# Outline

Compiler Optimization

- Local/peephole optimizations
- Other sources of optimization
- Recognizing loops
- Dataflow analysis

# Outline (cont'd)

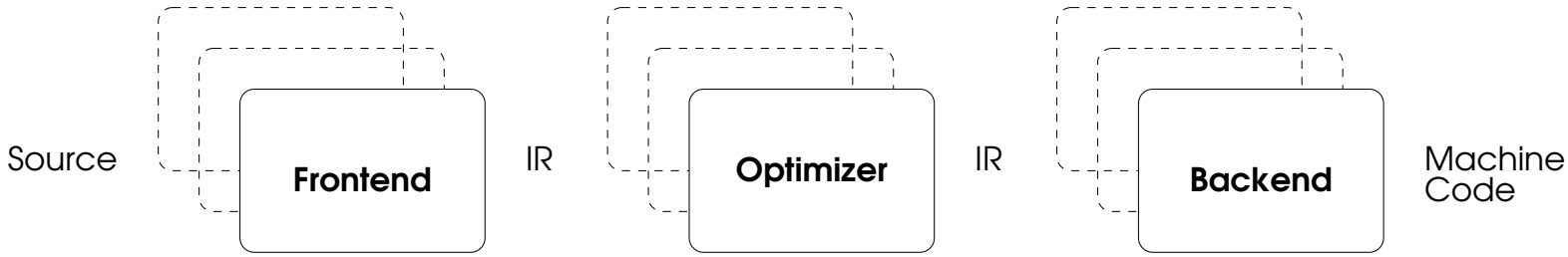
Optimizing (cont'd)

- Algorithms for global optimizations
- Alias analysis

- Code generation
  - Instruction selection
  - Register allocation
  - Instruction scheduling: improving ILP

Optimizations for cache behavior

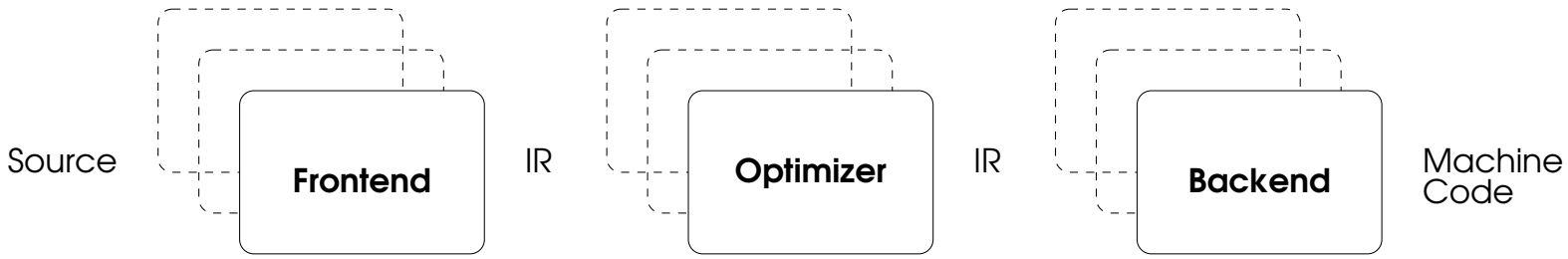
# Compilers




Frontend

- Dependent on source language
- Lexical Analysis
- Parsing
- Semantic analysis (e.g., type checking)

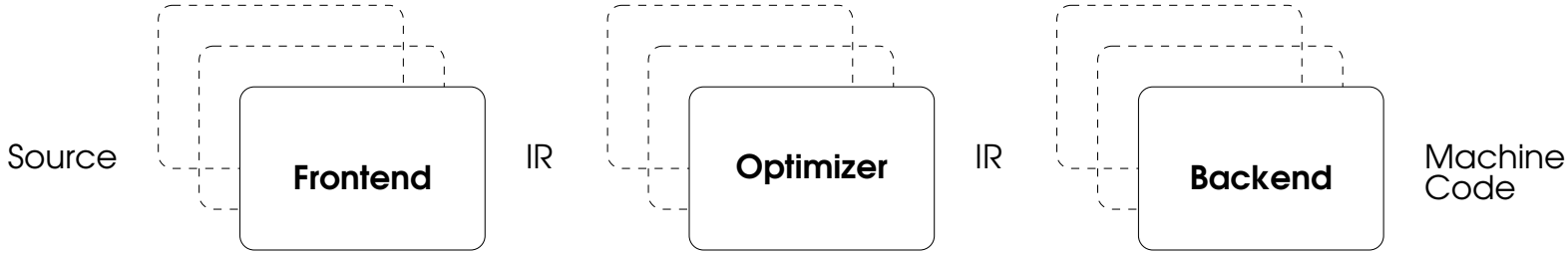
# Compilers




Optimizer

- Independent part of compiler
- Different optimizations possible
- IR to IR translation

# Compilers




- Backend
  - Dependent on target
  - Code selection
  - Code scheduling
  - Register allocation
  - Peephole optimization

# Intermediate Representation (IR)

Flow graph

- Nodes are basic blocks

  Basic blocks are single entry and single exit

- Edges represent control-flow

- Abstract Machine Code
  - Including the notion of functions and procedures
- Symbol table(s) keep track of scope and binding information about names

# Partitioning into basic blocks

1. Determine the leaders, which are:

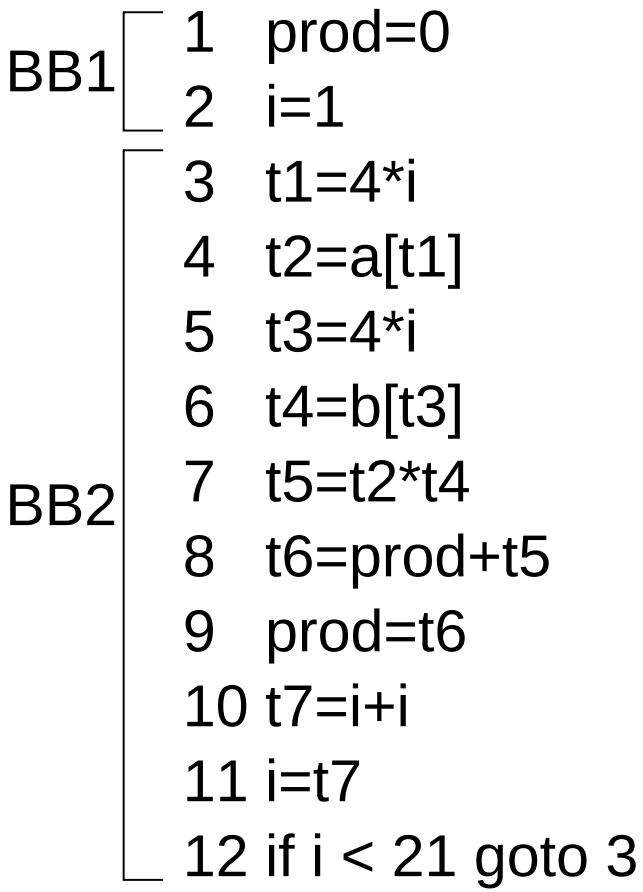
   - The first statement

     Any statement that is the target of a jump

   - Any statement that immediately follows a jump

2. For each leader its basic block consists of the leader and all statements up to but not including the next leader

# Partitioning into basic blocks (cont'd)



# Intermediate Representation (cont'd)

Structure within a basic block:

Abstract Syntax Tree (AST)

- Leaves are labeled by variable names or constants
- Interior nodes are labeled by an operator

Directed Acyclic Graph (DAG)

C-like

3 address statements (like we have already seen)

# Directed Acyclic Graph

Like ASTs:

- Leaves are labeled by variable names or constants
- Interior nodes are labeled by an operator

Nodes can have variable names attached that contain the value of that expression

Common subexpressions are represented by multiple edges to the same expression

# DAG creation

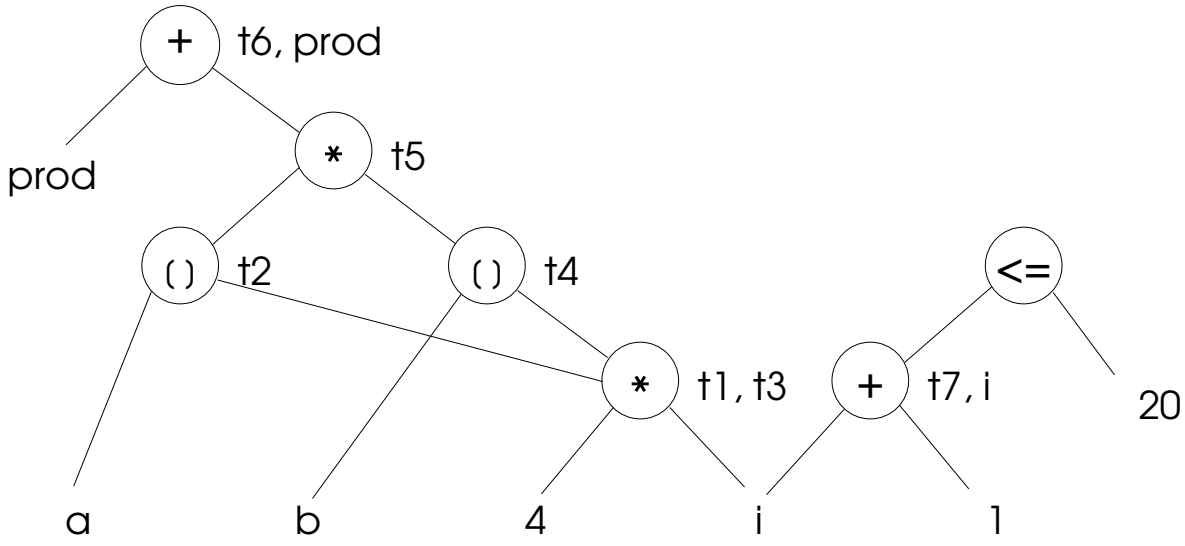
Suppose the following three address statements:

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

$if(i <= 20)$ ... will be treated like case 1 with $x$ undefined

# DAG example

```
 1    t1 = 4 * i
 2    t2 = a[t1]
 3    t3 = 4 * i
 4    t4 = b[t3]
 5    t5 = t2 * t4
 6    t6 = prod + t5
 7    prod = t6
 8    t7 = i + 1
 9    i = t7
10    if (i <= 20) goto 1
```



# Local optimizations

On basic blocks in the intermediate representation

- Machine independent optimizations

As a post code-generation step (often called peephole optimization)

- On a small “instruction window” (often a basic block)
- Includes machine specific optimizations

# Transformations on basic blocks

Examples

Function-preserving transformations

- Common subexpression elimination
- Constant folding
- Copy propagation
- Dead-code elimination
- Temporary variable renaming
- Interchange of independent statements

# Transformations on basic blocks (cont'd)

Algebraic transformations

Machine dependent eliminations/transformations

- Removal of redundant loads/stores
- Use of machine idioms

# Common subexpression elimination

If the same expression is computed more than once it is called a common subexpression

If the result of the expression is stored, we don't have to recompute it

- Moving to a DAG as IR, common subexpressions are automatically detected!

$$\begin{array}{lcl} x = \quad + b & & x = \quad + b \\ \ldots & \Rightarrow & \ldots \\ y = a + b & & y = x \end{array}$$

# Constant folding

Compute constant expression at compile time

- May require some emulation support

$$\begin{array}{lcl} x = \quad +5 & & x = 8 \\ \ldots & \Rightarrow & \ldots \\ y = x & & y = 16 \end{array}$$

# Copy propagation

Propagate original values when copied

Target for dead-code elimination

$$
\begin{array}{lcl}
x = y & & x = y \\
\ldots & \Rightarrow & \ldots \\
z = x & & z = y
\end{array}
$$

# Dead-code elimination

A variable $x$ is dead at a statement if it is not used after that statement

An assignment $x = y + z$ where $x$ is dead can be safely eliminated

Requires live-variable analysis (discussed later on)

# Temporary variable renaming

$$
\begin{array}{ll}
1 = \quad + b & 1 = \quad + b \\
\quad = \ 1 & \quad = \ 1 \\
\ldots & \Rightarrow \quad \ldots \\
1 = d - e & t3 = d - e \\
c = t1 + 1 & c = t3 + 1
\end{array}
$$

- If each statement that defines a temporary defines a new temporary, then the basic block is in normal-form
  - Makes some optimizations at BB level a lot simpler (e.g. common subexpression elimination, copy propagation, etc.)

# Algebraic transformations

There are many possible algebraic transformations

Usually only the common ones are implemented

$$x = x + 0$$

$$x = x \quad 1$$

$$x = x \qquad \Rightarrow x = x << 1$$

$$x = x \;\; \Rightarrow x = x \quad x$$

# Machine dependent eliminations/transformations

Removal of redundant loads/stores

```
1    mov R0, a
2    mov a, R0              // can be removed
```

Removal of redundant jumps, for example

```
1            beq ...,$Lx                  bne ...,$Ly
2            j $Ly          ⇒    $Lx:     ...
3    $Lx:     ...
```

Use of machine idioms, e.g.,

- Auto increment/decrement addressing modes
- SIMD instructions

Etc., etc. (see practical assignment)

# Other sources of optimizations

Global optimizations

- Global common subexpression elimination
- Global constant folding
- Global copy propagation, etc.

