

Static Program Analysis

Prof. A. Morzenti

COMPILATION:

FIRST STEP: translates a program into an intermediate representation that is closer to machine language

SECOND STEP: applied to the intermediate code, can have several purposes:

- **VERIFICATION** – further check of program correctness
- **OPTIMIZATION** – transform the program to improve execution efficiency (variable allocation to registers, ...)
- **SCHEDULING** – modify instruction execution sequence to improve exploitation of pipeline and of processor functional units

It is convenient to represent the *program control-flow graph* (on which the previous tasks are based) as an automaton

WARNING: each automaton defines *one* program *not* the entire source language!
The approach is quite different from that used so far

IN THIS CASE: a *string accepted* by the automaton is a possible *execution trace*

STATIC ANALYSIS: the study of certain properties of the control flow graph of a program, using the methods of automata theory, logics, or statistics
We only consider methods based on automata

THE PROGRAM AS AN AUTOMATON

- we consider only the simplest instructions (those of the intermediate representation):
 - simple variables and constants
 - variable assignments
 - simple arithmetic, relational, logic operations
 - assignment and conditional instructions not iterative ones
- we consider only INTRAPROCEDURAL ANALYSIS (not interprocedural)

PROGRAM CONTROL FLOW:

- every node is an instruction
- if instr. p is immediately followed by instr. q then graph has arc $p \rightarrow q$
an arc is also called *program point*
- first program instruction is the initial node
- an instruction with no successor is an exit (final) node
- nonconditional instructions have at most one successor, conditional ones have one or more
- an instruction with many predecessors is a *confluence node*

the control-flow graph is not a completely faithful program representation:

- the TRUE/FALSE value in conditional instructions not represented
- **assignment, read, write**, instructions are substituted by the following abstractions:
 - variable *assignment* and *reading* ... **define** that variable
 - a variable occurrence in the right part of an assignment, in an expression, in a write operation ... **uses** that variable
 - hence in the graph every node (instruction) ***p*** is associated with two sets:

def(p) and ***use(p)***

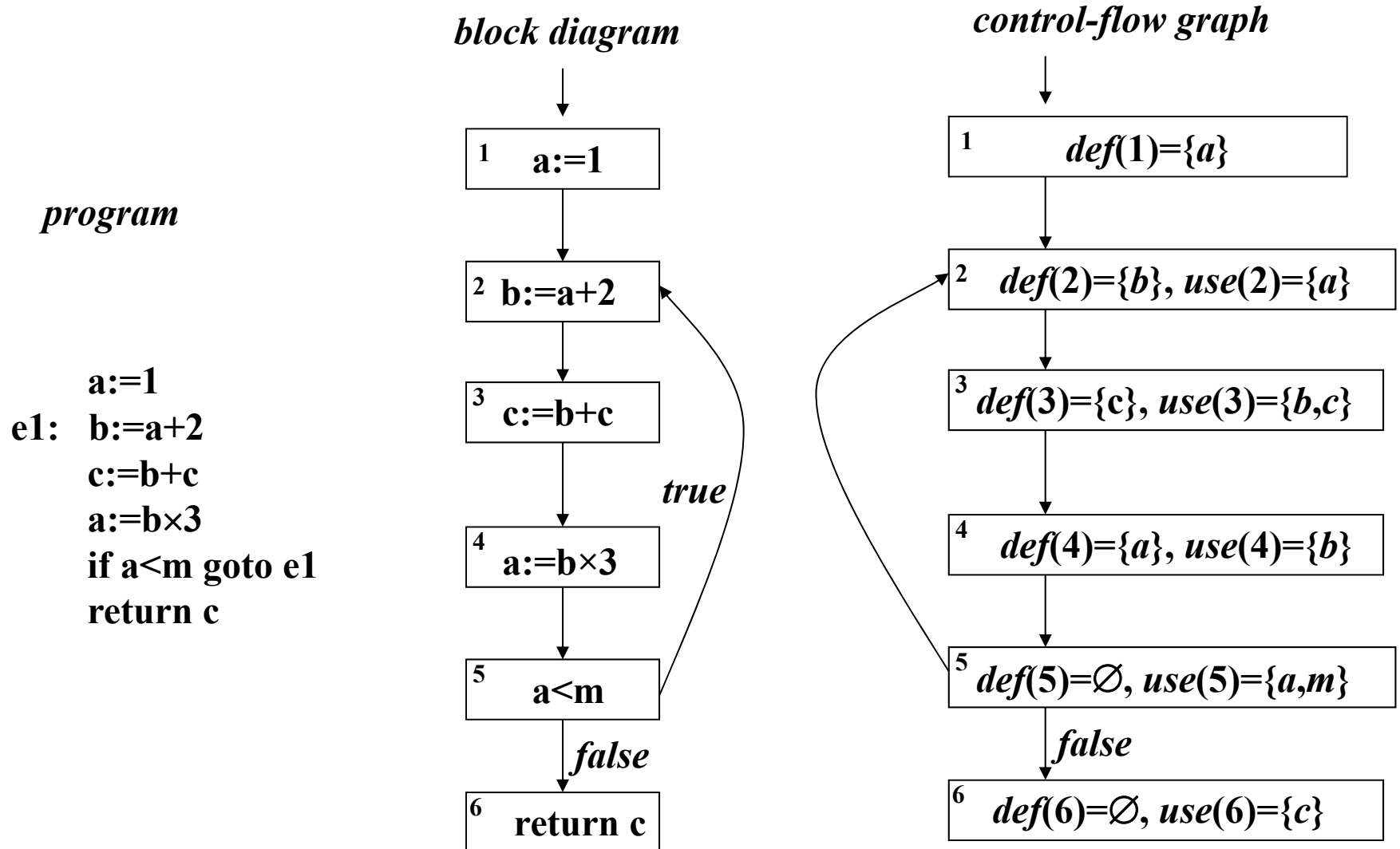
$p: \quad a := a \oplus b$ $def(p) = \{a\}, use(p) = \{a, b\}$
--

