SOEN 6431
Software Comprehension and Maintenance

Week 4

Program Analysis: Data flow Analysis

Outline

- Motivations.
- Program Analysis.
- Static Analysis.
- Dynamic Analysis
- Representation schemes
- Flow Analysis.
- Type-Based Analysis.

Motivations

- Program Analysis is needed for the purpose of:
  - Software maintenance.
  - Software quality.
  - Reverse engineering.
  - Optimizing compilers.
  - Security.
  - Certification and verification.

Program Analysis

- Program analysis:
  - Determine program/expression/statement/data properties.
  - Extract information from programs.

- Two types of analyses:
  - *Static Analysis*:
    - Analyze programs without executing them.
  - *Dynamic Analysis*:
    - Analyze programs at runtime.

What are we looking for?

- Depends on our goals and the system
  - In almost any language, we can find out information about variable usage.
  - In an OO environment, we can find out which classes use other classes, which are a base of an inheritance structure, etc.
  - Highlight possible coding errors
  - We can also find potential blocks of code that can never be executed in running the program (dead code)
  - Typically, the information extracted is in terms of entities and relationships

Entities

- Entities are individuals that live in the system, and attributes associated with them.

- Some examples:
  - Classes, along with information about their superclass, their scope, and 'where' in the code they exists.
  - Methods/functions and what their return type or parameter list is, etc.
  - Variables and what their types are, and whether or not they are static, etc.

Relationships

- Relationships are interactions between the entities in the system.
- Relationships include:
  - Classes inheriting from one another.
  - Methods in one class calling the methods of another class, and methods within the same class calling one another.
  - One variable referencing another variable.

Static Analysis

- Involves parsing the source code
- Usually creates an Abstract Syntax Tree
- Borrows heavily from compiler technology but stops before code generation
- Requires a grammar for the programming language
- Can be very difficult to get right

Example program

```
#include <iostream.h>
class Hello {
public: Hello(); ~Hello();
};
Hello::Hello()
{ cout << "Hello, world.\n"; }
Hello::~Hello()
{ cout << "Goodbye, cruel world.\n"; }
main() {
        Hello h;
        return 0;
}
```

Example Q&A

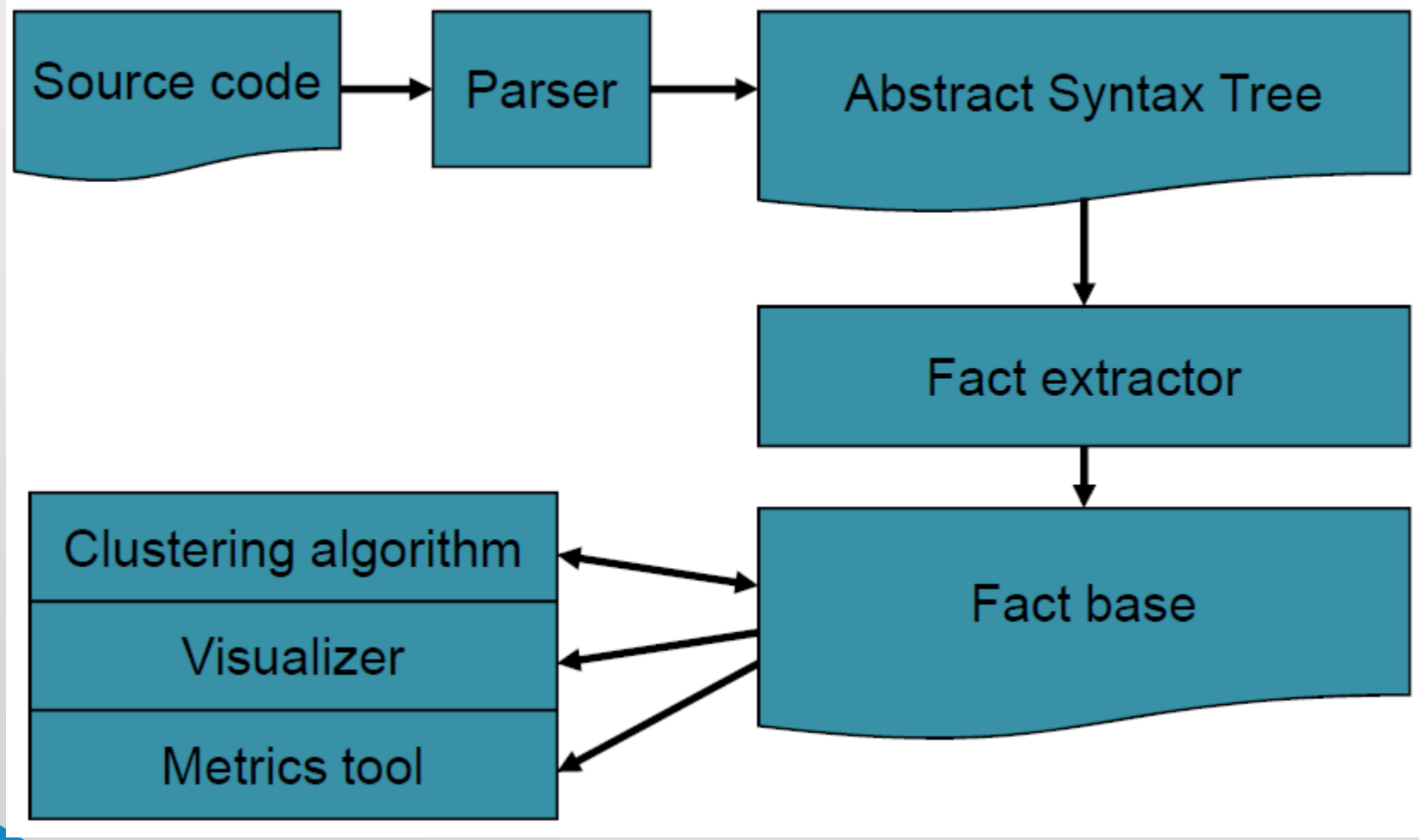
- How many member methods are in the Hello class?
- Answer: Two, the constructor (Hello::Hello()) and destructor (Hello::~Hello()).

- Where are these member methods used?
- Answer: The constructor is called explicitly when an instance of the class is created. The destructor is called implicitly when the execution leaves the scope of the instance.

Static analysis in IDEs

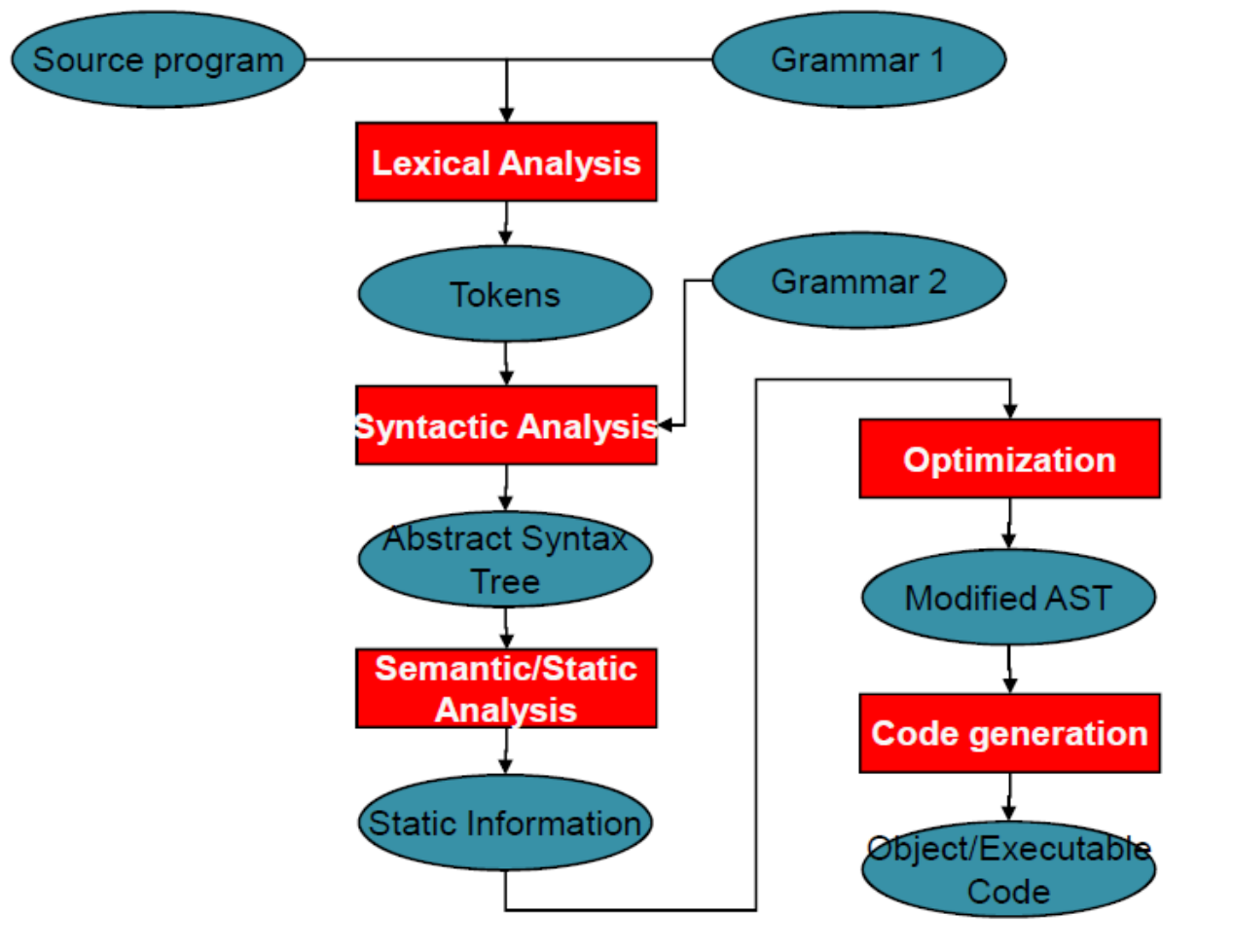
- High-level languages lend themselves better to static analysis needs
  - Rational Rose does the same with UML and Java
- Unfortunately, most legacy systems are not written in either of these languages

Static analysis pipeline



Static Analysis

- Initially emerged in the domain of optimizing compilers:



Examples

- Common expression elimination.

```
int f (int i)                          int f (int i)
{                                      {
int j, k, l;                           int j, k, l;

  j = 4*i;                    ⟹         j = 4*i;
  k = 4*i + 1;                           k = j + 1;
  l = 4*i*k ;                            l = j*k ;
  return l;                              return l;
}                                      }
```

Examples

- Moving loop invariants

```
void g (int n,                          void g (int n,
    int a[], int x)                       int a[], int x)
{                                       {
                                         int i;
                          ⟹             int xx = x*x;
 int i;                                  for (i=1; i<n; i++)
 for (i=1; i<n; i++)                      {a[i]=a[i-1]+xx;}
  {a[i]=a[i-1]+x*x;}                   }
}
```

Examples

- Dead code elimination

```
void h (int n,                          void h (int n,
   int a[], int x)                        int a[], int x)
{                                       {
 int i;                                  int i;
 for (i=0; i<n; i++) ⟹
  {a[i]=a[i]+ x;}                        for (i=0; i<n; i++)
 if (i < n)                              {a[i]=a[i]+x;}
   printf("i < n");
}                                       }
```

Examples

- Non-useful expression elimination.

```
void h (int n,                          void h (int n,
    int a[], int x)                       int a[], int x)
{                                       {
 int i;                                  int i;
 for (i=0; i<n; i++)          ⟹
  {a[i]=a[i]+ x;}                        for (i=0; i<n; i++)
(i < n);                                 {a[i]=a[i]+x;}
}                                       }
```

Examples

- Constant Propagation.

```
void h (int n, int a[])          void h (int n, int a[])
{                                {
 int i;                           int i;
 int x = 4;              ⟹       int x = 4;
 for (i=0; i<n; i++)              for (i=0; i<n; i++)
  {a[i]=a[i]+ x*x+1;}              {a[i]=a[i]+17;}
}                                }
```

Examples

- Variable that are used before their initialization.

```
int f (int i)
{
   int  j;
   int k = i+j;
   j=3;
   return k;

}
```

Examples

- Unauthorized access detection.

```
void main ()
{
int  i, Ligne[100];
FILE  * f;
void send (int j)
...
f = fopen("secret.data","r");
i = 0;
 while ( i < 100 && (fscanf(f, "%d ", &Ligne[i]) != EOF))
{ send(Ligne[i]);
  i = i+1;};
fclose(f);
}
```

Taxonomy

- Different approaches to static analysis:
  - Flow analysis.
  - Type-based analysis.

