Introduction

- Concerns with machine-independent code optimization
  - 90-10 rule: execution spends 90% time in 10% of the code.
    - It is moderately easy to achieve 90% optimization. The rest 10% is very difficult.
    - Identification of the 10% of the code is not possible for a compiler – it is the job of a profiler.
- In general, loops are the hot-spots

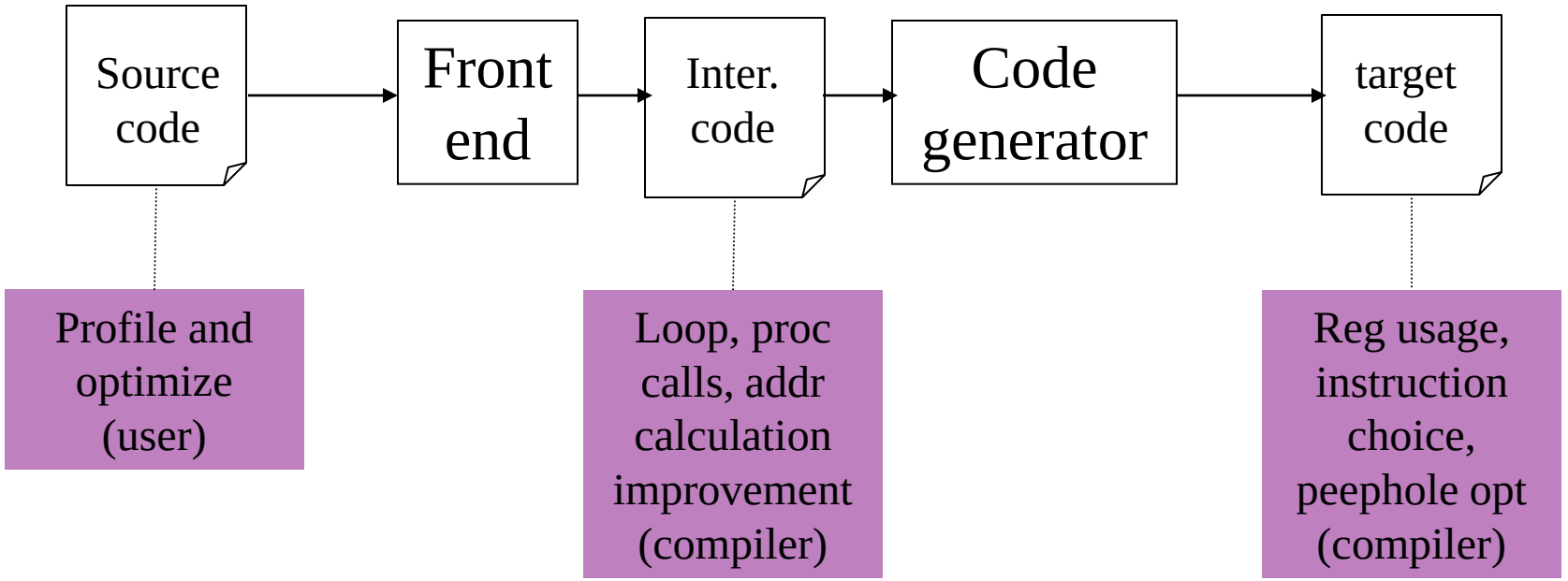
Introduction

- Criterion of code optimization
  - Must preserve the semantic equivalence of the programs
  - The algorithm should not be modified
  - Transformation, on average should speed up the execution of the program
  - Worth the effort: Intellectual and compilation effort spend on insignificant improvement.

    Transformations are simple enough to have a good effect

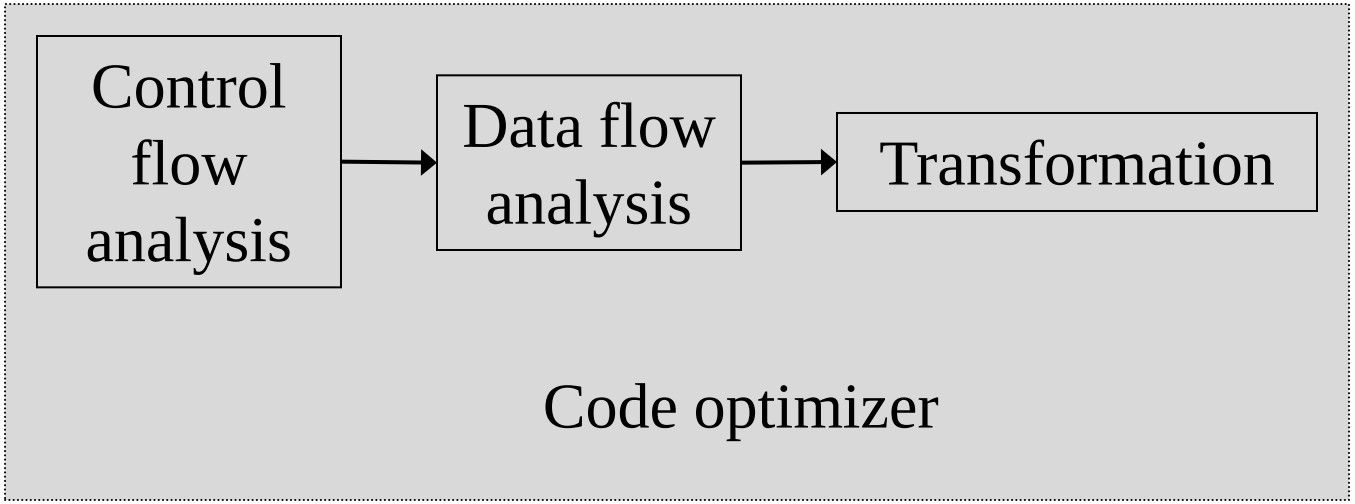
Introduction

- Optimization can be done in almost all phases of compilation.



Introduction

- Organization of an optimizing compiler



Classifications of Optimization techniques

- Peephole optimization
- Local optimizations
- Global Optimizations
  - Inter-procedural
  - Intra-procedural
- Loop optimization

Factors influencing Optimization

- The target machine: machine dependent factors can be parameterized to compiler for fine tuning
- Architecture of Target CPU:
  - Number of CPU registers
  - RISC vs CISC
  - Pipeline Architecture
  - Number of functional units
- Machine Architecture
  - Cache Size and type
  - Cache/Memory transfer rate

Themes behind Optimization Techniques

- 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

Redundancy elimination

- Redundancy elimination = determining that two computations are equivalent and eliminating one.
- There are several types of redundancy elimination:
  - Value numbering
    - Associates symbolic values to computations and identifies expressions that have the same value
  - Common subexpression elimination
    - Identifies expressions that have operands with the same name
  - Constant/Copy propagation
    - Identifies variables that have constant/copy values and uses the constants/copies in place of the variables.
  - Partial redundancy elimination
    - Inserts computations in paths to convert partial redundancy to full redundancy.

Optimizing Transformations

- Compile time evaluation
- Common sub-expression elimination
- Code motion
- Strength Reduction
- Dead code elimination
- Copy propagation
- Loop optimization
  - Induction variables and strength reduction

Compile-Time Evaluation

- Expressions whose values can be pre-computed at the compilation time
- Two ways:
  - Constant folding
  - Constant propagation

Compile-Time Evaluation

- **Constant folding**: Evaluation of an expression with constant operands to replace the expression with single value
- Example:

```
area := (22.0/7.0) * r ** 2
```

```
area := 3.14286 * r ** 2
```

Compile-Time Evaluation

- **Constant Propagation**: Replace a variable with constant which has been assigned to it earlier.
- Example:

```
pi := 3.14286
area = pi * r ** 2
```

```
area = 3.14286 * r ** 2
```

Constant Propagation

- What does it mean?
  - Given an assignment x = c, where c is a constant, replace later uses of x with uses of c, provided there are no intervening assignments to x.
    - Similar to copy propagation
    - Extra feature: It can analyze constant-value conditionals to determine whether a branch should be executed or not.
- When is it performed?
  - Early in the optimization process.
- What is the result?
  - Smaller code
  - Fewer registers

Common Sub-expression Evaluation

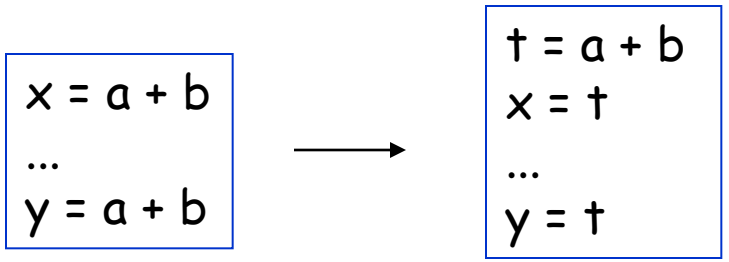
- Identify common sub-expression present in different expression, compute once, and use the result in all the places.
  - The *definition* of the variables involved should not change

Example:

```
a := b * c                  temp := b * c
…                           a := temp
…                           …
x := b * c + 5      ⟶      x := temp + 5
```

Common Subexpression Elimination

- Local common subexpression elimination
  - Performed within basic blocks
  - Algorithm sketch:
    - Traverse BB from top to bottom
    - Maintain table of expressions evaluated so far
      - if any operand of the expression is redefined, remove it from the table
    - Modify applicable instructions as you go
      - generate temporary variable, store the expression in it and use the variable next time the expression is encountered.



Common Subexpression Elimination

```
c = a + b
d = m * n
e = b + d
f = a + b
g = - b
h = b + a
a = j + a
k = m * n
j = b + d
a = - b
if m * n go to L
```

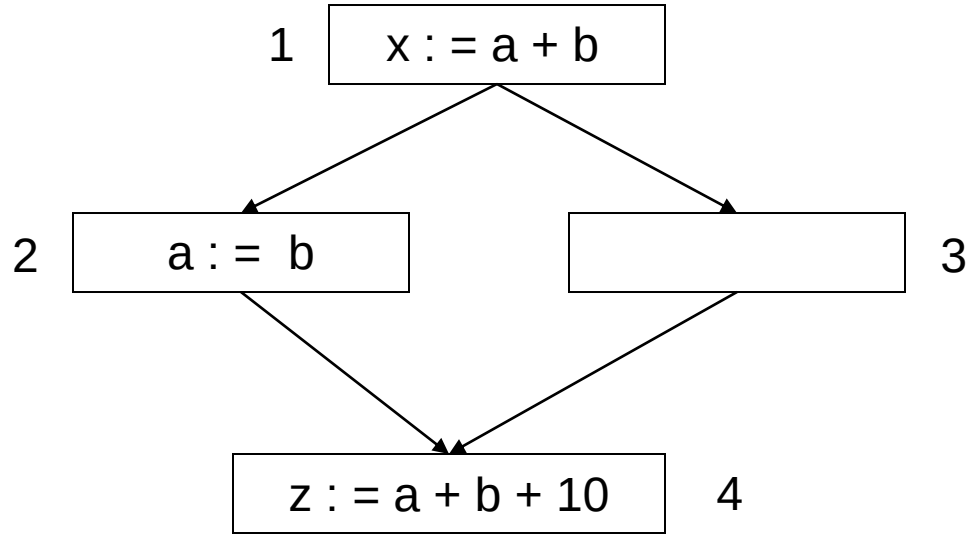
```
t1 = a + b
c = t1
t2 = m * n
d = t2
t3 = b + d
e = t3
f = t1
g = -b
h = t1 /* commutative */
a = j + a
k = t2
j = t3
a = -b
if t2 go to L
```

the table contains quintuples:
(pos, opd1, opr, opd2, tmp)

Common Subexpression Elimination

- Global common subexpression elimination
  - Performed on flow graph
  - Requires available expression information
    - In addition to finding what expressions are available at the endpoints of basic blocks, we need to know where each of those expressions was most recently evaluated (which block and which position within that block).

