Topic-I-C

Dataflow Analysis

Global Dataflow Analysis

Motivation

We need to know variable ***def*** and ***use*** information between basic blocks for:

- constant folding
- dead-code elimination
- redundant computation elimination
- code motion
- induction variable elimination
- build data dependence graph (DDG)
- etc.

Topics of DataFlow Analysis

- Reaching definition
- Live variable analysis
- ud-chains and du-chains
- Available expressions
- Others ..

Definition and Use

1. **Definition & Use**

$$S: v_1 = \ ... v_2$$

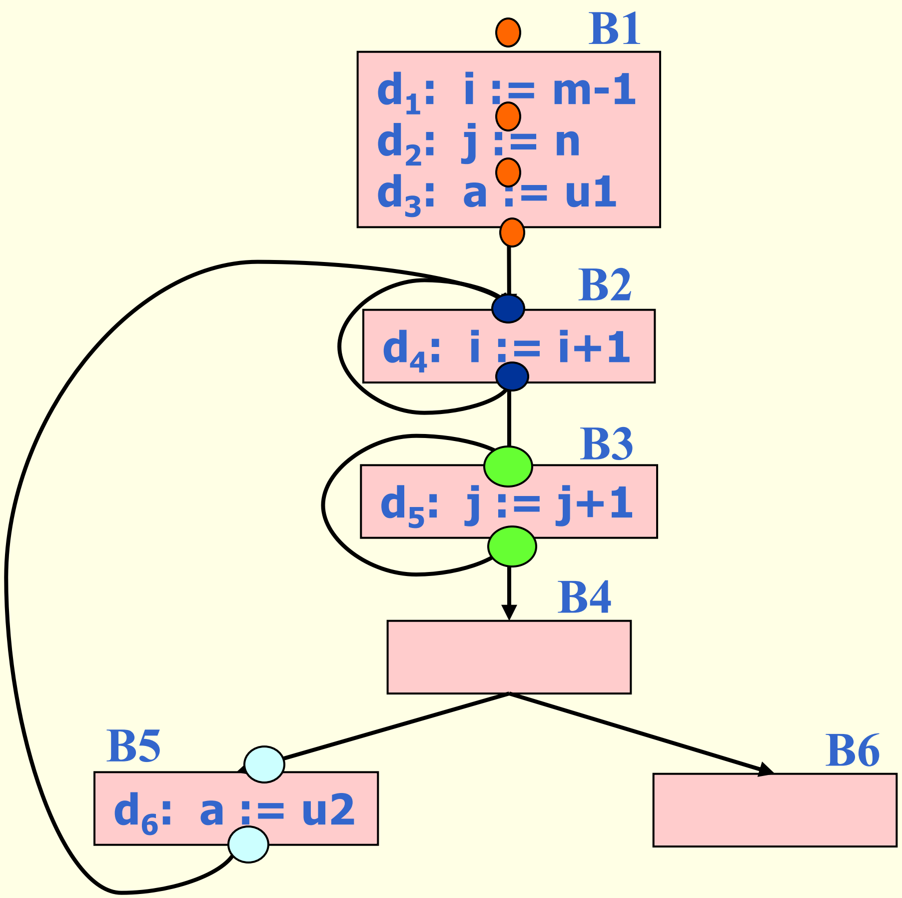
S is a "definition" of v_1

S is a "use" of v_2

Compute Def and Use Information of a Program P?

- Case 1: P is a basic block ?
- Case 2: P contains more than one basic blocks ?

Points and Paths



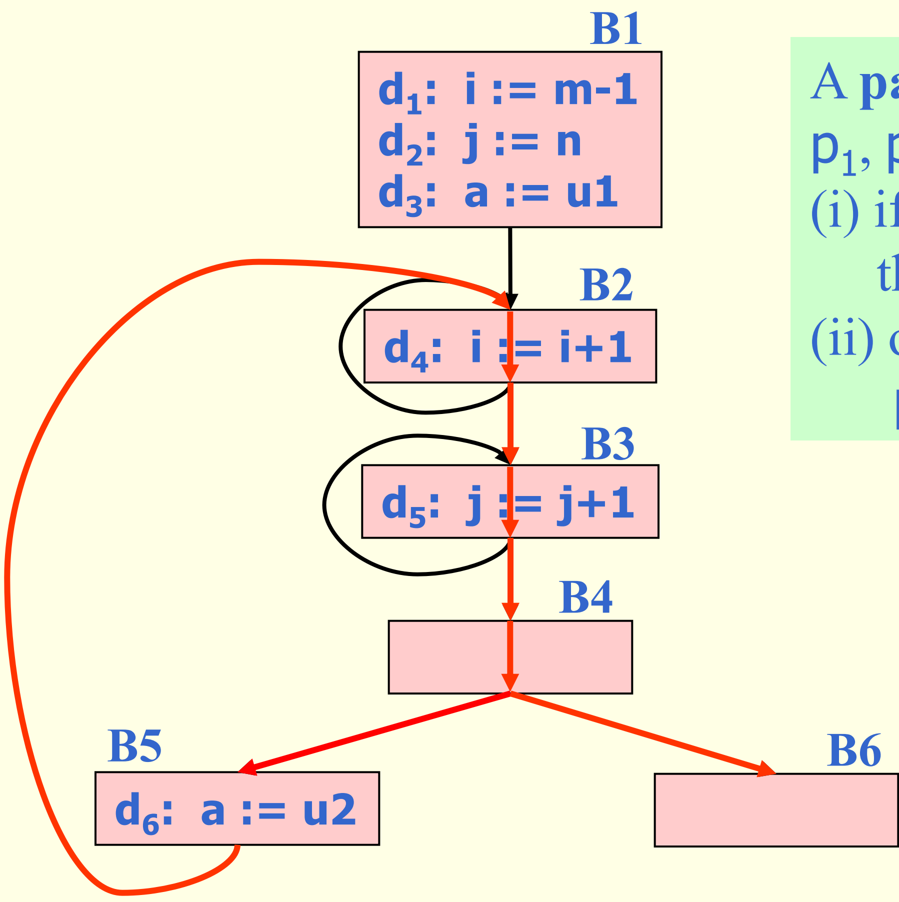
points in a basic block:

- between statements
- before the first statement
- after the last statement

In the example, how many points basic block B1, B2, B3, and B5 have?

B1 has four, B2, B3, and B5 have two points each.

Points and Paths



A **path** is a sequence of points $p_1, p_2, \ldots, p_n$ such that either:

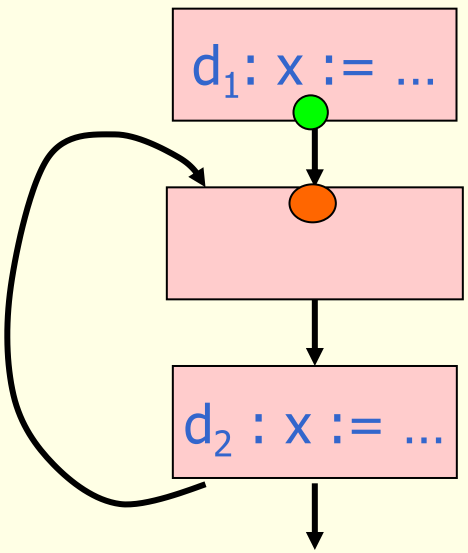
(i) if p_i immediately precedes S, than p_{i+1} immediately follows S.

(ii) or p_i is the end of a basic block and p_{i+1} is the beginning of a successor block

In the example, is there a path from the beginning of block B5 to the beginning of block B6?

Yes, it travels through the end point of B5 and then through all the points in B2, B3, and B4.

Reach and Kill



Kill

a definition $\mathbf{d_1}$ of a variable **v** is killed between $\mathbf{p_1}$ and $\mathbf{p_2}$ if <u>in every path</u> from $\mathbf{p_1}$ to $\mathbf{p_2}$ there is another definition of **v**.