Flow Analysis

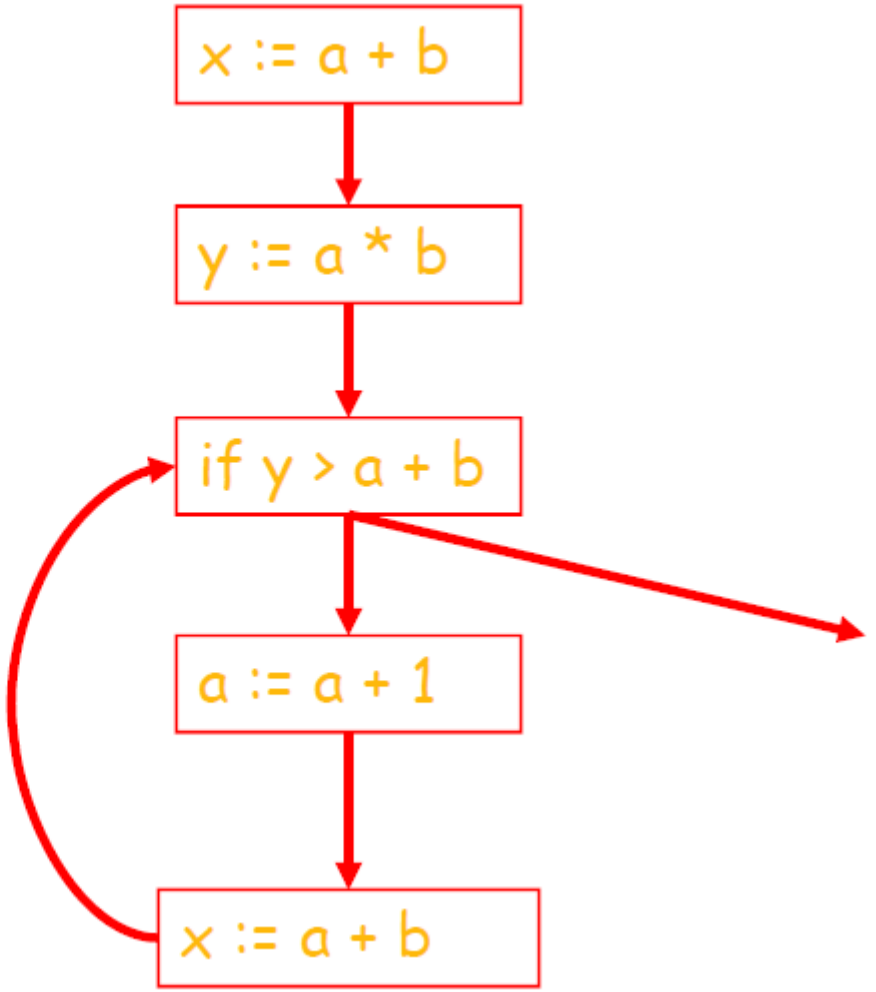
- The purpose of flow analysis is to determine information about functions and data structures that can be called from various program points during execution of the program.

- Generally, flow analysis refers to:
  - Control flow analysis, and,
  - Data flow analysis.

Control-Flow Graphs

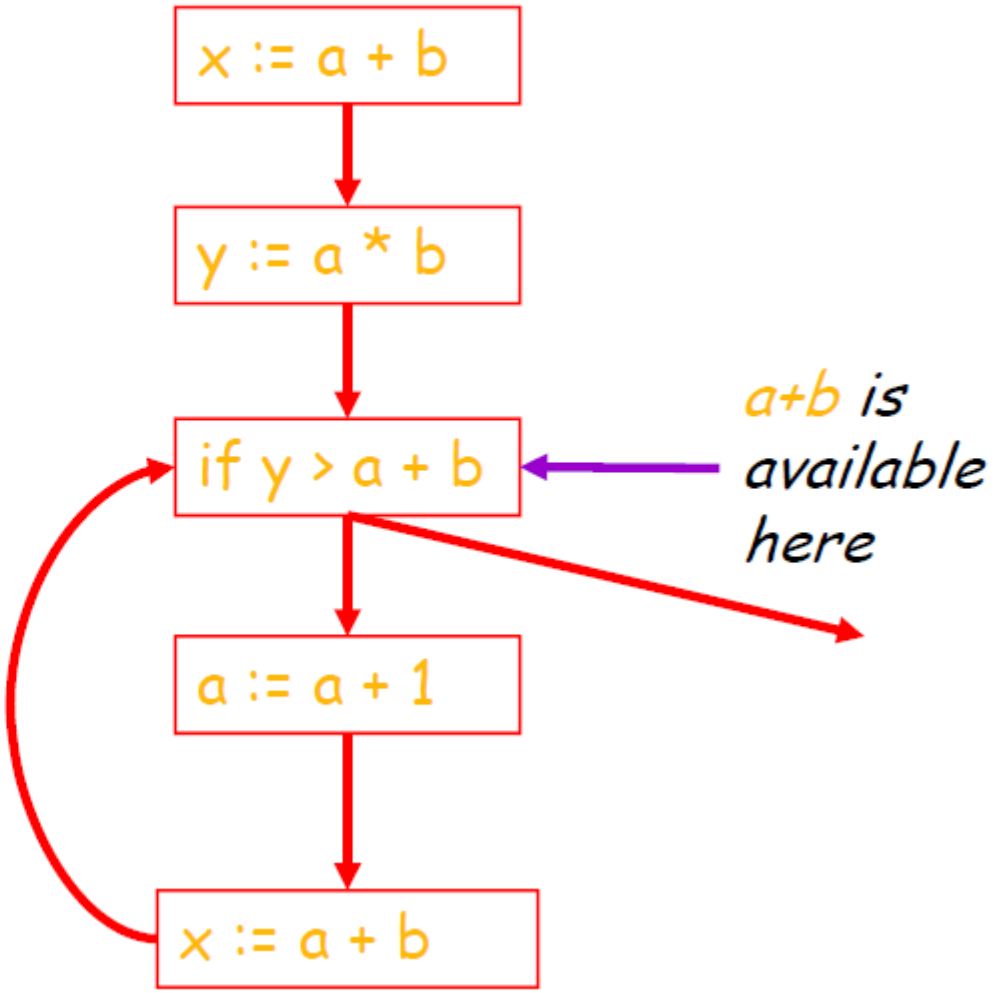
```
x := a + b;
y := a * b;
while y > a + b {
        a := a + 1;
    x := a + b
}
```

Control-flow graphs are state-transition systems.

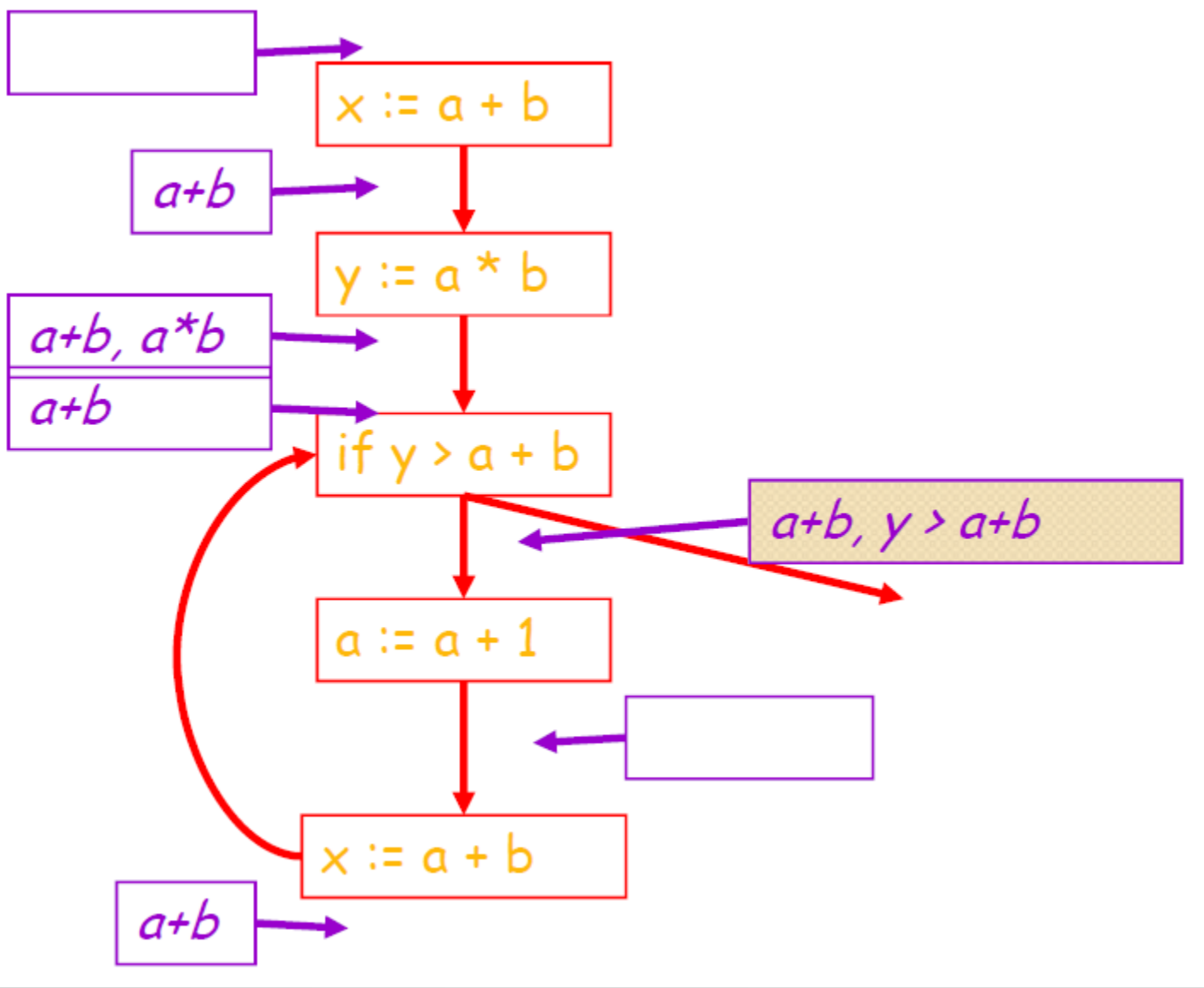


Available Expressions

- For each program point p, which expressions must have already been computed, and not later modified, on all paths to p.
- Optimization: Where available, expressions need not be recomputed.



