1. statement-by-statement code generation
2. peephole optimization
3. “global” optimization

# *Peephole optimization*



**Peephole:** a short sequence of target instructions that may be replaced by a shorter/faster sequence
One optimization may make further optimizations possible
$\Rightarrow$ several passes may be required

1. Redundant load/store elimination:
   `MOV R0 a`
   `MOV a R0` $\leftarrow$ delete if in same B.B.
2. Algebraic simplification: eliminate instructions like the following:
   `x = x + 0`    `x = x * 1`
3. Strength reduction: replace expensive operations by equivalent cheaper operations, e.g.
   - $\mathtt{x}^2 \rightarrow$ `x * x`
   - fixed-point multiplication/division by power of 2 $\rightarrow$ shift

# Peephole optimization

4. Jumps:

```
        goto L1                          if a < b goto L1
        ...                              ...
L1:     goto L2                   L1:    goto L2
          ⇓                                ⇓
        goto L2                          if a < b goto L2
        ...                              ...
L1:     goto L2                   L1:    goto L2
```

If there are no other jumps to `L1`, and it is preceded by an unconditional jump, it may be eliminated.

If there is only jump to `L1` and it is preceded by an unconditional goto

```
        goto L1                              if a < b goto L2
        ...                                  goto L3
L1:     if a < b goto L2      ⇒              ...
L3:                                     L3:
```

# *Peephole optimization*

5. Unreachable code: unlabeled instruction following an unconditional jump may be eliminated

```
#define DEBUG 0                              if debug = 1 goto L1
...                                          goto L2
if (debug) {                    ⇒        L1: /* print */
    /* print stmts */                    L2:
}
                                                   ⇓

    if 0 != 1 goto L2                             if debug != 1 goto L2
    /* print stmts */               ⇐             /* print stmts */
L2:                                           L2:
```

eliminate

# Example

```
i = m-1; j = n; v = a[n];
while (1) {
    do i = i+1; while (a[i] < v);
    do j = j-1; while (a[j] > v);
    if (i >= j) break;
    x = a[i]; a[i] = a[j]; a[j] = x;
}
x = a[i]; a[i] = a[n]; a[n] = x;
```