Reach

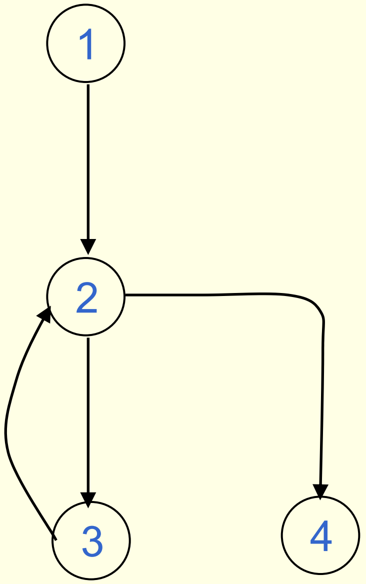
a definition **d** reaches a point **p** if $\exists$ a path $\mathbf{d} \rightarrow \mathbf{p}$, and **d** is not killed along the path

In the example, do d_1 and d_2 reach the points and ?

both d_1, d_2 reach point
but only d_1 reaches point

Reach Example

B1	*S0:*	PROGRAM
	S1:	READ(L)
	S2:	N = 0
	S3:	K = 0
	S4:	M = 1
B2	*S5:*	K = K + M
	S6:	C = K > L
	S7:	IF C THEN GOTO S11
B3	*S8:*	N = N + 1
	S9:	M = M + 2
	S10:	GOTO S5
B4	*S11:*	WRITE(N)
	S12:	END



The set of defs reaching the use of *N* in S8: {S2, S8}

def S2 reach S11 along statement path: (S2, S3, S4, S5, S6, S7, S11)

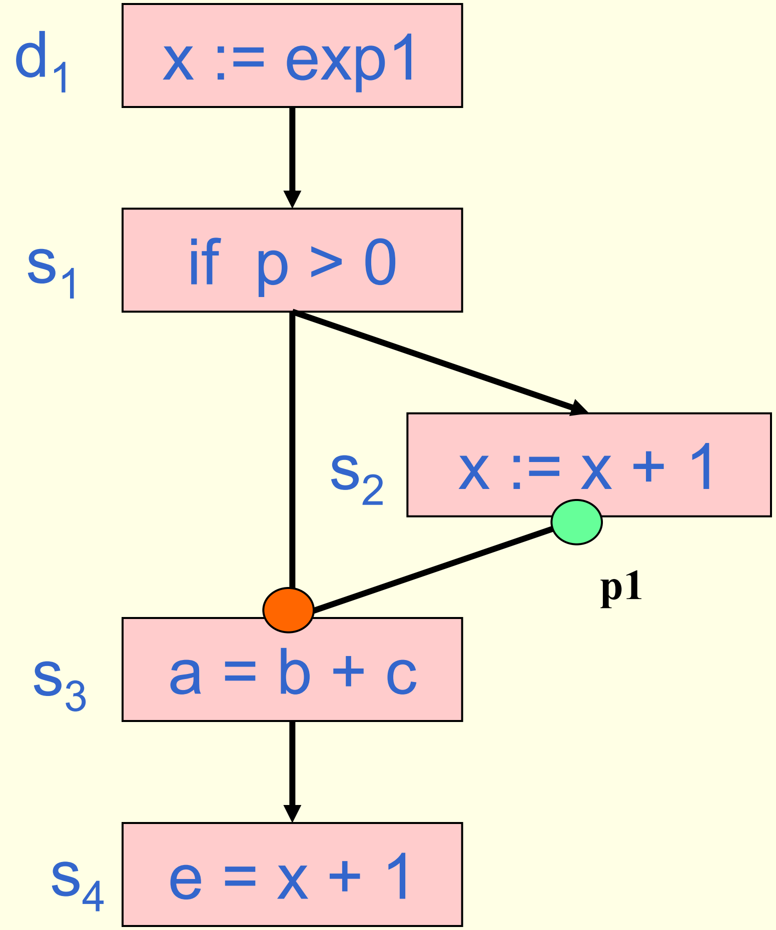
S8 reach S11 along statement path: (S8, S9, S10, S5, S6, S7, S11)

Problem Formulation:
Example 1

Can d_1 reach point p_1?

```
d1    x := exp1
s1    if  p > 0
s2        x := x + 1      <——    p1
s3    a = b + c
s4    e = x + 1
```

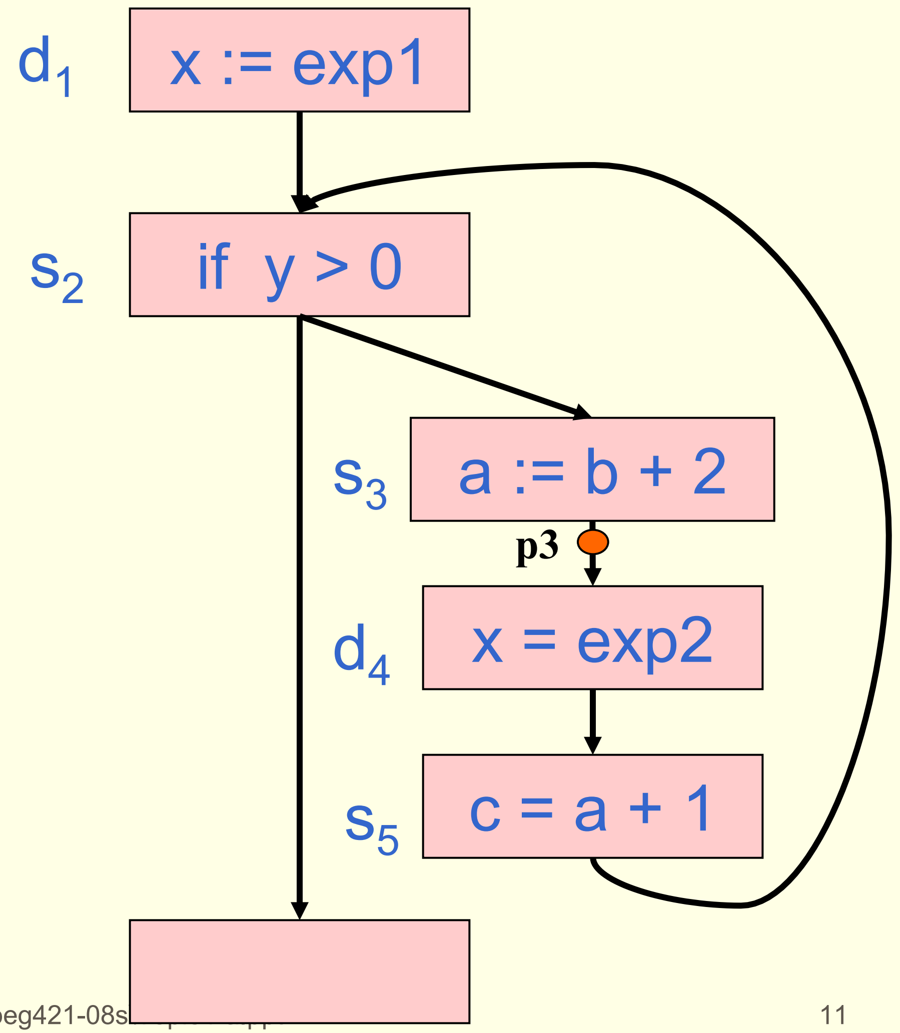
It depends on what point p_1 represents!!!



Problem Formulation:
Example 2

Can d_1 and d_4 reach point p_3?

d_1	x := exp1
s_2	while y > 0 do
s_3	a := b + 2 ← p_3
d_4	x := exp2
s_5	c := a + 1
	end while



Data-Flow Analysis of Structured Programs

Structured programs have an useful property: there is *a single point of entrance and a single exit point* for each statement.

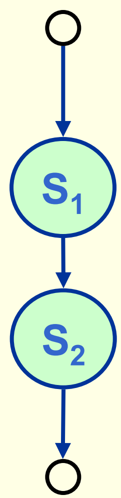
We will consider program statements that can be described by the following syntax:

Statement $\rightarrow$ **id** := Expression
| Statement ; Statement
| **if** Expression **then** Statement **else** Statement
| **do** Statement **while** Expression
Expression $\rightarrow$ id + id
| id

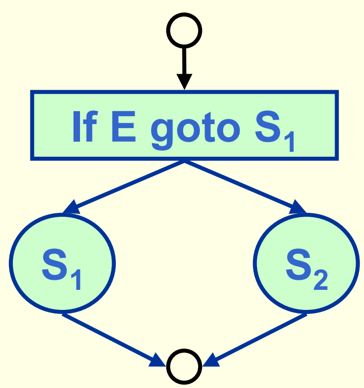
Structured Programs

S ::= **id** := E
| S ; S
| **if** E **then** S **else** S
| **do** S **while** E
E ::= id + id
| id

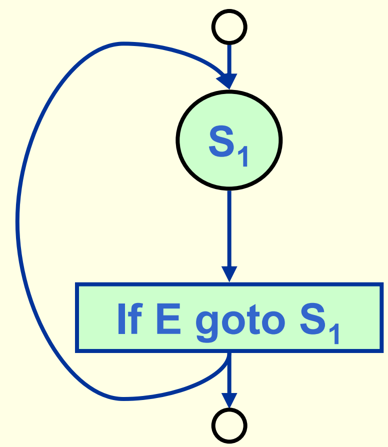
This restricted syntax results in the forms depicted below for flowgraphs



$S_1\ \ S_2$



if E **then** S_1 **else** S_2



do S_1 **while** E

Data-Flow Values

1. Each program point associates with a data-flow value
2. A data-flow value represents the possible program states that can be observed for that program point.
3. The data-flow value depends on the goal of the analysis.

Given a statement S, $in(S)$ and $out(S)$ denote the data-flow values before and after S, respectively.

Data-Flow Values of Basic Block

Assume basic block B consists of statement $s_1, s_2, \ldots, s_n$ (s_1 is the first statement of B and s_n is the last statement of B), the data-flow values immediately before and after B is denoted as:

$$in(B) = in(s_1)$$
$$out(B) = out(s_n)$$

Instances of Data-Flow Problems

- **Reaching Definitions**
- **Live-Variable Analysis**
- **DU Chains and UD Chains**
- **Available Expressions**

To solve these problems we must take into consideration the data-flow and the control flow in the program. A common method to solve such problems is to create a set of data-flow equations.

