CS 346: Code Optimization

Code Optimization

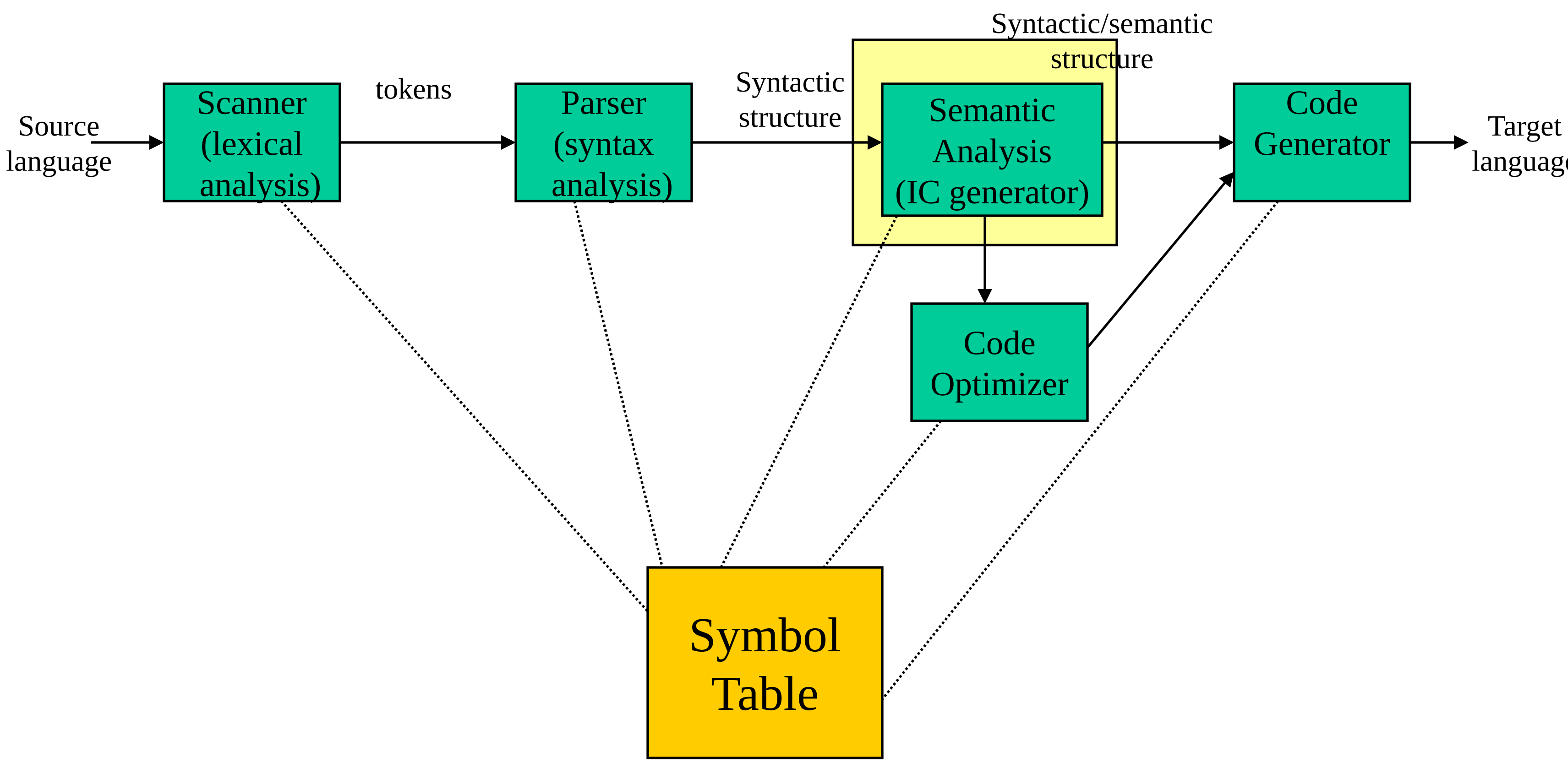
REQUIREMENTS:

- Meaning must be preserved (correctness)
- Speedup must occur on average
- Work done must be worth the effort

OPPORTUNITIES:

- Programmer (algorithm, directives)
- Intermediate code
- Target code

Code Optimization



Why Optimization ?

- Avoid redundancy: *something already computed need not be computed again*
- Smaller code: *less work for CPU, cache, and memory!*
- Less jumps: *jumps interfere with code pre-fetch*
- Code locality: *codes executed close together in time is generated close together in memory – increase locality of reference*
- Extract more information about code: *More info – better code generation*

Criteria for Transformations

- Must preserve the meaning of a program
  - Can not change the output produced for any input
  - Can not introduce an error
- Transformations should, on average, speed up programs
- Transformations should be worth the effort

Beyond Optimizing Compilers

- Really improvements can be made at various phases
- Source code:
  - Algorithmic transformations can produce spectacular improvements
  - Profiling can be helpful to focus a programmer's attention on important code
- Intermediate code:
  - Compiler can improve loops, procedure calls, and address calculations
  - Typically only optimizing compilers include this phase
- Target code:
  - Compilers can use registers efficiently
  - Peephole transformation can be applied

Local vs. Global Transformations

- *Local transformations* involve statements within a single basic block
- All other transformations are called *global transformations*
- *Local transformations* are generally performed first
- Many types of transformations can be performed either locally or globally

Levels

- Window – peephole optimization
- Basic block
- Procedural – global (control flow graph)
- Program level – intraprocedural (program dependence graph)

Peephole Optimizations

- Simple technique to improve target code locally
  - can also be applied to intermediate code
- Peephole: *small, moving window on the target program*
- Each improvement replaces the instructions of the peephole with a shorter or faster sequence
- Each improvement may create opportunities for additional improvements
- Repeated passes may be necessary

Peephole Optimizations

- Constant Folding

  `x := 32` becomes **`x := 64`**

  `x := x + 32`
- Unreachable Code

  `goto L2`

  `x := x + 1` ← unneeded
- Flow of control optimizations

  `goto L1` becomes **`goto L2`**

  …

  `L1: goto L2`

Peephole Optimizations

- Algebraic Simplification

  `x := x + 0` ← unneeded

- Dead code

  `x := 32` ← where x not used after statement

  `y := x + y` → `y := y + 32`

- Reduction in strength

  `x := x * 2` → `x := x + x`

Peephole Optimizations

- Local in nature
- Pattern driven
- Limited by the size of the window

Basic Block Level

- Common Subexpression elimination
- Constant Propagation
- Dead code elimination
- Many others such as copy propagation, value numbering, partial redundancy elimination, …

Simple example: a[i+1] = b[i+1]

- t1 = i+1
- t2 = b[t1]
- t3 = i + 1
- a[t3] = t2

- t1 = i + 1
- t2 = b[t1]
- t3 = i + 1 ← *no longer live*
- a[t1] = t2

Common expression can be eliminated

Now, suppose i is a constant:

- i = 4
- t1 = i+1
- t2 = b[t1]
- a[t1] = t2

- i = 4
- t1 = 5
- t2 = b[t1]
- a[t1] = t2

- i = 4
- t1 = 5
- t2 = b[5]
- a[5] = t2

Final Code:

- i = 4
- t2 = b[5]
- a[5] = t2

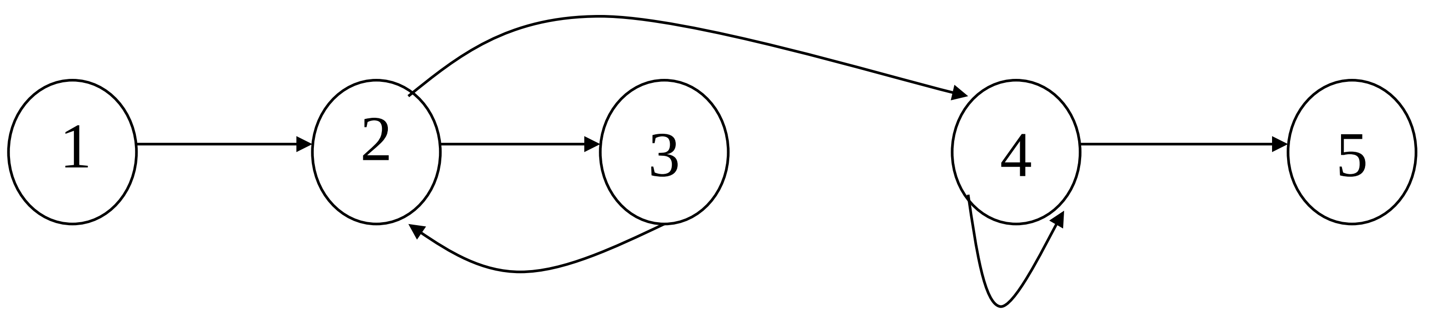
Control Flow Graph - CFG

CFG = < V, E, Entry >, where

- V = vertices or nodes, representing an instruction or basic block (group of statements).
- E = (V x V) edges, potential flow of control
- Entry is an element of V, the unique program entry

Two sets used in algorithms:

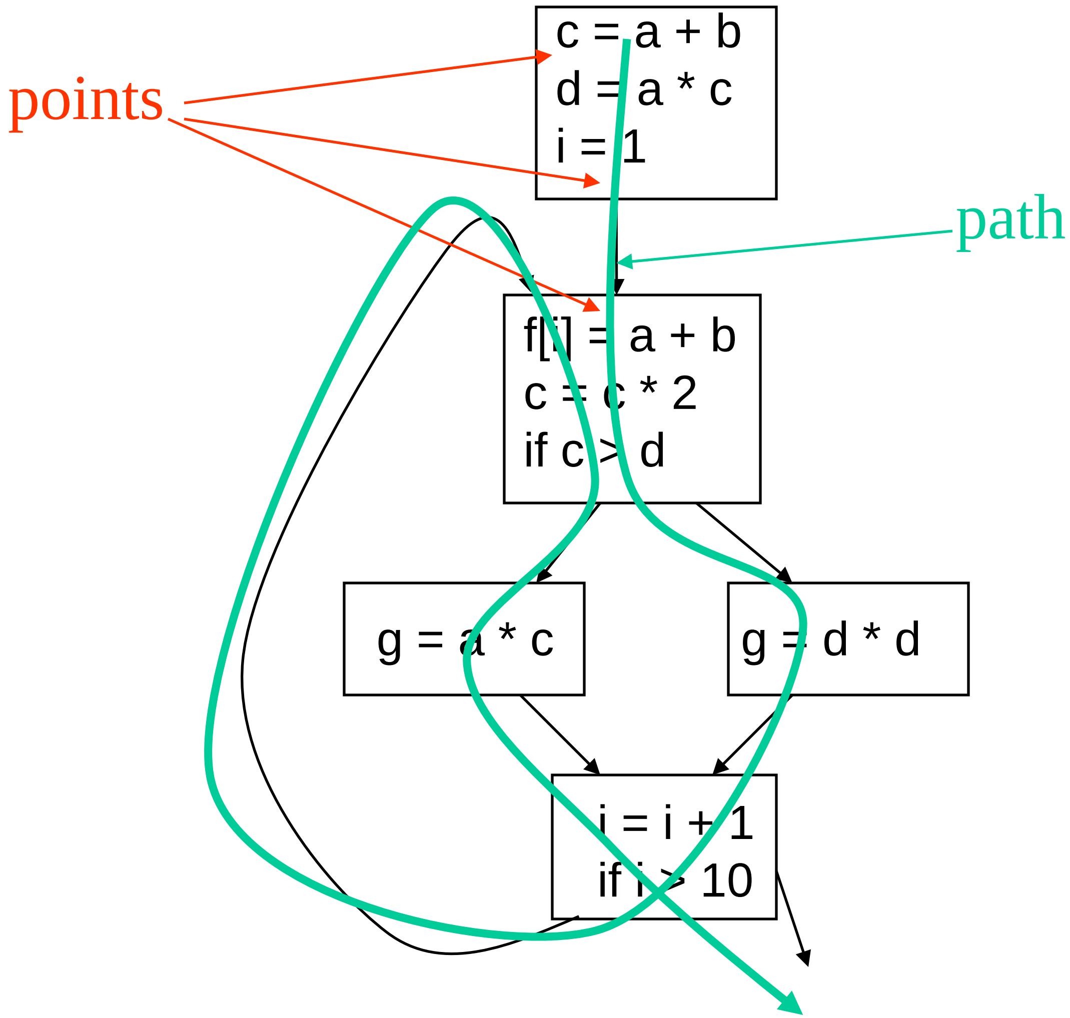
- Succ(v) = {x in V| exists e in E, e = v → x}
- Pred(v) = {x in V| exists e in E, e = x → v}



Definitions

- point - any location between adjacent statements and before and after a basic block
- A path in a CFG from point p_1 to p_n is a sequence of points such that $\forall$ j, 1 <= j < n, either p_i is the point immediately preceding a statement and p_{i+1} is the point immediately following that statement in the same block, or p_i is the end of some block and p_{i+1} is the start of a successor block

CFG



Optimizations on CFG

- Must take control flow into account
  - Common Sub-expression Elimination
  - Constant Propagation
  - Dead Code Elimination
  - Partial redundancy Elimination
  - …
- Applying one optimization may create opportunities for other optimizations.

Redundant Expressions

An expression **`x op y`** is redundant at a point p if it has already been computed at some point(s) and no intervening operations redefine **`x`** or **`y`**.

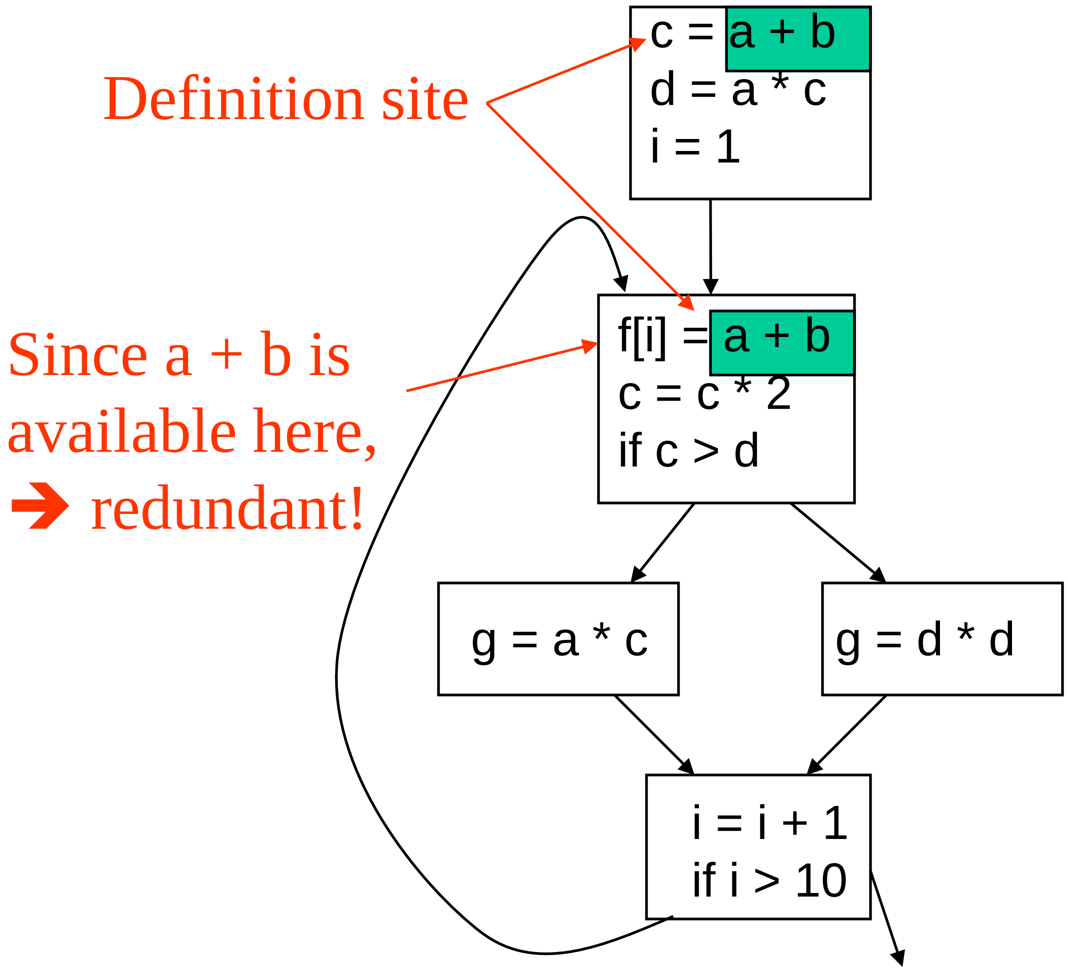
```
m = 2*y*z

n = 3*y*z

o = 2*y-z
```

redundant

```
t0 = 2*y
m = t0*z
t1 = 3*y
n = t1*z
t2 = 2*y
o = t2-z
```

```
t0 = 2*y
m = t0*z
t1 = 3*y
n = t1*z

o = t0-z
```