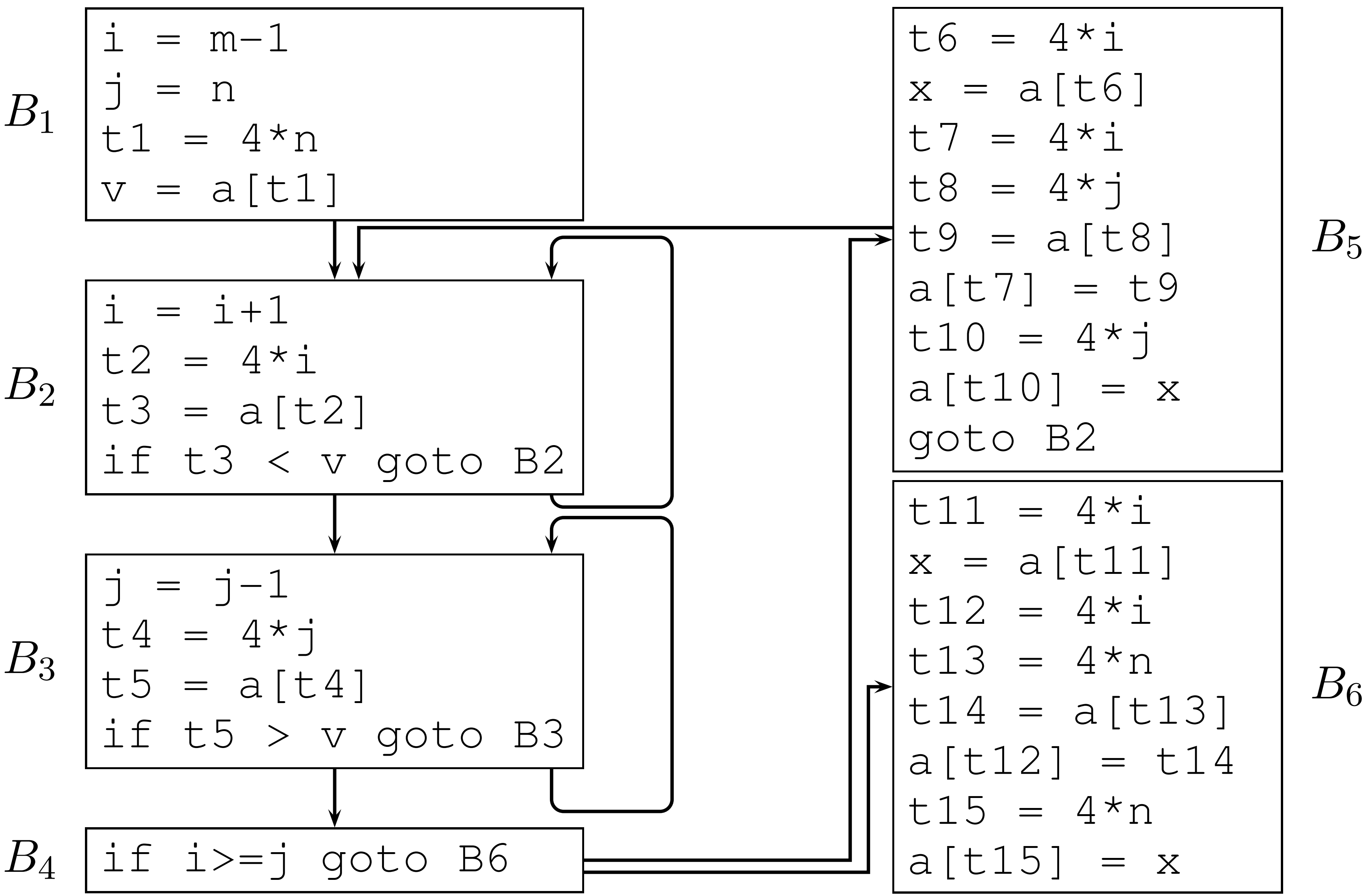
# Intermediate code

```
 1   i = m-1
 2   j = n
 3   t1 = 4*n
 4   v = a[t1]
 5   i = i+1
 6   t2 = 4*i
 7   t3 = a[t2]
 8   if t3 < v goto 5
 9   j = j-1
10   t4 = 4*j
11   t5 = a[t4]
12   if t5 > v goto 9
13   if i>=j goto 23
14   t6 = 4*i
15   x = a[t6]
16   t7 = 4*i
17   t8 = 4*j
18   t9 = a[t8]
19   a[t7] = t9
20   t10 = 4*j
21   a[t10] = x
22   goto 5
23   t11 = 4*i
24   x = a[t11]
25   t12 = 4*i
26   t13 = 4*n
27   t14 = a[t13]
28   a[t12] = t14
29   t15 = 4*n
30   a[t15] = x
```

# Basic blocks

```
1   i = m-1
2   j = n
3   t1 = 4*n
4   v = a[t1]
5   i = i+1
6   t2 = 4*i
7   t3 = a[t2]
8   if t3 < v goto 5
9   j = j-1
10  t4 = 4*j
11  t5 = a[t4]
12  if t5 > v goto 9
13  if i>=j goto 23
14  t6 = 4*i
15  x = a[t6]
16  t7 = 4*i
17  t8 = 4*j
18  t9 = a[t8]
19  a[t7] = t9
20  t10 = 4*j
21  a[t10] = x
22  goto 5
23  t11 = 4*i
24  x = a[t11]
25  t12 = 4*i
26  t13 = 4*n
27  t14 = a[t13]
28  a[t12] = t14
29  t15 = 4*n
30  a[t15] = x
```

