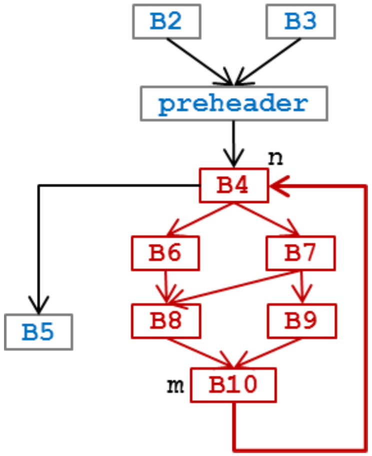
Control and Data Flow Analysis



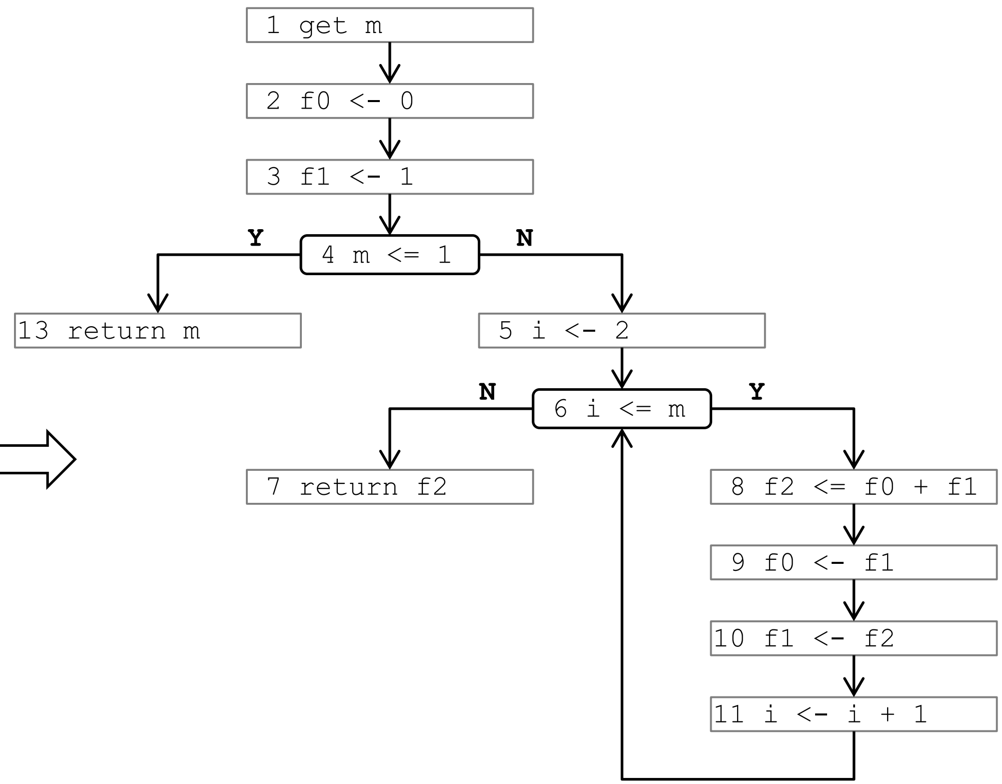
Control-Flow Analysis

Control-Flow Analysis

- prerequisite for data-flow analysis
- simply put a flowchart illustrating the different paths (“control flows”) a program may take

```
int fib(int m) {
  int i, f0, f1, f2;
  f0 = 0; f1 = 1;
  if (m <= 1) return m;
  else {
    for (i=2; i<=m; i++) {
      f2 = f0 + f1;
      f0 = f1;
      f1 = f2;
    }
    return f2;
  }
}
```

```
 1      get m
 2      f0 <- 0
 3      f1 <- 1
 4      if m <= 1 goto L3
 5      i <- 2
 6 L1:  if i <= m goto L2
 7      return f2
 8 L2:  f2 <- f0 + f1
 9      f0 <- f1
10      f1 <- f2
11      i <- i + 1
12      goto L1
13 L3:  return m
```



Control-Flow Analysis

- three main approaches for control-flow analysis
  - dominators + loop detection
  - interval analysis
  - structural analysis

- elimination methods have advantages over dominators and iterative data-flow analysis
  - faster
  - easier to update
  - better structure to do low-level control-flow optimizations

- current compilers use dominators + iterative data-flow analysis
  - easier to implement
  - provide sufficient information

