Compilers

Topic: Data Flow Analysis

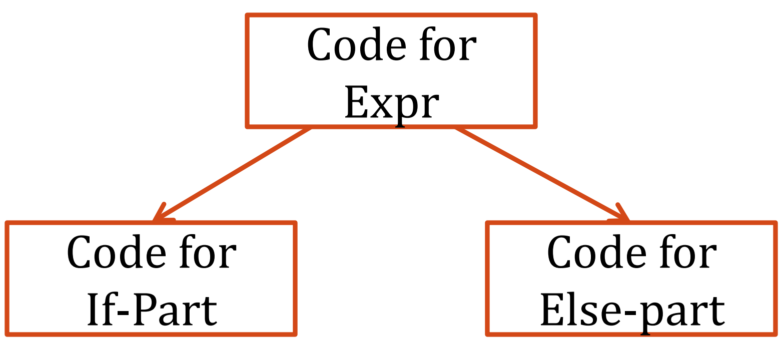
Monsoon 2011, IIIT-H, Suresh Purini

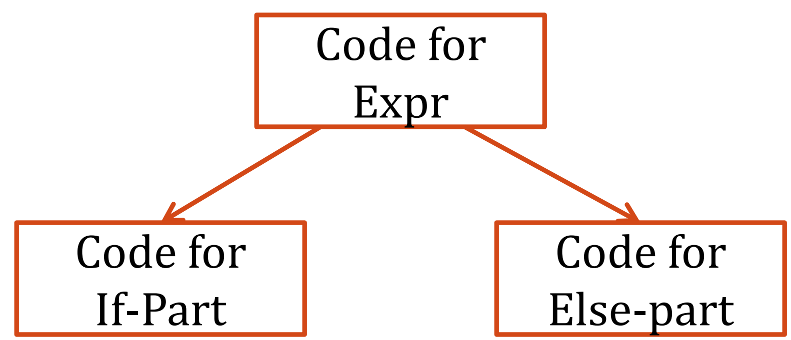
Control Flow Graphs

We shall assume that a Control Flow Graph (CFG)

- Has a special Entry Node
  - Control enters the program through the Entry Node only
- Has a special Exit Node
  - Control leaves the program through the Exit Node only
- Entry and Exit Nodes has no instructions in them

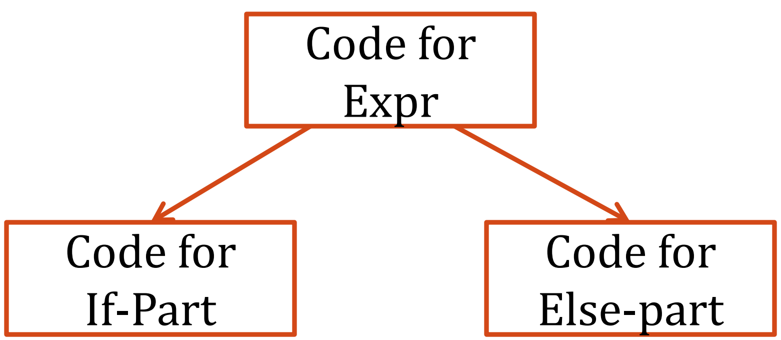
```
if ( expr ) {
    ……….
}
else {
    ……….
}
```

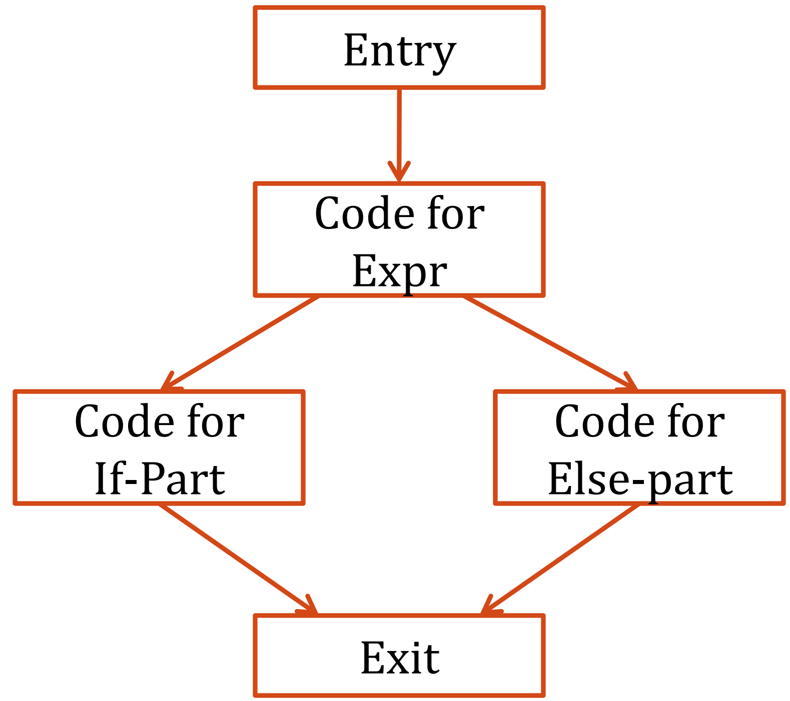




CFGs, Entry and Exit Nodes

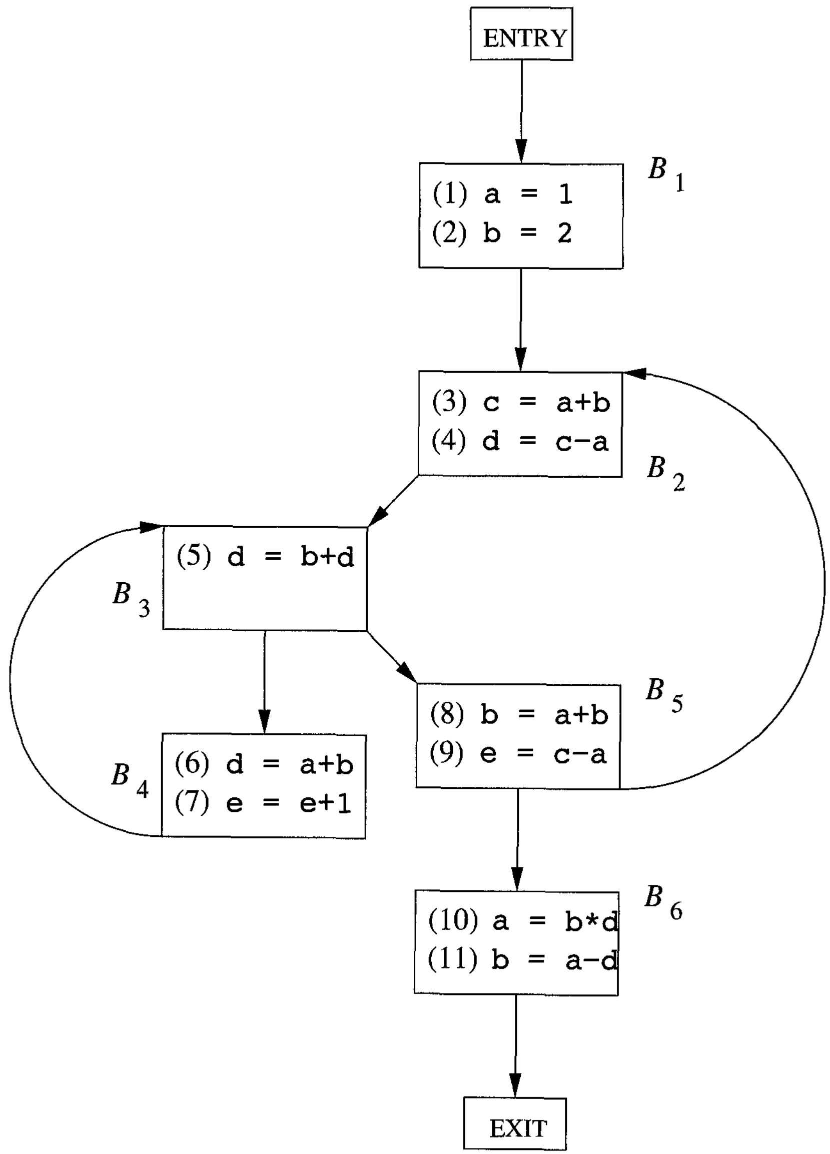
```
if ( expr ) {
    ……….
}
else {
    ……….
}
```





CFGs, Entry and Exit Nodes

Note: All the Instructions in this CFG are numbered for convenience.

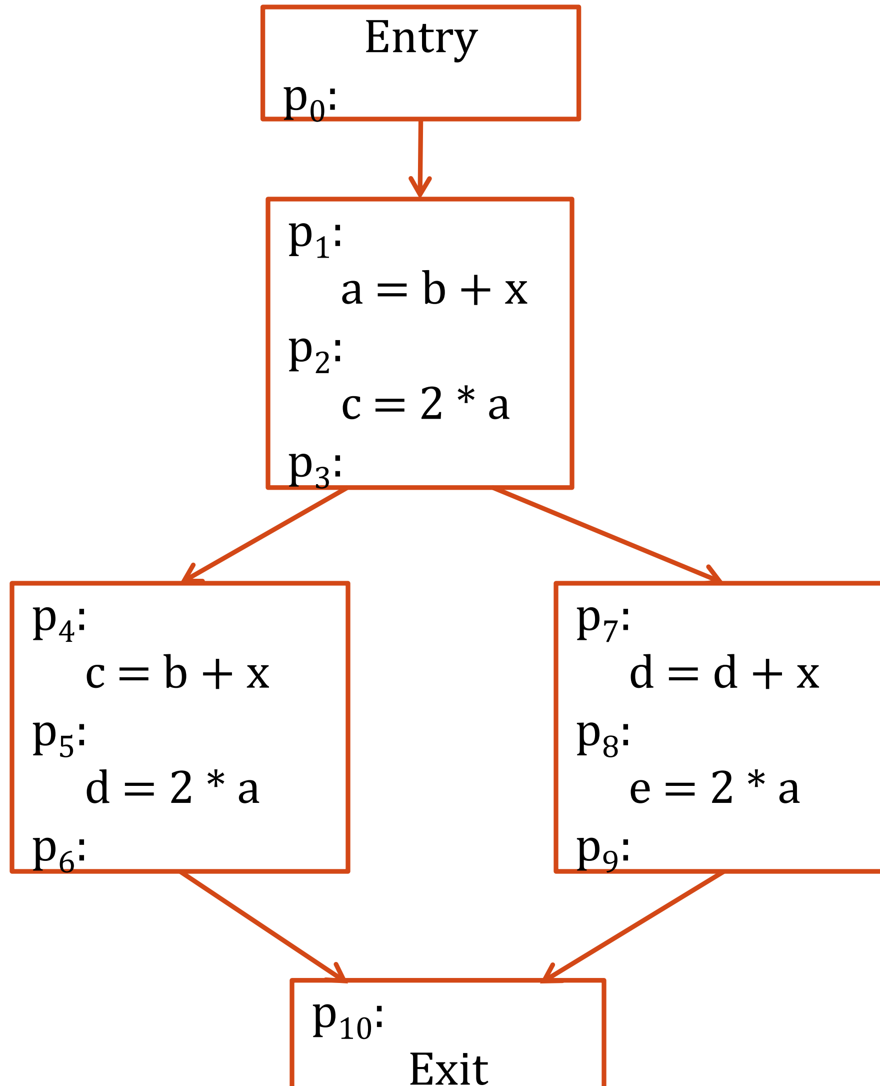


Program Points

- Def: A Program Point is a marker after and before each statement in the CFG of a program.