Common Sub-expression Evaluation



"a + b" is not a common sub-expression in 1 and 4

None of the variable involved should be modified in any path

Code Motion

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

Code Motion

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

Example: Code hoisting

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

```
temp : = x ** 2
if (a< b) then
    z := temp
else
    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*: reduce 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;
```

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

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

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)

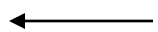
Dead Code Elimination

- Dead Code are portion of the program which will not be executed in any path of the program.
  - Can be removed
- Examples:
  - No control flows into a basic block
  - A variable is dead at a point -> its value is not used anywhere in the program
  - An assignment is dead -> assignment assigns a value to a dead variable

Dead Code Elimination

- Examples:

```
DEBUG:=0
if (DEBUG) print
```



Can be eliminated

Copy Propagation

- What does it mean?
  - 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.
- What is the result?
  - Smaller code

Copy Propagation

- `f := g` are called copy statements or copies
- Use of `g` for `f`, whenever possible after copy statement

Example:

x[i] = a;	x[i] = a;
sum = x[i] + a;	sum = a + a;

- May not appear to be code improvement, but opens up scope for other optimizations.

Local Copy Propagation

- Local copy propagation
  - Performed within basic blocks
  - Algorithm sketch:
    - traverse BB from top to bottom
    - maintain table of copies encountered so far
    - modify applicable instructions as you go

Loop Optimization

- Decrease the number if instruction in the inner loop
- Even if we increase no of instructions in the outer loop
- Techniques:
  - Code motion
  - Induction variable elimination
  - Strength reduction

PEEPHOLE OPTIMIZATION

- The Peephole Optimization is a kind of optimization technique performed over a very small set of instructions in a segment of generated assembly code.
- The set of instructions is called a "peephole" or a "window".
- It works by recognizing sets of instructions that can be replaced by shorter or faster set of instructions to achieve speed or performance in the execution of the instruction sequences

PEEPHOLE OPTIMIZATION

- It works by recognizing sets of instructions that can be replaced by shorter or faster set of instructions to achieve speed or performance in the execution of the instruction sequences.
- Basically Peephole Optimization is a method which consists of a local investigation of the generated object code means intermediate assembly code to identify and replace inefficient sequence of instructions to achieve optimality in targeted machine code in context of execution or response time, performance of the algorithm and memory or other resources usage by the program.

COMMON TECHNIQUES APPLIED IN PEEPHOLE OPTIMIZATION

- Constant folding - Assess constant sub expressions in advance.

  E.g. **r2 := 3 X 2** **becomes** **r2 := 6**

- Strength reduction - Faster Operations will be replaced with slower one.

  E.g. **r1:= r2 X 2** **becomes** **r1 := r2 + r2 then r1 := r2<<1**

  r1 := r2/2 **becomes** **r1 := r2>>1**

- Null sequences – Operations that are ineffective will be removed.

  E.g. **r1 := r1 + 0** **or** **r1 := r1 X 1 has no effect**

- Combine Operations - Replacement of the few operations with similar effective single operation.

  E.g. **r2 := r1 X 2**

  r3 := r2 X 1 **becomes** **r3 := r1 + r1**

- Algebraic Laws - Simplification and reordering of the instructions using algebraic laws.

  E.g. **r1 := r2**

  r3 := r1 **becomes** **r3 := r2;**

MACHINE SPECIFIC PEEPHOLE OPTIMIZERS

- Peephole optimization is simple but effective optimization technique.
- It was noted that when a program gets compiled, the code emitted from the code generators contained many redundancies around borders of basic blocks like chains of jump instructions. It becomes complicated case analysis to reduce these redundancies during the code generation phase. So it was appropriate to define a separate phase that would deal with them.
- The concept was described as follow:
  - A peephole, a small window which consisting no more than two assembly code instructions, is passed over the code.
  - Whenever redundant instructions are found in the sequence, they are replaced by shorter or faster instruction sequences.

MACHINE SPECIFIC PEEPHOLE OPTIMIZERS

- For that the peephole optimizer uses the simple hand written pattern rules.
- These are first matched with the assembly instructions for the applicability of testing, and if the match found, the instructions are replaced.
- Therefore, a typical pattern rule consists of two parts a match part and a replacement part.
- The pattern set is usually small as this is sufficient for fast and efficient optimization.

Local Optimization

Optimization of Basic Blocks

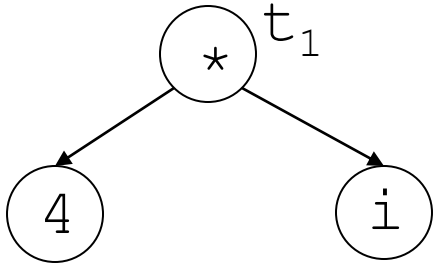
- Many structure preserving transformations can be implemented by construction of DAGs of basic blocks

DAG representation of Basic Block (BB)

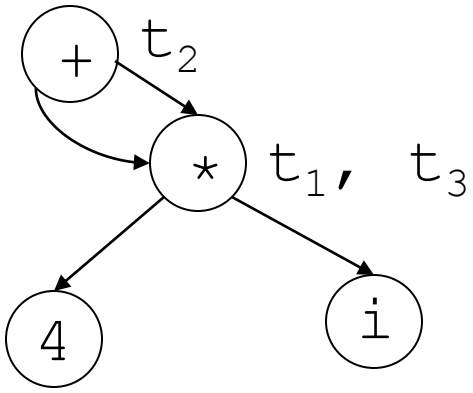
- Leaves are labeled with unique identifier (var name or const)
- Interior nodes are labeled by an operator symbol
- Nodes optionally have a list of labels (identifiers)
- Edges relates operands to the operator (interior nodes are operator)
- Interior node represents computed value
  - Identifier in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

Example: DAG for BB

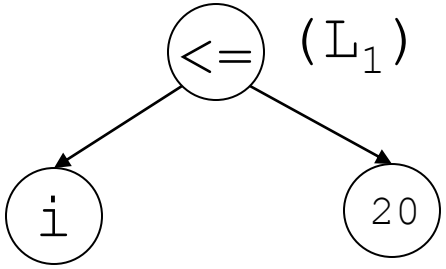
t_1 := 4 * i



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:

  For each 3AC, *A* in *B*

  A if 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*);

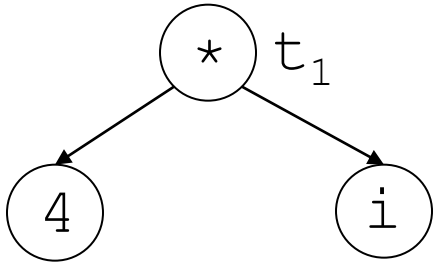
3. Remove x from Node(x).list

   Add *x* in n.list

   Node(*x*) = n;

Example: DAG construction from BB

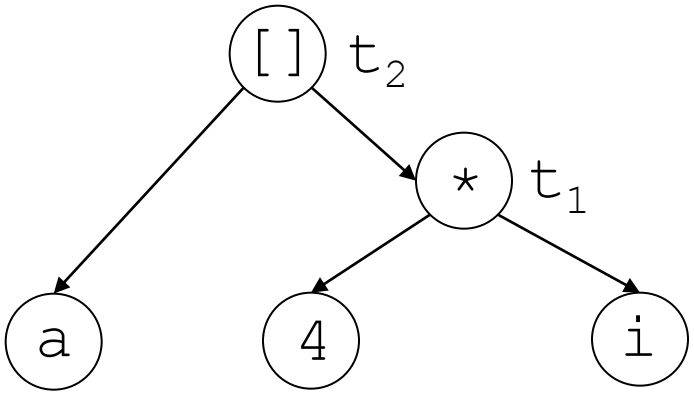
t_1 := 4 * i



Example: DAG construction from BB

t_1 := 4 * i

t_2 := a [t_1]

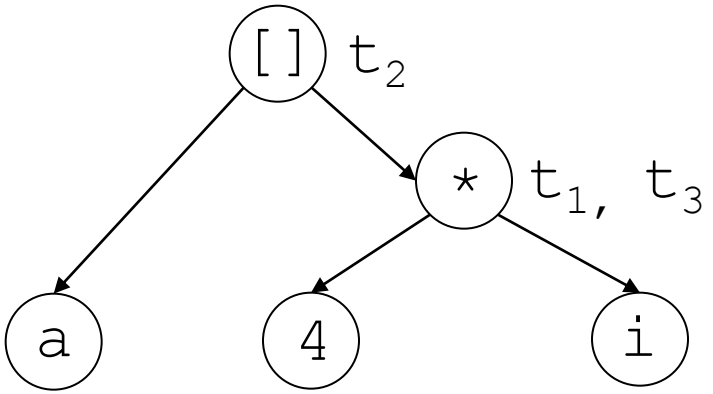


Example: DAG construction from BB

t_1 := 4 * i

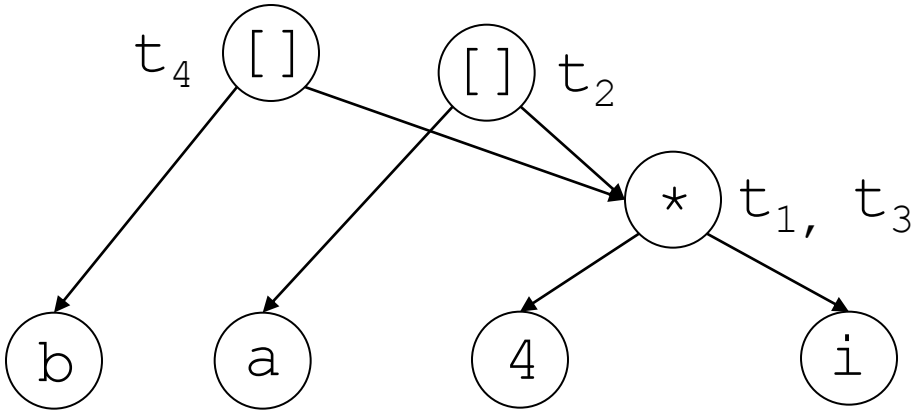
t_2 := a [t_1]

t_3 := 4 * i



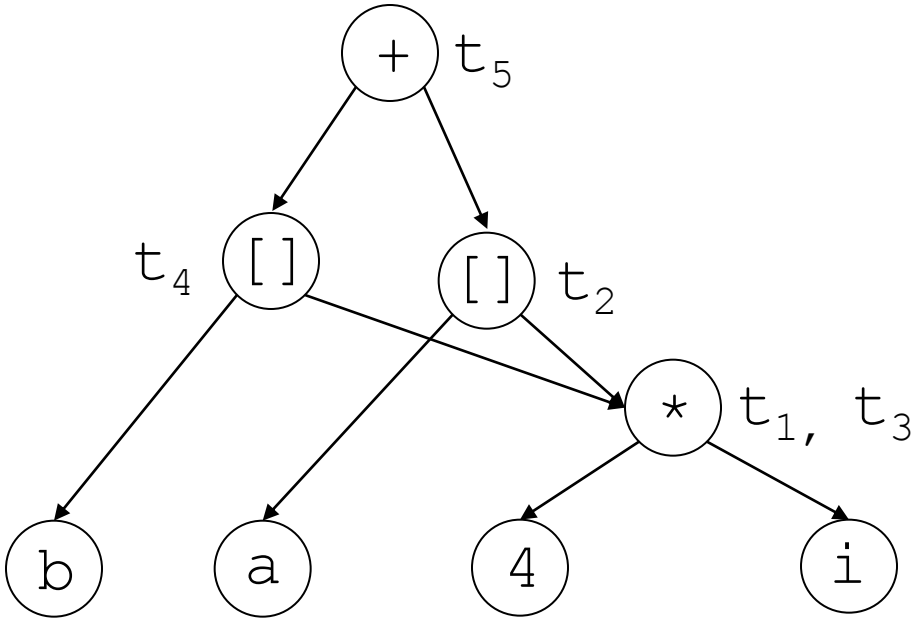
Example: DAG construction from BB

t_1 := 4 * i
t_2 := a [t_1]
t_3 := 4 * i
t_4 := b [t_3]



