Program Analysis

https://www.cse.iitb.ac.in/~karkare/cs618/

Data Flow Analysis (contd…)

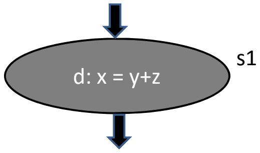
Amey Karkare
Dept of Computer Science and Engg
IIT Kanpur
Visiting IIT Bombay
karkare@cse.iitk.ac.in
karkare@cse.iitb.ac.in

Available Expressions

- Expression e is available at a point p:
  - **Every** path from entry to p has at least one evaluation of e
  - There is no assignment to any component variable of e **after last evaluation** of e prior to p
- Expression e is *generated* by its evaluation
- Expression e is *killed* by assignment to its component variables

2

Available Expr Analysis



gen(s1) = {y+z}

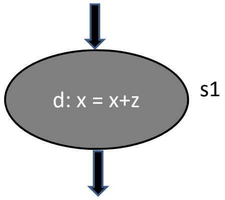
kill(s1) = E_x // E_x : set of all expressions having x as a component

out(s1) = in(s1) - kill(s1) U gen(s1)

THIS MAY NOT WORK IN GENERAL! WHY?

3

Available Expr Analysis



INCORRECT FORMULATION

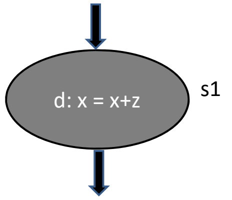
out(s1) = in(s1) - kill(s1) U gen(s1)

gen(s1) = {x+z}

kill(s1) = E_x // E_x : set of all expressions having x as a component

4

Available Expr Analysis



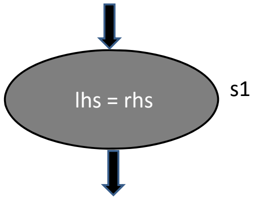
INCORRECT FORMULATION

~~out(s1) = in(s1) - kill(s1) U gen(s1)~~

~~gen(s1) = {x+z}~~

~~kill(s1) = E_x // E_x : set of all expressions having x as a component~~

Available Expr Analysis



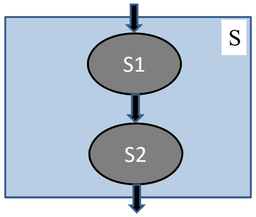
CORRECT FORMULATION

out(s1) = in(s1) - kill(s1) U gen(s1)

gen(s1) = { rhs | lhs is not part of rhs }

kill(s1) = E_{lhs} // E_{lhs} : set of all expressions having lhs as a component

Available Expr Analysis



gen(s) = gen(s2) U (gen(s1) – kill(s2))

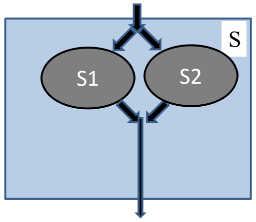
kill(s) = kill(s2) U (kill(s1) – gen(s2))

in(s1) = in(s)

in(s2) = out(s1)

out(s) = out(s2)

Analysis of Structured Programs



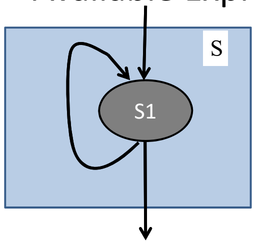
gen(s) = gen(s1) $\cap$ gen(s2)

kill(s) = kill(s1) U kill(s2)

in(s1) = in(s2) = in(s)

out(s) = out(s1) $\cap$ out(s2)

Available Expr Analysis



```
z = x*y;
do {} while(…);
// is x*y available
// here?
```

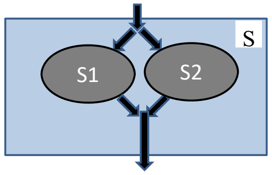
gen(s) = gen(s1)

kill(s) = kill(s1)

in(s1) = in(s) ∩ ~~gen~~ out(s1)

out(s) = out(s1)

Again: Conservative Analysis

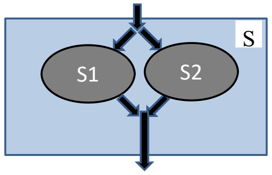


- Assumption: All paths are feasible.
  - Consider: if (true) s1; else s2
  - s2 is never executed

  gen(s) = gen(s1) ⊇ gen(s1) ∩ gen(s2)

  kill(s) = kill(s1) ⊆ kill(s1) ∪ kill(s2)

