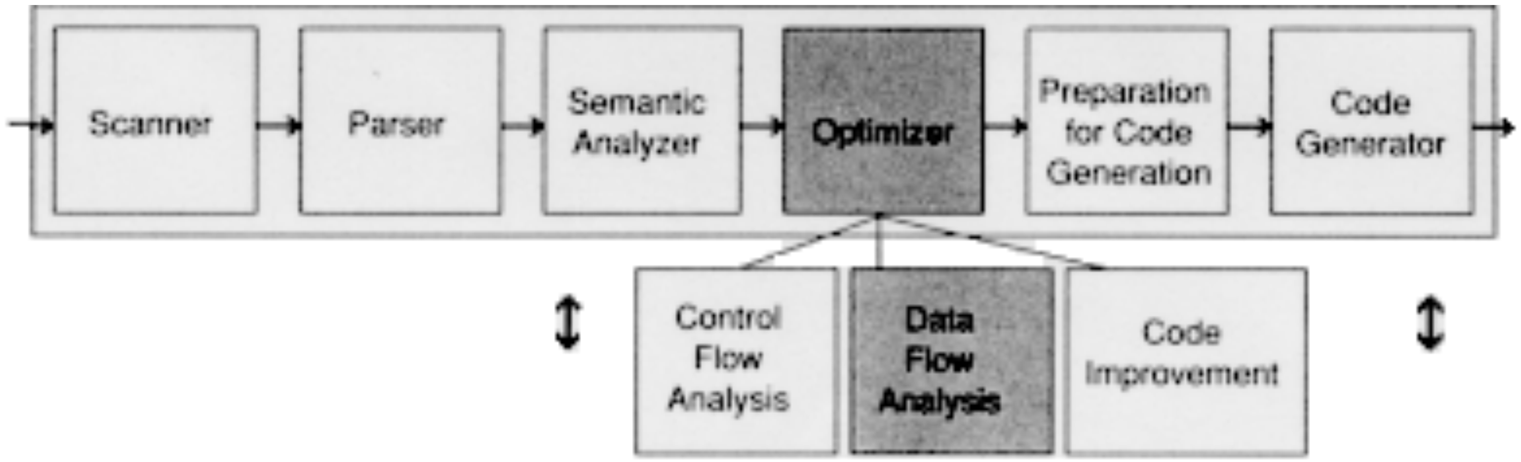
Dataflow Analyses

Code Optimization in Compilers



Correctness Above All!

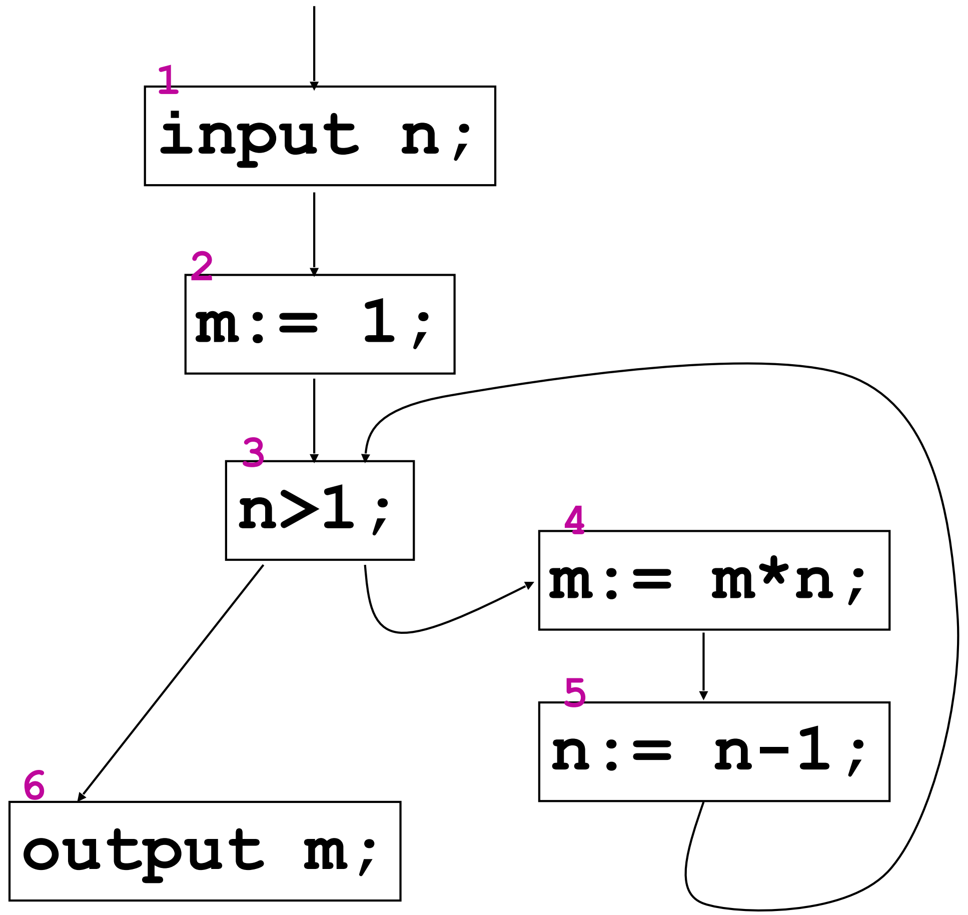
If may seem obvious, but it bears repeating that optimization should not change the correctness of the generated code. Transforming the code to something that runs faster but incorrectly is of little value. It is expected that the unoptimized and optimized variants give the same output for all inputs. This may not hold for an incorrectly written program (e.g., one that uses an uninitialized variable).

Control flow graph

- Program commands are encoded by nodes in a control flow graph
- If a command S may be directly followed by a command T then the control flow graph must include a direct arc from the node encoding S to the node encoding T

Example

```
[ input n; ]1
[ m:= 1; ]2
  [ while n>1 do ]3
[ m:= m * n; ]4
    [ n:= n - 1; ]5
[ output m; ]6
```



Data-Flow analyses

We will see data-flow analyses:

-Liveness analysis

- Reaching definitions analysis

-Available Expressions analysis

Liveness or Live Variables Analysis

- We need to translate the source program in the intermediate representation IR that can use a **large (potentially unbounded) number of registers**.