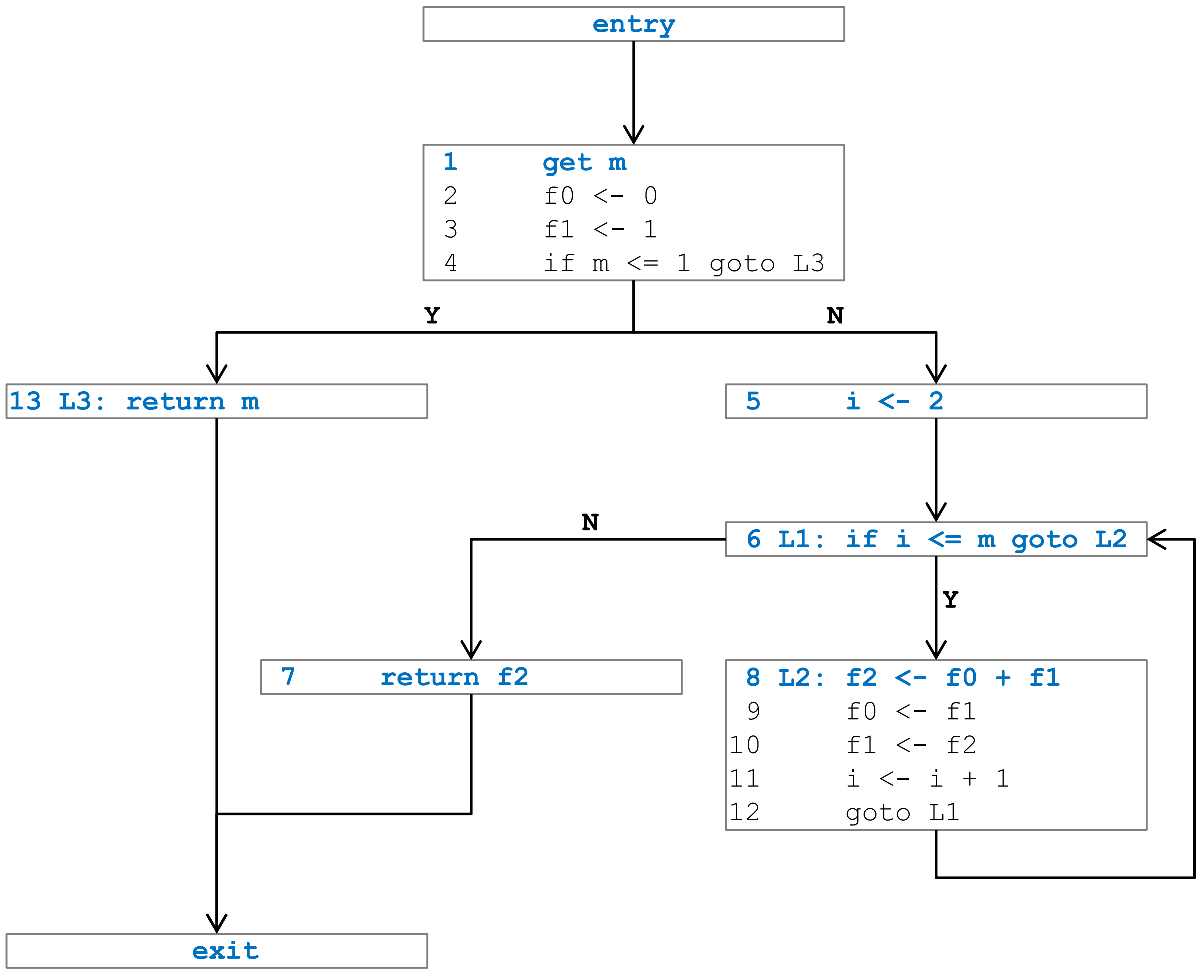
Basic Blocks and Flow Graphs

- **Def. basic block**
  a maximal sequence of instructions that can only be entered at the first and exited at the last instruction
  - potential leaders:
    - entry point
    - branch target
    - instruction following a control-flow instruction
      - what about subroutine calls?
  - construction is straight forward for straight-line three-address intermediate code
    - identify all leaders
    - include all instructions until next leader
    - add extra `entry` and `exit` blocks

Basic Blocks and Flow Graphs

- **Example**

```
 1     get m
 2     f0 <- 0
 3     f1 <- 1
 4     if m <= 1 goto L3
 5     i <- 2
 6 L1: if i <= m goto L2
 7     return f2
 8 L2: f2 <- f0 + f1
 9     f0 <- f1
10     f1 <- f2
11     i <- i + 1
12     goto L1
13 L3: return m
```



Basic Blocks and Flow Graphs

- **Def. flow graph**
  G = <N, E>
  N: basic blocks (incl. entry & exit)
  E: control-flow edges

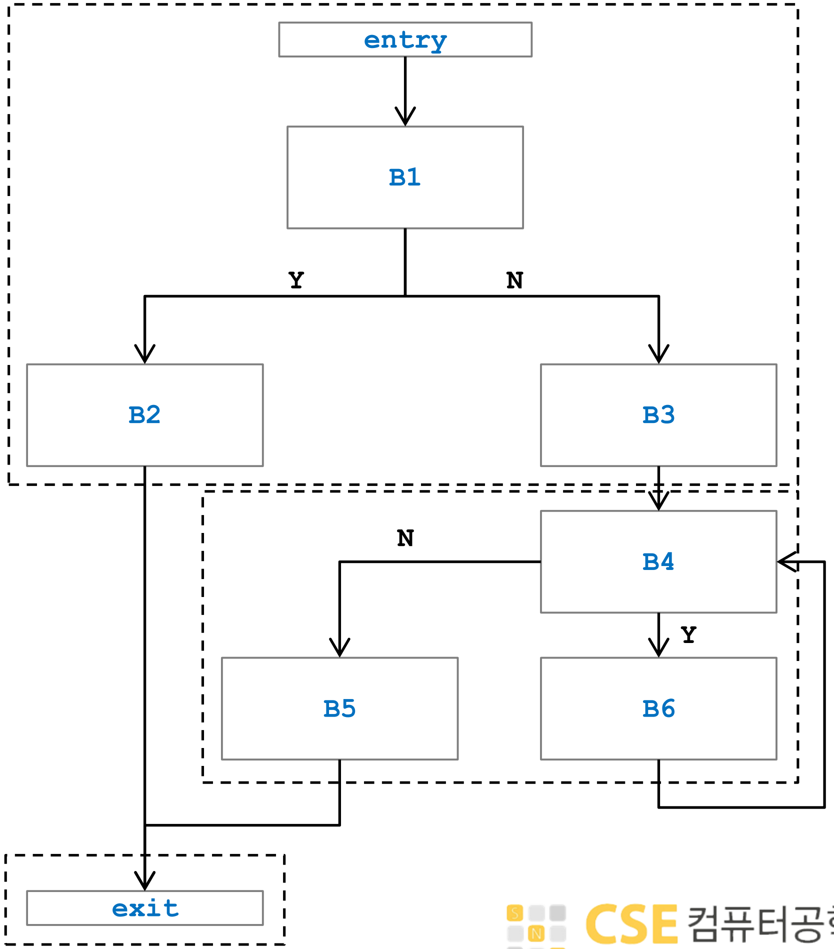
- **Def. (immediate) successor/predecessor** sets of a basic block b