Again: Conservative Analysis



- Thus: true gen (s) ⊇ analysis gen(s)

  true kill (s) ⊆ analysis kill(s)
- True is what is computed at run time
- This is SAFE estimate
  - prevents optimization
  - but no wrong optimization

Available Expressions

- Expr e is available at the start of a block
  - It is available at the end of **all** predecessors

  $$in(B) = \cap_{P \text{ is pred of } B}\ out(P)$$
- Expr e is available at the end of a block
  - either it is generated by the block
  - or it is available at the start of the block and not killed by the block

  $$out(B) = in(B) – kill(B) \cup gen(B)$$

Available Expressions

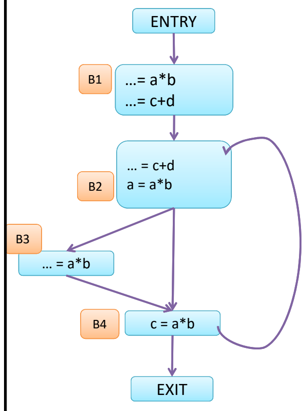
- Kill & gen known for each block.
- A program with N blocks has 2N equations with 2N unknowns
  - –solution is possible.
  - –iterative approach (on next slide)

```
for each block B {
  out(B) = U; // U = “universal” set of all exprs
}
out(Entry) = φ; // remember reaching defs?
change = true;
while (change) {
  change = false;
  for each block B other than Entry {
    in(B) = ∩ P is pred of B out(P);
    oldOut = out(B);
    out(B) = in(B) - kill(B) ∪ gen(B);
    if (oldOut != out(B)) then {
      change = true;
    }
  }
}
```

Some Issues

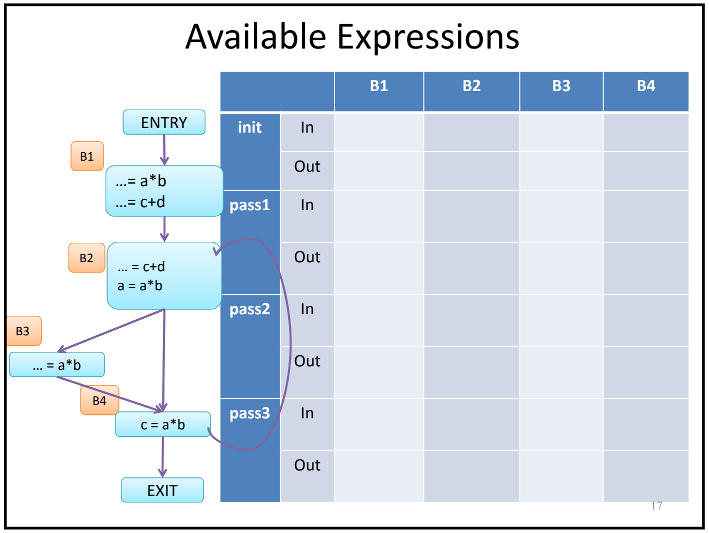
- What is the set of all expressions?
- How to compute it efficiently?
- Why Entry block is initialized differently?

Available Expressions



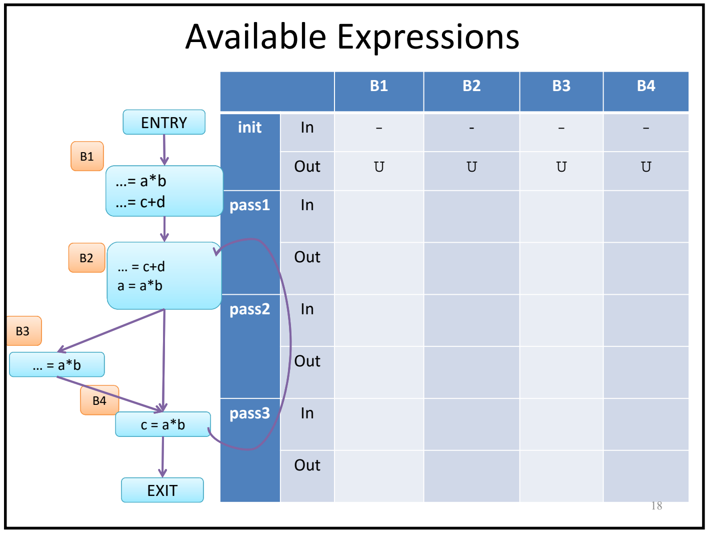
BLOCK	GEN	KILL
B1	{a*b, c+d}	{}
B2	{c+d}	{a*b}
B3	{a*b}	{}
B4	{a*b}	{c+d}

