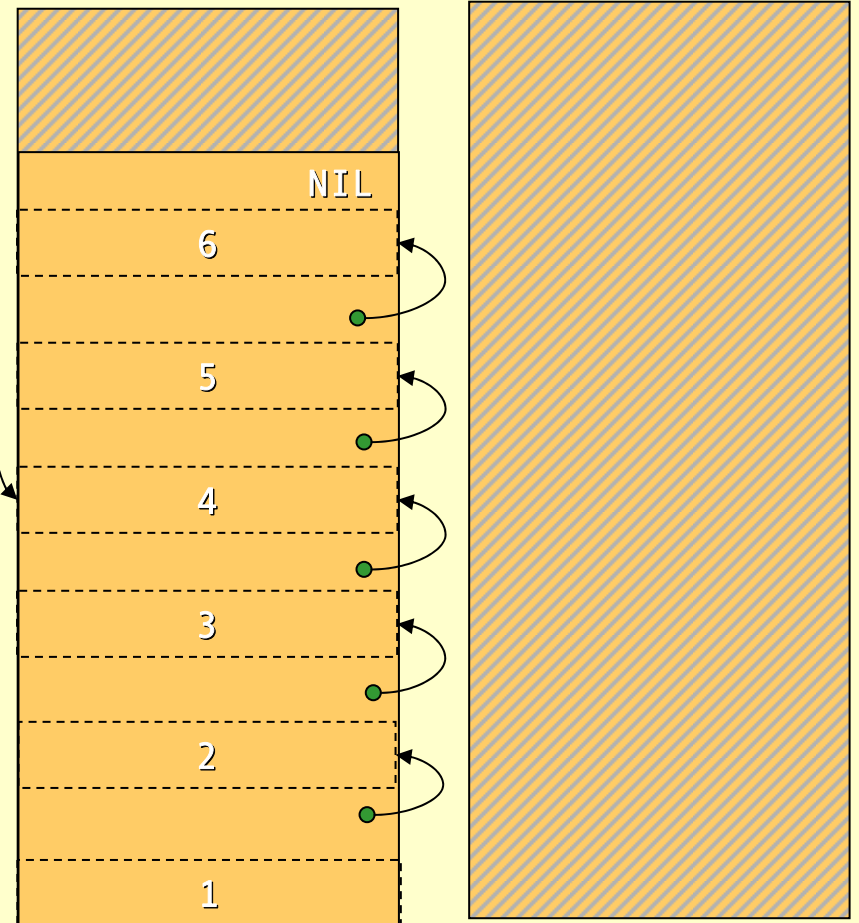
Before GC

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```



from-space

to-space

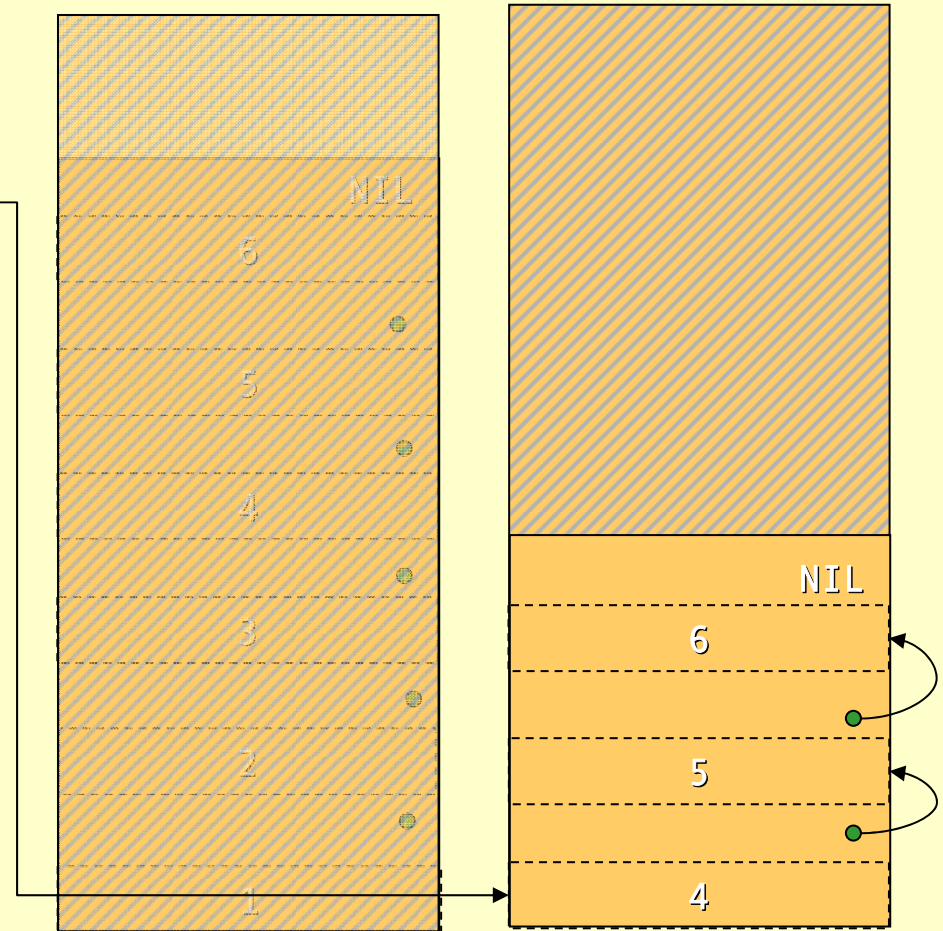


After GC

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```

to-space

from-space



Forwarding Pointers

- ◆ Given a pointer p that point to **from-space** make it point to **to-space**:
 - ◆ If p points to a from-space record that contains a pointer to to-space, then $*p$ is a forwarding-pointer that indicates where the copy is. set $p = *p$.
 - ◆ If $*p$ has not been copied, copy $*p$ to location $next$, $*p = next$, $p = next$, $next += size(*p)$.

Cheney's Copying Collector

- ◆ Cheney's algorithm uses breadth-first to traverse the live data.
- ◆ The algorithm is non-recursive, requires no extra space or time consuming tricks (such as pointer reversal), and it is very simple to implement.
- ◆ The disadvantage is that breadth-first does not give as good locality of references as depth-first.

Cheney's Copying Collector

- ◆ The algorithm:
 1. Forward all roots.
 2. Use the area between `scan` and `next` as a queue for copied records whose children has yet not been forwarded.