Iterative Method for Dataflow Analysis

- Establish a set of dataflow relations for each basic block
- Establish a set dataflow equations between basic blocks
- Establish an initial solution
- Iteratively solve the dataflow equations, until a ***fixed point*** is reached.

Generate set: gen(S)

In general, $d \in gen(S)$ if d reaches the end of S independent of whether it reaches the beginning of S.

We restrict $gen(S)$ contains only the definition in S.

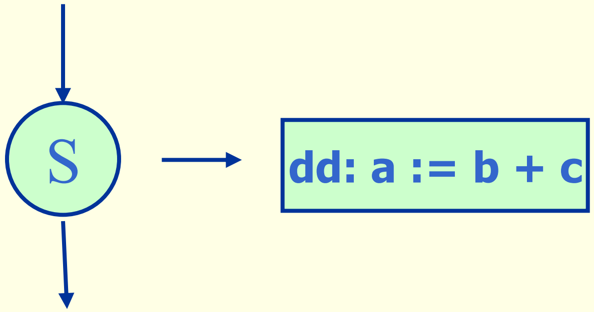
If S is a basic block, $\mathrm{gen}(S)$ contains all the definitions inside the basic block that are "visible" immediately after the block.

Kill Set: kill(S)

$\boldsymbol{d} \in kill(\boldsymbol{S}) \Rightarrow d$ never reaches the end of $\boldsymbol{S}$.

This is equivalent to say:

$\boldsymbol{d}$ reaches end of $\boldsymbol{S} \Rightarrow \boldsymbol{d} \notin kill(S)$



d1 a :=
d2 a :=
⋮
dk a :
dd a :=

//

$$kill(s) = D_a - \{ dd \}$$

Of course the statements d1,d2, …, dk all get killed except dd itself.

A basic block's kill set is simply the union of all the definitions killed by its individual statements!

Reaching Definitions

Problem Statement:

Given a program and a program point determine the set of definitions reaching this point in a program.

Iterative Algorithm for Reaching Definitions

dataflow equations

The set of definitions reaching the entry of basic block *B*:

$$in(B) = \bigcup_{P \in predecessor(B)} out(P)$$

The set of definitions reaching the exit of basic block *B*:

$$out(B) = gen(B) \cup \{ in(B) - \text{kill}(\mathbf{B})\}$$

(AhoSethiUllman, pp. 606)

Iterative Algorithm for Reaching Definitions

Algorithm

1) $out(ENTRY) = \varnothing$;

2) for (each basic block B other than $ENTRY$)

   $out(B) = \varnothing$;

3) while (changes to any out occur)

4) for (each B other than $ENTRY$)

{

$$in(B) = \cup_{P \in predecessors\ of\ B}\ out(P)\ ;$$

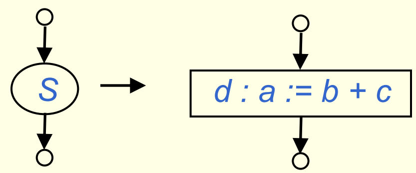
$$out(B) = gen(B) \cup \left(in(B) - kill(b)\right);$$

}

Need a flag to test if a out is changed! The initial value of the flag is true.

Dataflow Equations – a simple case

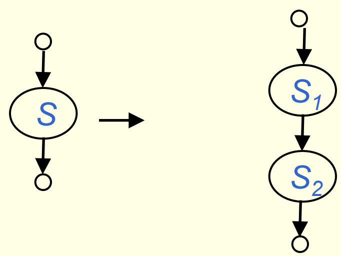
Date-flow equations for reaching definitions



$$gen(S) = \{d\}$$

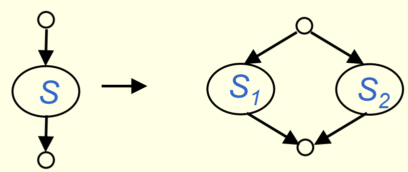
$$kill(S) = Da - \{d\}$$

Da is the set of all definitions in the program for variable a!



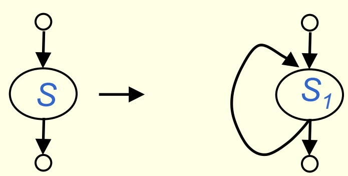
$$gen(S) = gen(S_2) \cup (gen(S_1) - kill(S_2))$$

$$kill[S] = kill(S_2) \cup (kill(S_1) - gen(S_2))$$



$$gen(S) = gen(S_1) \cup gen(S_2)$$

$$kill(S) = kill(S_1) \cap kill(S_2)$$

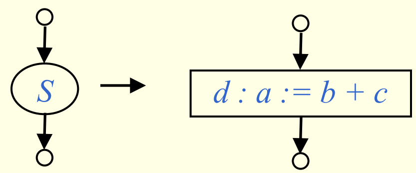


$$gen(S) = gen(S_1)$$

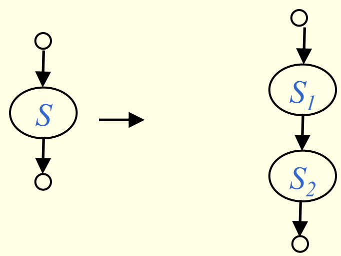
$$kill(S) = kill(S_1)$$

Dataflow Equations

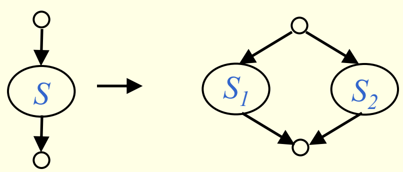
Date-flow equations for reaching definitions



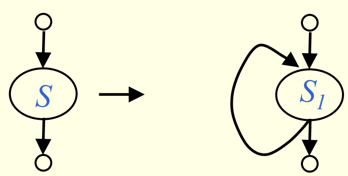
$$out(S) = gen(S) \cup (in(S) - kill(S))$$



$$\begin{aligned} in(S) &= in(S_1) \\ in(S_2) &= out(S_1) \\ out(S) &= out(S_2) \end{aligned}$$



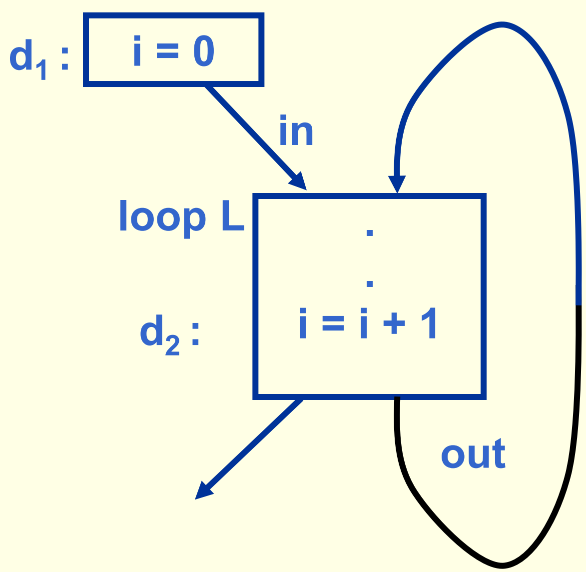
$$\begin{aligned} in(S) &= in(S_1) = in(S_2) \\ out(S) &= out(S_1) \cup out(S_2) \end{aligned}$$



$$\begin{aligned} in(S) &= in(S_1) \cup out(S_1) \\ out(S) &= out(S_1) \end{aligned}$$

Dataflow Analysis: An Example

Using RD (reaching def) as an example:



Question:

What is the set of reaching definitions at the exit of the loop L?

$$in\ (L) = \{d_1\} \cup out(L)$$
$$gen\ (L) = \{d_2\}$$
$$kill\ (L) = \{d_1\}$$
$$out(L) = gen(L) \cup \{in(L) - kill(L)\}$$

$in(L)$ depends on $out(L)$, and $out(L)$ depends on $in(L)$!!

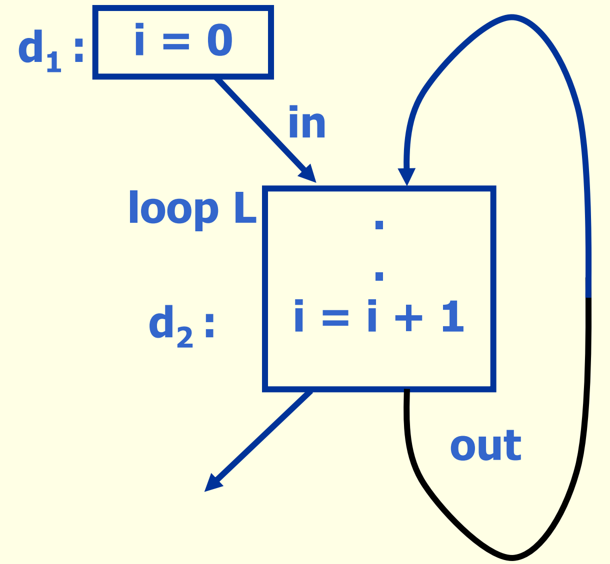
Initialization

out[L] = $\varnothing$

Solution?