- but the program will be executed by a processor with a (finite and) **small number of registers**

- Two variables **a** and **b** can be stored on the same register when it turns out that **a** and **b** are **never simultaneously "used"**

IR: Three Address Code

Three-address instruction has at most three operands and is typically a combination of an assignment and a binary operator.

For example: t1 := t2 + t3.

The name derives from the use of three operands in these statements even though instructions with fewer operands may occur.

IR: Three Address Code example

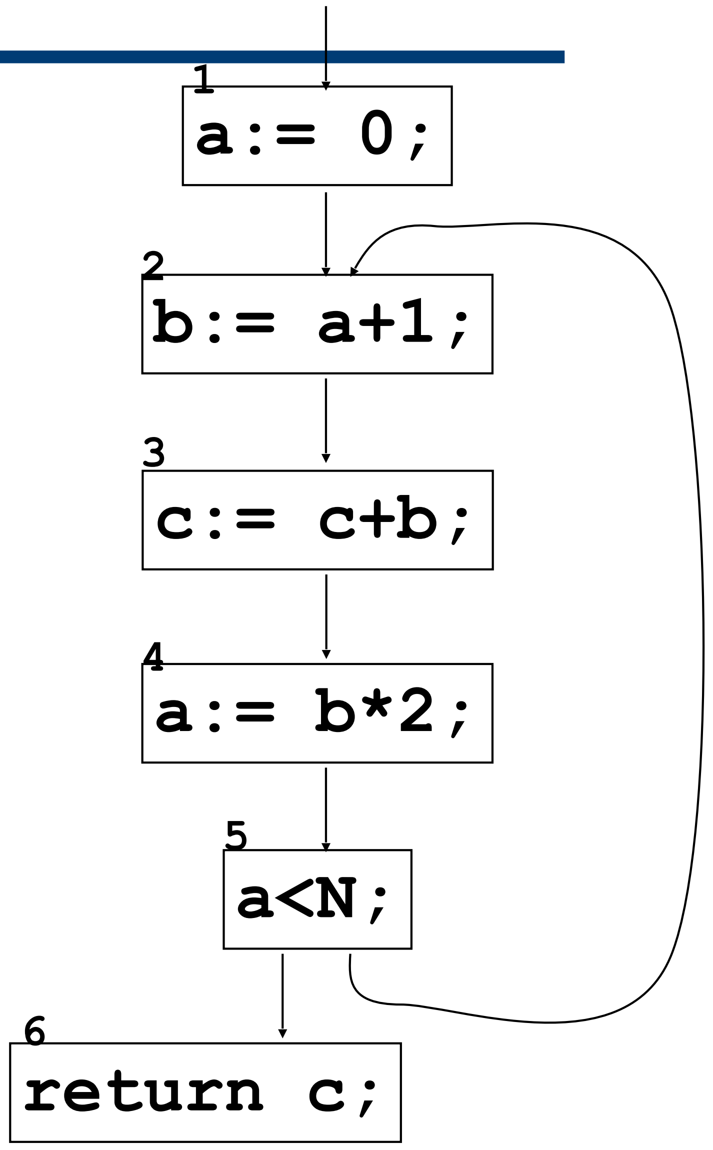
```
# Calculate one solution to the
[[quadratic equation]].
x = (-b + sqrt(b^2 - 4*a*c)) / (2*a)
```

```
t1 := b * b
t2 := 4 * a
t3 := t2 * c
t4 := t1 - t3
t5 := sqrt(t4)
t6 := 0 - b
t7 := t5 + t6
t8 := 2 * a
t9 := t7 / t8
x := t9
```

Example

```
a = 0;
do {
   b = a+1;
   c += b;
   a = b*2;
}
while (a<N);
return c;
```

We may observe that there is not need for two distinct variables **an** and **b** we could use a unique variable, because **a** and **b** are never simultaneously used.