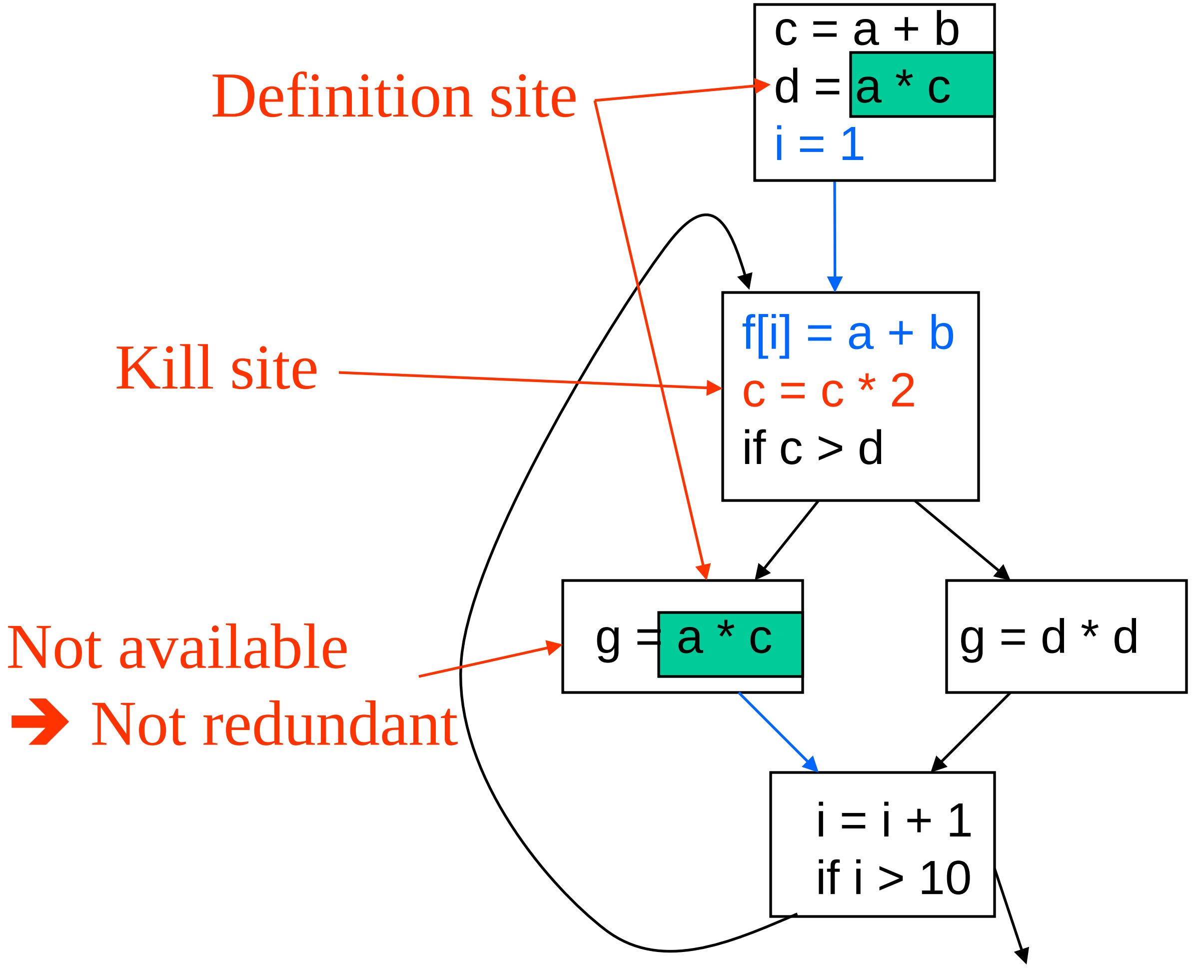
Redundant Expressions



Candidates:

a + b
a * c
d * d
c * 2
i + 1

Redundant Expressions

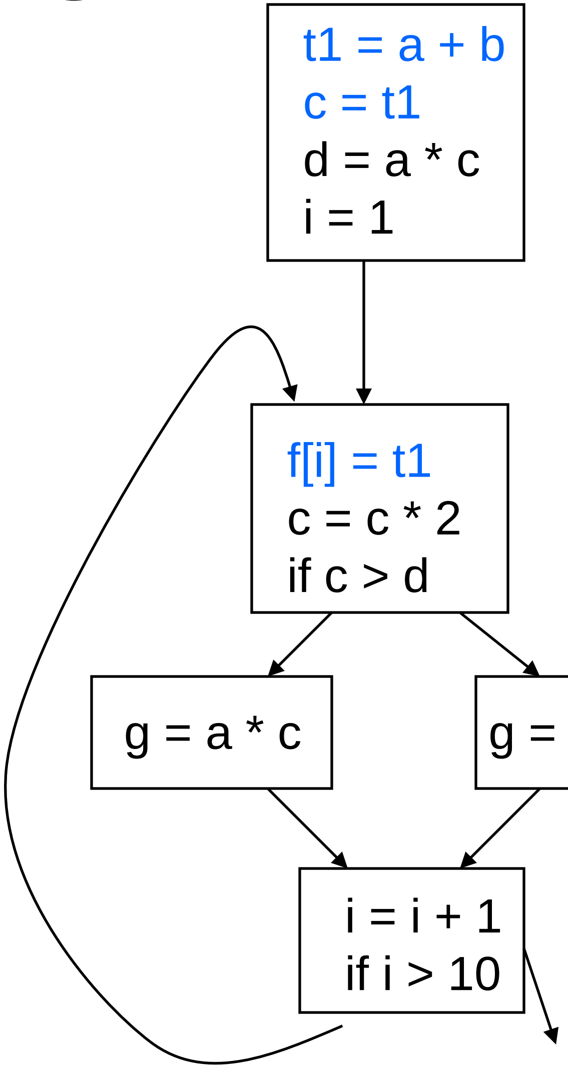


Candidates:
a + b
a * c
d * d
c * 2
i + 1

Redundant Expressions

- An expression *e* is defined at some point *p* in the CFG if its value is computed at *p* (*definition site*)
- An expression *e* is killed at point *p* in the CFG if one or more of its operands is defined at *p* (*kill site*)
- An expression is ***available*** at point *p* in a CFG if every path leading to *p* contains a prior definition of *e* and *e* is not killed between that definition and *p*.

Removing Redundant Expressions



Candidates:

a + b

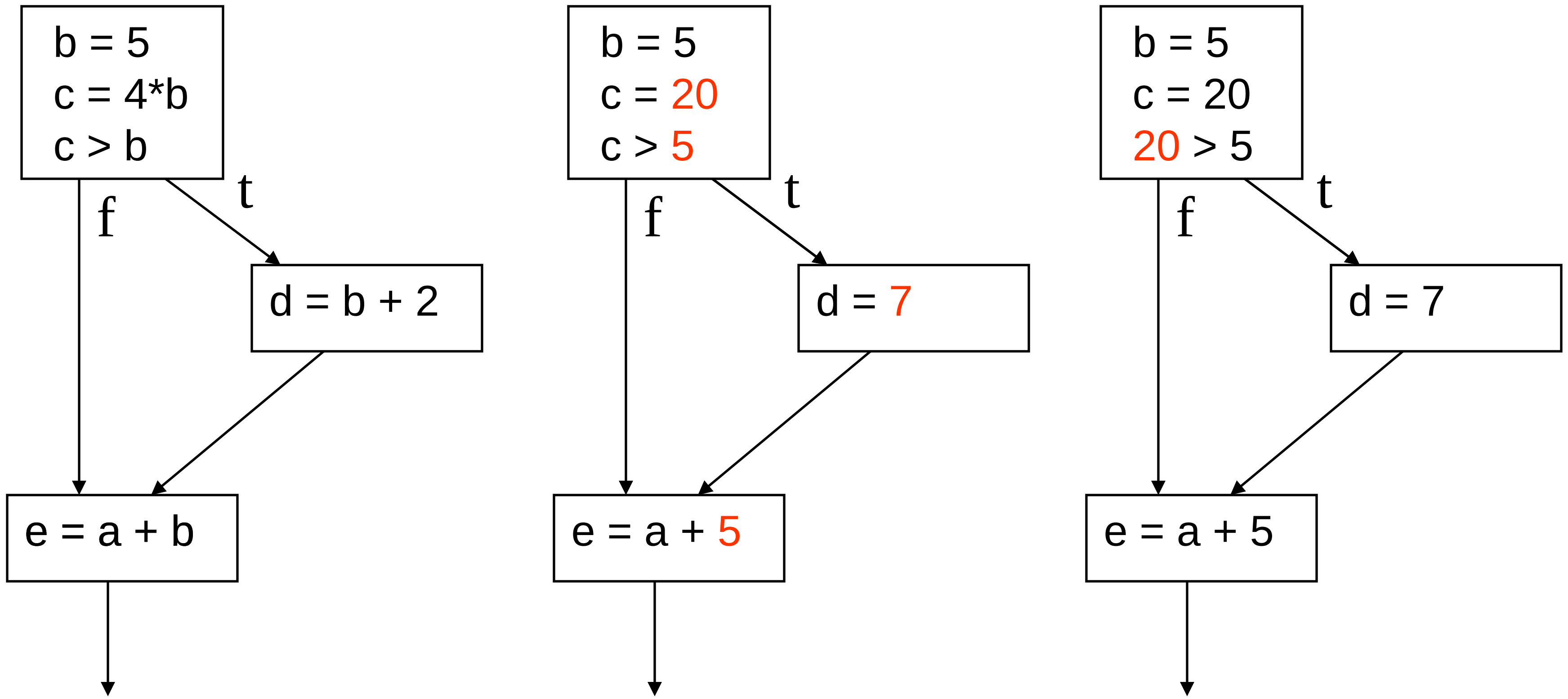
~~a * c~~

~~d * d~~

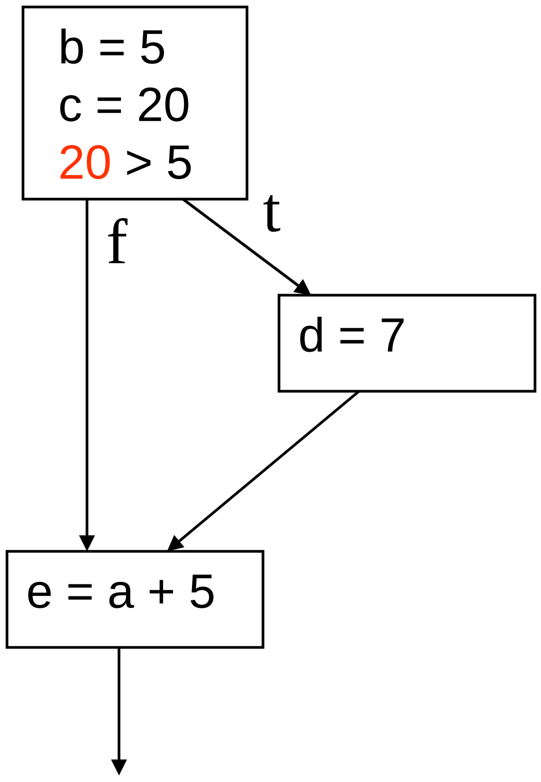
~~c * 2~~

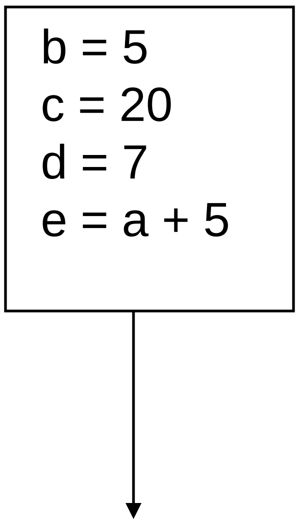
~~i + 1~~

Constant Propagation



Constant Propagation

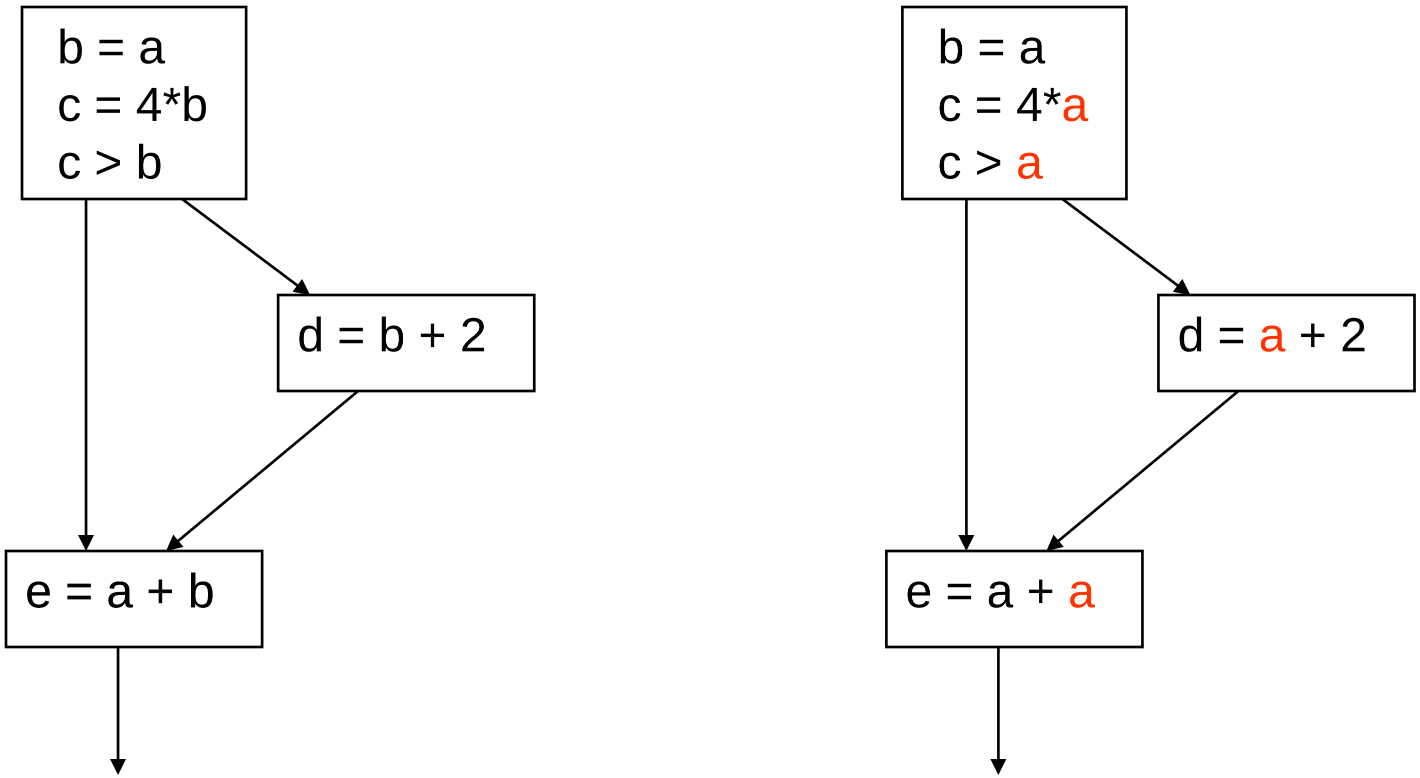




Constant Propagation

- *Goal*- to discover all the values that are constant at all possible executions
- Expressions whose operands are all constants can be evaluated at compile time
- Expressions evaluated at compile time need not be evaluated at execution time
- Code that is never executed can be deleted
- Produces smaller code
- Can lead towards the requirement of fewer registers

Copy Propagation



Copy Propagation

Definition: Given an assignment x = y, replace later uses of x with uses of y, provided there are no intervening assignments to x or y

When is it performed?