Loop optimizations

They all need some dataflow analysis on the flow graph

# Loop optimizations

**Code motion**

- Decrease amount of code inside loop

  Take a loop-invariant expression and place it before the loop

while ($i <= limi \ - \ $) $\Rightarrow$ $= limi \ - 2$
while ($i <= t$)

# Loop optimizations (cont'd)

**Induction variable elimination**

Variables that are locked to the iteration of the loop are called induction variables

Example: in `for (i = 0; i < 10; i++)` $i$ is an induction variable

Loops can contain more than one induction variable, for example, hidden in an array lookup computation

- Often, we can eliminate these extra induction variables

# Loop optimizations (cont'd)

**Strength reduction**

Strength reduction is the replacement of expensive operations by cheaper ones (algebraic transformation)

- Its use is not limited to loops but can be helpful for induction variable elimination

$$
\begin{array}{l}
i = i + 1 \\
1 = i * 4 \\
t2 = a[t1] \\
\text{if } (i < 10) \text{ goto top}
\end{array}
\quad \Rightarrow \quad
\begin{array}{l}
i = i + 1 \\
t1 = t1 + 4 \\
t2 = a[t1] \\
\text{if } (i < 10) \text{ goto top}
\end{array}
$$

# Loop optimizations (cont'd)

**Induction variable elimination (2)**

Note that in the previous strength reduction we have to initialize $1$ before the loop

After such strength reductions we can eliminate an induction variable

$$
\begin{array}{l}
i = i + 1 \\
t1 = t1 + 4 \\
t2 = a[t1] \\
\text{if } (i < 10) \text{ goto top}
\end{array}
\quad \Rightarrow \quad
\begin{array}{l}
1 = 1 + 4 \\
t2 = a[t1] \\
\text{if } (t1 < 0) \text{ goto top}
\end{array}
$$

# Finding loops in flow graphs

**Dominator relation**

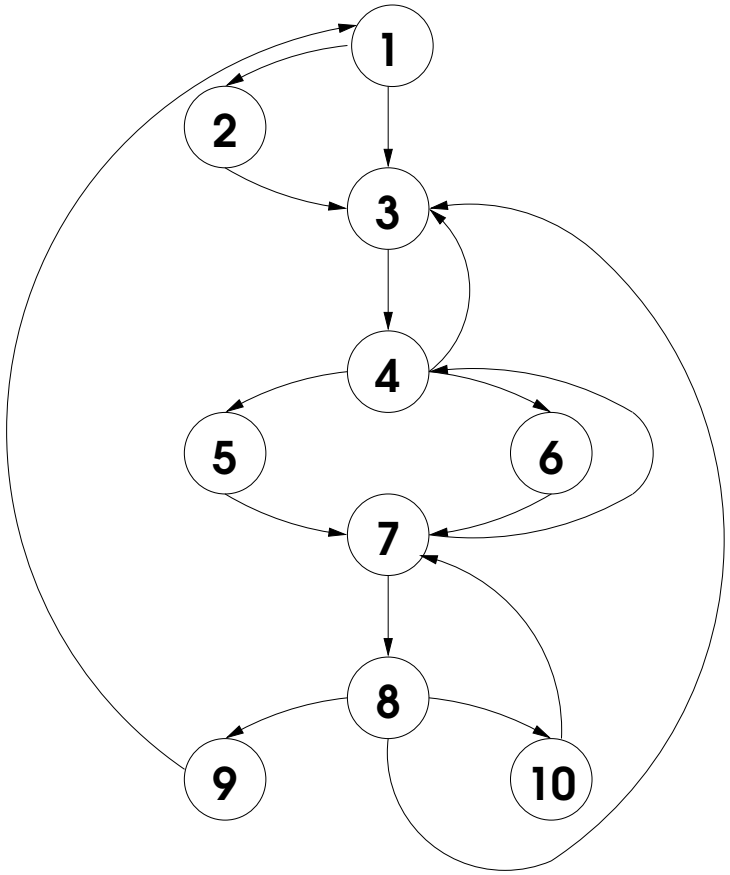
Node A dominates node B if all paths to node B go through node A

A node always dominates itself

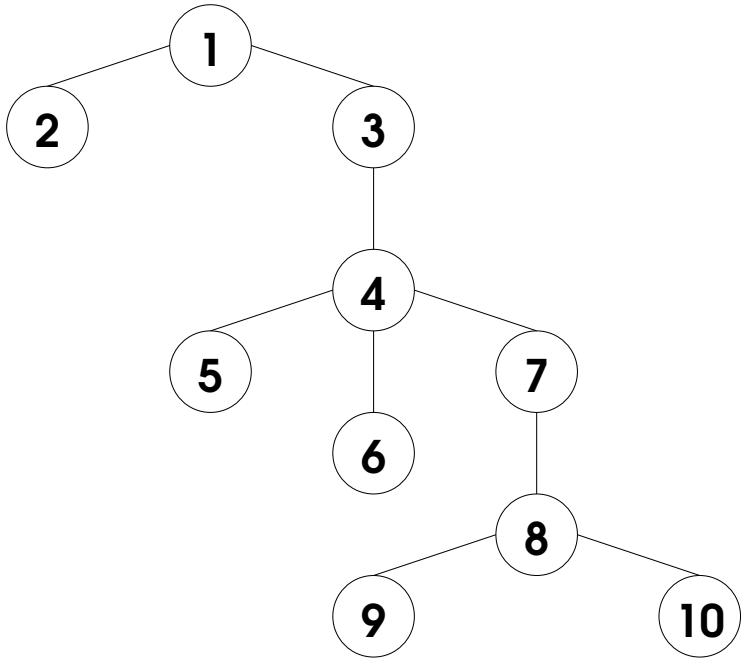
We can construct a tree using this relation: the Dominator tree

# Dominator tree example

Flow graph




Dominator tree



# Natural loops

A loop has a single entry point, the header, which dominates the loop

- There must be a path back to the header
- Loops can be found by searching for edges of which their heads dominate their tails, called the backedges
- Given a backedge $n \rightarrow d$, the natural loop is $d$ plus the nodes that can reach $n$ without going through $d$

# Finding natural loop of $\rightarrow d$

procedure insert($m$) {
  if (not $m \in loop$) {
    $loop = loop \cup m$
    push($m$)
  }
}

$st\ \ ck = \emptyset$
$loop = \{d\}$
insert( )
while ($st\ \ ck \neq \emptyset$) {
  $m$ = pop()
  for ($p \in pred(m\ )$ insert($p$)
}