First iteration

$$\begin{aligned} out(L) &= gen(L) \cup (in(L) - kill(L)) \\ &= \{d_2\} \cup (\{d_1\} - \{d_1\}) \\ &= \{d_2\} \end{aligned}$$



Second iteration

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

but now:

$$\begin{aligned} in(L) &= \{d_1\} \cup out(L) = \{d_1\} \cup \{d_2\} \\ &= \{d_1, d_2\} \end{aligned}$$

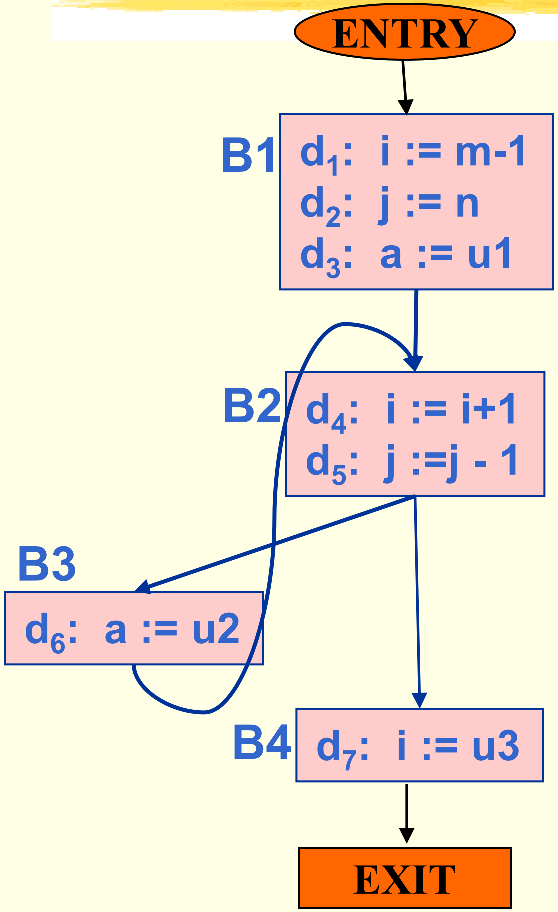
therefore:

$$\begin{aligned} out(L) &= \{d_2\} \cup (\{d_1, d_2\} - \{d_1\}) \\ &= \{d_2\} \cup \{d_2\} \\ &= \{d_2\} \end{aligned}$$

$$\begin{aligned} in(L) &= \{d_1\} \cup out(L) \\ gen(L) &= \{d_2\} \\ kill(L) &= \{d_1\} \\ out(L) &= gen(L) \cup \{in(L) - kill(L)\} \end{aligned}$$

So, we reached the fixed point!

Reaching Definitions: Another Example



Step 1: Compute gen and kill for each basic block

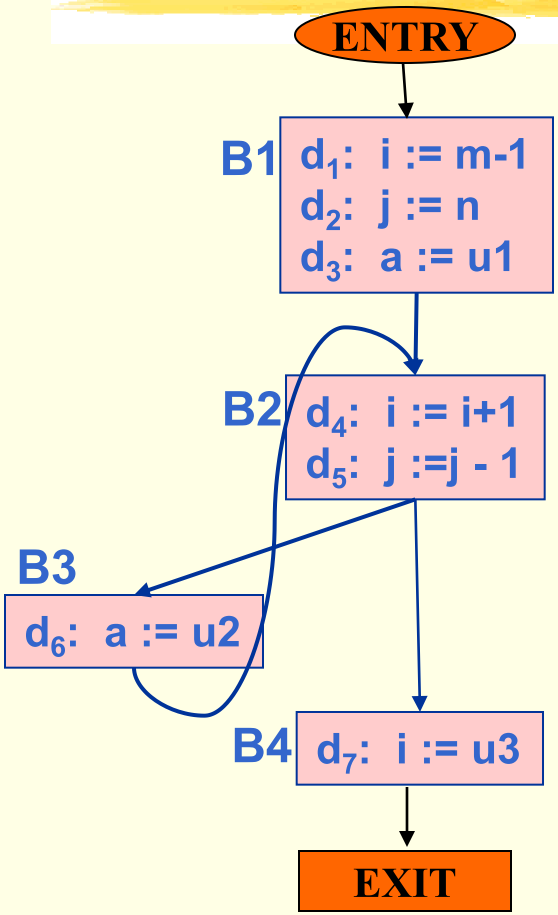
$$gen(B_1) = \{d_1, d_2, d_3\}$$
$$kill(B_1) = \{d_4, d_5, d_6, d_7\}$$

$$gen(B_2) = \{d4, d5\}$$
$$kill(B_2) = \{d1, d2, d7\}$$

$$gen(B_3) = \{d6\}$$
$$kill(B_3) = \{d3\}$$

$$gen(B_4) = \{d7\}$$
$$kill(B_4) = \{d1, d4\}$$

Reaching Definitions: Another Example (Con't)



Step 2: For every basic block, make:

out[B] = ∅

Initialization:

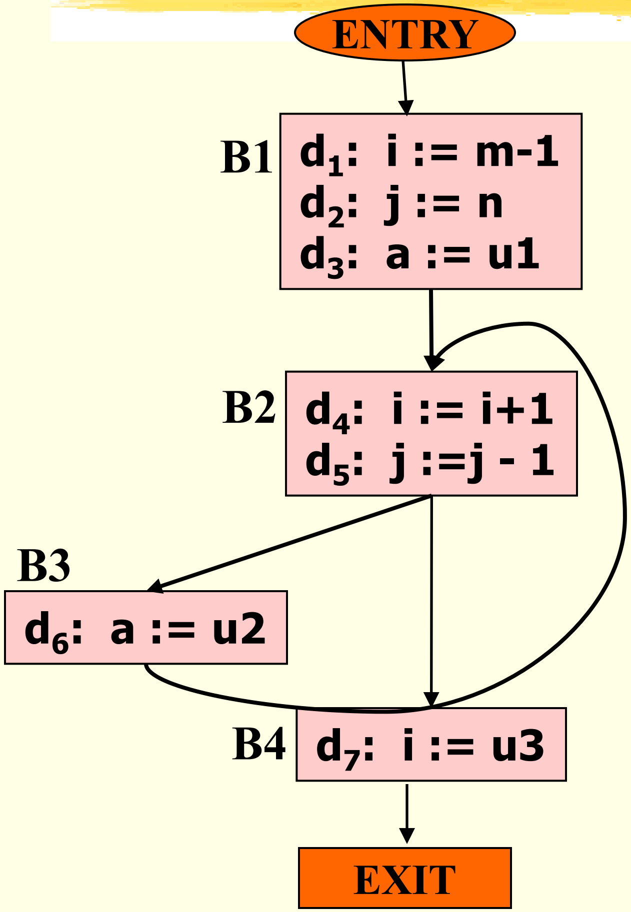
$$out(B_1) = \varnothing$$

$$out(B_2) = \varnothing$$

$$out(B_3) = \varnothing$$

$$out(B_4) = \varnothing$$

Reaching Definitions: Another Example (Con't)



To simplify the representation, the in[B] and out[B] sets are represented by bit strings. Assuming the representation $d_1d_2d_3\, d_4d_5d_6d_7$ we obtain:

Initialization:

$$out(B_1) = \varnothing$$

$$out(B_2) = \varnothing$$

$$out(B_3) = \varnothing$$

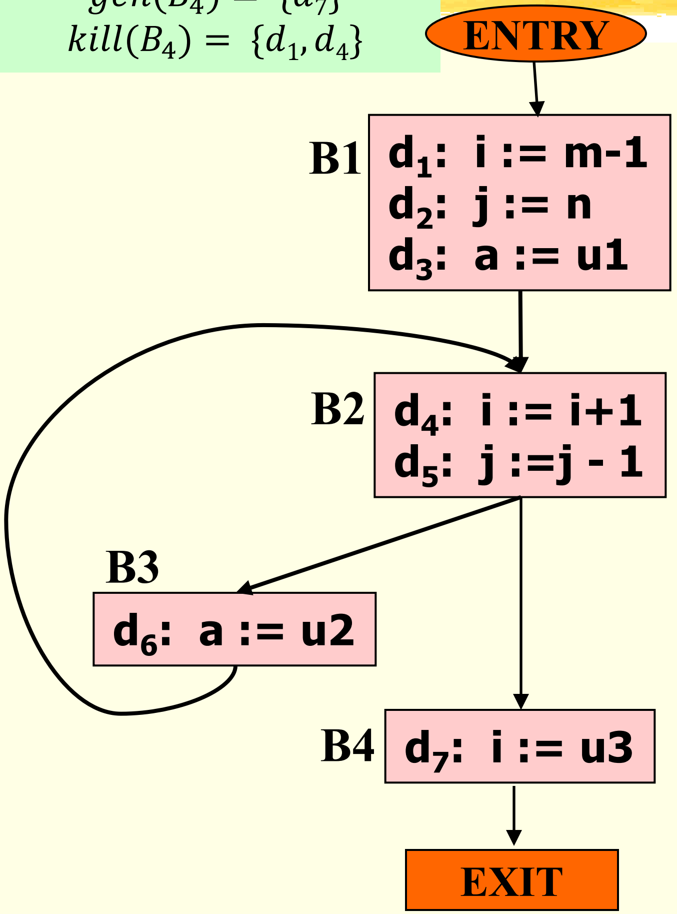
$$out(B_4) = \varnothing$$

Block	Initial	
	in[B]	out[B]
B_1		000 0000
B_2		000 0000
B_3		000 0000
B_4		000 0000

