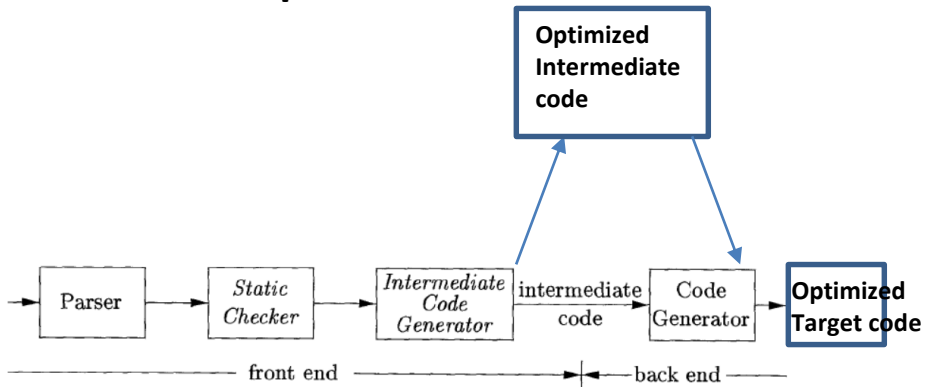


Code Optimization



Basic Blocks & Flow graphs

- Introduce a **graph representation of intermediate code** that is helpful for discussing code generation
 - Even if the graph is not constructed explicitly by a code-generation algorithm.
- Code generation **benefits from context**.
- We can do a **better job of register allocation** if we know how variables are **defined and used**.

Basic Blocks & Flow graphs

The representation is constructed as follows:

1. Partition the intermediate code into *basic blocks*, which are maximal sequences of consecutive three-address instructions with the properties that
 - (a) The flow of control can only enter the basic block through the first instruction in the block. That is, there are no jumps into the middle of the block.
 - (b) Control will leave the block without halting or branching, except possibly at the last instruction in the block.
2. The basic blocks become the nodes of a *flow graph*, whose edges indicate which blocks can follow which other blocks.

Basic Blocks

- We begin a **new basic block** with the **first instruction**
- Keep adding instructions
 - until we meet either a **jump, a conditional jump,**
 - or a **label** on the following instruction.
- In the **absence of jumps and labels**, control proceeds **sequentially** from one instruction to the next.
- **Task:** *Identify leaders*, that is, the **first instructions** in some **basic block**.

Basic Blocks - Leaders

1. The first three-address instruction in the intermediate code is a leader.
2. Any instruction that is the target of a conditional or unconditional jump is a leader.
3. Any instruction that immediately follows a conditional or unconditional jump is a leader.

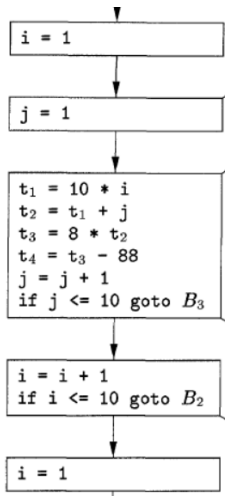
Basic Blocks

```
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
     $a[i, i] = 1.0;$ 
```

leaders are instructions
1, 2, 3, 10, 12, and 13

```
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  $j \leq 10$  goto (3)
10)  $i = i + 1$ 
11) if  $i \leq 10$  goto (2)
12)  $i = 1$ 
13)  $t5 = i - 1$ 
14)  $t6 = 88 * t5$ 
15)  $a[t6] = 1.0$ 
16)  $i = i + 1$ 
17) if  $i \leq 10$  goto (13)
```

Basic Blocks



Flow Graphs

- We represent the **flow of control** by a **flow graph**.
- The **nodes** of the flow graph are the **basic blocks**.
- There is an **edge from block B to block C** if and only if
 - it is possible for the **first instruction in block C** to immediately follow the **last instruction in block B**.

