Static Flow Analysis of Programs

Translated and adapted by L. Breveglieri

STATIC FLOW ANALYSIS OF PROGRAMS

Static flow analysis is a technique widely employed by compilers to translate a high level source program into machine code

COMPILATION

FIRST STAGE – translates the program into an intermediate form more similar to the machine language

NEXT STAGE – is applied to the program in the intermediate form and has several potential aims

- *verification* – refined check of the correctness of the program
- *optimization* – program transformation to improve execution efficiency, e.g., register allocation, etc
- *scheduling* – change of the order of the instruction flow to improve execution efficiency and pipeline utilization

It is convenient to model the *control flow graph* of the program as an automaton and to process such an automaton to carry out the above mentioned tasks

CAUTION – the automaton defines a single program and not the entire source programming language !

Static flow analysis is something different form *syntax driven translation* !

IN THE PRESENT CASE – a string recognized by the automaton represents the time sequence of the operations the program can execute, i.e., an execution trace

STATIC FLOW ANALYSIS – consists of studying some properties of the program by means of the methods and techniques of automata theory, formal logic and statistics

Here only automata theory is of our interest and therefore is considered

THE PROGRAM AS AN AUTOMATON

- consider only the simplest instructions, i.e., those in an intermediate form
  - scalar variable and constant
  - assignment to a variable
  - simple unary arithmetic, logic or relational operation
- and consider only *intraprocedural* analysis (not *interprocedural*)

PROGRAM CONTROL GRAPH

- every graph node represents a program instruction (statement)
- if at execution time an instruction *p* is immediately followed by *q*, then the graph has an arc directed from *p* to *q*, i.e., *p* is the *predecessor* of *q*
- the first instruction of the program is the graph input node (*initial* node)
- an instruction without successors is a graph output node (*final* node)
- unconditional instructions have one successor, whereas conditional instructions have two or more successors
- an instruction with two or more predecessors is a *confluence* node (also called *merge* or *join* node)

The control graph is not a complete and faithful representation of the program

- the logical value of a condition (*true* or *false*) that selects the appropriate successor of a conditional instruction is not represented
- the assignment operation is replaced by the following abstractions
  - assigning or reading a variable *defines* that variable
  - referencing a variable in the right member of an assignment, e.g., an expression or a write operation, *uses* that variable

Each node *p* of the control flow graph is associated with these two sets

def (*p*) and *use* (*p*) both are sets of variables

EXAMPLE

$p\colon a := a \oplus b$

$def(p) = \{ a \}$

$use(p) = \{ a, b \}$

$\mathbf{read}\,(a)$

$a := 7$

$def = \{ a \}$

$use = \emptyset$

these two statements are undistinguishable in the control graph

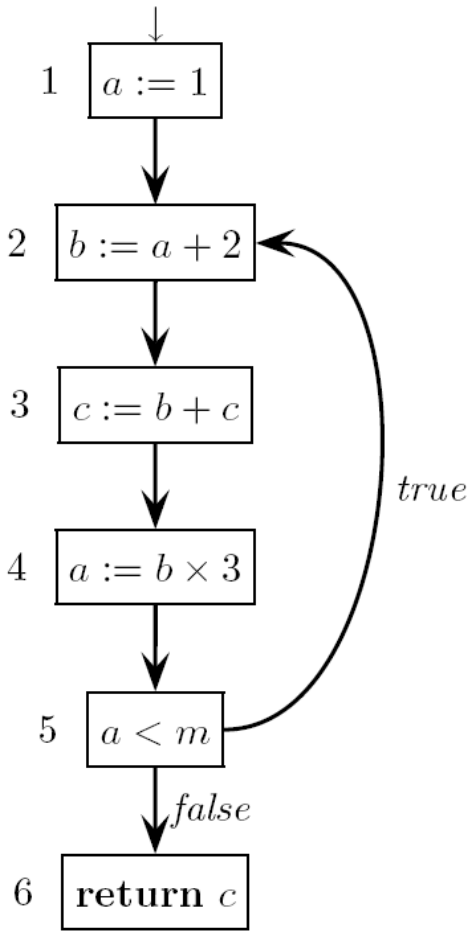
they have the same *def* and *use* sets

EXAMPLE – block scheme of a program and related control flow graph

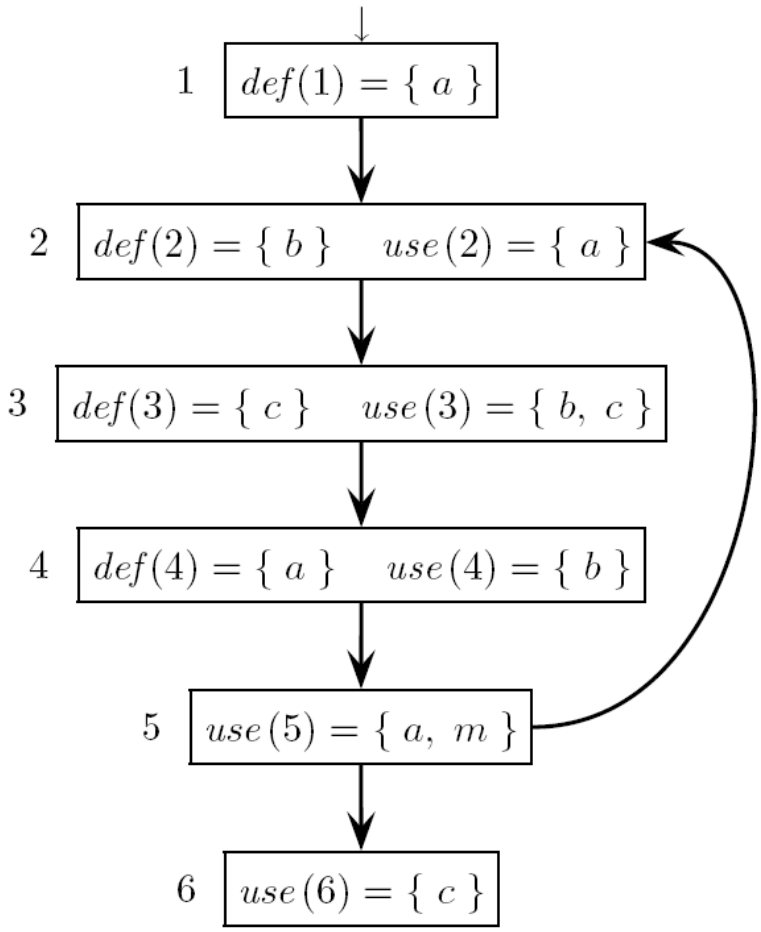
program

```
      a := 1
e_1:  b := a + 2
      c := b + c
      a := b × 3
      if a < m goto e_1
      return c
```