Notation: $d_1d_2d_3\, d_4d_5d_6d_7$

$gen(B_2) = \{d_1, d_2, d_3\}$
$kill(B_1) = \{d_4, d_5, d_6, d_7\}$
$gen(B_2) = \{d_4, d_5\}$
$kill(B_2) = \{d_1, d_2, d_7\}$
$gen(B_3) = \{d_6\}$
$kill(B_3) = \{d_3\}$
$gen(B_4) = \{d_7\}$
$kill(B_4) = \{d_1, d_4\}$

aching Definitions: her Example (Con't)



Notation: $d_1 d_2 d_3\ d_4 d_5 d_6 d_7$

while a fixed point is not found:

$$in(B) = \bigcup_{P \in pred(B)} out(P)$$

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

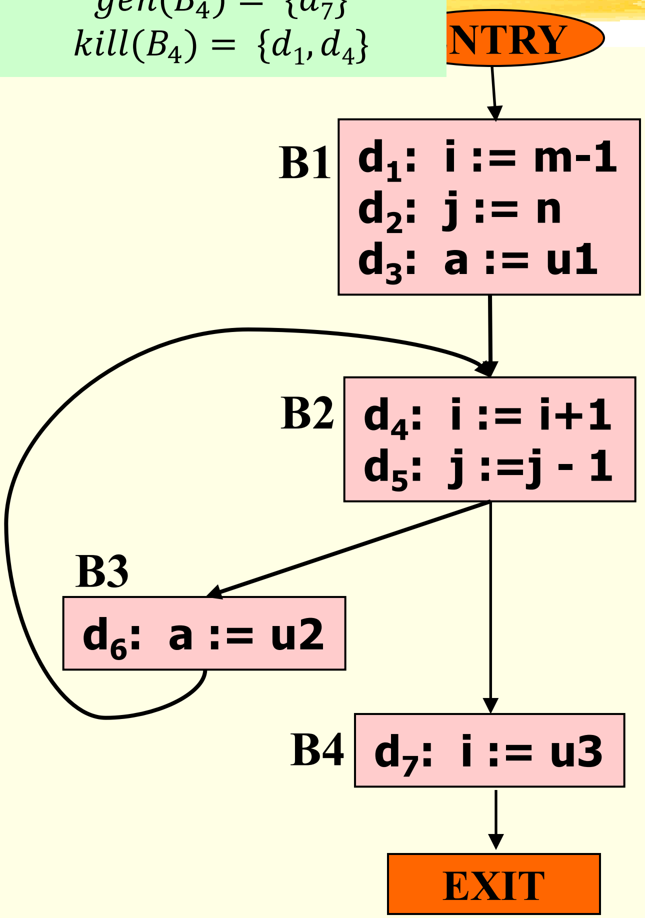
Block	Initial	
	in[B]	out[B]
B_1		000 0000
B_2		000 0000
B_3		000 0000
B_4		000 0000

Block	First Iteration	
	in[B]	out[B]
B_1	000 0000	111 0000
B_2	000 0000	000 1100
B_3	000 0000	000 0010
B_4	000 0000	000 0001

out(B) = gen(B)

$$gen(B_2) = \{d_1, d_2, d_3\}$$
$$kill(B_1) = \{d_4, d_5, d_6, d_7\}$$
$$gen(B_2) = \{d_4, d_5\}$$
$$kill(B_2) = \{d_1, d_2, d_7\}$$
$$gen(B_3) = \{d_6\}$$
$$kill(B_3) = \{d_3\}$$
$$gen(B_4) = \{d_7\}$$
$$kill(B_4) = \{d_1, d_4\}$$

aching Definitions: her Example (Con't)



while a fixed point is not found:

$$in(B) = \bigcup_{P \in pred(B)} out(P)$$

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

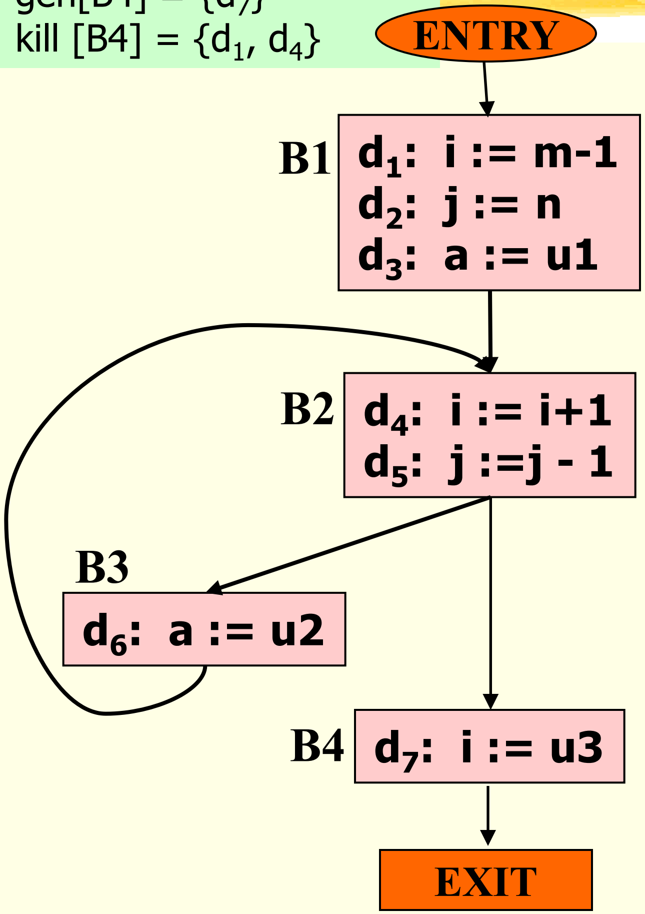
Block	First Iteration	
	in[B]	out[B]
B_1	000 0000	111 0000
B_2	000 0000	000 1100
B_3	000 0000	000 0010
B_4	000 0000	000 0001

Block	Second Iteration	
	in[B]	out[B]
B_1	000 0000	111 0000
B_2	111 0010	001 1110
B_3	000 1100	000 1110
B_4	000 1100	000 0101

Notation: $d_1 d_2 d_3\ d_4 d_5 d_6 d_7$

gen[B1] = $\{d_1, d_2, d_3\}$
kill[B1] = $\{d_4, d_5, d_6, d_7\}$
gen[B2] = $\{d_4, d_5\}$
kill [B2] = $\{d_1, d_2, d_7\}$
gen[B3] = $\{d_6\}$
kill [B3] = $\{d_3\}$
gen[B4] = $\{d_7\}$
kill [B4] = $\{d_1, d_4\}$

aching Definitions: ther Example (Con't)



Notation: $d_1d_2d_3\ d_4d_5d_6d_7$

while a fixed point is not found:

in[B] = ∪ out[P] where P is a predecessor of B

out[B] = gen[B] ∪ (in[B]-kill[B])

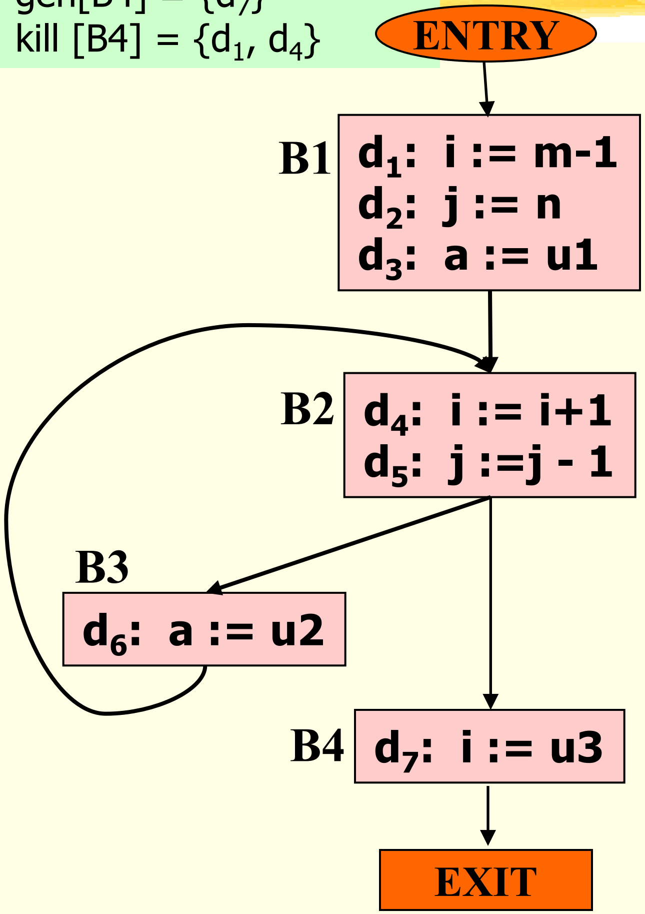
Block	Second Iteration	
	in[B]	out[B]
B_1	000 0000	111 0000
B_2	111 0010	001 1110
B_3	000 1100	000 11 10
B_4	000 1100	001 0111