Remarks

1. Don't confuse between program points and labels in a program.
2. An execution trace is uniquely identified by a sequence of program points.

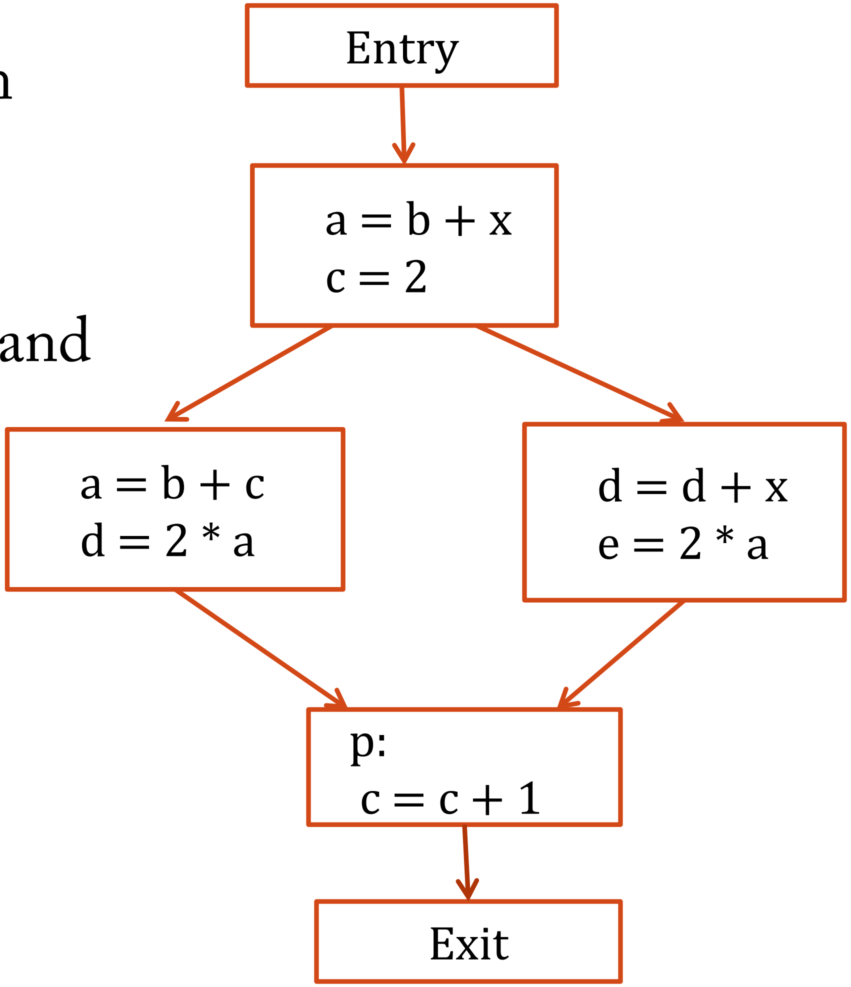


Dynamic State of a Program

1. Initial State: A program when invoked starts in a certain state S_0.
2. As the program progresses through the program points the state of the program evolves (or changes).
   1. Each statement in the program changes the program state from S_i to S_{i+1} according to some rules.
3. When a program reaches a program point p_i , it could be in many possible states. Why?
4. Static Program Analysis Techniques would try to deduce properties for a state which hold at a program point p_i independent of the path taken to reach p_i.

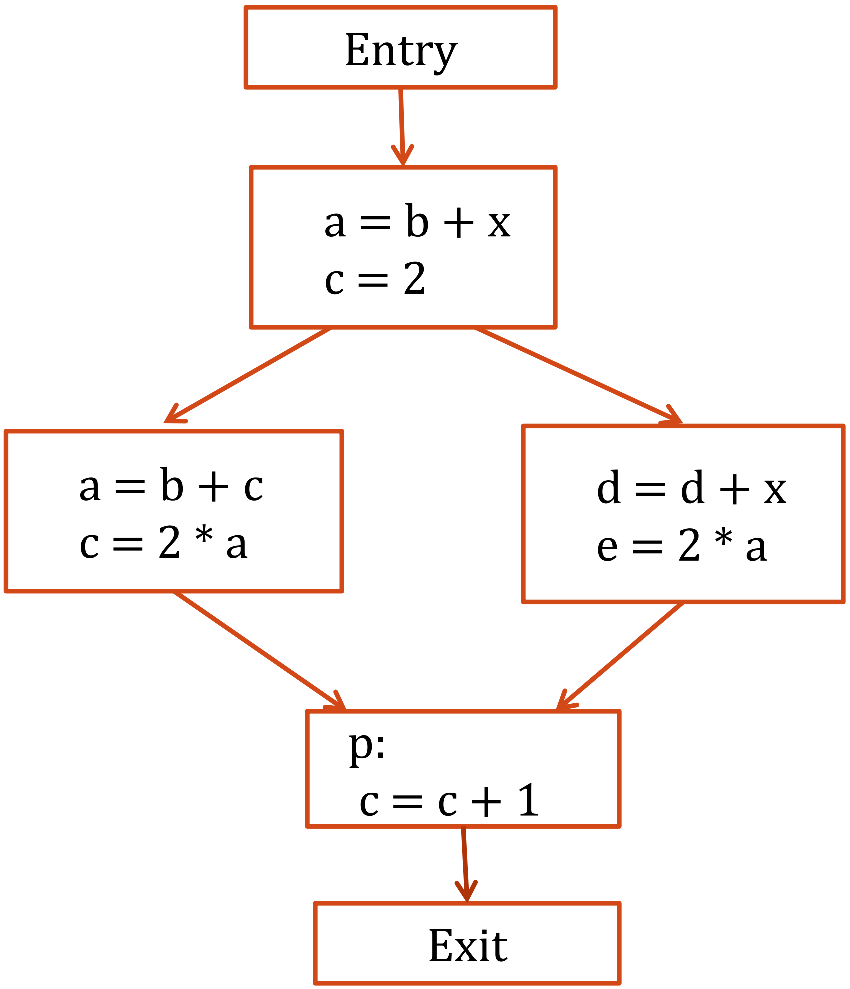
Constant Propagation

- Is the variable c constant at program point p?
- If Yes, we can do Constant Folding and propagate the constant further.



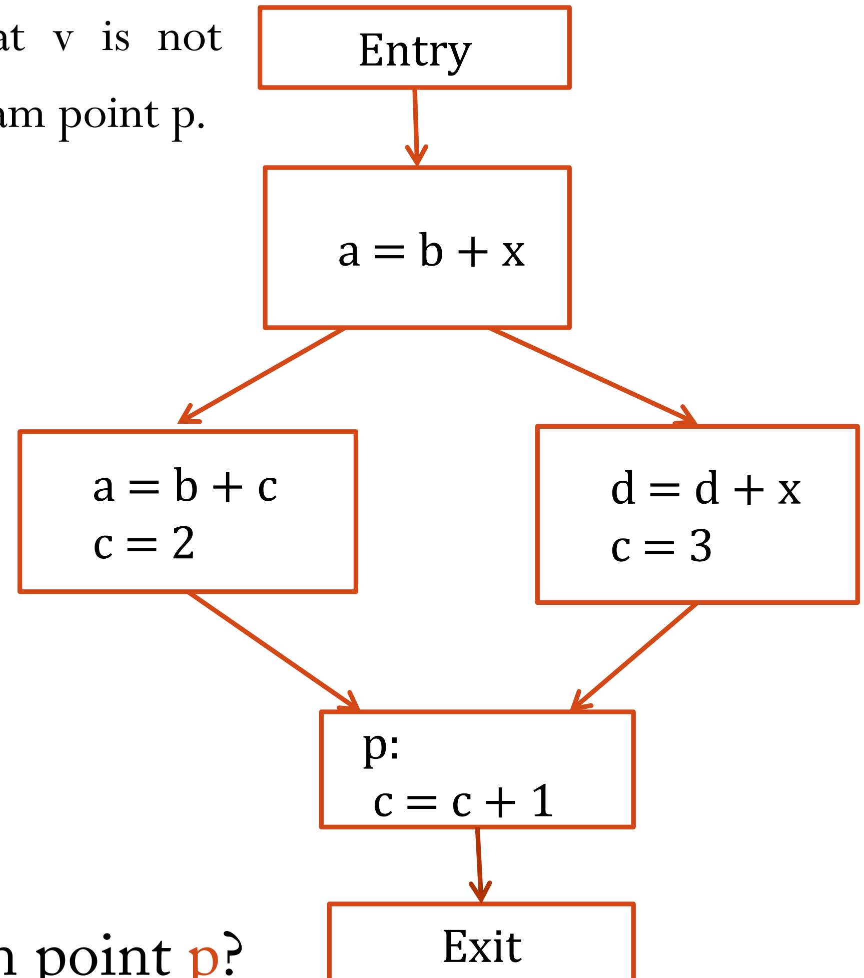
Constant Propagation

- Is the variable c constant at program point p?



Constant Propagation

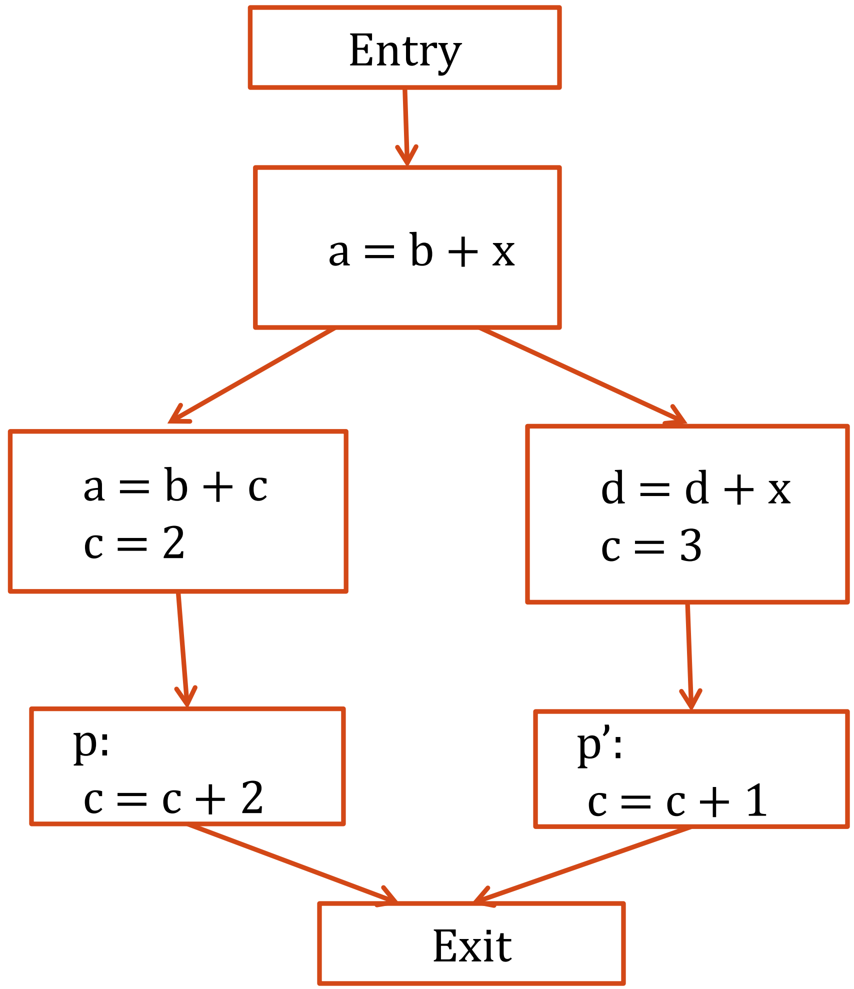
Def: A variable v is said to be holding a constant value c at a program point if along all the paths from Entry to p, there exists a assignment statement "v = c" such that v is not redefined after the statement and before the program point p.