U = {a*b, c+d}
We are not interested in other expressions/variables



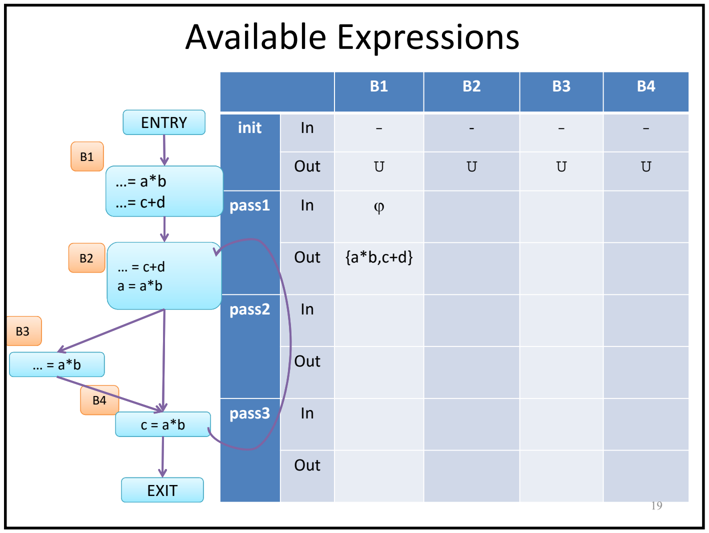
Available Expressions

		B1	B2	B3	B4
init	In				
	Out				
pass1	In				
	Out				
pass2	In				
	Out				
pass3	In				
	Out				



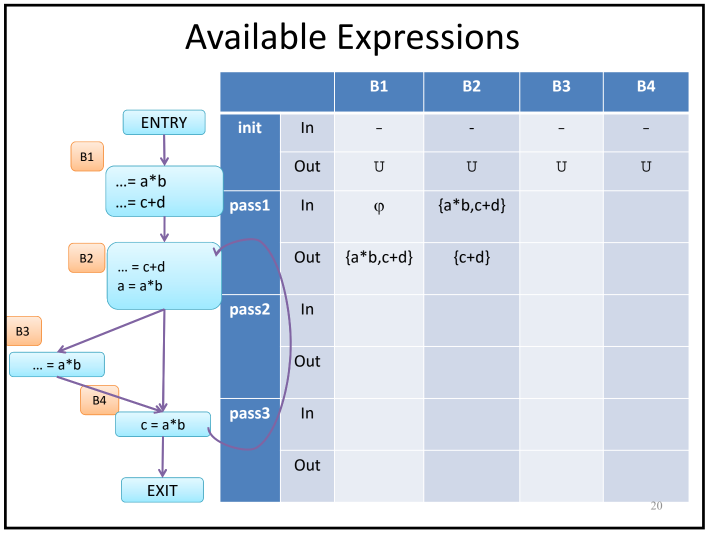
Available Expressions

		B1	B2	B3	B4
init	In	-	-	-	-
	Out	U	U	U	U
pass1	In				
	Out				
pass2	In				
	Out				
pass3	In				
	Out				