$q: \quad read(a) \quad def(q) = \{a\}, use(q) = \emptyset$ $w: \quad a := 7 \quad def(w) = \{a\}, use(w) = \emptyset$
--

The set **def(p)** can include more than one variable in case of read instructions

such as **read(a, b, c)**

Example 1 – block diagram and control-flow graph



DEFINITION: LANGUAGE OF THE CONTROL-FLOW GRAPH

the finite state automaton A corresponding to the control-flow graph has

- ***terminal alphabet***: the set of program instructions I , each represented by the triple

$$\langle n, \text{def}(n) = \{\dots\}, \text{use}(n) = \{\dots\} \rangle$$

- ***initial state*** : the state with no predecessor
- ***final states*** : the nodes with no successor

The language $L(A)$ is the set of strings over alphabet I labeling paths from initial to final states

A path represents an instruction sequence that the machine can execute when the program is launched

Previous example : $I = \{1 \dots 6\}$

An accepted path is $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 = 1234523456$

The set of paths is the language denoted by $1(2345)^+6$

CONSERVATIVE APPROXIMATION

some paths might not be executable: the control-flow graph ignores the value of the boolean conditions which determine the execution of conditional statements

1: if $a**2 \geq 0$ then $istr_2$ else $istr_3$

The accepted language includes the two paths {12,13} but the path 13 cannot be executed

In general, it is **undecidable** if a generic path of the control-flow graph can be executed (the halting problem of the Turing Machine can be reduced to it ...)

conservative approximation: considering all paths from input to output can lead to the diagnosis of nonexistent errors, or to the allocation of unnecessary resources, but it never leads to ignoring actual, existing error conditions

HYPOTHESIS: we assume that the automaton is clean :
every instruction is on a path from the initial node to a final one

otherwise:

- the program execution might never finish
- the program might have instructions that are never executed (*unreachable code*)

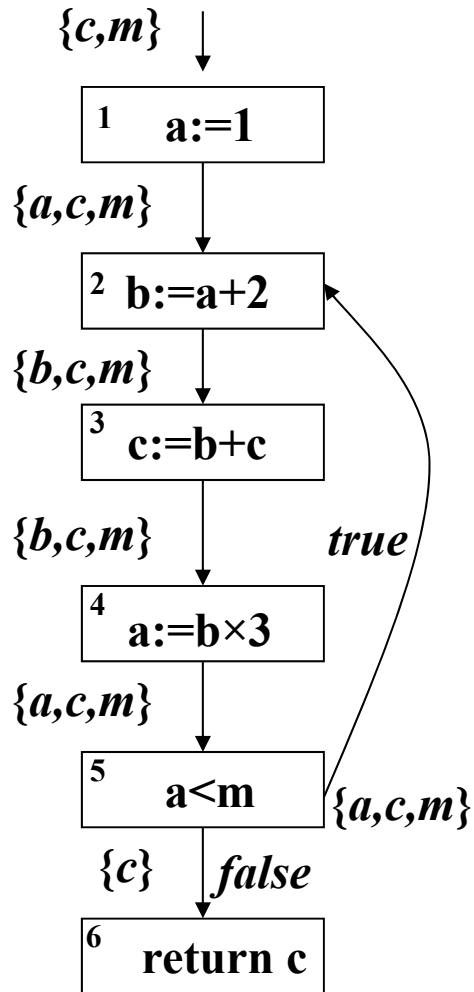
LIVENESS INTERVALS OF VARIABLES

A variable is live at some program node if some instruction that could be executed subsequently uses the value that the variable has at that point (i.e., the variable will be used before being assigned)

DEFINITION: a variable a is *live on an arc*, input or output of a program node p , if the graph admits a path from p to another node q such that

- instruction q uses a , that is, $a \in use(q)$ AND
- the path does not traverse an instruction r , $r \neq q$ that defines a , that is, such that $a \in def(r)$

live variables on arcs



A variable is

- *live-out for a node* if it is live on an outgoing arc of that node
- *live-in for a node* if it is live on an arc entering that node

EXAMPLE:

c is live-in for node 1 because of the path 123: $c \in use(3)$ and neither 1 nor 2 define c

It is customary to define variable liveness on *intervals* (of paths)

a is live in the *intervals* 12 and 452,

a is *not* live in intervals 234 and 56

out of node 5 the live variables are $\{a,c,m\} \cup \{c\}$

METHOD TO COMPUTE SETS OF LIVE VARIABLES: DATA-FLOW EQUATIONS

It computes simultaneously all sets of live variables at every point on the graph