There are two ways that such an edge could be justified:

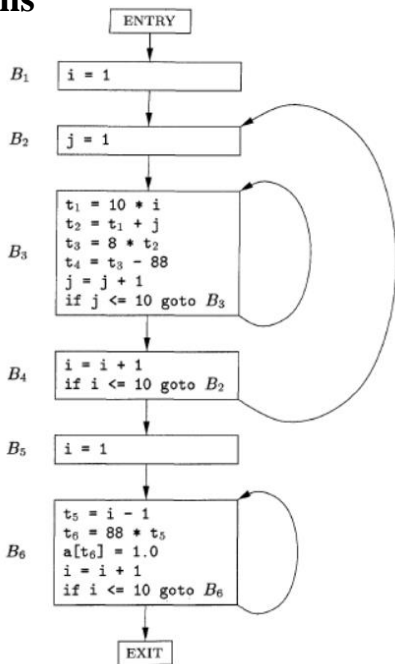
- There is a **conditional or unconditional jump** from the end of B to the beginning of C.
- **Block C immediately follows Block B** in the original order of the three-address instructions
 - B does not end in an unconditional jump
 - **Maybe due to labels**

We say that **B is a predecessor of C**, and **C is a successor of B**.

Flow Graphs

- Often we add two nodes, called the **entry and exit**,
- There is an **edge from the entry** to the **first executable node** of the flow graph,
 - that is, to the **basic block** that comes from the **first instruction** of the intermediate code.
- There is an edge **to the exit** from **any basic block** that contains an instruction that could be the **last executed instruction** of the program.

Flow Graphs



Loops

Many code transformations depend upon the identification of "loops" in a flow graph. We say that a set of nodes L in a flow graph is a loop if

1. There is a node in L called the *loop entry* with the property that no other node in L has a predecessor outside L . That is, every path from the entry of the entire flow graph to any node in L goes through the loop entry.
2. Every node in L has a nonempty path, completely within L , to the entry of L .

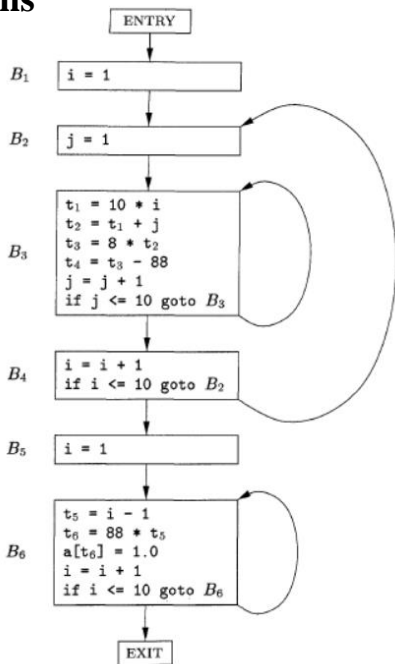
Loops

1. B_3 by itself.
2. B_6 by itself.
3. $\{B_2, B_3, B_4\}$.

The first two are single nodes with an edge to the node itself. For instance, B_3 forms a loop with B_3 as its entry. Note that the second requirement for a loop is that there be a nonempty path from B_3 to itself. Thus, a single node like B_2 , which does not have an edge $B_2 \rightarrow B_2$, is not a loop, since there is no nonempty path from B_2 to itself within $\{B_2\}$.

The third loop, $L = \{B_2, B_3, B_4\}$, has B_2 as its loop entry. Note that among these three nodes, only B_2 has a predecessor, B_1 , that is not in L . Further, each of the three nodes has a nonempty path to B_2 staying within L . For instance, B_2 has the path $B_2 \rightarrow B_3 \rightarrow B_4 \rightarrow B_2$.

Flow Graphs



Next-Use Information

- Knowing when the value of a **variable will be used next** is essential for generating good code.
 - If the value of a variable that is currently in a register **will never be referenced subsequently**, then that register can be **re-assigned to another variable**.
- Suppose **three-address statement i assigns a value to x**.
- If **statement j has x as an operand**, and control can flow from statement i to j along a path that has no intervening assignments to x, then we say **statement j uses the value of x computed at statement i**.
 - We further say that **x is live at statement i**.
- We wish to determine for **each three-address statement $x = y + z$ what the next uses of x, y, and z are**.
 - We store the information in the symbol table.
- Focus on the basic block containing this three-address statement.
- The algorithm to determine **liveness and next-use information makes a backward pass over each basic block**.

Next-Use Information

INPUT: A basic block B of three-address statements. We assume that the symbol table initially shows all nontemporary variables in B as being live on exit.

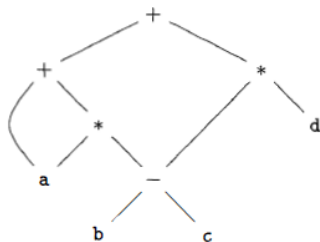
OUTPUT: At each statement $i: x = y + z$ in B , we attach to i the liveness and next-use information of x , y , and z .