Block	Third Iteration	
	in[B]	out[B]
B_1	000 0000	111 0000
B_2	111 1110	001 1110
B_3	001 1110	000 11 10
B_4	001 1110	001 0111

we reached the fixed point!

gen[B1] = {d_1, d_2, d_3}
kill[B1] = {d_4, d_5, d_6, d_7}
gen[B2] = {d_4, d_5}
kill [B2] = {d_1, d_2, d_7}
gen[B3] = {d_6}
kill [B3] = {d_3}
gen[B4] = {d_7}
kill [B4] = {d_1, d_4}

aching Definitions: ther Example (Con't)



Notation: $d_1d_2d_3\, d_4d_5d_6d_7$

while a fixed point is not found:

in[B] = ∪ out[P]   where P is a predecessor of B

out[B] = gen[B] ∪ (in[B]-kill[B])

Block	Third Iteration	
	in[B]	out[B]
B_1	000 0000	111 0000
B_2	111 0010	001 1110
B_3	000 1100	000 1110
B_4	000 1100	001 0111

Block	Forth Iteration	
	in[B]	out[B]
B_1	000 0000	111 0000
B_2	111 1110	001 1110
B_3	001 1110	000 1110
B_4	001 1110	001 0111

we reached the fixed point!

Other Applications of Data flow Analysis

- Live Variable Analysis
- DU and UD Chains
- Available Expressions
- Constant Propagation
- Constant Folding
- Others ..

Live Variable Analysis: Another Example of Flow Analysis

- A variable V is *live* at the exit of a basic block n, if there is a ***def-free*** path from n to an outward exposed use of V in a node n'.

  "Live variable analysis problem" – determine the set of variables which are live at the exit from each program point.

Live variable analysis is a "backwards dataflow" analysis, that is the analysis is done in a backwards order .

Live Variable Analysis: Another Example of Flow Analysis

L1: b := 3;

L2: c := 5;

L3: a := b + c;

goto L1;

The set of live variables at line L2 is $\{b, c\}$, but the set of live variables at line L1 is only $\{b\}$ since variable "c" is updated in line 2. The value of variable "a" is never used, so the variable is never live.

Copy from Wikipedia, the free encyclopedia

Live Variable Analysis: Def and use set

- $def(B)$: *the set of variables defined in basic block* B *prior to any use of that variable in* B
- $use(B)$: *the set of variables whose values may be used in* B *prior to any definition of the variable.*

Live Variable Analysis

dataflow equations

The set of variables live at the entry of basic block ***B***:

$$in(B) = use(B) \cup \{ out(B) - def(B)\}$$

The set of variables live at the exit of basic block ***B***:

$$out(B) = \bigcup_{S \in successors(B)} in(S)$$

Iterative Algorithm for Live Variable Analysis

Algorithm

1) out(EXIT) = ∅ ;

2) for (each basic block B other than EXIT)

in(B) = ∅;

3) while (changes to any "in" occur)

4) for (each B other than EXIT)

{

$$out(B) = \bigcup_{S \in successors(B)} in(S)$$

$$in(B) = use(B) \cup \{out(B) - def(B)\}$$

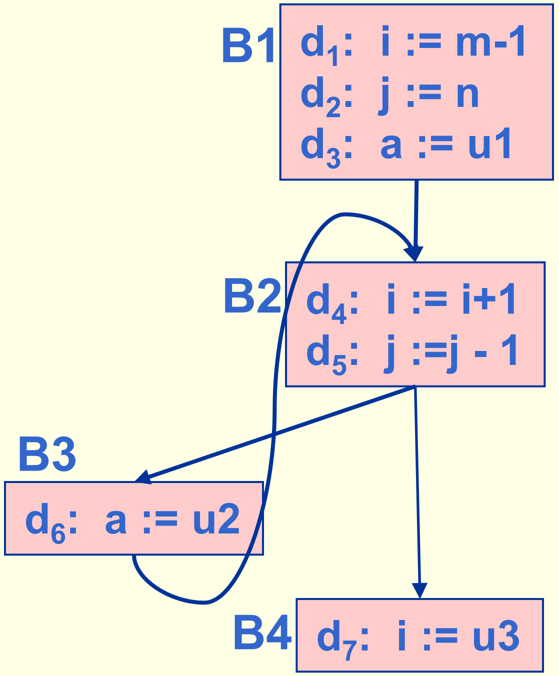
}

> Need a flag to test if a in is changed! The initial value of the flag is true.

(AhoSethiUllman, pp. 607)

Live Variable Analysis: a Quiz

Calculate the live variable sets $in(B)$ and $out(B)$ for the program:



D-U and U-D Chains

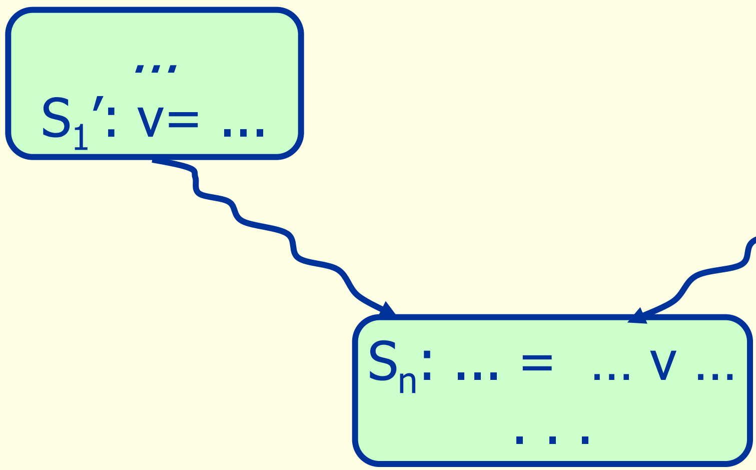
Many dataflow analyses need to find the use-sites of each defined variable or the definition-sites of each variable used in an expression.

Def-Use (D-U), and Use-Def (U-D) chains are efficient data structures that keep this information.

Notice that when a code is represented in Static Single-Assignment (SSA) form (as in most modern compilers) there is no need to maintain D-U and U-D chains.

UD Chain

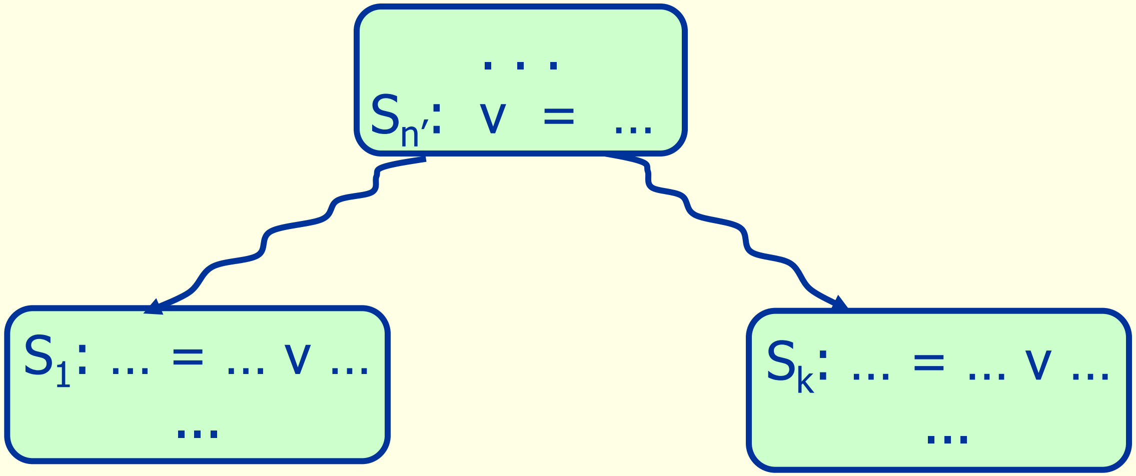
An UD chain is a list of all definitions that can reach a given use of a variable.



A **UD chain**: $UD(S_n, v) = (S_1', \ldots, S_m')$.

DU Chain

A DU chain is a list of all uses that can be reached by a given definition of a variable. DU Chain is a counterpart of a *UD Chain*.



A **DU chain**: $DU(S_n', v) = (S_1, \ldots, S_k)$.