  $\text{Succ}(b) = \{ n \in N \mid \exists e \in E \text{ such that } e = b \rightarrow n \}$
  $\text{Pred}(b) = \{ n \in N \mid \exists e \in E \text{ such that } e = n \rightarrow b \}$

  - a branch node has more than one successor
  - a join node has more than one predecessor

Basic Blocks and Flow Graphs

- **Def. extended basic block**
  a maximal sequence of instructions that beginning with a leader that contains no join nodes other than its leader
  - multiple exits
  - subtree of flow graph



Dominators and Postdominators

- **Def. dominator**
  A node d *dominates* node i, d *dom* i, iff every possible execution path from `entry` to i includes d.
  - reflexive, transitive, antisymmetric

- **Def. strict dominator**
  A node d *strictly dominates* node i, d *sdom* i, if d *dom* i and $d \neq i$.

- **Def. immediate dominator**
  A node d *immediately dominates* node i, d *idom* i, iff a *sdom* b, and there is no node c for which a *sdom* c and c *sdom* b.

- **Def. postdominator**
  A node p *postdominates* node i, p *pdom* i, if every possible execution path from i to `exit` includes p.

Computing Dominator Relations

- Plenty of code out there, one example:
  - idea: $a\ dom\ b$ iff
    - $a = b$
    - a is the unique immediate predecessor of b
    - b has more than one immediate predecessor and for all immediate predecessors c of b, $c \neq a$ and $a\ dom\ c$.
  - more efficient algorithms exist (Lengauer & Tarjan) but are much more complicated (see Muchnick)

```
procedure dom_rel(N, Pred, root)
begin
  dom(root) := { root }
  for each n in N-{root} do
    dom(n) := N
  od
  repeat
    change := false
    for each n in N-{root} do
      T := N
      for each p in Pred(n) do
        T := T ∩ dom(p)
      od
      D := {n} ∪ T
      if D ≠ dom(n) then
        change := true
        dom(n) := D
      fi
    od
  until !change
  return dom
end
```

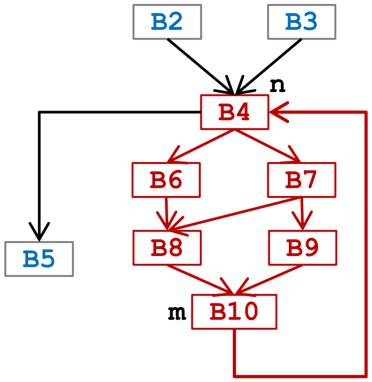
- idom is easy as well (see e.g. Cytron's SSA paper)

Natural Loops

- **Def. backedge**
  A back edge is an edge in a flowgraph whose head dominates its tail, i.e., $tail \rightarrow head$ and $head\ dom\ tail$.

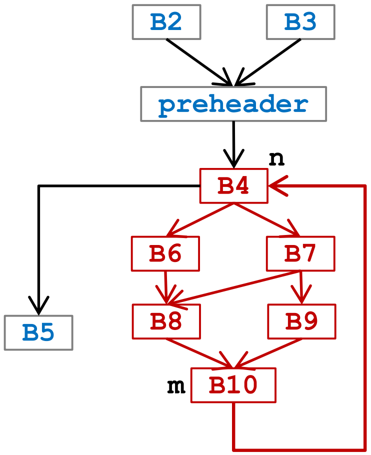
- **Def. natural loop**
  The natural loop of $m \rightarrow n$ is the subgraph consisting of n and all the nodes from which m can be reached without passing through n plus the connecting edges.

  n is called the **loop header**.

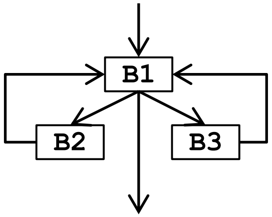


Natural Loops

- Often, it is beneficial to insert a **preheader** just before the loop header.
  - (initially) empty
  - single edge from preheader to header
  - all incoming edges to the loop header are moved to the preheader



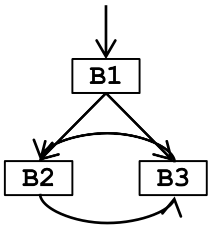
- Properties of natural loops
  - different loop header: disjoint or nested
  - same header: not clear



Strongly Connected Components

- A natural loop is well-structured.