It considers all paths from a given point to some instruction that uses a variable

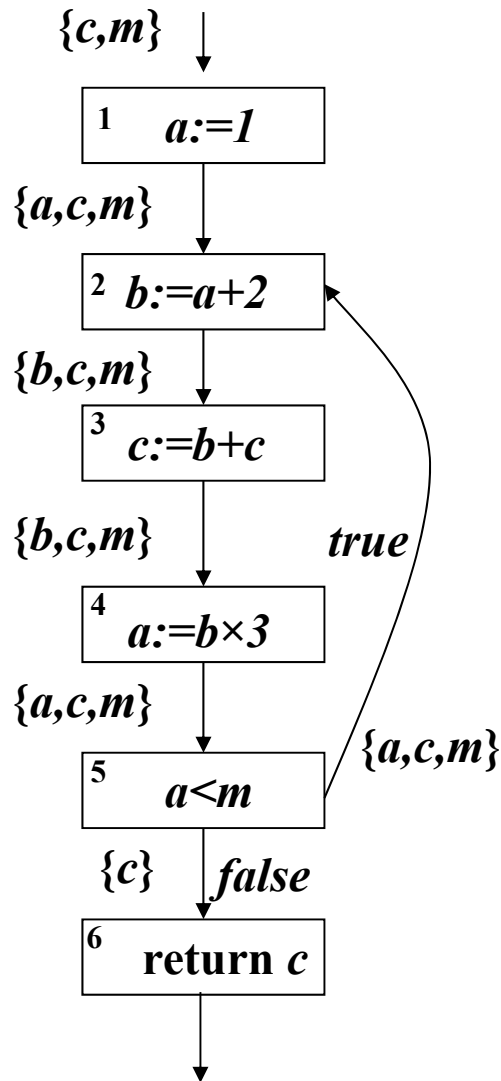
for every final node p :

$$live_{out}(p) = \emptyset$$

for every other node p :

$$live_{in}(p) = use(p) \cup (live_{out}(p) \setminus def(p))$$

$$live_{out}(p) = \bigcup_{\forall q \in succ(p)} live_{in}(q)$$



for every final node p :

$$(1) \quad live_{out}(p) = \emptyset$$

for every other node p :

$$(2) \quad live_{in}(p) = use(p) \cup (live_{out}(p) \setminus def(p))$$

$$(3) \quad live_{out}(p) = \bigcup_{\forall q \in succ(p)} live_{in}(q)$$

From (1): no variable is live at the graph exit

From (2): $live_{in}(4) = \{b, m, c\} = \{b\} \cup (\{a, c, m\} \setminus \{a\})$

From (3): $succ(5) = \{2, 6\}$

$$live_{out}(5) = live_{in}(2) \cup live_{in}(6) = \{a, c, m\} \cup \{c\} = \{a, c, m\}$$

SOLUTION OF DATA-FLOW EQUATIONS

For a graph with $|I| = n$ nodes one gets a system of $2 \cdot n$ equations in $2 \cdot n$ unknowns $live_{in}(p)$ and $live_{out}(p)$, $\forall p \in I$

The system solution is a vector of $2 \cdot n$ sets

The system is solved iteratively

by initially assigning the empty set \emptyset to every unknown (iteration $i = 0$) :

$$\forall p : live_{in}(p) = \emptyset; live_{out}(p) = \emptyset$$

the values for iteration i are inserted into the system and used to compute values of iteration $i+1$

If at least one of them is different from the previous one, continue

Otherwise stop and the values of iteration $i + 1$ are the solution

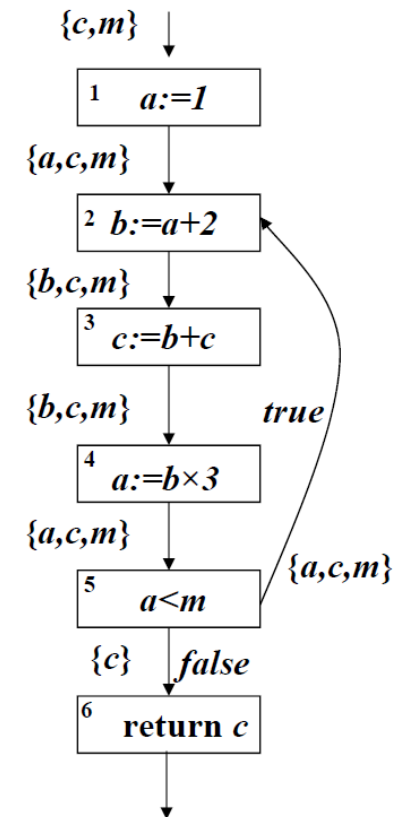
(the usual fixpoint technique...)

