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*** an 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 jump 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 are 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*)**

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

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

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

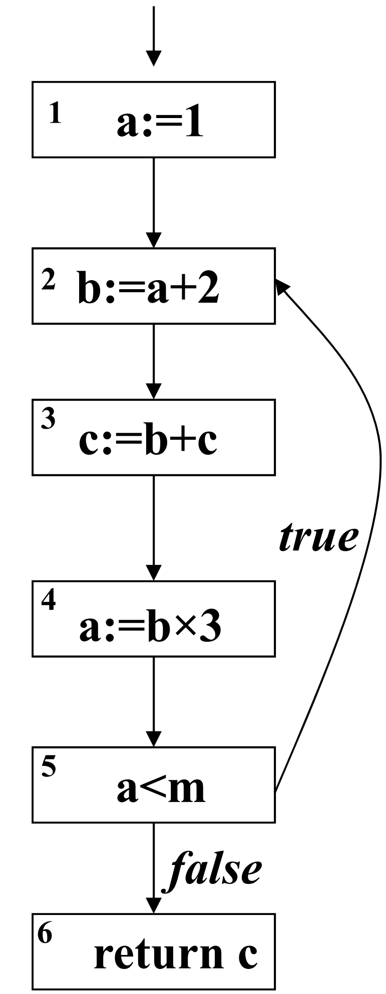
such as **read(*a, b, c*)**

Example 1 – block diagram and control-flow graph

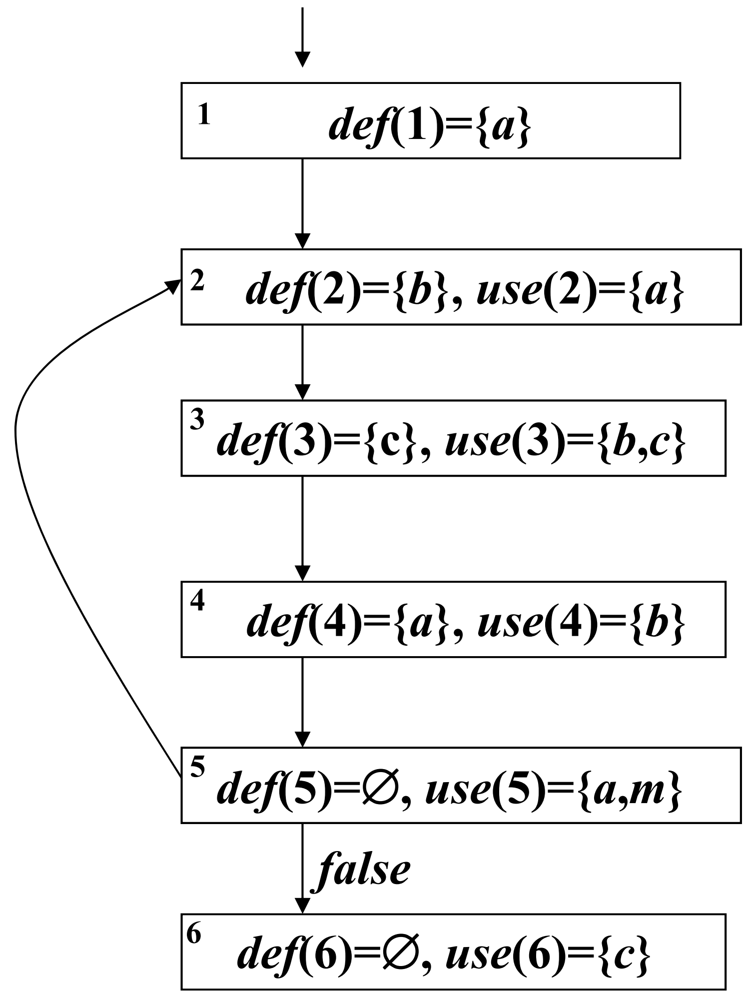
program

```
     a:=1
e1:  b:=a+2
     c:=b+c
     a:=b×3
     if a<m goto e1
     return c
```

block diagram



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

$$\boxed{\left\langle n, def(n) = \{...\}, use(n) = \{...\} \right\rangle}$$

- ***initial state*** : the node 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 … 6}
An accepted path is **1 →2 → 3→ 4→ 5→ 2→ 3→4 → 5→6** = **1234523456**
The set of paths is the language denoted by $\mathbf{1(2345)^{+}6}$