Use of DU/UD Chains in Optimization/Parallelization

- Dependence analysis
- Live variable analysis
- Alias analysis
- Analysis for various transformations

Available Expressions

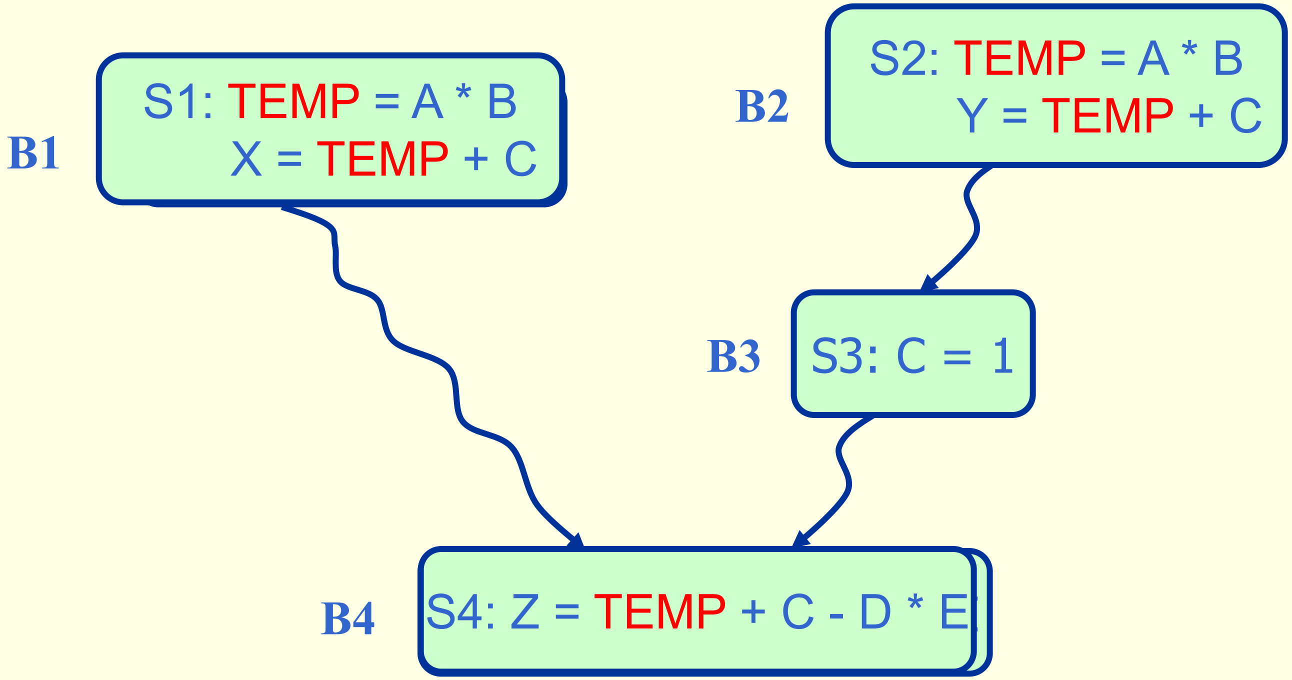
An expression $x + y$ is *available* at a point p if:

1. Every path from the $start$ node to p evaluates $x + y$.

2. After the last evaluation prior to reaching p, there are no subsequent assignments to x or y.

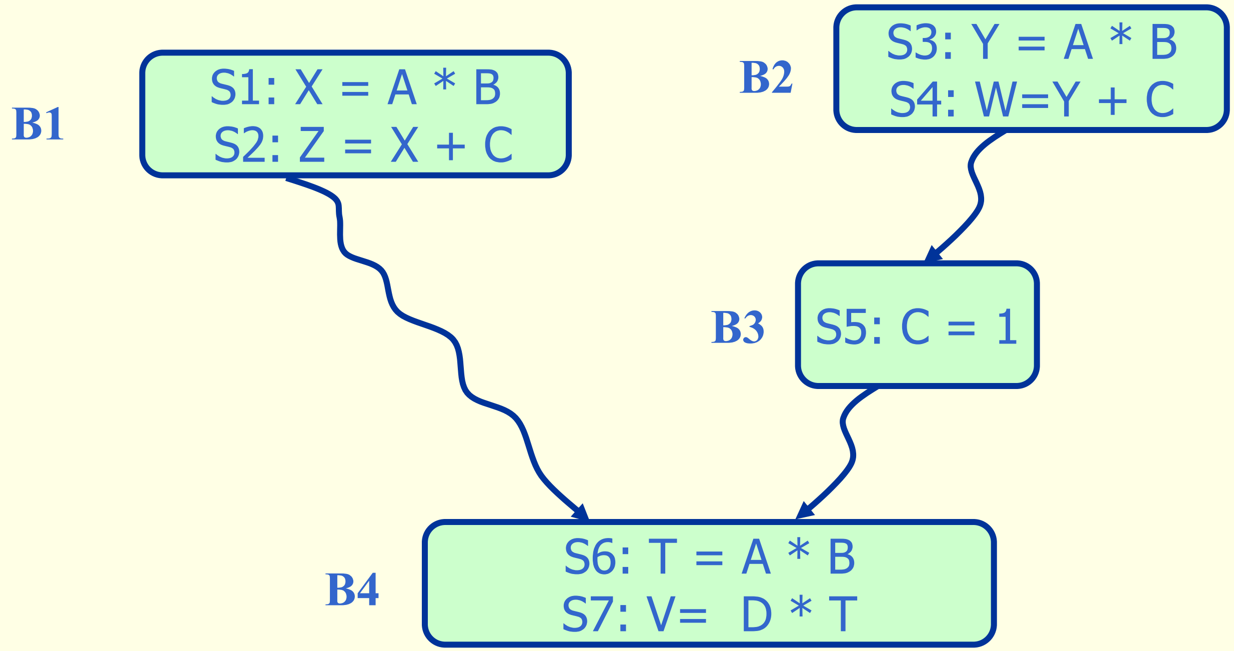
We say that a basic block kills expression $x + y$ if it may assign x or y, and does not subsequently recompute $x + y$.

Available Expression: Example



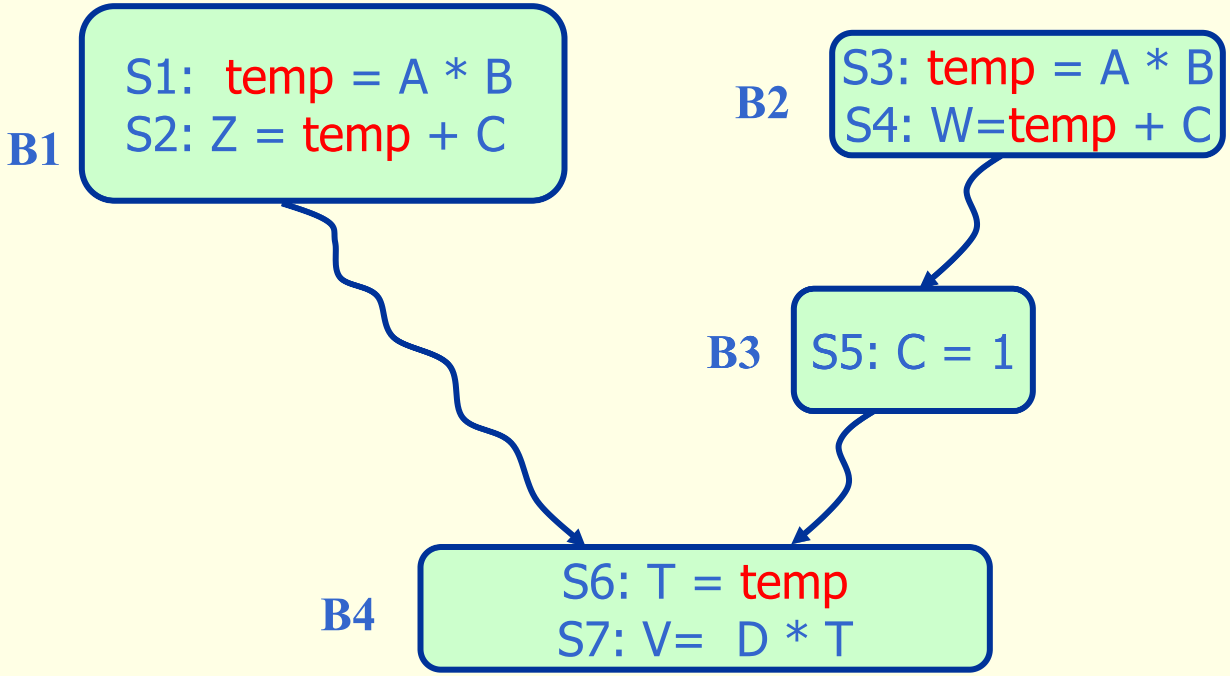
Yes. It is generated in all paths leading to **B4** and it is not killed after its generation in any path. Thus the redundant expression can be eliminated.

Available Expression: Example



Yes. It is generated in all paths leading to **B4** and it is not killed after its generation in any path. Thus the redundant expression can be eliminated.

Available Expression: Example



Yes. It is generated in all paths leading to **B4** and it is not killed after its generation in any path. Thus the redundant expression can be eliminated.

