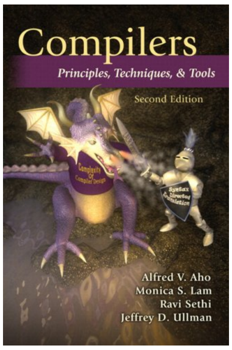
Data Flow Analysis

CS 544

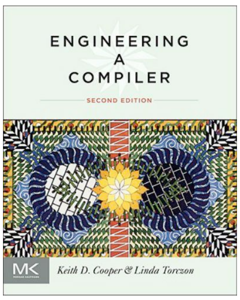
Baris Aktemur

1

- Contents from
  - Alfred V. Aho, Monica Lam, Ravi Sethi, and Jeffrey D. Ullman
    Compilers: Principles, Techniques, and Tools, Second Edition
    Addison-Wesley, 2007, ISBN 0-321-48681-1



- And, where noted as EaC2e, from
  - Keith D. Cooper and Linda Torczon
    Engineering a Compiler, Second Edition
    Morgan Kaufmann, 2011



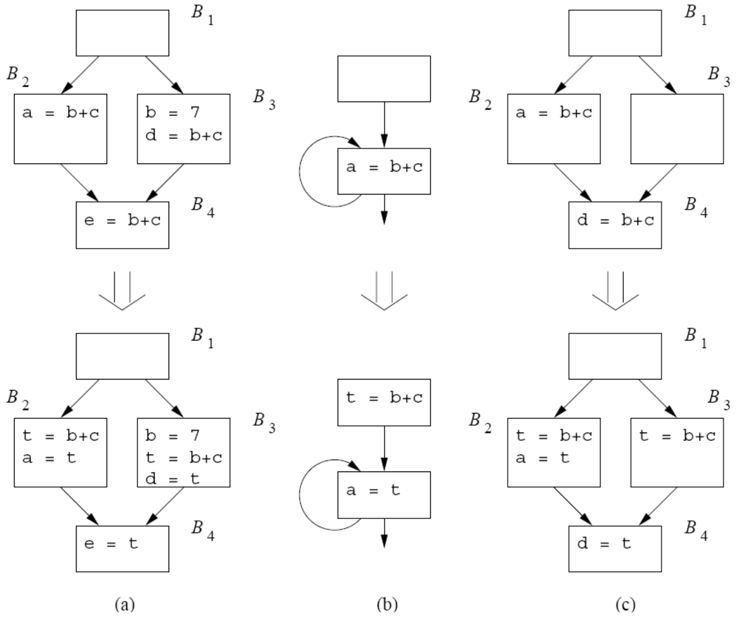


Figure 9.30: Examples of (a) global common subexpression, (b) loop-invariant code motion, (c) partial-redundancy elimination.

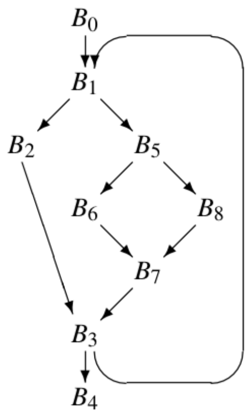
3

Data Flow Analysis (DFA)

- Find opportunities for improving the efficiency of the code
- We must be sure that a particular transformation is safe
- DFA: Compile-time reasoning about the runtime flow of values
- Performed on Control Flow Graph (CFG)

4

Dominance (from EaC2e)



d dominates n (write “d dom n”) iff every path in G from s to n contains d.

$$\text{Dom}(n) = \{n\} \cup \left(\bigcap_{m \in preds(n)} \text{Dom}(m) \right)$$

Initial conditions:

$$\text{Dom}(n_0) = \{n_0\}, \text{ and } \forall n \neq n_0, \text{Dom}(n) = N$$

	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
DOM(*n*)	{0}	{0,1}	{0,1,2}	{0,1,3}	{0,1,3,4}	{0,1,5}	{0,1,5,6}	{0,1,5,7}	{0,1,5,8}

Dominance (from EaC2e)

- Forward data-flow problem:
  - Compute a node’s data based on its predecessors’

```
n ← |N| - 1
Dom(0) ← {0}
for i ← 1 to n
    Dom(i) ← N

changed ← true
while (changed)
   changed ← false

   for i ← 1 to n
       temp ← {i} ∪ ( ∩_{j∈preds(i)} Dom( j ) )

       if temp ≠ Dom(i) then
            Dom(i) ← temp
            changed ← true
```