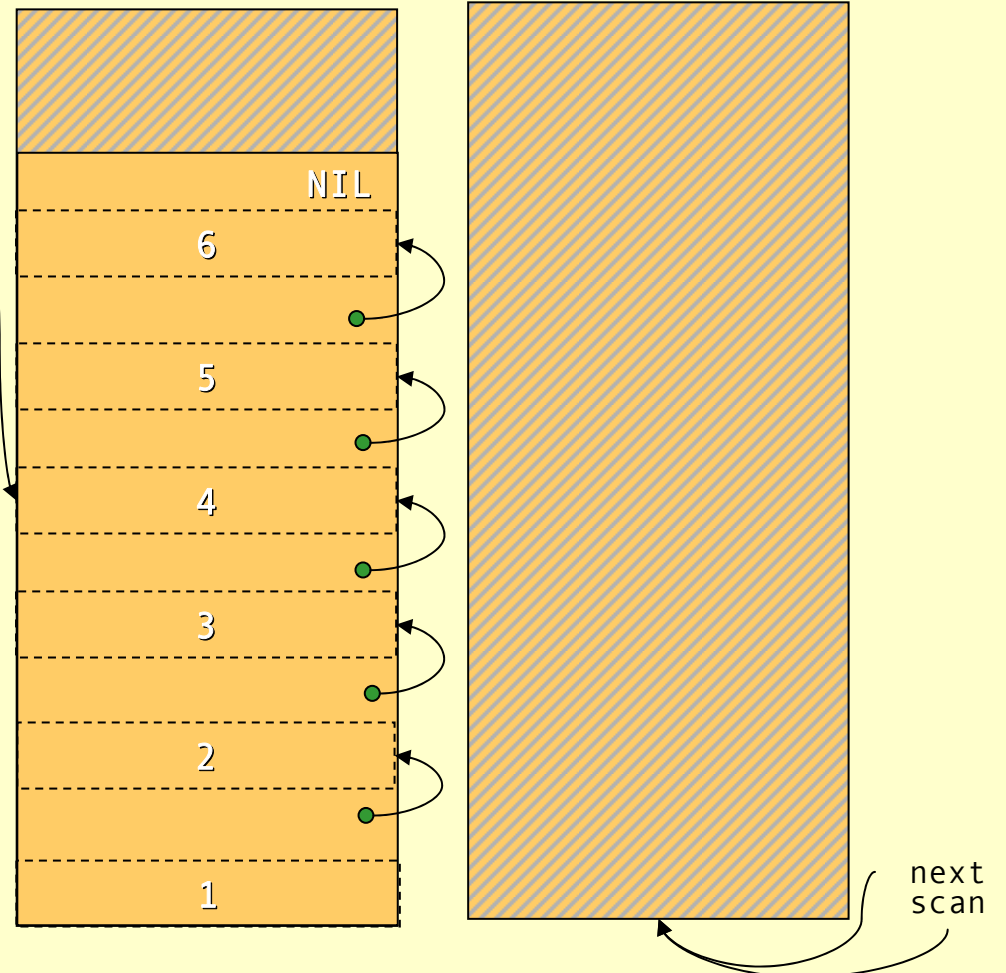
```
scan = next = start of to-space
for each root r { r = forward(r); }
while scan < next {
    for each field f of *scan
        scan->f = forward(scan->f)
    scan += size(*scan)
}
```