At any level, but usually early in the optimization process

Why?

To produce smaller code

Code Motion

- Moving code from one part of the program to other without modifying the algorithm
  - Reduces size of the program
  - Reduces execution frequency of the code subjected to movement

Simple Loop Optimizations: Code Motion

```
while (i <=  limit - 2)
```

```
L1:
    t1 = limit – 2
    if (i > t1) goto L2
    body of loop
    goto L1
L2:
```

```
t := limit - 2
   while (i <= t)
```

```
    t1 = limit – 2
L1:
    if (i > t1) goto L2
    body of loop
    goto L1
L2:
```

Code Motion

1. *Code Space reduction*: Similar to common sub-expression elimination but with the objective to reduce code size

Example: Code hoisting

```
                              temp : = x ** 2
if (a< b) then                if (a< b) then
    z := x ** 2      ⟶             z := temp
else                          else
    y := x ** 2 + 10               y := temp + 10
```

“x ** 2“ is computed once in both cases, but the code size in the second case reduces.

Code Motion

2 *Execution frequency reduction*: reduces execution frequency of partially available expressions (expressions available atleast in one path)

Example:

```
if (a<b) then
        z = x * 2

else
y = 10

g = x * 2
```

⟶

```
if (a<b) then
   temp = x * 2
   z = temp
else
   y = 10
   temp = x * 2
g = temp
```

```
temp = x * 2
if (a<b)
z=temp
else
y=10

 g=temp
```

Code Motion

- Move expression out of a loop if the evaluation does not change inside the loop

Example:

```
while ( i < (max-2) ) …
```

Equivalent to:

```
t :=  max - 2
while ( i < t ) …
```

Code Motion

- Safety of Code movement

  Movement of an expression *e* from a basic block b_i to another block b_j, is safe if it does not introduce any new occurrence of *e* along any path

Example: Unsafe code movement (?- *find out*)

```
if (a<b) then
   z = x * 2
else
   y = 10
```

$\longrightarrow$

```
temp = x * 2
if (a<b) then
   z = temp
else
y = 10
```

Simple Loop Optimizations: Strength Reduction

- Replacement of an operator with a less costly one

Example:

```
for i=1 to 10 do
  …
  x = i * 5
  …

end
```

→

```
temp = 5;
for i=1 to 10 do
  …
  x = temp
  …
  temp = temp + 5
end
```

- Typical cases of strength reduction occurs in address calculation of array references
- Applies to integer expressions involving induction variables (*loop optimization*)

Local Optimization

Optimization of Basic Blocks

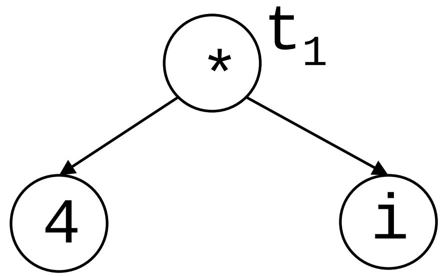
Many structure preserving transformations can be implemented by construction of DAGs of basic blocks

DAG representation of Basic Block (BB)

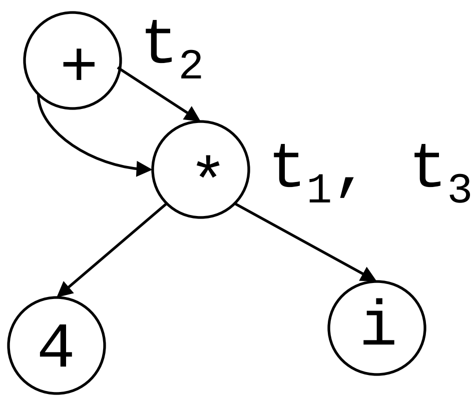
- Leaves are labeled with unique identifier (*var name or const*)
- Interior nodes are labeled by an operator symbol
- Nodes optionally have a list of labels (*identifiers*)
- Edges relate operands to the operator (*interior nodes are operator*)
- Interior node represents computed value
  - Identifier in the label are deemed to hold the value

Example: DAG for BB

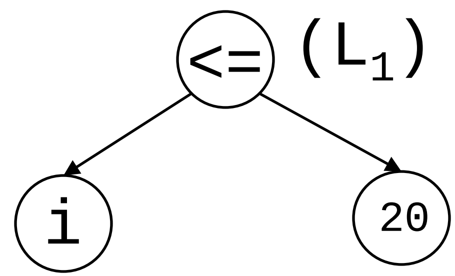
t_1 := 4 * i



t_1 := 4 * i
t_3 := 4 * i
t_2 := t_1 + t_3



if (i <= 20)goto L_1



Construction of DAGs for BB

- *I/p*: Basic block, *B*
- *O/p*: A DAG for *B* containing the following information:

1) A label for each node
2) For leaves the labels are *ids* or *consts*
3) For interior nodes the labels are *operators*
4) For each node a list of attached ids (*possible empty list*, *no consts*)

Construction of DAGs for BB

- Data structure and functions:
  - Node:
    1) Label: label of the node
    2) Left: pointer to the left child node
    3) Right: pointer to the right child node
    4) List: list of additional labels (empty for leaves)
  - **Node (*id*)**: returns the most recent node created for *id*. Else return *undef*
  - **Create(*id,l,r*)**: create a node with label *id* with *l* as left child and *r* as right child. *l* and *r* are optional params.

Construction of DAGs for BB

- Method:

  A is of the following forms:

  1. $x := y$ op z
  2. $x :=$ op y
  3. $x := y$

  1. if $((n_y = node(y)) == undef)$

     n_y = Create (y);

     if (A == type 1)

     and $((n_z = node(z)) == undef)$

     n_z = Create(z);

Construction of DAGs for BB

2. If (*A* == type 1)

   Find a node labelled '*op*' with left and right as n_y and n_z respectively [determination of common sub-expression]

   If (not found) n = Create (op, n_y, n_z);

   If (*A* == type 2)

   Find a node labelled '*op*' with a single child as n_y

   If (not found) n = Create (op, n_y);

   If (*A* == type 3) n = Node (*y*);

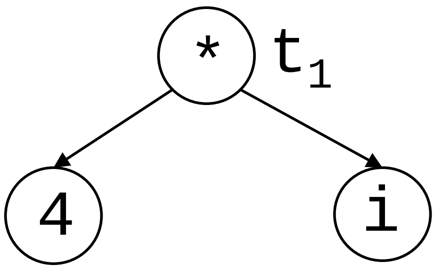
3. Remove x from Node(x).list

   Add *x* in n.list

   Node(*x*) = n;

Example: DAG construction from BB

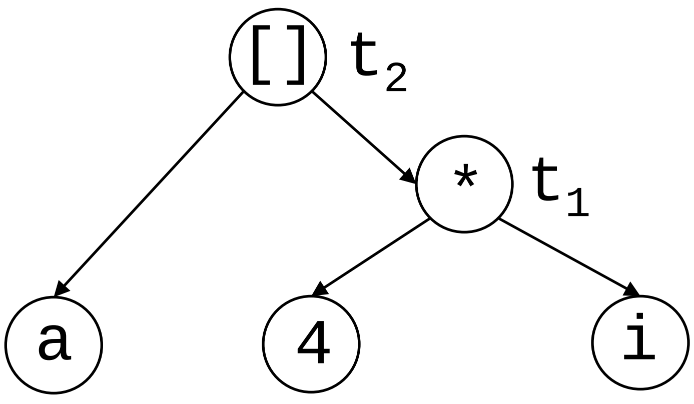
```
t₁ := 4 * i
```



Example: DAG construction from BB

t_1 := 4 * i

t_2 := a [t_1]

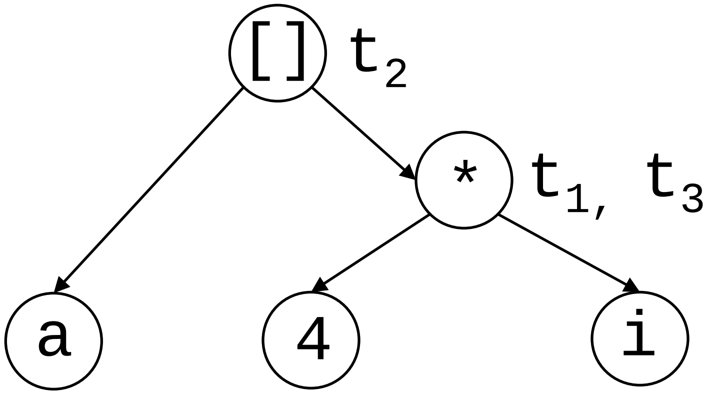


Example: DAG construction from BB

t_1 := 4 * i

t_2 := a [t_1]

t_3 := 4 * i



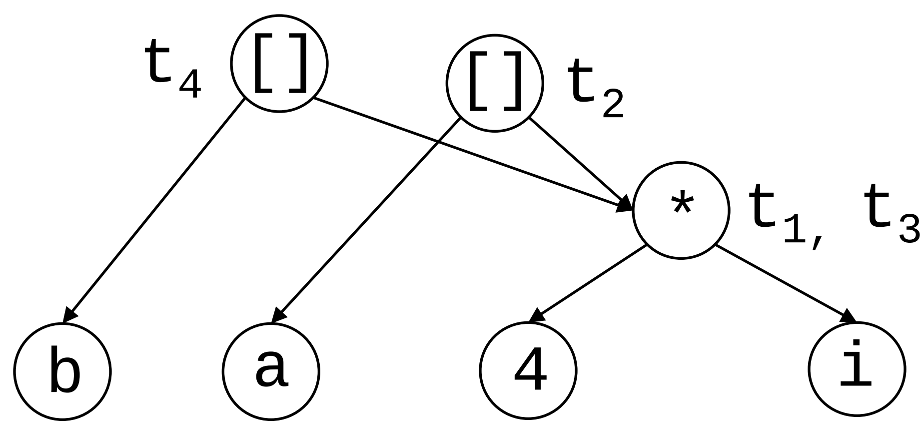
Example: DAG construction from BB

t_1 := 4 * i

t_2 := a [t_1]

t_3 := 4 * i

t_4 := b [t_3]



Example: DAG construction from BB

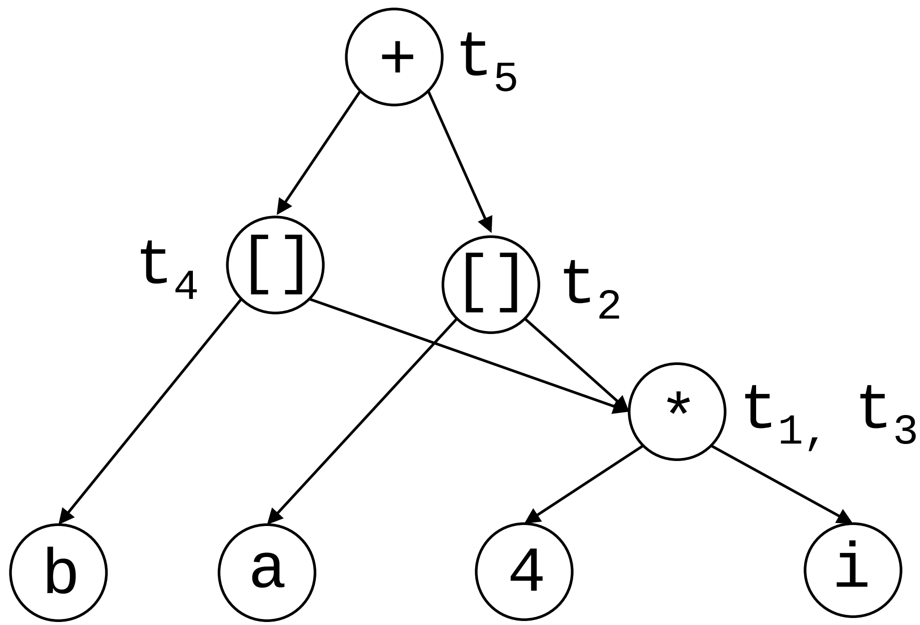
t_1 := 4 * i

t_2 := a [t_1]

t_3 := 4 * i

t_4 := b [t_3]

t_5 := t_2 + t_4



Example: DAG construction from BB

t_1 := 4 * i

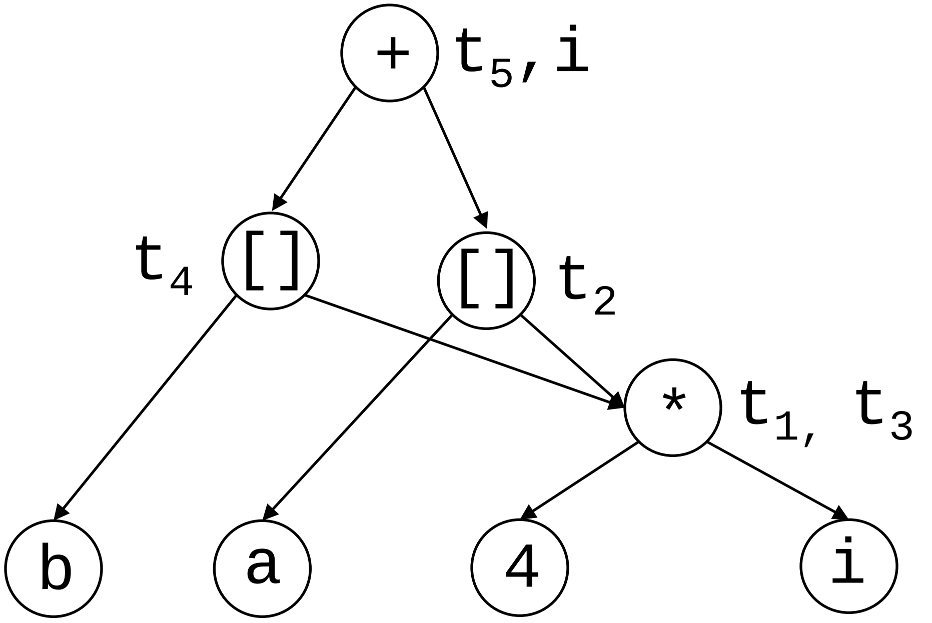
t_2 := a [t_1]

t_3 := 4 * i

t_4 := b [t_3]

t_5 := t_2 + t_4

i := t_5



DAG of a Basic Block

- Observations:
  - A leaf node for the initial value of an *id*
  - A node *n* for each statement *s*
  - Children of node *n* are the last definition (prior to *s*) of the operands of *n*

Optimization of Basic Blocks

- Common sub-expression elimination: Using DAG
  - *Note: for common sub-expression elimination, we are actually targeting for expressions that compute the same value*

```
a := b + c
b := b – d
c := c + d
e := b + c
```

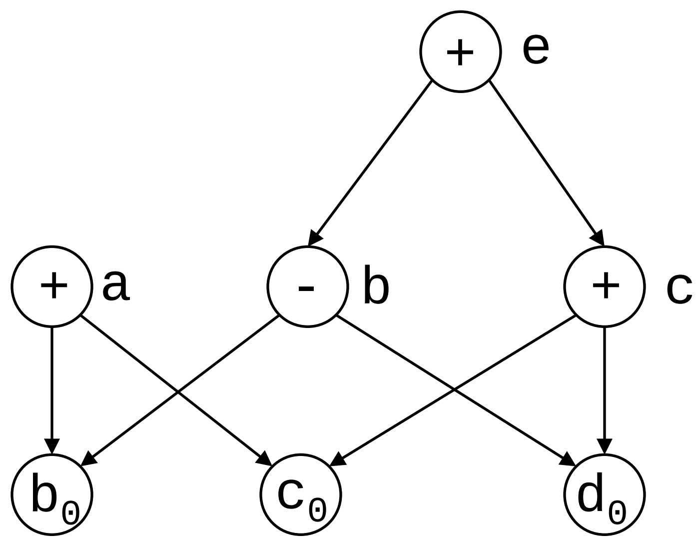
Common expressions

But do not generate the same result

Optimization of Basic Blocks

- DAG representation identifies expressions that yield the same result

```
a := b + c
b := b – d
c := c + d
e := b + c
```

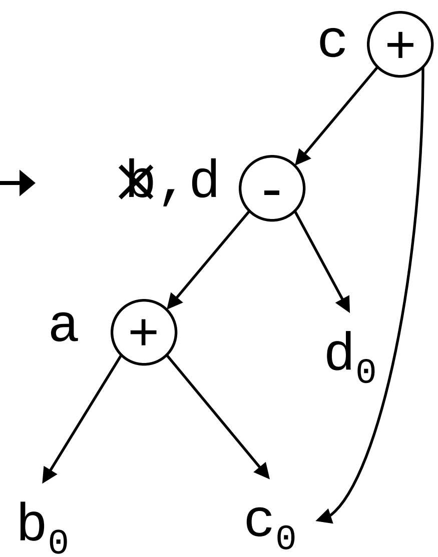


Optimization of Basic Blocks

- Dead code elimination: Code generation from DAG eliminates dead code

```
a := b + c
b := a – d
d := a – d
c := d + c
```

b is not live




```
a := b + c
d := a - d
c := d + c
```

Loop Optimization

Loop Optimizations

- Most important set of optimizations
  - Programs are likely to spend more time in loops
- Presumption: Loop has been identified
- *Optimizations*:
  - Loop invariant code removal
  - Induction variable strength reduction
  - Induction variable reduction

Loops in Flow Graph

- Dominators:

  A node *d* of a flow graph *G* dominates a node *n*, if every path in *G* from the initial node to *n* goes through *d*.