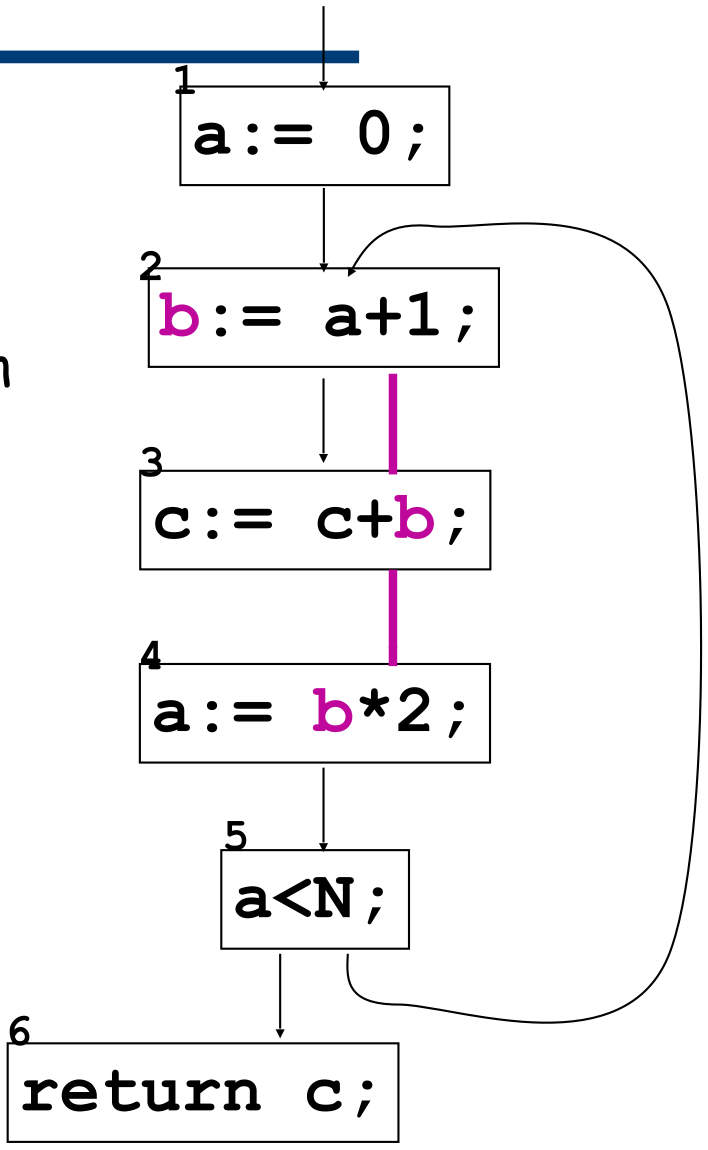


Live Variables Analysis

- A compiler needs to analyze programs in IR in order to find out which variables are simultaneously used
- A variable X is live at the exit of a command C if X stores a value which will be actually used in the future, that is, X will be used as R-value with no previous use as L-value
- A variable X which is not live at the exit of C is also called dead (this information can be used for dead code elimination)
- This is an undecidable property

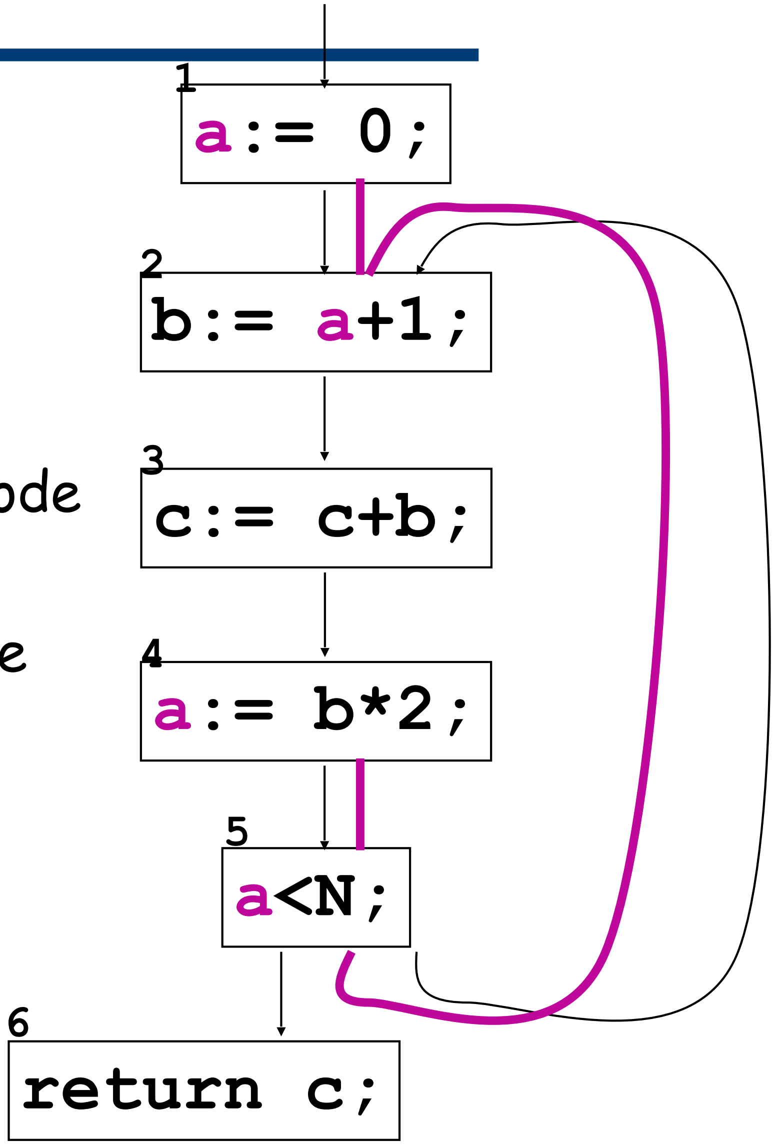
Back to the example

- A variable X is live when it stores a value which will be later used with no prior assignment to X
- The "last" use of the variable **b** as r-value is in command 4
- The variable **b** is used in command 4: it is therefore live along the arc 3 → 4
- Command 3 does not assign **b**, hence **b** is live along 2 → 3
- Command 2 assigns **b**. This means that the value of **b** along 1 → 2 will not be used later
- Thus, the "live range" of **b** turns out to be: {2 → 3, 3 → 4}