The computation converges after a bounded number of iterations:

- 1) $live_{in}(p)$ and $live_{out}(p)$ have a cardinality upperbound determined by the number of program variables
- 2) no iteration will remove elements from sets (these either increase or are unchanged)
- 3) when an iteration does not change any set, the algorithm terminates

Example – Iterative computation of the live variables

1	$in(1) = out(1) \setminus \{a\}$	$out(1) = in(2)$
2	$in(2) = \{a\} \cup (out(2) \setminus \{b\})$	$out(2) = in(3)$
3	$in(3) = \{b, c\} \cup (out(3) \setminus \{c\})$	$out(3) = in(4)$
4	$in(4) = \{b\} \cup (out(4) \setminus \{a\})$	$out(4) = in(5)$
5	$in(5) = \{a, m\} \cup out(5)$	$out(5) = in(2) \cup in(6)$
6	$in(6) = \{c\}$	$out(6) = \emptyset$



NB: at each iteration, we first compute the *in*, then the *out*

	$in = out$	in	out	in	out	in	out	in	out	in	out
1	\emptyset	\emptyset	a	\emptyset	a, c	c	a, c	c	a, c, m	c, m	a, c, m
2	\emptyset	a	b, c	a, c	b, c	a, c	b, c, m	a, c, m	b, c, m	a, c, m	b, c, m
3	\emptyset	b, c	b	b, c	b, m	b, c, m	b, c, m	b, c, m	b, c, m	b, c, m	b, c, m
4	\emptyset	b	a, m	b, m	a, c, m	b, c, m	a, c, m	b, c, m	a, c, m	b, c, m	a, c, m
5	\emptyset	a, m	a, c	a, c, m	a, c	a, c, m	a, c	a, c, m	a, c, m	a, c, m	a, c, m
6	\emptyset	c	\emptyset	c	\emptyset	c	\emptyset	c	\emptyset	c	\emptyset

Complexity: $O(n^2)$ in the worst case; hardly higher than linear in practice

APPLICATION: MEMORY ALLOCATION

If two variables are never simultaneous live, they **do not interfere** and can be stored in the same memory cell or register

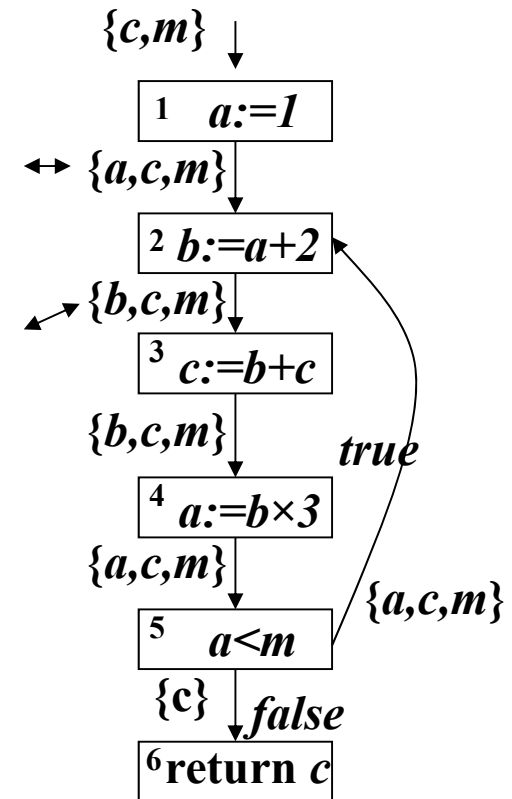
(NB: in general, given n variables there are $n*(n-1)/2$ distinct unordered pairs of variables)

Pairs (a,c) (c,m) and (a,m) interfere
(they are present in $in(2)$)

Pairs (b,c) (b,m) and (c,m) interfere
(they are present in $in(3)$)

a and b do not interfere

the four variables a, b, c, m can be stored in
three 'memory cells'



Modern compilers use such heuristics based on the interference relation
to assign registers to variables

APPLICATION: USELESS DEFINITIONS

An instruction defining a variable is useless if the variable is *not live* out of the instruction

To identify useless definitions: for each instruction p defining a variable a check that $a \in \text{out}(p)$

In the previous example no definition is useless

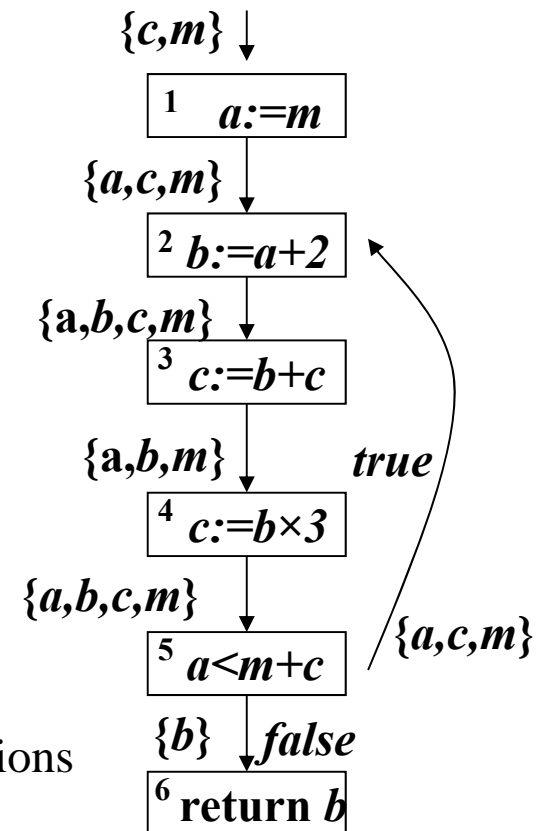
Example – Useless definitions

Variable c is not live out of 3: hence instruction 3 is useless

Removing instruction 3 the program is shortened

c is removed from $\text{in}(1)$, $\text{in}(2)$, $\text{in}(3)$ and $\text{out}(5)$