- Languages with unstructured control flow ("goto"s) can have arbitrary looping structures with more than one entry point



- General loop structure: strongly connected components (SCC)
- **Def. strongly connected component**
  A subgraph $G_s = <N_s, E_s>$ of G such that every node in N_s is reachable from every other node by a path that includes only edges in E_s.

- Computation of SCCs: Tarjan's algorithm

Well-Structured Flow Graphs

- **Def. well-structured flowgraphs (aka reducible flowgraphs)**
  A flowgraph $G=<N, E>$ is well-structured iff E can be partitioned into a forward set, E_f, and a disjoint backward set, E_b, such that $G=<N, E_f>$ forms a DAG in which each node can be reached from the entry node and the node is E_b are all back-edges.

- In reducible flowgraphs, all loops are entered through their headers.

- Most modern programming languages (Java, C#, Modula-2 and descendants) only generate reducible flowgraphs.

Data-Flow Analysis

Data-Flow Analysis

- provide global information about how a block, procedure, or larger segment of a program manipulates its data

- example

```
...                ...                ...
i = 16;     ⇨      i = 16;     ⇨      i = 16;
...                ...                ...
a = b / i;         a = b / 16;        a = b >> 4;
```

but what if

```
...
if (cond) i = 16;
...
a = b / i;
```

Data-Flow Analysis

- data-flow approximation
  often, we cannot determine exactly how the data is manipulated

- conservative approximations
  if we cannot determine the exact manipulation, then at least a conservative approximation of it

- aggressive data-flow analysis
  as aggressive as possible while still being conservative

- we will look into iterative data-flow analysis based on the dominator control-flow analysis

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Example: Reaching Definitions

- **Def. reaching definition**
  a definition of a variable d is said to *reach* a given point p in a procedure if there is an execution path from the d to p such that the variable use at p may have the value assigned at d.

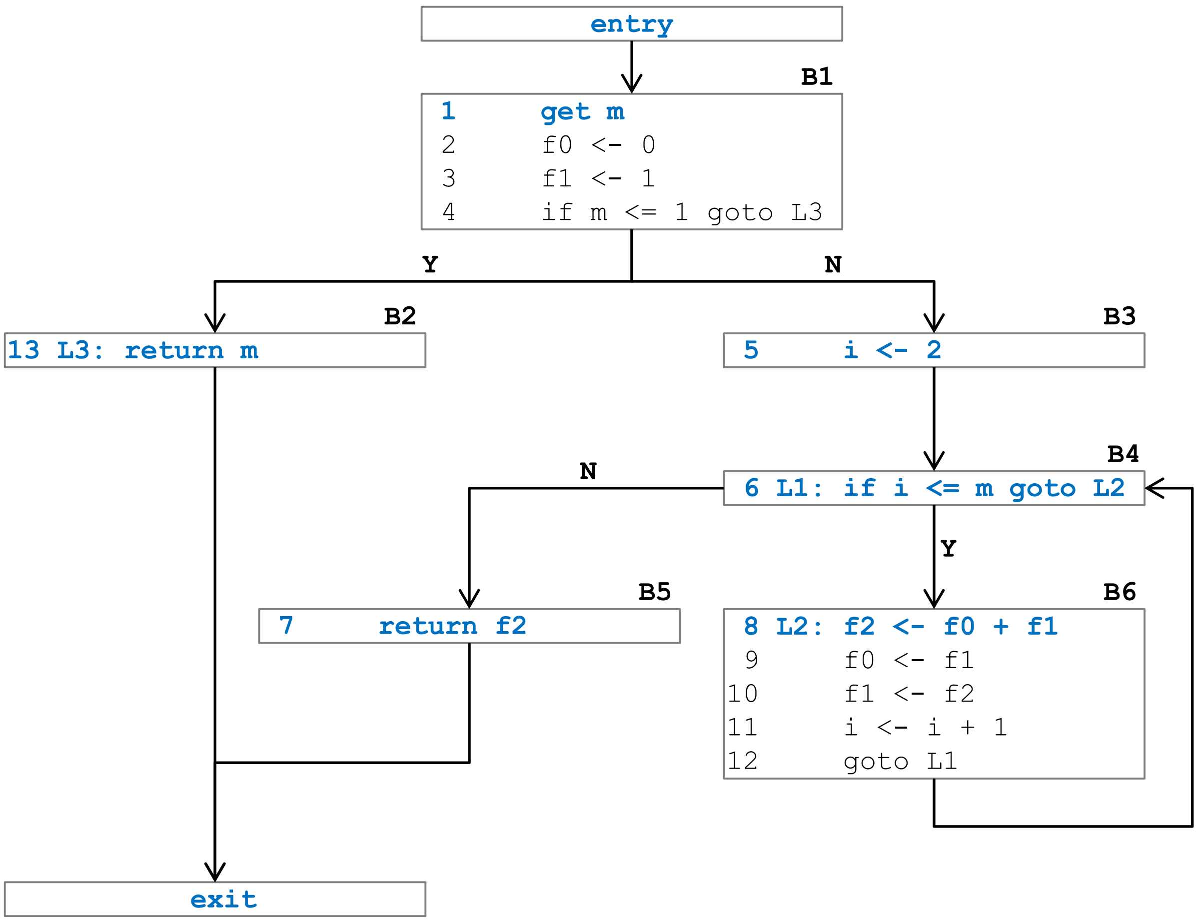
- Obvious pre-requisite: control-flow graph

- Typical approach:
  1. local flow analysis
  2. global flow analysis

- Bit-vector or set representation
  - model the problem as bit-vectors with operations defined on them
  - typically GEN/KILL (alternative: GEN/PRSV)

Example: Reaching Definitions

- **Iterative forward bit-vector formulation**
  Assign a bit position to every definition. For each position, “1” signifies that the definition may, “0” that the definition does not reach a given point.