program block scheme



control flow graph



the control flow graph is similar to a local automaton with labeled nodes

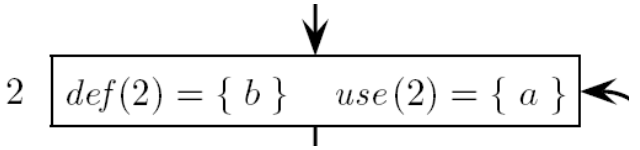
DEFINITION – language of the control graph

The finite state automaton represented by the control flow graph has an instruction set *I* as *terminal alphabet* and each instruction is defined as

$$\langle\, 2,\ def(2) = \{\, b \,\},\ use(2) = \{\, a \,\} \,\rangle$$

a triple with node name and sets *def* and *use*

e.g., from this node



The control flow graph has some special nodes

- *initial state*: the (unique) node without predecessors
- *final state(s)*: any nodes(s) without successors

The formal language *L* recognized by the automaton contains the strings over the alphabet *I* that label the paths from the initial state to a final state

Each such path represents a sequence of program instructions the machine can execute when the program is run

Language *L* is *local*, i.e., it belongs to a subfamily of *REG*, because each node has a distinguished label unique to that node

PREVIOUS EXAMPLE – instruction set $I = \{1 \ldots 6\}$

A recognized 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 all the paths that form the language is the regexp $1\ (2345)^{+}\ 6$

PRECAUTIONARY APPROXIMATION

The graph may contain sequences of instructions (paths) that are not executable as it does not model the clauses of the conditional instructions, like

$$1\colon \textbf{if } a**2 \geq 0 \textbf{ then } 2\colon istr_2 \textbf{ else } 3\colon istr_3$$

The formal language accepted by the automaton contains the paths *12* and *13*, yet actually path *13* is not executable as a square may not be negative

In general it is undecidable whether a path in the control flow graph is executable or not, as the Turing halting problem is reducible to such a problem

PRECAUTIONARY DIAGNOSIS – examining all the paths from the input to an output node may lead to diagnose inexistent errors or to make a precautionary assignment of unnecessary resources; however all effective errors are always diagnosed and in conclusion the method is *error safe*, though possibly inefficient

HYPOTHESIS – for the static flow analysis the automaton has to be in the reduced form, i.e., all the nodes are useful (reachable and defined)

If this does not happen, then some inefficiencies may show up

- the program may not end and so have redundant instructions
- the program may have unreachable instructions, i.e., *dead code*

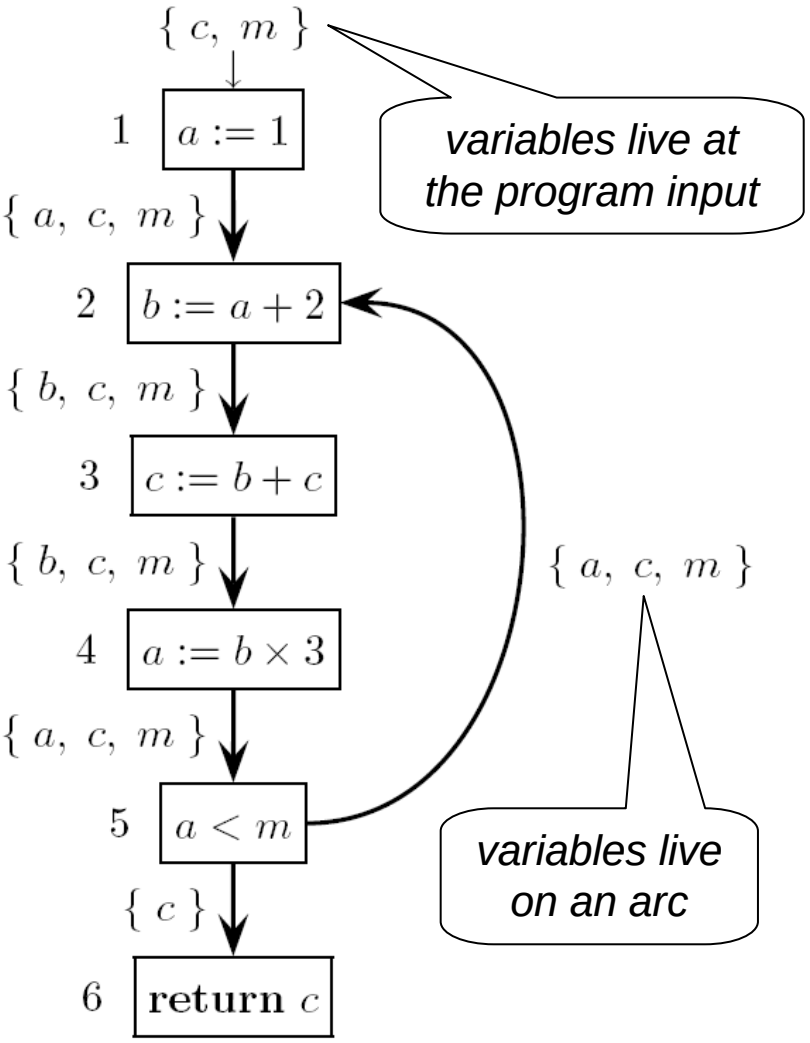
LIVENESS INTERVAL OF A VARIABLE

DEFINITION – a variable *a* is *live* at a point *p,* if the program control graph has a path from point *p* to another point *q* and both conditions below hold