  Represented as: *d dom n*

  Corollaries:

  Every node dominates itself

  The initial node dominates all nodes in *G*

  The entry node of a loop dominates all nodes in the loop

Loops in Flow Graph

- Each node n has a unique *immediate dominator* m, which is the last dominator of n on any path in G from the initial node to n

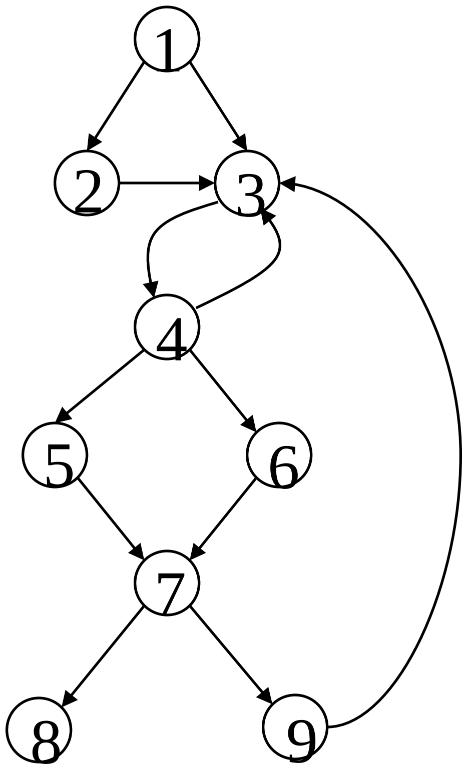
  $$(d \neq n) \ \&\& \ (d \ dom \ n) \rightarrow d \ dom \ m$$

- Dominator tree (T):

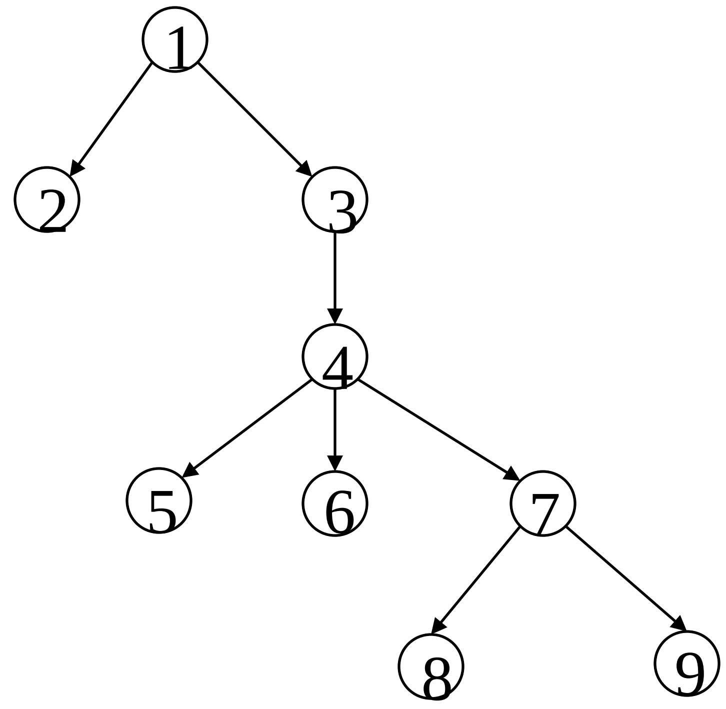
  A representation of dominator information of flow graph G

  - Root node of T is the initial node of G
  - Node d in T dominates all nodes in its sub-tree

Example: Loops in Flow Graph



Flow Graph



Dominator Tree

Loops in Flow Graph

- Natural loops:
  1. A loop has a single entry point, called the "*header*"
     Header dominates all nodes in the loop
  2. There is at least one path back to the header from the loop nodes (i.e. *there is at least one way to iterate the loop*)

- Natural loops can be detected by *back edges*
  - B*ack edges*: edges where the sink node (*head*) dominates the source node (*tail*) in *G*

Natural loop construction

- Construction of natural loop for a back edge

<u>Input:</u> A flow graph G,

A back edge $n \rightarrow d$

<u>Output:</u> The set *loop* consisting of all nodes in the natural loop of $n \rightarrow d$

<u>Method:</u>

stack := ϵ ; *loop* := $\{d\}$;
insert(n);
while (*stack* not empty)
  m := *stack*.pop();
  for each predecessor p of m do
    insert(p)

<u>Function:</u> insert (m)
if !(m ϵ *loop*)
  loop := *loop* $\cup$ $\{m\}$
  stack.push(m)

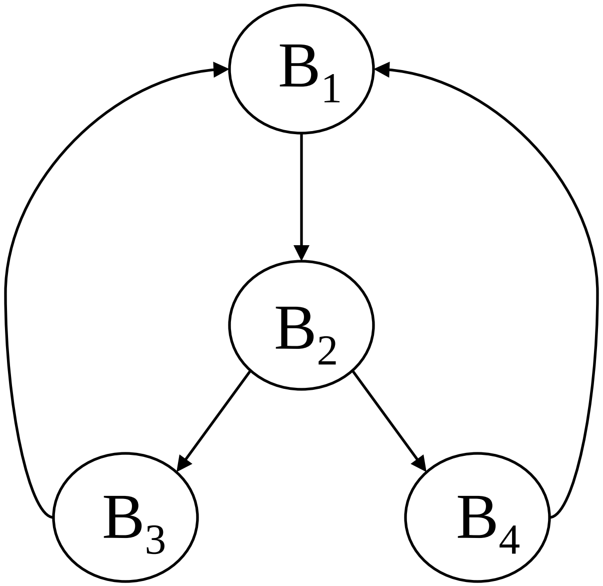
Inner loops

- Property of natural loops:
  - If two loops l_1 and l_2 do not have the same header,
    - l_1 and l_2 are disjoint
    - One is an inner loop of the other

- Inner loop: loop that contains no other loop
    - Loops which do not have the same header

Inner loops

- Loops having the same header:



Difficult to conclude which one of $\{B_1, B_2, B_3\}$ and $\{B_1, B_2, B_4\}$ is the *inner loop* without detailed analysis of code

<u>Assumption:</u>

When two loops have the same header they are treated as a single Loop

Loop Optimization

- Loop interchange: exchange inner loops with outer loops

- Loop splitting: attempts to simplify a loop or eliminate dependencies by breaking it into multiple loops which have the same bodies but iterate over different contiguous portions of the index range

  – A useful special case is ***loop peeling*** - simplify a loop with a problematic first iteration by performing that iteration separately before entering the loop

An Example: Loop Peeling

```
int p = 10;
for (int i=0; i<10; ++i)
{
y[i] = x[i] + x[p];
 p = i;
}
```

P=10 is required only for the first iteration and in all other subsequent iterations the value of p=i-1

After loop peeling:

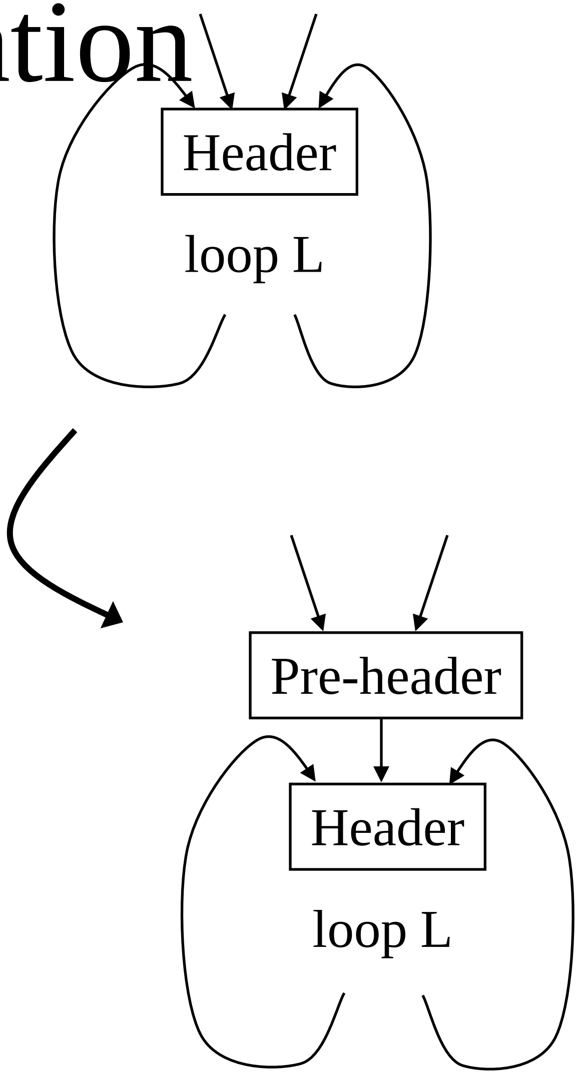
```
y[0] = x[0] + x[10];
for (int i=1; i<10; ++i)
{
y[i] = x[i] + x[i-1];
}
```

Loop Optimization

- Loop fusion: two adjacent loops would iterate the same number of times, their bodies can be combined as long as they make no reference to each other's data
- Loop fission: break a loop into multiple loops over the same index range but each taking only a part of the loop's body
- Loop unrolling: duplicates the body of the loop multiple times

Loop Optimization

- Pre-header:
  - Targeted to hold statements that are moved out of the loop
  - A basic block which has only the header as successor
  - Control flow that used to enter the loop from outside the loop, through the header, enters the loop from the pre-header



Loop Invariant Code Removal

- Move out to pre-header the statements whose *source operands do not change within the loop*
  - Be careful with the memory operations
  - Be careful with statements which are executed in some of the iterations

Loop Invariant Code Removal

- Rules: A statement S: x:=y op z is loop invariant:
  - y and z not modified in loop body
  - S is the only statement to modify x
  - For all uses of x, x is in the available def set
  - For all exit edges from the loop, S is in the available def set of the edges
  - If S is a load or store (*mem ops*), then there is no write to address (x) in the loop

Loop Induction Variable

- Induction variables: Variables such that every time they change value, they are *incremented* or *decremented*
  - Basic induction variable: induction variable whose only assignment within a loop is of the form:

    `i = i +/- C`, where C is a constant.
  - Primary induction variable: basic induction variable that controls the loop execution

    (`for i=0; i<100; i++`)

    `i` (register holding i) is the primary induction variable
  - Derived induction variable: variable that is a linear function of a basic induction variable

Loop Induction Variable

- Basic: r4, r7, r1
- Primary: r1
- Derived: r2

```
       r1 = 0
       r7 = &A
Loop:  r2 = r1 * 4
       r4 = r4 + 3
       r7 = r7 + 1
       r10 = *r2
       r3 = *r4
       r9 = r1 * r3
       r10 = r9 >> 4
       *r2 = r10
       r1 = r1 + 4
       If(r1 < 100) goto Loop
```

Global Data Flow Analysis

Global Data Flow Analysis

- Collect information about the whole program
- Distribute the information to each block in the flow graph
- *Data flow information*: Information collected by data flow analysis
- *Data flow equations*: A set of equations solved by data flow analysis to gather data flow information

Data flow analysis

- IMPORTANT!
  - *Data flow analysis should never tell us that a transformation is safe when in fact it is not*
  - When doing data flow analysis we must be
    - Conservative
      - Do not consider information that may not preserve the behavior of the program
    - Aggressive
      - Try to collect information that is as exact as possible, so we can get the greatest benefit from our optimizations

Global Iterative Data Flow Analysis

- **Global**:
  - Performed on the flow graph
  - *Goal* = to collect information at the beginning and end of each basic block
- **Iterative:**
  - Construct data flow equations that describe how information flows through each basic block
  - Solve them by iteratively converging on a solution

Global Iterative Data Flow Analysis

- Components of data flow equations
  - Sets containing collected information
    - **in** set: information coming into the BB from outside (following flow of data)
    - **gen** set: information generated/collected within the BB
    - **kill** set: information that, due to action within the BB, will affect what has been collected outside the BB
    - **out** set: information leaving the BB
  - Functions (operations on these sets)
    - **Transfer functions** describe how information changes as it flows through a basic block
    - **Meet functions** describe how information from multiple paths is combined

Global Iterative Data Flow Analysis

- Algorithmic steps
  - Store information in terms of bit vectors
    - For example, in reaching definitions, each bit position corresponds to one definition
  - Use an iterative fixed-point algorithm
  - Depending on the nature of the problem, traverse each basic block in a forward (*top-down*) or backward direction
    - Order of BB visits is not important in terms of algorithm correctness
    - But important in terms of efficiency
  - *In* & *Out* sets should be initialized in a conservative and aggressive way

Global Iterative Data Flow Analysis

```
Initialize gen and kill sets
Initialize in or out sets (depending on "direction")
while there are no changes in in and out sets {
    for each BB {
        apply meet function
        apply transfer function
    }
}
```

Typical problems

- Reaching definitions
  - For each use of a variable, find all definitions that reach it
- Upward exposed uses
  - For each definition of a variable, find all uses that it reaches
- Live variables
  - For a point p and a variable v, determine whether v is live at p
- Available expressions
  - Find all expressions whose value is available at some point p