Example



Type-Based Analysis

- The analysis is defined in terms of a type system.
  - What is a type?
  - What is a type system?
  - How to define a type system?
  - How to elaborate a type-based analysis?

Type and Type System

- What is a type?
  - An abstraction of a concrete domain of values.
  - A partition of the program values into groups of objects with some homogeneity.

- What is a type system?
  - An algebraic structure whose elements are:
    - The types of data that can be manipulated in the language, together with,
    - A mapping that ascribes to programs and their constructs, types.
  - A Programming language definition usually comes with a type system.
  - Well-typed programs do not go wrong.
  - Safe programming.
  - Strongly typed languages are very safe: Java, ML, ADA, CLU, etc.

Type System Definition

- How to define a type system:
  - Informally:
    - List of types.
    - Natural languages to explain the mapping of types to program constructs.
  - Formally:
    - Type algebra (grammar).
    - Typing rules.
    - Typing algorithms.

Dynamic Analysis

- Provides information about the run-time behavior of software systems, e.g.
  - Component interactions
  - Event traces
  - Concurrent behavior
  - Code coverage
  - Memory management
- Can be done with a profiler or a debugger

Dynamic analysis issues

- Ensuring good code coverage is a key concern
- A comprehensive test suite is required to ensure that all paths in the code will be exercised
- Results may not generalize to future executions

Static Analysis

- *Advantages:*
  - No overhead at runtime.
  - A lot of research results (algorithms, methodologies, frameworks, tools, etc.).
  - Analyze all the execution paths.
- *Disadvantages:*
  - Somewhat elaborate to design and implement.
  - Non-decidability issues (e.g. aliasing of pointers).

Dynamic Analysis

- *Advantages:*
  - Information is available at runtime.
  - Easy to implement.
  - Sometimes inevitable.
- *Disadvantages:*
  - Valid only for one execution path.
  - Significant overhead during program execution (time and space).

Representation schemes

- Chosen based on objectives and tasks to be performed. Popular ones are:
  - Abstract Syntax Trees
  - Control Flow Graphs
  - Data Flow Graphs
  - Program Dependency Graphs

Abstract Syntax Trees

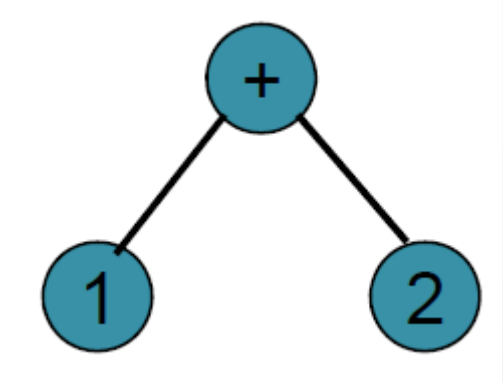
- A translation of the source text in terms of operands and operators
- Omits superficial details, such as comments, whitespace
- All necessary information to generate further abstractions is maintained

AST production

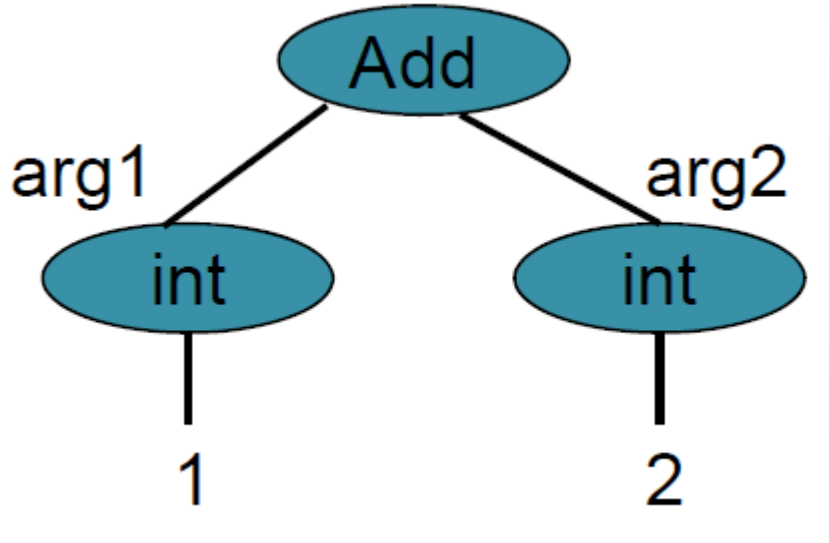
- Four necessary elements to produce an AST:
  - Lexical analyzer (turn input strings into tokens)
  - Grammar (turn tokens into a parse tree)
  - Domain Model (defines the nodes and arcs allowable in the AST)
  - Linker (annotates the AST with global information, e.g. data types, scoping etc.)

AST example

- Input string: **1 + /* two */ 2**
- Parse Tree:



- AST (without global info)



Control Flow Graphs

- Offer a way to eliminate variations in control statements by providing a normalized view of the possible flow of execution of a program

- To produce a CFG:
  - AST of the program
  - Decomposition of the program into basic blocks
  - Basic semantics on the control statements of the language

Data Flow Graphs

- Focus mostly on the exchange of information between program components, i.e. basic blocks, functions, modules

- To produce a DFG:
  - AST of the program
  - Decomposition of the program into basic blocks (or more coarsely-grained level)
  - Annotations on uses and definitions of variables

Program-Dependence Graphs

- The Program-Dependence Graph (PDG) for a program is an intermediate-code form consisting of a Control-Dependence Graph (CDG) and a Data-Dependence Graph (DDG)

Data Flow Analysis

Outline

- Taxonomy of Data-Flow Analysis
- Problems
- Points and Paths
- D-U and U-D Chains
- Reaching Definitions
- Live-Variable Analysis
- Available Expressions

Causes of Redundancy

- The redundancy is available at the source level. However, most often, it is the side effect of having written the program in a high-level language.
  - E.g., access elements of an array through A[i][j], which is equivalent to computation of the location of the (i, j)th element of a matrix A.

Semantics-Preserving Transformations

- Generally, we want to improve a program without changing the function it computes, called as *function preserving* or *semantics-preserving*.

- examples of such semantics-preserving transformations include:
  - Common-subexpression elimination
  - Copy propagation
  - Dead-code elimination
  - Constant folding
  - Code motion
  - Induction variables and strength reduction

Taxonomy of Data-Flow Analysis Problems

- Data-flow analysis problems are categorized along several dimensions:
  - The information they are designed to provide
  - Whether they are relational or involve independent attributes
  - The type of lattices used in them and the meaning attached to the lattice elements and functions defined on them
  - The direction of information flow:
    - In the direction of program execution (forward problem)
    - Opposite the direction of execution (backward problem)
    - In both direction (bidirectional problems)

Types of Data-Flow Analysis

- Reaching Definition
- Available Expressions
- Live Variables
- Upwards Exposed Uses
- Copy-Propagation Analysis
- Constant-Propagation Analysis
- Partial-Redundancy Analysis

Reaching Definition

- *Reach Definition* determines which definition of a variable (i.e., assignments to it) may reach by each use of the variable in a procedure.

Available Expressions

- *Available Expression* determines which expressions are available at each point in a procedure, in the sense that on every path from the entry to the point there is an evaluation of the expression, and none of the variables occurring in the expression are assigned values between the last such evaluation on a path and the point.

Live Variables

- *Live Variables* determines for a given variable and a given point in a program whether there is a use of the variable along some path from the point to the exit.

Upwards Exposed Uses

- *Upwards Exposed Uses* determines what uses of variables at particular points are reached by particular definitions.

Copy-Propagation Analysis

- *Copy-Propagation Analysis* determines that on every path from a copy assignment, say x ← y, to a use of variable x there are no assignments to y.

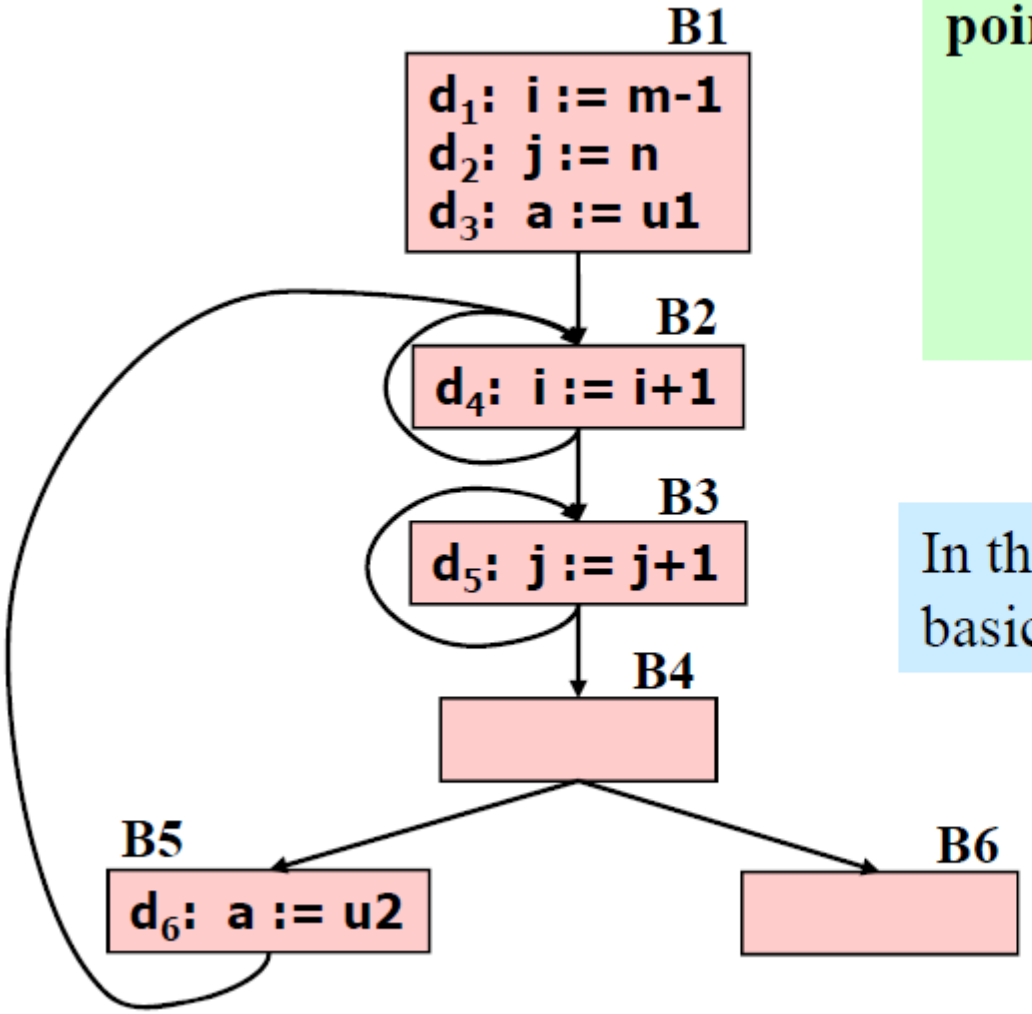
Constant-Propagation Analysis

- *Constant-Propagation Analysis* determines that on every path from an assignment of a constant to a variable, say, x ← const, to a use of x the only assignment to x assign it the value const.