# Natural loops (cont'd)

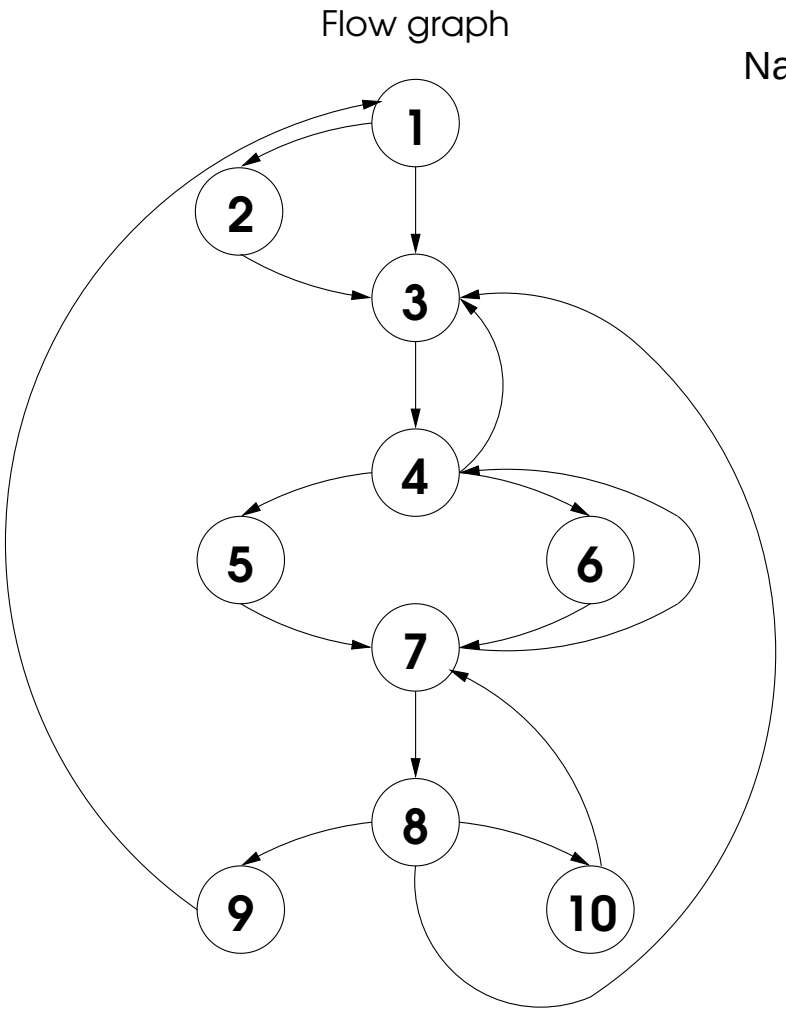
When two backedges go to the same header node, we may join the resulting loops

When we consider two natural loops, they are either completely disjoint or one is nested inside the other

The nested loop is called an inner loop

A program spends most of its time inside loops, so loops are a target for optimizations. This especially holds for inner loops!

# Our example revisited




Natural loops:

1. backedge 10 −> 7:  {7,8,10}  (the inner loop)
2. backedge 7 −> 4: {4,5,6,7,8,10}
3. backedges 4 −> 3 and 8 −> 3: {3,4,5,6,7,8,10}
4. backedge 9 −> 1: the entire flow graph

# Reducible flow graphs

A flow graph is reducible when the edges can be partitioned into forward edges and backedges

The forward edges must form an acyclic graph in which every node can be reached from the initial node

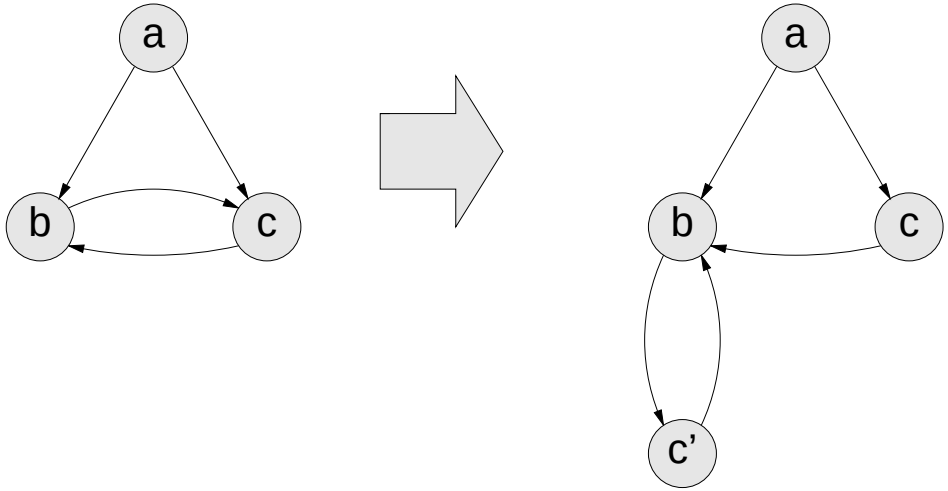
- Exclusive use of structured control-flow statements such as `if-then-else`, `while` and `break` produces reducible control-flow

  Irreducible control-flow can create loops that cannot be optimized

# Reducible flow graphs (cont'd)

Irreducible control-flow graphs can always be made reducible

This usually involves some duplication of code



# Dataflow analysis

Data analysis is needed for global code optimization, e.g.:

- Is a variable live on exit from a block? Does a definition reach a certain point in the code?

- Dataflow equations are used to collect dataflow information
  - A typical dataflow equation has the form

$$o\ \ [S] = ge\ \ [S] \cup (i\ \ [S] - kil\ [S])$$

- The notion of generation and killing depends on the dataflow analysis problem to be solved

- Let's first consider Reaching Definitions analysis for structured programs

# Reaching definitions

A definition of a variable $x$ is a statement that assigns or may assign a value to $x$

- An assignment to $x$ is an unambiguous definition of $x$
- An ambiguous assignment to $x$ can be an assignment to a pointer or a function call where $x$ is passed by reference

# Reaching definitions (cont'd)

When $x$ is defined, we say the definition is generated

An unambiguous definition of $x$ kills all other definitions of $x$

When all definitions of $x$ are the same at a certain point, we can use this information to do some optimizations

Example: all definitions of $x$ define $x$ to be 1. Now, by performing constant folding, we can do strength reduction if $x$ is used in $z = y \quad x$

# Dataflow analysis for reaching definitions

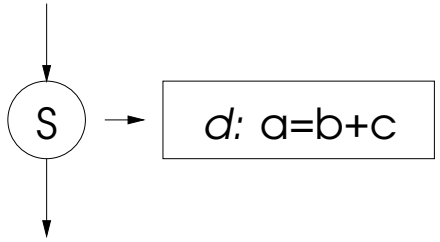
During dataflow analysis we have to examine every path that can be taken to see which definitions reach a point in the code