# Flow graph


$B_1$
i = m-1
j = n
t1 = 4*n
v = a[t1]
$B_2$
i = i+1
t2 = 4*i
t3 = a[t2]
if t3 < v goto B2
$B_3$
j = j-1
t4 = 4*j
t5 = a[t4]
if t5 > v goto B3
$B_4$
if i>=j goto B6
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2
$B_5$
t11 = 4*i
x = a[t11]
t12 = 4*i
t13 = 4*n
t14 = a[t13]
a[t12] = t14
t15 = 4*n
a[t15] = x
$B_6$

# *Optimization methods*

- **Common subexpression elimination:**
  **Def.** $E$ is a common subexpression at some point in the program if it has been previously computed and the values of variables in $E$ have not changed since the last computation

  NOTE: local vs. global C.S.E.
  array expressions

Section 10.2

# *Optimization methods*

- **Common subexpression elimination:**
  **Def.** $E$ is a common subexpression at some point in the program if it has been previously computed and the values of variables in $E$ have not changed since the last computation

  NOTE: local vs. global C.S.E.
  array expressions

- **Copy propagation:** after the copy statement `x = y`, use `y` wherever possible in place of `x`

# Optimization methods



- **Common subexpression elimination:**
  **Def.** $E$ is a common subexpression at some point in the program if it has been previously computed and the values of variables in $E$ have not changed since the last computation
  NOTE: local vs. global C.S.E.
  array expressions
- **Copy propagation:** after the copy statement `x = y`, use `y` wherever possible in place of `x`
- **Dead code elimination:**
  **Dead variable:** `v` is dead at a point if its value is not used after that point
  **Dead code:** statements which compute values that are never used

# Loop optimizations

- **Code motion:** if a statement is independent of the number of times a loop is executed ($\equiv$ loop invariant computation), move it outside the loop
  Example:

```
while (i <= N-1) ...
```

$\Rightarrow$

```
t = N-1
while (i <= t) ...
```

- **Induction variable elimination:**
  **Induction variable:** variable whose value has a simple relation with no. of loop iterations
- **Strength reduction:** replacing expensive operation by cheaper one (e.g. multiplication by addition)

# *Data flow analysis*

Section 10.5

**Motivation:** collect information like live variables, common subexpressions, etc. about entire program for optimization (and code generation)

**Plan:**

- Structured programs
  - reaching definitions
- Flow graphs / Iterative solutions
  - reaching definitions
  - available expressions
  - live variables

# *Structured programs*

- Control-flow changes only via **if** and **while** stmts (no arbitrary **goto**s)
- Source-level grammar:

$$
\begin{array}{lcl}
S & \rightarrow & \textbf{id} = E \\
 & | & S \ ; \ S \\
 & | & \textbf{if } E \textbf{ then } S \textbf{ else } S \\
 & | & \textbf{do } S \textbf{ while } E
\end{array}
$$

# *Reaching definitions*

**Point:** position between 2 adjacent stmts within a BB, before the 1st stmt in a BB, and after the last stmt in a BB

**Path:** sequence of points $p_1, p_2, \ldots, p_n$ s.t. $p_i$ immediately precedes and $p_{i+1}$ immediately follows an instruction, or $p_i$ ends a block, and $p_{i+1}$ begins a successor block

**Definition** of a variable $\mathrm{x}$ is a stmt that assigns or may assign a value to $\mathrm{x}$

- **Unambiguous** defn. – stmt that definitely assigns a value to $\mathrm{x}$, e.g. direct assignments, I/O
- **Ambiguous** defn. – stmt that *may* change the value of $\mathrm{x}$, e.g. indirect assignment, procedure call

**Kill:** a definition of a variable is killed on a path, if there is a (unambiguous) definition of the variable along that path

# Reaching definitions

**Reaching Definition:** A definition $d$ reaches point $p$ if there is a path from the point immediately following $d$ to $p$, and $d$ is not killed along that path

Applications: used for constant folding, code motion, induction variable elimination, dead code elimination, etc.