Partial-Redundancy Analysis

- *Partial-Redundancy Analysis* determines what computations are performed twice (or more times) on some execution path without the operands being modified between the computations.

Points and Paths



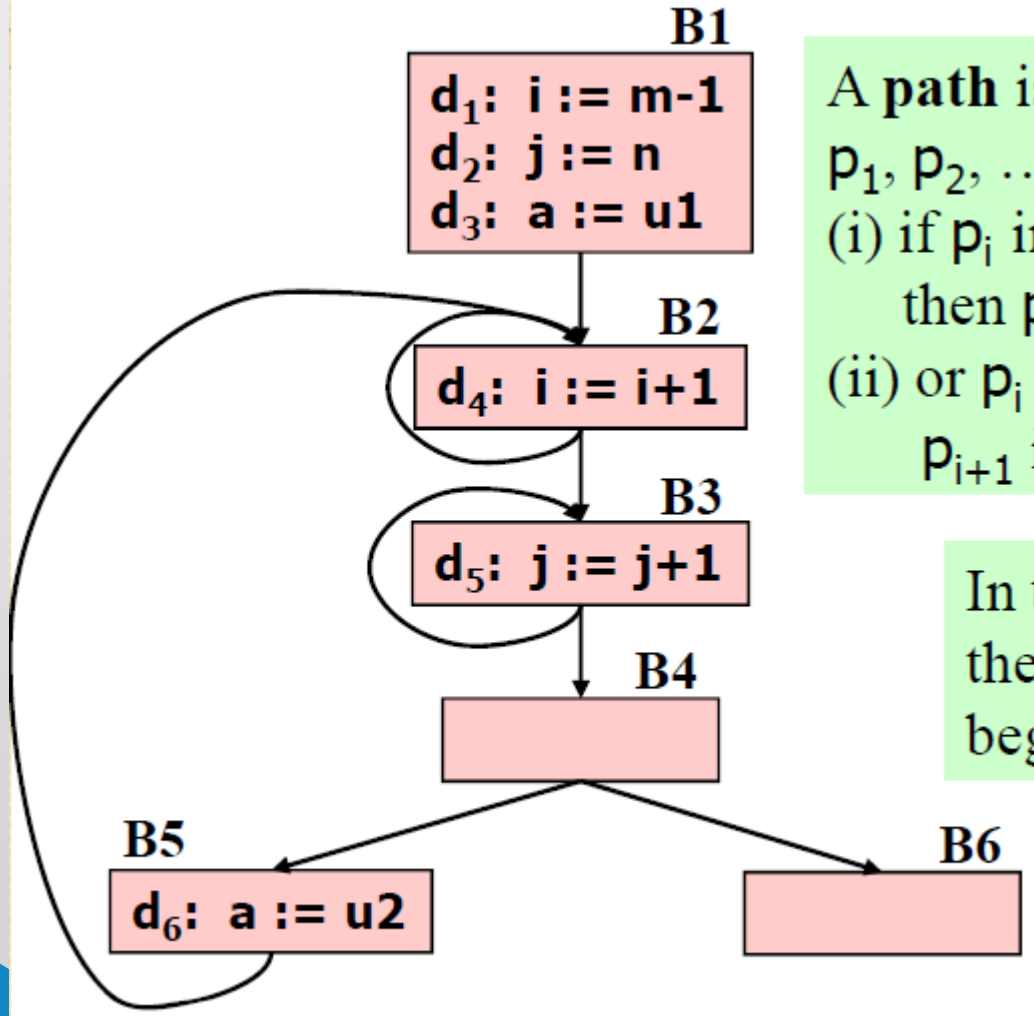
points in a basic block:

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

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

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

Points and Paths



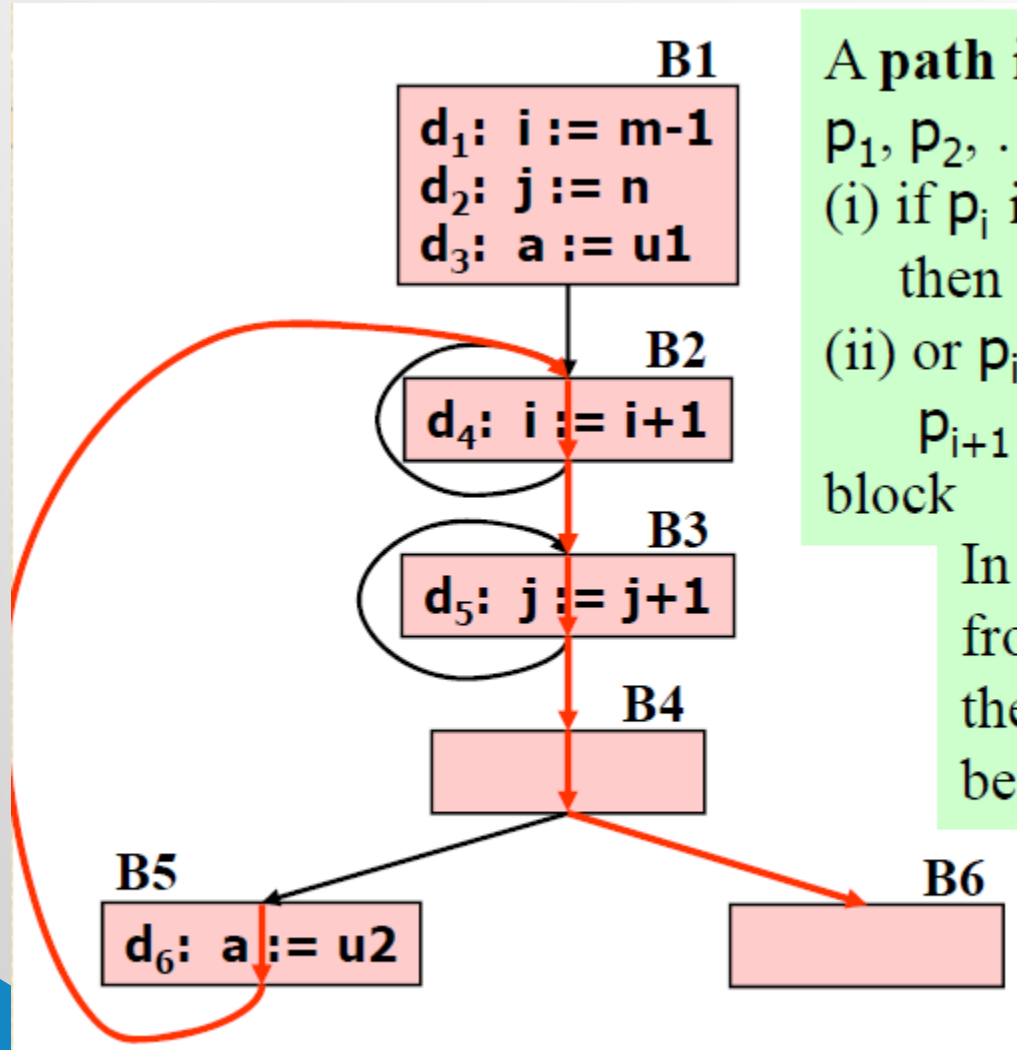
A **path** is a sequence of points $p_1, p_2, \ldots, p_n$ such that either:

(i) if p_i immediately precedes S, then 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?

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,
then 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.

Data-Flow Analysis Schema

- In each application of data-flow analysis, we associate with every point a *data-flow value* that represents an abstraction of the set of all the possible program states that can be observed for that point.

- The set of possible data-flow values is the *domain* for this application.

Data-Flow Problem

- We denote the data-flow values before and after each statement s by IN[s] and OUT[s], respectively.
- The *data-flow problem* is to find a solution to a set of constraints on the IN[s]'s and OUT[s]'s.
- There are two types of constraints:
  - Transfer functions
  - Control-flow constraints

Transfer Functions

- The relationship between the data-flow values before and after an assignment statement is known as a transfer function.
- Transfer functions come in two flavors:
  - Propagate forward the execution path
  - Propagate backward the execution path
- Let denote the transfer function of a statement s as fs. We have:
  - OUT[s] = fs(IN[s]) in a forward-flow problem
  - IN[s] = fs(OUT[s]) in a backward-flow problem

Control-Flow Constraints

- With a basic block, control flow is simple.
- If a block consists of statements s_1, s_2, …, s_n in that order, then the control-flow value out of s_i is the same as the control-flow value into s_i+1.

$$\text{IN}[s_{i+1}] = \text{OUT}[s_i], \text{ for all } i = 1,2,\ldots,n-1$$

- However control flow edges between basic blocks create more complex constraints between the last statement of a basic block and the first statement of the following block.

Data-Flow Schemas on Basic Blocks – Transfer Functions

- We can restate the data-flow values immediately before and immediately after each basic block B by IN[B] and OUT[B], respectively.
- The constraints involving IN[B] and OUT[B] can be derived from those involving IN[s] and OUT[s].
- Suppose block B consists of statement s_1, s_2, …, s_n. Let f_B and f_{si} denote the transfer function of a basic block B and the transfer function of a statement s_i, respectively.

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

$$f_B = f_{s_n} \circ \ldots \circ f_{s_2} \circ f_{s_1}$$

Data-Flow Schemas on Basic Blocks – Control-Flow Constraints

- In a forward-flow problem, we have

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

- In a backward-flow problem, we have

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

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

- *Data-flow equations* usually do not have a unique solution.
- Our goal is to find the most "precise" solution that satisfies the two sets of constraints.