A program improvement typically allows for further optimizations



ANOTHER USEFUL ANALYSIS: REACHING DEFINITIONS

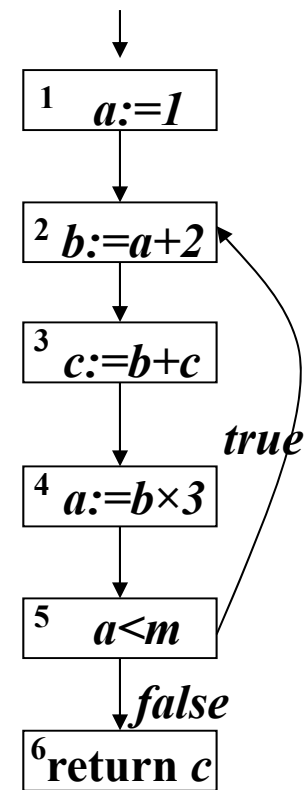
We want to identify the definitions that reach various program points

DEFINITION: A definition of a in q , a_q , reaches the entrance of instruction p (not necessarily distinct from q) if there exists a path from q to p that does not traverse any node, distinct from q , where a is defined

In such a case instruction p is using the value of a as defined by q

Previous example (p.16):

- definition a_1 reaches the entrance of 2, 3, 4 but not of 5
- definition a_4 reaches the entrance of 5, 6, 2, 3, 4



DATA-FLOW EQUATIONS FOR REACHING DEFINITIONS

Computing reaching definitions in various program points as solutions of data-flow equations

If node p defines variable a , we say that

every other definition a_q , $q \neq p$, of a is *suppressed* by p

$$\mathit{sup}(p) = \{ a_q \mid a \in \mathit{def}(p) \wedge a \in \mathit{def}(q) \wedge q \neq p \}$$

NB: recall that $\mathbf{def}(p)$ can include more than one variable in case of read instructions

such as $\mathbf{read}(a, b, c)$

DATA-FLOW EQUATIONS FOR REACHING DEFINITIONS:

Eq. (1) assumes that no varbl. is
passed as input parameter
Otherwise ***in*(1)** contains external
definitions, denoted e.g. as ***x***,

For the initial node 1 :

$$(1) \quad in(1) = \emptyset$$

For every other node $p \in I$:

$$(2) \quad out(p) = def(p) \cup (in(p) \setminus sup(p))$$

$$(3) \quad in(p) = \bigcup_{\forall q \in pred(p)} out(q)$$

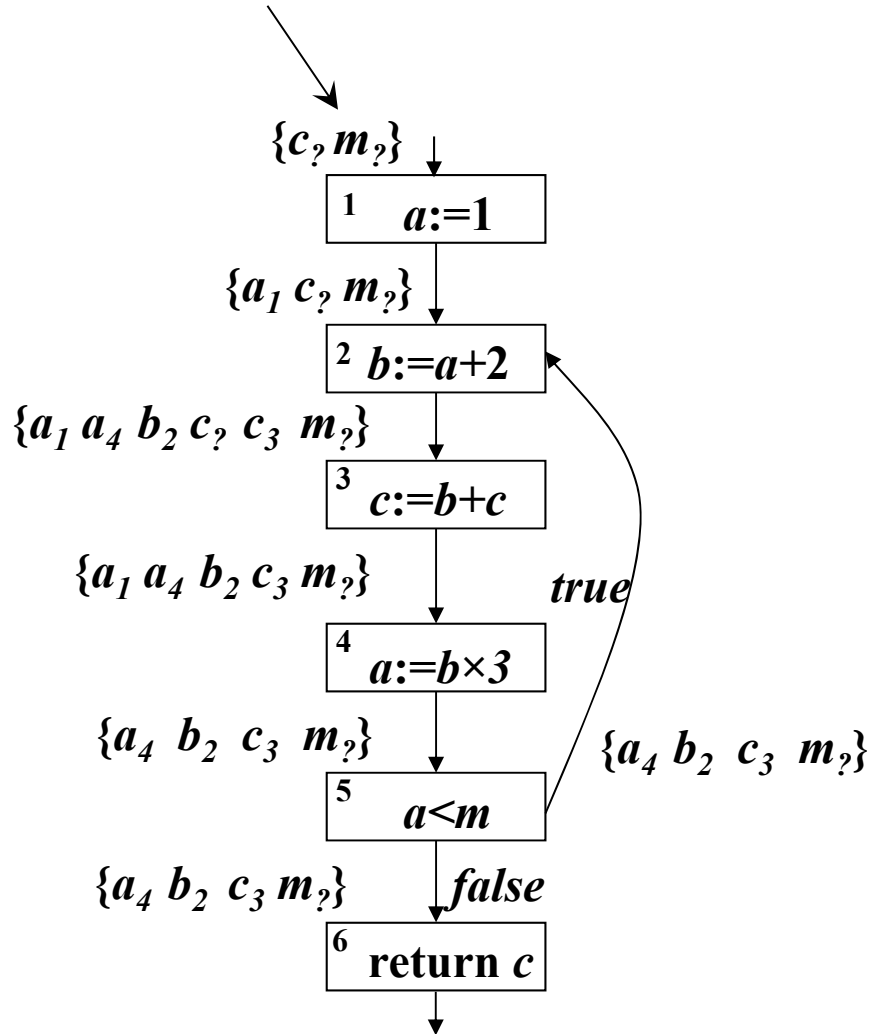
Eq. (2) includes in ***out*(*p*)** the definitions of ***p*** and those reaching the entrance of ***p***,
except for those suppressed by ***p***