- the path *p–q* does not pass through any instruction *r* (with *r* != *q*) that defines variable *a*, i.e,. an instruction *r* such that it holds $a \in def(r)$
- instruction *q* uses variable *a,* i.e., it holds $a \in use(q)$

According to the definition the variable *a* is live also at all the points on the path from *p* to *q*, which so constitute a *liveness interval* for *a*

Informally a *variable is live at a point p*, if some instruction that sooner or later can be executed starting from *p* uses (= is influenced by) the value the variable happens to have at point *p*



A variable is *live at the*

- *output of a node* if it is live on at least one arc outgoing from the node
- *input of a node* if it is live on at least one arc ingoing into the node

EXAMPLE

c is live at the input of node *1* because there is path *123*: $c \in use(3)$ and none of the instructions *1* or *2* defines *c*

a is live on both paths *12* and *452*
a is not live on either path *234* or *56*

At the output of node *5* the variables { *a, c, m* } U { *c* } are all live

HOW TO COMPUTE THE LIVENESS INTERVALS – let *I* be the set of instructions, and let *D* (*a*), *U* (*a*) $\subseteq$ *I* be the sets of the instructions that define and use variable *a*

A variable *a* is live at the output of node *p* if the following condition holds for the language accepted by the automaton (of the control graph of the program)

LIVENESS CONDITION – a variable *a* is *live at the output of p*, if in the language *L* (A) of the control graph automaton *A* there is a phrase $x = u\ p\ v\ q\ w$ such that

$$u,\ w \in I^* \ \wedge\ p \in I \ \wedge\ v \in \left(I \setminus D(a)\right)^* \ \wedge\ q \in U(a)$$

The set of all the strings *x* that meet this condition is a regular language $L_p \subseteq L$

$$L_p = L(A) \cap R_p$$
$$R_p = I^*\ p\ \left(I \setminus D(a)\right)^*\ U(a)\ I^*$$

The liveness condition prescribes that char *p* has to be followed, immediately or at some distance, by a char *q* of set *U* (*a*), and that all the chars on the path(s) from *p* to *q* may not belong to set *D* (*a*)

To check if variable *a* is live at the output of *p*, check if the language L_p is *empty*

To check if language L_p is empty, we can build the recognizer of L_p, i.e., the product machine of A and of the recognizer of R_p, and check if there are paths that connect the input node to some final node

But this method is time consuming for large programs and a different one comes in

FLOW EQUATION METHOD – it simultaneously determines all the live variables

This method examines the existence of some paths that connect the point where the variable under examination is defined, to the point where it is used

for every final node *p*

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

live$_{in}$ (*p*) – set of all the vars live at the input of *p*

live$_{out}$ (*p*) – set of all the vars live at the output of *p*

for every other node *p*

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

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

succ (*p*) is the set of all the nodes that immediately follow *p*

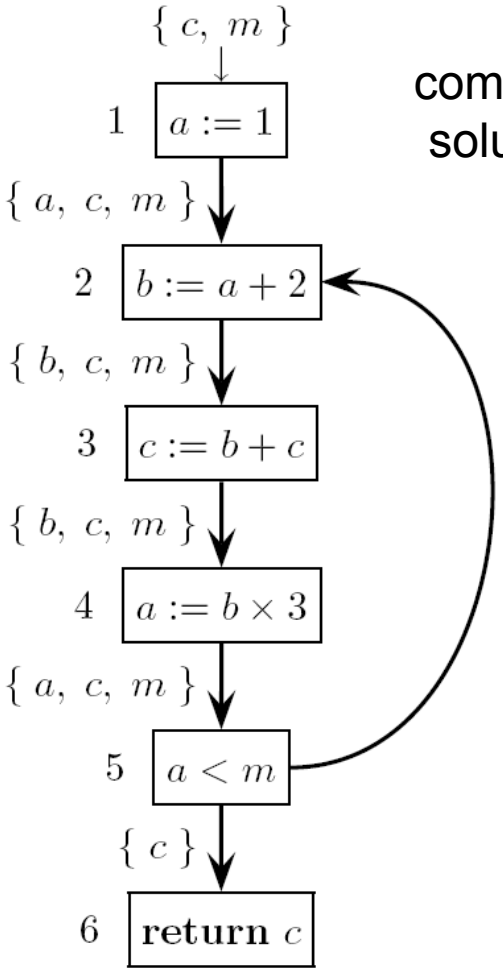
it is a back-propagation computation scheme !

EXAMPLE – computing a few live variables

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

$$live_{out}(p) = \bigcup_{\forall\, q \in succ(p)} live_{in}(q)$$ non-final nodes

$live_{out}(p) = \emptyset$ final node(s)



complete solution

succ (5) = { 2, 6 }

{ a, c, m }

var.	D	U
a	1, 4	2, 5
b	2	3, 4
c	3	3, 6
m	∅	5

instruction sets that define and use the variables

solution verification

$live_{out}(6) = \Phi$

$live_{in}(6) = use(6) \cup (live_{out}(6) \setminus def(6)) = \{ c \} \cup \Phi = \{ c \}$

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

$live_{in}(5) = use(5) \cup (live_{out}(5) \setminus def(5)) =$

$= \{ a, m \} \cup (\{ a, c, m \} \setminus \{ a, m \}) = \{ a, c, m \}$

$live_{out}(4) = live_{in}(5) = \{ a, c, m \}$

$live_{in}(4) = use(4) \cup (live_{out}(4) \setminus def(4)) =$

$= \{ b \} \cup (\{ a, c, m \} \setminus \{ a \}) = \{ b, c, m \}$

and so on verify the remaining nodes *3*, *2* and *1*

HOW TO SOLVE THE FLOW EQUATIONS

For a graph with a number $|I| = n \geq 1$ of nodes, we have a set of $2 \times n$ equations that contain $2 \times n$ unknowns, i.e., $live_{in}(p)$, $live_{out}(p)$ for every node $p \in I$