Bit	Definition	BB
1	m (1)	
2	f0 (2)	B1
3	f1 (3)	
4	i (5)	B3
5	f2 (8)	
6	f0 (9)	B6
7	f1 (10)	
8	i (11)	

Example: Reaching Definitions

- **Iterative forward bit-vector formulation**
  Let RCHin/RCHout(b) be the set of reaching definitions that may reach the beginning/end of a basic block b. Then

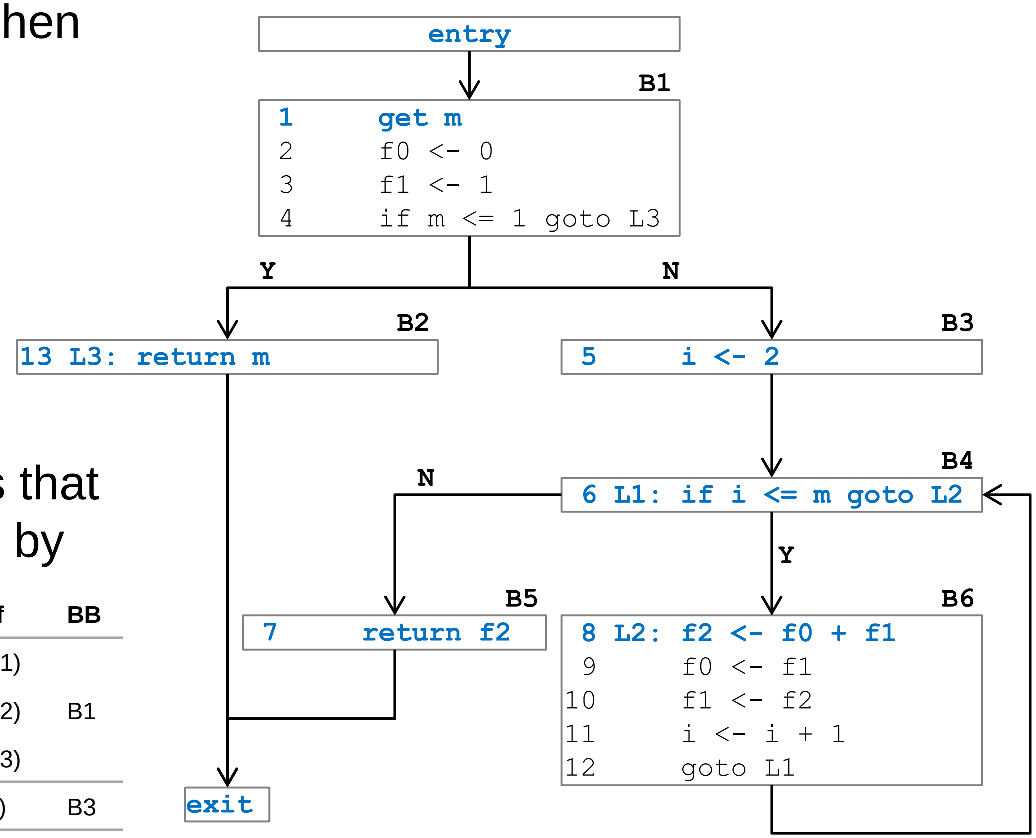
  RCHin(entry) = <00000000>

  conservative initialization:
  RCHin(i) = <00000000>
  RCHout(i) = <00000000>

  Compute PRSV(i) as the set of bits that represent the definitions preserved by block i

  PRSV(B1) = <00011001>
  PRSV(B3) = <11101110>
  PRSV(B6) = <10000000>
  all other blocks ($i \notin \{1,3,6\}$):
  PRSV(B_i) = <11111111>

Bit	Def	BB
1	m (1)	
2	f0 (2)	B1
3	f1 (3)	
4	i (5)	B3
5	f2 (8)	
6	f0 (9)	B6
7	f1 (10)	
8	i (11)	