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

$$1: \quad \text{if } a**2 \geq 0 \text{ then } istr_2 \text{ else } istr_3$$

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 non-existing 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***)

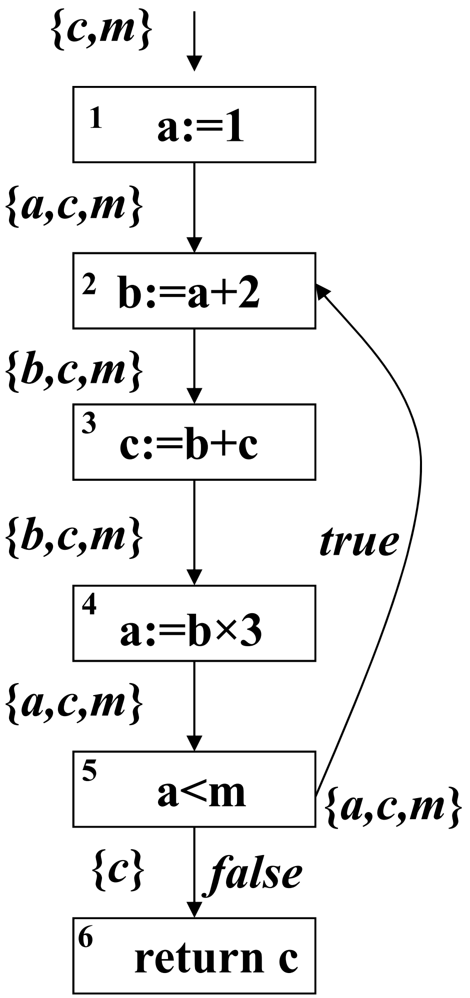
LIVENESS INTERVALS OF VARIABLES

A variable is live at some program point (i.e., ***on an arc***) 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 $\boldsymbol{a}$ is ***live on an arc***, input or output of a program node $\boldsymbol{p}$, if the graph admits a path from $\boldsymbol{p}$ to a node $\boldsymbol{q}$ such that

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

live variables on arcs



A variable is

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

EXAMPLE:

c is live-in for node 1 because of the path
123: $\boldsymbol{c \in use(3)}$ *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 $\{\boldsymbol{a},\boldsymbol{c},\boldsymbol{m}\} \cup \{\boldsymbol{c}\}$

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

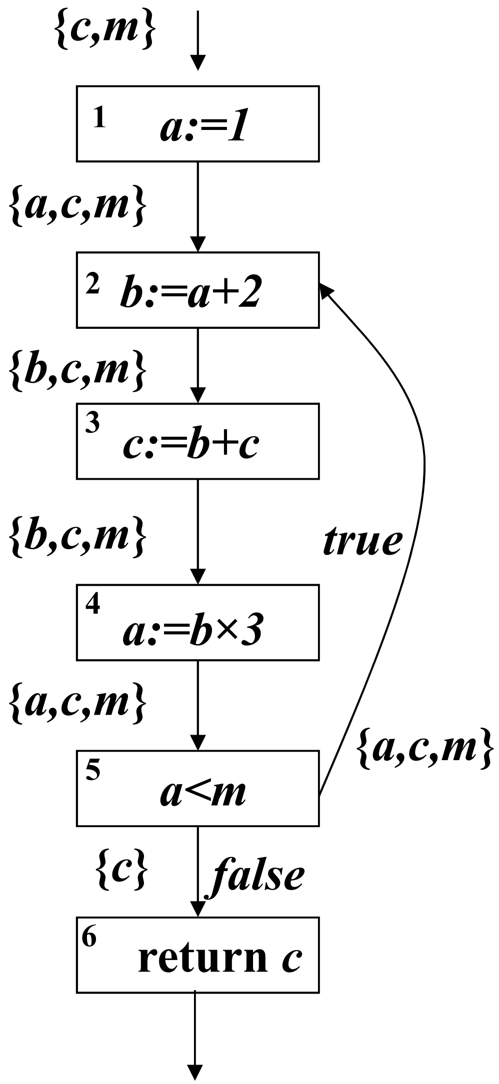
for every final node p :