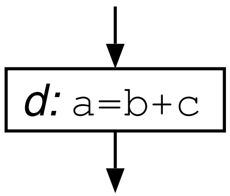
# Reaching definitions

**Reaching Definition:** A definition $d$ reaches point $p$ if there is a path from the point immediately following $d$ to $p$, and $d$ is not killed along that path

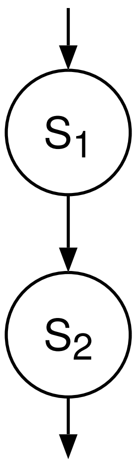
Applications: used for constant folding, code motion, induction variable elimination, dead code elimination, etc.

| | | |
|---|---|---|
| $in(S)$ | – | set of definitions reaching the beginning of stmt $S$ |
| $out(S)$ | – | set of definitions reaching the end of stmt $S$ |
| $gen(S)$ | – | set of definitions that reach the end of $S$, irrespective of whether they reach the beginning of $S$ |
| $kill(S)$ | – | set of definitions that never reach the end of $S$, even if they reach the beginning of $S$ |

# Data flow equations



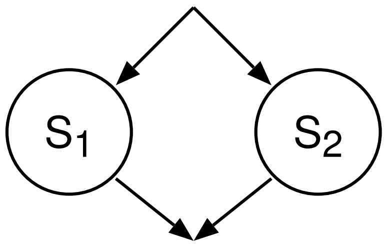

$$gen(S) = \{d\}$$
$$kill(S) = D_a - \{d\}$$




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

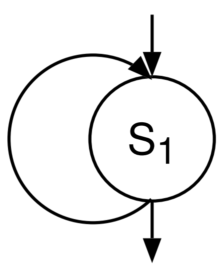
$$in(S_1) = in(S)$$
$$in(S_2) = out(S_1)$$
$$out(S) = out(S_2)$$

# Data flow equations




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

$$in(S_1) = in(S_2) = in(S)$$
$$out(S) = out(S_1) \cup out(S_2)$$




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

$$in(S_1) = in(S) \cup gen(S_1)$$
$$out(S) = out(S_1)$$

# *Data flow equations*

1. Compute *gen*, *kill* (synthesized attributes) bottom up from smallest stmt to largest stmt.
2. Let $S_0$ represent the complete program. Then $in(S_0) = \emptyset$.
3. For $S_1$, a sub-statement of $S$:
   (i) calculate $in(S_1)$ in terms of $in(S)$;
   (ii) calculate $out(S_1)$ using the equation
   $$\boxed{out(S) = gen(S) \cup (in(S) - kill(S))}$$
4. Calculate $out(S)$ in terms of $out(S_1)$.

# Iterative approach

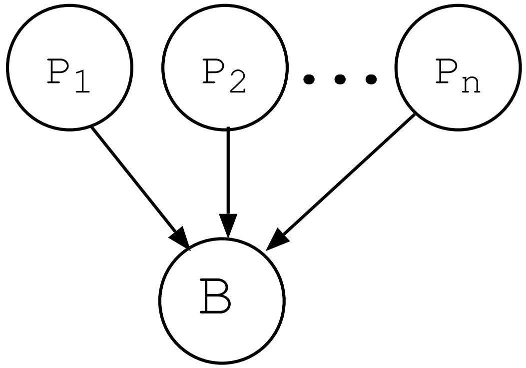


**Reaching definitions**

**Input:** flow graph with $gen$, $kill$ sets computed for each BB

**Output:** $in$, $out$ sets for each block

**Method:**




1. For each block, initialize
   $out(B) \leftarrow gen(B)$
   (assume $in(B) = \emptyset$)
2. While changes occur:
   for each block $B$
   $in(B) \leftarrow \bigcup_P out(P)$ where $P$ - predecessor of $B$
   $out(B) \leftarrow gen(B) \cup (in(B) - kill(B))$

# *Reaching definitions*

- Changes are monotonic $\Rightarrow$ method converges.
- While loop (step 2) simulates all possible alternatives for control flow during the execution of the program.
- *If a definition reaches a point, it can do so along a cycle-free path.*
  Longest cycle free path in the graph can cover at most all nodes, at most once.
  $\Rightarrow$ Upper bound on # iterations = # of nodes in the flow graph.
  Avg. # of iterations for covergence for real programs = 5