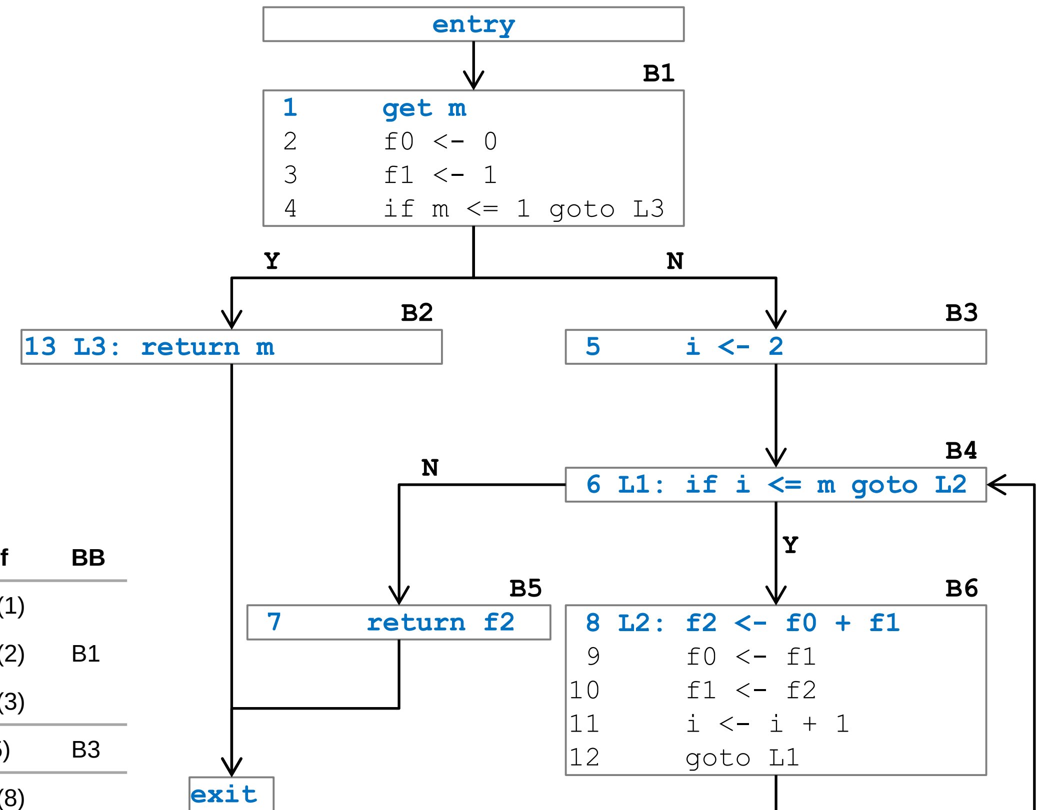
Example: Reaching Definitions

- **Iterative forward bit-vector formulation**
  Let GEN(b) be the set of generated definitions (and not subsequently killed) by block b.

GEN(B1) = <11100000>
GEN(B3) = <00010000>
GEN(B6) = <00001111>

all other blocks ($i \notin \{1,3,6\}$):
GEN(i) = <00000000>

Bit	Def	BB
1	m (1)	
2	f0 (2)	B1
3	f1 (3)	
4	i (5)	B3
5	f2 (8)	
6	f0 (9)	
7	f1 (10)	B6
8	i (11)	



Example: Reaching Definitions

- **Iterative forward bit-vector formulation**
  Data-flow formulation: a definition may reach the end of block b iff either it is generated within b or reaches the beginning of and is preserved by b.

$$\mathrm{RCHout}(b) = \mathrm{GEN}(b) \cup (\mathrm{RCHin}(b) \cap \mathrm{PRSV}(b)) \quad \text{for all i}$$

as bitvector operations

$$\mathrm{RCHout}(b) = \mathrm{GEN}(b) \vee (\mathrm{RCHin}(b) \wedge \mathrm{PRSV}(b)) \quad \text{for all i}$$

Likewise, a definition may reach the beginning of block b if it may reach the end of some predecessor of b.

$$\mathrm{RCHin}(b) = \bigcup_{j \in \mathrm{Pred}(b)} \mathrm{RCHout}(j) \quad \text{for all i}$$

$$\mathrm{RCHin}(b) = \bigvee_{j \in \mathrm{Pred}(b)} \mathrm{RCHout}(j) \quad \text{for all i}$$

Example: Reaching Definitions

- **Iterative forward bit-vector formulation**
  To solve this system, we initialize RCHin() and iteratively apply the iterative flow equations until no changes occur.
  - 1st round:

RCHout(entry)	=	<00000000>	RCHin(entry)	=	<00000000>
RCHout(B1)	=	<11100000>	RCHin(B1)	=	<00000000>
RCHout(B2)	=	<11100000>	RCHin(B2)	=	<11100000>
RCHout(B3)	=	<11110000>	RCHin(B3)	=	<11100000>
RCHout(B4)	=	<11110000>	RCHin(B4)	=	<11110000>
RCHout(B5)	=	<11110000>	RCHin(B5)	=	<11110000>
RCHout(B6)	=	<10001111>	RCHin(B6)	=	<11110000>
RCHout(exit)	=	<11110000>	RCHin(exit)	=	<11110000>

  - 2nd round:

RCHout(entry)	=	<00000000>	RCHin(entry)	=	<00000000>
RCHout(B1)	=	<11100000>	RCHin(B1)	=	<00000000>
RCHout(B2)	=	<11100000>	RCHin(B2)	=	<11100000>
RCHout(B3)	=	<11110000>	RCHin(B3)	=	<11100000>
RCHout(B4)	=	<11111111>	RCHin(B4)	=	<11111111>
RCHout(B5)	=	<11111111>	RCHin(B5)	=	<11111111>
RCHout(B6)	=	<10001111>	RCHin(B6)	=	<11111111>
RCHout(exit)	=	<11111111>	RCHin(exit)	=	<11111111>

  - termination?