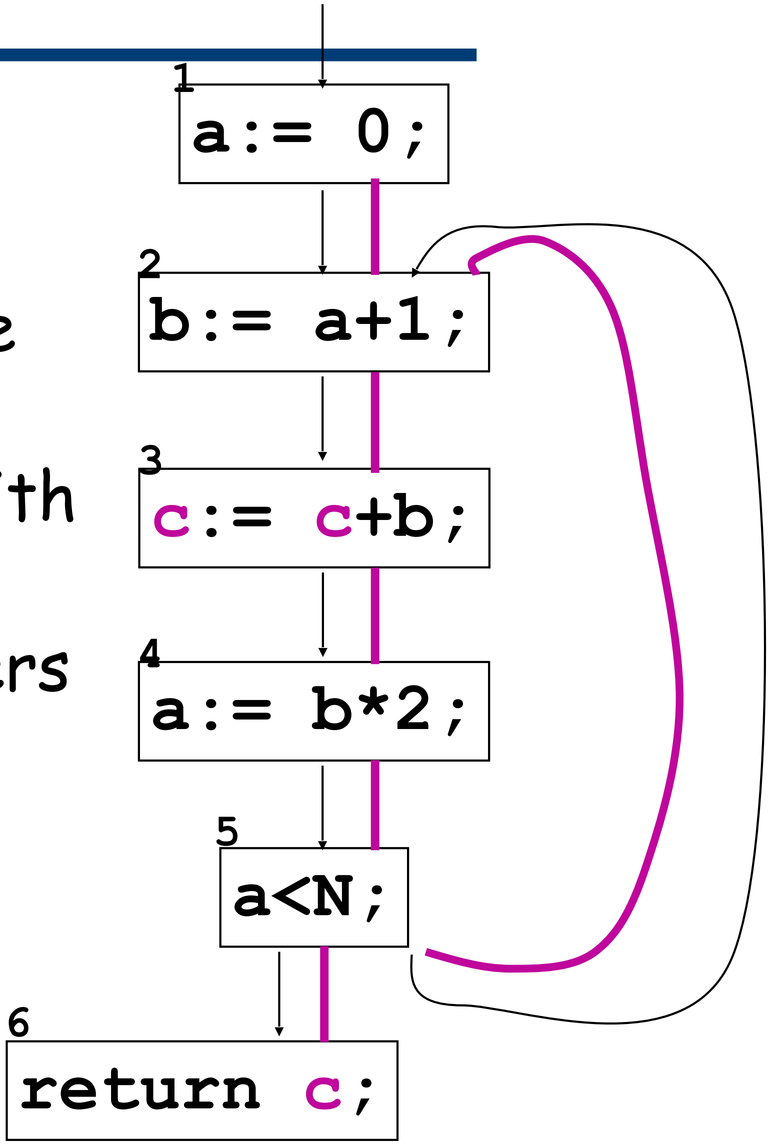
Live variables

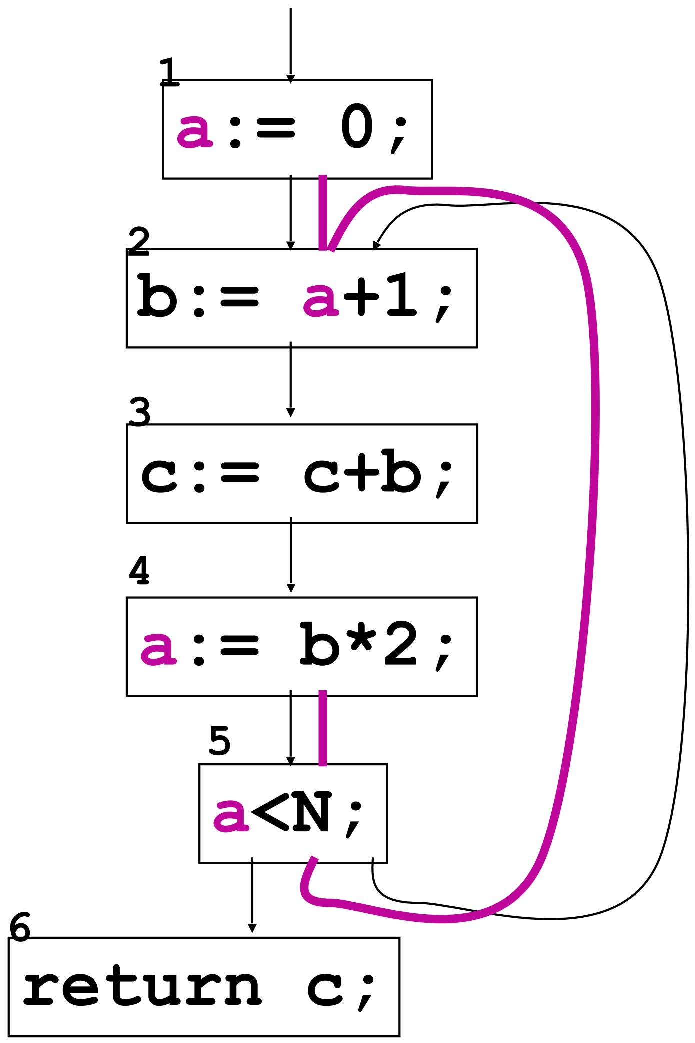
- **a** is live along 4 → 5 and 5 → 2
- **a** is live along 1 → 2
- **a** is not live along 2 → 3 and 3 → 4
- Even if the variable **a** stores a value in node 3, this value will not be later used, since node 4 assigns a new value to the variable **a**.

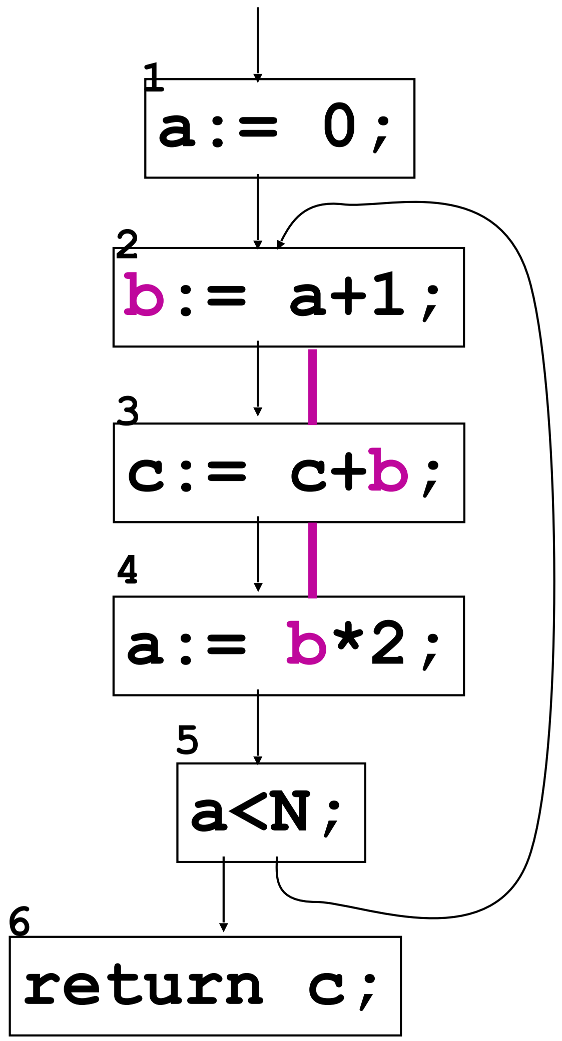


More on live variables

- **c** is live along all the arcs
- By the way: liveness analysis can be exploited to deduce that if **c** is a local variable then **c** will be used with no prior initialization (this information can be used by compilers to raise a warning message)