# U-D chains

**Definition:** for a given *use* of a variable `a`, the *ud chain* is a list of all definitions of `a` that reach that use

**Method:**
Given a use of variable `a` in block $B$

1. If the use is not preceded by an unambiguous defn. of `a` within $B$, *ud* = set of defns of `a` in $in(B)$.
2. If there is an unambiguous defn of `a` within $B$ prior to this use, then *ud* = { most recent unambiguous defn. }
3. In addition, if there are ambiguous definitions of `a`, then add all those for which no unambiguous defn. lies between it and the current use to the *ud* chain.

# *Available expressions*

**Definition:** `x+y` is *available* at point $p$ if every path from the initial node to $p$ evaluates `x+y` and there are no subsequent assignments to `x` or `y` after the last evaluation

**Kill:** `x+y` is killed if `x` or `y` is assigned and `x+y` is not subsequently recomputed

**Gen:** `x+y` is generated if the value of `x+y` is computed and `x`, `y` are not subsequently redefined

# *Available expressions*

**Calculating AEs within a block**

1. Initialize $A \leftarrow \emptyset$.
2. Consider each assignment `x = y+z` within the block in turn:
   (i) add `y+z` to $A$
   (ii) delete any expression involving `x` from $A$
3. At the end, $gen = A$
   $kill$= set of all expressions `y+z` s.t. `y` or `z` is defined within the block and `y+z` $\notin A$

# Available expressions

**Input:** flow graph with $e\_gen$, $e\_kill$ sets for each BB

**Output:** $in$, $out$ sets for each block

**Method:**

1. Initialize $in(B_1) \leftarrow \emptyset$, $out(B_1) \leftarrow e\_gen(B_1)$ ($B_1$ - initial node)
2. For each $B \neq B_1$, initialize $out(B) \leftarrow \mathcal{U} - e\_kill(\mathcal{B})$
3. While changes occur:
   for each $B \neq B_1$
   $in(B) \leftarrow \bigcap_P out(P)$ where $P$ - predecessor of $B$
   $out(B) \leftarrow e\_gen(B) \cup (in(B) - e\_kill(B))$

# Live variables

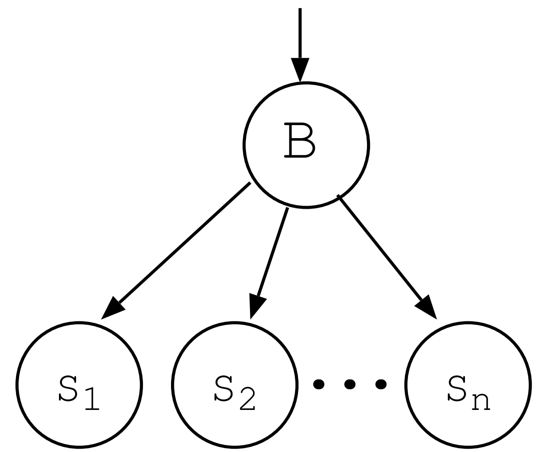
| | | |
|---|---|---|
| $in(B)$ | – | set of variables live at the initial point of $B$ |
| $out(B)$ | – | set of variables live at the end point of $B$ |
| $def(B)$ | – | set of variables definitely assigned values in $B$ prior to any use in $B$ |
| $use(B)$ | – | set of variables whose values may be used in $B$ prior to any definition of the variable |

# *Live variables*

**Input:** flow graph with $def$, $use$ sets for each BB

**Output:** $in$, $out$ sets for each block

**Method:**




1. For each $B$, initialize
   $in(B) \leftarrow use(B)$

2. While changes occur:
   for each block $B$
   $out(B) \leftarrow \bigcup_S in(S)$ where $S$ - successor of $B$
   $in(B) \leftarrow use(B) \cup (out(B) - def(B))$

**D-U chain:** *du-chain* for a variable x at a given point $p$ is the set of uses $s$ of the variable s.t. there is a path from $p$ to $s$ that does not redefine x