eq. (3) states that all definitions reaching the exit of some predecessor of ***p***
also reach the entrance of ***p***

The equation system is solved by iteration, starting from empty sets, until the first fixed point
(just as we do for the liveness equations)

Example – Reaching definitions

NB: c and m program parameters, defined in some unknown external point



node	$\ def\ sup$
1 $a := 1$	$\ a_1 \ a_4$
2 $b := a + 2$	$\ b_2 \ \emptyset$
3 $c := b + c$	$\ c_3 \ c_?$

node	$\ def\ sup$
4 $a := b \times 3$	$\ a_4 \ a_1$
5 $a < m$	$\ \emptyset \ \emptyset$
6 return c	$\ \emptyset \ \emptyset$

$$in(1) = \{c?, m?\}$$

$$out(1) = \{a_1\} \cup (in(1) \setminus \{a_4\})$$

$$in(2) = out(1) \cup out(5)$$

$$out(2) = \{b_2\} \cup (in(2) \setminus \emptyset) = \{b_2\} \cup in(2)$$

$$in(3) = out(2)$$

$$out(3) = \{c_3\} \cup (in(3) \setminus \{c_?\})$$

$$in(4) = out(3)$$

$$out(4) = \{a_4\} \cup (in(4) \setminus \{a_1\})$$

$$in(5) = out(4)$$

$$out(5) = \emptyset \cup (in(5) \setminus \emptyset) = in(5)$$

$$in(6) = out(5)$$

$$out(6) = \emptyset \cup (in(6) \setminus \emptyset) = in(6)$$

NB: b not defined elsewhere

CONSTANT PROPAGATION a very useful and effective optimization technique

consider the possibility to replace a variable with a constant

for instance (following previous example)

it is **not** possible to replace variable a (instr. 2) with constant 1 (assigned by instr. 1: def. a_1)

because the set **in**(2) includes also another definition of a , namely definition a_4

GENERAL CONDITION

It is possible to replace a variable a , inside instruction p , with constant k if:

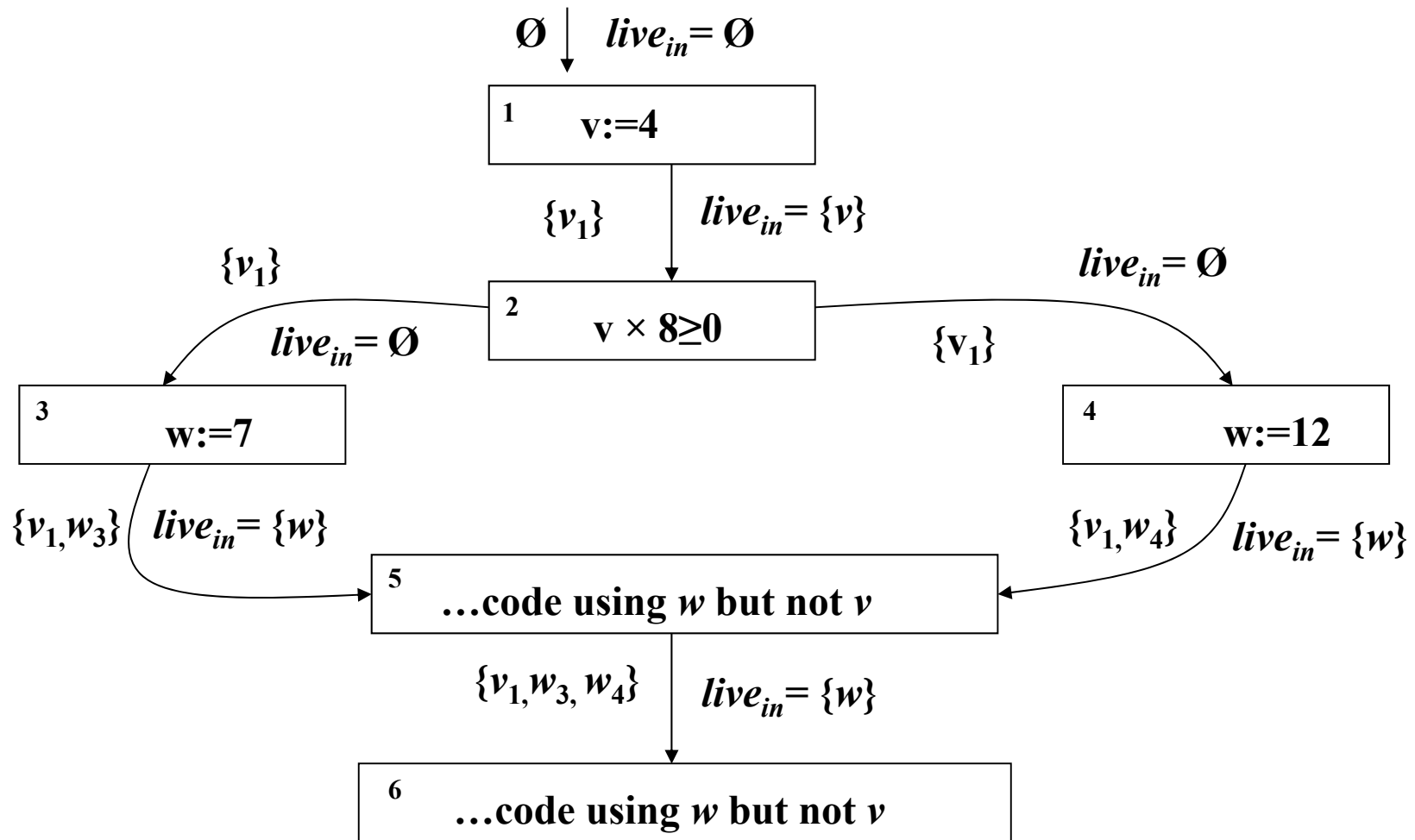
1) there exists an instruction $q: a := k$, such that definition a_q reaches the entrance of p