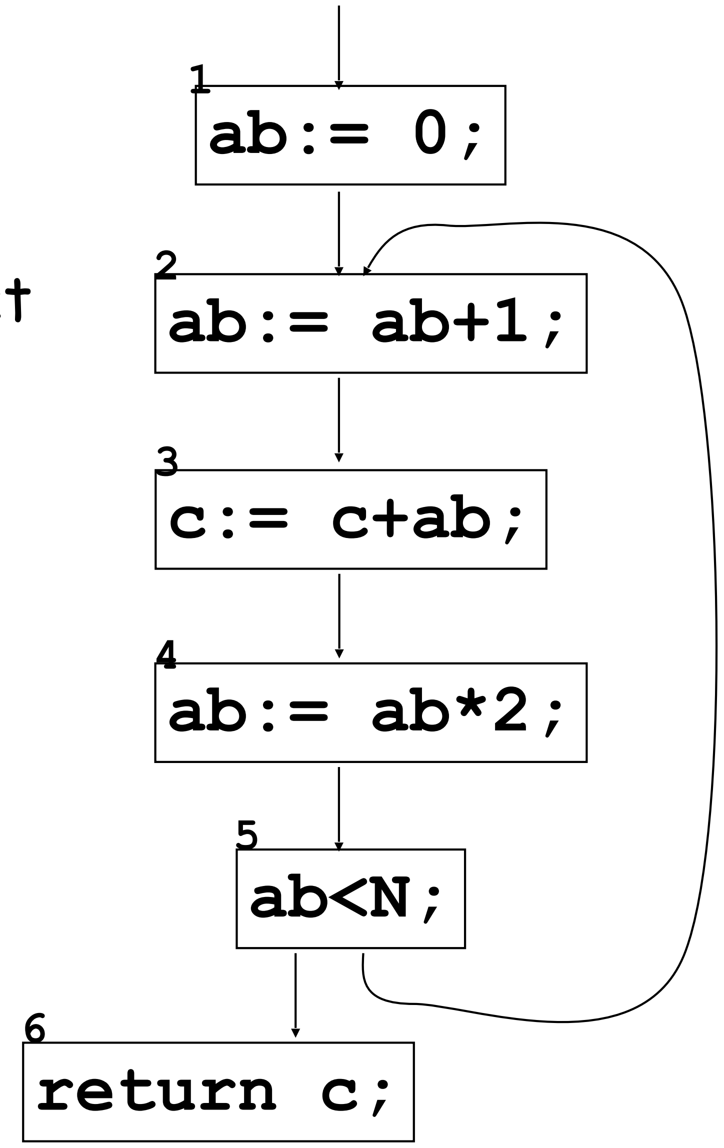






➔ Two registers are enough: variables **a** and **b** will be never simultaneously live along the same arc

Variables **a** and **b** will be never simultaneously live along the same arc. Hence, instead of using two distinct variables **a** and **b** we can correctly employ a single variable **ab**



We need a way to compute live variables

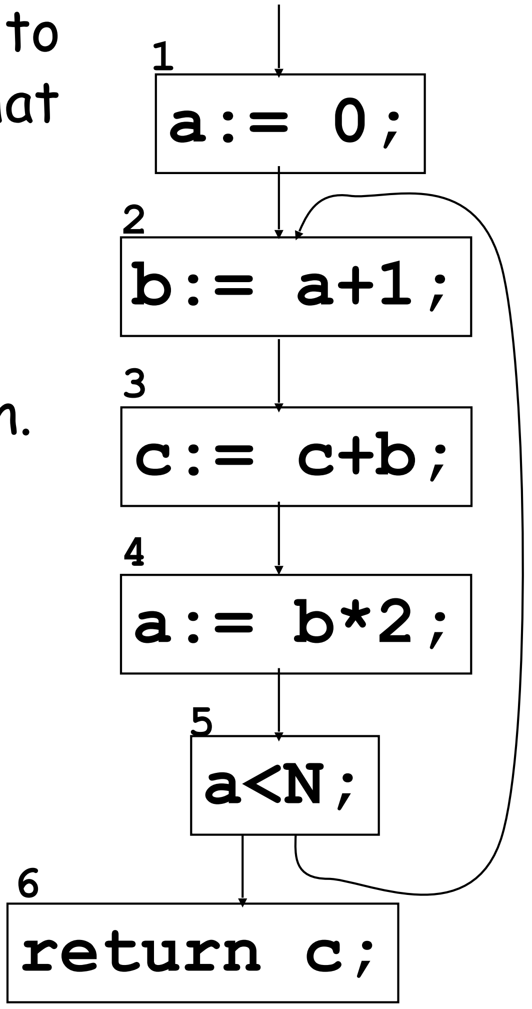
- A CFG has outgoing edges (out-edges) that lead to successor nodes, and ingoing edges (in-edges) that originate from predecessor nodes.

- pre[n] and post[n] denote, respectively, the predecessor and successor nodes of some node n.

- As an example, in this CFG:
  - 2 and 6 are the out-edges of node 5 because

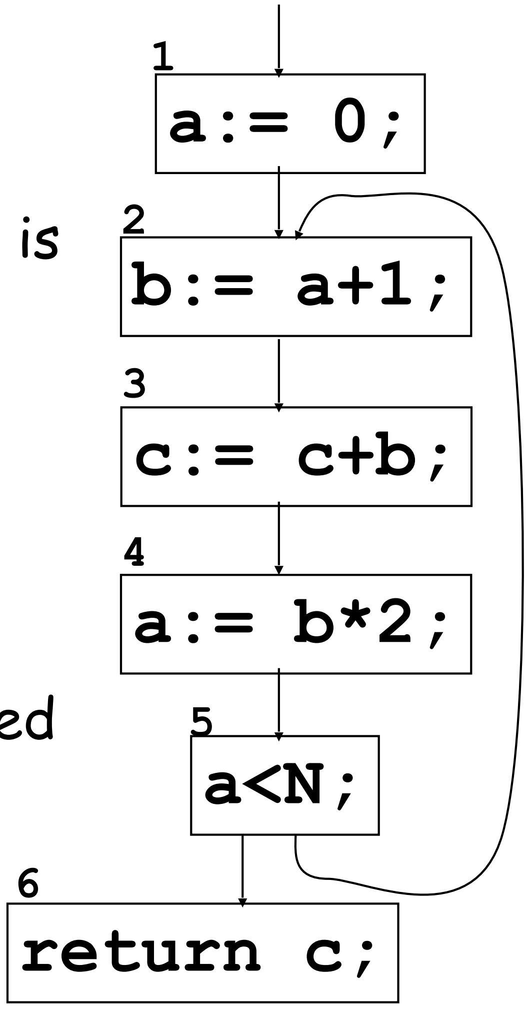
    5 → 6 and 5 → 2 are the out-edges of 5
  - 1 and 5 are the in-edges of node 2 since

    5 → 2 and 1 → 2 are the in-edges of 2
  - pre[2]={1,5}; post[5]={2,6}.