Global Data Flow Analysis

- A typical data flow equation:

$$out[S] = gen[S] \bigcup (in[S] - kill[S])$$

S: statement

in[S]: Information goes into S

kill[S]: Information get killed by S

gen[S]: New information generated by S

out[S]: Information goes out from S

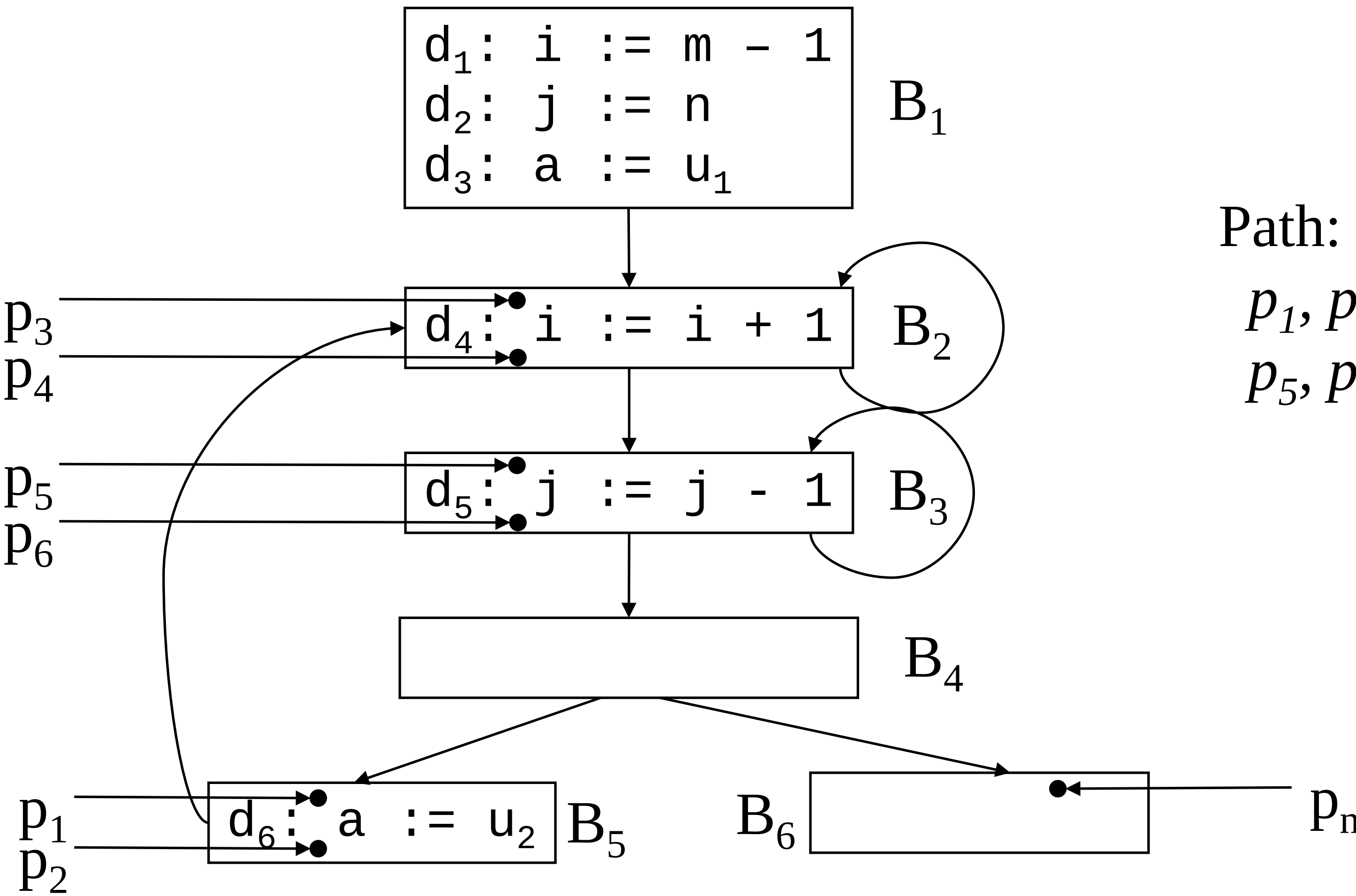
Global Data Flow Analysis

- Notions of *gen* and *kill* depends on the desired information
- In some cases, *in* may be defined in terms of *out* – equation
  - Solved as analysis traverses in the *backward direction*
- Data flow analysis follows control flow graph
  - Equations are set at the level of basic blocks, or even for a statement

Points and Paths

- *Point* within a basic block:
  - A location between two consecutive statements
  - A location before the first statement of the basic block
  - A location after the last statement of the basic block
- *Path*: A path from a point p_1 to p_n is a sequence of points $p_1, p_2, \ldots p_n$ such that for each $i : 1 \leq i \leq n$,
  - p_i is a point immediately preceding a statement and p_{i+1} is the point immediately following that statement in the same block, <u>or</u>
  - p_i is the last point of some block and p_{i+1} is first point in the successor block.

Example: Paths and Points



Path:

$p_1, p_2, p_3, p_4,$
$p_5, p_6 \ldots p_n$

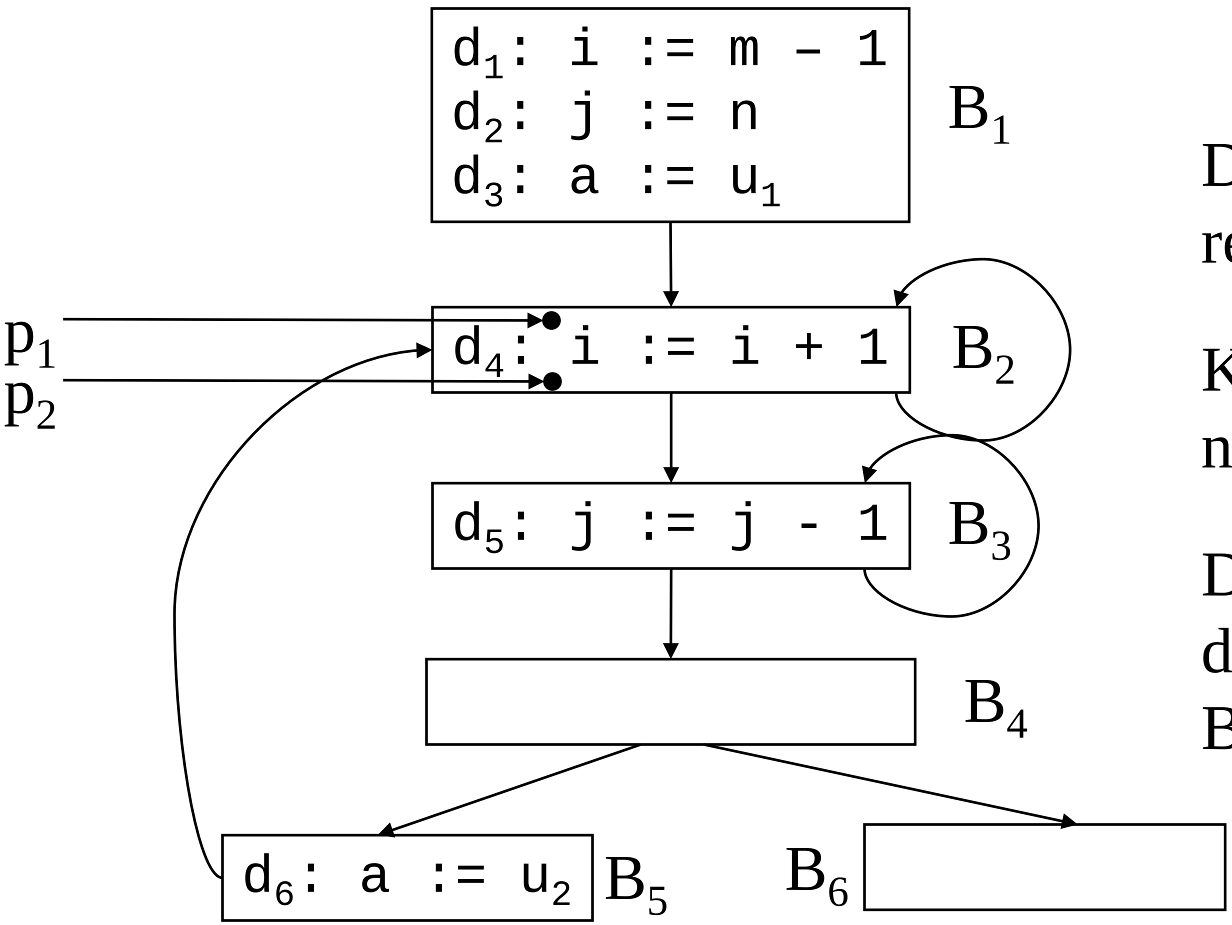
Reaching Definition

- Definition of a variable *x* is a statement that assigns or *may* assign a value to *x*
  - *Unambiguous Definition*: The statements that certainly assigns a value to *x*
    - Assignments to *x*
    - Read a value from I/O device to *x*
  - *Ambiguous Definition*: Statements that may assign a value to *x*
    - Call to a procedure with *x* as parameter (call by ref)
    - Call to a procedure which can access *x* (*x* being in the scope of the procedure)
    - *x* is an alias for some other variable (*aliasing*)
    - Assignment through a pointer that could refer *x*

Reaching Definition

- A definition *d reaches* a point *p*
  - if there is a path from the point immediately following *d* to *p* and
  - *d* is not *killed* along the path (*i.e.* there is no redefinition of the same variable in the path)
- A definition of a variable is *killed* between two points when there is another definition of that variable along the path

Example: Reaching Definition



Definition of i (d_1) reaches p_1

Killed, as d_1 does not reach p_2.

Definition of i (d_1) does not reach B_3, B_4, B_5 and B_6

Reaching Definition

- Non-Conservative view: A definition *might* reach a point even if it might not
  - Only unambiguous definition kills a earlier definition
  - All edges of flow graph are assumed to be traversed

```
if (a == b) then a = 2
 else if (a == b) then a = 4
```

Definition “a=4” is not reachable

Whether each path in a flow graph is taken is an undecidable problem

Data Flow analysis of a Structured Program

- Structured programs have well defined loop constructs – the resultant flow graph is always reducible
  - Without loss of generality consider *while-do* and *if-then-else* control constructs

$$S \rightarrow \textbf{id} \mathbin{\textbf{:=}} E \mid S \,\textbf{;}\, S$$
$$\mid \textbf{if}\ E\ \textbf{then}\ S\ \textbf{else}\ S \mid \textbf{do}\ S\ \textbf{while}\ E$$
$$E \rightarrow \textbf{id} + \textbf{id} \mid \textbf{id}$$

Non-terminals represent *regions*

Data Flow analysis of a Structured Program

- Region: A graph $G' = (N', E')$ which is portion of the control flow graph G
  - Set of nodes N' is in G' such that
    - N' includes a header h
    - h dominates all nodes in N'
  - Set of edges E' is in G' such that
    - All edges $a \rightarrow b$ such that a, b are in N'

Data Flow analysis of a Structured Program

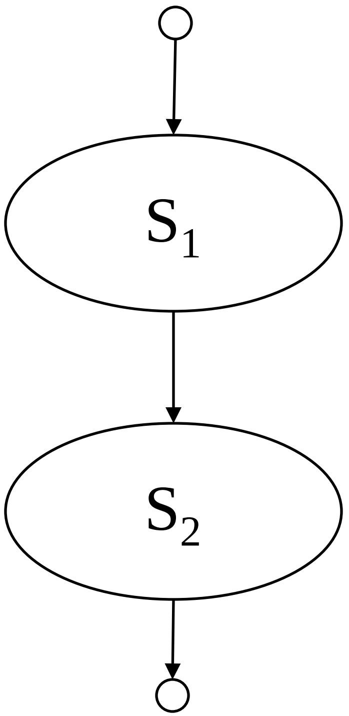
- Region consisting of a statement S:
  - Control can flow to only one block outside the region

- Loop is a special case of a region that is strongly connected and includes all its back edges

- Dummy blocks with no statements are used as technical convenience (indicated as open circles)

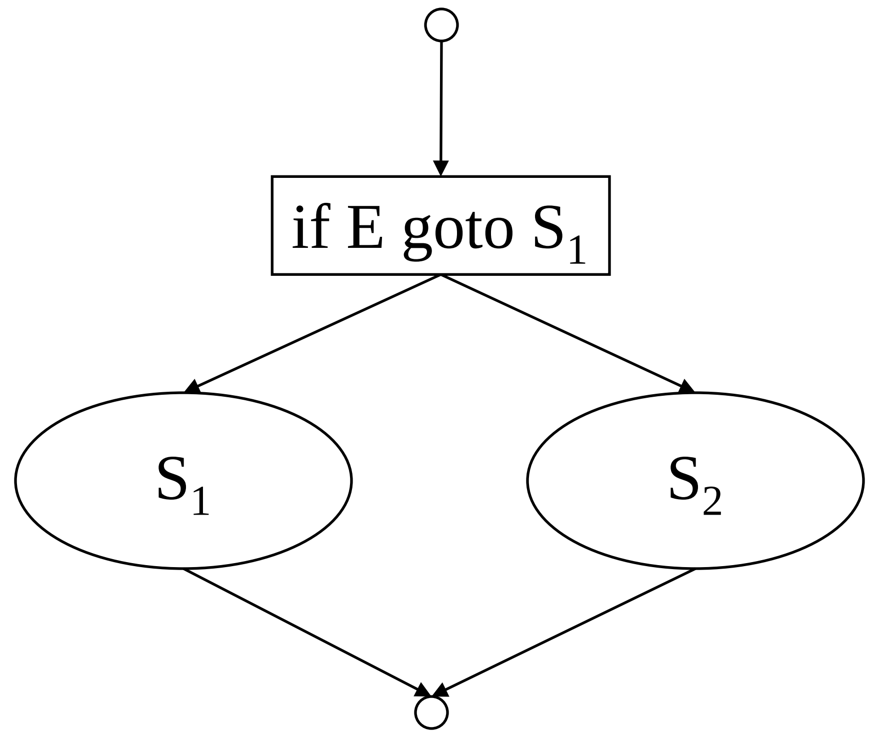
Data Flow analysis of a Structured Program: Composition of Regions

$S \rightarrow S_1 \; ; \; S_2$



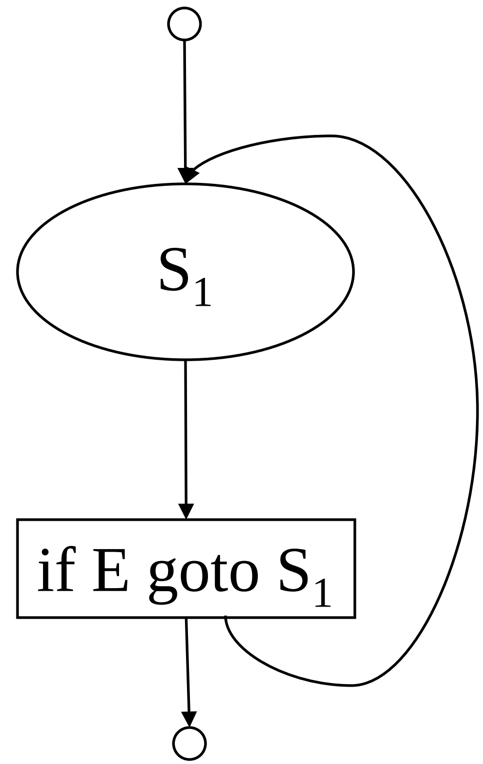
Data Flow analysis of a Structured Program:
Composition of Regions

$S \rightarrow$ **if** E **then** S_1 **else** S_2



Data Flow analysis of a Structured Program: Composition of Regions

$S \rightarrow$ **do** S_1 **while** E



Data Flow Equations

- Each region has four attributes:
  - *gen*[S]: Set of definitions generated by block S

    If a definition *d* is in *gen*[S], then *d* reaches the end of block S
  - *kill*[S]: Set of definitions killed by block S.