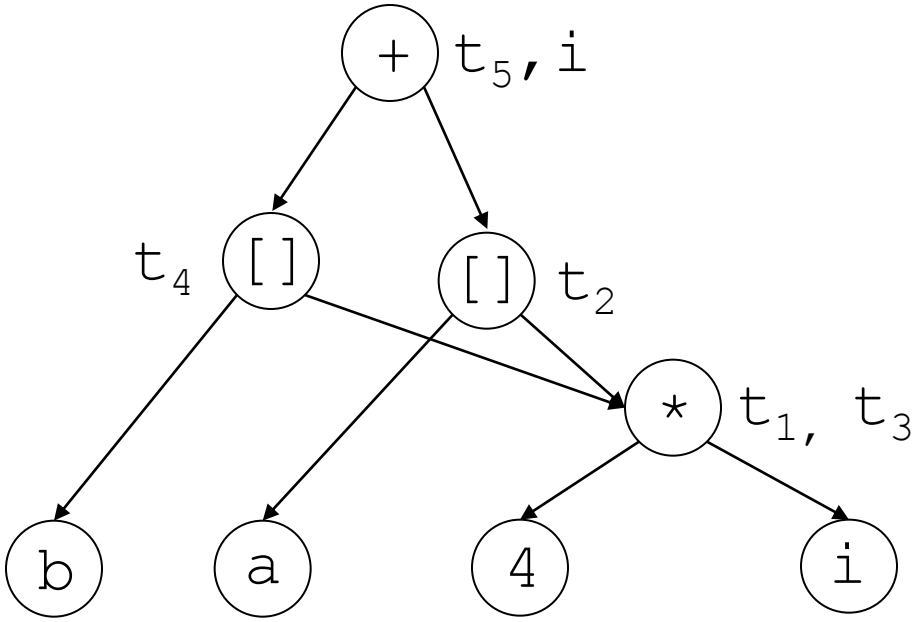
Example: DAG construction from BB

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

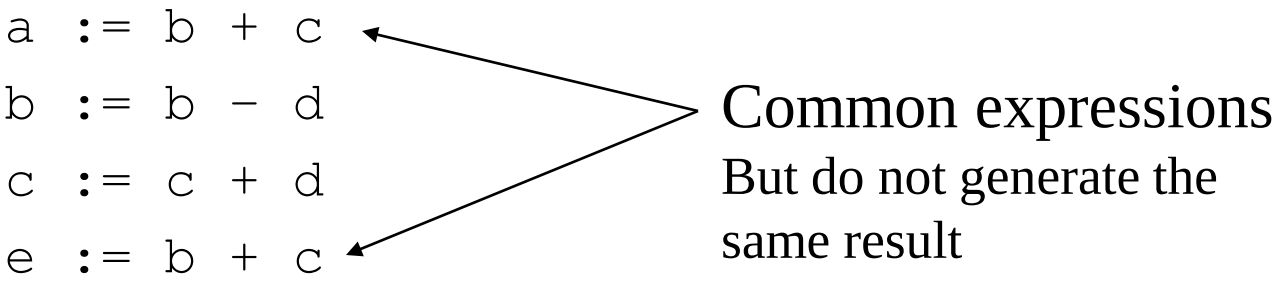


DAG of a Basic Block

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

Optimization of Basic Blocks

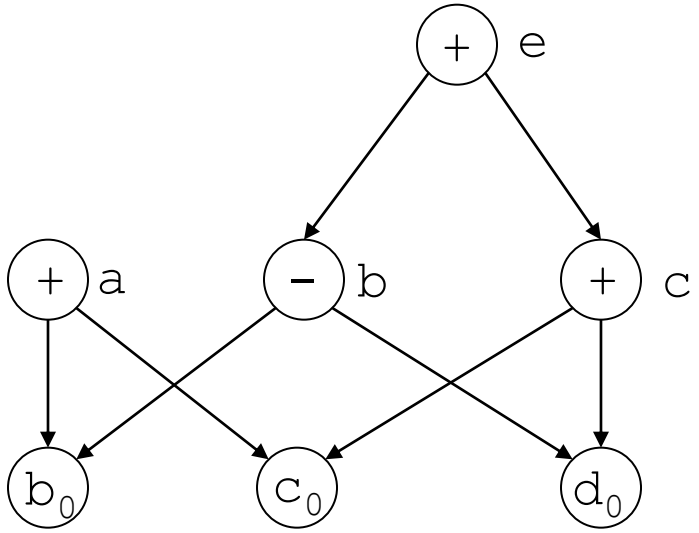
- Common sub-expression elimination: by construction of DAG
  - Note: for common sub-expression elimination, we are actually targeting for expressions that compute the same value.



Optimization of Basic Blocks

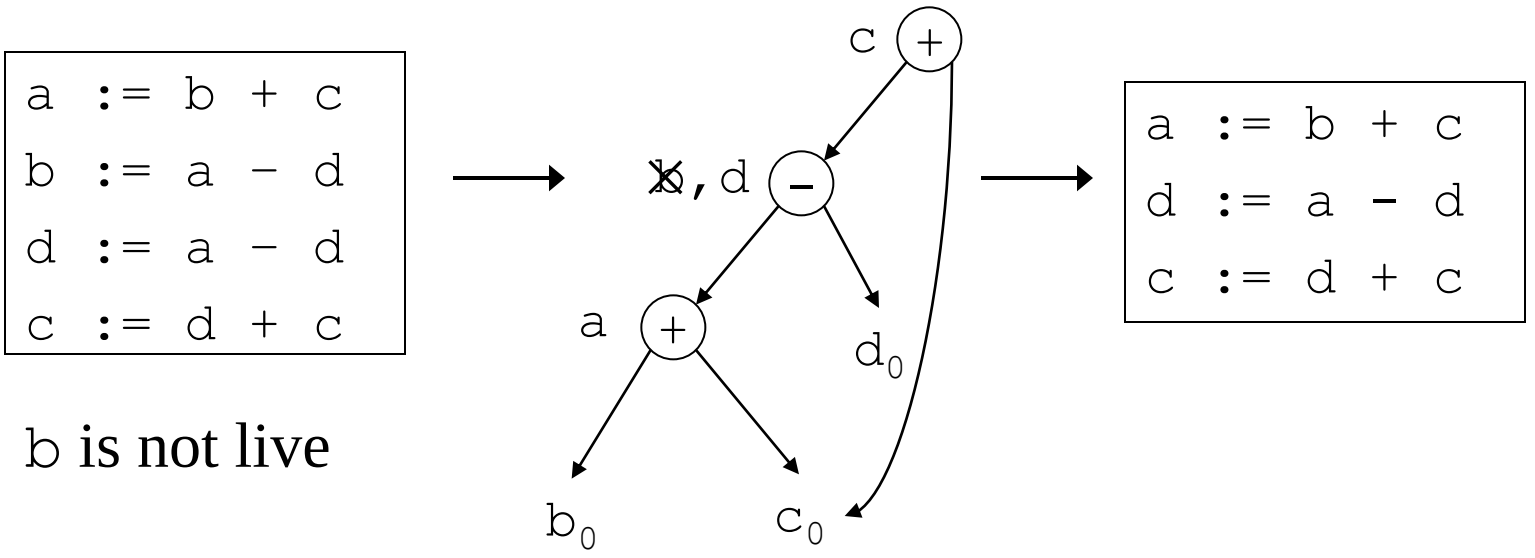
- DAG representation identifies expressions that yield the same result

```
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.



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*.

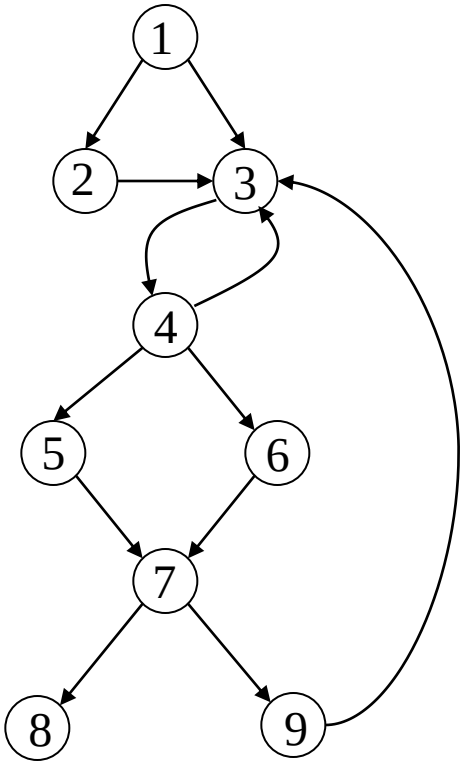
  $(d \neq n) \;\&\&\; (d \text{ dom } n) \rightarrow d \text{ dom } m$

- Dominator tree (*T*):

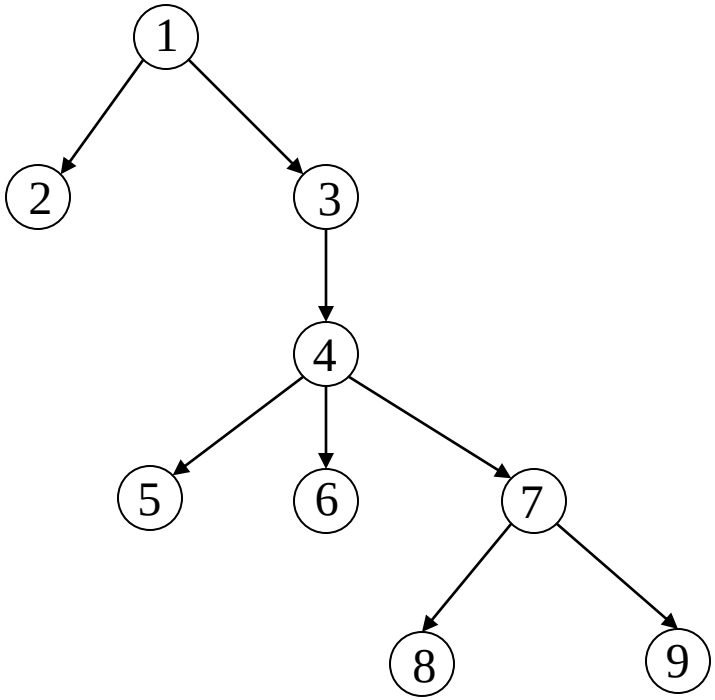
  A representation of dominator information of flow graph *G*.

  - The root node of *T* is the initial node of *G*
  - A node *d* in *T* dominates all node 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 node 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*

Loop Optimization

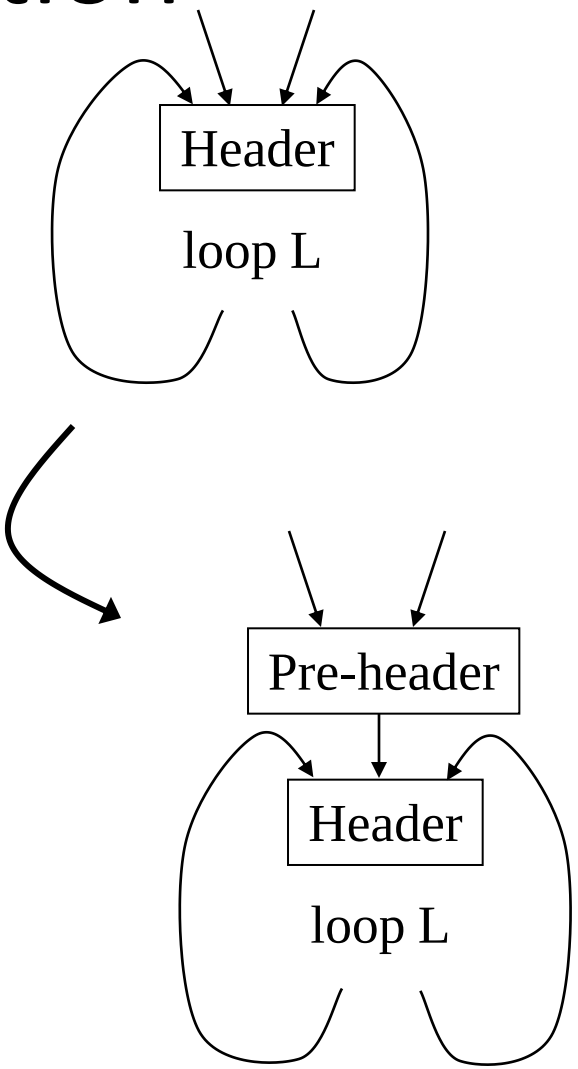
- **Loop interchange**: exchange inner loops with outer loops
- **Loop splitting**: attempts to simplify a loop or eliminate dependencies by breaking it into multiple loops which have the same bodies but iterate over different contiguous portions of the index range.
  - A useful special case is ***loop peeling*** - simplify a loop with a problematic first iteration by performing that iteration separately before entering the loop.

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 edge 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 writes to address(*x*) in the loop.

Loop Invariant Code Removal

- Loop invariant code removal can be done without available definition information.

Rules that need change:

- For all use of *x* is in the available definition set
- For all exit edges, if *x* is live on the exit edges, is in the available definition set on the exit edges

- Approx of First rule:
  - *d* dominates all uses of *x*
- Approx of Second rule
  - *d* dominates all exit basic blocks where *x* is live

Loop Induction Variable

- Induction variables are variables such that every time they change value, they are incremented or decremented.
  - Basic induction variable: induction variable whose only assignments within a loop are 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 = r7 + 3
        r7 = r7 + 1
        r10 = *r2
        r3 = *r4
        r9 = r1 * r3
        r10 = r9 >> 4
        *r2 = r10
        r1 = r1 + 4
        If(r1 < 100) goto Loop
```