Before GC

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```

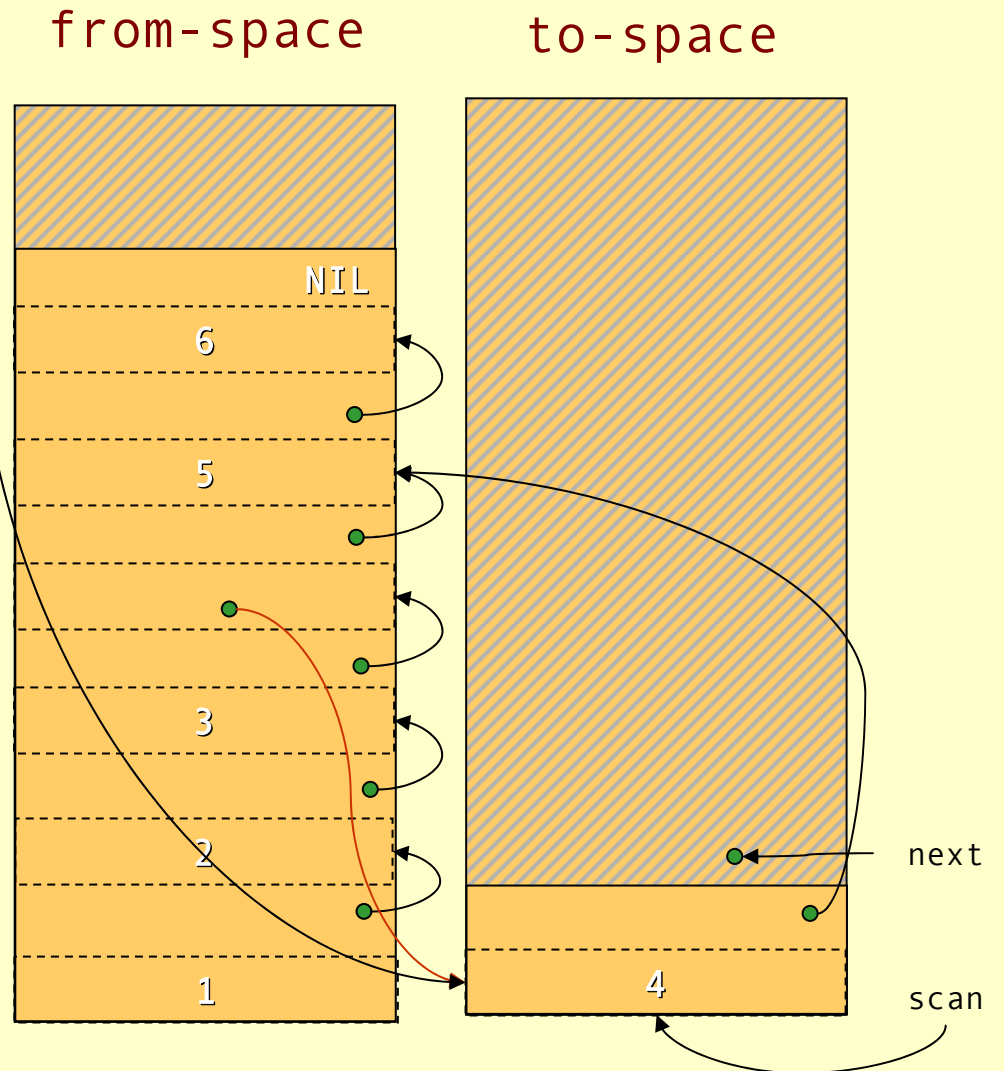
from-space

to-space



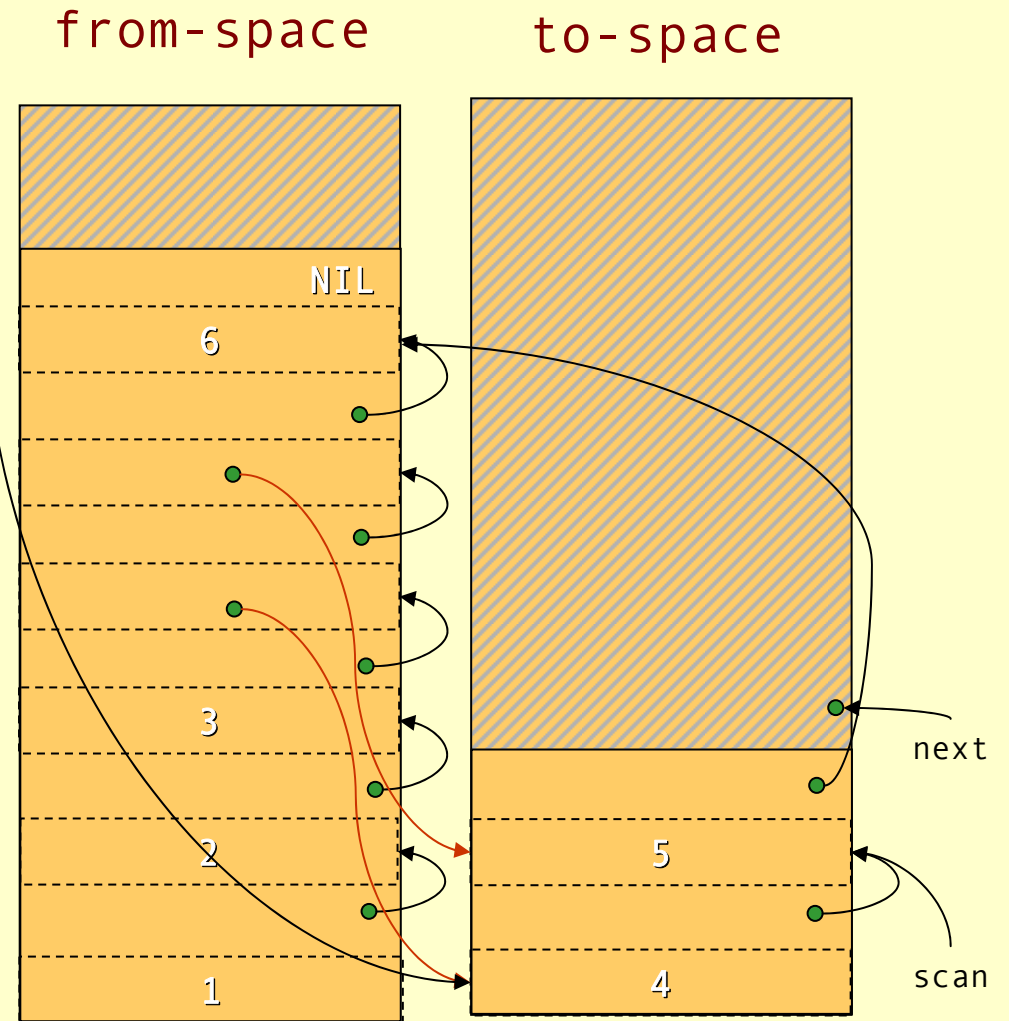
Forward Roots

```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
doLotsOfStuff();
return b;
```