# *Common subexpression elimination*



**Input:** flow graph with available expression information

**Output:** revised flow graph

**Method:**
For each stmt $s$ `x = y+z` s.t. `y+z` is available at the beginning of $s$' block and `y`, `z` are not defined prior to $s$ within the block:

1. follow flow graph edges backwards until a block containing an evaluation `w = y+z` is found
2. create a new variable `u`
3. replace `w = y+z` by `u = y+z    w = u`
4. replace `x = y+z` by `x = u`

# *Common subexpression elimination*

- Should be used with copy propagation
- May need multiple passes for best effect

# Copy propagation

**Principle:** Given $s$: $\boxed{\texttt{x=y}}$, $\texttt{y}$ can be substituted for $\texttt{x}$ in all uses $u$ of $\texttt{x}$ if

1. $s$ is the only definition of $\texttt{x}$ reaching $u$
2. on every path from $s$ to $u$ (including paths that go through $u$ several times but not more than once through $s$) there are no assignments to $\texttt{y}$

# Copy propagation

| | | |
|---|---|---|
| $in(B)$ | – | set of copies $s$: $\boxed{\texttt{x=y}}$ s.t. every path from initial node to beginning of $B$ contains $s$ and there are no assignments to `x`, `y` after the last occurrence of $s$ |
| $out(B)$ | – | as above |
| $gen(B)$ | – | all copies $s$: $\boxed{\texttt{x=y}}$ in $B$ s.t. there are no assignments to `y` within $B$ after $s$ |
| $kill(B)$ | – | all copies $s$: $\boxed{\texttt{x=y}}$ s.t. `x` or `y` is assigned in $B$ and $s \notin B$ |

# Copy propagation

$in(B)$ – set of copies $s$: $\boxed{\texttt{x=y}}$ s.t. every path from initial node to beginning of $B$ contains $s$ and there are no assignments to $\texttt{x}$, $\texttt{y}$ after the last occurrence of $s$

$out(B)$ – as above

$gen(B)$ – all copies $s$: $\boxed{\texttt{x=y}}$ in $B$ s.t. there are no assignments to $\texttt{y}$ within $B$ after $s$

$kill(B)$ – all copies $s$: $\boxed{\texttt{x=y}}$ s.t. $\texttt{x}$ or $\texttt{y}$ is assigned in $B$ and $s \notin B$

$$
\begin{aligned}
out &= gen \cup (in - kill) \\
in(B_1) &= \emptyset \\
in(B) &= \bigcap_P out(P) \text{ where } P \text{ - pred. of } B, \quad B \neq B_1
\end{aligned}
$$

# *Copy propagation*

**Input:** flow graph with ud chains, du chains, $in(B)$

**Output:** revised flow graph

**Method:** For each $s$: `x=y` do:

1. Determine the uses of `x` reached by this definition.
2. Determine whether for every use of `x` in (1), $s \in in(B)$ for the block containing the use and no definitions of `x`, `y` occur prior to this use within $B$.
3. If $s$ meets the conditions in (2), remove $s$ and replace all uses of `x` found in (1) by `y`.

# Loops



**Dominator:** a node $d$ dominates node $n$ if every path from the initial node to $n$ goes through $d$

**Back edge:** an edge $a \rightarrow b$ in a flow graph is a back edge if $b$ dominates $a$

**Natural loop:** given a back edge $n \rightarrow d$, the natural loop for this edge consists of $d$ along with all nodes from which we can reach $n$ without going through $d$

**Header:** the node that dominates all other nodes in a loop

**Pre-header:** Given a loop $L$ with header $h$:

1. create an empty block $p$;
2. make $h$ the only successor of $p$;
3. all edges which entered $h$ from outside $L$ are changed to point to $p$ (edges to $h$ from inside $L$ are not changed).