METHOD: We start at the last statement in B and scan backwards to the beginning of B . At each statement $i: x = y + z$ in B , we do the following:

1. Attach to statement i the information currently found in the symbol table regarding the next use and liveness of x , y , and y .
2. In the symbol table, set x to “not live” and “no next use.”
3. In the symbol table, set y and z to “live” and the next uses of y and z to i .

DAG Representation of Expression

$a + a * (b - c) + (b - c) * d$



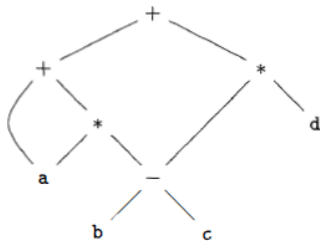
Directed Acyclic Graphs for Expressions

Like the syntax tree for an expression, a DAG has leaves corresponding to atomic operands and interior nodes corresponding to operators. The difference is that a node N in a DAG has more than one parent if N represents a common subexpression; in a syntax tree, the tree for the common subexpression

would be replicated as many times as the subexpression appears in the original expression. Thus, a DAG not only represents expressions more succinctly, it gives the compiler important clues regarding the generation of efficient code to evaluate the expressions.

DAG Representation of Expression

$$a + a * (b - c) + (b - c) * d$$

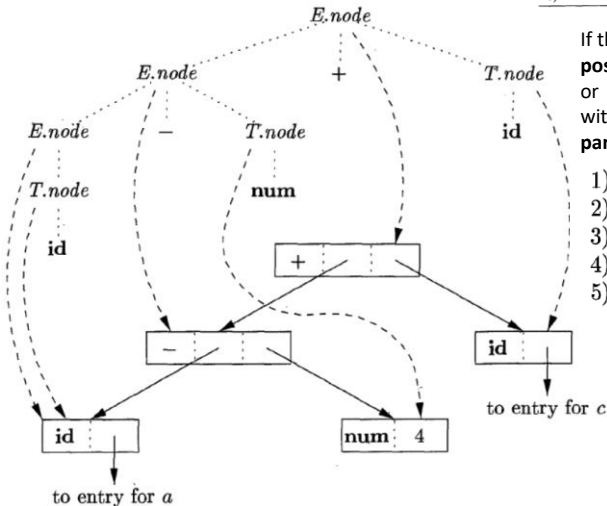


The leaf for a has two parents, because a appears twice in the expression. More interestingly, the two occurrences of the common subexpression $b-c$ are represented by one node, the node labeled $-$. That node has two parents, representing its two uses in the subexpressions $a*(b-c)$ and $(b-c)*d$. Even though b and c appear twice in the complete expression, their nodes each have one parent, since both uses are in the common subexpression $b-c$.

Application of SDD – Syntax tree construction

Syntax tree for $a - 4 + c$

PRODUCTION	SEMANTIC RULES
1) $E \rightarrow E_1 + T$	$E.node = \text{new Node}('+', E_1.node, T.node)$
2) $E \rightarrow E_1 - T$	$E.node = \text{new Node}('-', E_1.node, T.node)$
3) $E \rightarrow T$	$E.node = T.node$
4) $T \rightarrow (E)$	$T.node = E.node$
5) $T \rightarrow \text{id}$	$T.node = \text{new Leaf}(\text{id}, \text{id.entry})$
6) $T \rightarrow \text{num}$	$T.node = \text{new Leaf}(\text{num}, \text{num.val})$



If the rules are **evaluated** during a **postorder traversal** of the parse tree, or with reductions during a **bottom-up parse**, then the sequence of steps

- 1) $p_1 = \text{new Leaf}(\text{id}, \text{entry-}a)$;
- 2) $p_2 = \text{new Leaf}(\text{num}, 4)$;
- 3) $p_3 = \text{new Node}('-', p_1, p_2)$;
- 4) $p_4 = \text{new Leaf}(\text{id}, \text{entry-}c)$;
- 5) $p_5 = \text{new Node}('+', p_3, p_4)$;

It will construct a DAG if,

before creating a new node, these functions first check whether an identical node already exists. If a previously created identical node exists, the existing node is returned. For instance, before constructing a new node, $Node(op, left, right)$ we check whether there is already a node with label op , and children $left$ and $right$, in that order. If so, $Node$ returns the existing node; otherwise, it creates a new node.