- Is the variable c constant at program point p?

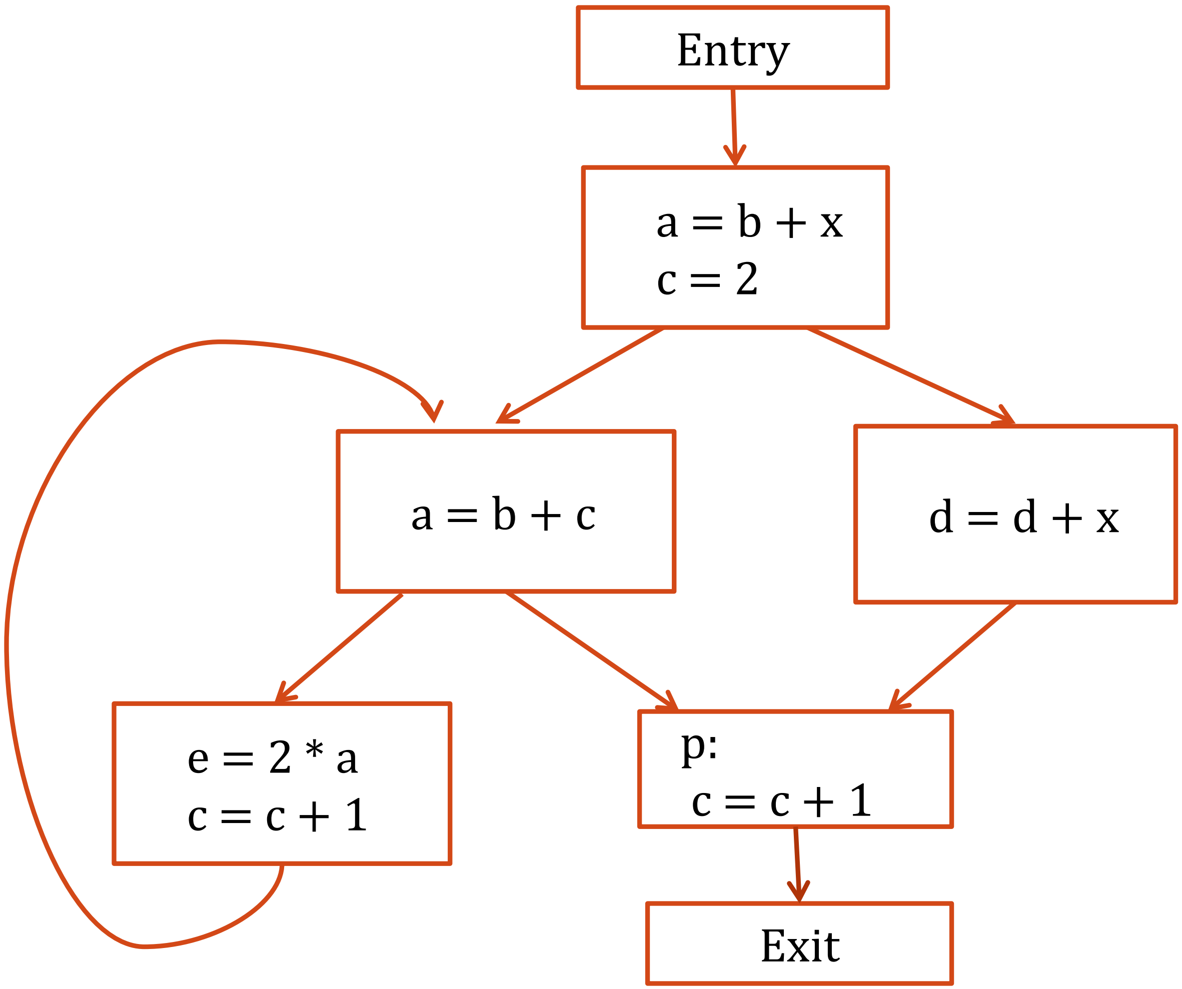
Constant Propagation

- Is the variable c constant at program point p?



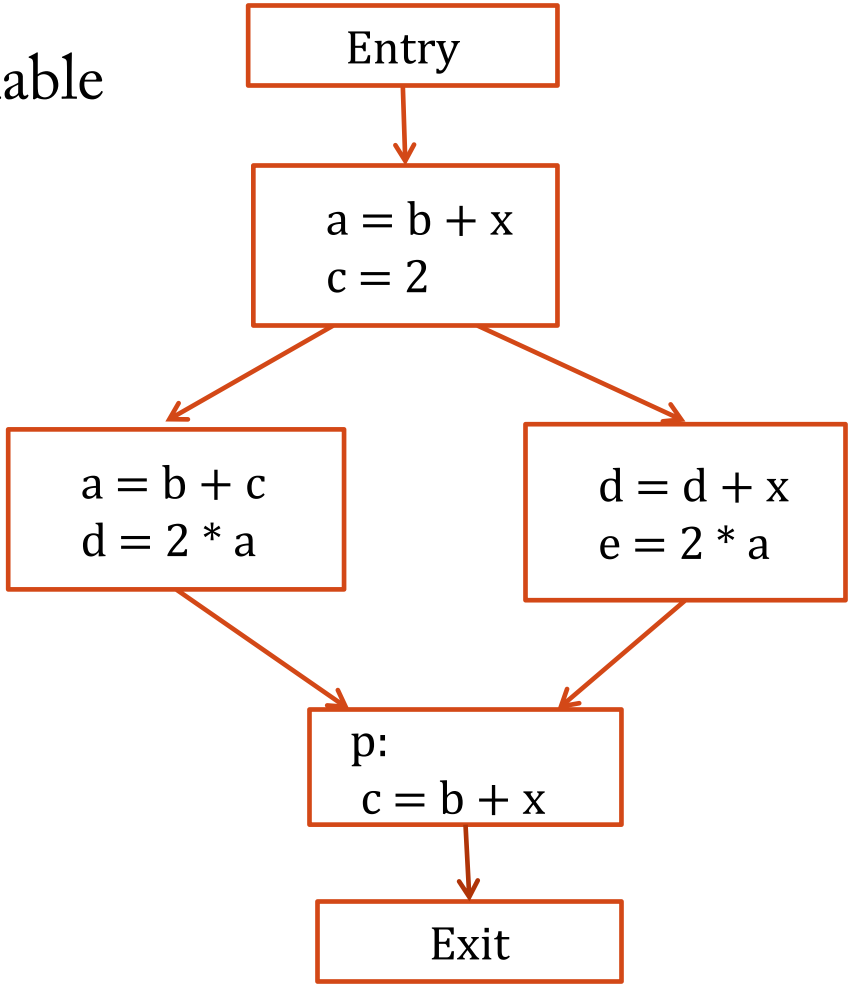
Constant Propagation

- Is the variable c constant at program point p?



Available Expressions

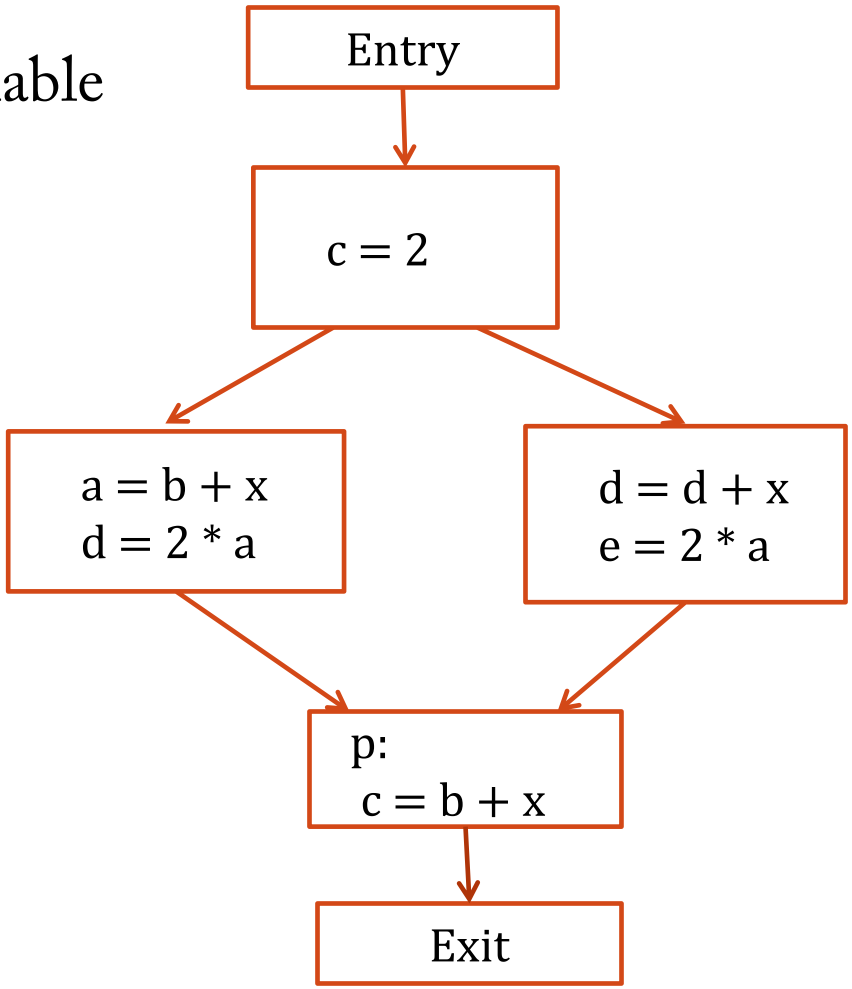
Question: Is the expression "b+x" available at the program point p?



Available Expressions

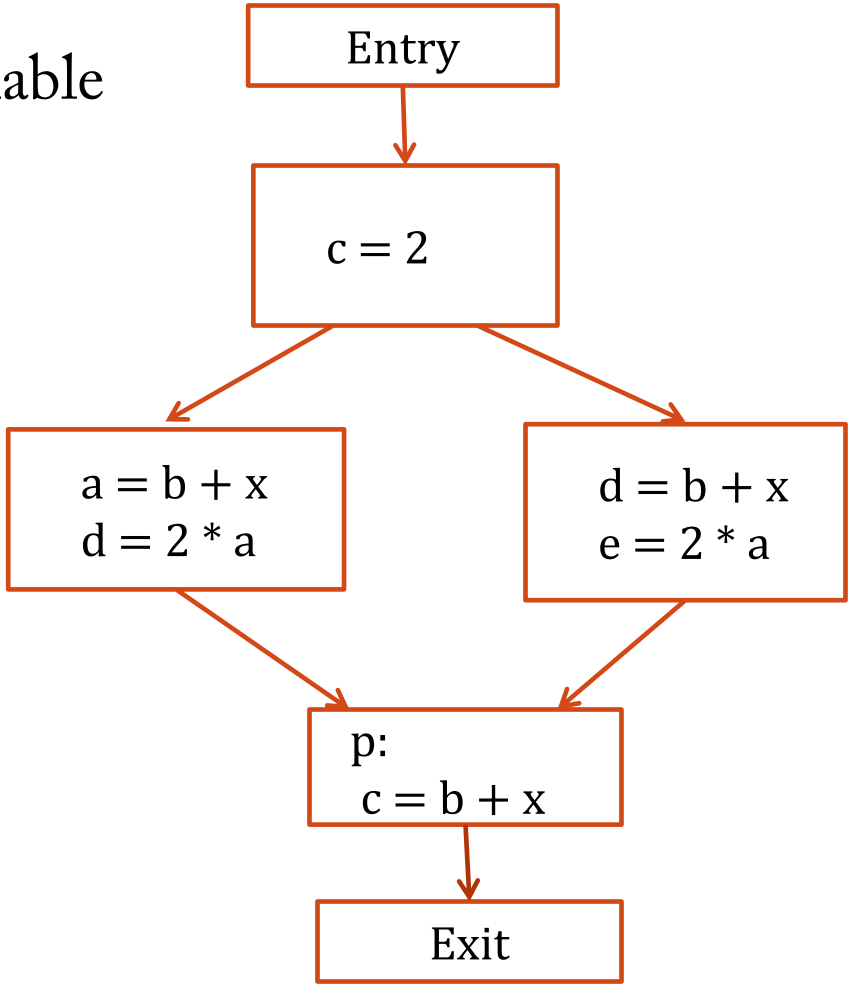
Question: Is the expression "b+x" available

at the program point p?