# *Loop-invariant computations*

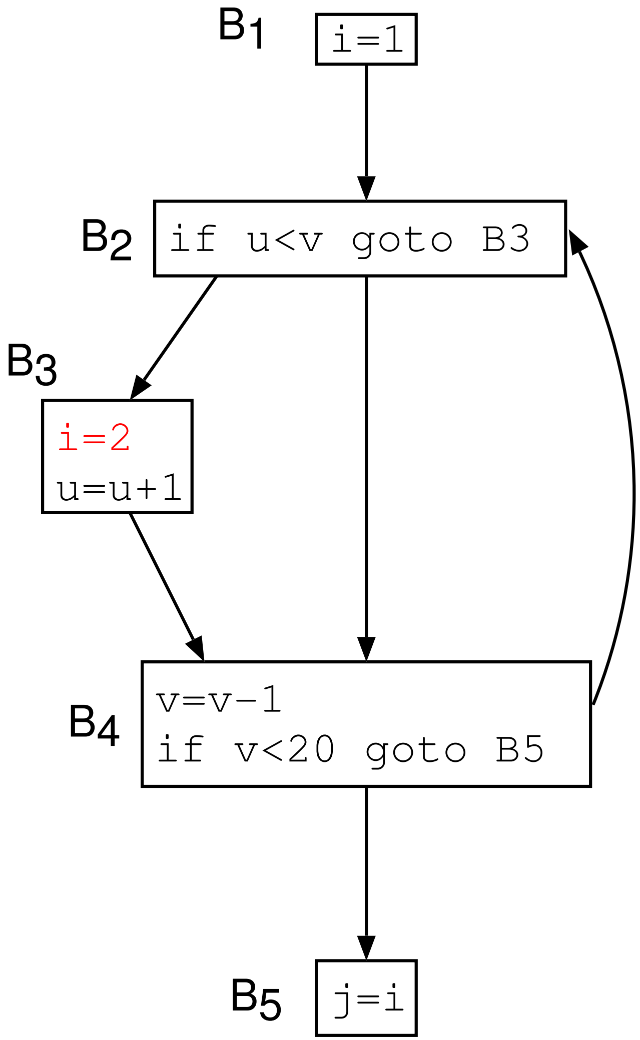
**Input:** loop $L$ + ud chains for statements in the loop

**Output:** statements that perform loop-invariant computations

**Method:**

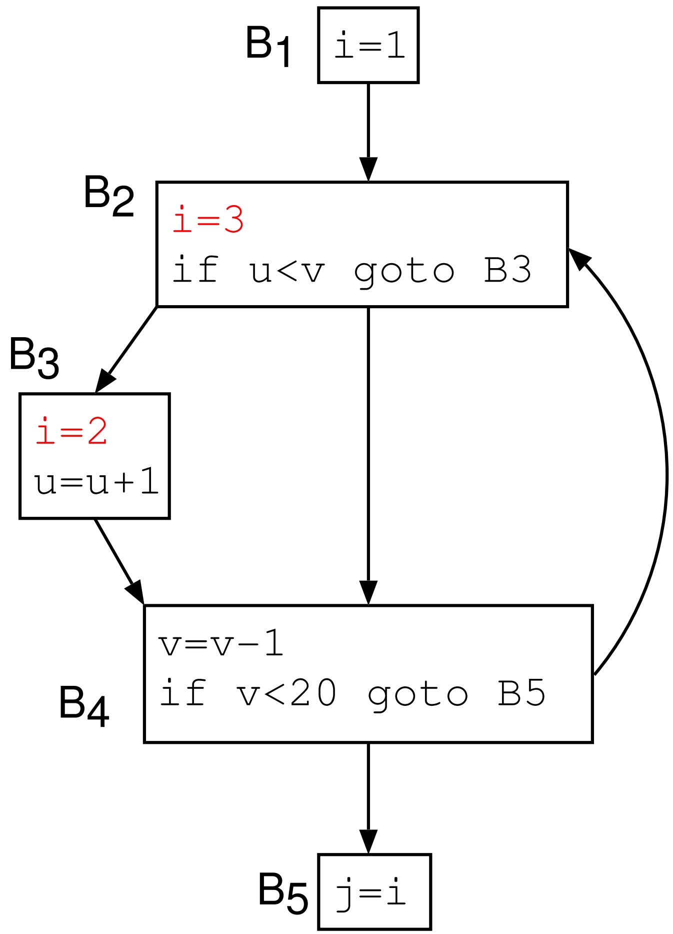
1. Mark “invariant” any statement whose operands are all either constants or have all their reaching definitions outside $L$.
2. Repeat until no further changes:
   mark “invariant” any statement that is not already marked and all of whose operands satisfy one of the following conditions:
   (i) the operand is a constant
   (ii) the operand has all its reaching definitions outside $L$
   (iii) the operand has exactly one reaching definition, and that definition is a statement in $L$ that has been marked invariant