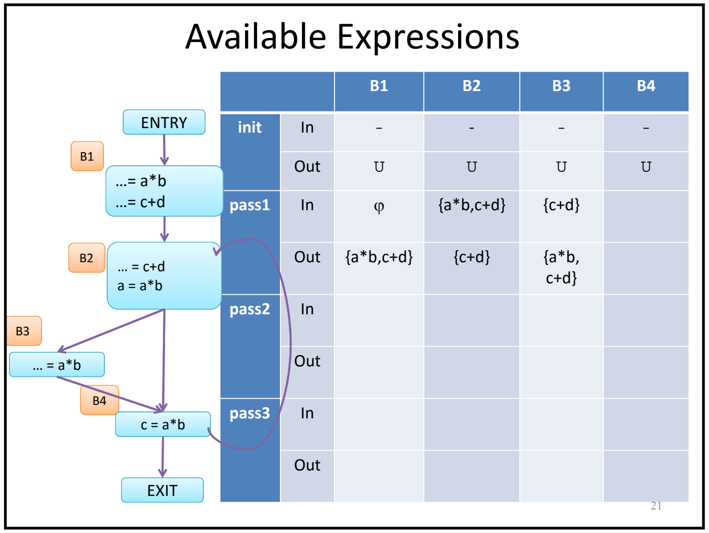
Available Expressions

		B1	B2	B3	B4
init	In	-	-	-	-
	Out	U	U	U	U
pass1	In	φ			
	Out	{a*b,c+d}			
pass2	In				
	Out				
pass3	In				
	Out				



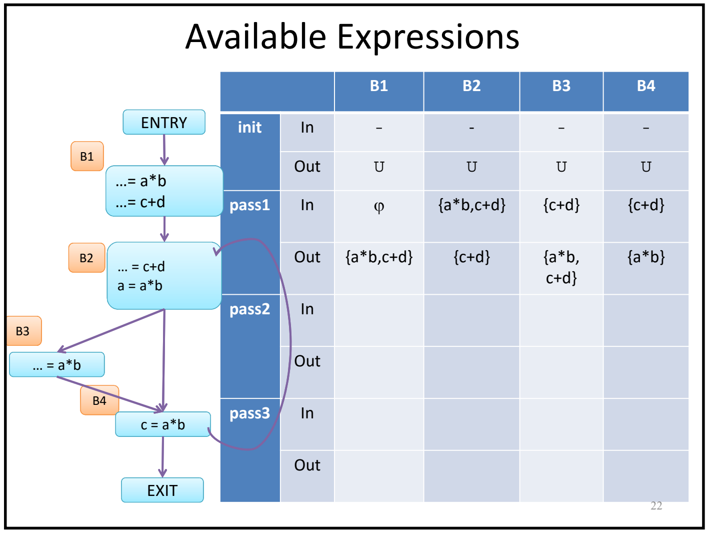
Available Expressions

		B1	B2	B3	B4
init	In	-	-	-	-
	Out	U	U	U	U
pass1	In	φ	{a*b,c+d}		
	Out	{a*b,c+d}	{c+d}		
pass2	In				
	Out				
pass3	In				
	Out				



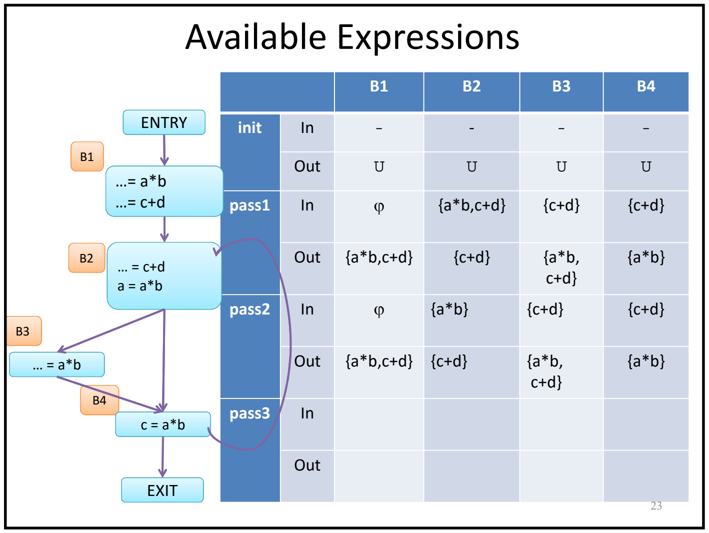
Available Expressions

		B1	B2	B3	B4
init	In	-	-	-	-
	Out	U	U	U	U
pass1	In	φ	{a*b,c+d}	{c+d}	
	Out	{a*b,c+d}	{c+d}	{a*b, c+d}	
pass2	In				
	Out				
pass3	In				
	Out				