The solution of such an equation set is a family of $2 \times n$ sets of variables

We solve the equation set iteratively by initially assigning the empty set to each unknown (this is iteration number $i = 0$) as follows

$$\forall\, p \in I: \qquad live_{in}(p) = \emptyset \qquad live_{out}(p) = \emptyset$$

Then in the current solution i we replace the equation values and number the new solution as $i + 1$; if solutions i and $i + 1$ differ, we go on and make one more step; otherwise the procedure has converged to the final solution and so we stop

The computation converges after a finite number of iteration steps because

1. every set $live_{in}(p)$ and $live_{out}(p)$ has a cardinality upper bounded by the total number of variables in the program
2. every iteration step either increases the cardinality of the above sets or at worst leaves them unchanged, as the equations are monotonic
3. when the solution does not change any more, the algorithm terminates

EXAMPLE – iterative computation of live variables

instruction sets that define and use the variables

var.	D	U
a	$1,\ 4$	$2,\ 5$
b	2	$3,\ 4$
c	3	$3,\ 6$
m	$\emptyset$	5

flow equations

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	$in(6) = \{ c \}$	$out(6) = \emptyset$

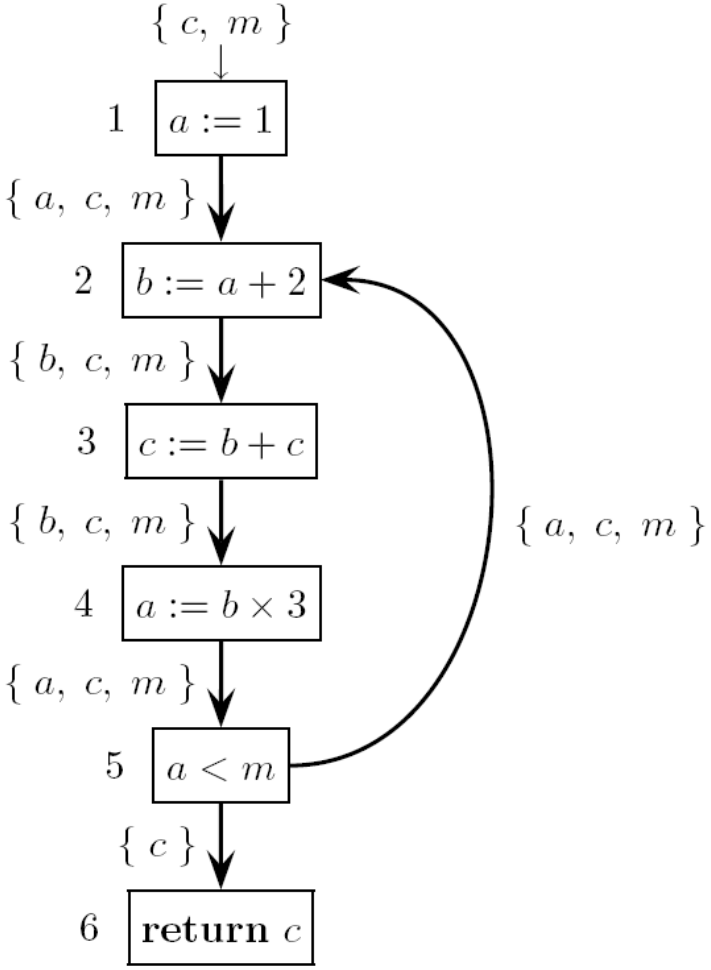
the next step does not change anything

	$in = out$	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	$\emptyset$

OTHER POSSIBLE APPLICATIONS OF THE FLOW EQUATION METHOD

MEMORY ALLOCATION – if two variables are not simultaneously live, they do not interfere and can be allocated in the same register or memory location

EXAMPLE – variables that do interfere or don't



Pairs *a, c* and *c, m* interfere as both are in *in* (2)

Pairs *b, c* and *b, m* interfere as both are in *in* (3)

Variables *a* and *b* do not interfere

For vars *a*, *b*, *c* and *m*, three locations suffice

Modern compilers allocate variables by using the above relationship and heuristic methods

USELESS DEFINITION – an instruction that defines a variable is *useless* if the variable is not live on at least one outgoing arc of the instruction

To identify useless instructions, first search an instruction *p* that defines a variable *a*, then check whether variable *a* belongs to the set *out* (*p*)

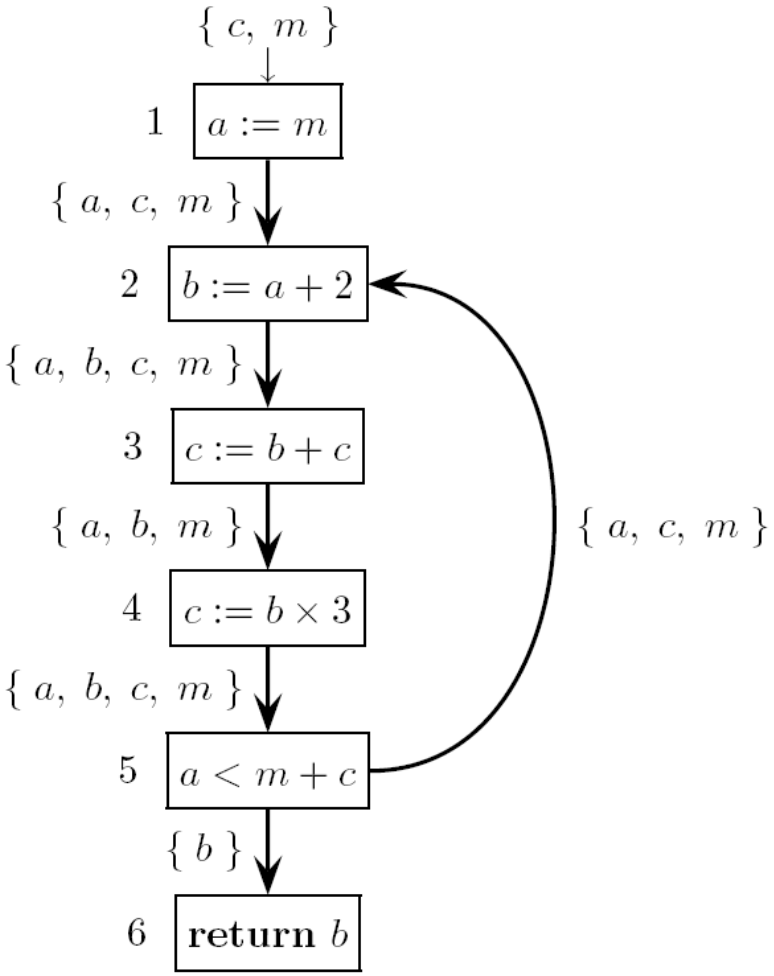
In the previous example there are not any useless instructions