Available Expressions

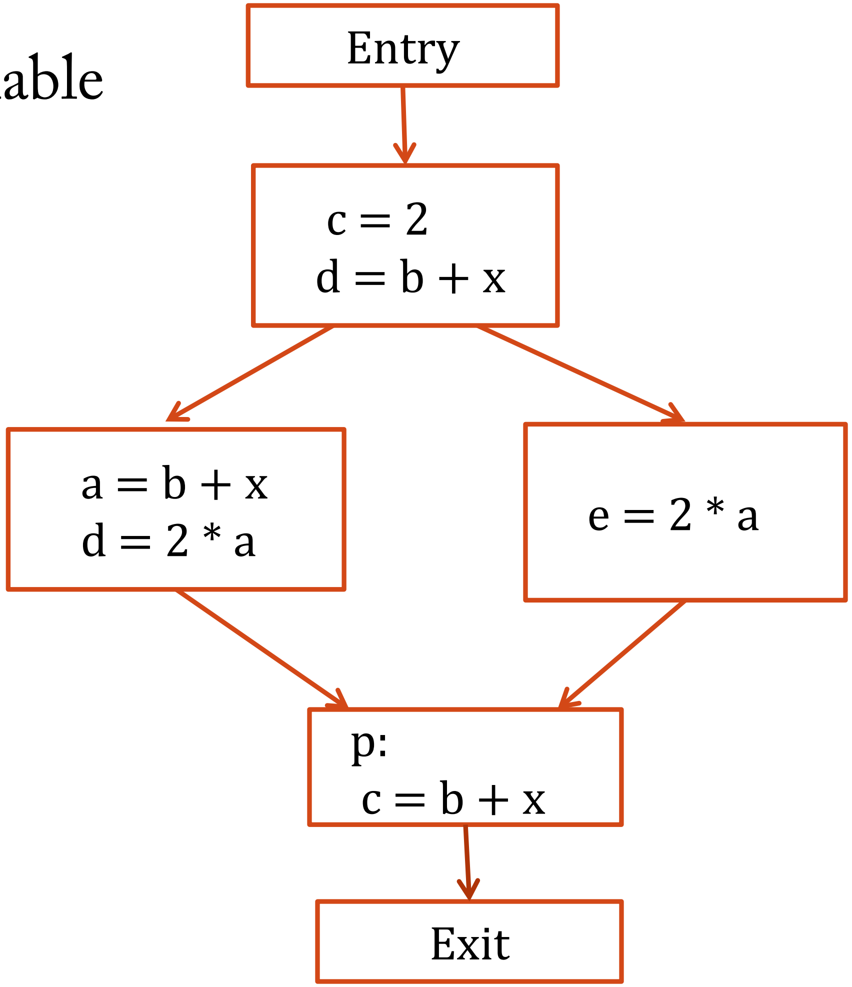
Question: Is the expression "b+x" available at the program point p?



Available Expressions

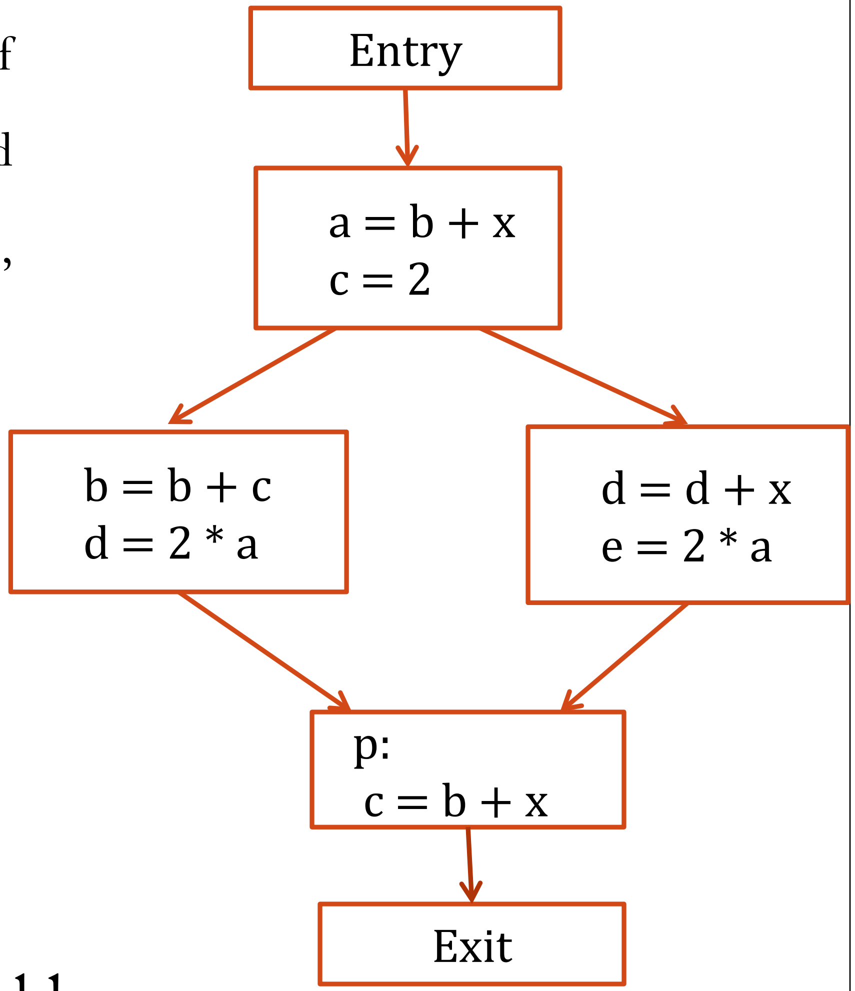
Question: Is the expression "b+x" available

at the program point p?



Available Expressions

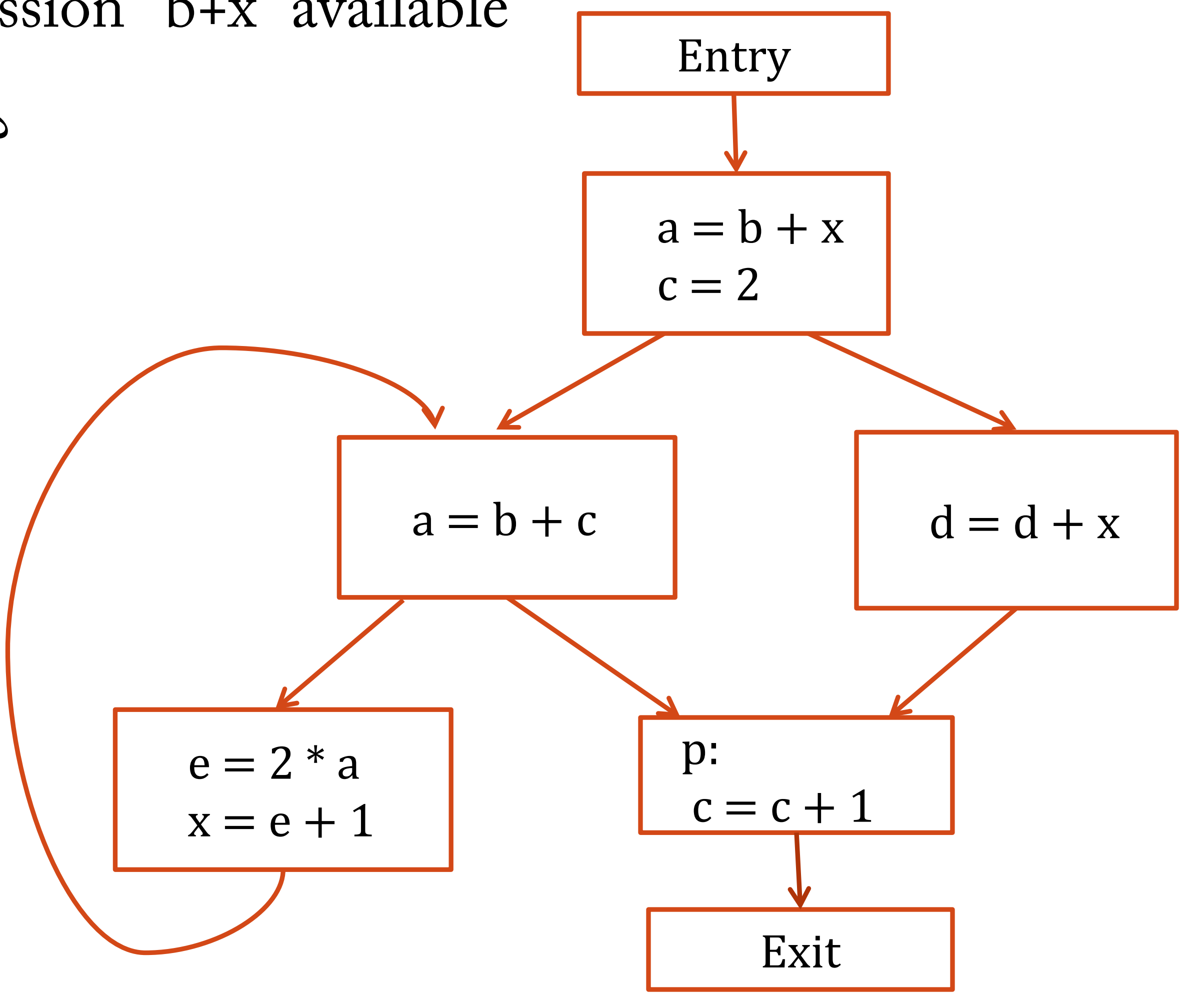
Def: An expression x + y is available at a point p if every path from Entry node to p evaluates x + y, and after the last such evaluation prior to reaching p, there are no subsequent assignments to x or y.



Question: Is the expression "b+x" available at the program point p?