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

for every other node p:

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

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



for every final node p :

(1) $$live_{out}(p) = \varnothing$$

for every other node p:

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

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

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

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

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

$\boldsymbol{live_{out}}(5) = \boldsymbol{live_{in}}(2) \cup \boldsymbol{live_{in}}(6) =$

$\{\boldsymbol{a, c, m}\} \cup \{\boldsymbol{c}\} = \{\boldsymbol{a, c, m}\}$

SOLUTION OF DATA-FLOW EQUATIONS

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

The system solution is a vector of $\mathbf{2} \cdot \boldsymbol{n}$ sets

The system is solved iteratively

by initially assigning the empty set $\varnothing$ to every unknown (iteration $\boldsymbol{i} = \mathbf{0}$) :

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

the values for iteration $\boldsymbol{i}$ are inserted into the system and used to compute values of interation $\boldsymbol{i}+\mathbf{1}$

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

Otherwise stop and the values of iteration $\boldsymbol{i} + \mathbf{1}$ are the solution

(the usual fixpoint technique…)

The computation converges after a bounded number of iterations:

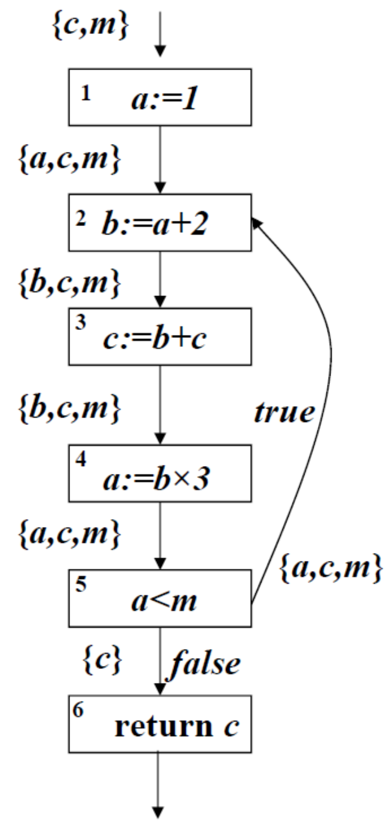
1) $\boldsymbol{live_{in}(p)}$ and $\boldsymbol{live_{out}(p)}$ have a cardinality upperbound determined by the number of program variables

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



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	$\varnothing$	$\varnothing$	a	$\varnothing$	a,c	c	a,c	c	a,c,m	c,m	a,c,m
2	$\varnothing$	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	$\varnothing$	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	$\varnothing$	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	$\varnothing$	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	$\varnothing$	c	$\varnothing$	c	$\varnothing$	c	$\varnothing$	c	$\varnothing$	c	$\varnothing$

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

APPLICATION: MEMORY ALLOCATION

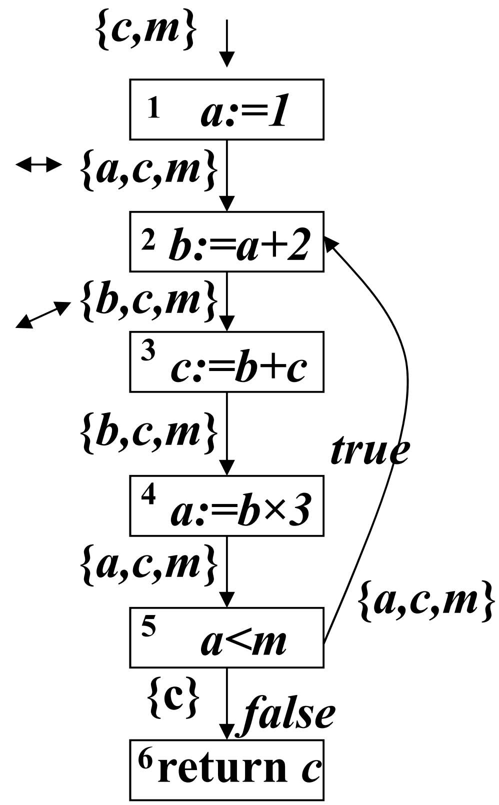
If two variables are never simultaneous live, they **do not interfere**
and can be stored in the same memory cell or register
(NB: in general, given ***n*** variables there are ***n**(*n*-1)/2** distinct unordered pairs of variables)
In the example, 6 pairs (***a***, ***b***), (***a***, ***c***), (***a***, ***m***), (***b***, ***c***), (***b***, ***m***), (***c***, ***m***)