Iterative approach to solve the Dominance problem

6

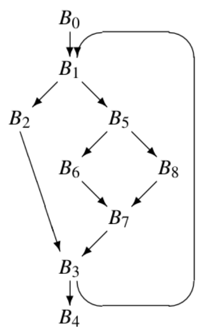
Dominance (from EaC2e)

- A 3-step process
  - Form a CFG
  - Compute initial information for each block
  - Solve the equations to find final information for each block
- Will see this process for any data-flow problem

7

Dominance (from EaC2e)

	DOM(n)								
	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
—	{0}	N	N	N	N	N	N	N	N
1	{0}	{0,1}	{0,1,2}	{0,1,2,3}	{0,1,2,3,4}	{0,1,5}	{0,1,5,6}	{0,1,5,6,7}	{0,1,5,8}
2	{0}	{0,1}	{0,1,2}	{0,1,3}	{0,1,3,4}	{0,1,5}	{0,1,5,6}	{0,1,5,7}	{0,1,5,8}
3	{0}	{0,1}	{0,1,2}	{0,1,3}	{0,1,3,4}	{0,1,5}	{0,1,5,6}	{0,1,5,7}	{0,1,5,8}




```
changed ← true
while (changed)
   changed ← false

   for i ← 1 to n
      temp ← {i} ∪ ( ⋂_{j∈preds(i)} DOM( j ) )

      if temp ≠ DOM(i) then
         DOM(i) ← temp
         changed ← true
```

Dominance (from EaC2e)

- Termination
- Correctness
- Efficiency

9

Termination of Dominance (from EaC2e)

- Dom sets monotonically shrink
- For the initial node, start with itself; for all others, start with N.
- A Dom set cannot grow (check the algorithm)
- A Dom set cannot be smaller than a single-element set.
- Hence, the while-loop eventually terminates

10

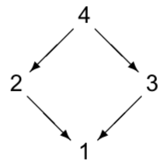
Correctness of Dominance (from EaC2e)

- There exists a unique fixed-point for the equations we solved
- The algorithm finds that unique solution
- Details are beyond our scope. Food for thought…

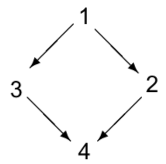
11

Efficiency of Dominance (from EaC2e)

- Unique solution => Order of computing the sets is irrelevant.
- Pick your favorite traversal
- A *reverse postorder* (rpo) traversal of the graph is particularly effective
- Idea: visit a node before its successors.



Postorder



Reverse Postorder

12

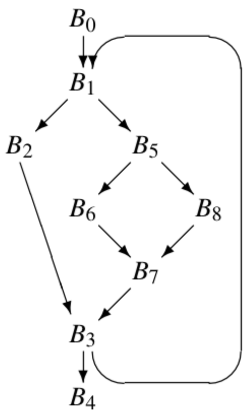
Efficiency of Dominance (from EaC2e)

- For a forward data-flow problem, use an RPO computed on the CFG.
- For a backward data-flow, use an RPO computed on the *reverse* CFG.

- Look up the definition of preorder, postorder, and reverse postorder traversal in your favorite graph theory course/book.

13

Efficiency of Dominance (from EaC2e)



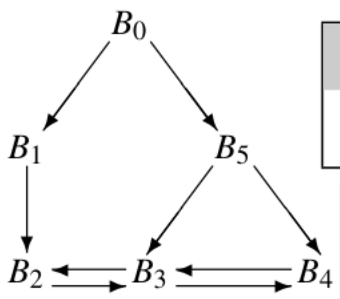
	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
RPO(n)	0	1	6	7	8	2	4	5	3

	DOM(n)								
	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
—	{0}	N	N	N	N	N	N	N	N
1	{0}	{0,1}	{0,1,2}	{0,1,3}	{0,1,3,4}	{0,1,5}	{0,1,5,6}	{0,1,5,7}	{0,1,5,8}
2	{0}	{0,1}	{0,1,2}	{0,1,3}	{0,1,3,4}	{0,1,5}	{0,1,5,6}	{0,1,5,7}	{0,1,5,8}

Two passes, rather than three.