    If *d* is in *kill*[S], *d* never reaches the end of block S. Every path from the beginning of S to the end of S must have a definition for a (where *a* is defined by *d)*

Data Flow Equations

- *in*[S]: Set of definition those are live at the entry point of block S
- *out*[S]: Set of definition those are live at the exit point of block S

• Data flow equations are inductive or syntax directed

- *gen* and *kill:* synthesized attributes
- *in* : inherited attribute

Data Flow Equations

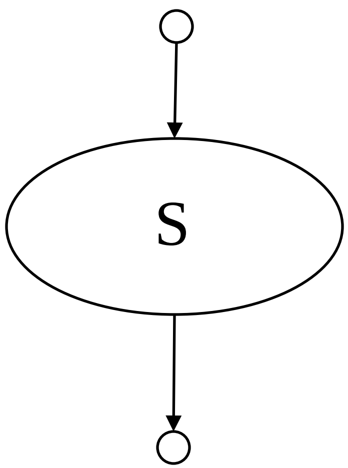
- *gen*[S]:
  - concerns with a single basic block
  - set of definitions in S that reaches the end of S
- *out*[S]:
  - Set of definitions (*possibly defined in some other block*)
  - Live at the end of S considering all paths through S

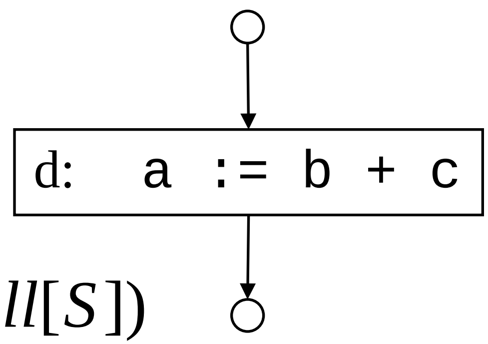
Data Flow Equations

Single statement

$$gen[S] = \{d\}$$

$$kill[S] = D_a - \{d\}$$





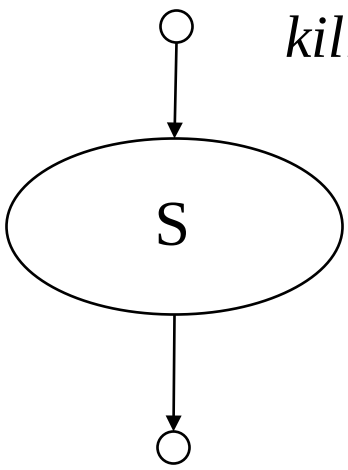
$$out[S] = gen[S] \bigcup (in[S] - kill[S])$$

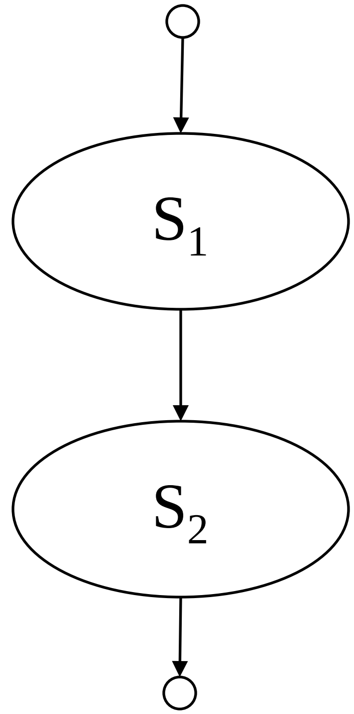
D_a: The set of definitions in the program for variable a

Data Flow Equations
Composition

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

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





$$in[S_1] = in[S]$$

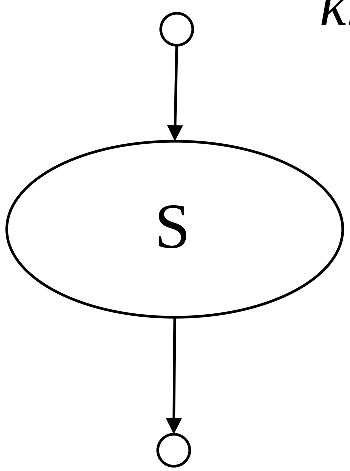
$$in[S_2] = out[S_1]$$

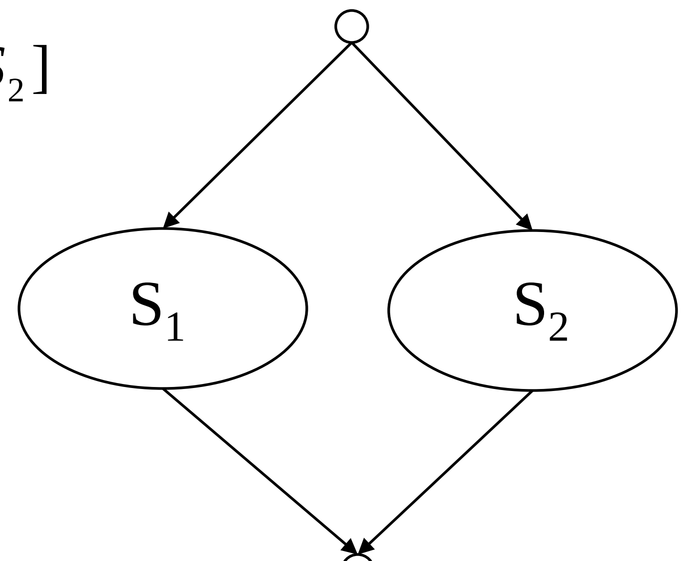
$$out[S] = out[S_2]$$

Data Flow Equations
if-then-else

$$gen[S] = gen[S_1] \bigcup gen[S_2]$$

$$kill[S] = kill[S_1] \bigcap kill[S_2]$$





$$in[S_1] = in[S]$$

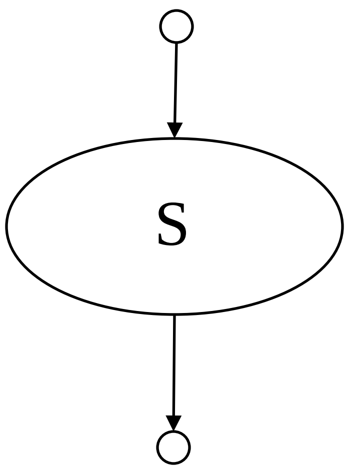
$$in[S_2] = in[S]$$

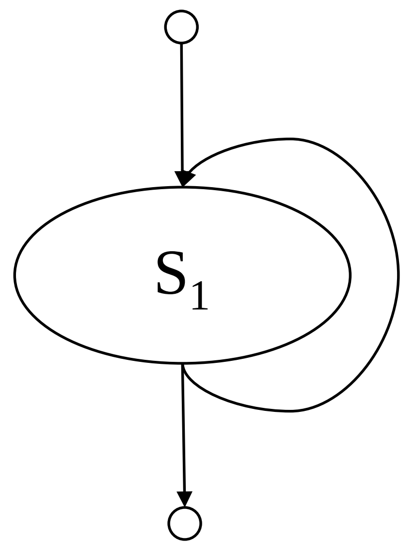
$$out[S] = out[S_1] \bigcup out[S_2]$$

Data Flow Equations
Loop

$$gen[S] = gen[S_1]$$

$$kill[S] = kill[S_1]$$





$$in[S_1] = in[S] \bigcup gen[S_1]$$

$$out[S] = out[S_1]$$

Data Flow Analysis

- For each region, compute attributes
- Equations can be solved in two phases:
  - *gen* and *kill* can be computed in a single pass of a basic block
  - *in* and *out* are computed iteratively
    - Initial condition for *in* for the whole program is $\emptyset$
      - can be computed top- down
    - Finally, *out* is computed

Dealing with loop

- Due to back edge, *in*[S] cannot be used as *in* [S_1]
- *in*[S_1] and *out*[S_1] are interdependent
- Equation is solved iteratively
- General equations for *in* and *out*:

$$in[S] = \bigcup (out[Y] : \text{Y is a predecessor of S})$$

$$out[S] = gen[S] \bigcup (in[S] - kill[S])$$

Computation of *gen* and *kill* sets

for each basic block BB do

  gen(BB) = $\emptyset$; *kill*(BB) = $\emptyset$;

  for each statement (d: x := y op z) in sequential order in BB, do

    kill(BB) = *kill*(BB) $\cup$ G[x];

    G[x] = d;

  endfor

  gen(BB) = $\cup$ G[x]: for all id x

endfor

Computation of *in* and *out* sets

for all basic blocks BB $\quad$ *in*(BB) = $\varnothing$

for all basic blocks BB $\quad$ *out*(BB) = gen(BB)

change = true

while (change) do

  change = false

  for each basic block BB, do

    old_out = *out*(BB)

    in(BB) = $\bigcup$(*out*(Y)) for all predecessors Y of BB

    out(BB) = *gen*(BB) + (*in*(BB) – *kill*(BB))

    if (*old_out* != *out*(BB)) then change = true

  endfor

endwhile

Live Variable (Liveness) Analysis

- *Liveness*: For each point p in a program and each variable *y*, determine whether *y* can be used before being redefined, starting at p

- Attributes
  - *use* = set of variable used in the BB prior to its definition
  - *def* = set of variables defined in BB prior to any use of the variable
  - *in* = set of variables that are live at the entry point of a BB
  - *out* = set of variables that are live at the exit point of a BB

Live Variable (Liveness) Analysis

- Data flow equations:

$$in[B] = use[B] \bigcup (out[B] - def[B])$$

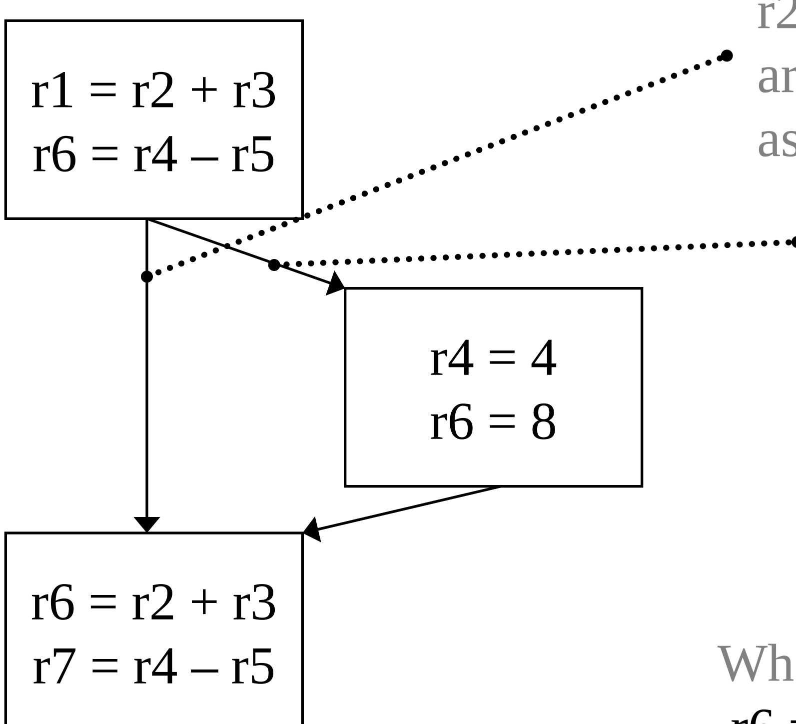
$$out[B] = \bigcup_{S = succ(B)} in[S]$$

  - 1st Equation: a *var* is live, *coming in* the block, if either
    - it is used before redefinition in B

    or

    - it is live coming out of B and is not redefined in B
  - 2nd Equation: a *var* is live *coming out* of B, iff it is live coming in to one of its successors

Example: Liveness



r2, r3, r4, r5 are all live as they are consumed later, r6 is dead as it is redefined later

r4 is dead, as it is redefined. So is r6. r2, r3, r5 are live

What does this mean?

r6 = r4 – r5 is useless,

it produces a dead value !!

Get rid of it!

Computation of *use* and *def* sets

```
for each basic block BB do
    def(BB) = ∅;    use(BB) = ∅ ;
    for each statement (x := y op z) in sequential order, do
        for each operand y, do
            if (y not in def(BB))
                use(BB) = use(BB) ∪ {y};
        endfor
        def(BB) = def(BB) ∪ {x};
endfor
```

def is the union of all the LHS's
use is all the ids used before defined

Computation of *in* and *out* sets

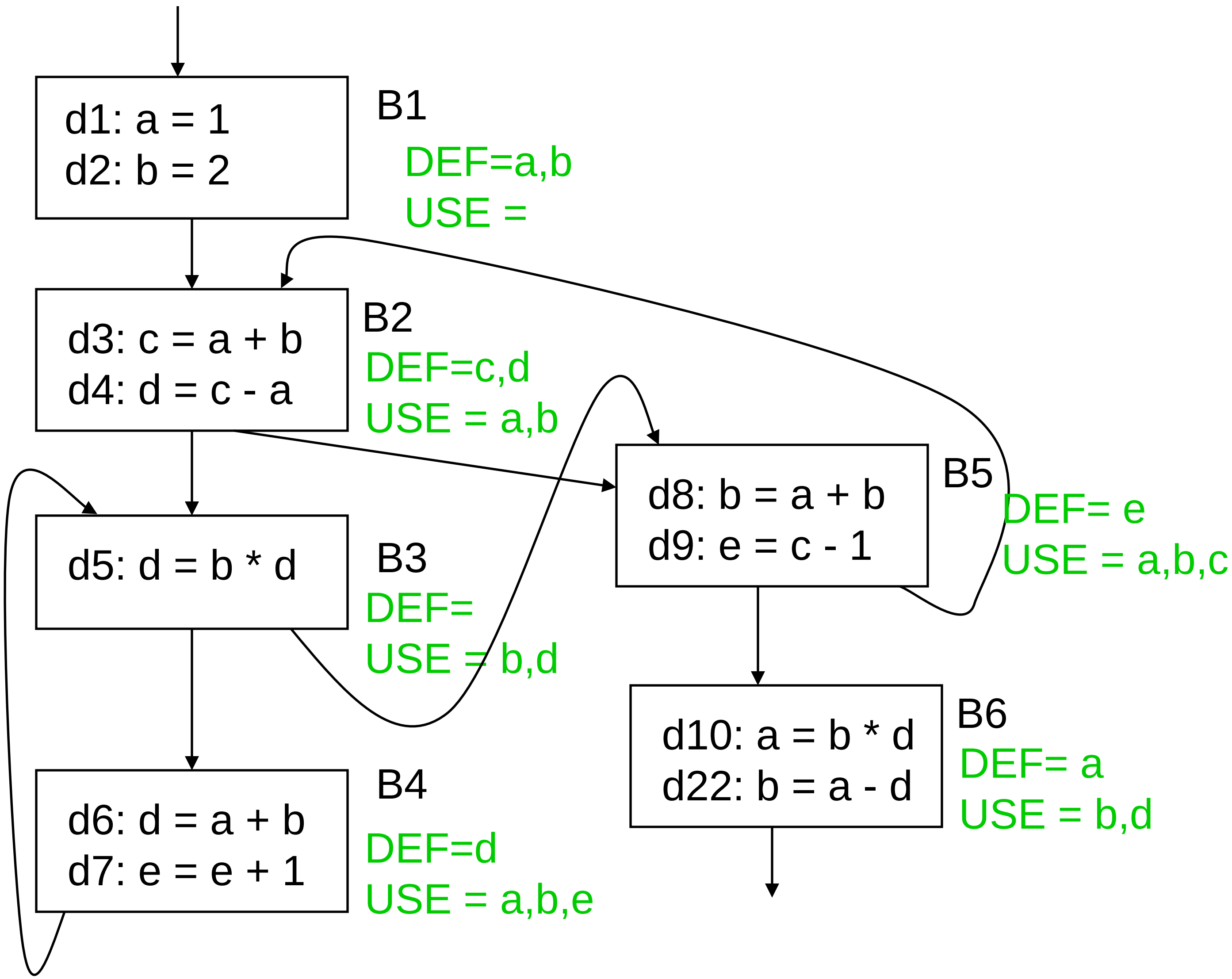
```
for all basic blocks BB
    in(BB) = ∅ ;

change = true;
while (change) do
   change = false
   for each basic block BB do
      old_in = in(BB);
      out(BB) = ∪{in(Y): for all successors Y of BB}
      in(BB) = use(BB) ∪ (out(BB) – def(BB))
      if (old_in != in(BB)) then change = true
   endfor
endwhile
```

Global Live Variable Analysis

Want to determine for some variable *x* and point *p* whether the value of *x* could be used along some path starting at *p*.