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'



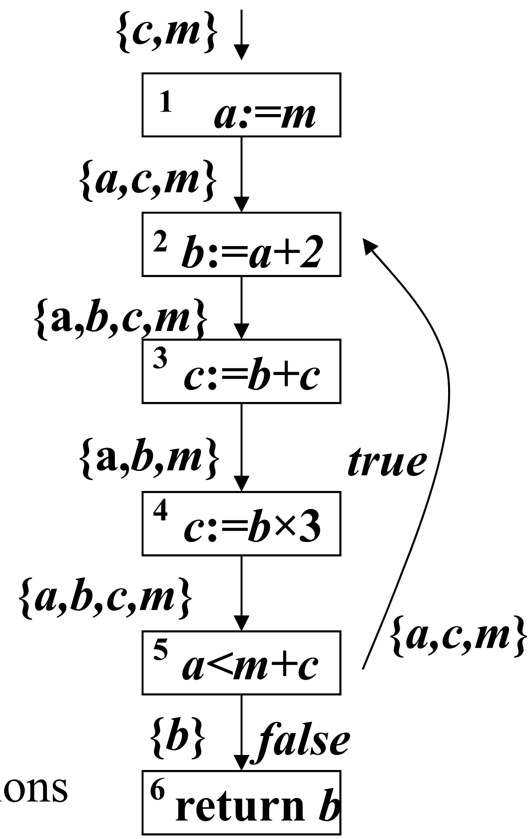
Modern compilers use such heuristics based on the interference relation
to assign registers to variables

ANOTHER 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 $\boldsymbol{p}$ defining a variable $\boldsymbol{a}$ check that $\boldsymbol{a \in out(p)}$

In the previous example no definition is useless
let us modify it



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 ***in*(1), *in*(2), *in*(3)** and ***out*(5)**

A program improvement typically allows for further optimizations

ANOTHER USEFUL ANALYSIS: REACHING DEFINITIONS

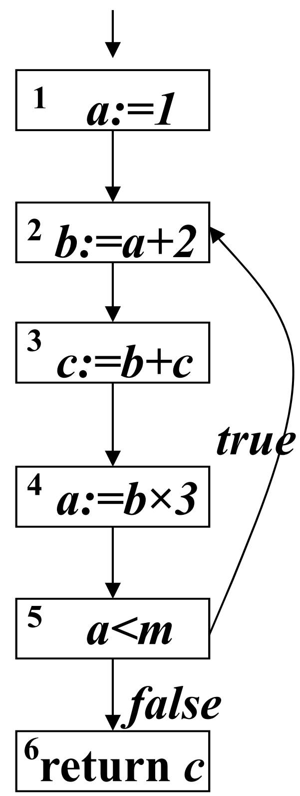
We want to identify the definitions that reach various program points

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

In such a case instruction $\boldsymbol{p}$ is using the value of $\boldsymbol{a}$ as defined by $\boldsymbol{q}$

Previous example (p.16):

- definition $\boldsymbol{a_1}$ reaches the entrance of 2, 3, 4 but not of 5
- definition $\boldsymbol{a_4}$ 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 $\boldsymbol{p}$ defines variable $\boldsymbol{a}$, we say that
every other definition $\boldsymbol{a_q}$, $\boldsymbol{q \neq p}$, of $\boldsymbol{a}$ is ***suppressed*** by $\boldsymbol{p}$

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

NB: recall that **def(*p*)** can include more than one variable in case of read instructions

such as **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) $$in(1) = \varnothing$$