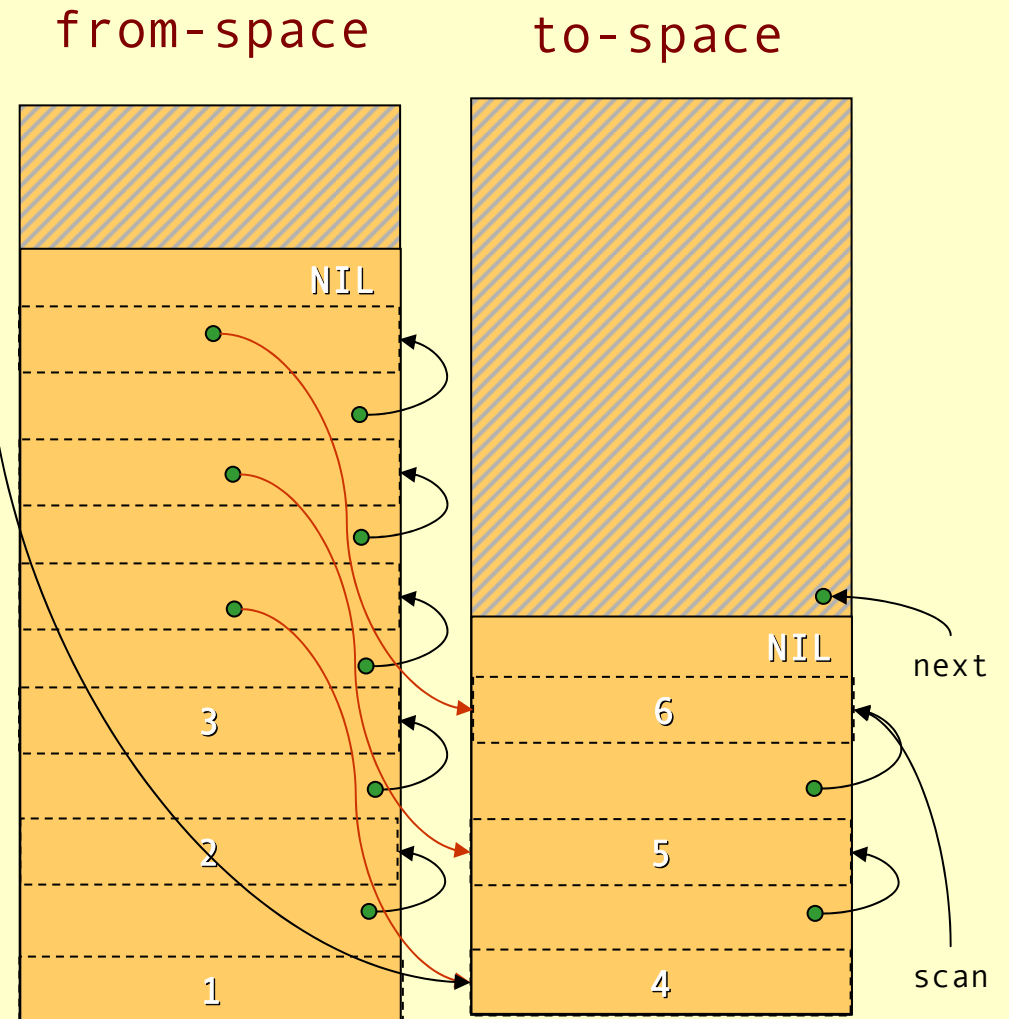
Scanning

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```



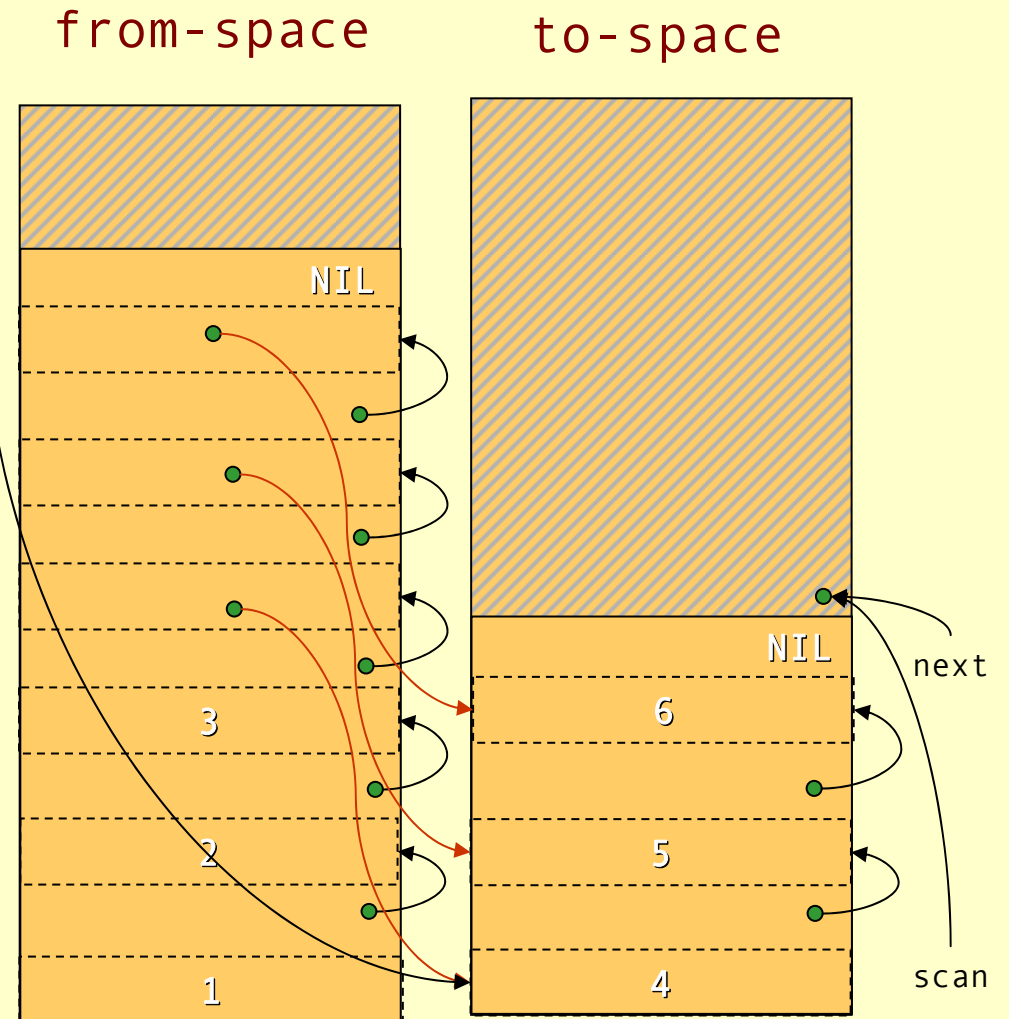
Scanning

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```



Scanning

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```

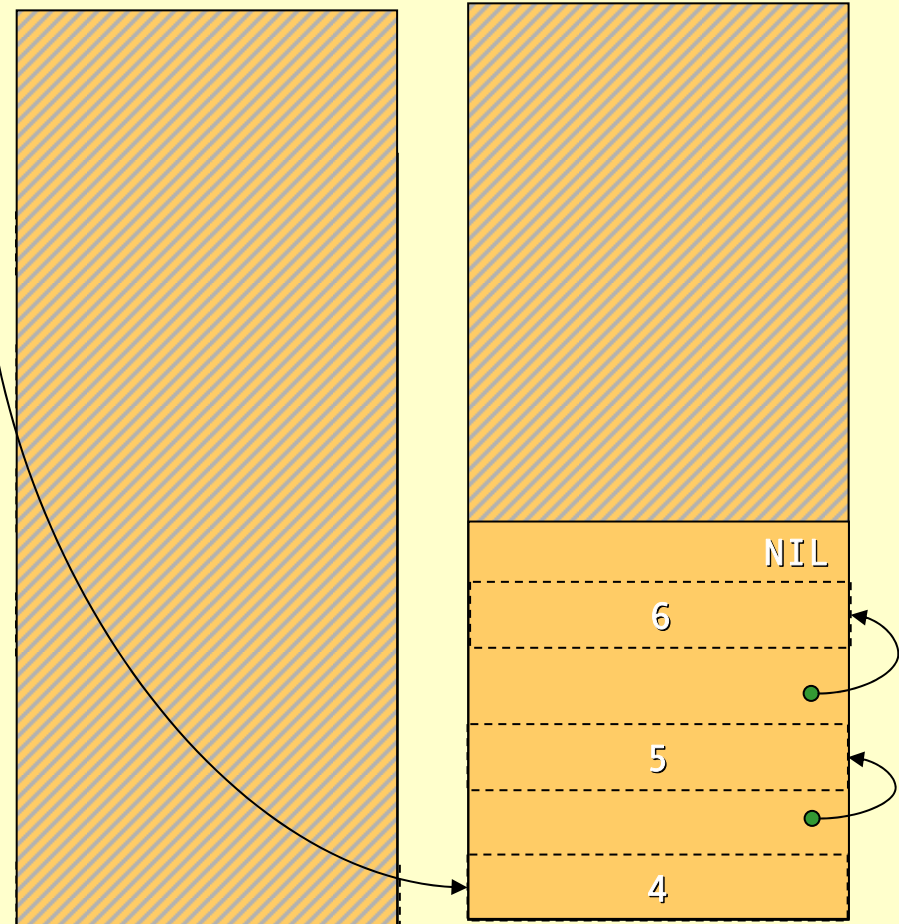


Scanning

```
list a = List(1,2,3);  
list b = List(4,5,6);  
list c = append(a,b);  
printList(c);  
doLotsOfStuff();  
return b;
```

from-space

to-space



Cost of Copying GC

- ◆ The GC takes time proportional to the amount of reachable data (\mathbf{R}).
- ◆ The work done by the GC is to recover $\mathbf{H}/2 - \mathbf{R}$ words of memory.
- ◆ The *amortized cost* of GC (overhead/allocated word) is:

$$\frac{c_1 \mathbf{R}}{(\mathbf{H}/2) - \mathbf{R}}$$

- ◆ If \mathbf{H} is much larger than \mathbf{R} then the cost approaches zero.
- ◆ The GC is often self-tuning so that $\mathbf{H} = 4\mathbf{R}$ giving a GC cost of c_1 per allocated word.

Copying GC

- ◆ Advantages of copying GC:
 - ◆ Can handle cyclic structures.
 - ◆ Very easy to implement.
 - ◆ Extremely fast allocation (no free-list) just a check and heap pointer increment.
 - ◆ Automatic compaction: no fragmentation.
 - ◆ Only visits live data – time only proportional to live data.
- ◆ Disadvantages of copying GC:
 - ◆ Double the space overhead since two heaps are needed.
 - ◆ Long lived live data might be copied several times.
 - ◆ Copying all the live data might lead to long stop times.

Generational GC

- ◆ **Empirical observation:** most objects die young. The longer an object lives the higher the probability it will survive the next GC.
- ◆ The benefit of GC is highest for young objects.
- ◆ **Idea:** Keep young objects in a small space which is GC more often than the whole heap.
- ◆ With such a *generational GC* each collection takes less time and yields proportionally more space.

Generational GC

- ◆ In a generational GC we want to collect the younger generation without having to look at older generations.
- ◆ But we have to consider all pointers from older generations to younger generations as roots.
 - ◆ (In a language without destructive updates this is not a problem, since there are no such pointers.)
- ◆ These inter-generational references must be remembered. The compiler has to ensure that all store operations in an older generation are checked.

Cost of Generational GC

- ◆ It is common for the youngest generation to have **less than 10%** live data.
- ◆ With a copying collector $H/R = 10$ in this generation.
- ◆ The *amortized cost* of a *minor* collection is:

$$\frac{c_1 R}{(10 R) - R}$$

- ◆ Performing a major collection can be very expensive.
- ◆ Maintaining the remembered set also takes time. If a program does many updates of old objects with pointers to new objects a generational GC can be more expensive than a non-generational GC.

Incremental GC

- ◆ An *incremental* (or *concurrent*) GC keeps the stop-times down by interleaving GC with program execution.
 - ◆ The *collector* tries to free memory while the program, called the *mutator* changes the reachability graph.
- ◆ An incremental GC only operates at request from the mutator.
- ◆ A concurrent GC can operate in between any two mutator instructions.