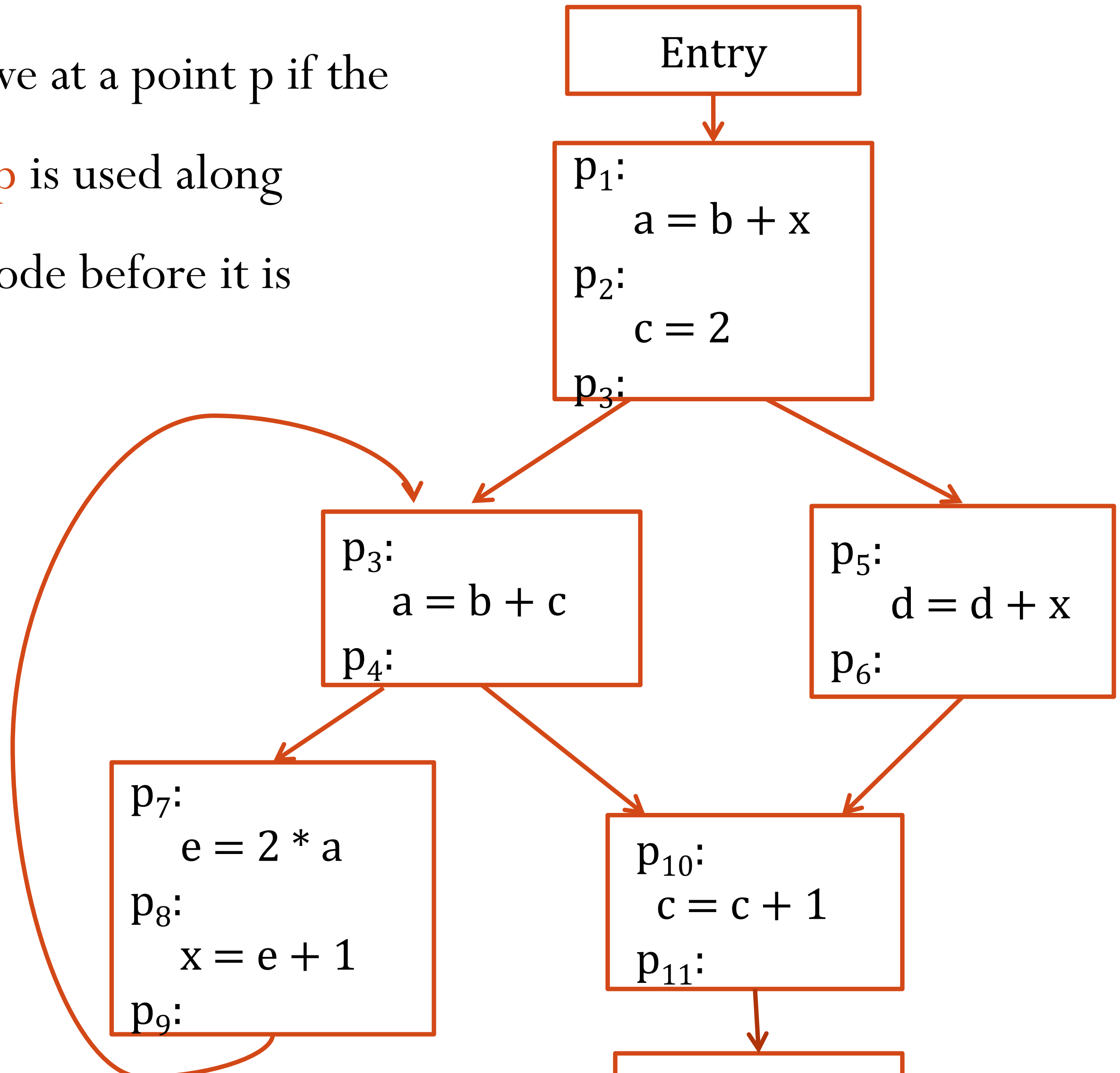
Available Expressions

- Question: Is the expression "b+x" available at the program point p?

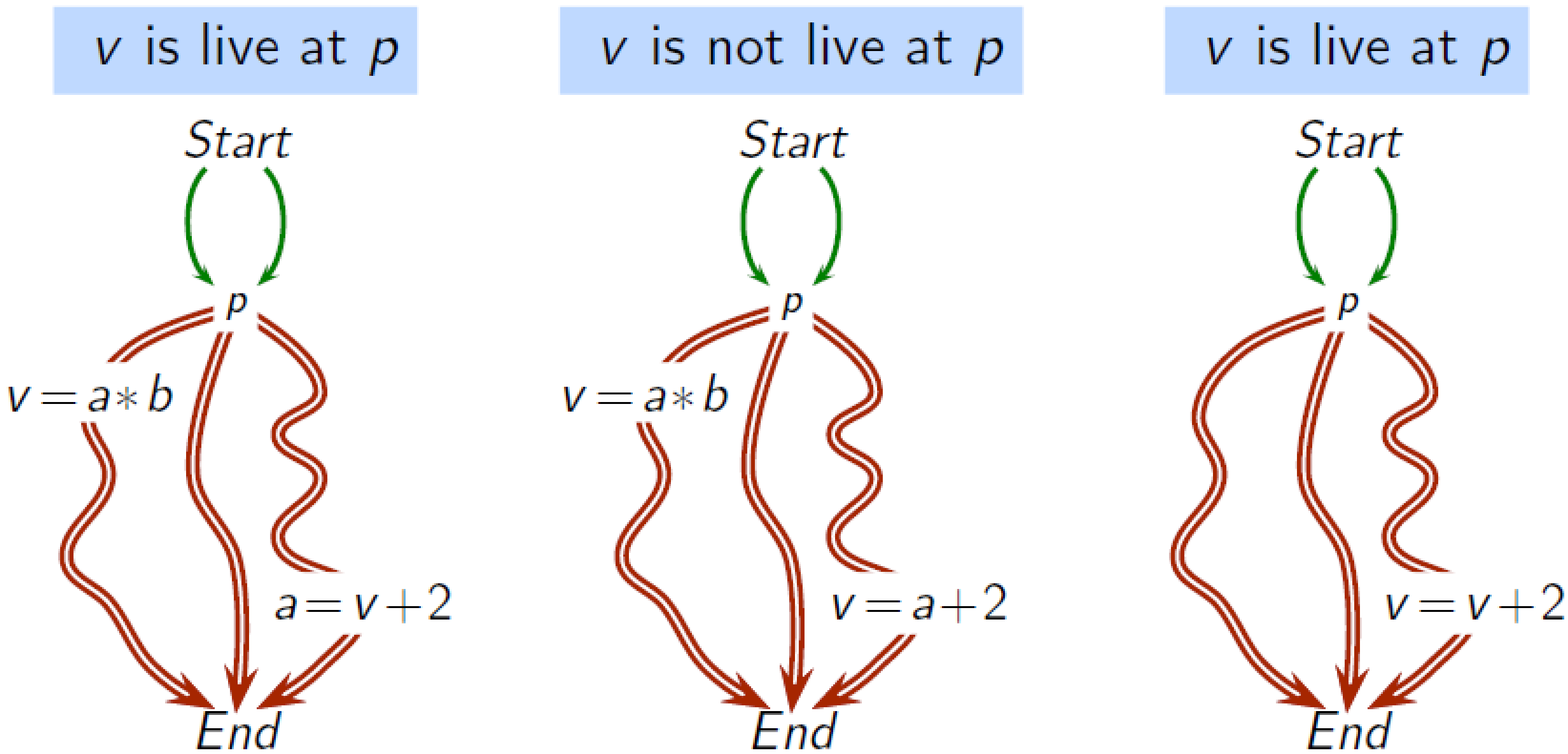


Live Variables

Def: A variable v is said to be live at a point p if the value of the variable v at point p is used along some path from p to the Exit node before it is being redefined.



Live Variables



Source: Uday Khedkar's Slides

Live Variable Analysis

For a basic block B, define:

- IN[B] – Set of variables that are live at the beginning of the basic block.
- OUT[B] – Set of variables that are live at the end of the basic block.

Questions: Given B alone

1. Can we compute IN[B] and OUT[B]?
2. Can we compute OUT[B] from IN[B]?
3. Can we compute IN[B] from OUT[B]?
4. Given OUT[B] and a program point p in B can we compute the set of live variables at the program point p?

Live Variable Analysis

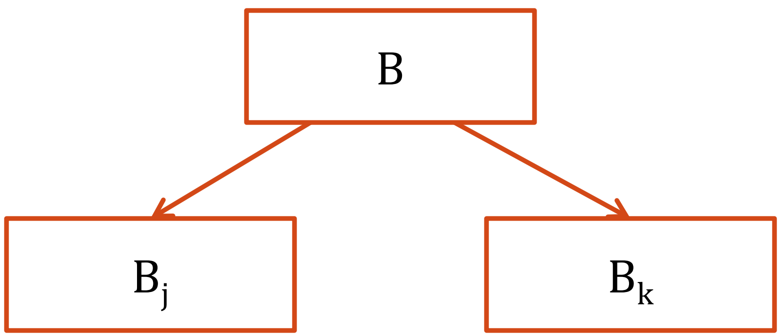
For a Basic Block B

$$IN[B] = GEN[B] \cup (OUT[B] - KILL[B])$$

- GEN[B] = { x | Variable x is used before being defined in B}
- KILL[B] = { x | Variable x is being defined in B }
- Variables in GEN[B] are called upwardly exposed variables.
- Remark: The above equations takes care of straight line sequence of Code.

Live Variable Analysis

- A variable is live at the end of the basic block B if it is live either at the beginning of either the successor basic block B_i or B_j .



$$OUT[B] = \cup_{S \text{ a successor of } B} \; IN[S]$$

Live Variable Analysis

- Equations (or Constraints) defining the Live Variables at the beginning and end of Basic Blocks.
- For each Basic Block B we have

$$IN[B] = GEN[B] \cup (OUT[B] - KILL[B])$$

$$OUT[B] = \cup_{S \text{ a successor of } B} \; IN[S]$$

- In terms of Bit-Vector Representation

$$in_B = gen_B \mid (out_B \,\&\, {\sim}kill_B)$$

$$out_B = in_{S1} \mid \ldots. \mid in_{Sk}$$

- How to solve for the unknowns?

A Brief Detour – Jacobi's Iteration Method

- Given a set of linear equations

$$\boxed{\begin{aligned} &a_1x_1 + a_2x_2 + \cdots + a_nx_n = s_1 \\ &b_1x_1 + b_2x_2 + \cdots + b_nx_n = s_2 \\ &\vdots \\ &d_1x_1 + d_2x_2 + \cdots + d_nx_n = s_n \end{aligned}}$$

- Rearrange the linear equations

$$\boxed{\begin{aligned} &x_1 = \frac{s_1}{a_1} - \frac{a_2}{a_1}x_2^{(0)} - \frac{a_3}{a_1}x_3^{(0)} - \cdots - \frac{a_n}{a_1}x_n^{(0)} \\ &x_2 = \frac{s_2}{b_{12}} - \frac{b_1}{b_2}x_1^{(0)} - \frac{b_3}{b_2}x_3^{(0)} - \cdots - \frac{b_n}{b_2}x_n^{(0)} \\ &\vdots \\ &x_n = \frac{s_n}{a_n} - \frac{d_1}{d_n}x_1^{(0)} - \frac{d_2}{d_n}x_2^{(0)} - \cdots - \frac{d_{n-1}}{d_n}x_{n-1}^{(0)} \end{aligned}}$$

Jacobi's Iteration Method

$$\begin{aligned} 7x_1 + 3x_2 + x_3 &= 18 \\ 2x_1 - 9x_2 + 4x_3 &= 12 \\ x_1 - 4x_2 + 12x_3 &= 6 \end{aligned}$$

$$x_1 = \frac{18}{7} - \frac{3}{7}x_2 - \frac{1}{7}x_3 \quad \Rightarrow \quad x_1 = 2.571 - 0.429x_2 - 0.143x_3$$

$$x_2 = -\frac{12}{9} + \frac{2}{9}x_1 + \frac{4}{9}x_3 \quad \Rightarrow \quad x_2 = -1.333 + 0.222x_1 + 0.444x_3$$

$$x_3 = \frac{6}{12} - \frac{1}{12}x_1 + \frac{4}{12}x_2 \quad \Rightarrow \quad x_3 = 0.500 - 0.083x_1 + 0.333x_2$$