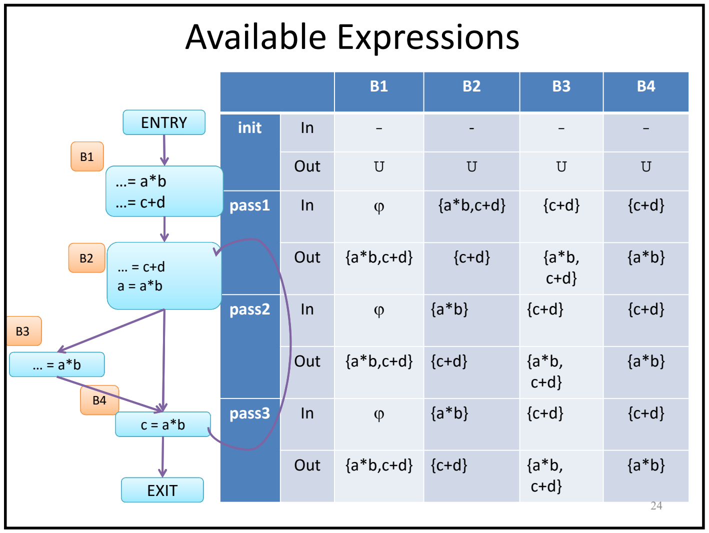
Available Expressions

		B1	B2	B3	B4
init	In	-	-	-	-
	Out	U	U	U	U
pass1	In	φ	{a*b,c+d}	{c+d}	{c+d}
	Out	{a*b,c+d}	{c+d}	{a*b, c+d}	{a*b}
pass2	In				
	Out				
pass3	In				
	Out				



Available Expressions

		B1	B2	B3	B4
init	In	-	-	-	-
	Out	U	U	U	U
pass1	In	φ	{a*b,c+d}	{c+d}	{c+d}
	Out	{a*b,c+d}	{c+d}	{a*b, c+d}	{a*b}
pass2	In	φ	{a*b}	{c+d}	{c+d}
	Out	{a*b,c+d}	{c+d}	{a*b, c+d}	{a*b}
pass3	In				
	Out				



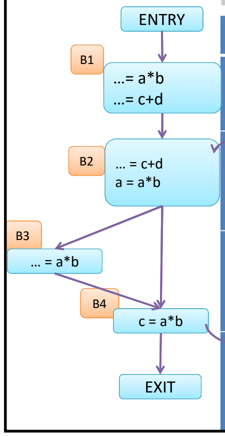
Available Expressions

		B1	B2	B3	B4
init	In	-	-	-	-
	Out	U	U	U	U
pass1	In	φ	{a*b,c+d}	{c+d}	{c+d}
	Out	{a*b,c+d}	{c+d}	{a*b, c+d}	{a*b}
pass2	In	φ	{a*b}	{c+d}	{c+d}
	Out	{a*b,c+d}	{c+d}	{a*b, c+d}	{a*b}
pass3	In	φ	{a*b}	{c+d}	{c+d}
	Out	{a*b,c+d}	{c+d}	{a*b, c+d}	{a*b}

Available Expressions: Bitvectors

a*b	c+d

bits for each expression



		B1	B2	B3	B4
init	In	-	-	-	-
	Out	11	11	11	11
pass1	In	00	11	01	01
	Out	11	01	11	10
pass2	In	00	10	01	01
	Out	11	01	11	10
pass3	In	00	10	01	01
	Out	11	01	11	10

Available Expressions: Bitvectors

$in(B) = \cap_{\text{P is pred of B}}\ out(P)$

$out(B) = in(B) - kill(B) \cup gen(B)$

- With bit vectors,

$in(B) = \wedge_{\text{P is pred of B}}\ out(P)$

$out(B) = (in(B) \wedge \neg kill(B)) \vee gen(B)$

- Bitwise $\wedge$, $\vee$, $\neg$ operations.