Definition and Use

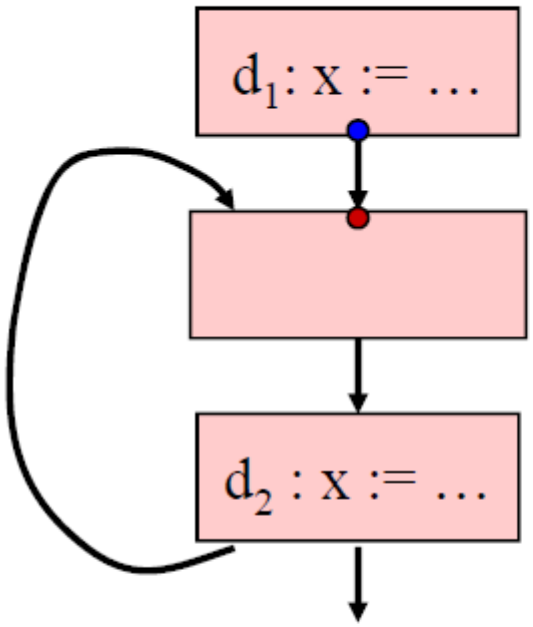
1. **Definition & Use**

S_k: $V_1 = V_2 + V_3$

S_k is a definition of V_1

S_k is an use of V_2 and V_3

Reach and Kill



Kill

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

Reach

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

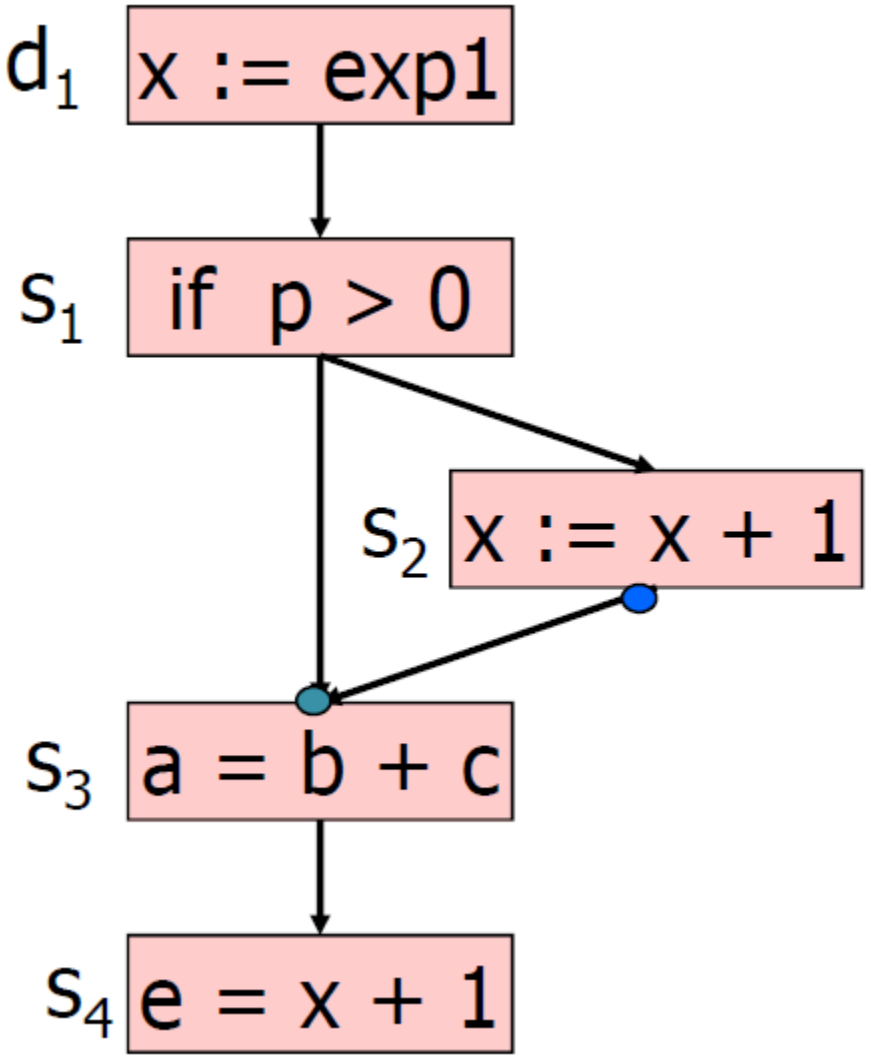
both d_1, d_2 reach point ●
but only d_1 reaches point ●

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

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
```

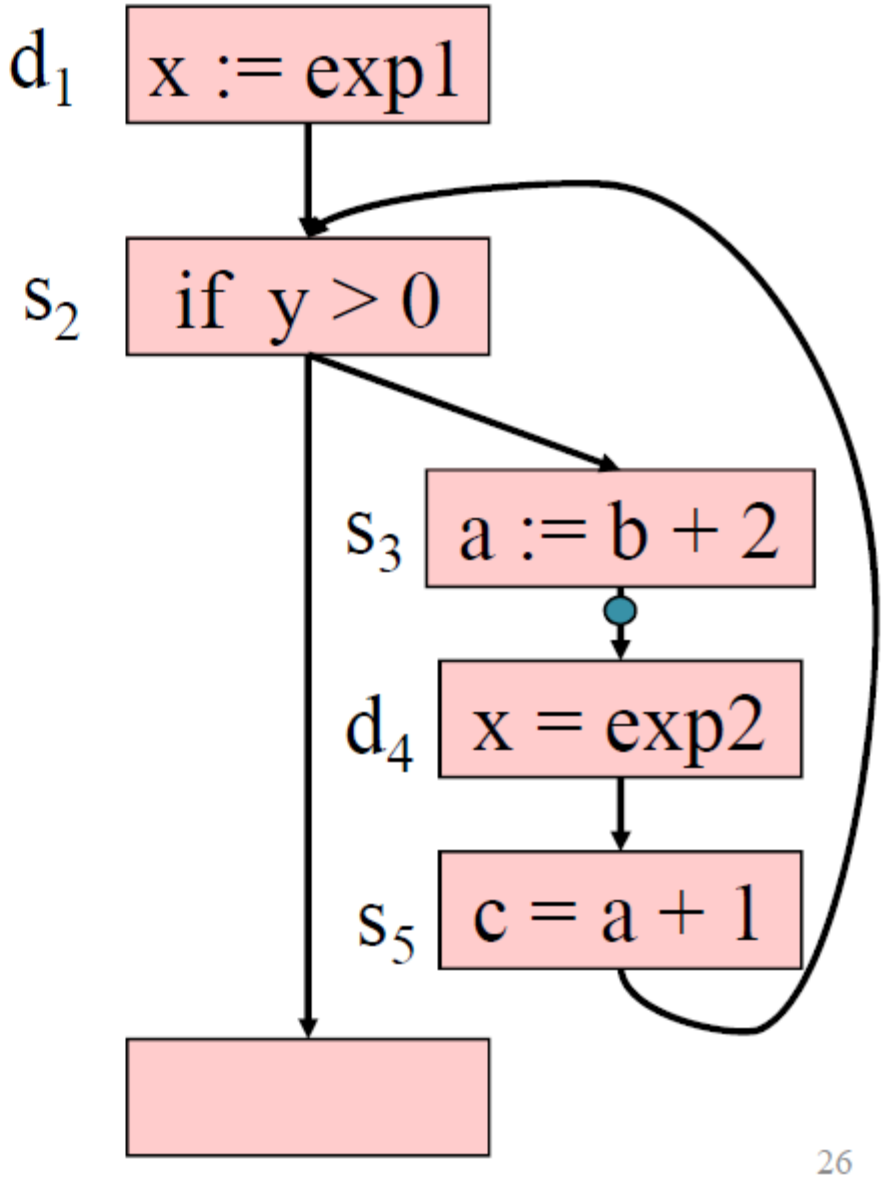


It depends on what point p_1 represents!!!

Problem Formulation: Example 2

Can d_1 and d_4 reach point p_3?