Jacob't Iteration Method

- Iteration 0

$$x_1^{(1)} = 2.571 - 0.429\,(0) - 0.143\,(0) = 2.571$$
$$x_2^{(1)} = -1.333 + 0.222\,(0) + 0.444\,(0) = -1.333$$
$$x_3^{(1)} = 0.500 - 0.083\,(0) + 0.333\,(0) = 0.500$$

- Iteration 1

$$x_1^{(2)} = 2.571 - 0.429\,(-1.333) - 0.143\,(0.500) = 3.071$$
$$x_2^{(2)} = -1.333 + 0.222\,(2.571) + 0.444\,(0.500) = -0.540$$
$$x_3^{(2)} = 0.500 - 0.083\,(2.571) + 0.333\,(-1.333) = -0.159$$

Jacob't Iteration Method

SOLVING LINEAR EQUATION USING THE JACOBI ITERATION METHOD

The estimated results after each iteration are shown as:

Iteration	x(1)	x(2)	x(3)	dx(1)	dx(2)	dx(3)
1	2.57143	-1.33333	.50000	2.57143	-1.33333	.50000
2	3.07143	-.53968	-.15873	.50000	.79365	-.65873
3	2.82540	-.72134	.06415	-.24603	-.18166	.22288
4	2.87141	-.67695	.02410	.04601	.04439	-.04005
5	2.85811	-.68453	.03506	-.01330	-.00757	.01096
6	2.85979	-.68261	.03365	.00168	.00192	-.00142

- What should be the initial estimate? Does any initial estimate works?
- Does the algorithm converge?
- If it converges how fast does it converge?
- What is the initial estimate's impact on the number of iterations required to converge?

Iterative Algorithm for Computing Live Variables

1. IN[EXIT] = { } ;
2. **for** (each basic block B other than Exit) IN[B] = { } ;
3. **while** (changes to any IN occur) {

    for (each basic block B other than Exit) {

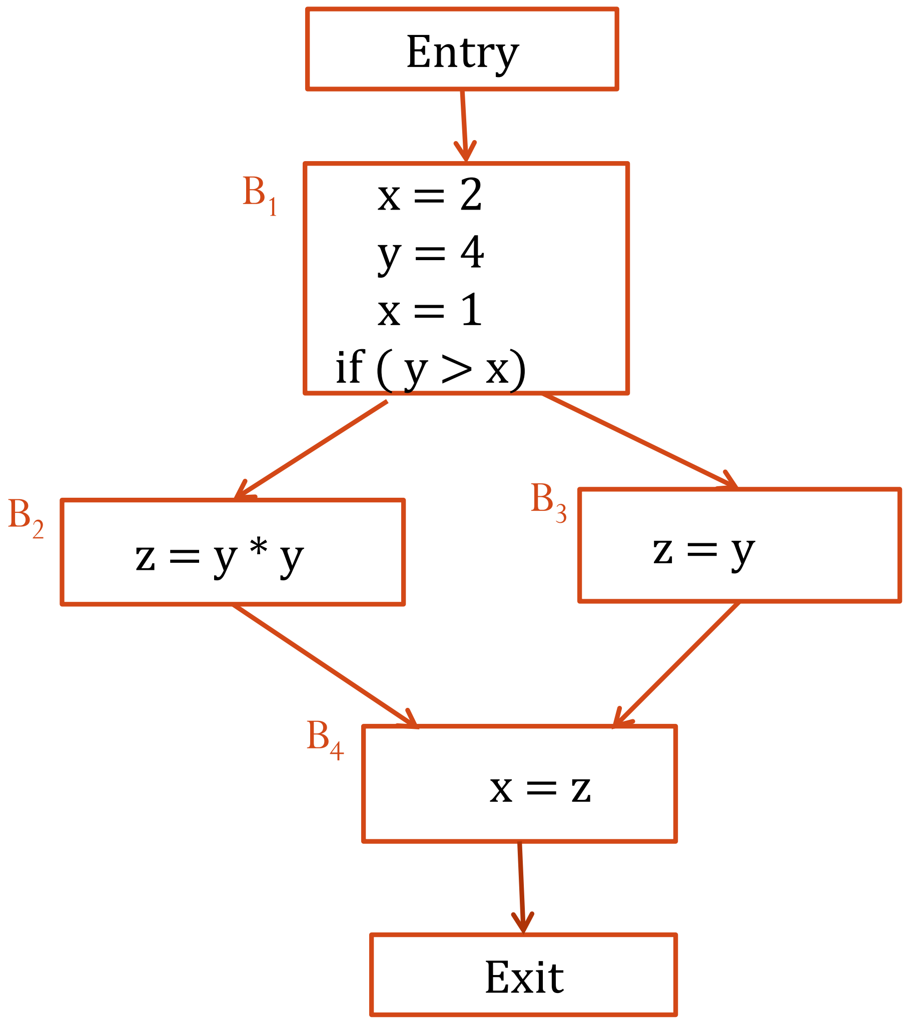
        $OUT[B] = \cup_{S \text{ a successor of } B} \; IN[S]$;

        $IN[B] = GEN[B] \cup (OUT[B] - KILL[B])$

    }

- Does the algorithm converge?
- What is the worst case number of iterations for the while-loop?

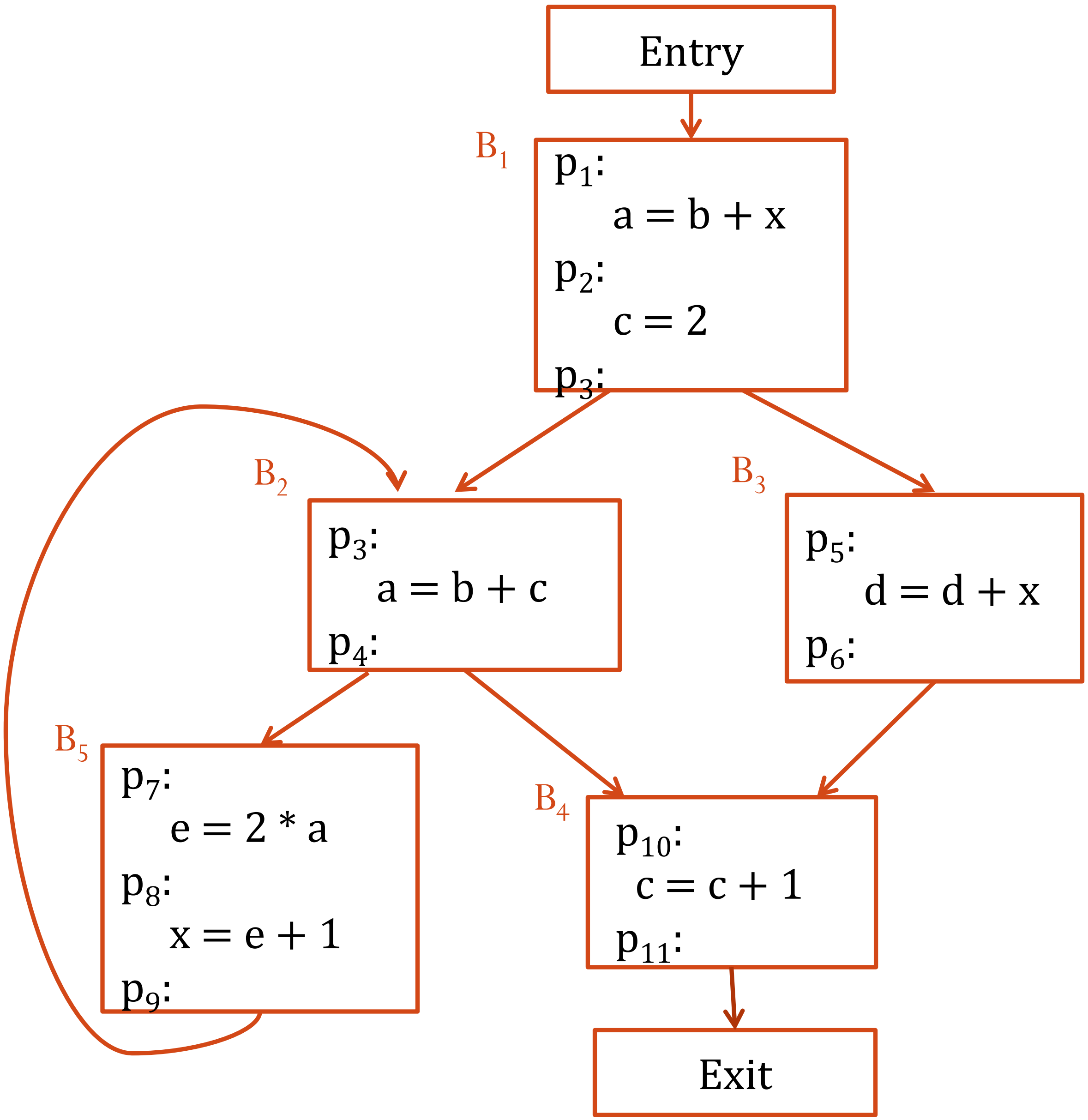
Iterative Algorithm for Computing Live Variables



Iteration	B_1		B_2		B_3		B_4	
	$IN[B_1]$	$OUT[B_1]$	$IN[B_2]$	$OUT[B_2]$	$IN[B_3]$	$OUT[B_3]$	$IN[B_4]$	$OUT[B_4]$
Init. App	{ }	{ }	{ }	{ }	{ }	{ }	{ }	{ }
1.1	{ }	{ }	{}	{ }	{ }	{ }	{ }	{ }
1.2	{ }	{ }	{ y }	{ }	{ }	{ }	{ }	{ }
1.3	{ }	{ }	{ y }	{ }	{ y }	{ }	{ }	{ }
1.4	{ }	{ }	{ y }	{ }	{ y }	{ }	{ z }	{ }
2.1	{ }	{ y }	{ y }	{ }	{ y }	{ }	{ z }	{ }
2.2	{ }	{ y }	{ y }	{ z }	{ y }	{ }	{ z }	{ }
2.3	{ }	{ y }	{ y }	{ z }	{ y }	{ z }	{ z }	{ }
2.4	{ }	{ y }	{ y }	{ z }	{ y }	{ z }	{ z }	{ }
3.1	{ }	{ y }	{ y }	{ z }	{ y }	{ z }	{ z }	{ }
3.2	{ }	{ y }	{ y }	{ z }	{ y }	{ z }	{ z }	{ }
3.3	{ }	{ y }	{ y }	{ z }	{ y }	{ z }	{ z }	{ }
3.4	{ }	{ y }	{ y }	{ z }	{ y }	{ z }	{ z }	{ }