AND

2) no other definition a_r , $r \neq q$ reaches the entrance of p

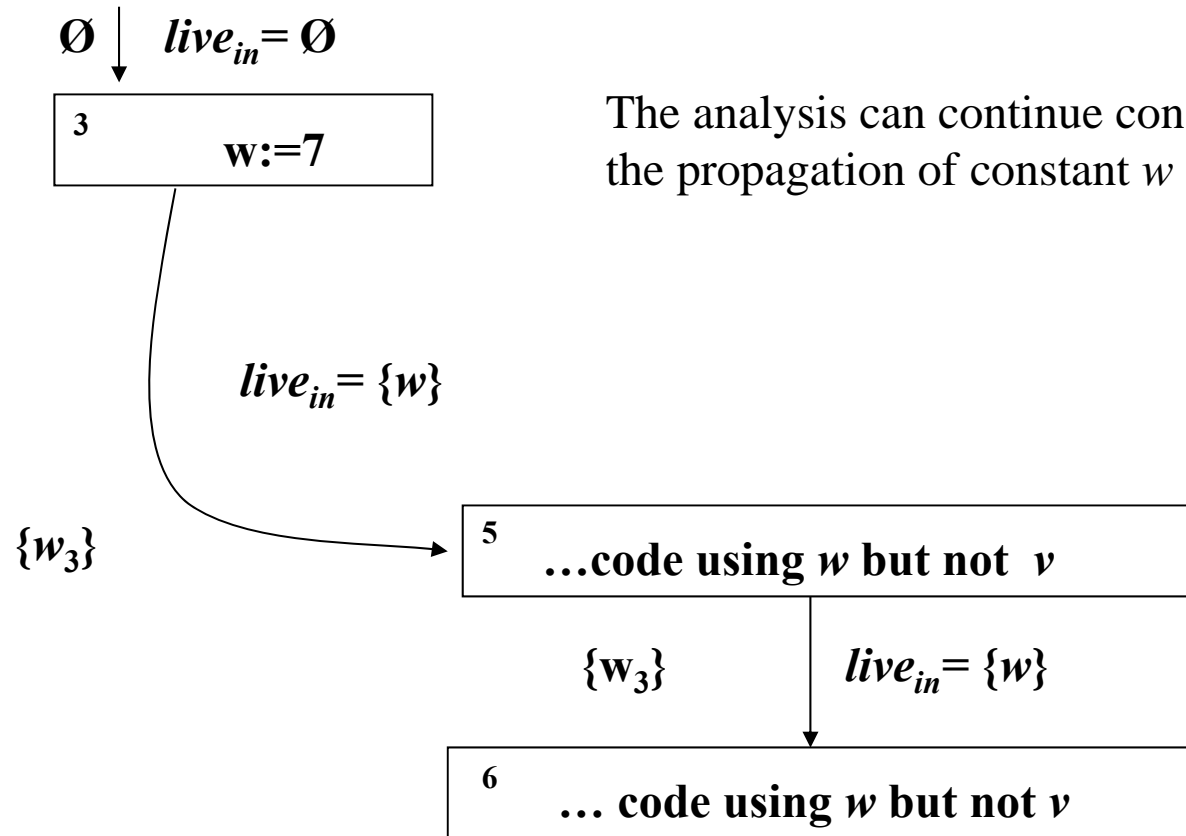
Example – constant propagation (improvements obtained and further induced simplifications)

reaching definitions and live variables:



propagation of constant 4 for v allows one to remove one of the two branches of the *if* instr.