NEW EXAMPLE – useless instructions

Variable *c* is not live at the output of *3*, hence instruction *3* is useless

If instruction *3* is eliminated, the program gets shorter and variable *c* vanishes from the sets *in* (*1*), *in* (*2*), *in* (*3*) and *out* (*5*)

After optimizing a program, sometimes more optimization is enabled



REACHING DEFINITION

Analysis of the variable definitions, e.g., assignments, that reach the points (instructions) of the program

DEFINITION – the definition of a variable *a* in an instruction *q* (denoted as a_q) *reaches* the input of an instruction *p* (*q* and *p* may coincide) if there is a path from *q* to *p* the inner nodes of which do not define the variable *a* again

Instruction *p* can see and use the value of the variable *a* defined in instruction *q*

Given an automaton *A* with instruction set *I*, here is the equivalent condition

REACHING DEFINITION CONDITION – definition a_q reaches the input of *p* if in the language $L(A) \subseteq I^*$ there is a phrase $x = u\ q\ v\ p\ w$ and it holds

$$u,\ w \in I^* \ \wedge\ q \in D(a) \ \wedge\ v \in \left(I \setminus D(a)\right)^* \ \wedge\ p \in I$$

q and *p* may coincide

PREVIOUS EXAMPLE

- definition a_1 reaches the input of nodes *2*, *3* and *4*, but not of node *5*
- definition a_4 reaches the input of nodes *5*, *6*, *2*, *3* and *4*

FLOW EQUATIONS FOR THE REACHING DEFINITIONS

The computation of the reaching definitions in a program can be reformulated in terms of a set of flow equations

If a node *p* defines a variable *a*, then every other definition a_q of the same variable *a* in some node *q* (with $q \neq p$) is said to be *suppressed* by *p*

Assume *q* is any node, not necessarily one that reaches *p*; the set *sup* (*p*) of the definitions suppressed by *p* is expressed as follows

$$sup\,(p) = \left\{ a_q \;\middle|\; \begin{array}{l} q \in I \;\wedge\; q \neq p \;\wedge \\ a \in def(q) \;\wedge\; a \in def(p) \end{array} \right\} \quad \text{if } def(p) \neq \emptyset$$

$$sup\,(p) = \emptyset \quad \text{if } def(p) = \emptyset$$

The set *def* (*p*) may contain two or more variable names as for instance in the case of an I/O instruction like "**read** (*a*, *b*, *c*)"

FLOW EQUATIONS FOR THE REACHING DEFINITIONS

The first equation assumes there are not any variables passed as input parameters

Otherwise the set *in* (*1*) may also contain some definitions external to the subprogram

pred (*p*) is the set of all the nodes that immediately precede *p*

for the initial node *1*

$$in(1) = \emptyset$$

for every other node *p*

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

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

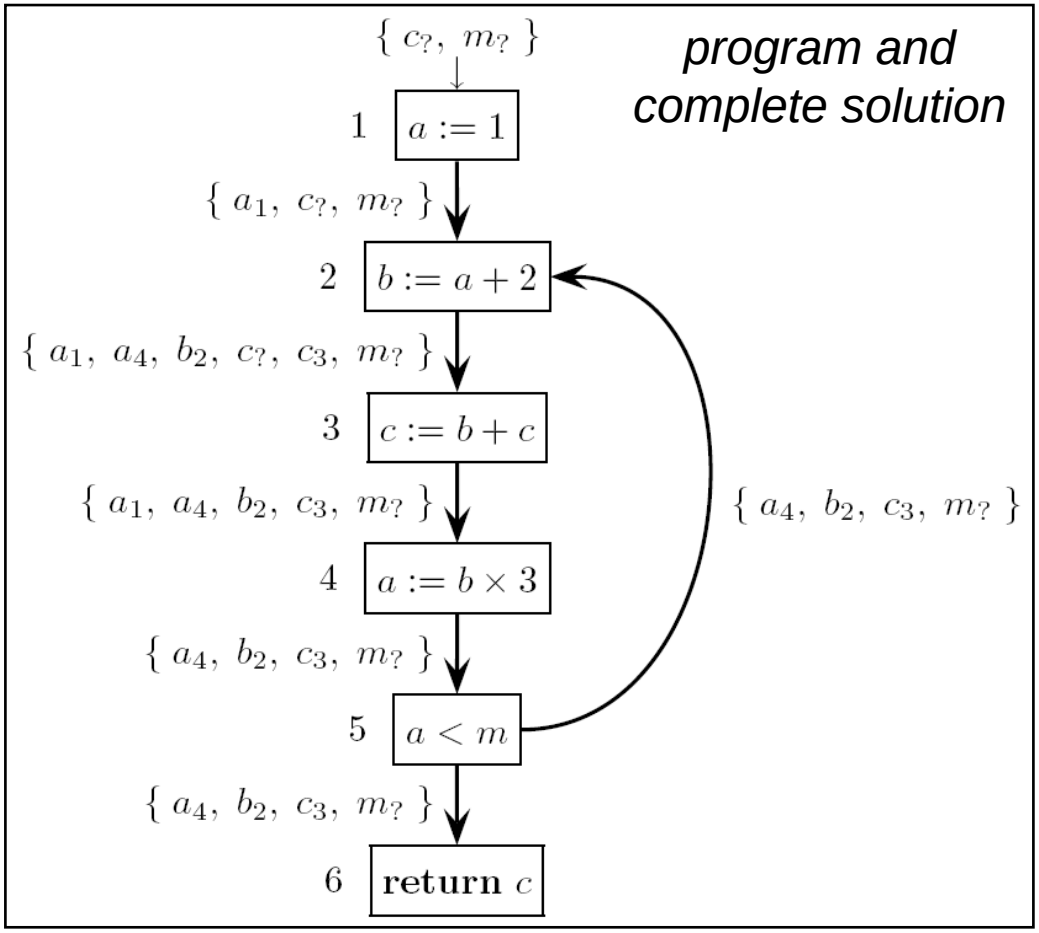
The second equation puts the definitions of *p* into the output of *p*, and puts therein any other definition that reaches the input of *p* and is not suppressed by *p* itself