Sometimes a certain path will never be taken, even if it is part of the flow graph

Since it is undecidable whether a path can be taken, we simply examine all paths

This won’t cause false assumptions to be made for the code: it is a conservative simplification

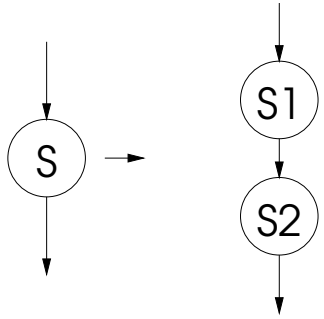
- It merely causes optimizations not to be performed




gen[S]= $d\}$

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

out[S]=gen[S]∪(in[S]-kill[S])



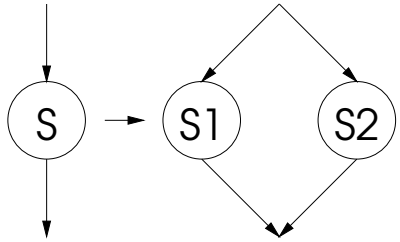

gen[S]=gen[S2]∪(gen[S1]-kill[S2])

kill[S]=kill[S2]∪(kill[S1]-gen[S2])

in[S1]=in[S]

in[S2]=out[S1]

out[S]=out[S2]

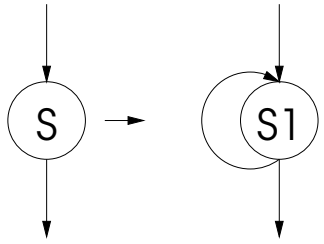



gen[S]=gen[S1]∪gen[S2]

kill[S]=kill[S1]∩kill[S2]

in[S1]=in[S2]=in[S]

out[S]=out[S1]∪out[S2]




gen[S]=gen[S1]

kill[S]=kill[S1]

in[S1]=in[S]∪gen[S1]

out[S]=out[S1]

# Dealing with loops

The in-set to the code inside the loop is the in-set of the loop plus the out-set of the loop: $i \ \ [S1] = in[S] \cup out[S1]$

- The out-set of the loop is the out-set of the code inside: $out[S] = out[S1]$

  Fortunately, we can also compute $out[S1]$ in terms of $in[S1]$: $o \ \ [S1] = ge \ \ [S1] \cup (i \ \ [S1] - kil \ [S1]$

# Dealing with loops (cont'd)

$I = in[S1], \quad = out[S1], J = in[S], G = gen[S]$ and $K = kill[S]$

$I = I \cup O$

- $O = G \cup (I - K)$

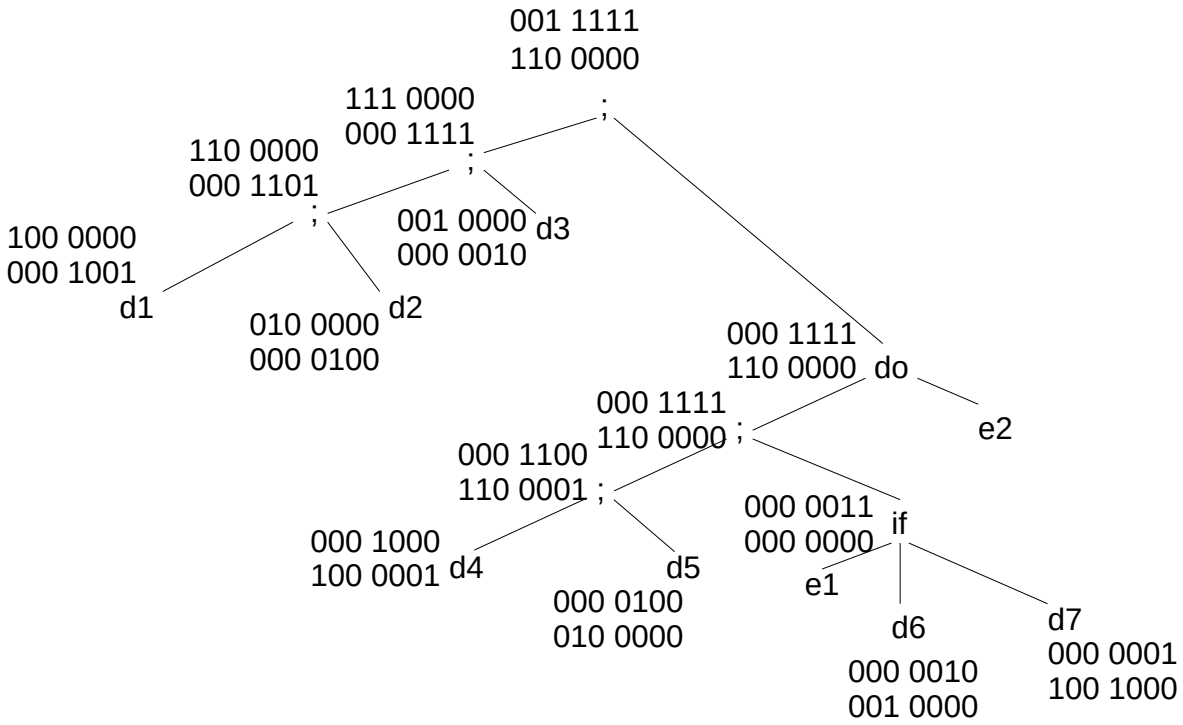
  Assume $O = \emptyset$, then $I^1 = J$

  $O^1 = G \cup (I^1 - K) = G \cup (J - K)$

- $I^2 = J \cup O^1 = J \cup G \cup (J - K) = J \cup G$
- $O^2 = G \cup (I^2 - K) = G \cup (J \cup G - K) = G \cup (J - K)$
- $O^1 = O^2$ so $in[S1] = in[S] \cup gen[S1]$ and $out[S] = out[S1]$

# Reaching definitions example

$d_1$ i = m - 1
$d$ j = n
$d_3$ a = u1
do
$d_4$ i = i + 1
$d_5$ j = j - 1
if (e1)
$d_6$ a = u2
else
$d_7$ i = u3
while (e2)




In reality, dataflow analysis is often performed at the granularity of basic blocks rather than statements

# Iterative solutions

Programs in general need not be made up out of structured control-flow statements

We can do dataflow analysis on these programs using an iterative algorithm

The equations (at basic block level) for reaching definitions are:

