Code Optimization

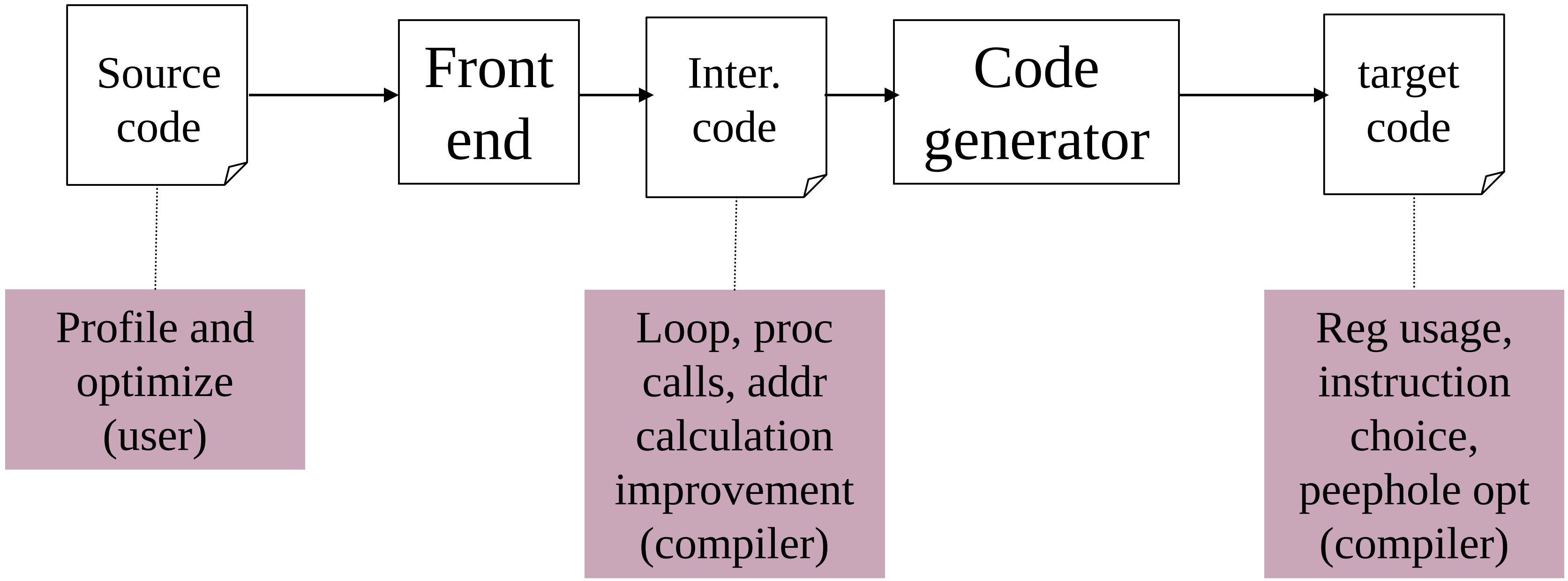
Introduction

- 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

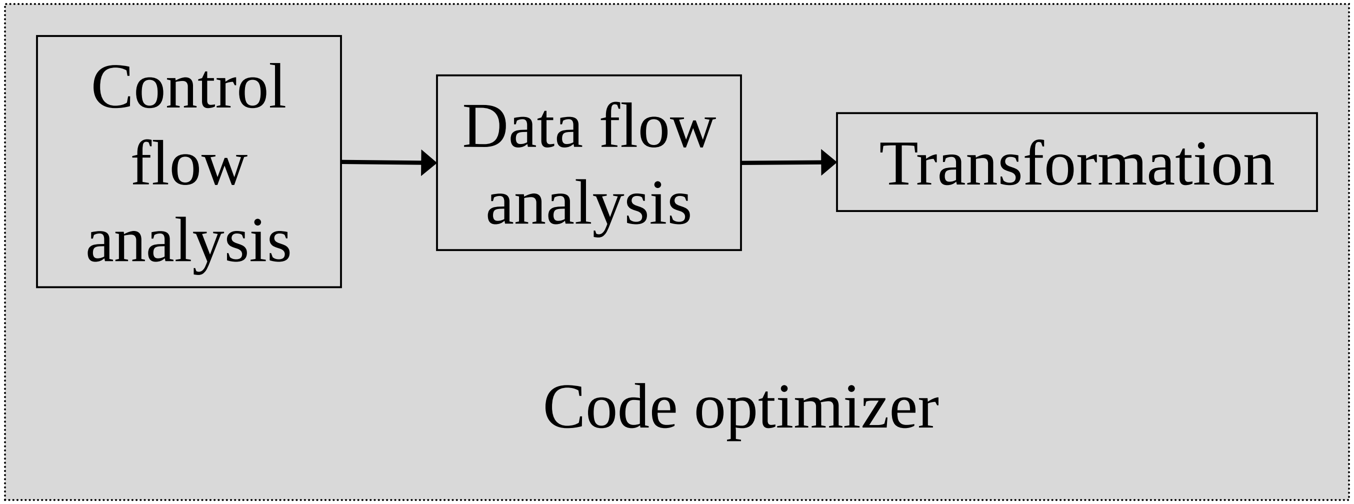
Introduction

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



Introduction

- Organization of an optimizing compiler



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

Basic Blocks

- A **basic block** is a maximal sequence of consecutive three-address instructions with the following properties:
  - The flow of control can only enter the basic block thru the 1st instr.
  - Control will leave the block without halting or branching, except possibly at the last instr.
- Basic blocks become the nodes of a **flow graph**, with edges indicating the order.