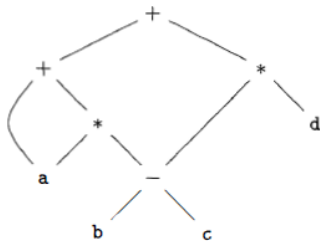
PRODUCTION	SEMANTIC RULES
1) $E \rightarrow E_1 + T$	$E.node = \mathbf{new Node}('+', E_1.node, T.node)$
2) $E \rightarrow E_1 - T$	$E.node = \mathbf{new Node}('-', E_1.node, T.node)$
3) $E \rightarrow T$	$E.node = T.node$
4) $T \rightarrow (E)$	$T.node = E.node$
5) $T \rightarrow \mathbf{id}$	$T.node = \mathbf{new Leaf}(\mathbf{id}, \mathbf{id}.entry)$
6) $T \rightarrow \mathbf{num}$	$T.node = \mathbf{new Leaf}(\mathbf{num}, \mathbf{num}.val)$

$a + a * (b - c) + (b - c) * d$

- 1) $p_1 = \mathit{Leaf}(\mathbf{id}, \mathit{entry-a})$
- 2) $p_2 = \mathit{Leaf}(\mathbf{id}, \mathit{entry-a}) = p_1$
- 3) $p_3 = \mathit{Leaf}(\mathbf{id}, \mathit{entry-b})$
- 4) $p_4 = \mathit{Leaf}(\mathbf{id}, \mathit{entry-c})$
- 5) $p_5 = \mathit{Node}('-', p_3, p_4)$
- 6) $p_6 = \mathit{Node}('*', p_1, p_5)$
- 7) $p_7 = \mathit{Node}('+', p_1, p_6)$
- 8) $p_8 = \mathit{Leaf}(\mathbf{id}, \mathit{entry-b}) = p_3$
- 9) $p_9 = \mathit{Leaf}(\mathbf{id}, \mathit{entry-c}) = p_4$
- 10) $p_{10} = \mathit{Node}('-', p_3, p_4) = p_5$
- 11) $p_{11} = \mathit{Leaf}(\mathbf{id}, \mathit{entry-d})$
- 12) $p_{12} = \mathit{Node}('*', p_5, p_{11})$
- 13) $p_{13} = \mathit{Node}('+', p_7, p_{12})$

DAG Representation of Expression

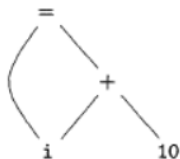
$$a + a * (b - c) + (b - c) * d$$



The leaf for a has two parents, because a appears twice in the expression. More interestingly, the two occurrences of the common subexpression $b-c$ are represented by one node, the node labeled $-$. That node has two parents, representing its two uses in the subexpressions $a*(b-c)$ and $(b-c)*d$. Even though b and c appear twice in the complete expression, their nodes each have one parent, since both uses are in the common subexpression $b-c$.

DAG Representation of Basic Blocks: Value Number Method

$$i = i + 10$$



(a) DAG

1	id			→ to entry for i
2	num	10		
3	+	1	2	
4	=	1	3	
5		...		

(b) Array.

- The **nodes** of a syntax tree or DAG are stored in an **array of records**
- In this array, we refer to nodes by giving the **integer index** of the record for that node within the array.
- **Each row** of the array represents **one record**, and therefore **one node**.
- In each record, the **first field is an operation code**, indicating the label of the node.

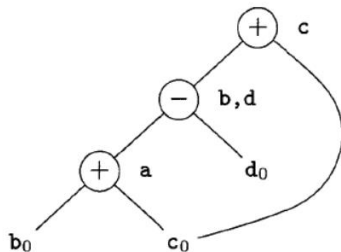
Optimization of Basic Blocks

DAG Representation of Basic Blocks

1. There is a node in the DAG for each of the initial values of the variables appearing in the basic block.
2. There is a node N associated with each statement s within the block. The children of N are those nodes corresponding to statements that are the last definitions, prior to s , of the operands used by s .
3. Node N is labeled by the operator applied at s and also attached to N is the list of variables for which it is the last definition within the block.
4. Certain nodes are designated *output nodes*. These are the nodes whose variables are *live on exit* from the block; that is, their values may be used later, in another block of the flow graph.