Iteration	B_1		B_2		B_3		B_4	
	$IN[B_1]$	$OUT[B_1]$	$IN[B_2]$	$OUT[B_2]$	$IN[B_3]$	$OUT[B_3]$	$IN[B_4]$	$OUT[B_4]$
Init. App	{ }	{ }	{ }	{ }	{ }	{ }	{ }	{ }
1.1	{ }	{ }	{}	{ }	{ }	{ }	{ z }	{ }
1.2	{ }	{ }	{}	{ }	{ y }	{ z }	{ z }	{ }
1.3	{ }	{ }	{ y }	{ z }	{ y }	{ z }	{ z }	{ }
1.4	{ }	{ y }	{ y }	{ z }	{ y }	{ z }	{ z }	{ }
2.1	{ }	{ y }	{ y }	{ z }	{ y }	{ z }	{ z }	{ }
2.2	{ }	{ y }	{ y }	{ z }	{ y }	{ z }	{ z }	{ }
2.3	{ }	{ y }	{ y }	{ z }	{ y }	{ z }	{ z }	{ }
2.4	{ }	{ y }	{ y }	{ z }	{ y }	{ z }	{ z }	{ }

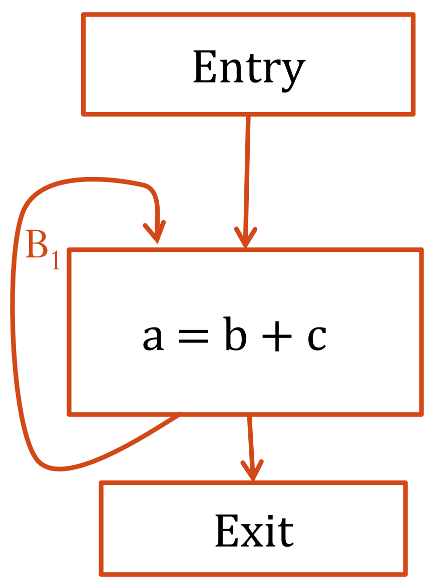
Iterative Algorithm for Computing Live Variables



Iteration	B_1		B_2		B_3		B_4		B_5	
	$IN[B_1]$	$OUT[B_1]$	$IN[B_2]$	$OUT[B_2]$	$IN[B_3]$	$OUT[B_3]$	$IN[B_4]$	$OUT[B_4]$	$IN[B_5]$	$OUT[B_5]$
Init. App	{ }	{ }	{ }	{ }	{ }	{ }	{ }	{ }	{ }	{ }
1.1	{ }	{ }	{ }	{ }	{ }	{ }	{ }	{ }	{ a }	{ }
1.2	{ }	{ }	{ }	{ }	{ }	{ }	{ c }	{ }	{ a }	{ }
1.3	{ }	{ }	{ }	{ }	{ c, d, x }	{ c }	{ c }	{ }	{ a }	{ }
1.4	{ }	{ }	{ c, b }	{ a, c }	{ c, d, x }	{ c }	{ c }	{ }	{ a }	{ }
1.5	{ b, d, x }	{ c, b, d, x }	{ c, b }	{ a, c }	{ c, d, x }	{ c }	{ c }	{ }	{ a }	{ }
2.1	{ b, d, x }	{ c, b, d, x }	{ c, b }	{ a, c }	{ c, d, x }	{ c }	{ c }	{ }	{ c, b, a }	{ c, b }
2.2	{ b, d, x }	{ c, b, d, x }	{ c, b }	{ a, c }	{ c, d, x }	{ c }	{ c }	{ }	{ c, b, a }	{ c, b }
2.3	{ b, d, x }	{ c, b, d, x }	{ c, b }	{ a, c, b }	{ c, d, x }	{ c }	{ c }	{ }	{ c, b, a }	{ c, b }
2.4	{ b, d, x }	{ c, b, d, x }	{ c, b }	{ a, c, b }	{ c, d, x }	{ c }	{ c }	{ }	{ c, b, a }	{ c, b }
2.5	{ b, d, x }	{ c, b, d, x }	{ c, b }	{ a, c, b }	{ c, d, x }	{ c }	{ c }	{ }	{ c, b, a }	{ c, b }
3.1	{ b, d, x }	{ c, b, d, x }	{ c, b }	{ a, c, b }	{ c, d, x }	{ c }	{ c }	{ }	{ c, b, a }	{ c, b }
3.2	{ b, d, x }	{ c, b, d, x }	{ c, b }	{ a, c, b }	{ c, d, x }	{ c }	{ c }	{ }	{ c, b, a }	{ c, b }
3.3	{ b, d, x }	{ c, b, d, x }	{ c, b }	{ a, c, b }	{ c, d, x }	{ c }	{ c }	{ }	{ c, b, a }	{ c, b }
3.4	{ b, d, x }	{ c, b, d, x }	{ c, b }	{ a, c, b }	{ c, d, x }	{ c }	{ c }	{ }	{ c, b, a }	{ c, b }
3.5	{ b, d, x }	{ c, b, d, x }	{ c, b }	{ a, c, b }	{ c, d, x }	{ c }	{ c }	{ }	{ c, b, a }	{ c, b }

Iterative Algorithm for Computing Live Variables

- Run the iterative algorithm setting $IN[B_1] = \{ a, b, c, t \}$ and $IN[Exit] = \{ \}$



Iterative Algorithm for Computing Live Variables

- Run the Iterative Algorithm

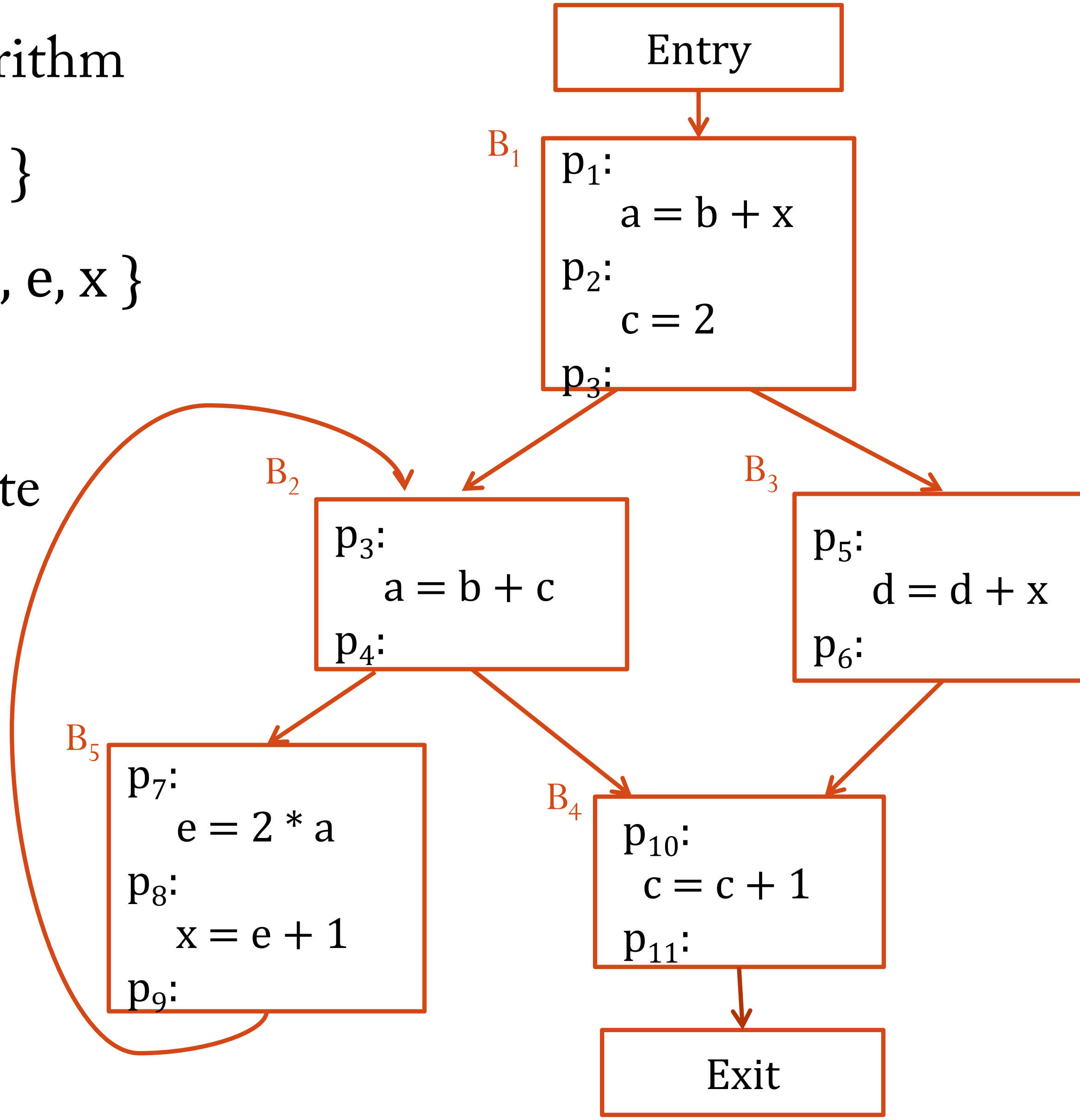
  by setting IN[Exit] = { }

  and IN[B] = { a, b, c, d, e, x }

  for other basic blocks.

- Should we over-estimate

  or under-estimate?

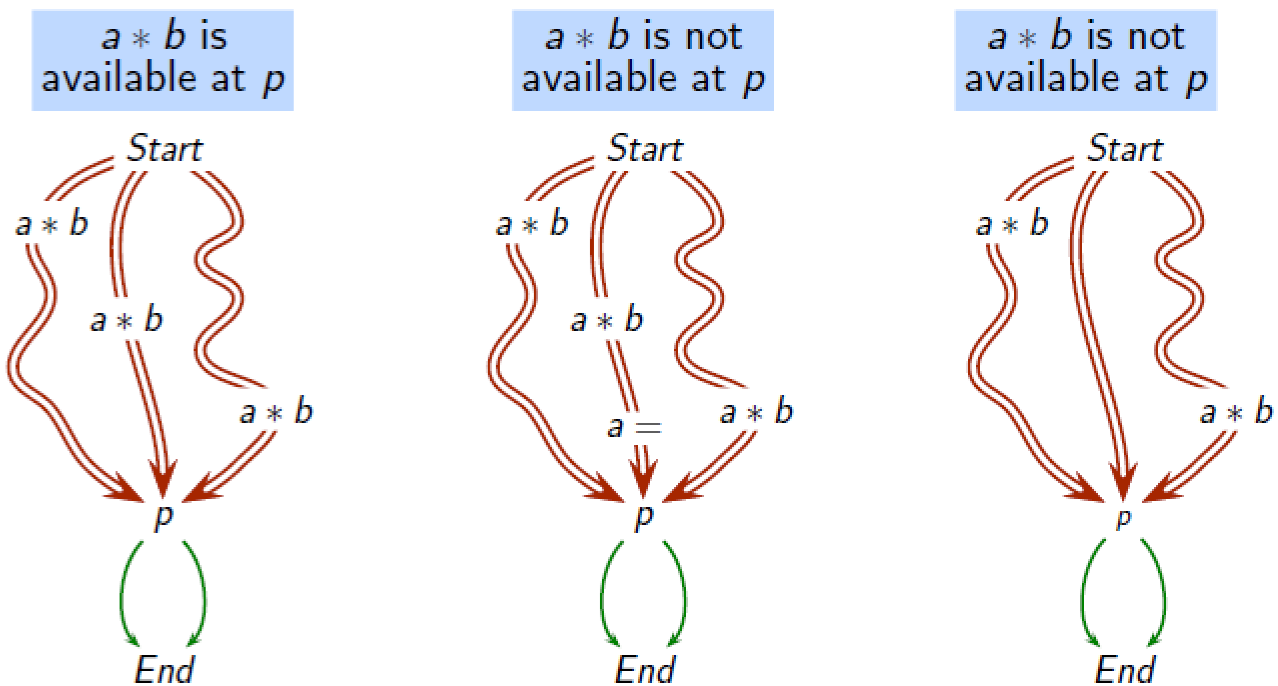


Iterative Algorithm for Computing Live Variables

1. Does the algorithm converge?
2. Does the solution to which the algorithm converges depends on the initial approximation?
3. What should be the initial approximation?
4. Does the number of iterations to converge depends on the order in which we process the basic blocks?
5. What is the worst case number of iterations?