Examples

```
for i from 1 to 10 do
  for j from 1 to 10 do
     a[i,j]=0.0

for i from 1 to 10 do
     a[i,i]=0.0
```

```
1)   i = 1
2)   j = 1
3)   t1 = 10 * i
4)   t2 = t1 + j
5)   t3 = 8 * t2
6)   t4 = t3 - 88
7)   a[t4] = 0.0
8)   j = j + 1
9)   if j <= 10 goto (3)
10)  i = i + 1
11)  if i <= 10 goto (2)
12)  i = 1
13)  t5 = i - 1
14)  t6 = 88 * t5
15)  a[t6] = 1.0
16)  i = i + 1
17)  if i <= 10 goto (13)
```

Identifying Basic Blocks

- Input: sequence of instructions *instr(i)*
- Output: A list of basic blocks
- Method:
  - Identify **leaders**:
    the first instruction of a basic block
  - Iterate: add subsequent instructions to basic block until we reach another leader
  - Any instruction that is the target of a (conditional or unconditional) jump is a leader
  - Any instruction that immediately follow a (conditional or unconditional) jump is a leader

Basic Block Example

1.	i = 1	A
2.	j = 1	B
3.	t1 = 10 * i	C
4.	t2 = t1 + j	
5.	t3 = 8 * t2	
6.	t4 = t3 - 88	
7.	a[t4] = 0.0	
8.	j = j + 1	
9.	if j <= 10 goto (3)	
10.	i = i + 1	D
11.	if i <= 10 goto (2)	
12.	i = 1	E
13.	t5 = i - 1	F
14.	t6 = 88 * t5	
15.	a[t6] = 1.0	
16.	i = i + 1	
17.	if i <= 10 goto (13)	

Leaders

Basic Blocks

Control-Flow Graphs

- Node: an instruction or sequence of instructions (a basic block)
  - Two instructions i, j in same basic block

    iff execution of i *guarantees* execution of j
- Directed edge: *potential* flow of control
- Distinguished start node *Entry & Exit*
  - First & last instruction in program

Control-Flow Edges

- Basic blocks = nodes
- Edges:
  - Add directed edge between B1 and B2 if:
    - Branch from last statement of B1 to first statement of B2 (B2 is a leader), or
    - B2 immediately follows B1 in program order and B1 does not end with unconditional branch (goto)
  - Definition of predecessor and successor
    - B1 is a predecessor of B2
    - B2 is a successor of B1