Induction Variable Strength Reduction

- Create basic induction variables from derived induction variables.
- Rules: (S: *x* := *y* op *z*)
  - *op* is *, <<, +, or –
  - *y* is a induction variable
  - *z* is invariant
  - No other statement modifies *x*
  - *x* is not *y* or *z*
  - *x* is a register

Induction Variable Strength Reduction

- Transformation:

Insert the following into the bottom of pre-header:

new_reg = expression of target statement S

if (opcode(S)) is not add/sub, insert to the bottom of the preheader

new_inc = inc(y,op,z)

else

new_inc = inc(x)

Insert the following at each update of *y*

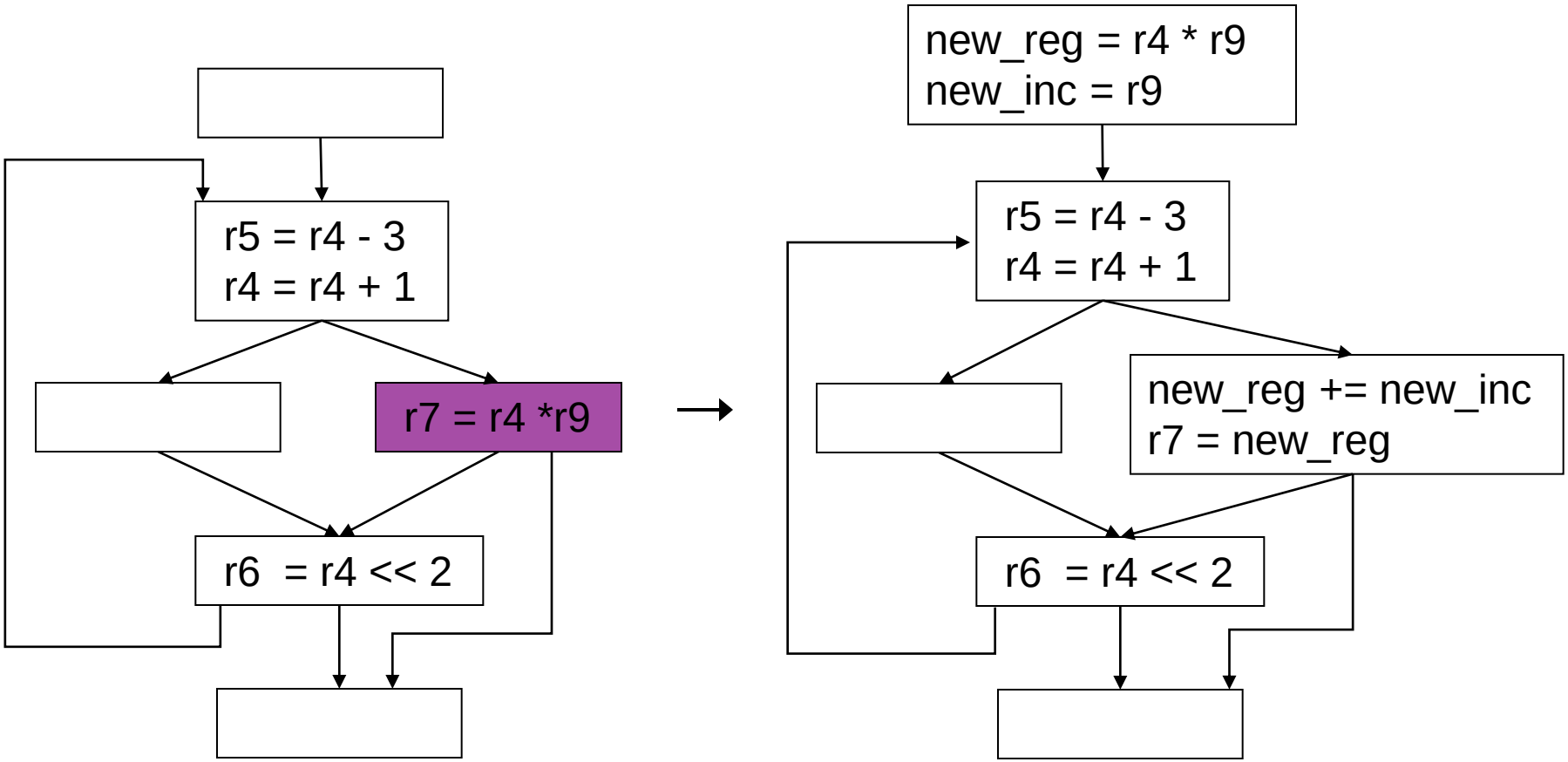
new_reg = new_reg + new_inc

Change S: *x = new_reg*

Function: inc()

Calculate the amount of inc for 1st param.

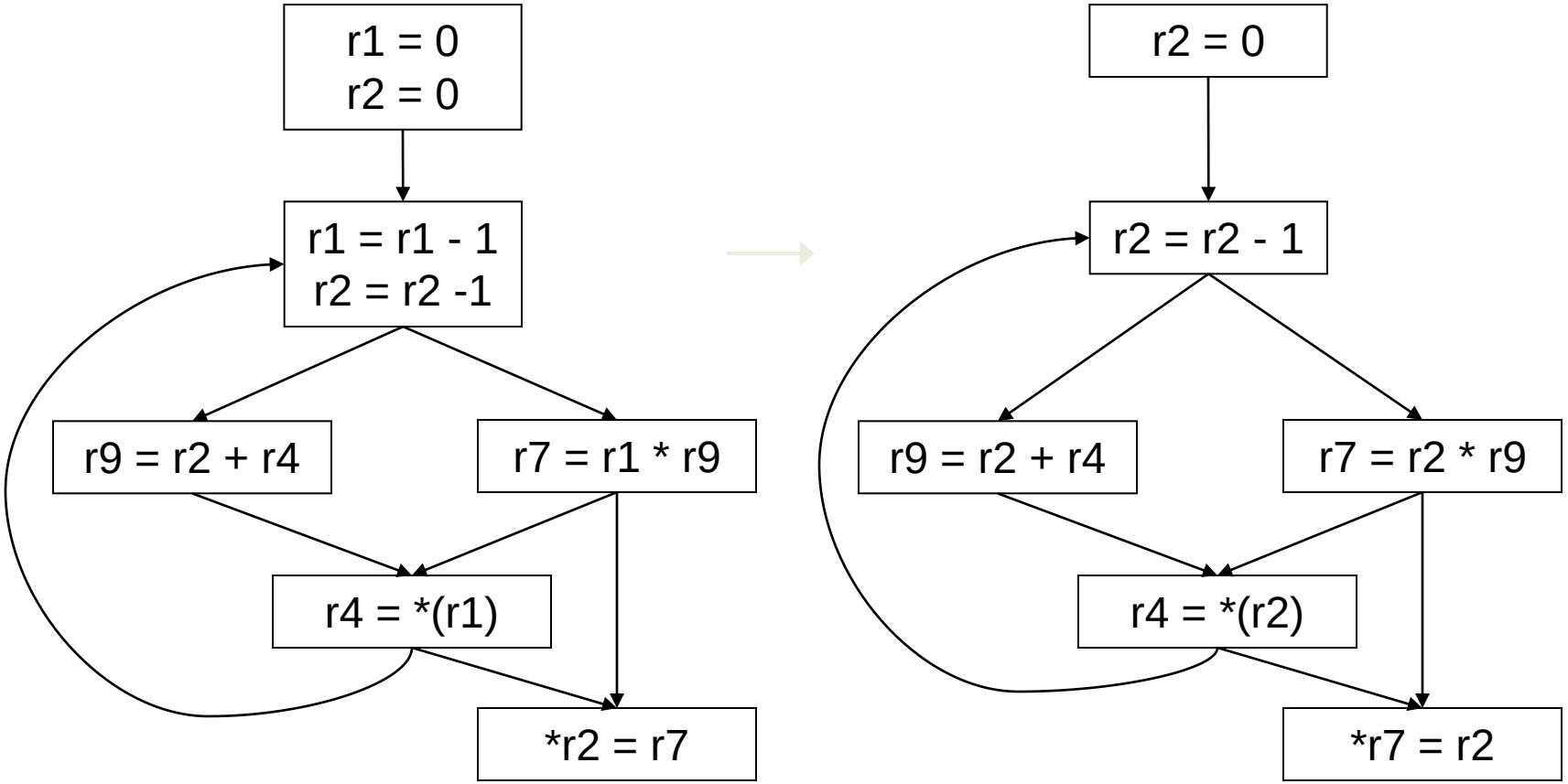
Example: Induction Variable Strength Reduction



Induction Variable Elimination

- Remove unnecessary basic induction variables from the loop by substituting uses with another basic induction variable.
- Rules:
  - Find two basic induction variables, *x* and *y*
  - *x* and *y* in the same family
    - Incremented at the same place
  - Increments are equal
  - Initial values are equal
  - *x* is not live at exit of loop
  - For each BB where *x* is defined, there is no use of x between the first and the last definition of *y*