The third equation collects all the definitions that reach the output of any node that immediately precedes *p*, and puts all of them into the input of *p*

This is a forward-propagation computation scheme !

Solve the above system of equations iteratively, similarly to the live variables

EXAMPLE – reaching definitions



constant terms

node	*instruction*	*def*	*sup*
1	$a := 1$	a_1	a_4
2	$b := a + 2$	b_2	$\emptyset$
3	$c := b + c$	c_3	$c_?$
4	$a := b \times 3$	a_4	a_1
5	$a < m$	$\emptyset$	$\emptyset$
6	return c	$\emptyset$	$\emptyset$

$$
\begin{aligned}
in(1) &= \{\, c_?,\ m_? \,\} \\
out(1) &= \{\, a_1 \,\} \cup (\, in(1) \setminus \{\, a_4 \,\}) \\
in(2) &= out(1) \cup out(5) \\
out(2) &= \{\, b_2 \,\} \cup (\, in(2) \setminus \emptyset \,) = \{\, b_2 \,\} \cup in(2) \\
in(3) &= out(2) \\
out(3) &= \{\, c_3 \,\} \cup (\, in(3) \setminus \{\, c_? \,\}) \\
in(4) &= out(3) \\
out(4) &= \{\, a_4 \,\} \cup (\, in(4) \setminus \{\, a_1 \,\}) \\
in(5) &= out(4) \\
out(5) &= \emptyset \cup (\, in(5) \setminus \emptyset \,) = in(5) \\
in(6) &= out(5) \\
out(6) &= \emptyset \cup (\, in(6) \setminus \emptyset \,) = in(6)
\end{aligned}
$$

the flow equations of the reaching definitions contain the unknowns *in* (…), *out* (…) and can be iteratively solved as the live variables

CONSTANT PROPAGATION

Going on with the previous example, consider the possibility of replacing a constant value to a variable that occurs in an expression

For instance in the instruction *2* (next slide) it is wrong to replace constant 1 to variable *a*, which is assigned a value 1 in the node *1* (definition a_1), as the set *in* (*2*) contains also another definition of the same variable *a* (definition a_4)

In fact definition a_1 holds only for the first loop iteration, whereas definition a_4 holds for all the subsequent iterations

CONSTANT PROPAGATION CONDITION

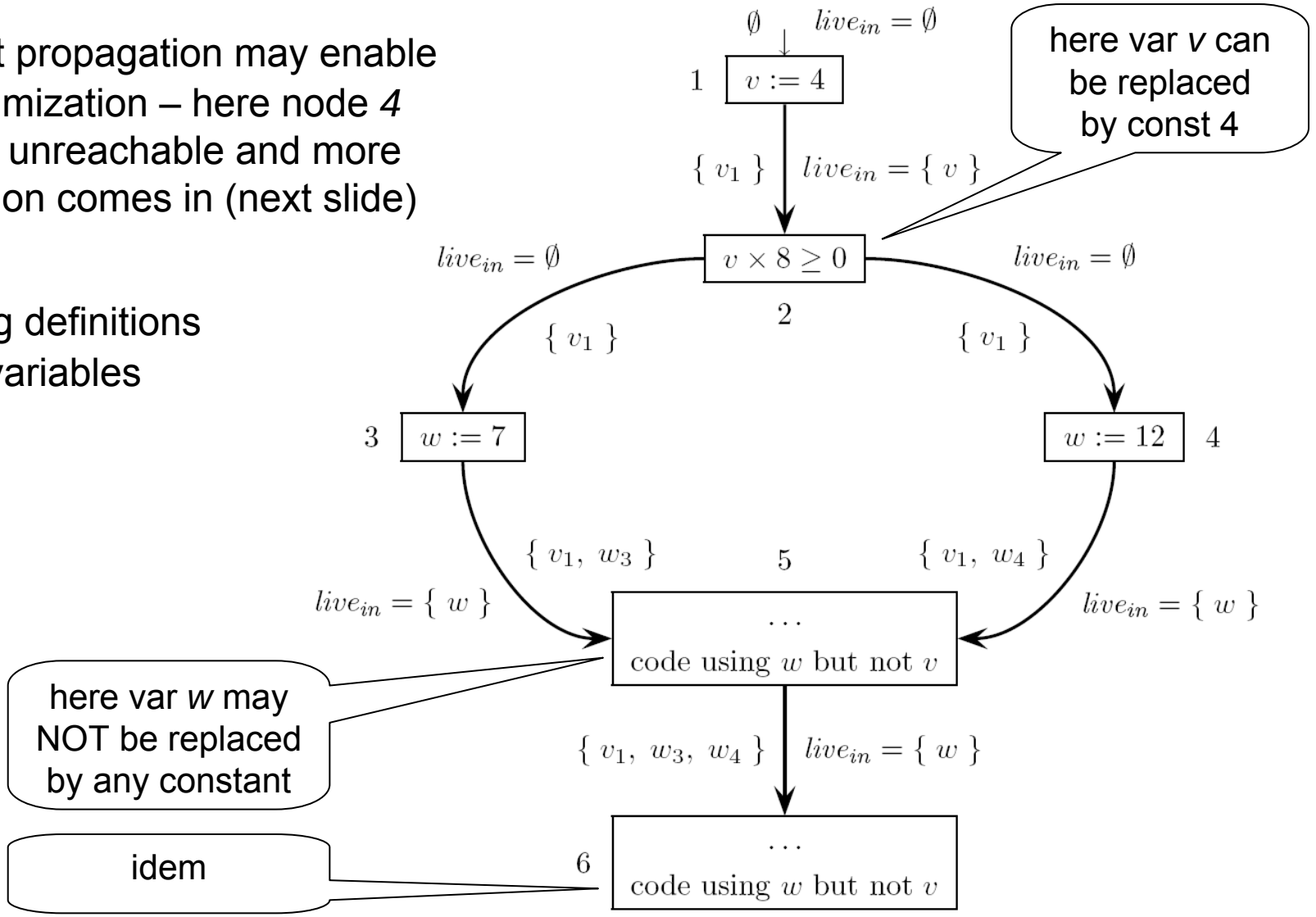
When a variable *a* is used in an instruction *p*, it can be replaced by a constant *k* if both clauses below hold