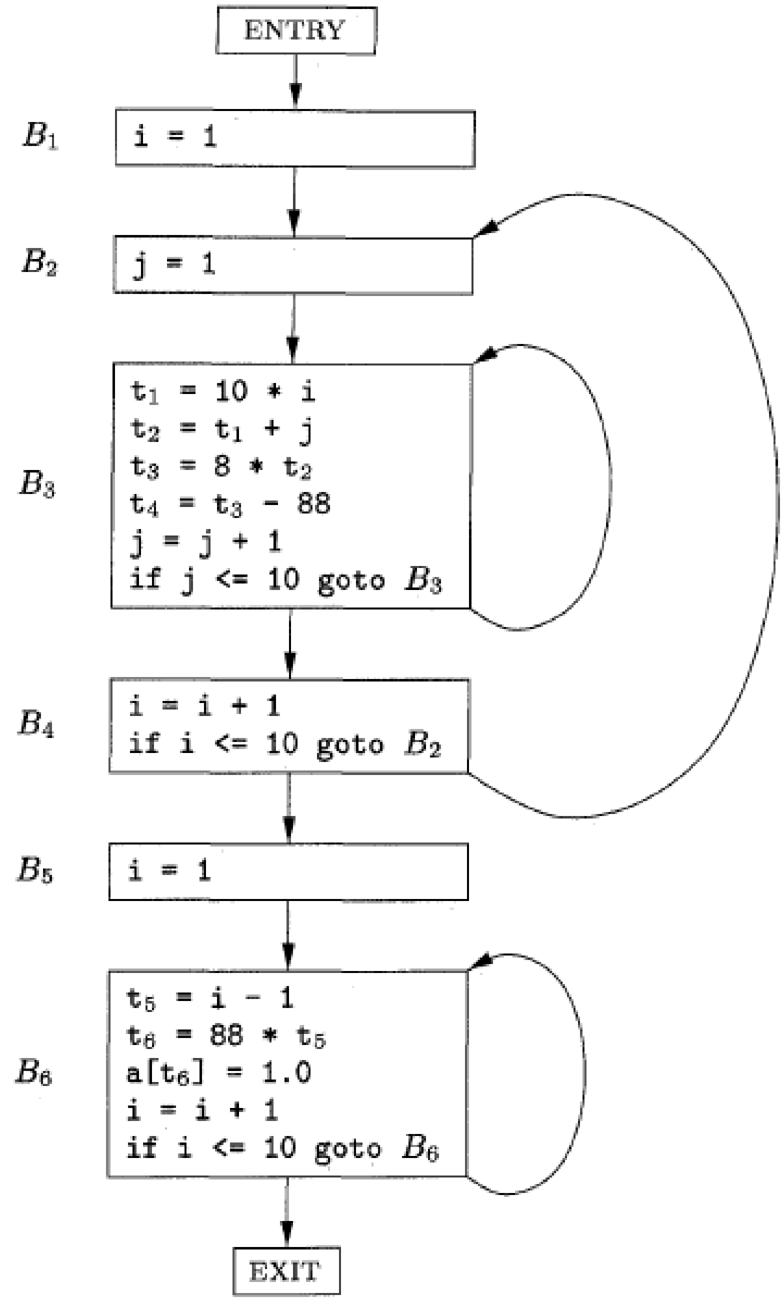
Control-Flow Edge Algorithm

Input: block(i), sequence of basic blocks

Output: CFG where nodes are basic blocks

```
for i = 1 to the number of blocks
   x = last instruction of block(i)
   if instr(x) is a branch
         for each target y of instr(x),
               create edge (i -> y)
   if instr(x) is not unconditional branch,
         create edge (i -> i+1)
```

CFG Example

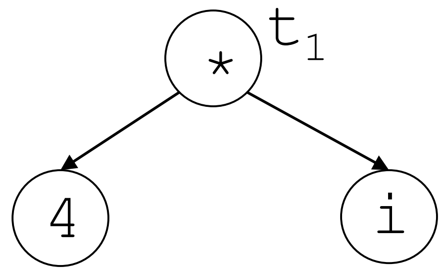

ENTRY
B_1
i = 1
B_2
j = 1
B_3
t_1 = 10 * i
t_2 = t_1 + j
t_3 = 8 * t_2
t_4 = t_3 - 88
j = j + 1
if j <= 10 goto B_3
B_4
i = i + 1
if i <= 10 goto B_2
B_5
i = 1
B_6
t_5 = i - 1
t_6 = 88 * t_5
a[t_6] = 1.0
i = i + 1
if i <= 10 goto B_6
EXIT

Optimization of Basic Blocks

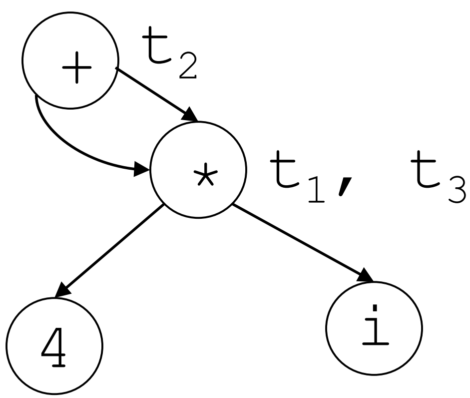
- Many structure preserving transformations can be implemented by construction of DAGs of basic blocks
- Leaves are labeled with unique identifier (var name or const)
- Interior nodes are labeled by an operator symbol
- Nodes optionally have a list of labels (identifiers)
- Edges relates operands to the operator (interior nodes are operator)
- Interior node represents computed value
    - Identifier in the label are deemed to hold the value

DAG for BB

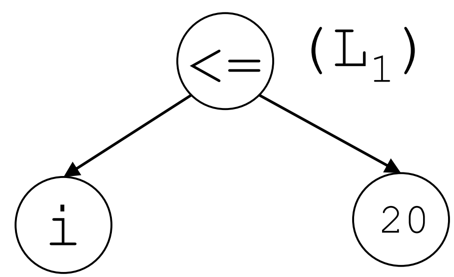
$t_1 := 4 * i$



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



if (i <= 20)goto L_1



Construction of DAGs for BB

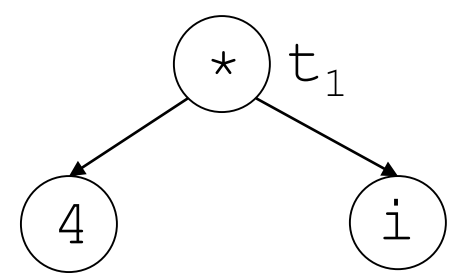
- I/p: Basic block, *B*
- O/p: A DAG for *B* containing the following information:
  1. A label for each node
  2. For leaves the labels are ids or consts
  3. For interior nodes the labels are operators
  4. For each node a list of attached ids (possible empty list, no consts)

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.