14

Efficiency of Dominance (from EaC2e)



	B_0	B_1	B_2	B_3	B_4	B_5
RPO(n)	0	2	3	4	5	1

	DOM(n)					
	B_0	B_1	B_2	B_3	B_4	B_5
—	{0}	N	N	N	N	N
1	{0}	{0,1}	{0,1,2}	{0,3}	{0,4}	{0,5}
2	{0}	{0,1}	{0,2}	{0,3}	{0,4}	{0,5}
3	{0}	{0,1}	{0,2}	{0,3}	{0,4}	{0,5}

More than two passes needed.

Data Flow Analysis

- Dominance is based only on the structure of the graph.
  - a form of control-flow analysis.
- Behavior of the code is ignored.
- Most data-flow problems reason about the behavior of the code.

Data Flow Abstraction

- Program State:
  - Values of all the variables
  - Value of the program counter
- Execution of a program
  - Series of transformations of the program state
- Each statement transforms an input state to an output state

17

Data Flow Abstraction

- Data Flow Analysis
  - Extract information for all the possible program states
  - Regarding the problem we're trying to solve
- Must consider all the possible paths
- An abstraction of the all possible executions
- Complex problems: Interprocedural
- This lecture: Intraprocedural

18

Data Flow Abstraction

- Program points: just before or after executing a statement
- Program state/data are associated with program points
- Within one basic block, the program point after a statement is the same as the program point before the next statement.
- Execution path: sequence of program points

19

Data Flow Abstraction

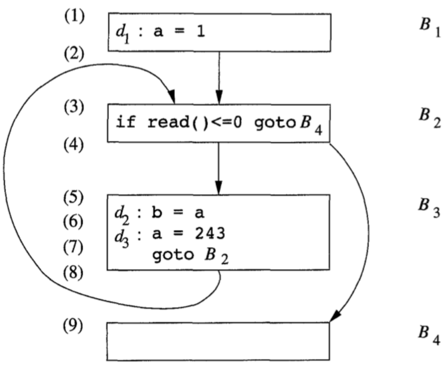
- In general, there is an infinite number of possible execution paths
- No finite upper bound on the length of an execution path
- Program analyses summarize all the possible program states that can occur at a point in the program with a finite set of facts
- Summary is analysis-dependent

20

Reaching Definitions

- A definition of a variable x is a statement that assigns a value to x. (ignore aliasing for simplicity)
- "What definitions of the variable x may be reaching at point p?"

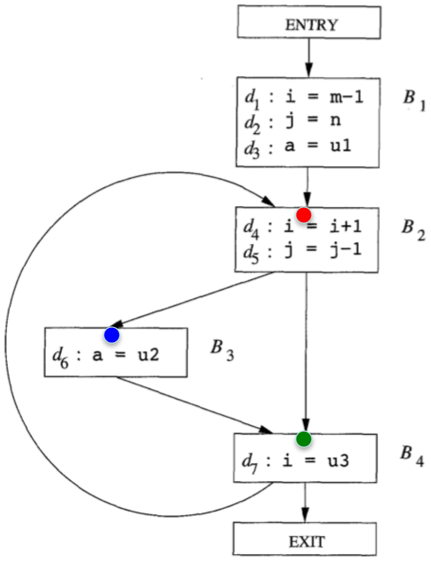
21



- The first time program point (5) is executed, the value of *a* is 1 due to definition d1.
- In subsequent iterations, d3 reaches point (5) and the value of *a* is 243.
- At point (5), the value of *a* is one of {1,243}.
- It may be defined by one of {d1,d3}.

22

Reaching Definitions Exercise



- What def. of i/j/a are reaching specified points?
- {d1,d2,d3,d5,d6,d7}
- {d3,d4,d5,d6}
- {d3,d4,d5,d6}

23

Data Flow Abstraction

- Reaching definitions: The definitions that *may* reach a program point along some path.
- Constant propagation: The unique definition that reaches a point, AND that has a constant value.
  - Distinguish def's as constant vs. non-constant
  - Same information, different summary

24

DFA Schema

- Domain: The set of possible DFA values
  - Analysis-specific
- IN[*s*]: data-flow values before statement *s*
- OUT[*s*]: data-flow values after statement *s*
- The data-flow problem is to find a solution to a set of constraints on the IN[*s*]'s and OUT[*s*]'s, for all statements *s*.