- DEF[B] - set of variables assigned values in B prior to any use of that variable
- USE[B] - set of variables used in B prior to any definition of that variable
- OUT[B] - variables live immediately after the block OUT[B] - $\cup$IN[S] for all S in succ(B)
- IN[B] - variables live immediately before the block

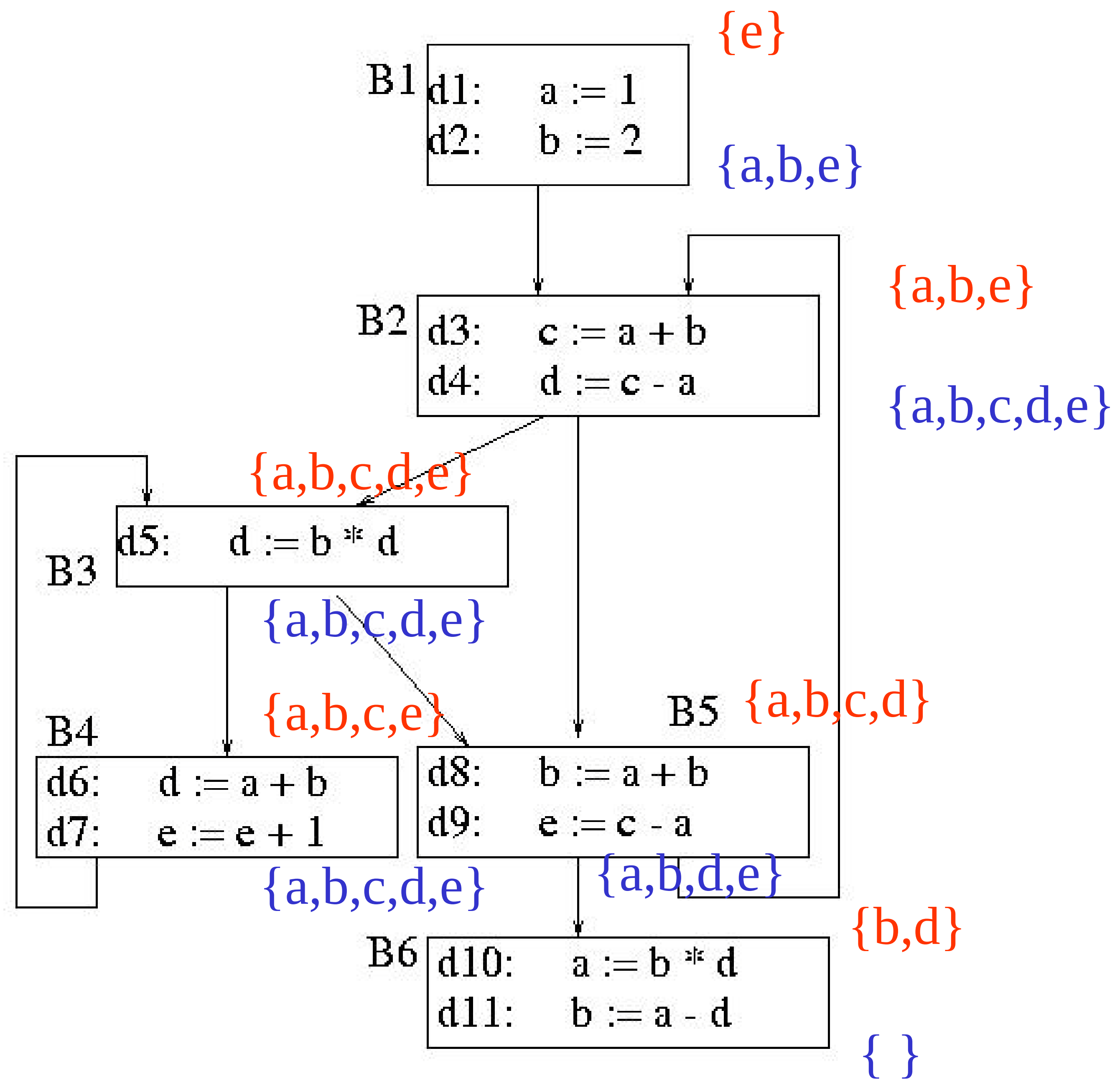
  IN[B] = USE[B] + (OUT[B] - DEF[B])


d1: a = 1
d2: b = 2
B1
DEF=a,b
USE =
d3: c = a + b
d4: d = c - a
B2
DEF=c,d
USE = a,b
d5: d = b * d
B3
DEF=
USE = b,d
d6: d = a + b
d7: e = e + 1
B4
DEF=d
USE = a,b,e
d8: b = a + b
d9: e = c - 1
B5
DEF= e
USE = a,b,c
d10: a = b * d
d22: b = a - d
B6
DEF= a
USE = b,d

	IN	OUT	IN	OUT	IN	OUT
B1	∅	a,b	∅	a,b	e	a,b,e
B2	a,b	a,b,c,d	a,b,e	a,b,c,d ,e	a,b,e	a,b,c,d,e
B3	a,b,c,d e	a,b,c,e	a,b,c,d,e	a,b,c,d,e	a,b,c,d,e	a,b,c,d,e
B4	a,b,c,e	a,b,c,d,e	a,b,c,e	a,b,c,d,e	a,b,c,e	a,b,c,d,e
B5	a,b,c,d	a,b,d	a,b,c,d	a,b,d,e	a,b,c,d	a,b,d,e
B6	b,d	∅	b,d	∅	b,d	∅

Block	DEF	USE
B1	{a,b}	{ }
B2	{c,d}	{a,b}
B3	{ }	{b,d}
B4	{d}	{a,b,e}
B5	{e}	{a,b,c}
B6	{a}	{b,d}

OUT[B] = ∪ IN[S] for all S in succ(B)
IN[B] = USE[B] + (OUT[B] - DEF[B])

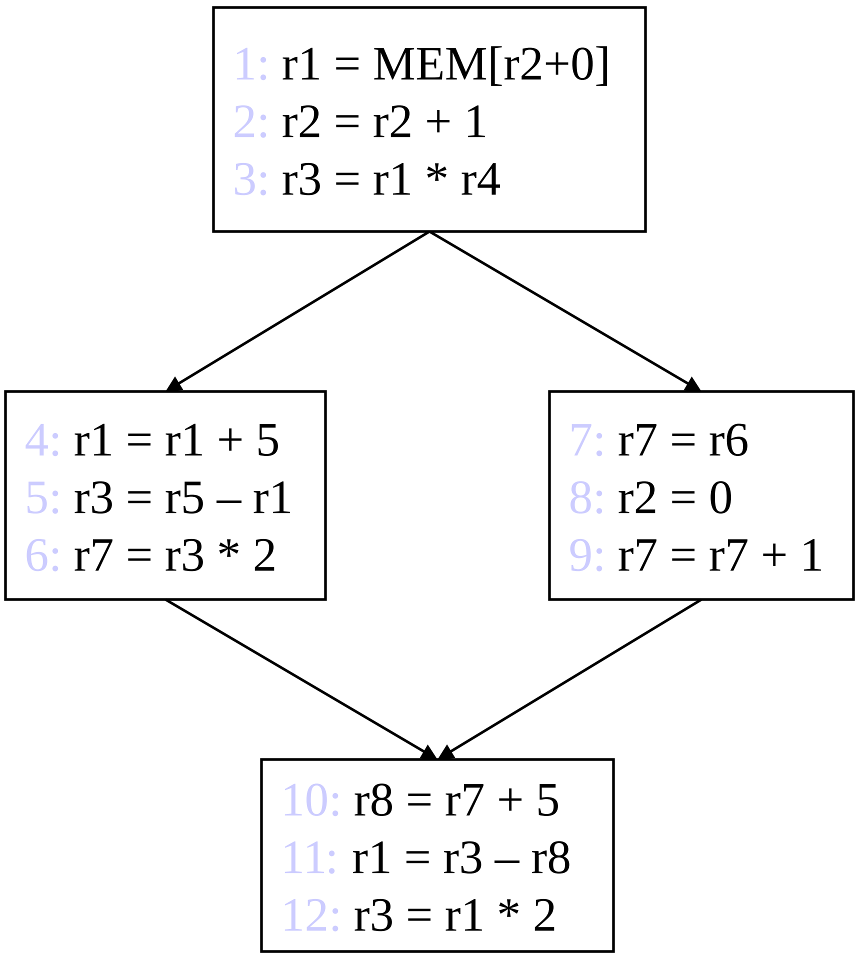


{e}
B1
d1: a := 1
d2: b := 2
{a,b,e}
{a,b,e}
B2
d3: c := a + b
d4: d := c - a
{a,b,c,d,e}
{a,b,c,d,e}
d5: d := b * d
B3
{a,b,c,d,e}
{a,b,c,e}
B4
d6: d := a + b
d7: e := e + 1
{a,b,c,d,e}
B5
{a,b,c,d}
d8: b := a + b
d9: e := c - a
{a,b,d,e}
{b,d}
B6
d10: a := b * d
d11: b := a - d
{ }

DU/UD Chains

- Convenient way to access/use reaching definition information
- Def-Use chains (DU chains)
  - Given a **def**, what are all the possible consumers of the definition produced?
- Use-Def chains (UD chains)
  - Given a **use**, what are all the possible producers of the definition consumed?

Example: DU/UD Chains



DU Chain of r1:

(1) -> 3,4

(4) ->5

(11)→12

DU Chain of r3:

(3) -> 11

(5) ->6, 11

(12) ->

UD Chain of r3:

(11) -> 5,3

UD Chain of r7:

(10) -> 6,9

Reachability Analysis: Unstructured Input

1. Compute GEN and KILL at block—level
2. Compute IN[B] and OUT[B] for B

   IN[B] = U OUT[P] *where P is a predecessor of B*

   OUT[B] = GEN[B] U (IN[B] - KILL[B])
3. Repeat step 2 until there are no changes to OUT sets

Reachability Analysis: Step 1

For each block, compute local (block level) information = GEN/KILL sets

- GEN[B] = set of definitions generated by B
- KILL[B] = set of definitions that can not reach the end of B

This information does not take control flow between blocks into account

Reasoning about Basic Blocks

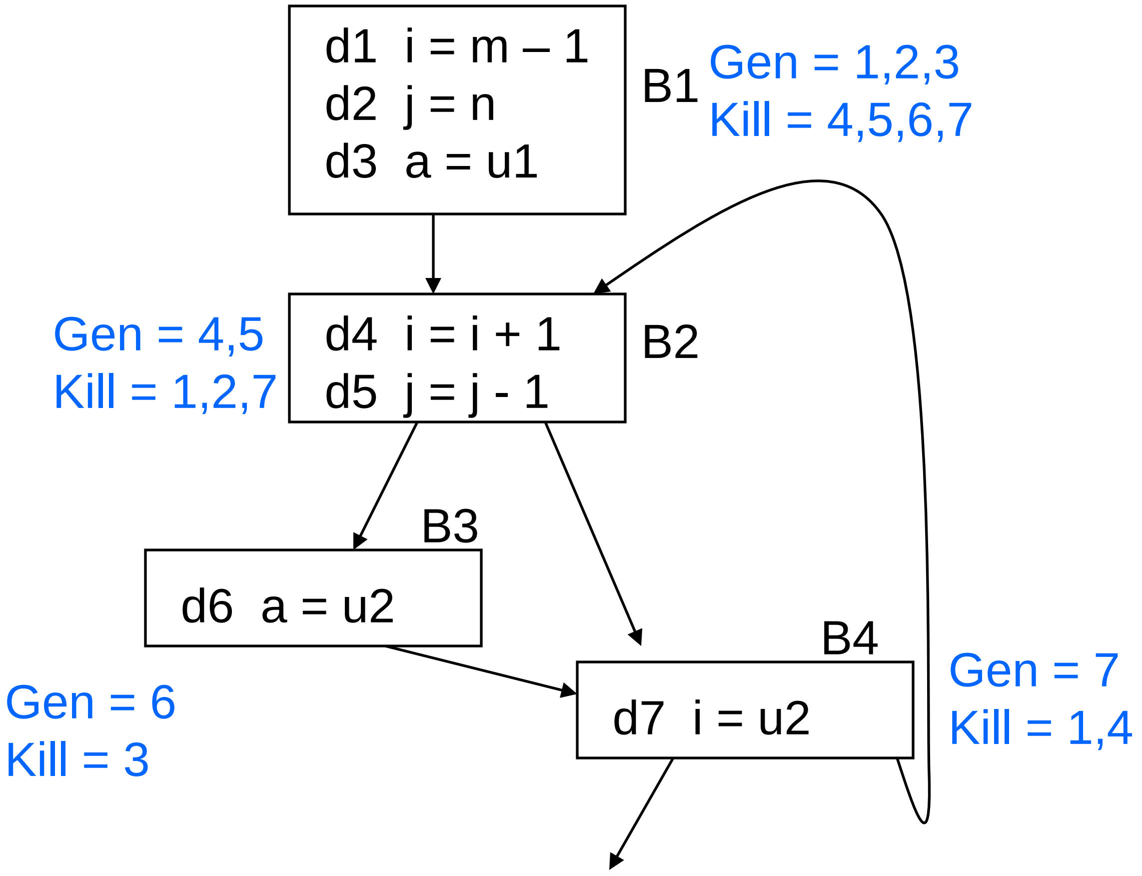
Effect of single statement: a = b + c

- Uses variables {b,c}
- **Kills all definitions of {a}**
- **Generates new definition (i.e. assigns a value) of {a}**

Local Analysis:

- Analyze the effect of each instruction
- Compose these effects to derive information about the entire block

Example


d1 i = m – 1
d2 j = n
d3 a = u1
B1
Gen = 1,2,3
Kill = 4,5,6,7
Gen = 4,5
Kill = 1,2,7
d4 i = i + 1
d5 j = j - 1
B2
B3
d6 a = u2
B4
Gen = 7
Kill = 1,4
d7 i = u2
Gen = 6
Kill = 3

Reachability Analysis: Step 2

Compute IN/OUT for each block in a *forward direction*.

Start with IN[B] = $\emptyset$

- IN[B] = *set of defns reaching the start of B*

  = $\cup$ (out[P]) for all predecessor blocks in the CFG

- OUT[B] = *set of defns reaching the end of B*

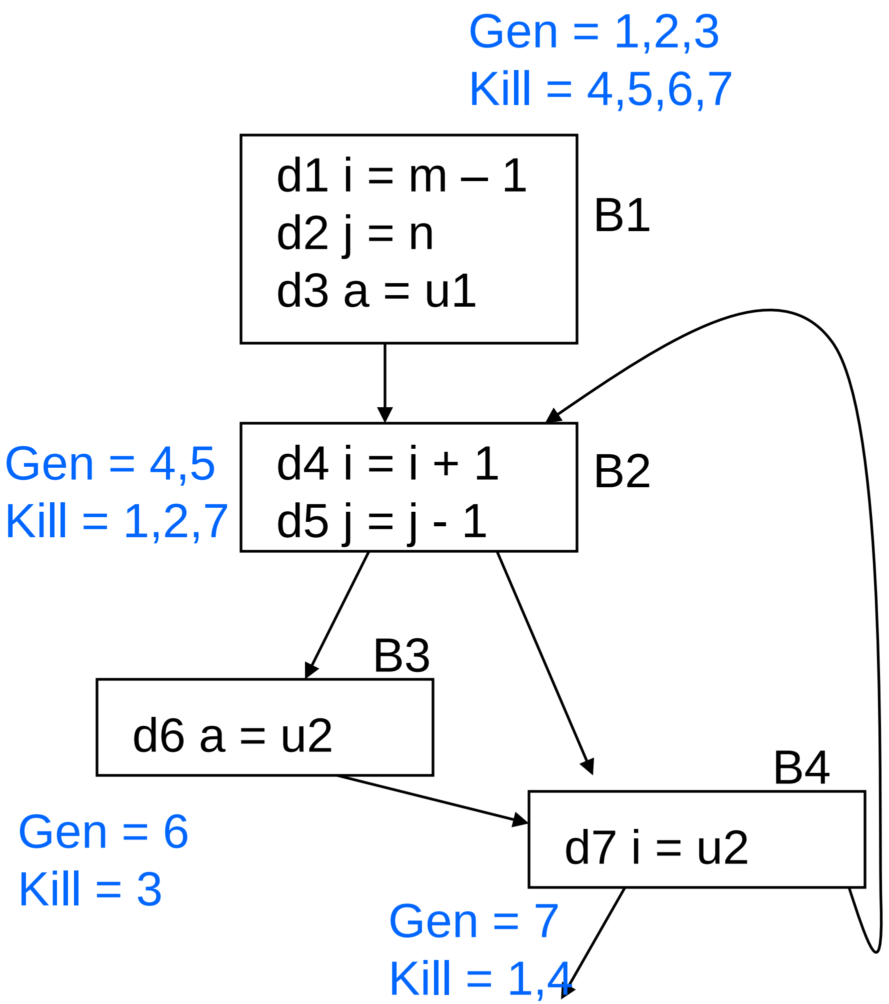
  = GEN[B] $\cup$ (IN[B] – KILL[B])

Keep computing IN/OUT sets until a fixed point is reached.