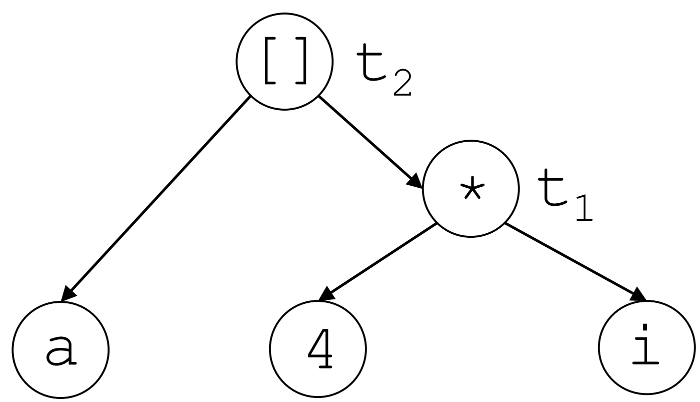
Example: DAG construction from BB

t_1 := 4 * i



Example: DAG construction from BB

```
t1 := 4 * i
t2 := a [ t1 ]
```

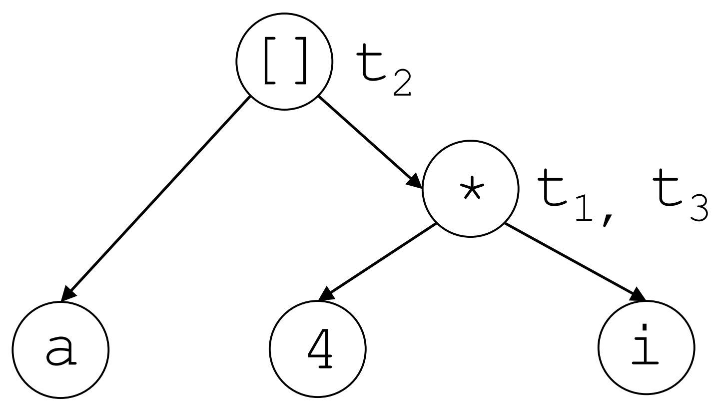


Example: DAG construction from BB

t_1 := 4 * i

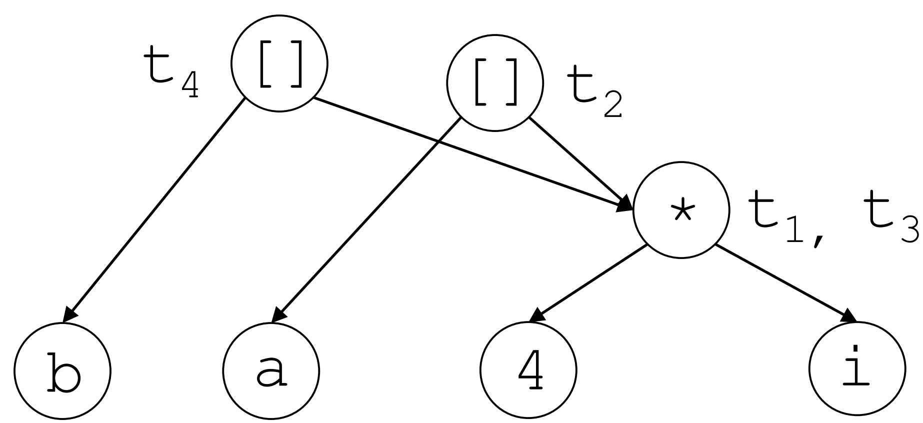
t_2 := a [t_1]

t_3 := 4 * i



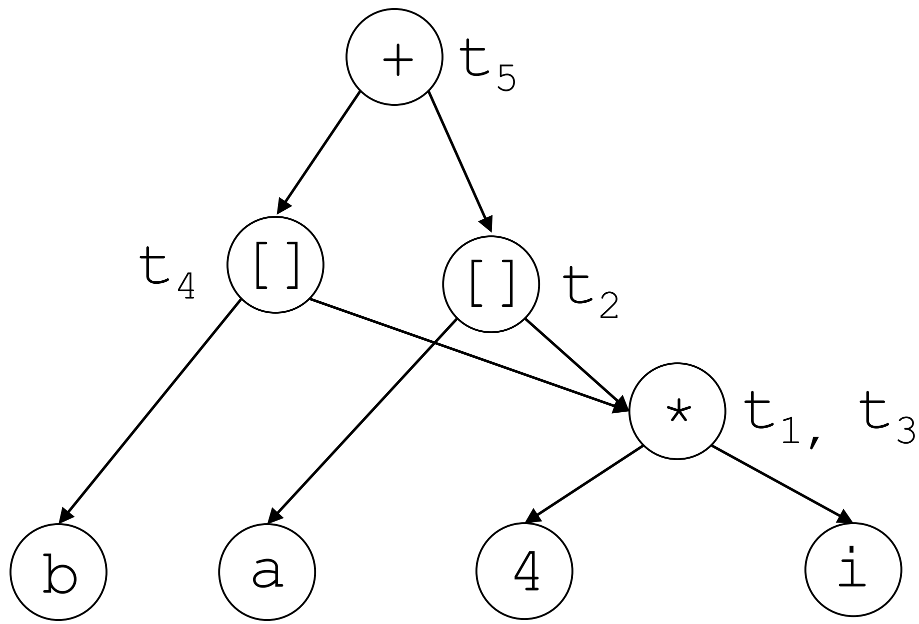
Example: DAG construction from BB

```
t1 := 4 * i
t2 := a [ t1 ]
t3 := 4 * i
t4 := b [ t3 ]
```