Available Expressions: Application

- Common subexpression elimination in a block B
  - Expression e available at the entry of B
  - e is also computed at a point p in B
  - Components of e are not modified from entry of B to p
- e is "*upward exposed*" in B
- Expressions generated in B are "*downward exposed*"

Recap: Summary of Reaching Definitions

gen = { d_x | d_x in B defines variable *x* and is not followed by another definition of x in B}

kill = { d_x | block contains some definition of x }

$in(B) = \cup_{\text{P is pred of B}}\ out(P)$

$out(B) = in(B) - kill(B) \cup gen(B)$

meet ($\wedge$) operator is $\cup$

Initialization:

$out(B_{entry})$ = Entry Info = φ

$out(B) = \varphi$

Summary of Available Expressions

gen = downward exposed expressions

kill = { e_x | block contains some definition of x }

in(B) = $\cap_{\text{P is pred of B}}$ out(P)

out(B) = in(B) –kill(B) $\cup$ gen(B)

meet (Λ) operator is $\cap$

Initialization:

out(B_{entry}) = Entry Info = φ

out(B) = Ʊ

Comparing Reaching Definition and Available Expressions Analysis

- Class Discussion about
  - –Similarities
  - –Differences

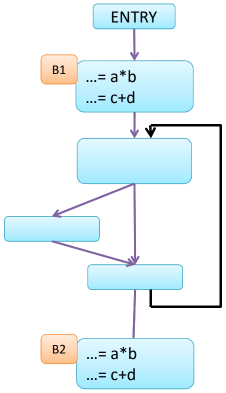
Summary of Available Expressions

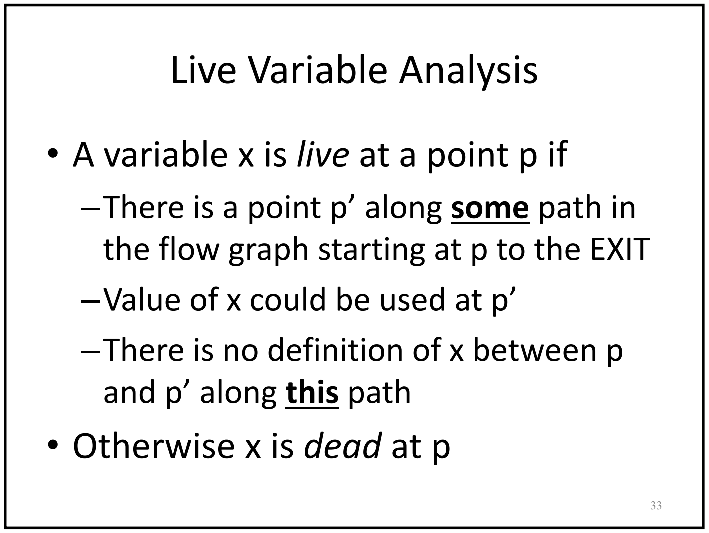
- What if we Initialize:

  out(B_{entry}) = Entry Info = φ

  out(B) = φ
- We might miss some expressions that are available
- Loose on opportunity to optimize!