# Conditions for moving `x = y+z`




The block containing $s$ must dominate all exit nodes of the loop (i.e. nodes with a successor not in the loop).

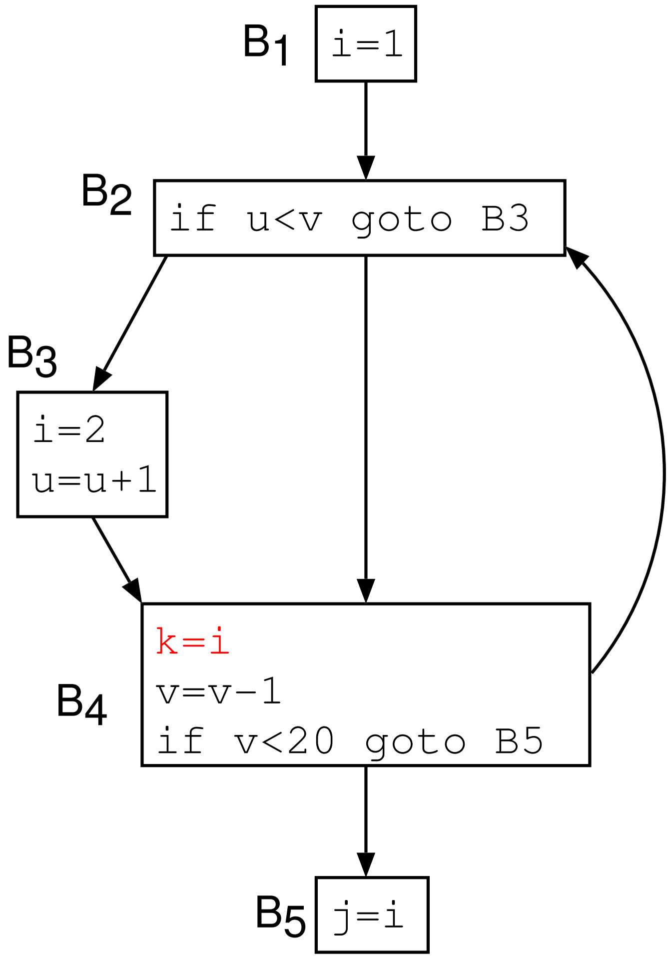
# *Conditions for moving `x = y+z`*




There is no other assignment to `x` in the loop.

(usually satisfied by temporaries)

# Conditions for moving `x = y+z`




No use of $\mathtt{x}$ in the loop is reached by any definition of $\mathtt{x}$ other than $s$.

(usually satisfied by temporaries)

# Code motion

**Input:** loop $L$ with ud chains and dominator information

**Output:** revised loop with a preheader

**Method:**

1. Find loop-invariant computations (see above).
2. For each statement $s$ defining $\mathrm{x}$ found in (1), check whether:
   (i) it is in a block that dominates all exits of $L$
   (ii) $\mathrm{x}$ is not defined elsewhere in $L$
   (iii) all uses in $L$ of $\mathrm{x}$ can only be reached by the definition of $\mathrm{x}$ in statement $s$
3. Move all stmts $s$ that satisfy (2) to the preheader in the order in which they were found in (1) provided any operands of $s$ that are defined in loop $L$ have also had their definitions moved to the preheader.