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

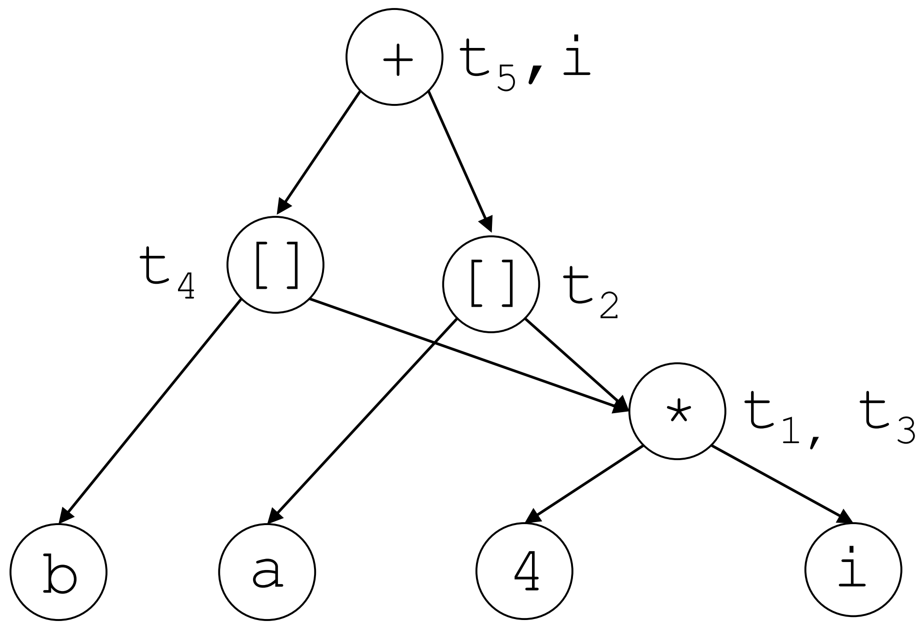
t_2 := a [t_1]

t_3 := 4 * i

t_4 := b [t_3]

t_5 := t_2 + t_4

i := t_5



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.

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

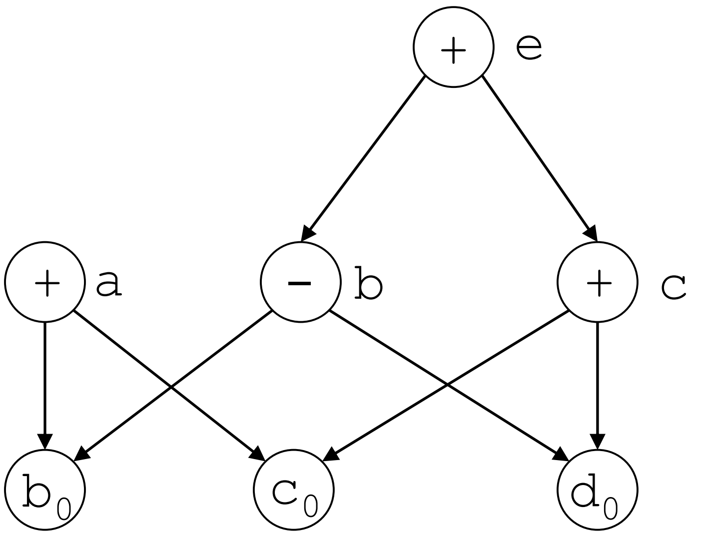
Common expressions

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
```

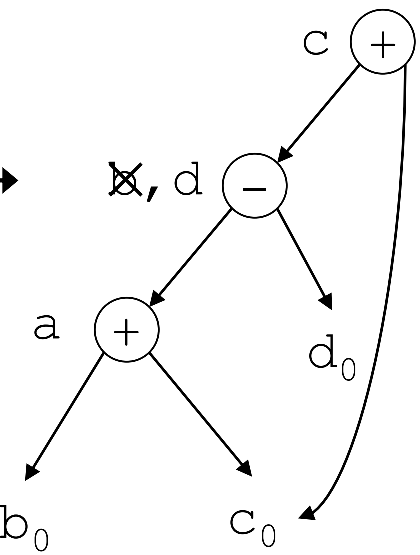


Optimization of Basic Blocks

- 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
```

Principle sources of optimization and transformations

1. Compile time evaluation
   a) Constant folding
   b) Constant propagation
2. Common sub-expression elimination
3. Code motion
4. Strength reduction
5. Dead code elimination
6. Copy propagation
7. Loop optimization

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.

1. Compile-Time Evaluation

- Expressions whose values can be pre-computed at the compilation time

1. **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

1. Compile-Time Evaluation

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

- Example:

  pi := 3.14286

  area = pi * r ** 2

  ↓

  area = 3.14286 * r ** 2

2. Common Sub-expression Elimination

- 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

3. 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
  - Similar to common sub-expression elimination but with the objective to reduce code size.

Example(Reduce Size): Code hoisting

		temp : = x ** 2
if (a< b) then		if (a< b) then
z := x ** 2		z := temp
else	$\longrightarrow$	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.

3. Code Motion

Execution frequency reduction: reduce execution frequency of partially available expressions (expressions available at least in one path)

Example:

`` if (a<b) then ```		``` if (a<b) then ```

```
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;
```