For every other node $p \in I$:

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

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

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

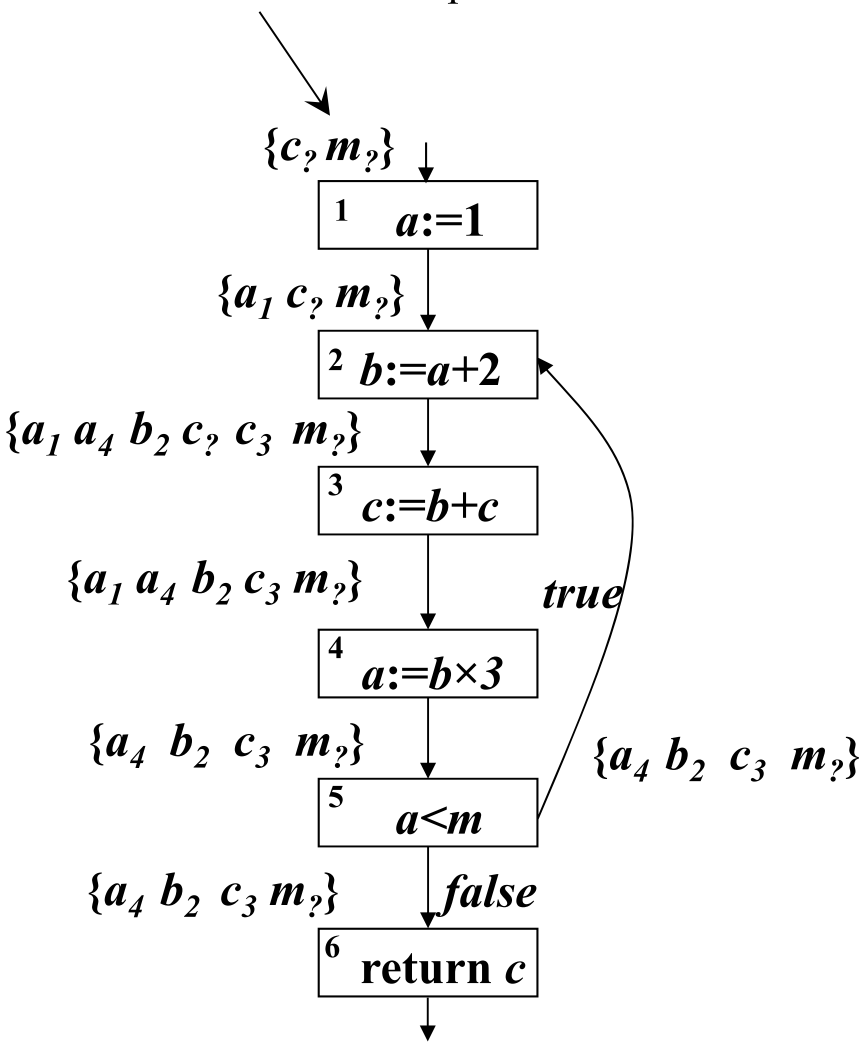
eq. (3) states that all definitions reaching the exit of some predecessor of ***p***
also reach the entrance of ***p***

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	$\varnothing$
3	$c := b + c$	c_3	$c_?$

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

$$in(1) = \{c_?, m_?\}$$
$$out(1) = \{a_1\} \cup (in(1) \setminus \{a_4\})$$
$$in(2) = out(1) \cup out(5)$$
$$out(2) = \{b_2\} \cup (in(2) \setminus \varnothing) = \{b_2\} \cup in(2)$$

NB: *b* not defined elsewhere

$$in(3) = out(2)$$
$$out(3) = \{c_3\} \cup (in(3) \setminus \{c_?\})$$
$$in(4) = out(3)$$
$$out(4) = \{a_4\} \cup (in(4) \setminus \{a_1\})$$
$$in(5) = out(4)$$
$$out(5) = \varnothing \cup (in(5) \setminus \varnothing) = in(5)$$
$$in(6) = out(5)$$
$$out(6) = \varnothing \cup (in(6) \setminus \varnothing) = in(6)$$