Reaching Definitions Algorithm

- Input: Flow graph with GEN and KILL for each block
- Output: in[B] and out[B] for each block.

```
For each block B do out[B] = gen[B], (true if in[B] = emptyset)
change := true;
while change do begin
        change := false;
        for each block B do begin
          in[B] := U out[P], where P is a predecessor of B;
          oldout = out[B];
          out[B] := gen[B] U  (in[B] - kill [B])
          if out[B] != oldout then change := true;
    end
end
```



	IN	OUT		
B1	∅	1,2,3		
B2	∅	4,5		
B3	∅	6		
B4	∅	7		

IN[B] = $\cup$(out[P]) for all predecessor blocks in the CFG

OUT[B] = GEN[B] $\cup$ (IN[B] – KILL[B])

	IN	OUT	IN	OUT		
B1	∅	1,2,3	∅	1,2,3		
B2	∅	4,5	OUT[1]+OUT[4] = 1,2,3,7	4,5 + (1,2,3,7 – 1,2,7) = 3,4,5		
B3	∅	6	OUT[2] = 3,4,5	6 + (3,4,5 – 3) = 4,5,6		
B4	∅	7	OUT[2]+OUT[3] = 3,4,5,6	7 + (3,4,5,6 – 1,4) = 3,5,6,7		

IN[B] = ∪(out[P]) for all predecessor blocks in the CFG
OUT[B] = GEN[B] + (IN[B] – KILL[B])

	IN	OUT	IN	OUT	IN	OUT
B1	∅	1,2,3	∅	1,2,3	∅	1,2,3
B2	∅	4,5	1,2,3,7	3,4,5	OUT[1] + OUT[4] = 1,2,3,5,6,7	4,5 + (1,2,3,5,6,7-1,2,7) = 3,4,5,6
B3	∅	6	3,4,5	4,5,6	OUT[2] = 3,4,5,6	6 + (3,4,5,6 – 3) = 4,5,6
B4	∅	7	3,4,5,6	3,5,6,7	OUT[2] + OUT[3] = 3,4,5,6	7+(3,4,5,6 – 1,4) = 3,5,6,7

IN[B] = ∪(out[P]) for all predecessor blocks in the CFG

OUT[B] = GEN[B] + (IN[B] – KILL[B])

Forward vs. Backward

Forward flow vs. Backward flow

Forward: Compute OUT for given IN, GEN, KILL

- Information propagates from the predecessors of a vertex
- Examples: Reachability, available expressions, constant propagation

Backward: Compute IN for given OUT, GEN, KILL

- Information propagates from the successors of a vertex
- Example: Live variable Analysis

Generalizing Dataflow Analysis

- Transfer function
  - How information is changed by BB

    $out[BB] = gen[BB] + (in[BB] - kill[BB])$ forward analysis

    $in[BB] = gen[BB] + (out[BB] - kill[BB])$ backward analysis
- Meet/Confluence function
  - How information from multiple paths is combined

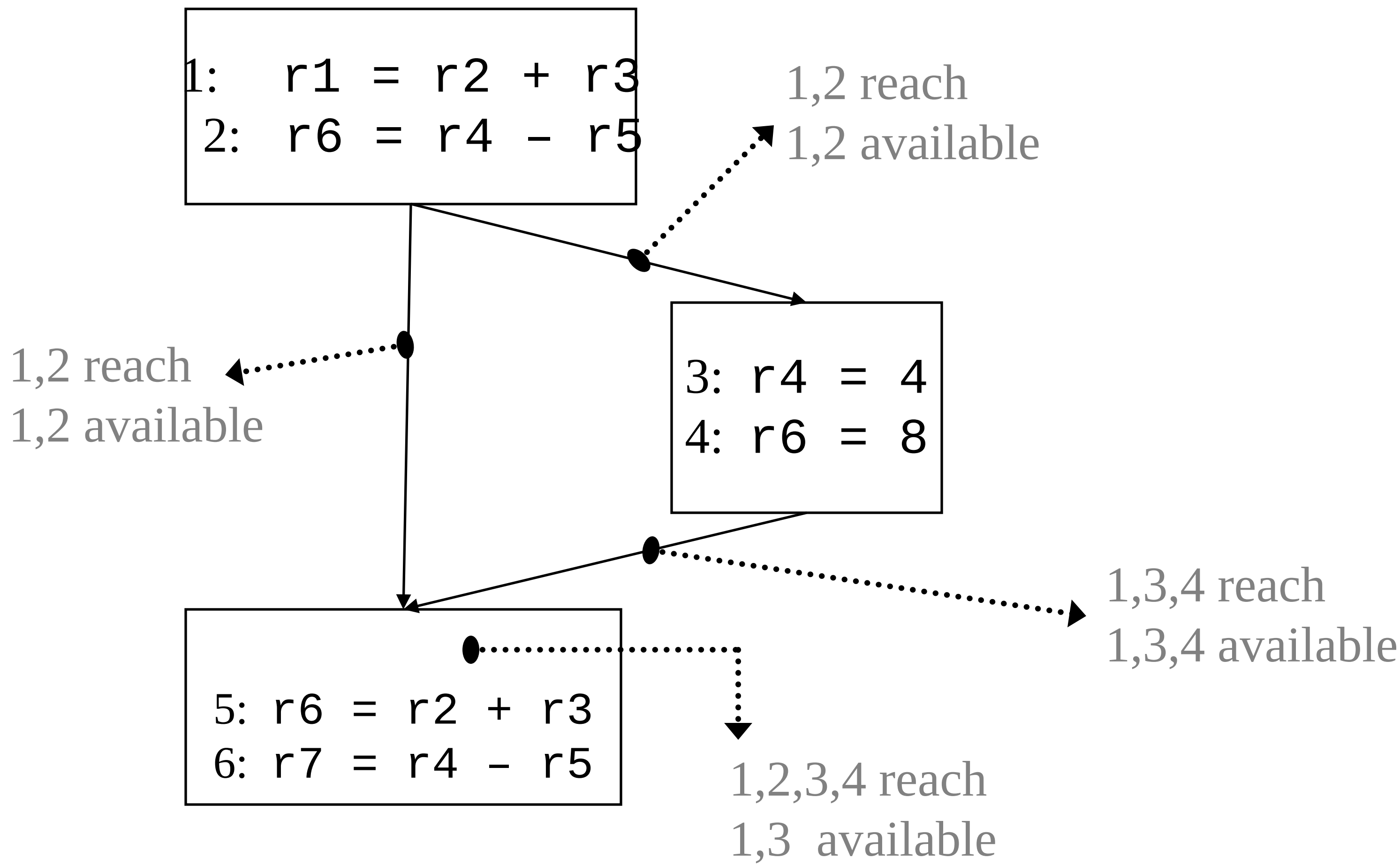
    $in[BB] = \cup\ out[P]$: P is pred of BB forward analysis

    $out[BB] = \cup\ in[P]$: P is succ of BB backward analysis

All Path Problem

- Up to this point
  - Any path problems *(may relations*)
    - Definition reaches along some path(s)
    - Some sequence of branches in which *def* reaches
    - Lots of *defs* of the same variable may reach a point
  - Use of Union operator in meet function
- All-path: Definition guaranteed to reach
  - Regardless of sequence of branches taken, *def* reaches
  - Only 1 def can be guaranteed to reach
  - Availability (as opposed to reaching)
    - Available definitions
    - Available expressions (*could also have reaching expressions, but not that useful*)

Reaching vs Available Definitions



Available Definition Analysis (Adefs)

- A definition d is *available* at a point p if along all paths from d to p, d is not killed
- Remember, a definition of a variable is *killed* between 2 points when there is another definition of that variable along the path
  - r1 = r2 + r3 kills previous definitions of r1
- Algorithm:
  - *Forward dataflow analysis* as propagation occurs from defs downwards
  - Use the *Intersect function* as the *meet operator* to guarantee the all-path requirement
  - *gen/kill/in/out* similar to reaching defs
    - Initialization of *in/out:* tricky part

Compute Adef *gen/kill* Sets

for each basic block BB do
  gen(BB) = $\emptyset$; *kill*(BB) = $\emptyset$;
  for each statement (d: x := y op z) in sequential order in BB, do
    kill(BB) = *kill*(BB) $\cup$ G[x];
    G[x] = d;
  endfor
  gen(BB) = $\cup$ G[x]: for all id x
endfor

Exactly the same as Reaching defs !!

Compute Adef *in/out* Sets

U = universal set of all definitions in the prog
in(0) = 0;   *out*(0) = *gen*(0)
for each basic block BB, (BB != 0), do
    ***in*(BB) = 0;   *out*(BB) = U – *kill*(BB)**

change = true
while (change) do
    change = false
    for each basic block BB, do
        old_out =   *out*(BB)
        in(BB) = $\bigcap$ *out*(Y) : for all predecessors Y of BB
        out(BB) = GEN(BB) + (IN(BB) – KILL(BB))
        if (*old_out* != *out*(BB)) then   change = true
    endfor
endfor

Available Expression Analysis (Aexprs)

- An *expression:* RHS of any statement
  - Ex: in “r2 = r3 + r4” “r3 + r4” is an expression
- An expression e is available at a point p if along *all paths* from e to p, e is not *killed*
- An expression is *killed* between two points when one of its source operands is redefined
  - Ex: “r1 = r2 + r3” kills all expressions involving r1
- Algorithm:
  - Forward dataflow analysis
  - Use the *Intersect function* as the meet operator to guarantee the all-path requirement
  - Looks exactly like *adefs*, except *gen/kill/in/out* are the RHS’s of operations rather than the LHS’s

Available expressions are for detecting global common sub-expression

Available Expression

- Input: A flow graph with *e_kill[B]* and *e_gen[B]*
- Output: *in[B]* and *out[B]*
- Method:

```
for each basic block B
        in[B1] := ∅ ;     out[B1] := e_gen[B1];
        out[B] = U - e_kill[B];
change=true
while(change)
        change=false;
        for each basic block B,
                in[B] := ⋂ out[P]: P is pred of B
                old_out := out[B];
                out[B] := e_gen[B] ⋃ (in[B] – e_kill[B])
                if (out[B] ≠ old_out[B])   change := true;
```

Efficient Calculation of Dataflow

- Order in which the basic blocks are visited is important (*faster convergence*)
- Forward analysis – DFS order
  - Visit a node only when all its predecessors have been visited
- Backward analysis – PostDFS order
  - Visit a node only when all of its successors have been visited

Representing Dataflow Information

- Requirements–Efficiency!
  - Large amount of information to store
  - Fast access/manipulation
- Bit-vectors
  - General strategy used by most compilers
  - Bit positions represent *defs*
  - Efficient set operations: *union*/*intersect*
  - Used for *gen*, *kill*, *in*, *out* for each BB

Optimization using Dataflow

- Classes of optimization
  1. Classical (machine independent)
     - Reducing operation count (redundancy elimination)
     - Simplifying operations
  2. Machine specific
     - Peephole optimizations
     - Takes advantage of specialized hardware features
  3. Instruction Level Parallelism (ILP) enhancing
     - Increasing parallelism
     - Possibly increase instructions

Types of Classical Optimizations

- Operation-level – One operation in isolation
  - Constant folding, strength reduction
  - Dead code elimination (global, but 1 op at a time)
- Local – Pairs of operations in same BB
  - May or may not use dataflow analysis
- Global – Again pairs of operations
  - Pairs of operations in different BBs
- Loop – Body of a loop

Dead Code Elimination

- Remove statement d: *x*:=*y* op *z* whose result is never consumed
- Rules:
  - DU chain for d is empty
  - *y* and *z* are not live at d

Constant Propagation

- Forward propagation of moves/assignment of the form

  d: `rx := L` where `L` is literal

  - Replacement of “rx” with “L” wherever possible
  - d must be available at point of replacement

Forward Copy Propagation

- Forward propagation of RHS of assignment or mov's.

```
r1 := r2            r1 := r2
    .                   .
    .       ──→         .
    .                   .
r4 := r1 + 1        r4 := r2 + 1
```

  - Reduce chain of dependency
  - Possibly create dead code

Forward Copy Propagation

- Rules:

  Statement d_S is source of copy propagation

  Statement d_T is target of copy propagation

  – d_S is a mov statement

  – $src(d_S)$ is a register

  – d_T uses $dest(d_S)$

  – d_S is available definition at d_T

  – $src(d_S)$ is a available expression at d_T

Backward Copy Propagation

- Backward propagation of LHS of an assignment.

  d_T: r1 := r2 + r3 → r4 := r2 + r3

  r5 := r1 + r6 → r5 := r4 + r6

  d_S: r4 := r1 → Dead Code

- Rules:
  - d_T and d_S are in the same basic block
  - dest(d_T) is register
  - *dest(d_T) is not live in out[B]*
  - dest(d_S) is a register
  - d_S uses dest(d_T)
  - dest(d_S) not used between d_T and d_S
  - dest(d_S) not defined between d_T and d_S
  - There is no use of dest(d_T) after the first definition of dest(d_S)

Local Common Sub-Expression Elimination

- Benefits:
  - Reduced computation
  - Generates mov statements, which can get copy propagated
- Rules:
  - d_S and d_T have the same expression
  - $src(d_S) == src(d_T)$ for all sources
  - For all sources *x*, *x* is not redefined between d_S and d_T

d_S: r1 := r2 + r3

d_T: r4 := r2 + r3

d_S: r1 := r2 + r3
r100 := r1

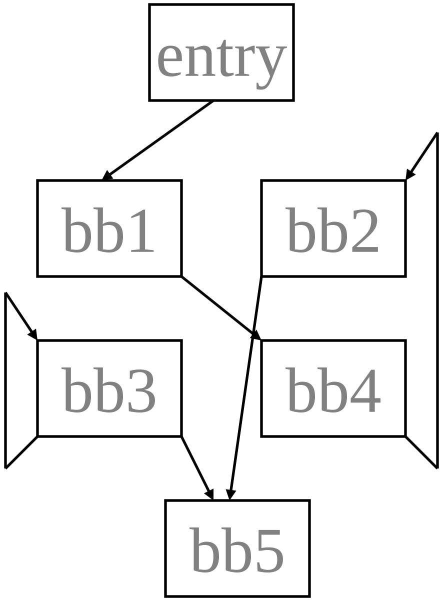
d_T: r4 := r100

Global Common Sub-Expression Elimination

- Rules:
  - d_S and d_T have the same expression
  - $src(d_S) == src(d_T)$ for all sources of d_S and d_T
  - Expression of d_S is available at d_T

Unreachable Code Elimination

```
Mark initial BB visited
to_visit = initial BB
while (to_visit not empty)
    current = to_visit.pop()
    for each successor block of current
        Mark successor as visited;
        to_visit += successor
    endfor
endwhile
Eliminate all unvisited blocks
```



Which BB(s) can be deleted?