```
d1    x := exp1
s2    while y > 0 do
s3      a := b + 2      <-- p3
d4      x := exp2
s5      c := a + 1
      end while
```



26

D-U and U-D Chains (Motivation)

- Many dataflow analyses need to find the use sites

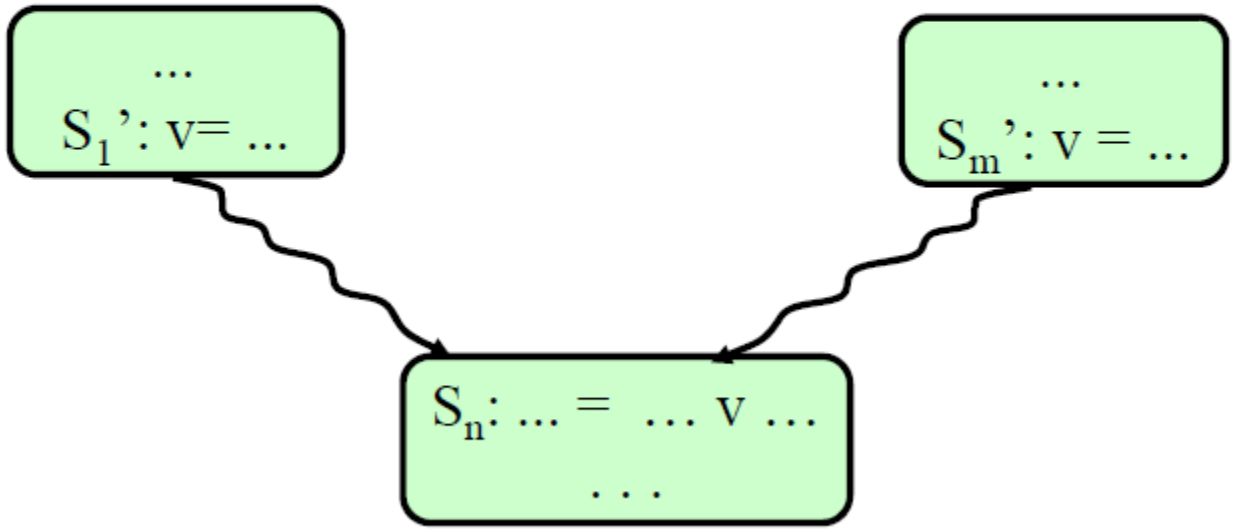
of each defined variable or the definition sites of each variable used in an expression.

- 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 there is no need to maintain D-U and U-D chains.

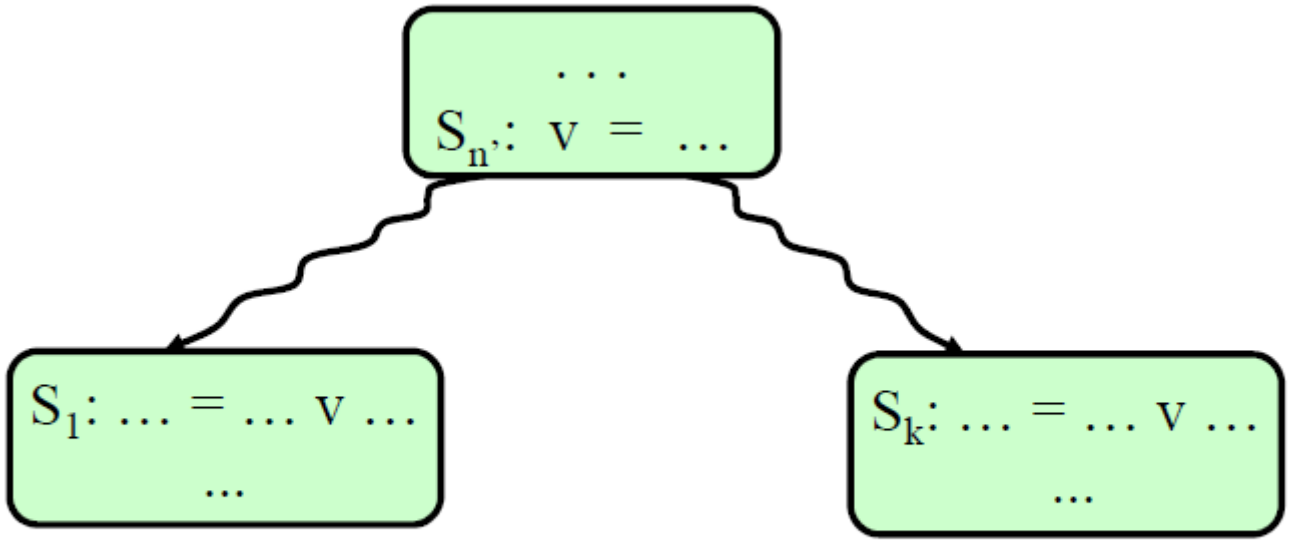
UD Chain

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', …, S_m').

DU Chain



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

Reaching Definitions

- Problem Statement:
  - Determine the set of definitions reaching a point in a program.
- To solve this problem we must take into consideration the data-flow and the control flow in the program.
- A common method to solve such a problem is to create a set of *data-flow equations*.

Transfer Equations for Reaching Definitions

- Consider a definition *d: u = v+w*, the transfer function of d can be expressed as:

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

- This rule can be extends to a block. Suppose block B has n statements with transfer functions $f_i(x) = gen_i \cup (x - kill_i)$. Then the transfer function for block B can be written as:

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

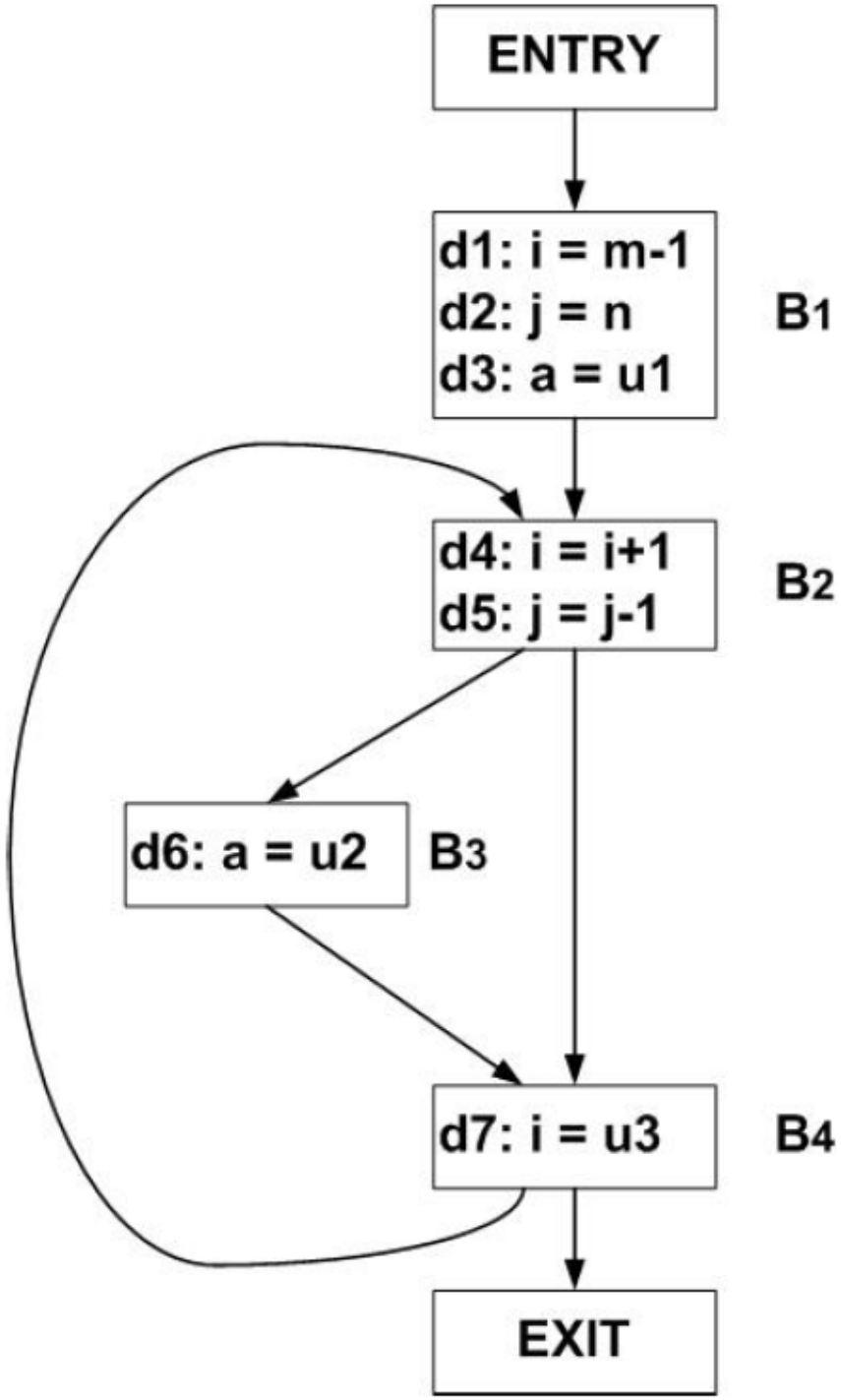
where

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

$$gen_B = gen_n \cup (gen_{n-1} - kill_n) \cup (gen_{n-2} - kill_{n-1} - kill_n) \cup$$
$$\cdots \cup (gen_1 - kill_1 - kill_2 - \cdots - kill_n)$$

gen and *kill* Sets of a Block

- The *gen* set contains all the definitions inside the block that are "visible" immediately after the block
  - Also referred as *downwards exposed*.
- The *kill* set is the union of all the definitions killed by the individual statements.
- In *gen-kill* form, the *kill* set is applied before the *gen* set.



$$gen_{B_1} = \{d_1, d_2, d_3\}$$
$$kill_{B_1} = \{d_4, d_5, d_6, d_7\}$$

$$gen_{B_2} = \{d_4, d_5\}$$
$$kill_{B_2} = \{d_1, d_2, d_7\}$$

$$gen_{B_3} = \{d_6\}$$
$$kill_{B_3} = \{d_3\}$$

$$gen_{B_4} = \{d_7\}$$
$$kill_{B_4} = \{d_1, d_4\}$$

Iterative Algorithm for Reaching Definitions

- The reaching definitions problem is defined by the following equations:

$$\mathrm{OUT}[\mathrm{ENTRY}] = \phi$$

and for all basic blocks B other than ENTRY

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

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

Iterative Algorithm for Reaching Definitions (cont.)

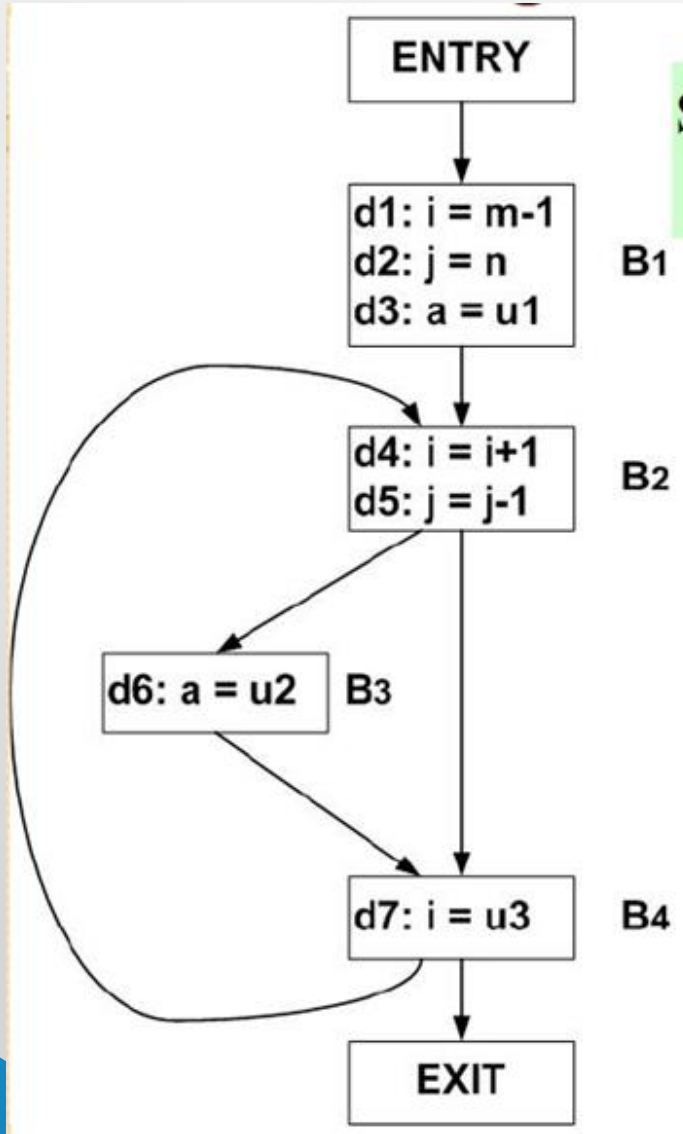
OUT[ENTRY] = ø;