Notation

- An assignment to some variable (a use of the variable as L-value) is called a definition of the variable
- A use of some variable as R-value in a command is called a use of this variable
- def[n] denotes the set of variables that are defined in the node n
- use[n] denotes the set of variables that are used in the node n
- As an example, in this CFG:
  - def[3]={**c**}, def[5]=$\emptyset$
  - use[3]={**b**,**c**}, use[5]={**a**}

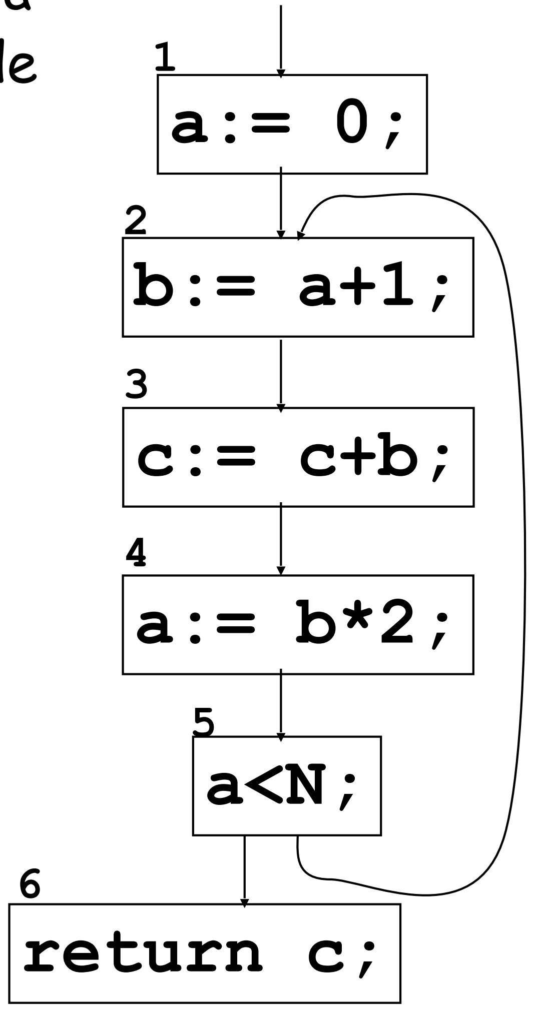


Formalization

- A variable x is live along an arc e→f if there exists a (real) execution path P from the node e to some node n such that:
  - e→f is the first arc of such path P
  - x ∈use[n]
  - for any node n'≠e and n'≠n in the path P, x∉def[n']

- A variable x is live-out in some node n if x is live along some (i.e., at least one) out-edge of n

- A variable **x** is live-in in some node n if **x** is live along **any** in-edge of n



Example

As an example, in this CFG:

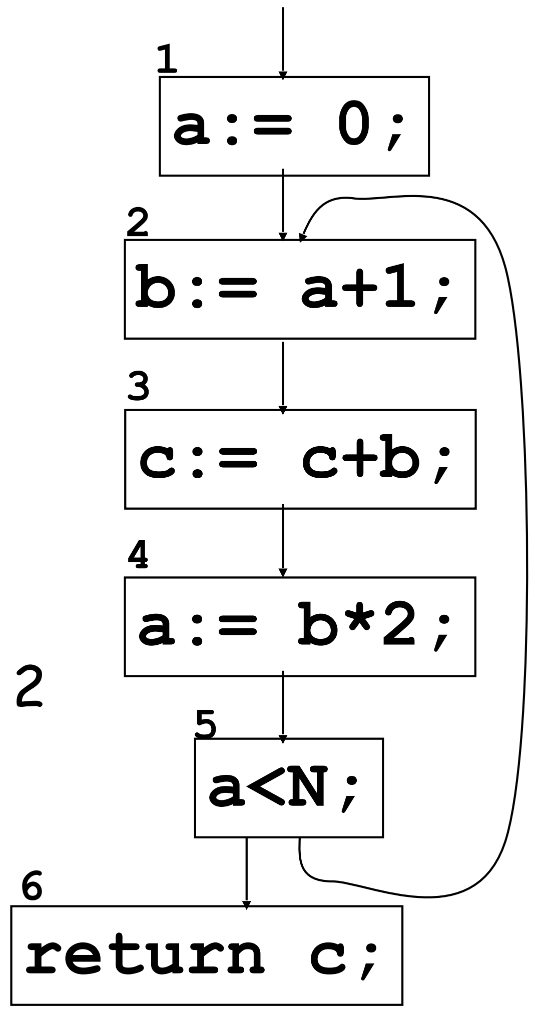
a is live along $1 \rightarrow 2$, $4 \rightarrow 5$ and $5 \rightarrow 2$

b is live along $2 \rightarrow 3$, $3 \rightarrow 4$

c is live along any arc

a is live-in in node 2, while it is not live-out in node 2

a is live-out in node 5



Computing Liveness

Let us define the following notation:

in[n] is the set of variables that the static analysis determines to be live-in at node n

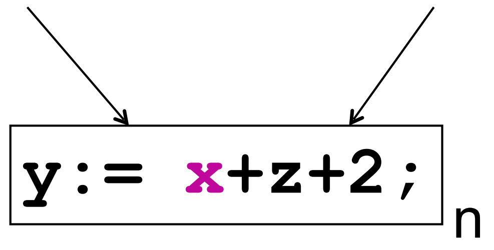
out[n] is the set of variables that the static analysis determines to be live-out at node n