- there is an instruction *q* that contains an assignment "$a := k$" to the variable *a* with *k* constant, and definition a_q reaches the input of *p*
- there is not any other instruction *r* (with $r \neq q$) with a definition a_r of the variable *a*, such that definition a_r reaches the input of *p*

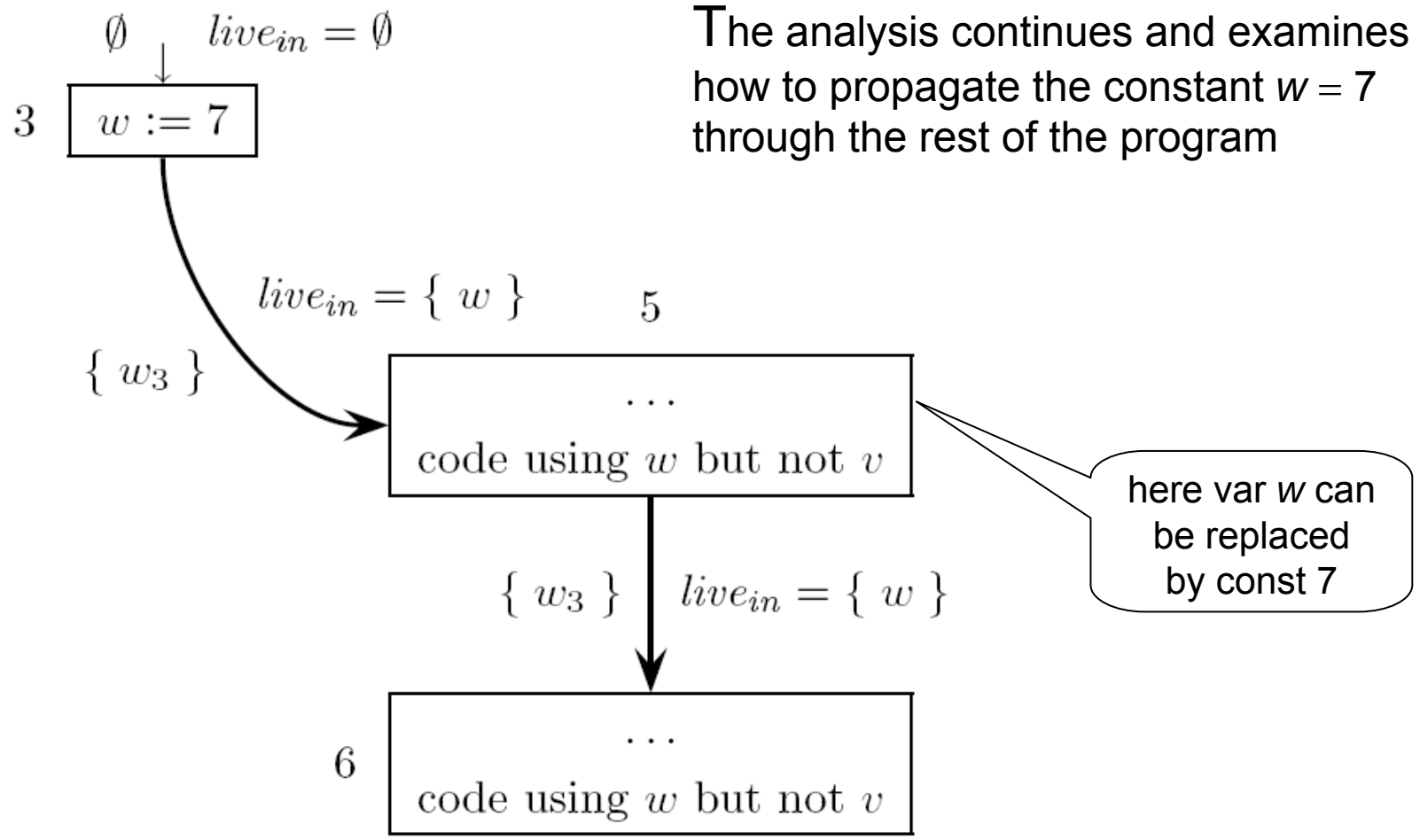
EXAMPLE – constant propagation

Constant propagation may enable more optimization – here node *4* becomes unreachable and more propagation comes in (next slide)

Reaching definitions and live variables



SIMPLIFIED PROGRAM – node *4* is unreachable and is removed



The analysis continues and examines how to propagate the constant $w = 7$ through the rest of the program