for (each basic block B other than ENTRY)
  OUT[B] = ø;

while (changes to any OUT occur)
  for (each basic block B other than ENTRY) {
    $\mathrm{OUT}[B] = gen_B \cup (\mathrm{IN}[B] - kill_B)$;
    $\mathrm{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \mathrm{OUT}[P]$;
  }

Will the while loop be executed infinitely in some case?

Iterative Algorithm for Reaching Definitions (cont.)



Step 1: Compute gen and kill for each basic block

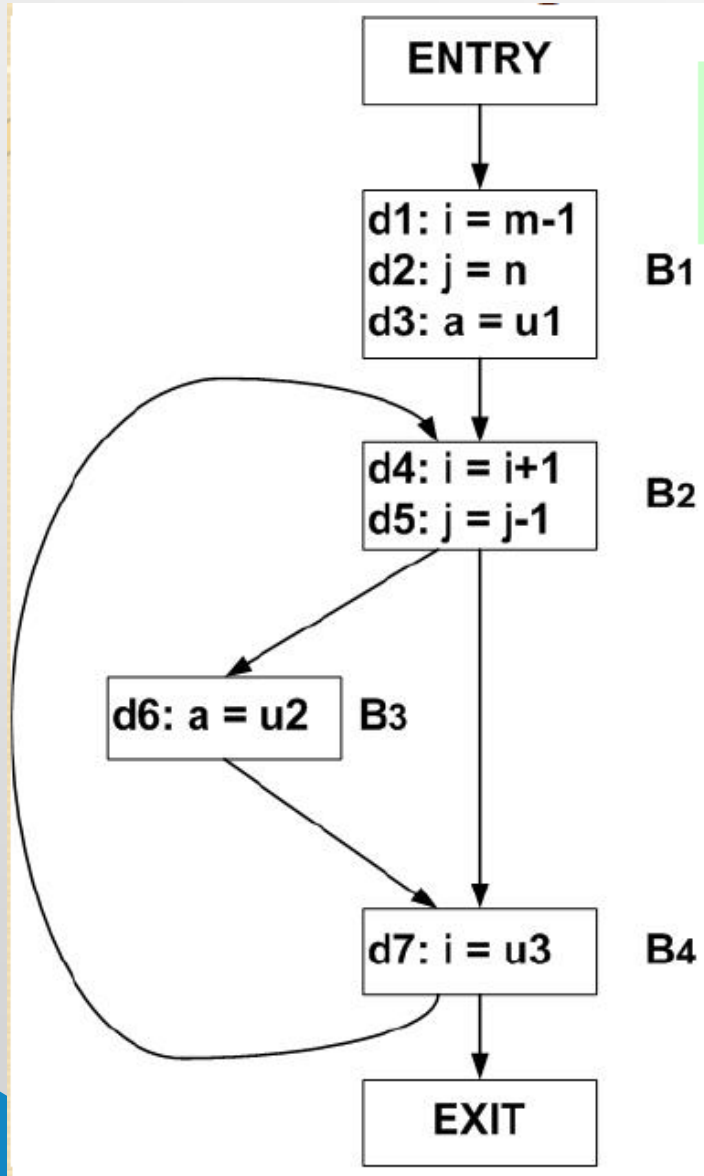
$gen[B1] = \{d_1, d_2, d_3\}$
$kill[B1] = \{d_4, d_5, d_6, d_7\}$

$gen[B2] = \{d_4, d_5\}$
$kill\ [B2] = \{d_1, d_2, d_7\}$

$gen[B3] = \{d_6\}$
$kill\ [B3] = \{d_3\}$

$gen[B4] = \{d_7\}$
$kill\ [B4] = \{d_1, d_4\}$

Iterative Algorithm for Reaching Definitions (cont.)



Step 2: For every basic block, make:

out[B] = ∅

Initialization:

out[ENTRY] = ∅

out[B1] = ∅

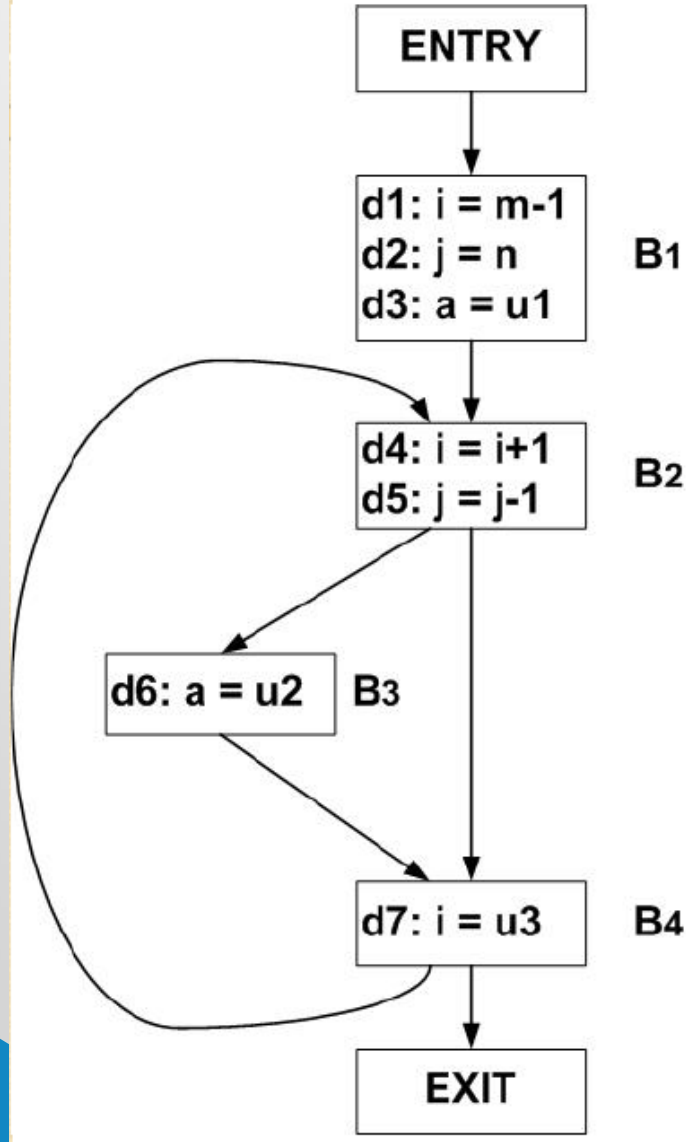
out[B2] = ∅

out[B3] = ∅

out[B4] = ∅

out[EXIT] = ∅

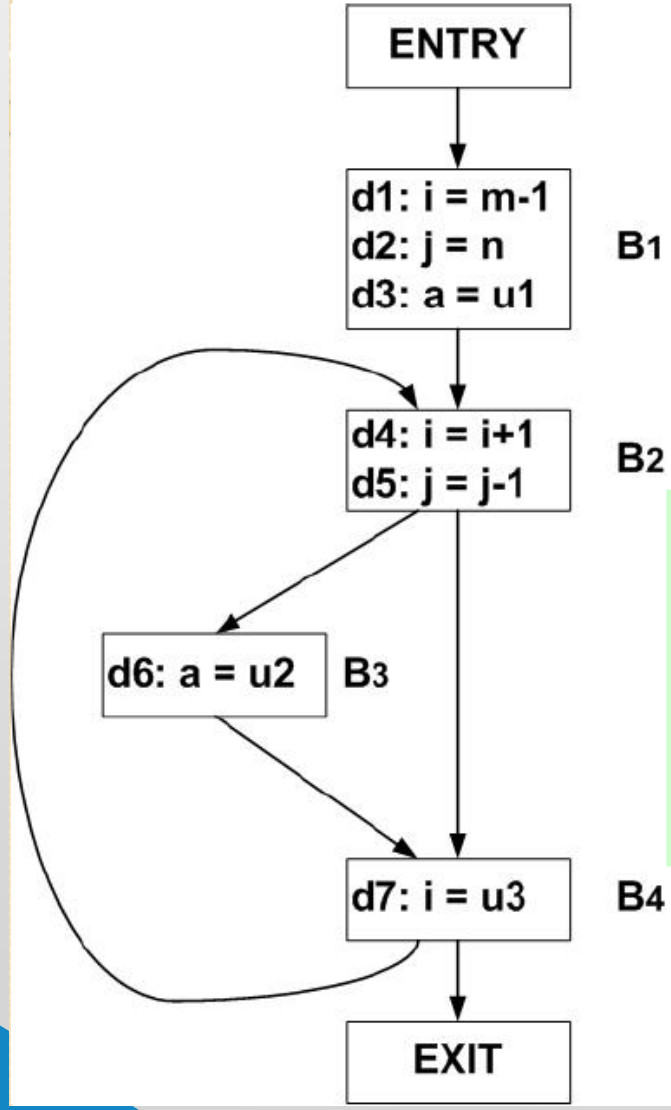
Iterative Algorithm for Reaching Definitions (cont.)



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:

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

Iterative Algorithm for Reaching Definitions (cont.)



while a fixed point is not found:

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

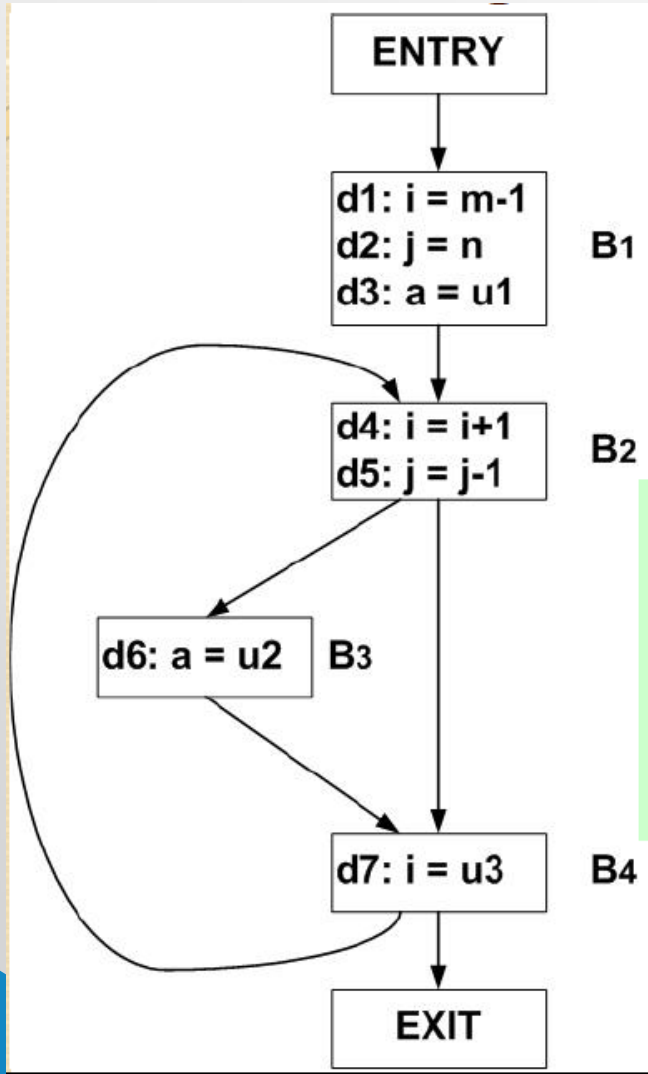
out[B] = gen[B] ∪ (in[B]-kill[B])

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] = $\{d_1, d_4\}$

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

Block	First Iteration	
	in[B]	out[B]
B_1		
B_2		
B_3		
B_4		
EXIT		

Iterative Algorithm for Reaching Definitions (cont.)



while a fixed point is not found:

$in[B] = \cup\ out[P]$ **where P is a predecessor of B**

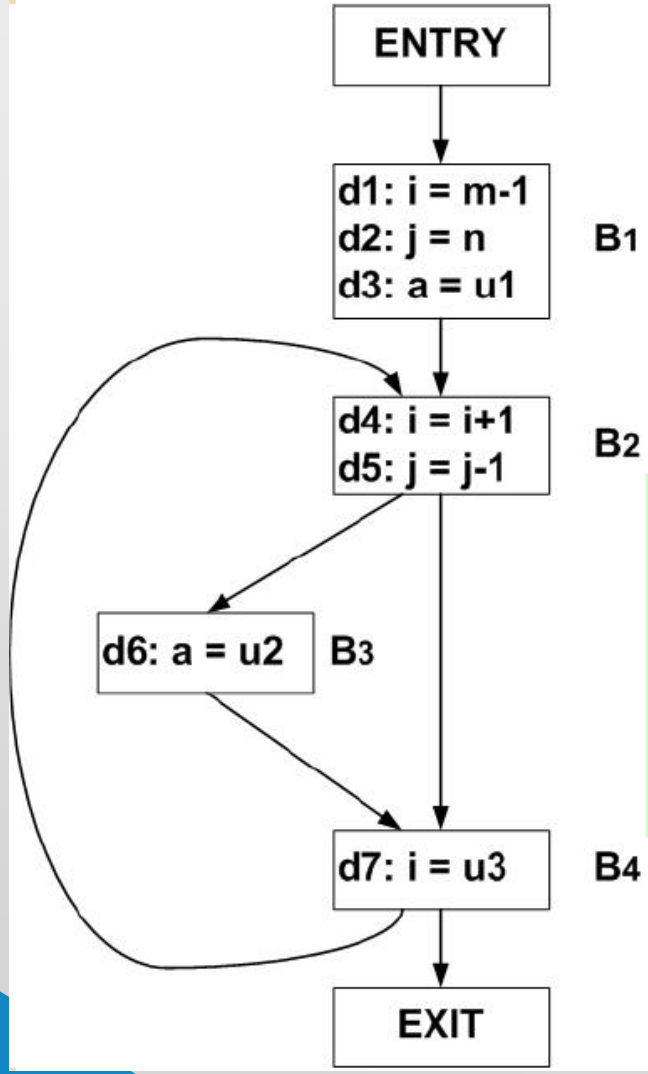
$out[B] = gen[B] \cup (in[B]-kill[B])$

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

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

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

Iterative Algorithm for Reaching Definitions (cont.)



while a fixed point is not found:

$in[B] = \cup\ out[P]$ **where P is a predecessor of B**

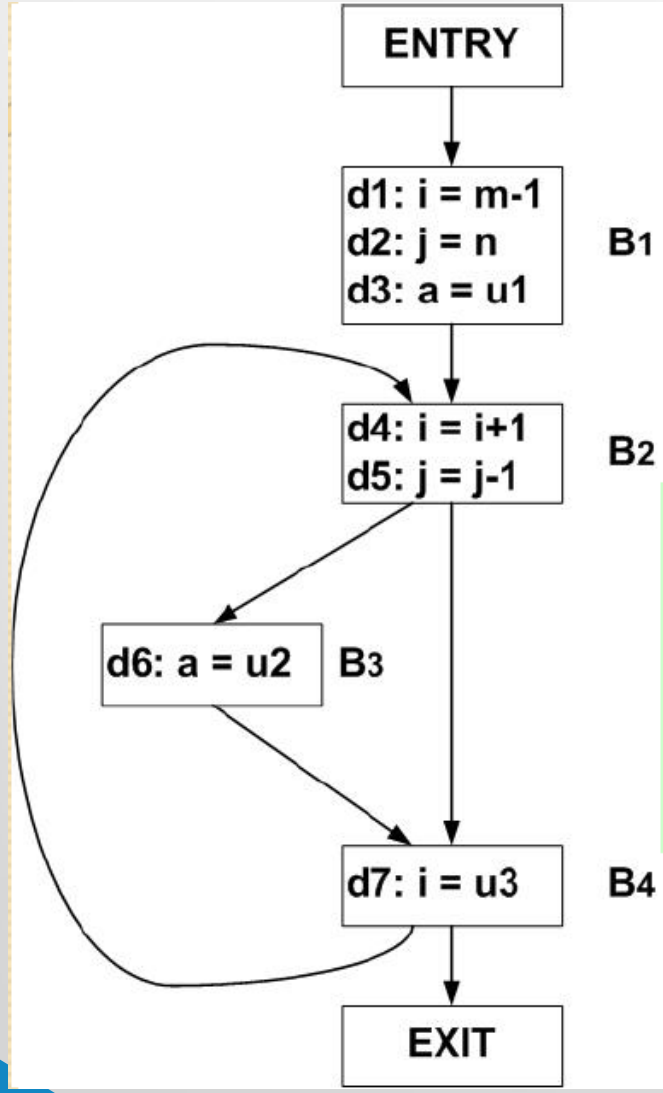
$out[B] = gen[B] \cup (in[B]-kill[B])$

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

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

Block	Second Iteration	
	in[B]	out[B]
B_1		
B_2		
B_3		
B_4		
EXIT		

Iterative Algorithm for Reaching Definitions (cont.)



while a fixed point is not found:

$in[B] = \cup\ out[P]$ **where P is a predecessor of B**

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

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