3. 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 ) …
```

3. 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

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

4. Strength Reduction

- Replacement of an operator with a less costly one.

Example:

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

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

5. 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

5. Dead Code Elimination

- Examples:

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

Can be eliminated

6. Copy Propagation

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

7. 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

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*

Loops in Flow Graph - Dominators

- We say that a node d in a flow graph dominates node n, written d *dom* n, if every path from the initial node of the flow graph to n goes through d.
- Initial node is the root, and each node dominates only its descendants in the tree (including itself)
- The node x strictly dominates y, if x dominates y and x≠y
- X is the immediate dominator of y(denoted idom(y)), if x is the closest strict dominator of y
- A dominator tree shows all the immediate dominator relationships
- Principle of the dominator algorithm
  - If p1,p2,...,pk, are all the predecessors of n, and d ≠ n, then d *dom* n, iff d *dom* p_i for each i

Loops in Flow Graph

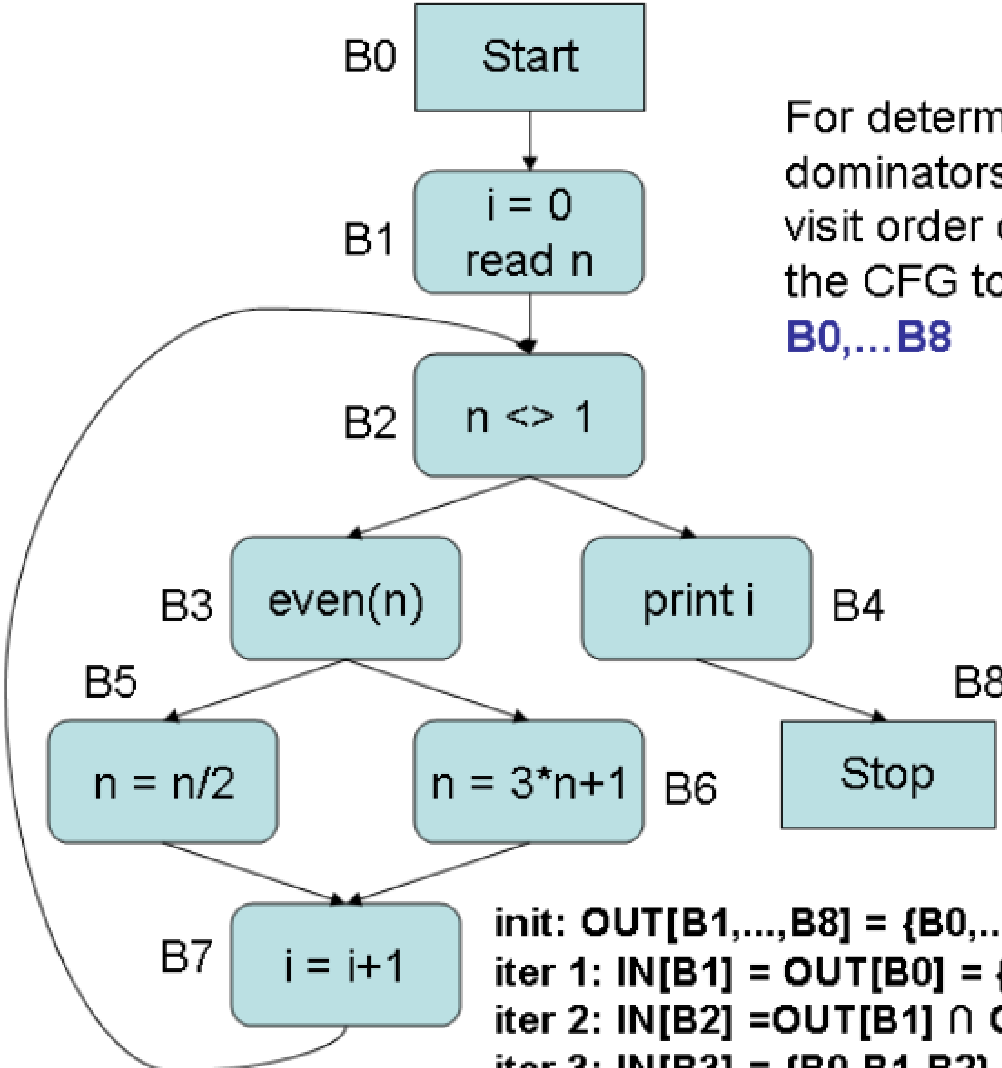
- Each node *n* has a unique *immediate dominator m*, which is the last dominator of *n* on any path in *G* from the initial node to *n*.

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

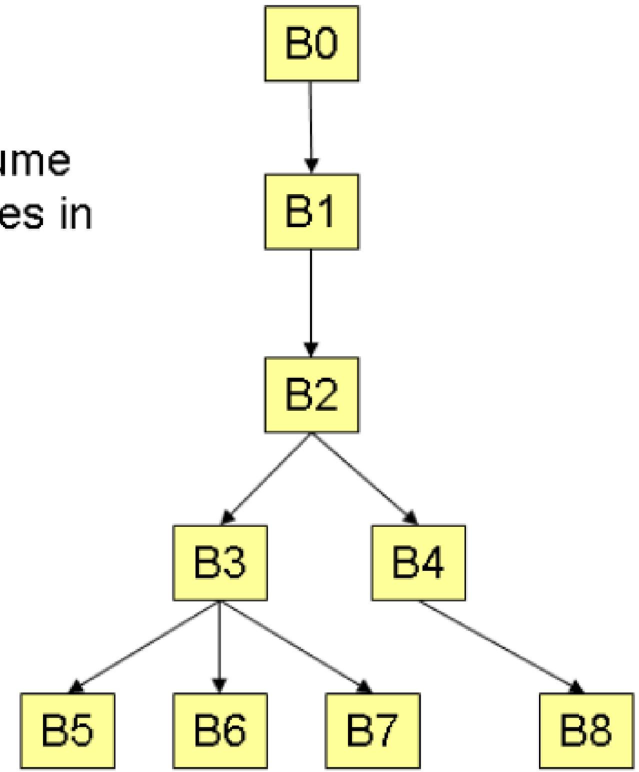
- Dominator tree (*T*):

  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

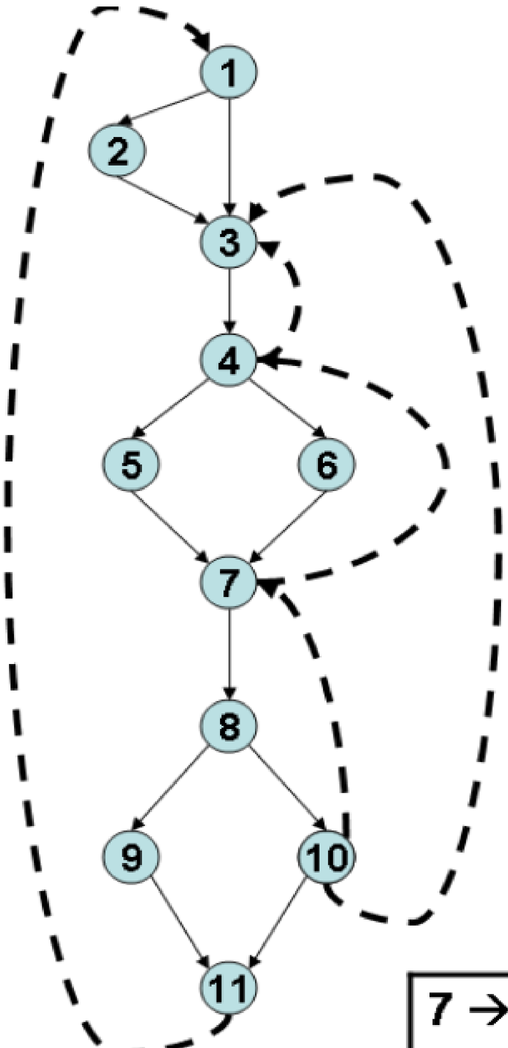


For determining dominators, assume visit order of nodes in the CFG to be **B0,…B8**

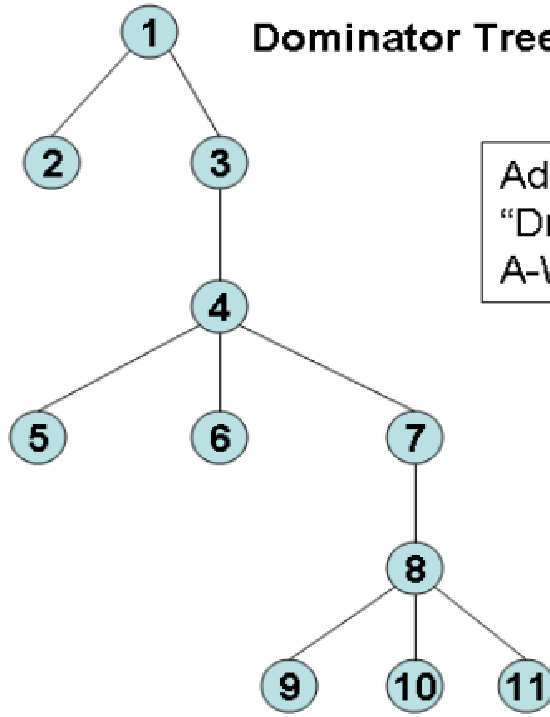


init: OUT[B1,...,B8] = {B0,...,B8}, OUT[B0] = {B0}
iter 1: IN[B1] = OUT[B0] = {B0}, OUT[B1] = {B0,B1}
iter 2: IN[B2] =OUT[B1] ∩ OUT[B7] = {B0,B1} , OUT[B2] = {B0,B1,B2}
iter 3: IN[B3] = {B0,B1,B2}, OUT[B3] = {B0,B1,B2,B3}
IN[B4] = {B0,B1,B2}, OUT[B4] = {B0,B1,B2,B4}
iter 4: IN[B5] = {B0,B1,B2,B3} = IN[B6], OUT[B5] = {B0,B1,B2,B3,B5}
OUT[B6] = {B0,B1,B2,B3,B6}, OUT[B8] = {B0,B1,B2,B4,B8}
iter 5: IN[B7] = OUT[B5] ∩ OUT[B6] = {B0,B1,B2,B3}
OUT[B7] = {B0,B1,B2,B3,B7}

Dominators, Back edges and Natural loops



Flow Graph



Dominator Tree

Adapted from the "Dragon Book", A-W, 1986

Back edges and their natural loops

7 → 4	10→7	4→3	10→3	11→1
{4,5,6,7,8, 10}	{7,8,10}	{3,4,5,6,7, 8,10}	{3,4,5,6,7, 8,10}	{1,2,3,4,5, 6,7,8,9,10,11}

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 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

Example:

```
for i = 1 to N
    x = x + 1
    for j = 1 to N
        a(i,j) = 100*N + 10*i + j + x
```

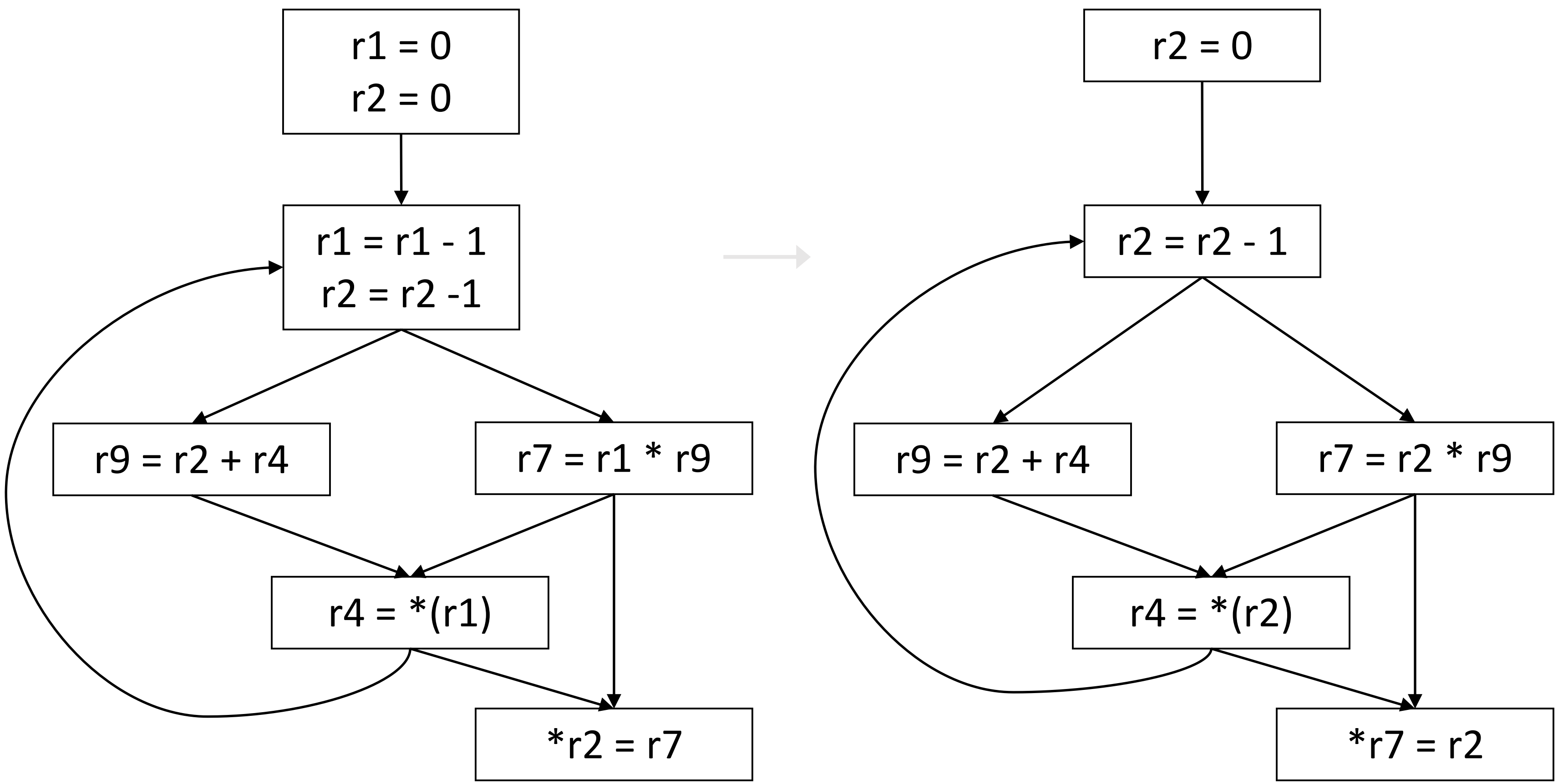
→