Example: Induction Variable Elimination



Induction Variable Elimination

- Variants:

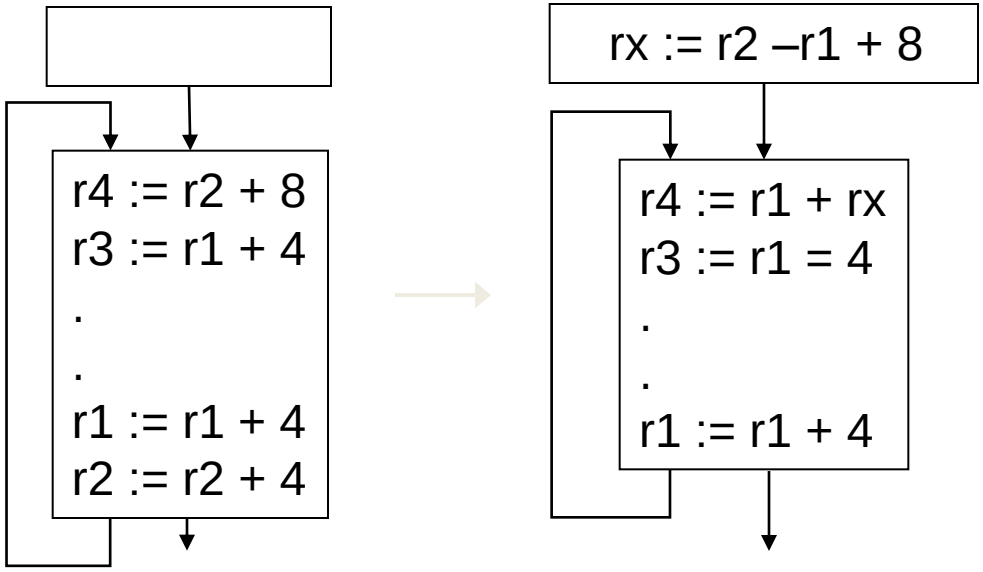
1. Trivial: induction variable that are never used except to increment themselves and not live at the exit of loop
2. Same increment, same initial value (discussed)
3. Same increment, initial values are a known constant offset from one another
4. Same increment, nothing known about the relation of initial value
5. Different increments, nothing known about the relation of initial value

- 1,2 are basically free
- 3-5 require complex pre-header operations

Example: Induction Variable Elimination

- Case 4: Same increment, unknown initial value

For the induction variable that we are eliminating, look at each non-incremental use, generate the same sequence of values as before. If that can be done without adding any extra statements in the loop body, then the transformation can be done.



Loop Unrolling

- Replicate the body of a loop (N-1) times, resulting in total N copies.
  - Enable overlap of operations from different iterations
  - Increase potential of instruction level parallelism (ILP)
- Variants:
  - Unroll multiple of known trip counts
  - Unroll with remainder loop
  - While loop unroll

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 and 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

- Algorithm sketch
  - Typically, a bit vector is used to store the information.
    - For example, in reaching definitions, each bit position corresponds to one definition.
  - We use an iterative fixed-point algorithm.
  - Depending on the nature of the problem we are solving, we may need to traverse each basic block in a forward (top-down) or backward direction.
    - The order in which we "visit" each BB is not important in terms of algorithm correctness, but is 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

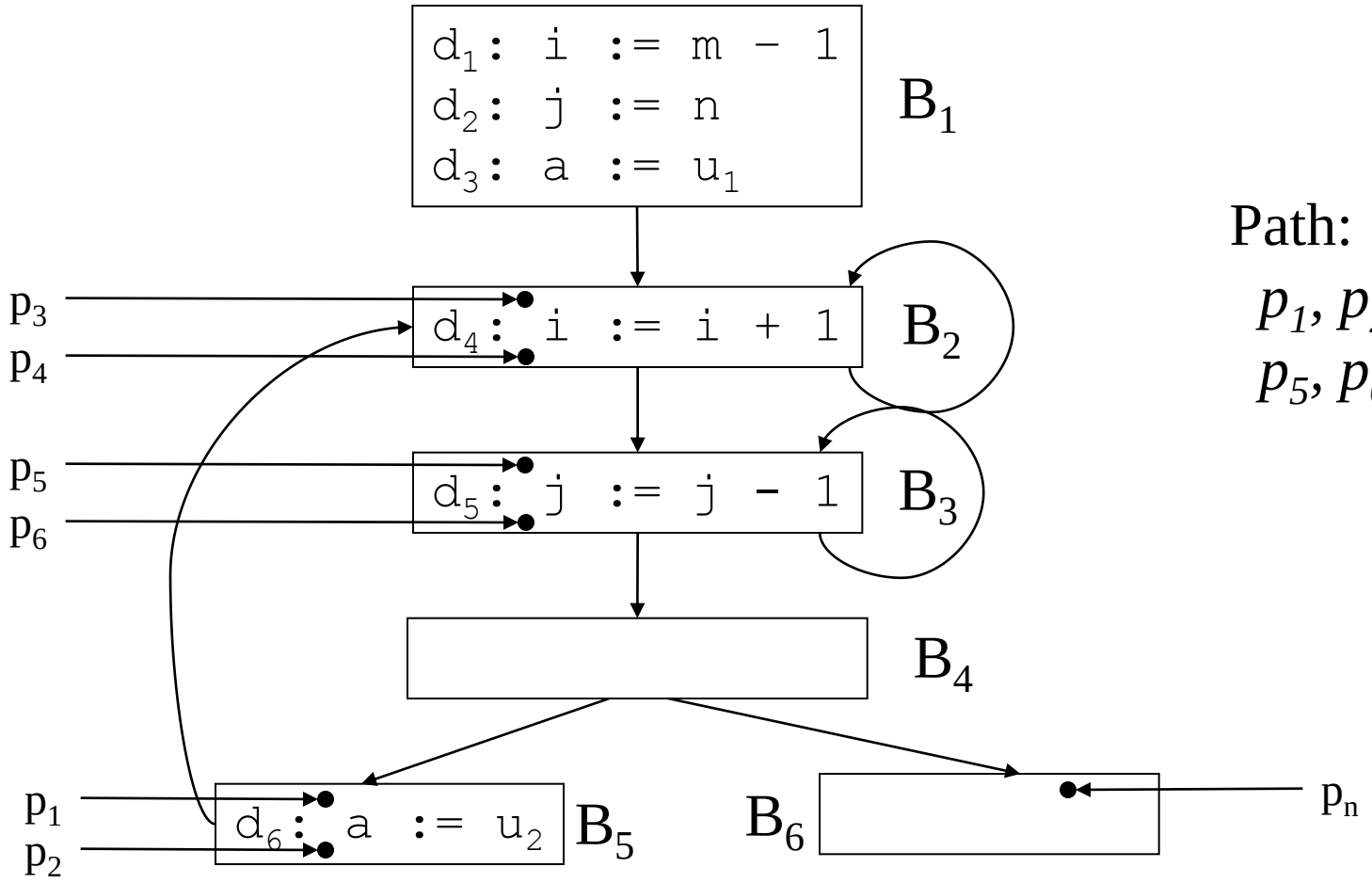
Global Data Flow Analysis

- The notion of *gen* and *kill* depends on the desired information.
- In some cases, *in* may be defined in terms of *out* - equation is 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, or
  - 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:

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

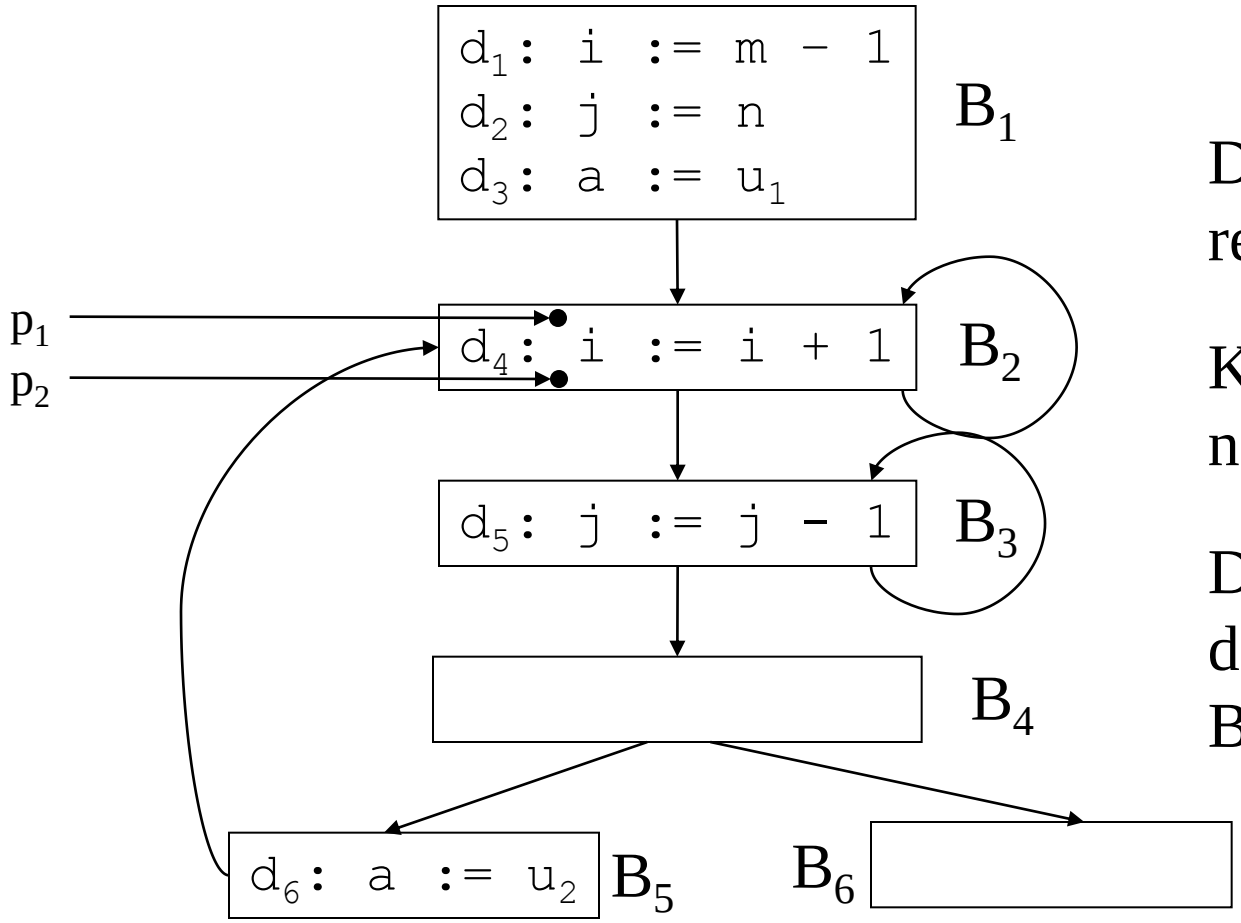
Reaching Definition

- 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 not 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_4, 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
```

The 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 we only consider while-do and if-then-else control constructs

    S → **id :=** E | S **;** S

    | **if** E **then** S **else** S | **do** S **while** E

    E → **id + id** | **id**

  The non-terminals represent *regions*.

Data Flow analysis of a Structured Program

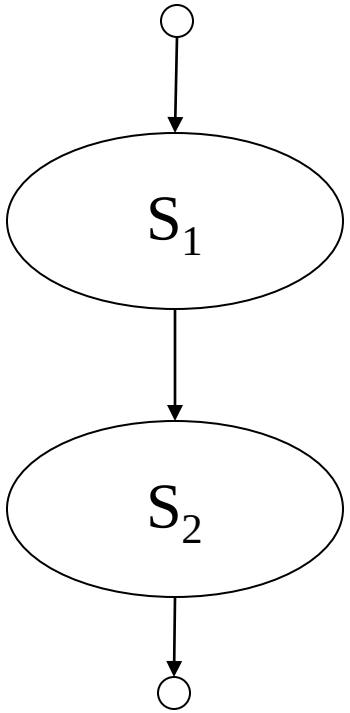
- Region: A graph *G'= (N',E')* which is portion of the control flow graph *G*.
  - The set of nodes *N'* is in *G'* such that
    - *N'* includes a header *h*
    - *h* dominates all node in *N'*
  - The set of edges *E'* is in *G'* such that
    - All edges $a \rightarrow b$ such that *a,b* are in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

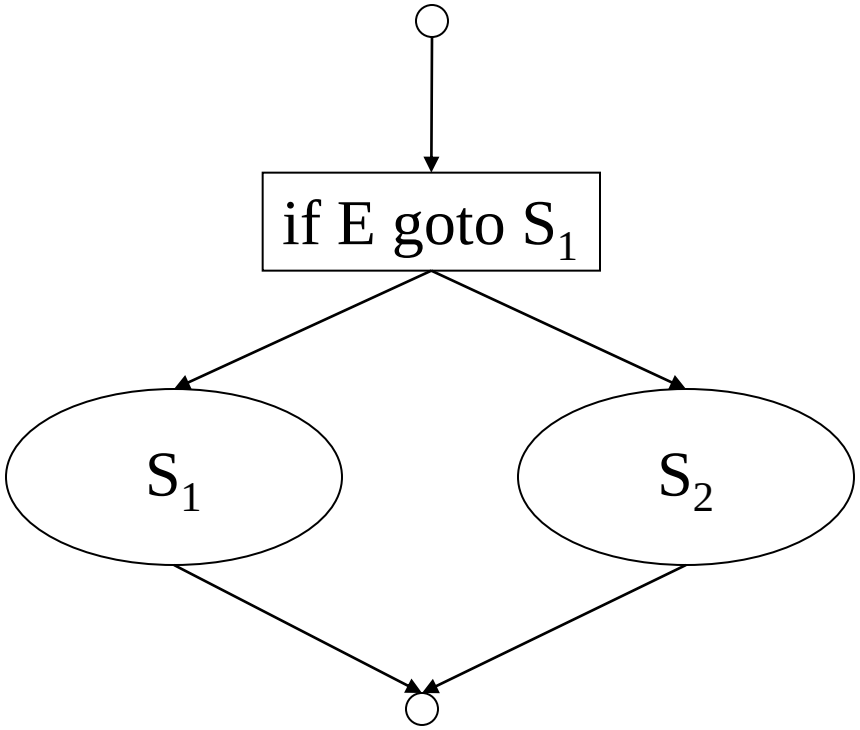
Data Flow analysis of a Structured Program:
Composition of Regions

$S \rightarrow S_1 ; S_2$



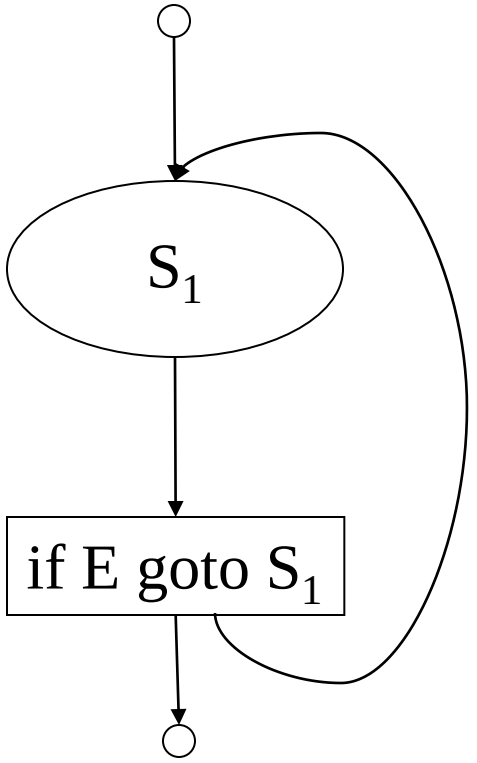
Data Flow analysis of a Structured Program:
Composition of Regions

$S \rightarrow$ **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 (or NT) has four attributes:
  - *gen*[S]: Set of definitions generated by the 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 S must have a definition for a (where *a* is defined by *d)*.

Data Flow Equations

  - *in*[S]: The set of definition those are live at the entry point of block S.
  - *out*[S]: The set of definition those are live at the exit point of block S.
- The data flow equations are inductive or syntax directed.
  - *gen* and *kill* are synthesized attributes.
  - *in* is an inherited attribute.

Data Flow Equations

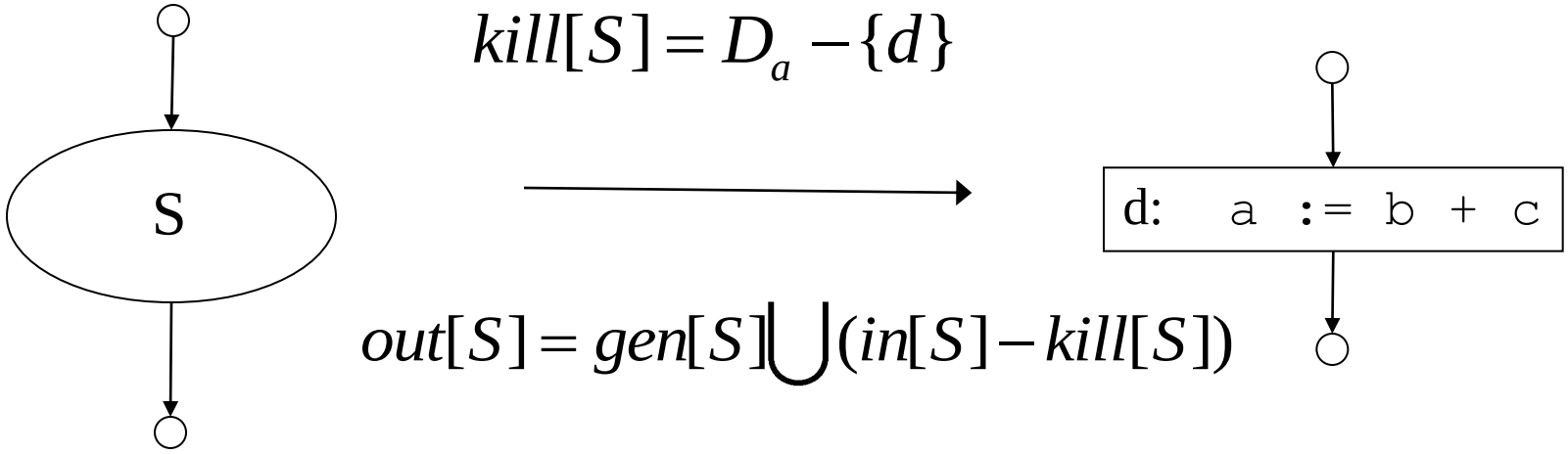
- *gen*[S] concerns with a single basic block. It is the set of definitions in S that reaches the end of S.
- In contrast *out*[S] is the set of definitions (possibly defined in some other block) live at the end of S considering all paths through S.

Data Flow Equations

Single statement

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

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



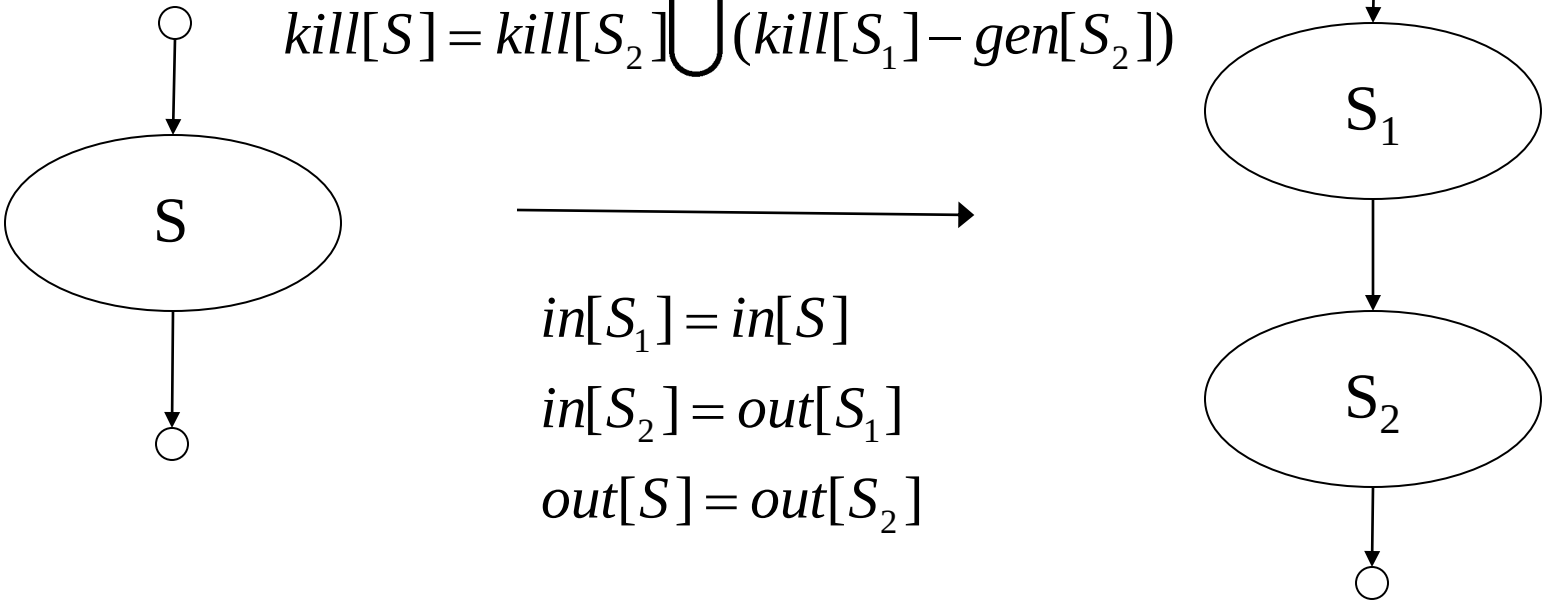
$$out[S] = gen[S] \bigcup (in[S] - kill[S])$$

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

Data Flow Equations
Composition

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

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



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

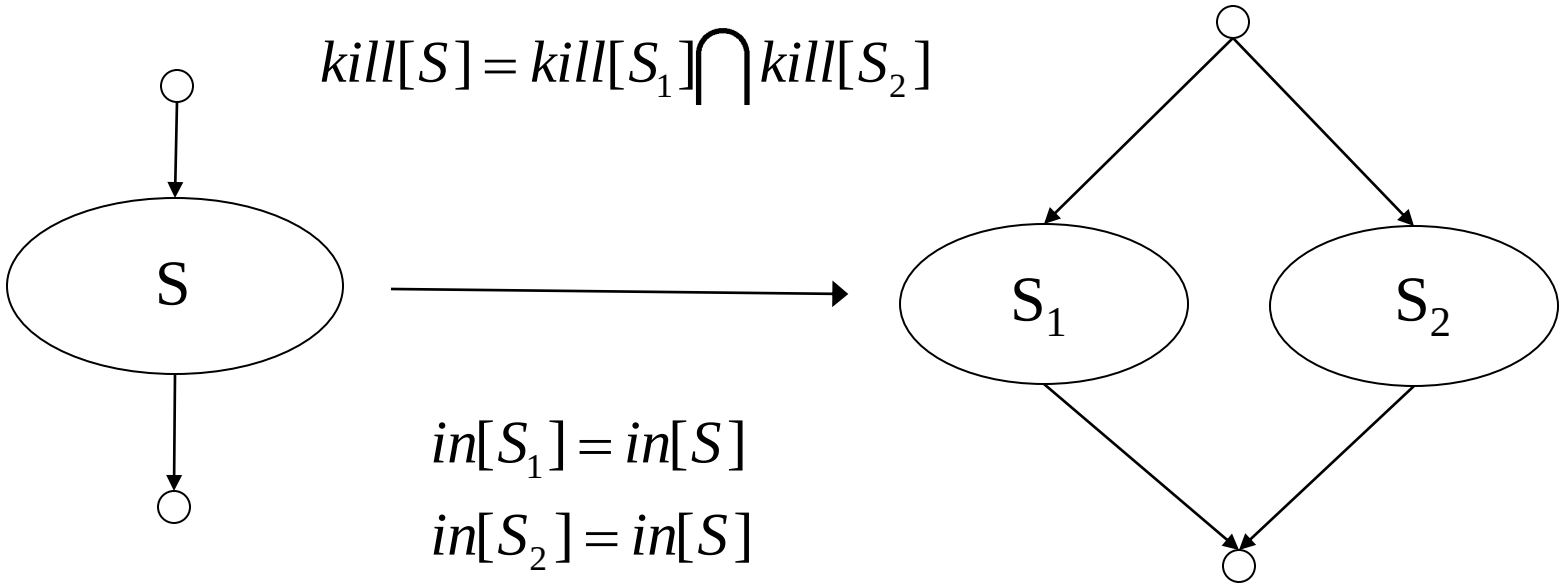
$$in[S_2] = out[S_1]$$

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

Data Flow Equations
if-then-else

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

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



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

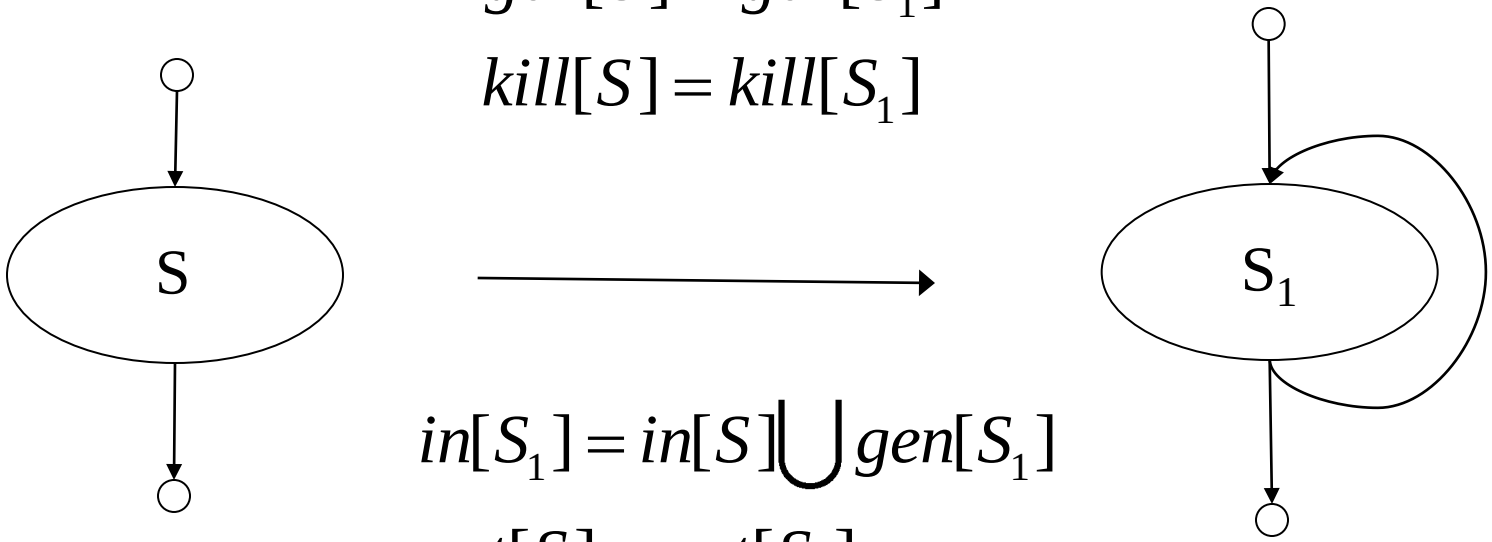
$$in[S_2] = in[S]$$

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

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

- The attributes are computed for each region. The 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
    - In 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.
- The equation is solved iteratively.
- The general equations for *in* and *out*:

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

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

Reaching definitions

- What is safe?
  - To assume that a definition reaches a point even if it turns out not to.
  - The computed set of definitions reaching a point *p* will be a superset of the actual set of definitions reaching *p*
  - Goal : make the set of reaching definitions as small as possible (i.e. as close to the actual set as possible)

Reaching definitions

- How are the **gen** and **kill** sets defined?
  - **gen**[B] = {definitions that appear in B and reach the end of B}
  - **kill**[B] = {all definitions that never reach the end of B}
- What is the direction of the analysis?
  - forward
  - **out**[B] = **gen**[B] $\cup$ (**in**[B] - **kill**[B])

Reaching definitions

- What is the **confluence** operator?
  - union
  - **in**[B] = $\cup$ **out**[P], over the predecessors P of B
- How do we initialize?
  - start small
    - Why? Because we want the resulting set to be as small as possible
  - for each block B initialize **out**[B] = **gen**[B]

Computation of *gen* and *kill* sets

for each basic block BB do
  gen(BB) = $\varnothing$; *kill*(BB) = $\varnothing$;
  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      in(BB) = ∅
for all basic blocks BB    out(BB) = gen(BB)
change = true
while (change) do
   change = false
   for each basic block BB, do
      old_out = out(BB)
      in(BB) = ∪(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
```