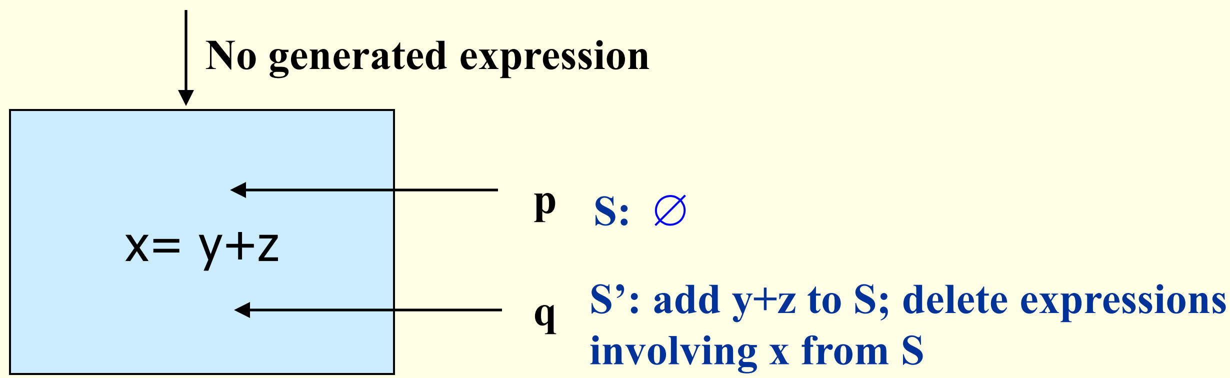
Available Expressions: gen and kill set

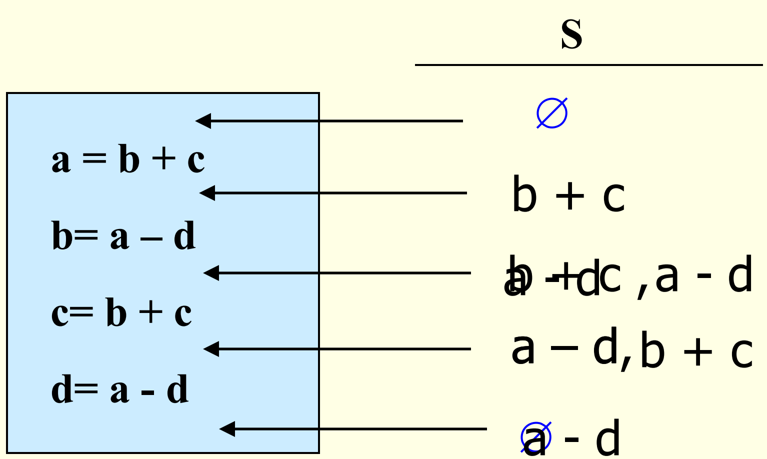
Assume U is the "universal" set of all expressions appearing on the right of one or more statements in a program.

$e_{gen}(B)$: *the set of expressions generated by* B

$e_{kill}(B)$: *the set of expressions in* U *killed in* B.

Calculate the Generate Set of Available Expressions





Iterative Algorithm for Available Expressions

dataflow equations

The set of expressions available at the entry of basic block B:

$$in(B) = \bigcap_{P \in predecessors(B)} out(P)$$

The set of expressions available at the exit of basic block B:

$$out(B) = e_{gen}(B) \cup \{\, in(B) - e_{kill}(B)\}$$

(AhoSethiUllman, pp. 606)

Iterative Algorithm for Reaching Definitions

Algorithm

1. $out(ENTRY) = \emptyset;$
2. *For each basic block B other than ENTRY*

$$out(B) = U$$

3. *While changes to any out occur*
4. *for each B other than ENTRY*

Need a flag to test if a out is changed! The initial value of the flag is true.

{

$$in(B) = \bigcap_{P \in predecessors(B)} out(P)$$

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

}

(AhoSethiUllman, pp. 607)

Use of Available Expressions

- Detecting global common subexpressions

More Useful Data-Flow Frameworks

Constant propagation is the process of substituting the values of known constants in expressions at compile time.

Constant folding is a compiler optimization technique where constant sub-expressions are evaluated at compiler time.

Constant Folding Example

i = 32*48-1530 → i = 6

Constant folding can be implemented :

- In a compiler's front end on the IR tree (before it is translted into three-address codes
- In the back end, as an adjunct to constant propagation

Constant Propagation Example

```
int x = 14;
int y = 7 - x / 2;

return y * (28 / x + 2);
```

↓ **Constant propagation**

```
int x = 14;
int y = 7 - 14 / 2;
return y * (28 / 14 +
2);
```

↓ **Constant folding**

```
int x = 14;
int y = 0;
return 0;
```

Summary

- Basic Blocks
- Control Flow Graph (CFG)
- Dominator and Dominator Tree
- Natural Loops
- Program point and path
- Dataflow equations and the iterative method
- Reaching definition
- Live variable analysis
- Available expressions

Remarks of Mathematical Foundations on Solving Dataflow Equations

- As long as the dataflow value domain is "nice" (e.g. semi-lattice)
- And each function specified by the dataflow equation is "nice" -- then iterative application of the dataflow equations at each node will eventually terminate with a stable solution (a fix point).
- For mathematical foundation -- read
  - Ken Kenedy: "*A Survey of Dataflow Analysis Techniques", In Programm Flow Analysis: Theory and Applications, Ed. Muchnik and Jones, Printice Hall, 1981.*
  - Muchnik's book*:* Section 8.2, pp 223

For a good discussion: also read 9.3 (pp 618-632) in new Dragon Book

Algorithm Convergence

Intuitively we can observe that the algorithm converges to a fix point because the $\boldsymbol{out(B)}$ set never decreases in size.

It can be shown that an upper bound on the number of iterations required to reach a fix point is the number of nodes in the flow graph.

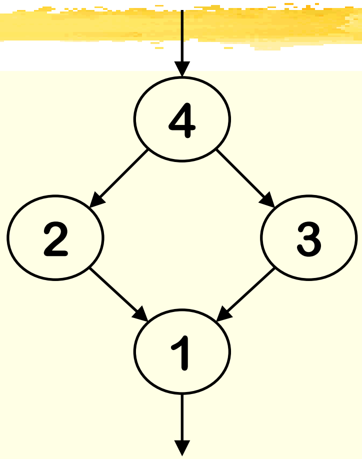
> Intuitively, if a definition reaches a point, it can only reach the point through a cycle free path, and no cycle free path can be longer than the number of nodes in the graph.

Empirical evidence suggests that for real programs the number of iterations required to reach a fix point is less then five.

More remarks

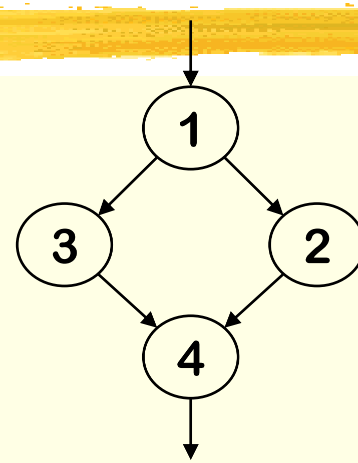
- If a data-flow framework meets "good" conditions then it has a unique fixed-point solution
- The iterative algorithm finds the (best) answer
- The solution does not depend on order of computation
- Algorithm can choose an order that converges quickly

Ordering the Nodes to Maximize Propagation



Postorder

Visit children first



Reverse Postorder

Visit parents first

- Reverse postorder visits predecessors before visiting a node
- Use reverse preorder for backward problems
  - Reverse postorder on reverse CFG is reverse preorder

Iterative solution to general data-flow frameworks

- INPUT: A data-flow framework with the following components:
  - 1. A data-flow graph, with specially labeled ENTRY and EXIT nodes,
  - 2. A direction of the data-flow *D*,
  - 3. A set of values *V*,
  - 4. A meet operator $\wedge$,
  - 5. A set of functions F, where f_B in *F* is the transfer function for block *B*, and
  - 6. A constant value v_{ENTRY} or v_{EXIT} *in V*, representing the boundary condition for forward and backward frameworks, respectively.
- OUTPUT: Values in V for IN[*B*] and OUT[*B*] for each block *B* in the data-flow graph.

(From p925 in Dragon Book Edition 2)

Iterative algorithm for a forward data-flow problem

1. $\boldsymbol{OUT(ENTRY) = V_{ENTRY};}$
2. $\boldsymbol{for}$ $(each\ basic\ block\ \boldsymbol{B}\ other\ than\ \boldsymbol{ENTRY})$

   $\boldsymbol{OUT(B) = T;}$
3. $\boldsymbol{while}$ $(changes\ to\ any\ \boldsymbol{OUT}\ occur)$
4. $\quad \boldsymbol{for}$ $(each\ basic\ block\ \boldsymbol{B}\ other\ than\ \boldsymbol{ENTRY})\ \{$

$$\boldsymbol{IN(B) = \bigcap_{P \in predecessors(B)} OUT(P);}$$

$$\boldsymbol{OUT(B) = f_B(\ IN(B)\ )}$$

$\}$

(From p926 in Dragon Book Edition 2)

Iterative algorithm for a backward data-flow problem

1. $IN(EXIT) = V_{EXIT}$;
2. for $(each\ basic\ block\ \boldsymbol{B}\ other\ than\ \boldsymbol{EXIT})$

   $IN(B) = T$;
3. $while$ $(changes\ to\ any\ \boldsymbol{IN}\ occur)$
4. for $(each\ basic\ block\ \boldsymbol{B}\ other\ than\ \boldsymbol{EXIT})$ {

   $OUT(B) = \bigcap_{S \in successors(B)} IN(S)$

   $IN(B) = f_B(\ OUT(B)\ )$;

   }

(From p926 in Dragon Book Edition 2)