25

DFA Schema

- Transfer function: How a statement changes the data-flow values
  - Analysis- and statement-specific
- Forward flow:
  - OUT[s] = f_s(IN[*s*])
- Backward flow
  - IN[s] = f_s(OUT[*s*])

26

DFA Schema

- Data flow values within a basic block
  - $IN[s_{i+1}] = OUT[s_i]$
  - Note that this is an equality; no difference for forward vs. backward
- Suppose block *B* consists of statements $s_1,...,s_n$, in that order
  - $IN[B] = IN[s_1]$
  - $OUT[B] = OUT[s_n]$
  - $OUT[B] = f_B(IN[B])$ where $f_B = f_{sn} \circ \ldots \circ f_{s2} \circ f_{s1}$

27

DFA Schema

- Constraints due to control flow between basic blocks
  - E.g: Definitions that may reach a point (a forward problem)

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

  - E.g: Live variables (backward problem)

$$\text{IN}[B] = f_B(\text{OUT}[B])$$

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

28

Reaching Definitions

- A definition *d reaches* a point *p* if there is a path from the point immediately following *d* to *p*, such that *d* is not "killed" along that path.
- We *kill* a definition of a variable x if there is any other definition of x anywhere along the path.

29

Transfer Equations

d: `u = v+w`

- This statement "generates" a definition d of variable u and "kills" all the other definitions in the program that define variable u, while leaving the remaining incoming definitions unaffected.

$$f_d(x) = gen_d \cup (x - kill_d)$$

where gen_d is {*d*} and $kill_d$ is the set of all other definitions of u.

30

Transfer Equations

- Composition.
  - Suppose we have

$f_1(x) = gen_1 \cup (x - kill_1)$ and $f_2(x) = gen_2 \cup (x - kill_2)$

then

$$
\begin{aligned}
f_2\big(f_1(x)\big) &= gen_2 \cup \big(gen_1 \cup (x - kill_1) - kill_2\big) \\
&= \big(gen_2 \cup (gen_1 - kill_2)\big) \cup \big(x - (kill_1 \cup kill_2)\big)
\end{aligned}
$$

31

Transfer Equations

- Composition.
  - Transfer function of a block B with n statements:

$$f_B(x) = gen_B \cup (x - kill_B),$$

where

$$kill_B = kill_1 \cup kill_2 \cup \cdots \cup kill_n$$

and

$$
\begin{aligned}
gen_B \; &= gen_n \cup (gen_{n-1} - kill_n) \cup (gen_{n-2} - kill_{n-1} - kill_n) \cup \\
&\quad \cdots \cup (gen_1 - kill_2 - kill_3 - \cdots - kill_n)
\end{aligned}
$$

gen/kill sets of basic blocks:

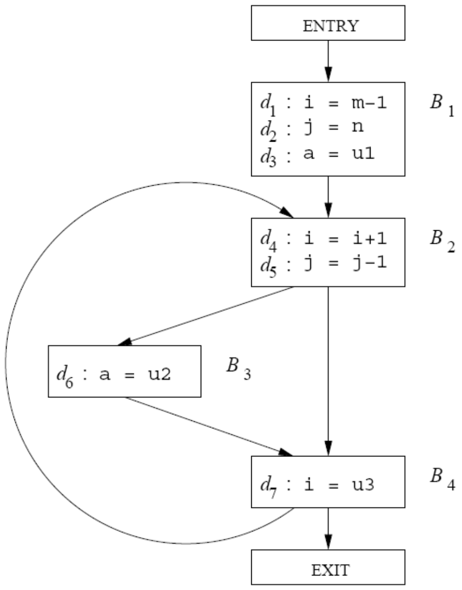
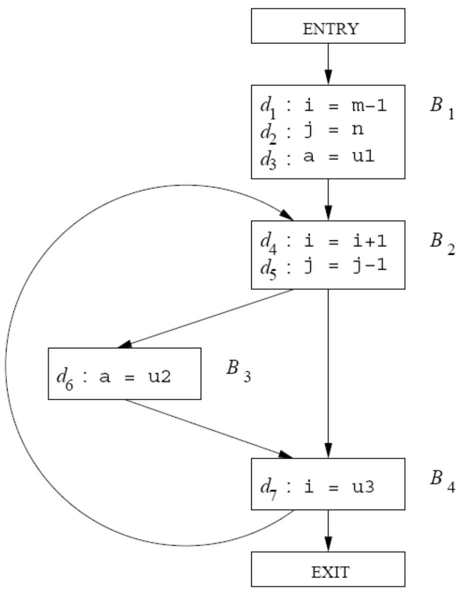