Data Layout

- ◆ The compiler and the runtime system has to agree on a *data layout*. The GC needs to know the size of records, and which fields of a record contains pointers to other records.
- ◆ In statically typed or OO languages, each record can start with a *header word* that points to a description of the type or class.
- ◆ In many functional languages the set of data types can not be extended; for such languages one can use a *tagging scheme* where unused bits in a pointer indicate what data type it points to.
- ◆ Another approach is to not give any information to the collector about which fields are pointers. The collector must then make a *conservative guess*, and treat all words that looks like pointers to the heap as such. Since it is unsafe to change such pointers a *conservative collector* has to be non-moving.

The Root Set

- ◆ The set of registers and stack slots that contain live data can be described by a *pointer map* (*stack map*).
- ◆ For each pointer that is live after a function call the pointer map identifies its register or stack slot.
- ◆ The *return address* can be used as a key in a hash map to find the pointer map.
- ◆ To mark/forward the roots the GC starts at the top of the stack and scans downwards frame by frame. (In a generational collector the stack scan can also be made generational.)

Finalizers

- ◆ Some languages (notably OO) has *finalizers*, that is, some code that should be executed before some data is deallocated.
- ◆ This is, e.g., useful to make sure that an object frees all resources (open files, locks, etc) before dying.
- ◆ With a **copying collector** the handling of finalizers becomes more difficult. Such a GC does not normally visit the dead data. So all finalizers has to be remembered and after GC a check has to be done to see if any freed data triggers a finalizer.
- ◆ A **mark & sweep** collector does not have this problem, but just as with a copying collector it might take a long time after the last use before garbage is actually collected.
- ◆ If one wants to ensure that a finalizer is executed as soon as the object dies then one has to use **reference counting**.

Summary

- ◆ Manual allocation is unsafe and should not be used. (It also comes at a cost, maintaining a free-list is not for free.)
- ◆ Garbage collection solves the problem of automatic memory management.
- ◆ In most cases a generational copying collector will be the most efficient solution.

Virtual Machines & Interpretation Techniques

Advanced Compiler Techniques 2004
Erik Stenman

Partially based on slides from
Kostis Sagonas (<http://user.it.uu.se/~kostis/Teaching/KT2-04/>) and
Antero Taivalsaari (<http://www.cs.tut.fi/~taivalsa/kurssit/VMDesign2003.html>)

Virtual Machines

- ◆ A virtual machine is an abstract computing architecture independent of any hardware.
- ◆ They are software machines that run on top of real hardware, providing an abstraction layer for language implementers.
 - ◆ There are other types of virtual machines intended to emulate some real hardware (e.g., VirtuTech-Simics, VMware, Transmeta), but they are not the focus of this course.

Characteristics of a VM

- ◆ A VM has its own instruction set independent of the host system.
- ◆ A VM usually has its own memory manager and can also provide its own concurrency primitives.
- ◆ Access to the host OS is usually limited and controlled by the VM.

Advantages of VMs

- ◆ A VM bridges the gap between the high level language and the low level aspects of a real machine.
- ◆ It is relatively easy to implement a VM, and it is easier to compile to a VM than to a real machine.
- ◆ A VM can be modified when experimenting with new languages.
- ◆ Portability is enhanced.
- ◆ Support for dynamic (down-)loading of software.
- ◆ VM code is usually smaller than real machine code.
- ◆ Safety features can be verified by the VM.
- ◆ Profiling and debugging are easy to implement.

Disadvantages of VMs

- ◆ Lower performance than with a native code compiler.
 - ◆ Overhead of interpretation.
 - ◆ Modern hardware is not designed for running interpreters.

Some VM History

- ◆ VMs have been built and studied since the late 1950s.
- ◆ The first Lisp implementations (1958) used virtual machines with garbage collection, sandboxing, reflection, and an interactive shell.
- ◆ Forth (early 70s) uses a very small and easy to implement VM with high level of reflection.
- ◆ Smalltalk (early 70s) is a very dynamic language where everything can be changed on the fly, the first truly interactive OO system.
- ◆ USCD Pascal (late 70s) popularized the idea of using pseudocode to improve portability.
- ◆ Self (late 80s) a prototype-based Smalltalk flavor with an implementation that pushed the limits of VM technology.
- ◆ Java (early 90s) made VMs popular and well known.

VM Design Choices

- ◆ When designing a VM one has some design choices similar to the choices when designing intermediate code for a compiler:
 - ◆ Should the machine be used on several different physical architectures and operating systems? (JVM)
 - ◆ Should the machine be used for several different source languages? (CLI/CLR (.NET))
- ◆ Some design choices are similar to those of the compiler backend:
 - ◆ Is performance more important than portability?
 - ◆ Is reliability more important than performance?
 - ◆ Is (smaller) size more important than performance?
- ◆ And some design choices are similar to when designing an OS:
 - ◆ How to implement memory management, concurrency, IO...
 - ◆ Is low memory consumption, scalability, or security more important than performance?

VM Components

- ◆ The components of a VM vary depending on several factors:
 - ◆ Is the language (environment) interactive?
 - ◆ Does the language support reflection and or dynamic loading?
 - ◆ Is performance paramount?
 - ◆ Is concurrency support required?
 - ◆ Is sandboxing required?
- ◆ In this lecture we will only talk about the interpreter of the VM.

VM Implementation

- ◆ Virtual machines are usually written in “portable” (in the sense that compilers for most architectures already exists) programming languages such as C or C++.
- ◆ For performance critical components assembly language can be used.
- ◆ Some VMs (Lisp, Forth, Smalltalk) are largely written in the language itself.
- ◆ Many VMs are written specifically for gcc, for reasons that will become clear in later slides.

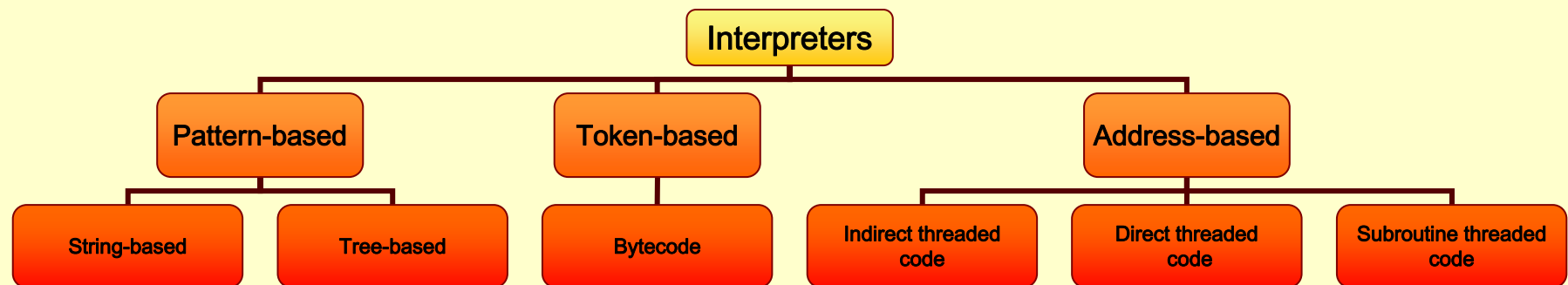
Interpreters

- ◆ Language runtime systems often uses two kinds of interpreters:
 1. Command-line interpreter.
 - ◆ Reads and parses instructions in source form.
 - ◆ Used in interactive systems.
 2. Instruction interpreter.
 - ◆ Reads and executes instructions in some intermediate form such as bytecode.