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

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

Algorithm Convergence

- Intuitively we can observe that the algorithm converges to a fix point because the 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 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 than five.

Live-Variable Analysis

- In live-variable analysis, we wish to know for variable x and point p whether the value of x at p could be used along some path in the flow graph starting at p.
  - If so, we say x is live at p
  - Otherwise, x is dead at p
- An important use for live-variable information is register allocation for basic blocks.

Live-Variable Analysis (cont.)

- The live-variable problem is defined by the following equations:

$$\mathrm{IN}[\mathrm{EXIT}] = \phi$$

and for all basic blocks B other than EXIT

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

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

Live-Variable Analysis (cont.)

IN[EXIT] = ø;

for (each basic block B other than EXIT)

  IN[B] = ø;

while (changes to any IN occur)

  for (each basic block B other than EXIT) {

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

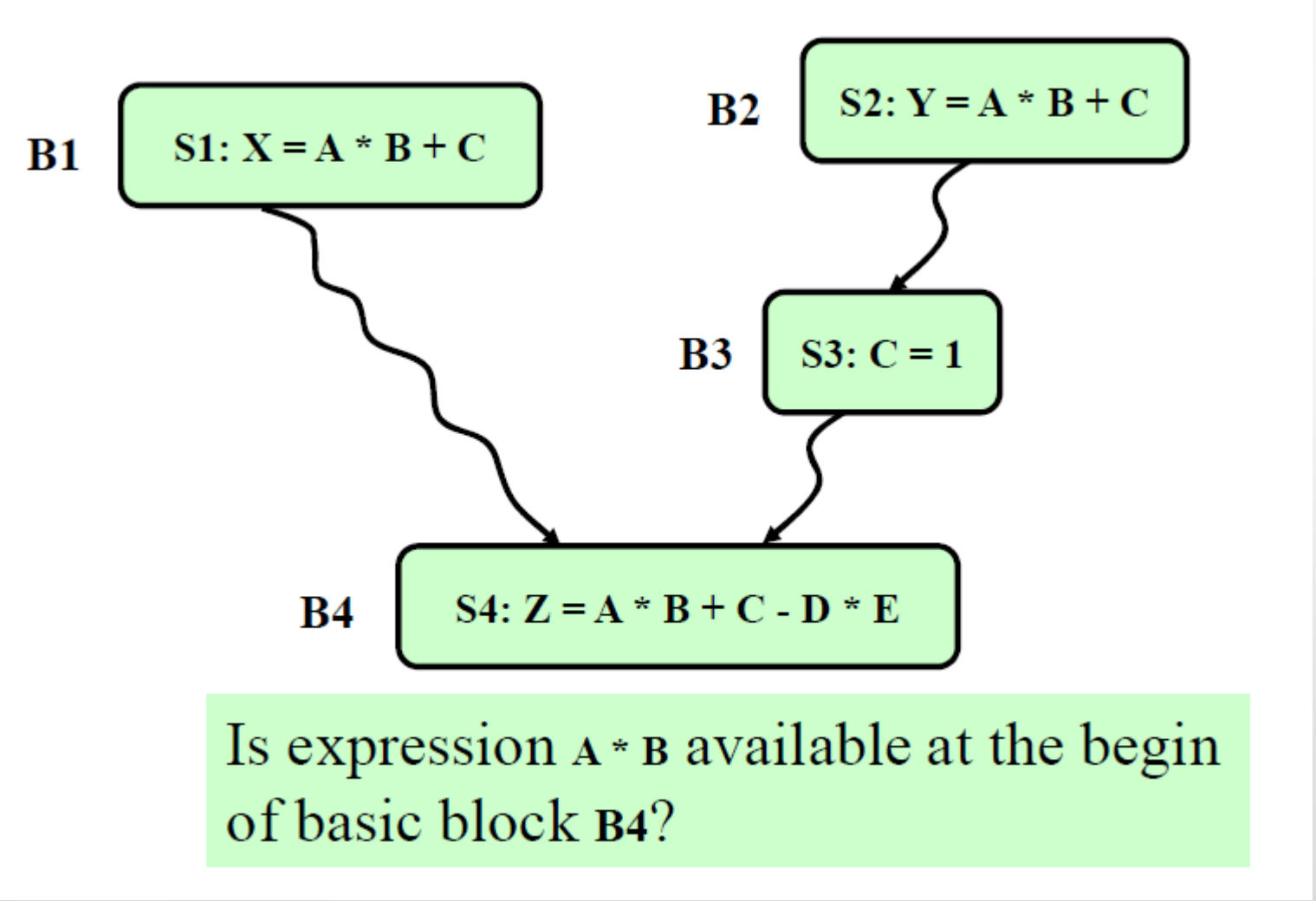
    $\mathrm{IN}[B] = use_B \cup (\mathrm{OUT}[B] - def_B)$;

  }

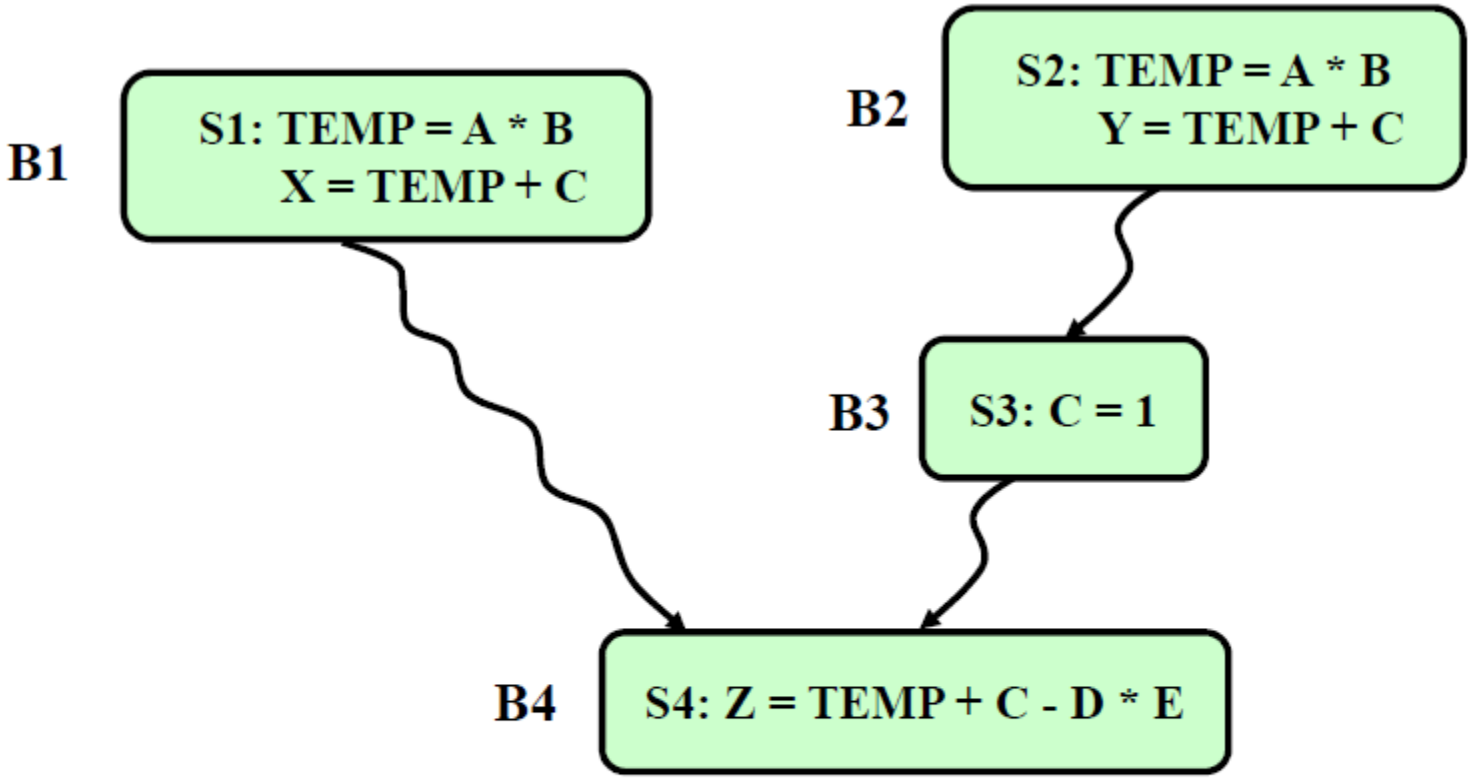
Available Expressions

- 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 to y.
- A basic block *kills* expression x+y if it *may* assign x or y, and does not subsequently recompute x+y.
- A basic block *generates* expression x+y if it *definitely* evaluates x+y, and does not subsequently define x or y.
- The primary use of available-expression information is for detecting common subexpressions.

Available Expression: Example 3



Redundant Expressions: Example 3



Yes, because 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 (cont.)

- The available expression problem is defined by the following equations:

$$\mathrm{OUT}[\mathrm{ENTRY}] = \phi$$

- and for all basic blocks B other than ENTRY

$$\mathrm{OUT}[B] = e\_gen_B \cup (\mathrm{IN}[B] - e\_kill_B)$$

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

- where *e_genB* and *e_killB* are the expressions generated by B and the expressions killed in B, respectively.

Available Expressions (cont.)

OUT[ENTRY] = ø;

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) {

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

$$\text{OUT}[B] = e\_gen_B \cup (\text{IN}[B] - e\_kill_B);$$

  }

References

- Principles of program analysis, by Hanne Riis Nielson, Flemming Nielson, and Chris Hankin, January 1999.