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:

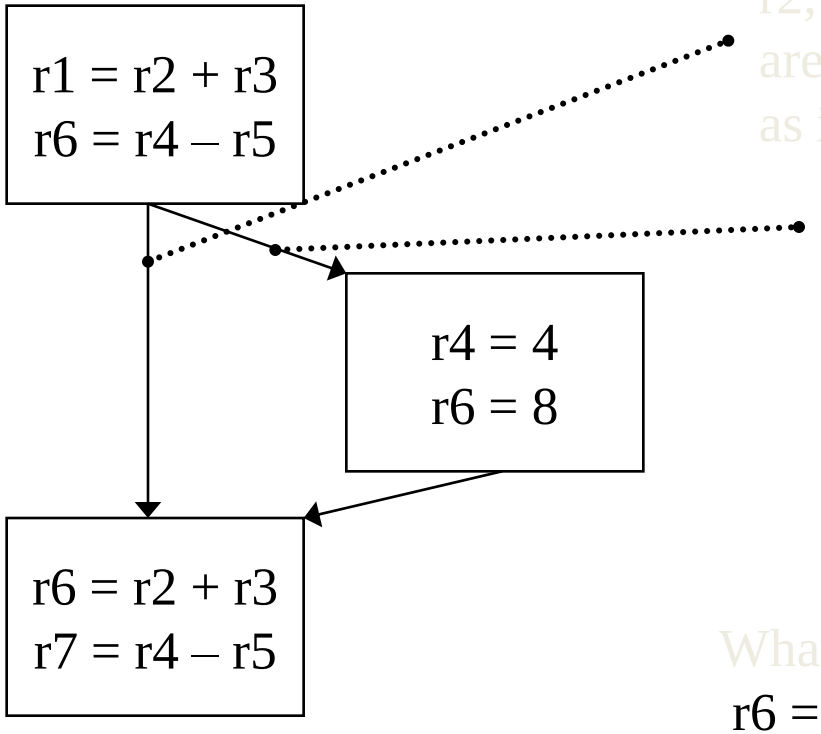
$$in[B] = use[B] \bigcup (out[B] - def[B])$$

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

  - 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) = $\varnothing$; *use*(BB) = $\varnothing$;
  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) $\cup$ {*y*};
    endfor
    def(BB) = *def*(BB) $\cup$ {*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

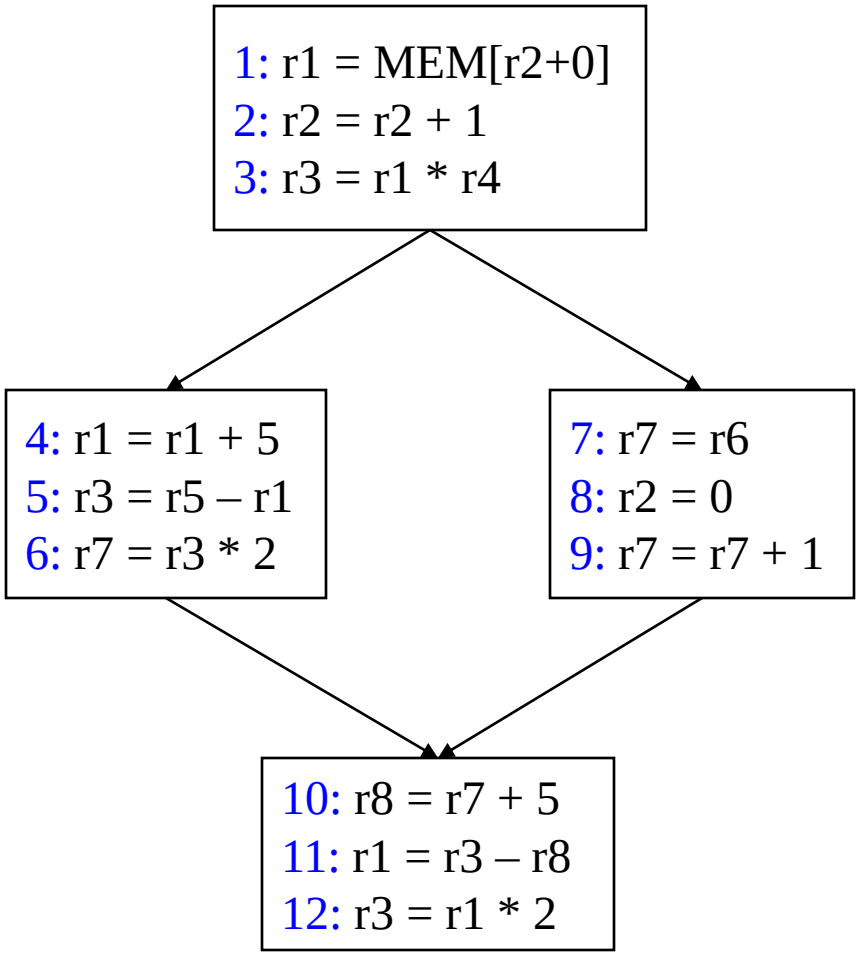
for all basic blocks BB
    in(BB) = $\varnothing$;

change = true;
while (change) do
    change = false
    for each basic block BB do
        old_in = *in*(BB);
        out(BB) = $\bigcup$ {*in*(Y): for all successors Y of BB}
        in(X) = *use*(X) $\bigcup$ (*out*(X) – *def*(X))
        if (*old_in* != *in*(X)) then change = true
    endfor
endfor

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

DU Chain of r3:

(3) -> 11

(5) -> 11

(12) ->

UD Chain of r1:

(12) -> 11

UD Chain of r7:

(10) -> 6,9

Some-things to Think About

- Liveness and Reaching definitions are basically the same thing!
  - All dataflow is basically the same with a few parameters
    - Meaning of gen/kill (use/def)
    - Backward / Forward
    - All paths / some paths (must/may)
      - So far, we have looked at may analysis algorithms
      - How do you adjust to do must algorithms?
- Dataflow can be slow
  - How to implement it efficiently?
  - How to represent the info?