Simplified program:



The analysis can continue considering the propagation of constant $w = 7$ to the rest of the program

AVAILABILITY OF VARIABLES AND INITIALIZATIONS

The compiler checks that at execution time every variable, when used, has a value
(it is not “undefined”)

Otherwise the variable is *unavailable* and an error occurs

DEFINITION – A variable *a* IS AVAILABLE on the entrance to a node *p*,
if *every* path from the initial node to the entrance of *p* includes a definition of *a*
(we ignore input parameters...)

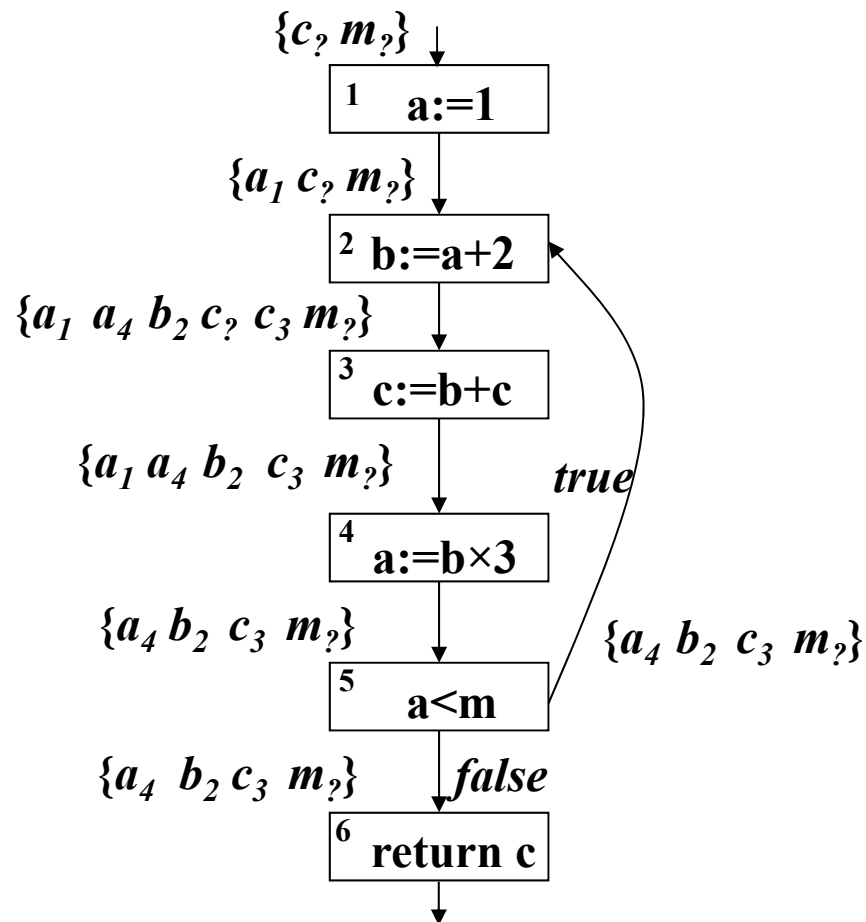
AVAILABILITY OF VARIABLES AND INITIALIZATIONS: Example

3 uses c , to which no value is assigned in path 123

c and m might be input parameters of the subprogram

a available on the entrance of 2 because it was defined in 1 and also in 4

b available on the entrance of 3 because it was defined in 2



AVAILABILITY vs REACHING DEFINITIONS

If a definition a_q reaches the entrance of p , certainly there exists a path from 1 to p that traverses the defining point q .

But there might be *another* path, from 1 to p , not traversing q nor any other definition of a :
in this case it is not guaranteed that variable a is available at entrance of p

The concept of availability is stronger (i.e., more constraining) than that of reaching definition:
there is a difference in *quantification over paths*

A variable a is *available* on the entrance of p if **some definition of a reaches node p in every circumstance**, i.e., *for every* node q predecessor of p , the set $out(q)$ of definitions reaching the exit of q contains a definition of a

For an instruction q we denote $out'(q)$ the set of variable definitions reaching the exit from q , with subscripts deleted. Ex.: if $out(q) = \{a_1 a_4 b_3 c_6\}$, then $out'(q) = \{a b c\}$

BADLY INITIALIZED VARIABLES: An instruction p is badly (or not not well) initialized if there exists a predecessor q of p , such that the definitions reaching its exit do not include all variables used in p (\Rightarrow when the computation follows a path through q , some variable used in p may not have a value)

formally: **instruction p is not well initialized if**

$\exists q \in pred(p)$ such that $\neg (use(p) \subseteq out'(q))$

Example – Detecting uninitialized variables

Condition of **bad** (lack of) **initialization**

$$\exists q \in \text{pred}(p) \quad \text{such that} \quad \text{use}(p) \not\subseteq \text{out}'(q)$$

This condition is false at node 2:
every predecessor (1 and 4)
includes in its set *out*
a definition of *a*, the only variable used in 2

It is true at node 3, because there is no
definition of *c* reaching the exit of 2
→ Error at 3: instr. 3 uses an uninitialized variable: *c*

Deleting instruction 3, updating the sets
of reaching definitions, re-evaluating the condition one finds
that 4 is not well initialized: def. *b*₃ is no more included in *out*(3)

The analysis might proceed by deleting instr. 4 etc. ...