Implementing Interpreters

- ◆ There are several ways to implement an interpreter.
 - ◆ Pattern (or string) based interpretation.
 - ◆ Interpreting source code (strings) directly is inefficient since most of the time is spent in lexical analysis.
 - ◆ A better alternative is to compile the source into e.g., an abstract syntax tree and then do the interpretation over that tree. (Jumps and calls are expensive.)
 - ◆ Token-based interpretation.
 - ◆ Compiling the code into a linear representation of instructions, where each instruction is represented by a token, e.g., bytecode.
 - ◆ Address-based interpretation.
 - ◆ Compiling the code into a linear representation where each instruction is represented by the address that implements the instruction.
 - ◆ There are several variants: Indirect threaded code, direct threaded code and subroutine threading.

Taxonomy of Interpreters



Implementing Interpreters

- ◆ We will now look at some details of how to implement an interpreter.
- ◆ We will start with a complete but simple string based interpreter for a very simple language. Then extend the language and the interpreter to show the different ways to implement interpreters.

Interpreting while Parsing (String-based Interpretation)

- ◆ For some really simple languages the interpretation can be done during parsing.
- ◆ We can e.g., implement a simple calculator directly in a parser generator.
- ◆ A parser generator is a program that takes a description of a grammar and generates a program that can parse the grammar.
- ◆ We will use CUP a parser generator for Java:
 - ◆ <http://www.cs.princeton.edu/~appel/modern/java/CUP/>
 - ◆ I will not go into the details of CUP.

A Calculator Language

◆ Grammar:

```
Expr ::= Expr MINUS Term
      | Expr PLUS Term
      | Term
```

```
Term ::= Term TIMES Factor
      | Term DIV Factor
      | Factor
```

```
Factor ::= NUMBER | LPAR Expr RPAR
```

Simple Interpreter .cup

```
terminal PLUS, MINUS, TIMES, DIV, LPAR, RPAR;  
terminal Integer NUMBER;  
  
non terminal Program;  
non terminal Integer Expression, Term, Factor;  
  
precedence left PLUS, MINUS;  
precedence left TIMES, DIV;  
  
start with Program;
```

Interpreter .cup

```
Program ::= Expression:e
        { : System.out.println(e.intValue()); : }
        ;
Expression ::= Expression:e PLUS Term:t
            { : RESULT = new Integer(e.intValue() +
                                     t.intValue()); : }
            | Expression:e MINUS Term:t
            { : RESULT = new Integer(e.intValue() -
                                     t.intValue()); : }
            | Term:t
            { : RESULT = t; : }
```

Interpreter .cup

```
Term ::= Term:t TIMES Factor:f
      { : RESULT = new Integer(t.intValue() *
                               f.intValue()); : }
      | Term:t DIV Factor:f
      { : RESULT = new Integer(t.intValue() /
                               f.intValue()); : }
      | Factor:f
      { : RESULT = f; : }
Factor ::= NUMLIT:n { : RESULT = n; : }
        | LPAR Expression:e RPAR
        { : RESULT = e; : }
```

Control Flow

- ◆ This approach works fine for simple expressions.
- ◆ Control flow constructs such as ‘if’ and ‘while’ are harder to handle.
- ◆ For ‘while’ we would need to “reparse” the statement that is to be repeated.
- ◆ Let us extend the language with control flow, variables, and boolean values.

Tree-based (pattern-based) Interpretation

- ◆ By representing the code by a data structure we can “reexecute” the same piece of code several times.
- ◆ This will lead to a slightly more complicated interpreter, which will require at least two passes over the code.
- ◆ The code will first be parsed and stored in the internal representation, then the interpretation will be performed.
- ◆ We can use an abstract syntax tree for representing the code.

Design choices

- ◆ How is the program represented?
 - ◆ As an Abstract Syntax Tree (AST) with the class `Tree`.
- ◆ How is data represented?
 - ◆ We have different types of values, integers and Booleans.
 - ◆ The value of each expression is either an `IntValue` or a `BoolValue`, subclasses of `Value`.
- ◆ How are variables represented?
 - ◆ With a symbol table where each symbol can have a value.

The Implementation

- ◆ The Interpreter itself can be implemented by a Visitor on the AST.
- ◆ We need a Value class:

```
class Value {  
    static class IntValue extends Value {  
        int i;  
        public IntValue(int i) { this.i = i; }  
    }  
    static class BoolValue extends Value {  
        boolean b;  
        public BoolValue(boolean b) { this.b = b; }  
    }  
}
```

Interpreting Expressions

```
public void caseOp(Op tree) {
    switch (tree.op) {
    case TRUE:
        result = new BoolVal(true);
        break;
    case FALSE:
        result = new BoolVal(false);
        break;
    case PLUS:
        IntValue lval = (IntValue) interpret(tree.left);
        IntValue rval = (IntValue) interpret(tree.right);
        result = new IntValue(lval.i + rval.i);
        break;
    ...
}
```

Semantic Analysis Needed

- ◆ This assumes that types are correct.
 - ◆ We could either have a prepass that does the type analysis.
 - ◆ Or we could do the type checking at the same time as interpreting.

Analyzing While Interpreting

```
public void caseOp(Op tree) {
    switch (tree.op) {
    case PLUS:
        Value lval = interpret(tree.left);
        Value rval = interpret(tree.right);
        if ((lval instanceof IntValue) &&
            (rval instanceof IntValue)) {
            result = new IntValue(
                ((IntValue)lval).i +
                ((IntValue)rval).i);
        } else error();
        break;
    ...
}
```

Control Flow

- ◆ Now we can try to interpret a control flow construct.
- ◆ It turns out to be very easy, since we are writing our interpreter in Java which supports the same control flow constructs.
- ◆ It becomes a bit complicated if the type analysis has to be done at the same time.

While (assuming type analysis)

```
public void caseWhile(While tree) {  
    while(((BoolValue)  
        interpret(tree.cond)).b) {  
        interpret(tree.body);  
    }  
}
```

Interpreting While, While Analyzing

```
public void caseWhile(While tree) {  
    Value cond=interpret(tree.cond);  
    while((cond instanceof BoolValue)  
        && ((BoolValue) cond).b) {  
        interpret(tree.body);  
        cond=interpret(tree.cond);  
    }  
}
```


Variables

- ◆ We need to keep track of the values of variables somehow. A simple solution is to store these values with the symbols in the symbol table.
- ◆ If we interpret an assignment we store the value in the symbol.
- ◆ If we interpret an identifier we read the value from the symbol.

Functions

- ◆ These techniques can handle simple languages without functions or more than one scope.
- ◆ In order to handle functions and especially recursive functions and local scopes we will need an *environment*.

Environments

- ◆ In an *environment* we store all values of parameters (arguments) and local variables of a function for one specific call.
- ◆ We create a new environment when we call a function or enter a local scope.
 - ◆ We store actual arguments of the call in the environment.
 - ◆ We initialize local variables.
 - ◆ After returning from a function, or leaving the local scope, the environment is not needed any more.
- ◆ The environment can be implemented as an array of values, the position in the array of an identifier can be stored in the symbol table.

```
class Environment {  
    Environment outer; // For nested scope.  
    Value[] values;  
}
```
- ◆ An environment is similar to how scopes are handled in the compiler.
- ◆ When compiling to native code the environment is stored on the stack as activation records.