Figure 9.13: Flow graph for illustrating reaching definitions

33

gen/kill sets of basic blocks:



$gen_{B_1} = \{ d_1, d_2, d_3 \}$

$kill_{B_1} = \{ d_4, d_5, d_6, d_7 \}$

$gen_{B_2} = \{ d_4, d_5 \}$

$kill_{B_2} = \{ d_1, d_2, d_7 \}$

$gen_{B_3} = \{ d_6 \}$

$kill_{B_3} = \{ d_3 \}$

$gen_{B_4} = \{ d_7 \}$

$kill_{B_4} = \{ d_1, d_4 \}$

Figure 9.13: Flow graph for illustrating reaching definitions

34

Control Flow Equations

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

$$\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)$$

$$\text{OUT}[\text{ENTRY}] = \emptyset$$ (Initial condition)

- Solution to the equations above is a *fixed-point* of the system. We are interested in finding the *least fixed-point*.

35

```
OUT[ENTRY] = ∅;
for (each basic block B other than ENTRY) OUT[B] = ∅;
while (changes to any OUT occur)
        for (each basic block B other than ENTRY) {
                IN[B] = ∪_{P a predecessor of B} OUT[P];
                OUT[B] = gen_B ∪ (IN[B] − kill_B);
        }
```

Figure 9.14: Iterative algorithm to compute reaching definitions

- Note the three-step process
  - Build a CFG (already done)
  - Initialize local information
  - Compute global information (i.e. propagate local info until the fixed-point)

36

Represent sets by bit-vectors

Block B	$\text{OUT}[B]^0$	$\text{IN}[B]^1$	$\text{OUT}[B]^1$	$\text{IN}[B]^2$	$\text{OUT}[B]^2$
B_1	000 0000	000 0000	111 0000	000 0000	111 0000
B_2	000 0000	111 0000	001 1100	111 0111	001 1110
B_3	000 0000	001 1100	000 1110	001 1110	000 1110
B_4	000 0000	001 1110	001 0111	001 1110	001 0111
EXIT	000 0000	001 0111	001 0111	001 0111	001 0111

E.g:

$$\begin{aligned}\text{IN}[B_2]^1 &= \text{OUT}[B_1]^1 \cup \text{OUT}[B_4]^0 \\ &= 111\ 0000 + 000\ 0000 = 111\ 0000 \\ \text{OUT}[B_2]^1 &= gen[B_2] \cup (\text{IN}[B_2]^1 - kill[B_2]) \\ &= 000\ 1100 + (111\ 0000 - 110\ 0001) = 001\ 1100\end{aligned}$$

37

Reaching Definitions

- Detecting uses before definitions (i.e. uninitialized variables)
  - Introduce a dummy definition for each variable x in the entry to the flow graph. If the dummy definition of x reaches a point p where x might be used, then there might be an opportunity to use x before definition.

38

Live Variables

- Can the value of x at p be used along some path in the flow graph starting at p?
- If so, x is *live*, otherwise, *dead* at p.
- Important analysis for register allocation.
- Backward analysis.

39

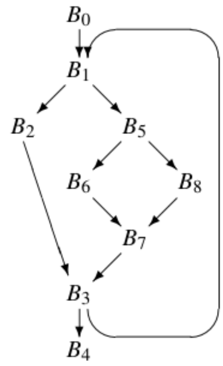
Transfer Functions

- def_B *:* the set of variables *defined* (i.e., definitely assigned values) in B
- use_B *:* the set of variables whose values may be used in B prior to any definition of the variable. (i.e. upwards exposed variables)

40

```
B0:  i ← 1
     → B1
B1:  a ← ···
     c ← ···
     (a < c) → B2,B5
B2:  b ← ···
     c ← ···
     d ← ···
     → B3
B3:  y ← a + b
     z ← c + d
     i ← i + 1
     (i ≤ 100) → B1,B4
B4:  return
B5:  a ← ···
     d ← ···
     (a ≤ d) → B6,B8
B6:  d ← ···
     → B7
B7:  b ← ···
     → B3
B8:  c ← ···
     → B7
```

(a) Code for the Basic Blocks



(b) Control-Flow Graph

	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
use	$\emptyset$	$\emptyset$	$\emptyset$	{a,b,c,d,i}	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
def	{i}	{a,c}	{b,c,d}	{y,z,i}	$\emptyset$	{a,d}	{d}	{b}	{c}

(c) Initial Information

From EaC2e

Live Variables

- Constraints

$$\text{IN}[B] = use_B \cup (\text{OUT}[B] - def_B)$$

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

- Initial condition

$$\text{IN}[\text{EXIT}] = \emptyset$$

42

LV vs. RD

- Both have union as the meet operator: In each, we care only about whether a path with desired properties exists, rather than whether something is true along all paths.
- Information flow for liveness travels "backward," whereas “forward” in reachability.
- gen/kill vs use/def.