A Framework for Data-Flow Analysis

- Data-flow analysis is performed by operating on **lattices**
- A *lattice* $\boldsymbol{L}$ is an algebraic structure consisting of a set of values and two operations *meet* ($\sqcap$) and *join* ($\sqcup$) with the following properties
  1. closure
     $\forall x, y \in \mathbf{L}$ there exist unique $z, w \in \mathbf{L}$ such that $x \sqcap y = z$ and $x \sqcup y = w$
  2. commutativity
     $\forall x, y \in \mathbf{L}$ $x \sqcap y = y \sqcap x$ and $x \sqcup y = y \sqcup x$
  3. associativity
     $\forall x, y, z \in \mathbf{L}$ $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$ and $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$
  4. existence of top ($\top$) and bottom ($\perp$) element
     $\exists \perp, \top$ such that $\forall x \in \mathbf{L}$ $x \sqcap \perp = \perp$ and $x \sqcup \top = \top$
  5. distributivity holds for most data-flow problems
     $\forall x, y, z \in \mathbf{L}$ $(x \sqcap y) \sqcup z = (x \sqcup z) \sqcap (y \sqcup z)$ and $(x \sqcup y) \sqcap z = (x \sqcap z) \sqcup (y \sqcap z)$

A Framework for Data-Flow Analysis

- Example: bitvector lattices

  Notation: $\mathbf{BV}^n$ lattice of bitvectors of length n

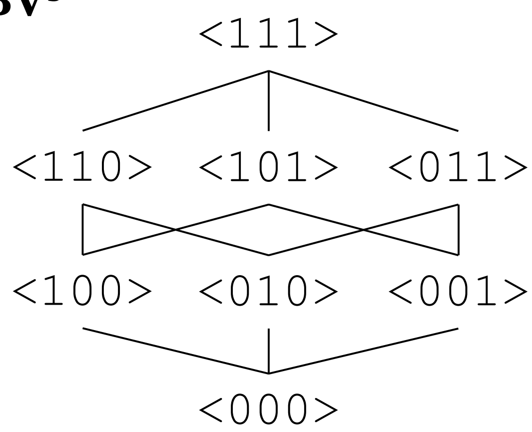
  For $\mathbf{BV}^n$, the top and bottom element are

  $\bot$ = <000…0>
  ⊤ = <111…1>

  and meet and join can be defined as

  ⊓ = bitwise AND
  ⊔ = bitwise OR

$\mathbf{BV}^3$



A Framework for Data-Flow Analysis

- Meet and join induce a partial order on the values, written as $\sqsubseteq$

  $x \sqsubseteq y$ iff $x \sqcap y = x$

  analog for join and $\sqsubset$, $\sqsupset$, and $\sqsupseteq$.

  Properties of $\sqsubseteq$:

  1. transitivity
     $\forall x, y, z \in \mathbf{L} \;\; x \sqsubseteq y$ and $y \sqsubseteq z \rightarrow x \sqsubseteq z$

  2. antisymmetry
     $\forall x, y \in \mathbf{L} \;\; x \sqsubseteq y$ and $y \sqsubseteq x \rightarrow x = y$

  3. reflexivity
     $\forall x \in \mathbf{L} \;\; x \sqsubseteq x$

A Framework for Data-Flow Analysis

- Function mappings for lattices

  f: $\mathbf{L} \rightarrow \mathbf{L}$

  - f is monotone if
    $\forall$ x, y $\in \mathbf{L}$ $x \sqsubseteq y \rightarrow f(x) \sqsubseteq f(y)$

- Height of a lattice

  the length of the longest strictly ascending chain such that

  $\perp = x_1 \sqsubset x_2 \sqsubset x_3 \sqsubset \ldots \sqsubset x_n = \top$

- Effective height of a lattice
  the effective height of a lattice is the longest strictly ascending chain obtained by iteratively applying a function f: $\mathbf{L} \rightarrow \mathbf{L}$ with

  $x_1 \sqsubset f(x_1) \sqsubset f(f(x_1)) \sqsubset f^3(x_1) \sqsubset \ldots \sqsubset f^n(x_1) \sqsubset \top$

  here, we use f^x to denote x-fold repeated application of f. The effective height is $n+1$.

A Framework for Data-Flow Analysis

- **Fixed-point** of a function f
  The fixed-point of a function f: $\mathbf{L} \rightarrow \mathbf{L}$ is reached when

  $f(\mathrm{x}) = \mathrm{x}$

- **Meet-over-all-paths (MOP)**
  Goal when solving data-flow equations

  Informally: start with some value Init at the entry (exit) node of a flowgraph, then apply the composition of the appropriate flow functions along all possible paths from the entry (exit) node and form for each node the meet of the results.

$MOP(B) = \sqcap_{p \in Path(B)} F_p(Init)$ for B = entry, B1, …, Bn, exit

with F_p being the composition of the flow functions F_{Bx} through the path p

$F_p = F_{Bn} \circ F_{Bn-1} \circ \ldots \circ F_{B1}$

A Framework for Data-Flow Analysis

- **Maximum fixed point (MFP) solution**
  Even with monotone flow functions there may be no algorithm that computes MOP for all possible flowgraphs. The algorithms compute the solution to the data-flow equation that is maximal in the ordering of the underlying lattice, that is, the solution that provides the most information. This point is called the *maximum fixed point* (MFP) solution.

- By the way: what gives more accurate results?
  associating the entry point of basic blocks with data-flow information or the edges?