Computing Liveness

n node of the CFG

Liveness information: the sets in[n] and out[n] is computed as an over-approximation in the following way

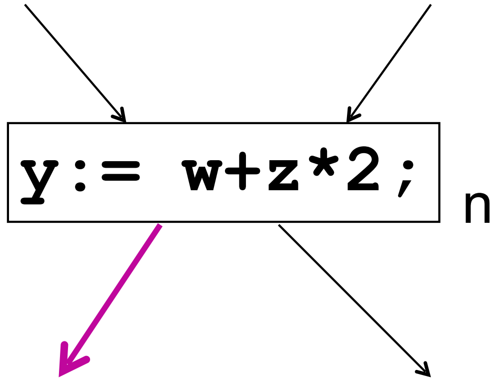
1. If a variable x ∈ use[n] then x is live-in in node n.
   In other terms, if a node n uses a variable x as R-value then this variable x is live along each arc that enters into n.



in[n] ⊇ use[n]

Computing Liveness

2. If a variable x is live-out in a node n and x ∉ def[n] then the variable x is also live-in in this node n.
   If a variable x is live for some arc that leaves a node n and x is not assigned in n then x is live for all the arcs that enter in n

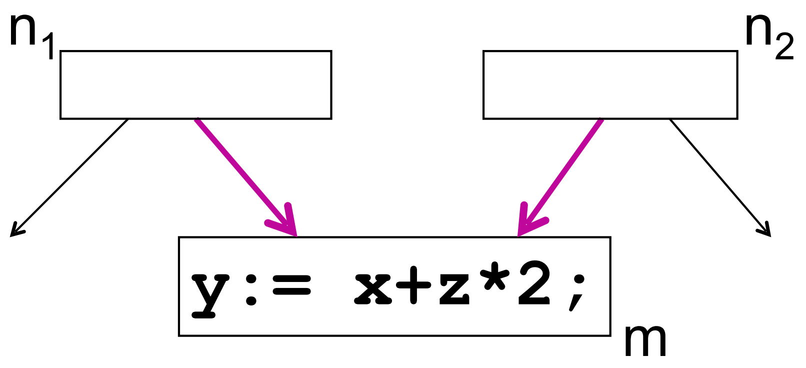


$$in[n] \supseteq out[n] - def[n]$$

Computing Liveness

3. If a variable x is live-in in a node m then x is live-out for all the nodes n such that m∈post[n].

   This is clearly correct by definition.



$$out[n_1] \supseteq \cup\{in[m] \mid m \in post[n_1]\}$$

$$out[n_2] \supseteq \cup\{in[m] \mid m \in post[n_2]\}$$

Dataflow Equations

The previous three rules of liveness analysis can be thus formalized by two equations for each node n:

1. $in[n] = use[n] \cup (out[n] - def[n])$ (rules 1 and 2)

2. $out[n] = \cup\{in[m] \mid m \in post[n]\}$ (rule 3)