43

$\text{IN}[\text{EXIT}] = \emptyset;$
for (each basic block B other than EXIT) $\text{IN}[B] = \emptyset;$
while (changes to any IN occur)
    for (each basic block B other than EXIT) {
        $\text{OUT}[B] \ = \ \bigcup_{S \text{ a successor of } B} \text{IN}[S];$
        $\text{IN}[B] = use_B \cup (\text{OUT}[B] - def_B);$
    }

Figure 9.16: Iterative algorithm to compute live variables

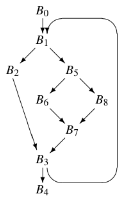
44

From EaC2e

```
B0:  i ← 1
     → B1
B1:  a ← ···
     c ← ···
     (a < c) → B2,B5
B2:  b ← ···
     c ← ···
     d ← ···
     → B3
B3:  y ← a + b
     z ← c + d
     i ← i + 1
     (i ≤ 100) → B1,B4

B4:  return
B5:  a ← ···
     d ← ···
     (a ≤ d) → B6,B8
B6:  d ← ···
     → B7
B7:  b ← ···
     → B3
B8:  c ← ···
     → B7
```

(a) Code for the Basic Blocks



(b) Control-Flow Graph

	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
UEVar	∅	∅	∅	{a,b,c,d,i}	∅	∅	∅	∅	∅
VarKill	{i}	{a,c}	{b,c,d}	{y,z,i}	∅	{a,d}	{d}	{b}	{c}

(c) Initial Information

RPO on CFG:

	LiveOut(n)								
	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
—	∅	∅	∅	∅	∅	∅	∅	∅	∅
1	∅	∅	{a,b,c,d,i}	∅	∅	∅	∅	{a,b,c,d,i}	∅
2	∅	{a,i}	{a,b,c,d,i}	{i}	∅	∅	{a,c,d,i}	{a,b,d,c,i}	{a,c,d,i}
3	{i}	{a,i}	{a,b,c,d,i}	{i}	∅	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{a,c,d,i}
4	{i}	{a,c,i}	{a,b,c,d,i}	{i}	∅	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{a,c,d,i}
5	{i}	{a,c,i}	{a,b,c,d,i}	{i}	∅	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{a,c,d,i}

From EaC2e

	LiveOut(n)								
	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
—	∅	∅	∅	∅	∅	∅	∅	∅	∅
1	{i}	{a,c,i}	{a,b,c,d,i}	∅	∅	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{a,c,d,i}
2	{i}	{a,c,i}	{a,b,c,d,i}	{i}	∅	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{a,c,d,i}
3	{i}	{a,c,i}	{a,b,c,d,i}	{i}	∅	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{a,c,d,i}

- If computed on RPO of the reverse CFG

46

Uninitialized Variables

- How can you use Live Variable analysis to detect if there may be uninitialized variables? (i.e. variables that are being used before being defined)
  - Check OUT[entry]. If non-empty, there may be a problem.

47

Uninitialized Variables

- Of course, this is a conservative analysis. There may be false positives. Consider the following code (Taken from EaC2e)

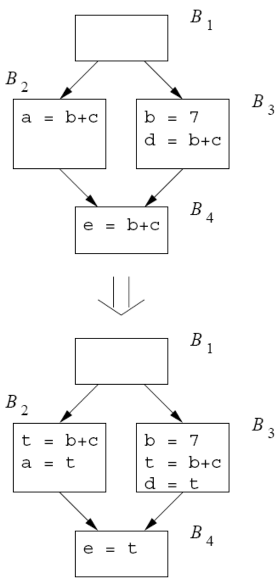
```
main() {
  int i, n, s;
  scanf("%d", &n);
  i = 1;
  while (i<=n) {
    if (i==1)
      s = 0;
    s = s + i++;
  }
}
```

48

Available Expressions

- Expression x+y is available at point p if
  - every path from the entry node to p evaluates x+y, and
  - after the last such evaluation prior to reaching p, there are no assignments to x or y
- Useful for common subexpression elimination

49



(a)

50

Available Expressions

- A block *kills* expression x+y if it assigns x or y and does not subsequently recompute x+y.
- A block *generates* expression x+y if it definitely evaluates x+y and does not subsequently define x or y.