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

$$out[B] = gen[B] \cup (i\ \ [B] - kil\ [B])$$

# Iterative algorithm for reaching definitions

for (each block B) $o \quad [ \quad ] = gen[B]$

do $\{$

change $=$ false

for (each block B) $\{$

$i \ [ \quad ] = \bigcup_{P \in pred(B)} o \quad [P]$

oldout $= out[B]$

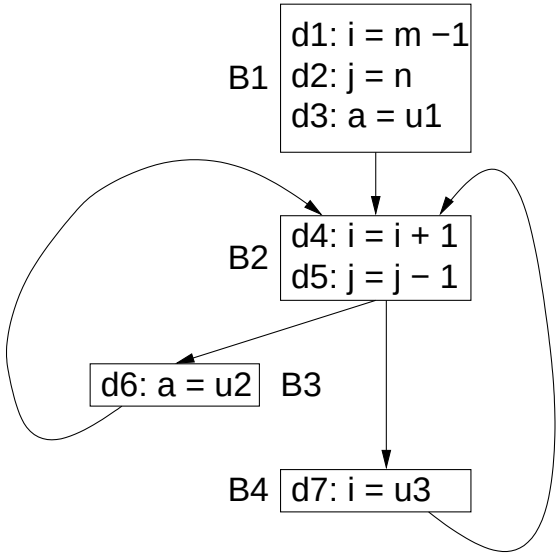
$o \quad [ \quad ] = ge \ [ \quad ] \cup (i \ [ \quad ] - kil \ [ \quad ]$

if ($out[B] =$ oldout) change $=$ true

$\}$

$\}$ while (change)

# Reaching definitions: an example




gen[B1] = {d1,d2,d3}
kill[B1] = {d4,d5,d6,d7}

gen[B2] = {d4,d5}
kill[B2] = {d1,d2,d7}

gen[B3] = {d6}
kill[B3] = {d3}

gen[B4] = {d7}
kill[B4] = {d1,d4}

| Block B | Initial | | Pass 1 | | Pass 2 | |
|---|---|---|---|---|---|---|
| | $in[B]$ | $out[B]$ | $in[B]$ | $out[B]$ | $in[B]$ | $out[B]$ |
| B1 | 000 0000 | 111 0000 | 000 0000 | 111 0000 | 000 0000 | 111 0000 |
| B2 | 000 0000 | 000 1100 | 111 0011 | 001 1110 | 111 1111 | 001 1110 |
| B3 | 000 0000 | 000 0010 | 001 1110 | 000 1110 | 001 1110 | 000 1110 |
| B4 | 000 0000 | 000 0001 | 001 1110 | 001 0111 | 001 1110 | 001 0111 |

# Available expressions

An expression $e$ is available at a point $p$ if every path from the initial node to $p$ evaluates $e$, and the variables used by $e$ are not changed after the last evaluations

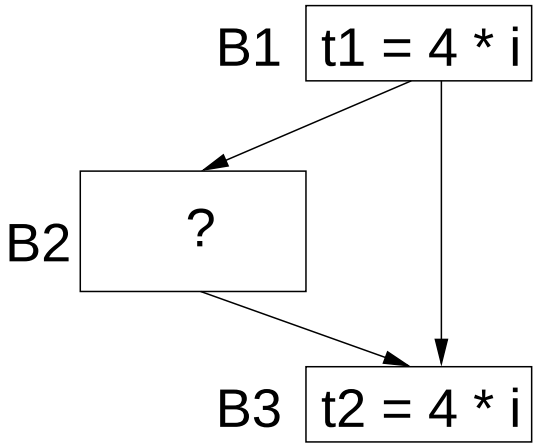
- An available expression $e$ is killed if one of the variables used by $e$ is assigned to

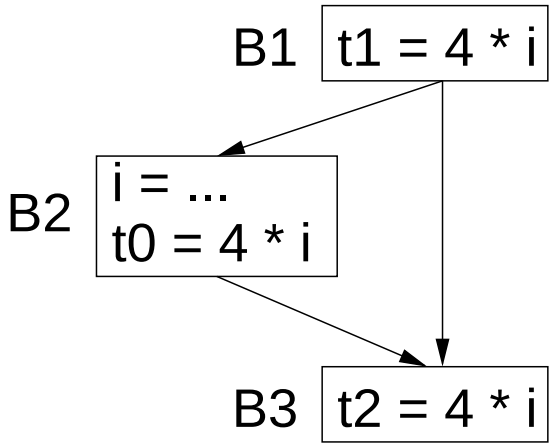
  An available expression $e$ is generated if it is evaluated

  Note that if an expression $e$ is assigned to a variable used by $e$, this expression will not be generated

# Available expressions (cont'd)

Available expressions are mainly used to find common subexpressions



# Available expressions (cont'd)

Dataflow equations:

$$o\ [\ ] = e\_ge\ [\ ] \cup (i\ [\ ] - e\_kil\ [\ ]$$

$$in[B] = \bigcap_{P \in pred(B)} out[P] \quad \text{for B not initial}$$

$$i\ [\ 1] = \emptyset \quad \text{where B1 is the initial block}$$

The confluence operator is intersection instead of the union!

# Liveness analysis

A variable is live at a certain point in the code if it holds a value that may be needed in the future

Solve backwards:

- Find use of a variable
- This variable is live between statements that have found use as next statement
- Recurse until you find a definition of the variable

# Dataflow for liveness

Using the sets $se[\ ]$ and $def[\ ]$

- $def[B]$ is the set of variables assigned values in prior to any use of that variable in
- $se[\ ]$ is the set of variables whose values may be used in prior to any definition of the variable

A variable comes live into a block (in $i\ [\ ]$), if it is either used before redefinition of it is live coming out of the block and is not redefined in the block

A variable comes live out of a block (in $out[B]$) if and only if it is live coming into one of its successors

# Dataflow equations for liveness

$$i\ [\quad] = use[B] \cup (out[B] - def[\quad])$$

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

Note the relation between reaching-definitions equations: the roles of $in$ and $out$ are interchanged

# Algorithms for global optimizations

**Global common subexpression elimination**

First calculate the sets of available expressions

For every statement $s$ of the form $x = y + z$ where $y + z$ is available do the following

- Search backwards in the graph for the evaluations of $y + z$
- Create a new variable $u$
- Replace statements $\quad = y + z$ by $\quad = y + z; \quad =$
- Replace statement $s$ by $x =$

# Copy propagation

Suppose a copy statement $s$ of the form $x = y$ is encountered. We may now substitute a use of $x$ by a use of $y$ if

- Statement $s$ is the only definition of $x$ reaching the use
- On every path from statement $s$ to the use, there are no assignments to $y$