Function Calls

```
void caseFunCall {
    // call interpreter recursively on
    //   function arguments;
    Arguments args = interpret_args(tree.args);

    // Create a new Environment
    currentEnv = new Environment(currentEnv);

    // Store the arguments in the new environment.
    insert_args(args, currentEnv);

    // Call the interpreter recursively on the
    //   body of the called function, using the new
    //   environment.
    result = interpret(find_code(tree.funName));

    // Restore the environment.
    currentEnv = currentEnv.outer;
}
```

Disadvantages with Tree-based Interpreters

- ◆ The tree representation has to be created somehow each time we want to run the program.
 - ◆ Parsing the source code each time is time consuming.
 - ◆ Storing the whole tree is space consuming.
- ◆ The tree representation uses a lot of space at runtime, which is infeasible for large programs.
- ◆ Using the stack of the host language adds to the space need at runtime.

Token-based Interpreters

- ◆ By compiling the program to a special instruction set of a virtual machine, and by adding tables that maps function names to offsets in the instruction sequence, some of the interpretation overhead can be reduced.
- ◆ Most VM instruction sets uses small integers to represent everything in the instruction stream (opcodes, registers, stack slots, functions, constants, etc.).
- ◆ By implementing the interpreter in C we can gain some speed, it also allows us to do nasty pointer tricks.

Token-based Interpreters

- ◆ The fundamental instruction unit is the *token*.
- ◆ A token is a predefined numeric value that represents a certain instruction.
 - ◆ E.g., BREAK=0, LOADLITERAL = 1, ADD=2.
- ◆ The most common case is *bytecode*:
 - ◆ The token with is 8 bits.
 - ◆ The total instruction set is limited to 256 tokens.

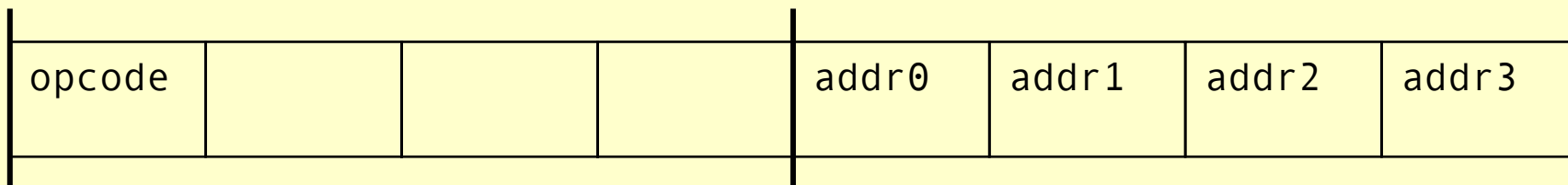
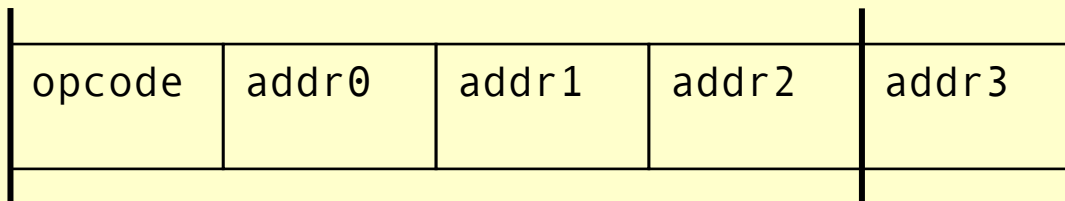
Basic Structure of a Token-based Interpreter

```
byte *pc = &program[0];
while(TRUE) {
    byte opcode = pc[0];
    switch(opcode) {
        ...
        case LOADLITERAL:
            destReg = pc[1];
            value = getTwoBytes(&pc[2]);
            regs[destReg] = value;
            pc += 4;
            break;

        case JUMP:
            jumpAddress = getFourBytes(&pc[1]);
            pc = &program[jumpAddress];
            break;
    }
}
```


Alignment

- ◆ Most modern machines loads data at least one word at the time (usually 4 bytes). By making sure that instructions are aligned on word offsets we get better performance.



Note: The padding is done by the loader, no extra space is needed in the external representation.

Token-based Interpreter with Aligned Instructions

```
byte *pc = &program[0];
while(TRUE) {
    byte opcode = pc[0];
    switch(opcode) {
        ...
        case LOADLITERAL:
            destReg = pc[1];
            value = getTwoBytes(&pc[2]);
            regs[destReg] = value;
            pc += 4;
            break;

        case JUMP:
            jumpAddress = getFourBytes(&pc[4]);
            pc = &program[jumpAddress];
            break;
    }
}
```

Token-based Interpreter with Abstract Encoding

```
byte *pc = &program[0];
while(TRUE) {
    byte opcode = pc[0];
    switch(opcode) {
    ...
    case LOADLITERAL:
        destReg = pc[LOADLITTERAL_ARG1];
        value = getTwoBytes(&pc[LOADLITTERAL_ARG2]);
        regs[destReg] = value;
        pc += LOADLITTERAL_SIZE;
        break;

    case JUMP:
        jumpAddress = getFourBytes(&pc[JUMP_ARG1]);
        pc = &program[jumpAddress];
        break;
    }
}
```

```
#define LOADLITTERAL_SIZE 4
#define JUMP_SIZE 8
#define LOADLITTERAL_ARG1 1
#define LOADLITTERAL_ARG2 2
#define JUMP_ARG1 4
```

Token-based Interpreter with Abstract Control

```
#define NEXT goto loop
```

```
byte *pc = &program[0];  
while(TRUE) {  
loop:  
    byte opcode = pc[0];  
    switch(opcode) {  
    ...  
    case LOADLITERAL:  
        destReg = pc[LOADLITTERAL_ARG1];  
        value = getTwoBytes(&pc[LOADLITTERAL_ARG2]);  
        regs[destReg] = value;  
        pc += LOADLITTERAL_SIZE;  
        NEXT;  
  
    case JUMP:  
        jumpAddress = getFourBytes(&pc[JUMP_ARG1]);  
        pc = &program[jumpAddress]  
        NEXT;  
    }  
}
```

Indirectly Threaded Interpreter

- ◆ In an indirectly threaded interpreter we do not switch on the tokens. Instead we use the tokens as indices into a table containing the addresses of the instruction implementations.
- ◆ The term threaded code refers to a code representation where every instruction is implicitly a function call to the next instruction.
- ◆ A threaded interpreter can be very efficiently implemented in assembler.
- ◆ In GNU C (gcc) we can use *labels as values* and take the address of a label with `&&labelname`.
- ◆ We can actually write the interpreter in such a way that it uses indirectly threaded code if compiled with gcc and a switch for compatibility.