VARIABLE AVAILABILITY AND BAD INITIALIZATION

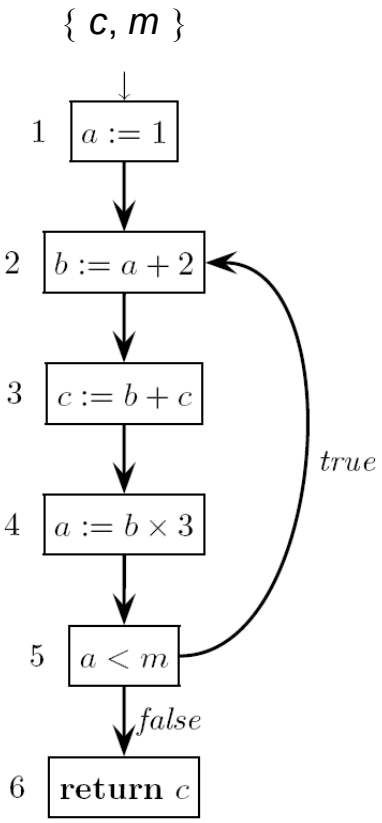
Compilers need to check if all the variables used in an instruction have a value when the instruction is executed

Such values have to come from previous variable assignments or definitions

Otherwise one or more variables are unavailable and an error occurs

EXAMPLE – in the control flow graph aside

- node *3* uses var *c*, which is not assigned any value in *1*, *2* or *3*
- vars *c* and *m* are input parameters of the subprogram
- var *a* is available at the input of *2* because it is assigned in *1*
- var *b* is available at the input of *3* because it is assigned in *2*



DEFINITION – a variable *a* is *available* at the input of a node *p*, i.e., immediately before being used by *p*, if in the control graph every path from the initial node to the input of *p* contains a definition of *a*

VARIABLE AVAILABILITY versus REACHING DEFINITION

If a definition a_q of a variable a at a node q reaches the input of a node p, then there has to be a path from the initial node to p that passes through q

Yet we cannot exclude the existence of another path also from the initial node to p, which does not pass through q or any other node with a definition of a

The notion of availability is more restrictive than that of reaching definition !

If for any node q predecessor of p the set *out* (q) of the reaching definitions on the output of q contains (at least) a definition of a variable a, such a variable is available at the input of $p \Rightarrow$ say some definition of *a always reaches* node p

Define a set *out'* (q) that contains the reaching definitions on the output of q and cancel the subscripts, e.g., *out* $(q) = \{ a_1, a_4, b_3, c_6 \} \Rightarrow$ *out'* $(q) = \{a, b, c\}$

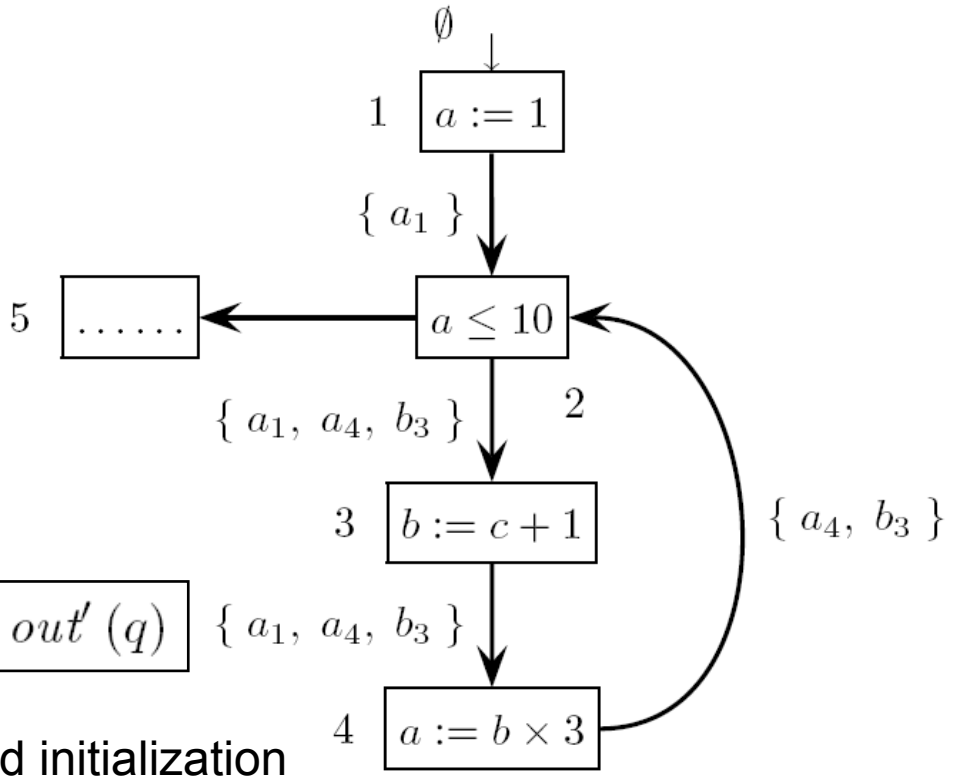
BAD INITIALIZATION – a node p is *badly initialized* if it has a predecessor q the reaching definitions of which do not include all the variables used in p

Equivalently in a computation that passes through q, one or more variables used in p do not have any value yet; formally node p is *badly initialized* if

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

EXAMPLE – discovering the variables that are badly initialized

program with reaching definitions



Condition of bad initialization

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

A FEW CASES – check if there is a bad initialization

False in node *2*, as both predecessors *1* and *4* have a definition of variable *a* in their sets *out'* (*1*) and *out'* (*4*), and variable *a* is the only one used in *2*

True in node *3*, as the predecessor *2* does not have a definition of variable *c* in its set *out'* (*2*) $\Rightarrow$ node *3* uses a badly initialized variable, namely *c*

Remove instruction *3*, update all the program reaching definitions and check again: now *4* is badly initialized as b_3 of *out* (*3*) is (has become) unavailable

Remove instruction *4* and check again: no more bad initializations are found !