Statement	Available Expressions
	$\emptyset$
`a = b + c`	
	$\{b+c\}$
`b = a - d`	
	$\{a-d\}$
`c = b + c`	
	$\{a-d\}$
`d = a - d`	
	$\emptyset$

51

OUT[ENTRY] = $\emptyset$;
for (each basic block B other than ENTRY) OUT[B] = U;
while (changes to any OUT occur)
  for (each basic block B other than ENTRY) {
    IN[B] = $\bigcap_{P \text{ a predecessor of } B}$ OUT[P];
    OUT[B] = $e\_gen_B \cup$ (IN[B] $- e\_kill_B$);
  }

Figure 9.20: Iterative algorithm to compute available expressions

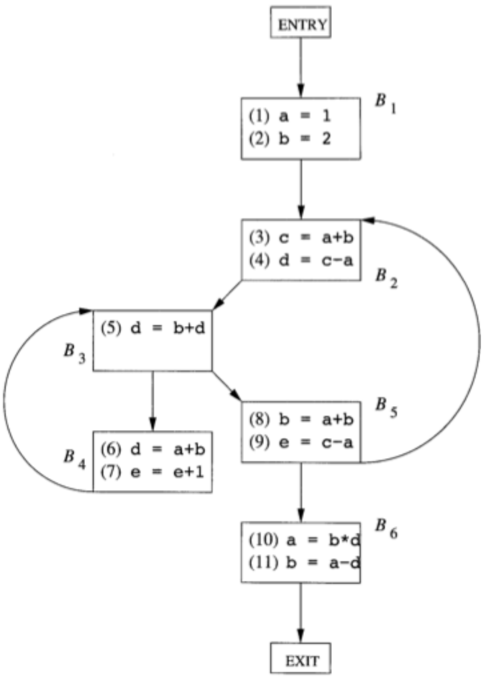
- Meet operation is intersection.
- OUT[B] are set to U, except the entry node.
  – U is the universal set of expressions.

52

Summary

	Reaching Definitions	Live Variables	Available Expressions
Domain	Sets of definitions	Sets of variables	Sets of expressions
Direction	Forwards	Backwards	Forwards
Transfer function	$gen_B \cup (x - kill_B)$	$use_B \cup (x - def_B)$	$e\_gen_B \cup (x - e\_kill_B)$
Boundary	$\text{OUT}[\text{ENTRY}] = \emptyset$	$\text{IN}[\text{EXIT}] = \emptyset$	$\text{OUT}[\text{ENTRY}] = \emptyset$
Meet ($\wedge$)	$\cup$	$\cup$	$\cap$
Equations	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P,pred(B)} \text{OUT}[P]$	$\text{IN}[B] = f_B(\text{OUT}[B])$ $\text{OUT}[B] = \bigwedge_{S,succ(B)} \text{IN}[S]$	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P,pred(B)} \text{OUT}[P]$
Initialize	$\text{OUT}[B] = \emptyset$	$\text{IN}[B] = \emptyset$	$\text{OUT}[B] = U$

53



Exercise: Compute

- def, use, IN and OUT for LV analysis.
- e_gen, e_kill, IN and OUT for AE analysis.

54

Interprocedural Summary Problems (from EaC2e)

- Function calls significantly degrade the information collected by an analysis
  - For safety, we have to assume that the callee function may modify any global or pass-by-ref variable
- Interprocedural may modify problem:
  - Determine which variables may be modified by called functions.
  - A data-flow analysis on the **call graph**
    - Flow insensitive

55

Interprocedural Summary Problems (from EaC2e)

$$\textsc{MayMod}(p) = \textsc{LocalMod}(p) \cup \left(\bigcup_{e=(p,q)} unbind_e(\textsc{MayMod}(q)) \right)$$

- *unbind* function maps one set of variables into another
- *e* is an edge in the call graph

56