```
t1 = 100*N
for i = 1 to N
        x = x + 1
        t2 = 10*i + x
        for j = 1 to N
                a(i,j) = t1 + t2 + j
```

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



Loop Unrolling

- The overhead of the loop control code can be reduced by executing more than one iteration in the body of the loop. *E.g.*

```
double picosy = Math.PI * Math.cos(y);
for (int i = 0; i < x.length; i++)
    x[i] *= picosy;
```

↓

```
double picosy = Math.PI * Math.cos(y);
for (int i = 0; i < x.length; i += 2) {
    x[i] *= picosy;
    x[i+1] *= picosy;
}
```

8. Peephole optimization

- Optimizing technique that operates on the target code considering few instructions at a time.
- Do machine dependent improvements
- Peeps into a single or sequence of two to three instructions and replace it by most efficient alternatives

8.1 Redundant instruction elimination

- Redundant load/store: see if an obvious replacement is possible

```
MOV  R0, a
MOV a, R0
```

Can eliminate the second instruction without needing any global knowledge of ***a***

- Unreachable code: identify code which will never be executed:

#define DEBUG 0		
if(DEBUG) {		if (0 != 1) goto L2
print debugging info	→	print debugging info
}		
		L2:

Algebraic identities

- Worth recognizing single instructions with a constant operand:

```
A * 1 = A
A * 0 = 0
A / 1 = A
A * 2 = A + A
```

More delicate with floating-point

- Strength reduction:

```
A ^ 2 = A * A
```

8.2 Folding Jumps to Jumps

- A jump to an unconditional jump can copy the target address

```
        JNE lab1
   ...
lab1:   JMP lab2
```

Can be replaced by:

```
        JNE lab2
```

As a result, lab1 may become dead (unreferenced)

Jump to Return

- A jump to a return can be replaced by a return

```
JMP lab1
...
lab1: RET
```

  - Can be replaced by

```
RET
```

lab1 may become dead code

8.3 Usage of Machine idioms

- Use machine specific hardware instruction which may be less costly.

i := i + 1

ADD i, #1 ⟶ INC i