Indirectly Threaded Interpreter

```

byte *pc = &program[0];
while(TRUE) {
loop:
    byte opcode = pc[0];
    switch(opcode) {
    ...
    case LOADLITERAL:
        loadlitteral_label:
            destReg = pc[LOADLITTERAL_ARG1];
            value = getTwoBytes(&pc[LOADLITTERAL_ARG2]);
            regs[destReg] = value;
            pc += LOADLITTERAL_SIZE;
            NEXT;

    case JUMP:
        jump_label:
            jumpAddress = getFourBytes(&pc[JUMP_ARG1]);
            pc = &program[jumpAddress]
            NEXT;
    }
}

```

```

static void *label_tab[] {
    ...
    &&loadlitteral_label;
    &&jump_label;
}
#define NEXT \
    goto ** (void **) (label_tab[*pc])

```

Directly Threaded Interpreter

- ◆ In a directly threaded interpreter we do not use tokens at all during runtime.
- ◆ Instead the loader replaces each token with the address of the implementation of the instruction.
- ◆ This means the opcodes will take one word or four bytes at runtime, slightly increasing the code size.

Directly Threaded Interpreter

```
byte *pc = &program[0];
while(TRUE) {
loop:
    byte opcode = pc[0];
    switch(opcode) {
    ...
    case LOADLITERAL:
        loadlitteral_label:
            destReg = pc[LOADLITTERAL_ARG1];
            value = getTwoBytes(&pc[LOADLITTERAL_ARG2]);
            regs[destReg] = value;
            pc += LOADLITTERAL_SIZE;
            NEXT;

    case JUMP:
        jump_label:
            jumpAddress = getFourBytes(&pc[JUMP_ARG1]);
            pc = &program[jumpAddress]
            NEXT;
    }
}
```

```
static void *label_tab[] {
    ...
    &&loadlitteral_label;
    &&jump_label;
}
#define NEXT \
goto ** (void **) (pc)
```


Subroutine Threaded Interpreter

- ◆ The only portable way to implement a threaded interpreter in C is to use subroutine threaded code.
- ◆ Each instruction is implemented as a function and at the end of each instruction the next function is called.

Subroutine Threaded Interpreter (with tail-calls)

```
byte *pc = &program[0];  
NEXT;
```

...

```
void loadlitteral(void) {  
    destReg = pc[LOADLITTERAL_ARG1];  
    value = getTwoBytes(&pc[LOADLITTERAL_ARG2]);  
    regs[destReg] = value;  
    pc += LOADLITTERAL_SIZE;  
    NEXT;  
}  
void jump(void) {  
    jumpAddress = getFourBytes(&pc[JUMP_ARG1]);  
    pc = &program[jumpAddress];  
    NEXT;  
}
```

```
static void *label_tab[] {  
    ...  
    &loadlitteral;  
    &jump;  
}  
#define NEXT ((void (*)(void))*pc)()
```

Subroutine Threaded Interpreter

```
byte *pc = &program[0];  
while (TRUE) NEXT;
```

```
...  
void loadlitteral(void) {  
    destReg = pc[LOADLITTERAL_ARG1];  
    value = getTwoBytes(&pc[LOADLITTERAL_ARG2]);  
    regs[destReg] = value;  
    pc += LOADLITTERAL_SIZE;  
}  
void jump(void) {  
    jumpAddress = getFourBytes(&pc[JUMP_ARG1]);  
    pc = &program[jumpAddress];  
}
```

```
static void *label_tab[] {  
    ...  
    &loadlitteral;  
    &jump;  
}  
#define NEXT ((void (*)(void))*pc)()
```

Subroutine Threaded Interpreter

```
(void (*)(void)) pc = &program[0];  
while (TRUE) *pc++;
```

...

```
void loadlitteral(void) {  
    destReg = ((int *)pc)[LOADLITTERAL_ARG1];  
    value = getTwoBytes(&pc[LOADLITTERAL_ARG2]);  
    regs[destReg] = value;  
    pc += LOADLITTERAL_SIZE;  
}  
void jump(void) {  
    jumpAddress = getFourBytes(&pc[JUMP_ARG1]);  
    pc = &program[jumpAddress];  
}
```

```
#define LOADLITTERAL_SIZE 1  
#define JUMP_SIZE 1  
#define LOADLITTERAL_ARG1 0  
#define LOADLITTERAL_ARG2 1  
#define JUMP_ARG1 0
```

Stack-based vs. Register-based VM

- ◆ A VM can either be *stack-based* or *register-based*.
 - ◆ In a stack-based machine most operands are on the stack. The stack can grow as needed.
 - ◆ In a register-based machine most operands are in (virtual) registers. The number of registers is limited.
- ◆ Most VMs are stack-based.
 - ◆ Stack machines are simpler to implement.
 - ◆ Stack machines are easier to compile to.
 - ◆ Less encoding/decoding to find the right register.
 - ◆ Virtual registers are no faster than stack slots.

Interpreter Tuning

- ◆ Common interpreter optimizations include:
 - ◆ Writing the interpreter loop and key instructions in assembler.
 - ◆ Keeping important variables in hardware registers (pc, stack-top, heap-top). (GNU C allow global register variables.)
 - ◆ Top of stack caching.
 - ◆ Splitting the most used instruction into a separate interpreter loop.

Interpreter Tuning

- ◆ More advanced interpreter optimizations includes:
 - ◆ Instruction merging: A common sequence of VM instructions is replaced by a single instruction.
 - ◆ Reduced interpretation overhead.
 - ◆ Enhances code locality.
 - ◆ More compact bytecode.
 - ◆ Gives C compiler bigger code block to optimize.
 - ◆ Instruction specialization: A special case VM instruction is created, typically with some arguments hard-coded.
 - ◆ Eliminates argument decoding cost.
 - ◆ More compact bytecode.
 - ◆ Reduces register pressure.

Just-in-time Compilation

- ◆ Native code is still faster than code interpreted in VMs. To get the best performance native code compilation is necessary. But bytecode is a nice format to distribute portable code.
- ◆ Solution: *dynamic compilation* or *just-in-time* (JIT) compilation.
- ◆ Native code takes more space than virtual machine code (4-8x). Don't compile everything to native code (some code is never executed).
- ◆ Compilation takes time, dynamic compilation has to be fast. No time for advanced optimization (unless the bytecode compiler has inserted hints in the bytecode).

JIT – What to Compile

- ◆ Only compile a method if the total execution time is reduced.
- ◆ How do we know this?
- ◆ Use the past to predict the future:
 - ◆ Use profiling to detect what and when to compile. There are two basic approaches:
 - ◆ Invocation counters.
 - ◆ Sample based profiling.

Invocation Counters

- ◆ Associate a counter with each function.
- ◆ When a function is called increment the counter.
- ◆ If the counter reaches a limit compile the function. Reset or use decay to only compile high-frequency functions.
- ◆ Hard to predict behavior, no control over time spent in compiler.

Sample Based Profiling

- ◆ Measure time spent in interpreter, compiler, and in compiled code.
- ◆ Harder to implement.
- ◆ Gives better picture of the hot-spots.

JIT Integration

- ◆ Integrating a JIT system where native code can coexist with interpreted code in the VM is not trivial.
- ◆ Context switches between native and interpreted code has to be fast. (They can occur at function calls, returns, and when exceptions are thrown.)
- ◆ Ensuring proper tail-calls with a mixed execution environment is also tricky.

Summary

- ◆ Virtual machines provides an abstraction from real hardware and make programming language implementation easier and languages more portable.
- ◆ A direct threaded interpreter gives the best performance.
- ◆ Virtual machines have been used for half a century but research didn't really take off until the JVM came along.