Available Expressions Analysis

- Def: An expression e is available at a program point p, if every path from program entry to p contains an evaluation of e which is not followed by a definition of any operand of e.



Available Expressions Analysis

For a Basic Block B

$$IN[B] = GEN[B] \cup (OUT[B] - KILL[B])$$

- GEN[B] = { e | Expression e evaluated in B and the operands of e are not modified after the last evaluation of e in B }
- KILL[B] = { e | Operands of e are modified in B }
- Expressions in GEN[B] are called downwardly exposed expressions.

- Remark: The above equations takes care of straight line sequence of Code.

Available Expressions Analysis

- Equations (or Constraints) defining the Available Expressions at the beginning and end of Basic Blocks.
- For each Basic Block B we have

$$\mathrm{IN[B]} = \cap_{\mathrm{P\ a\ predecessor\ of\ B}}\ \mathrm{OUT[P]}$$

$$\mathrm{OUT[B]} = \mathrm{GEN[B]} \cup (\mathrm{IN[B]} - \mathrm{KILL[B]})$$

- How to solve for the unknowns?

Iterative Algorithm for Computing Live Variables

1. OUT[ENTRY] = { } ;
2. **for** (each basic block B other than Exit) OUT[B] = U ;
3. **while** (changes to any OUT occur) {

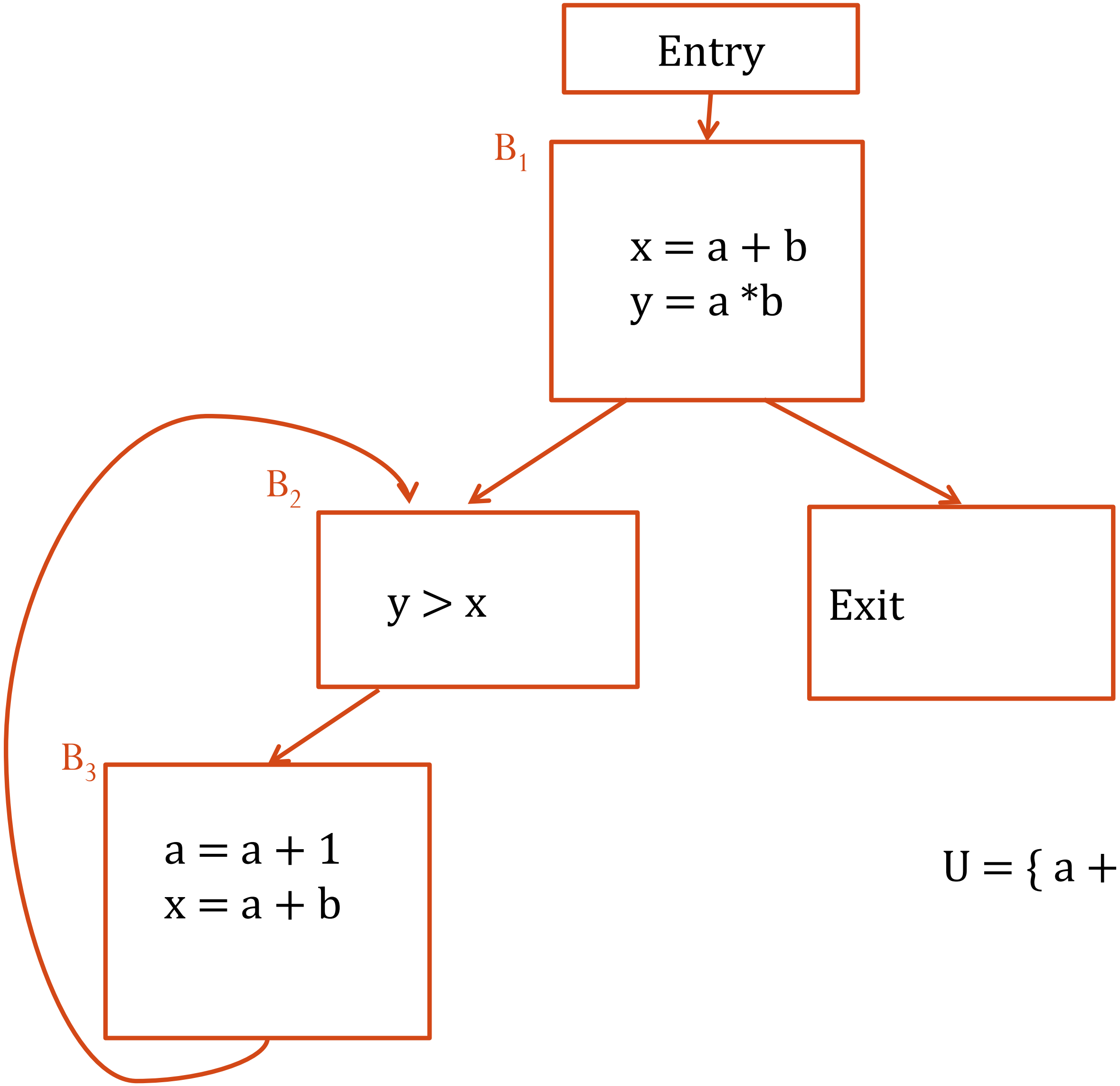
    for (each basic block B other than ENTRY)

    $$IN[B] = \cap_{P \text{ a predecessor of } B}\ OUT[P]$$

    $$OUT[B] = GEN[B] \cup (IN[B] - KILL[B])$$

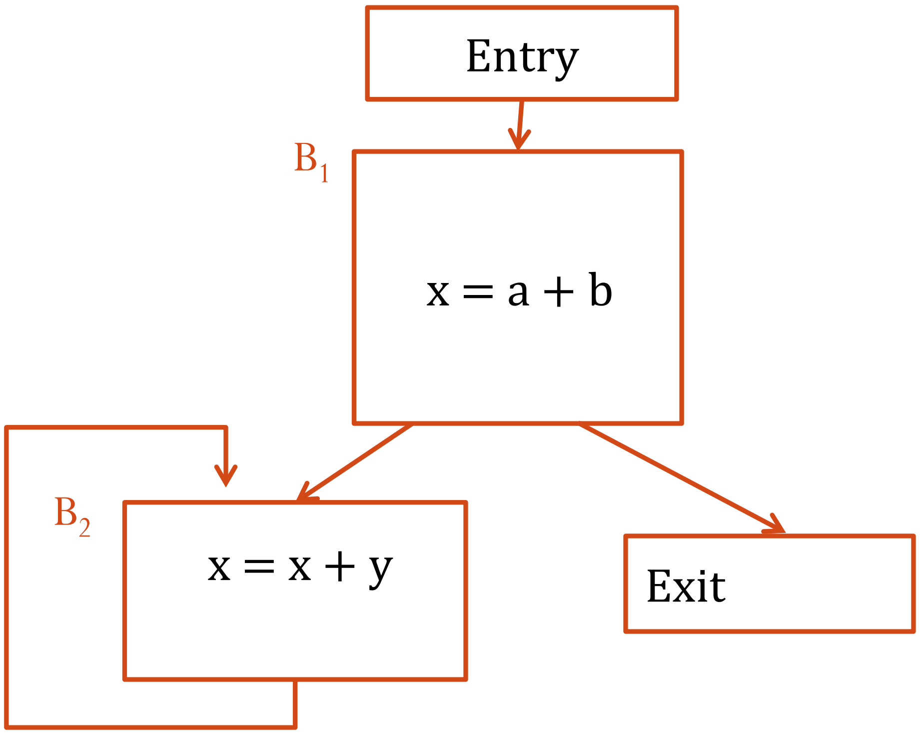
    }

Available Expressions Example



$U = \{ a + b\ (e_1),\ a * b\ (e_2),\ a + 1\ (e_3) \}$

Iteration	B_1		B_2		B_3	
	$IN[B_1]$	$OUT[B_1]$	$IN[B_2]$	$OUT[B_2]$	$IN[B_3]$	$OUT[B_3]$
Init. App	{ }	$\{e_1, e_2, e_3\}$	{ }	$\{e_1, e_2, e_3\}$	{ }	$\{e_1, e_2, e_3\}$
1.1	{ }	$\{e_1, e_2\}$	{ }	$\{e_1, e_2, e_3\}$	{ }	$\{e_1, e_2, e_3\}$
1.2	{ }	$\{e_1, e_2\}$	$\{e_1, e_2\}$	$\{e_1, e_2\}$	{ }	$\{e_1, e_2, e_3\}$
1.3	{ }	$\{e_1, e_2\}$	$\{e_1, e_2\}$	$\{e_1, e_2\}$	$\{e_1, e_2\}$	$\{e_1\}$
2.1	{ }	$\{e_1, e_2\}$	$\{e_1, e_2\}$	$\{e_1, e_2\}$	$\{e_1, e_2\}$	$\{e_1\}$
2.2	{ }	$\{e_1, e_2\}$	$\{e_1, e_2\}$	$\{e_1, e_2\}$	$\{e_1, e_2\}$	$\{e_1\}$
2.3	{ }	$\{e_1, e_2\}$	$\{e_1, e_2\}$	$\{e_1, e_2\}$	$\{e_1, e_2\}$	$\{e_1\}$
3.1	{ }	$\{e_1, e_2\}$	$\{e_1, e_2\}$	$\{e_1, e_2\}$	$\{e_1, e_2\}$	$\{e_1\}$
3.2	{ }	$\{e_1, e_2\}$	$\{e_1\}$	$\{e_1\}$	$\{e_1, e_2\}$	$\{e_1\}$
3.3	{ }	$\{e_1, e_2\}$	$\{e_1\}$	$\{e_1\}$	$\{e_1\}$	$\{e_1\}$
4.1	{ }	$\{e_1, e_2\}$	$\{e_1\}$	$\{e_1\}$	$\{e_1\}$	$\{e_1\}$
4.2	{ }	$\{e_1, e_2\}$	$\{e_1\}$	$\{e_1\}$	$\{e_1\}$	$\{e_1\}$
4.3	{ }	$\{e_1, e_2\}$	$\{e_1\}$	$\{e_1\}$	$\{e_1\}$	$\{e_1\}$



Compute the Available Expressions by setting $OUT[B_1] = \{\ \}$ and $OUT[B_2] = \{\ \}$.

Data Flow Analysis Micro-Summary

	Live Variables	Available Expressions
Domain	Sets of Variables	Sets of Expressions
Direction	Backwards	Forwards
Transfer Function	GEN[B] ∪ (OUT[B] –KILL[B])	GEN[B] ∪ (IN[B] – KILL[B])
Boundary	IN[Exit] = { }	OUT[Entry] = { }
Meet	∪ (May-Analysis)	∩ (Must-Analysis)
Equations	$\text{OUT}[B] = \cup_{S \text{ a successor of } B} \text{IN}[S]$ IN[B] = GEN[B] ∪ (OUT[B] – KILL[B])	$\text{IN}[B] = \cap_{P \text{ a predecessor of } B} \text{IN}[P]$ OUT[B] = GEN[B] ∪ (IN[B] – KILL[B])
Initialize	IN[B] = { }	OUT[B] = U