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Iterative Data-Flow Analysis

- General Framework

  Given:

  - a flowgraph G = <N, E>
  - a lattice $\mathbf{L}$
  - a data-flow transfer function for block B, $F_B()$
  - $\sqcap$ models the effect of combining the data-flow information on edges entering a block

  Then

$$in(B) = \begin{cases} Init & B = entry \\ \sqcap_{P \in Pred(B)} \quad out(P) & otherwise \end{cases}$$
$$out(B) = F_B(in(B))$$

  If $\sqcup$ models the effect of combining the data-flow information, then $\sqcup$ is used in the algorithm. *Init* is typically initialized to $\perp$ or $\top$.

Iterative Data-Flow Analysis

- Similarly, for backward problems

$$out(B) = \begin{cases} Init & B = exit \\ \prod_{P \in Succ(B)} in(P) & otherwise \end{cases}$$

$$in(B) = F_B(out(B))$$

- computing F_B for every block B

  apply the transfer function to each statement in the basic block given GEN() / KILL()

Examples of Iterative Data-Flow Analysis

- **Reaching Definitions** (revisited)

  For each definition d: u = v∘w, the transfer function is given as

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

  with $gen_d = \{d\}$, and $kill_d$ the set of all other definitions of u.

  For a basic block,

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

  with

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

  and

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

Examples of Iterative Data-Flow Analysis

- **Reaching Definitions** (revisited)
  The control-flow equations for basic blocks are

$$\mathrm{OUT}[\mathrm{ENTRY}] = \emptyset;$$

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

$$\mathrm{IN}[B] = \bigcup_{P \in \mathrm{Pred}(B)} \mathrm{OUT}[P]$$

  and can now be solved iteratively:

```
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 ∈Pred(B) OUT[P];
    OUT[B] = genB ∪ (IN[B] - killB);
  }
}
```

Examples of Iterative Data-Flow Analysis

- **Live-Variable Analysis**

  Problem formulation: for variable x and program point p, can the value of x at p be used along some path starting at p?

  We define

  1. def_B as the set of variables definitely assigned in B prior to any use
  2. use_B as the set of variables whose values used in B prior to any definition

  This is a *backward* problem: we follow the uses upwards to the definitions. Hence:

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

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

$$\text{OUT}[B] = \bigcup_{S \in \text{Succ}(B)} \text{IN}[S]$$

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Examples of Iterative Data-Flow Analysis

- **Live-Variable Analysis**

iterative algorithm

```
IN[EXIT] = ∅;
for (each basic block B other than ENTRY) IN[B] = ∅;
while (changes to any IN occur) {
  for (each basic block B other than ENTRY) {
    OUT[B] = ∪_{S ∈ Succ(B)} IN[S];
    IN[B] = use_B ∪ (OUT[B] - def_B);
  }
}
```

Examples of Iterative Data-Flow Analysis

- **Available Expressions**

  Problem formulation: an expression $x \oplus y$ is available at point p if every path from the entry node to p evaluates $x \oplus y$, and after the last such evaluation prior to reaching p, there are no subsequent assignments to x or y.

  We define

  1. e_gen_B: the set of expressions generated by B
  2. e_kill_B: the set of expressions killed in B

  This is a *forward* problem: an expression is available iff it is available at the end of all predecessor blocks:

$$\mathrm{OUT}[\mathrm{EXIT}] = \emptyset;\ \mathrm{OUT}[\mathrm{N}\text{-}\{\mathrm{EXIT}\}] = \mathrm{U};$$

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

$$\mathrm{IN}[B] = \bigcap_{P \in \mathrm{Pred}(B)} \mathrm{OUT}[P]$$

Examples of Iterative Data-Flow Analysis

- **Available Expressions**

Computing e_gen_B and e_kill_B for a block:

e_gen_B , the set of available expressions in B, is empty at the beginning of the block. So is e_kill_B , the set of expressions killed by the block.

For an assignment $z = x \oplus y$

1. add expression $x \oplus y$ to e_gen_B.
2. add all expressions involving z to e_kill_B.

```
OUT[ENTRY] = ∅;
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] = ∩_{P ∈Pred(B)} OUT[P];
    OUT[B] = e_gen_B ∪ (IN[B] - e_kill_B);
  }
}
```

Iterative Data-Flow Analysis: Summary

- Summary of three data-flow problems

	Reaching Definitions	Live Variables	Available Expressions
Domain	Set 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	$\mathrm{OUT}[\mathrm{ENTRY}] = \emptyset$	$\mathrm{IN}[\mathrm{EXIT}] = \emptyset$	$\mathrm{OUT}[\mathrm{ENTRY}] = \emptyset$
Initialize	$\mathrm{OUT}[B] = \emptyset$	$\mathrm{IN}[B] = \emptyset$	$\mathrm{OUT}[B] = \top$
Meet	$\cup$	$\cup$	$\cap$
Equations	$\mathrm{OUT}[B] = f_B(\mathrm{IN}[B])$ $\mathrm{IN}[B] = \cup_{P \in \mathrm{Pred}(B)} \mathrm{OUT}[P]$	$\mathrm{IN}[B] = f_B(\mathrm{OUT}[B])$ $\mathrm{OUT}[B] = \cup_{S \in \mathrm{Succ}(B)} \mathrm{IN}[S]$	$\mathrm{OUT}[B] = f_B(\mathrm{IN}[B])$ $\mathrm{IN}[B] = \cup_{P \in \mathrm{Pred}(B)} \mathrm{OUT}[P]$