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] = ∪ *out[P]* : *P* is pred of *BB* forward analysis

    out[BB] = ∪ *in[P]* : *P* is succ of *BB* backward analysis

Generalized Dataflow Algorithm

```
change = true;
while (change)
    change = false;
    for each BB
        apply meet function
        apply transfer function
        if any changes → change = true;
```

Example: Liveness by upward exposed uses

for each basic block BB, do

$gen[BB] = \varnothing$

$kill[BB] = \varnothing$

for each operation (x := y op z) in reverse order in BB, do

$gen[BB] = gen[BB] - \{x\}$

$kill[BB] = kill[BB] \bigcup \{x\}$

for each source operand of op, y, do

$gen[BB] = gen[BB] \bigcup \{y\}$

$kill[BB] = kill[BB] - \{y\}$

endfor

endfor

endfor

Beyond Upward Exposed Uses

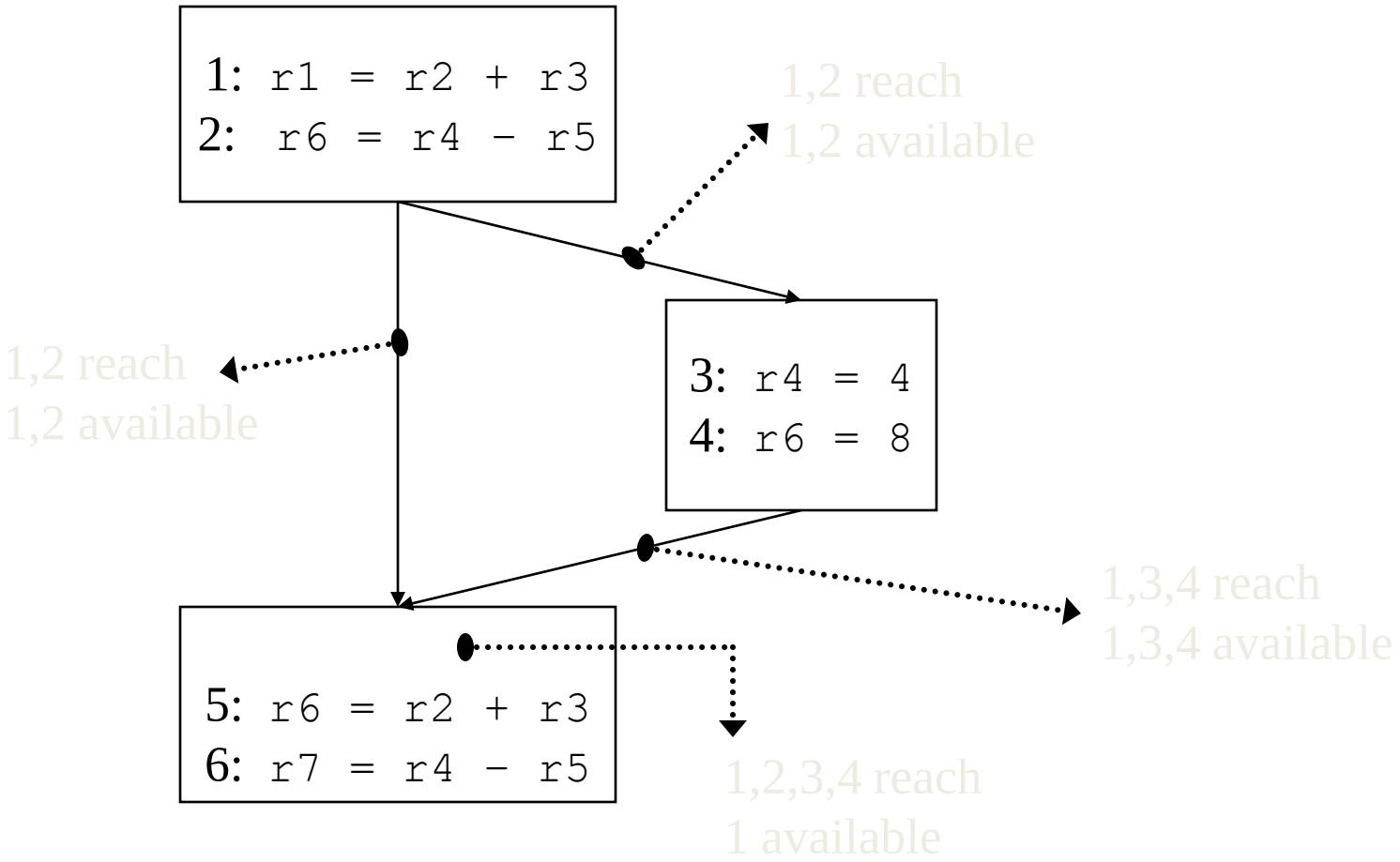
- Upward exposed defs
  - *in* = *gen* + (*out* – *kill*)
  - *out* = ∪(*in*(succ))
  - Walk ops reverse order
    - *gen* += {dest} kill += {dest}
- Downward exposed uses
  - *in* = ∪(*out*(pred))
  - *out* = *gen* + (*in* - *kill*)
  - Walk in forward order
    - *gen* += {src}; *kill* -= {src};
    - *gen* -= {dest}; *kill* += {dest};

- Downward exposed defs
  - *in* = ∪(*out*(pred))
  - *out* = *gen* + (*in* - *kill*)
  - Walk in forward order
    - *gen* += {dest}; *kill* += {dest};

All Path Problem

- Up to this point
  - Any path problems (maybe relations)
    - Definition reaches along some path
    - 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
  - Can always count on this
  - 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* is the tricky part

Compute Adef *gen/kill* Sets

for each basic block BB do
  gen(BB) = $\varnothing$; *kill*(BB) = $\varnothing$;
  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(X) + (IN(X) – KILL(X))
    if (*old_out* != *out*(X)) then change = true
  endfor
endfor

Available Expression Analysis (Aexprs)

- An *expression* is a RHS of an operation
  - 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 are 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 Expression

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

```
foreach 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
- Bitvectors
  - General strategy used by most compilers
  - Bit positions represent defs (rdefs)
  - Efficient set operations: union/intersect/isone
  - 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
     - Take 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

Constant Folding

- Simplify operation based on values of target operand
  - Constant propagation creates opportunities for this
- All constant operands
  - Evaluate the op, replace with a move
    - r1 = 3 * 4 → r1 = 12
    - r1 = 3 / 0 → ??? Don't evaluate excepting ops !, what about FP?
  - Evaluate conditional branch, replace with BRU or noop
    - if (1 < 2) goto BB2 → goto BB2
    - if (1 > 2) goto BB2 → convert to a noop Dead code
- Algebraic identities
  - r1 = r2 + 0, r2 – 0, r2 | 0, r2 ^ 0, r2 << 0, r2 >> 0 → r1 = r2
  - r1 = 0 * r2, 0 / r2, 0 & r2 → r1 = 0
  - r1 = r2 * 1, r2 / 1 → r1 = r2

Strength Reduction

- Replace expensive ops with cheaper ones
  - Constant propagation creates opportunities for this
- Power of 2 constants
  - Mult by power of 2: r1 = r2 * 8 → r1 = r2 << 3
  - Div by power of 2: r1 = r2 / 4 → r1 = r2 >> 2
  - Rem by power of 2: r1 = r2 % 16 → r1 = r2 & 15
- More exotic
  - Replace multiply by constant by sequence of shift and adds/subs
    - r1 = r2 * 6
      - r100 = r2 << 2; r101 = r2 << 1; r1 = r100 + r101
    - r1 = r2 * 7
      - r100 = r2 << 3; r1 = r100 – r2

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

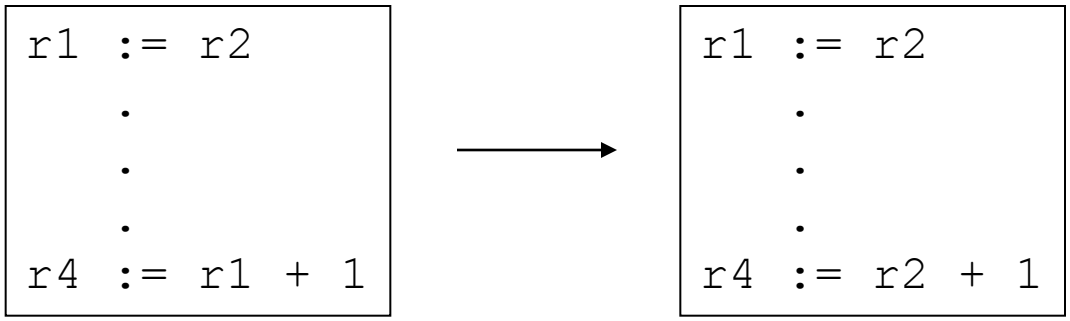
- Forward propagation of moves/assignment of the form

  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.



  - 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

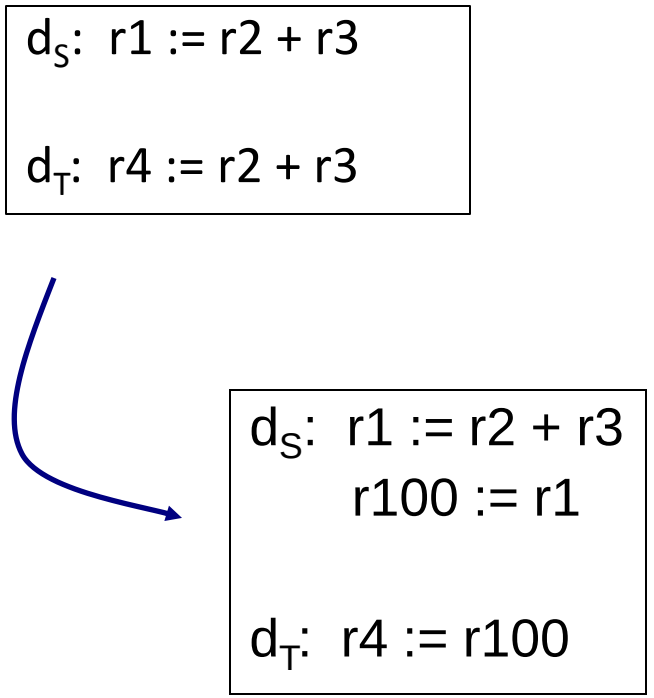
  r5 := r1 + r6 → r5 := r4 + r6

  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_1 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 has 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

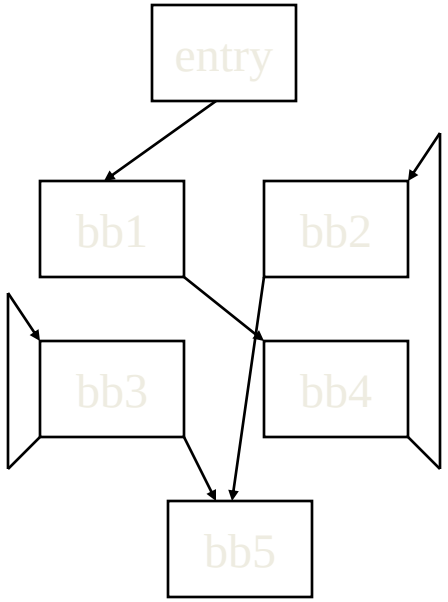


Global Common Sub-Expression Elimination

- Rules:
  - d_S and d_T has 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?