# Copy propagation (cont'd)

To find the set of copy statements we can use, we define a new dataflow problem

An occurrence of a copy statement generates this statement

An assignment to $x$ or $y$ kills the copy statement $x = y$

Dataflow equations:

$$out[B] = c\_gen[B] \cup (i\ \ [B] - c\_kil\ [B])$$

$$in[B] = \bigcap_{P \in pred(B)} out[P] \text{ for B not initial}$$

$$i\ \ [\ \ 1] = \emptyset \text{ where B1 is the initial block}$$

# Copy propagation (cont'd)

For each copy statement $s$: $x = y$ do

- Determine the uses of $x$ reached by this definition of $x$
- Determine if for each of those uses this is the only definition reaching it ( $s \in i\ [\ _{use}]$)
- If so, remove $s$ and replace the uses of $x$ by uses of $y$

# Detection of loop-invariant computations

1. Mark invariant those statements whose operands are constant or have reaching definitions outside the loop

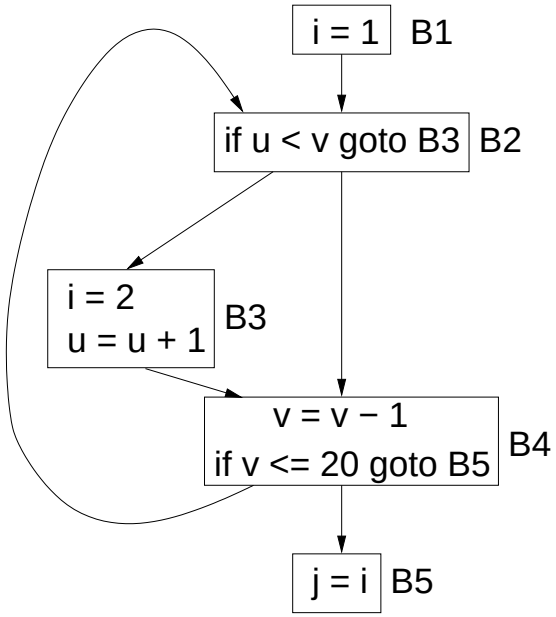
2. Repeat step 3 until no new statements are marked invariant

3. Mark invariant those statements whose operands either are constant, have reaching definitions outside the loop, or have one reaching definition that is marked invariant

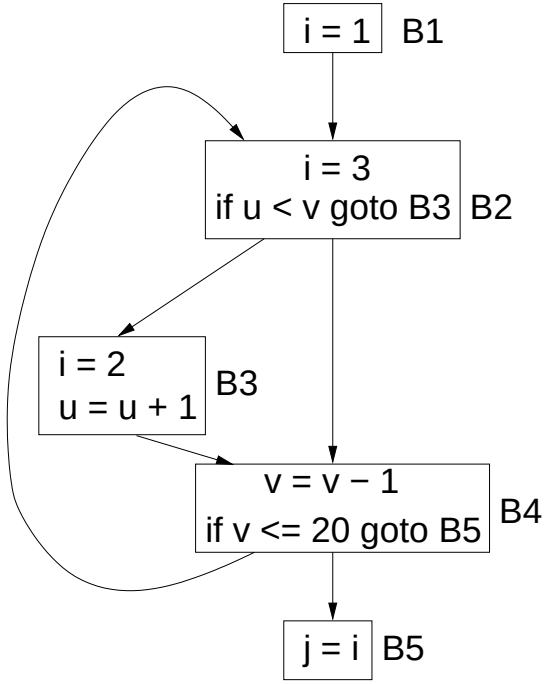
# Code motion

1. Create a pre-header for the loop
2. Find loop-invariant statements
3. For each statement $s$ defining $x$ found in step 2, check that
   (a) it is in a block that dominate all exits of the loop
   (b) $x$ is not defined elsewhere in the loop
   (c) all uses of $x$ in the loop can only be reached from this statement $s$
4. Move the statements that conform to the pre-header

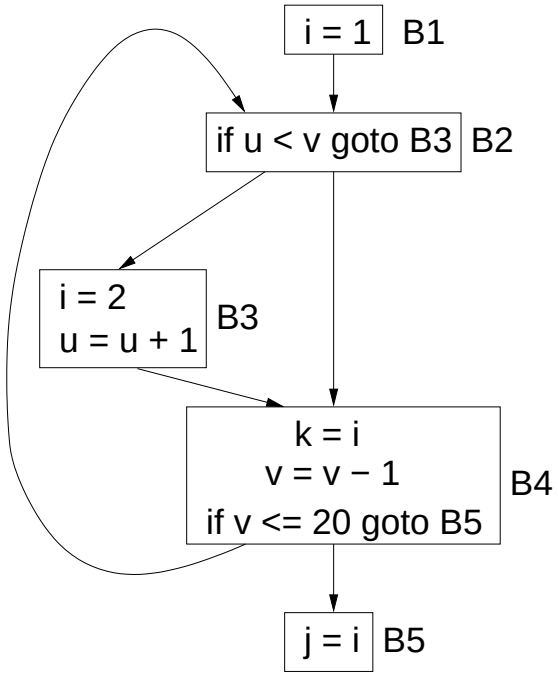
# Code motion (cont'd)




Condition (a)




Condition (b)




Condition (c)

# Detection of induction variables

A basic induction variable $i$ is a variable that only has assignments of the form $i = i \quad c$

Associated with each induction variable $j$ is a triple $(i, c, d)$ where $i$ is a basic induction variable and $c$ and $d$ are constants such that $j = c * i + d$

- In this case $j$ belongs to the family of $i$
- The basic induction variable $i$ belongs to its own family, with the associated triple $(i, 1, 0$

# Detection of induction variables (cont'd)

Find all basic induction variables in the loop

Find variables $k$ with a single assignment in the loop with one of the following forms:

- $k = j * b$, $k = b * j$, $k = j/b$, $k = j + b$, $k = b + j$, where $b$ is a constant and $j$ is an induction variable