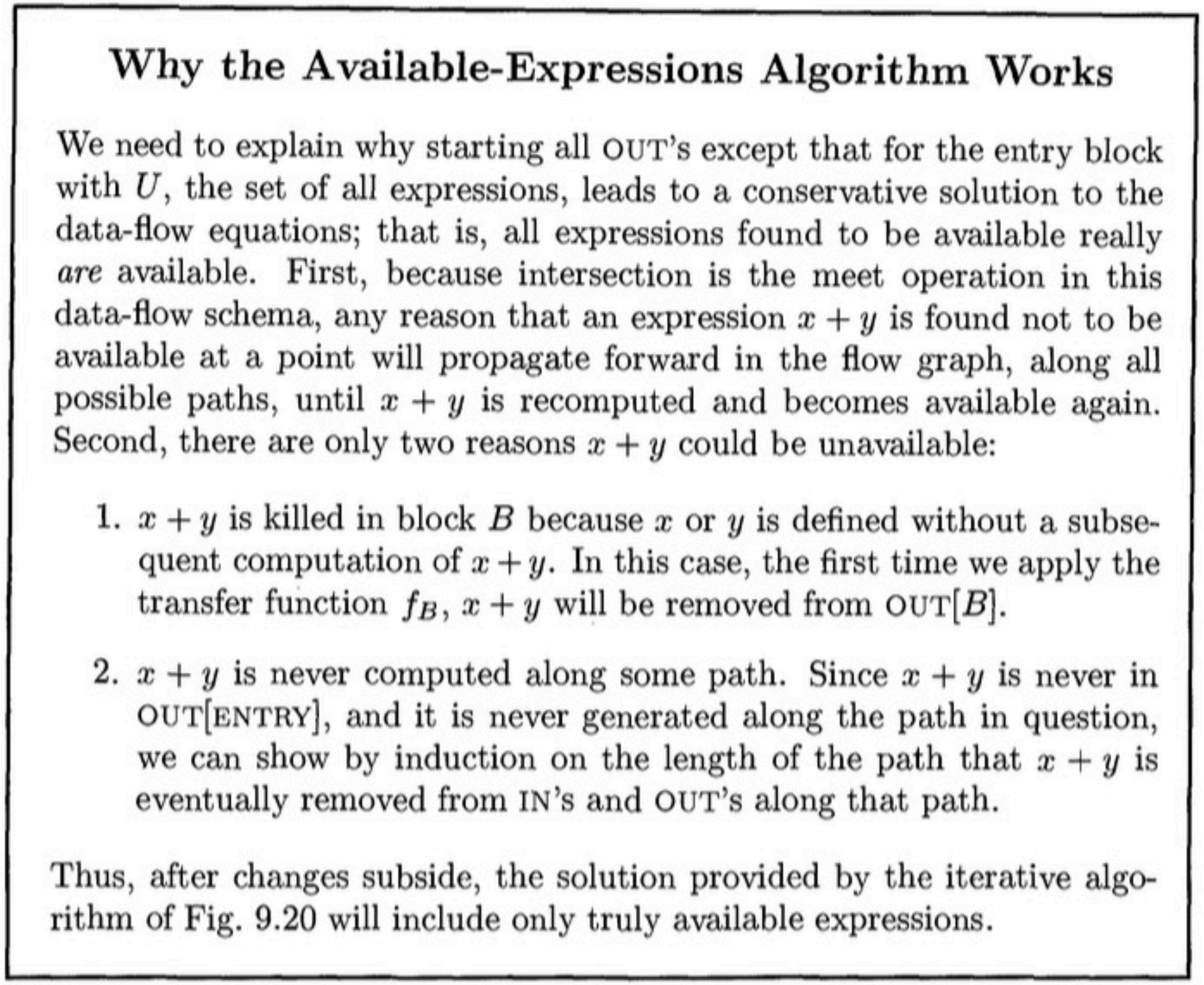
Correctness of Liveness

This definition of liveness analysis in[n] and out[n] is correct:
If x is concretely live-in (live-out) in some node n then the static analysis will detect that $x \in$ in[n] ($x \in$ out[n]):

$$\text{in}[n] \supseteq \text{live-in}[n]$$
$$\text{out}[n] \supseteq \text{live-out}[n]$$

In other terms, no actually live variable is neglected by liveness analysis.

Correctness in Dragon Book



Why the Available-Expressions Algorithm Works

We need to explain why starting all OUT's except that for the entry block with U, the set of all expressions, leads to a conservative solution to the data-flow equations; that is, all expressions found to be available really *are* available. First, because intersection is the meet operation in this data-flow schema, any reason that an expression $x + y$ is found not to be available at a point will propagate forward in the flow graph, along all possible paths, until $x + y$ is recomputed and becomes available again. Second, there are only two reasons $x + y$ could be unavailable:

1. $x + y$ is killed in block B because x or y is defined without a subsequent computation of $x + y$. In this case, the first time we apply the transfer function f_B, $x + y$ will be removed from OUT[B].

2. $x + y$ is never computed along some path. Since $x + y$ is never in OUT[ENTRY], and it is never generated along the path in question, we can show by induction on the length of the path that $x + y$ is eventually removed from IN's and OUT's along that path.

Thus, after changes subside, the solution provided by the iterative algorithm of Fig. 9.20 will include only truly available expressions.

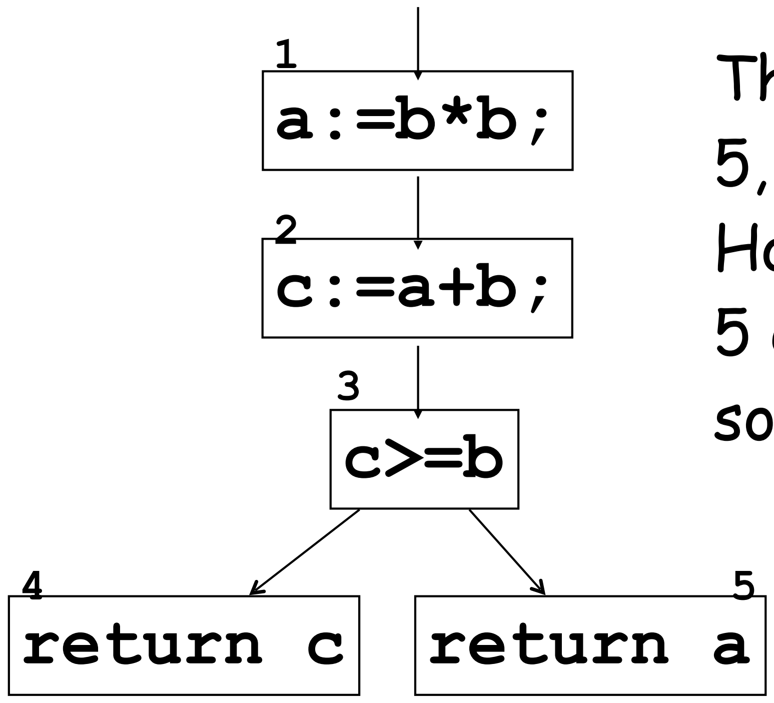
Computing Liveness

Liveness analysis is approximate:

it assumes that each path of the CFG is a feasible path while this hypothesis is obviously not true

Computing Liveness

Liveness analysis is approximate: it assumes that each path of the CFG actually is a feasible path while this hypothesis is obviously not true.



The analysis determines that a is live-in in 5, and therefore a is live-out in 3. However, no real execution path from 3 to 5 exists (because b+b*b<b is always false) so that a is not really live when exiting 3!

Least Fixpoint

1. $in[n] = use[n] \cup (out[n] - def[n])$

2. $out[n] = \cup \{in[m] \mid m \in post[n]\}$

Live variable analysis is computed as the least fixpoint of the set of equations {1,2} for the sets in and out of each program point

Correctness tells us that $in[n] \supseteq$ live-in[n] and $out[n] \supseteq$ live-out[n]

Computing the fix point

1. in[n] = use[n] ∪ (out[n] - def[n])
2. out[n] = ∪ {in[m] | m ∈ post[n]}

- Let **Vars** be the finite set of variables that occur in the program P to analyze. Let N be the number of nodes of the CFG of P.

Thus, the map Live:

$(\mathcal{P}(\textbf{Vars}) \times \mathcal{P}(\textbf{Vars}))^N \to (\mathcal{P}(\textbf{Vars}) \times \mathcal{P}(\textbf{Vars}))^N$ defined by

$\mathrm{Live}(\langle in_1, out_1, \ldots, in_N, out_N \rangle) =$

$$\langle use[1] \cup (out_1 - def[1]), \bigcup_{m \in post[1]} in_m, \ldots, use[N] \cup (out_N - def[N]), \bigcup_{m \in post[N]} in_m \rangle$$

is a monotonic (and therefore continuous) function on the finite lattice

$\langle (\mathcal{P}(\textbf{Vars}) \times \mathcal{P}(\textbf{Vars}))^N, \subseteq^{2N} \rangle$ and therefore Live has

a least fixpoint