What are the expressions available at B2 when out(B) initialized with
i) U ii) φ

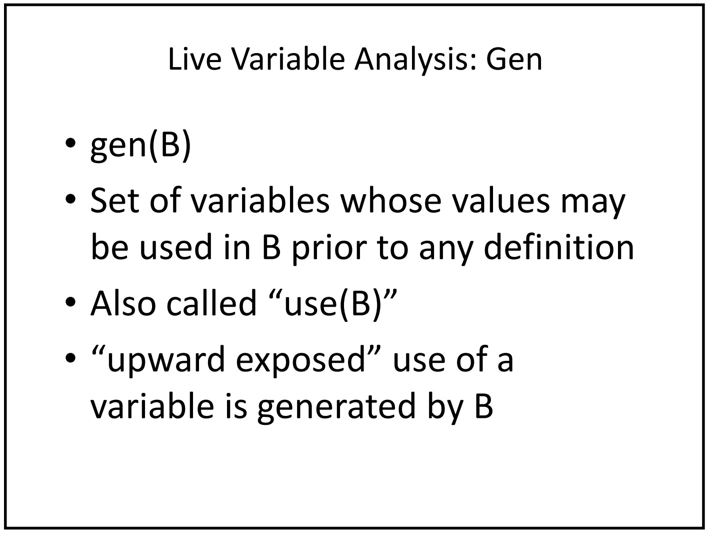




Live Variable Analysis

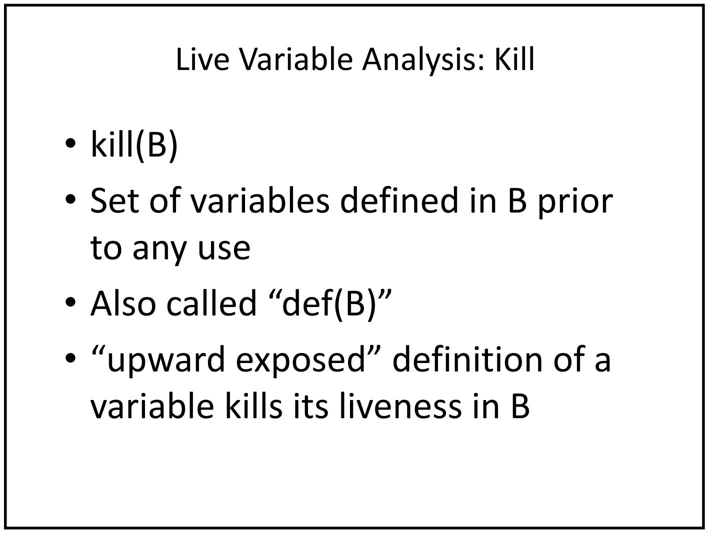
- A variable x is *live* at a point p if
  - There is a point p’ along **some** path in the flow graph starting at p to the EXIT
  - Value of x could be used at p’
  - There is no definition of x between p and p’ along **this** path
- Otherwise x is *dead* at p

33



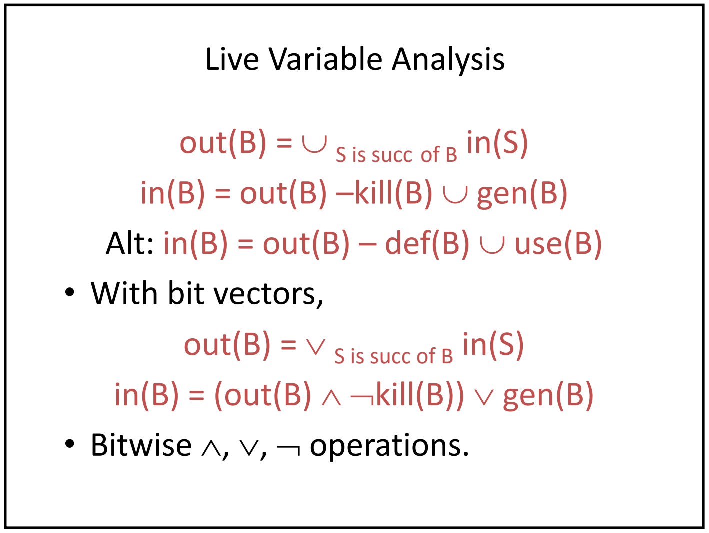
Live Variable Analysis: Gen

- gen(B)
- Set of variables whose values may be used in B prior to any definition
- Also called “use(B)”
- “upward exposed” use of a variable is generated by B



Live Variable Analysis: Kill

- kill(B)
- Set of variables defined in B prior to any use
- Also called “def(B)”
- “upward exposed” definition of a variable kills its liveness in B



Live Variable Analysis

$$out(B) = \cup_{\text{S is succ of B}}\ in(S)$$

$$in(B) = out(B) - kill(B) \cup gen(B)$$

Alt: $in(B) = out(B) - def(B) \cup use(B)$

- With bit vectors,

$$out(B) = \vee_{\text{S is succ of B}}\ in(S)$$

$$in(B) = (out(B) \wedge \neg kill(B)) \vee gen(B)$$

- Bitwise $\wedge$, $\vee$, $\neg$ operations.

Very Busy Expressions

- Expression e is very busy at a point p
  - –**Every** path from p to exit has at least one evaluation of e
  - –There is no assignment to any component variable of e **before first evaluation** of e following p
- Also called ***Anticipable*** expression

Very Busy Expression

- Practice Assignment
  - –Set the data flow equations for Very Busy Expression
  - –Hint: Available Expression Analysis