If $j$ is not basic and in the family of $i$ then there must be

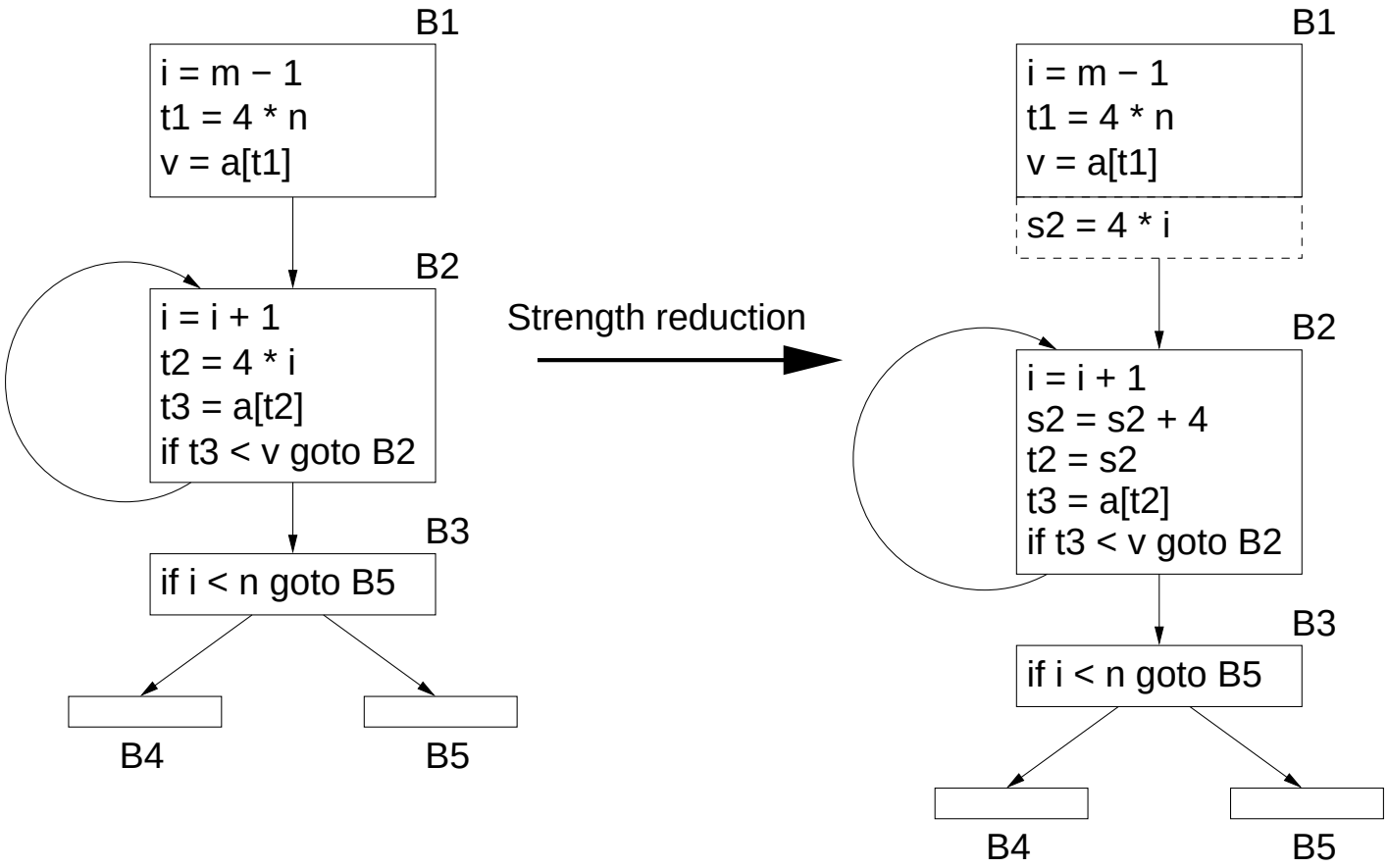
- No assignment of $i$ between the assignment of $j$ and $k$
- No definition of $j$ outside the loop that reaches $k$

# Strength reduction for induction variables

Consider each basic induction variable $i$ in turn. For each variable $j$ in the family of $i$ with triple $(i, c, d)$:

- Create a new variable $s$
- Replace the assignment to $j$ by $j = s$
- Immediately after each assignment $i = i$ append $s = s + c$
- Place $s$ in the family of $i$ with triple $(i, c, d)$
- Initialize $s$ in the preheader: $s = c * i + d$

# Strength reduction for induction variables (cont'd)



# Elimination of induction variables

Consider each basic induction variable $i$ only used to compute other induction variables and tests

Take some $j$ in $i$'s family such that $c$ and $d$ from the triple $(i, c, d)$ are simple

Rewrite tests `if (`$i$ `relop` $x$`)` to

$$= c * x + d;\ \texttt{if (}j\ \texttt{relop )}$$

- Delete assignments to $i$ from the loop
- Do some copy propagation to eliminate $j = s$ assignments formed during strength reduction

# Alias Analysis

Aliases, e.g. caused by pointers, make dataflow analysis more complex (uncertainty regarding what is defined and used: $x = *p$ might use any variable)

Use dataflow analysis to determine what a pointer might point to

$in[B]$ contains for each pointer $p$ the set of variables to which $p$ could point at the beginning of block

- Elements of $i$ $[$ $]$ are pairs $(p,$ where $p$ is a pointer and a variable, meaning that $p$ might point to

- $o$ $[$ $]$ is defined similarly for the end of

# Alias Analysis (cont'd)

Define a function $s_B$ such that $s_B(i\ [B]) = out[B]$

- $ra\ \ s_B$ is composed of $s_s$, for each statement $s$ of block
  - If $s$ is $p =$ or $p =$ $\pm c$ in case is an array, then
    $$s_s(S) = (S - \{(p, b\ |\text{any variable b}\}) \cup \{(p,\ \ \}$$
  - If $s$ is $p = q \pm c$ for pointer $q$ and nonzero integer $c$, then
    $$ra\ \ s_s(S) = (S - \{(p, b\ |\text{any variable b}\}$$
    $$\cup \{(p, b)|(q, b) \in S \text{ and b is an array variable}\}$$
  - If $s$ is $p = q$, then
    $$ra\ \ s_s(S) = (S - \{(p, b)|\text{any variable b}\})$$
    $$\cup \{(p, b)|(q, b) \in S\}$$

# Alias Analysis (cont'd)

– If $s$ assigns to pointer $p$ any other expression, then

$$s_s(S) = S - \{(p, b \mid \text{any variable b}\}$$

– If $s$ is not an assignment to a pointer, then $ra \;\; s_s(S) = S$

Dataflow equations for alias analysis:

$$o \;\; [\;\;] = \qquad s_B(i \;\; [\;\;]$$

$$i \;\; [\;\;] = \bigcup_{P \in pred(B)} out[P]$$

where $\qquad s_B(S) = \qquad s_{s_k}( \qquad s_{s_{k-1}}(\cdots(tra \;\; s_{s_1}(S$

# Alias Analysis (cont'd)

How to use the alias dataflow information? Examples:

- In reaching definitions analysis (to determine $ge$ and $kil$ )

  statement $p =$ generates a definition of every variable $b$ such that $p$ could point to $b$

  $\rightarrow$ $*p = a$ kills definition of $b$ only if $b$ is not an array and is the only variable $p$ could possibly point to (to be conservative)

- In liveness analysis (to determine $def$ and $se$)

  $p = a$ uses $p$ and $a$. It defines $b$ only if $b$ is the unique variable that $p$ might point to (to be conservative)

  $\rightarrow$ $= *p$ defines , and represents the use of and a use of any variable that could point to