Finding Local Common Subexpressions

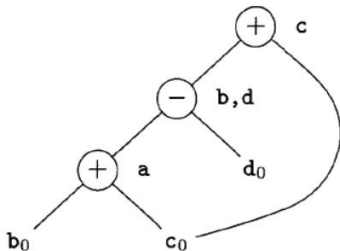
$a = b + c$
 $b = a - d$
 $c = b + c$
 $d = a - d$



Common subexpressions can be detected by noticing, as a new node M is about to be added, whether there is an existing node N with the same children, in the same order, and with the same operator.

Finding Local Common Subexpressions

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



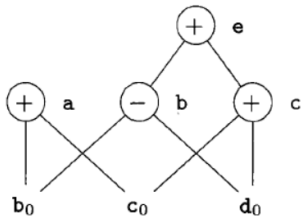
Since there are **only three nonleaf nodes** in the DAG, the basic block in can be replaced by a block with only **three statements**.

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

- If **b** is not live on exit from the block, then we do not need to compute **b** variable, and can use **d** to receive the value
- If **both b and d** are live on exit, then a fourth statement must be used to copy the value from one to the other

Dead Code Elimination

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



If **a** and **b** are **live** but **c** and **e** are **not**, we can immediately **remove the root labeled e**.

Then, the node labeled **c** **becomes a root and can be removed**.

The roots labeled **a** and **b** remain, since they each have live variables attached

The Use of Algebraic Identities

$$x + 0 = 0 + x = x$$

$$x \times 1 = 1 \times x = x$$

$$x - 0 = x$$

$$x/1 = x$$

reduction in strength,

EXPENSIVE

$$x^2$$

=

CHEAPER

$$x \times x$$

$$2 \times x$$

=

$$x + x$$

$$x/2$$

=

$$x \times 0.5$$

constant folding.

Thus the expression $2 * 3.14$ would be replaced by 6.28.

* is commutative

$$x * y = y * x.$$

The Use of Algebraic Identities

Associative laws might also be applicable to **expose common subexpressions**

$$a = b + c$$

$$t = c + d$$

$$e = t + b$$

If t is not needed outside this block, we can change this sequence to

$$a = b + c$$

$$e = a + d$$

Representation of Array References

$x = a[i]$

$a[j] = y$

$z = a[i]$

$z = x \quad ???$

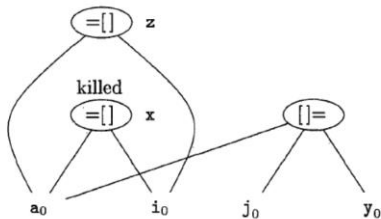
An assignment from an array, like $x = a[i]$, is represented by creating a node with operator $x = []$ and two children representing the initial value of the array, a_0 in this case, and the index i . Variable x becomes a label of this new node.

An assignment to an array, like $a[j] = y$, is represented by a new node with operator $[] =$ and three children representing a_0 , j and y . There is no variable labeling this node.

this node *kills* all currently constructed nodes whose value depends on a_0 . A node that has been killed cannot receive any more labels; that is, it cannot become a common subexpression.

Representation of Array References

```
x = a[i]
a[j] = y
z = a[i]
```



An assignment from an array, like `x = a[i]`, is represented by creating a node with operator `=[]` and two children representing the initial value of the array, `a0` in this case, and the index `i`. Variable `x` becomes a label of this new node.

An assignment to an array, like `a[j] = y`, is represented by a new node with operator `[]=` and three children representing `a0`, `j` and `y`. There is no variable labeling this node.

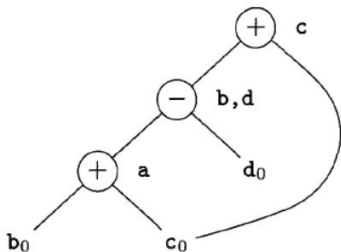
this node *kills* all currently constructed nodes whose value depends on `a0`. A node that has been killed cannot receive any more labels; that is, it cannot become a common subexpression.

Reassembling Basic Blocks From DAG 's

- After we perform whatever optimizations are possible while constructing the DAG or by manipulating the DAG once constructed,
 - we may reconstitute the three-address code for the basic block from which we built the DAG
- For each node that has one or more attached variables,
 - we construct a three-address statement that computes the value of one of those variables.
- We prefer to compute the result into a variable that is live on exit from the block.
- However, if we do not have global live-variable information to work from, we need to assume that every variable of the program (but not temporaries that are generated by the compiler to process expressions) is live on exit from the block.

Reassembling Basic Blocks From DAG 's

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



Since there are **only three nonleaf nodes** in the DAG, the basic block in can be replaced by a block with only **three statements**.

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

- If **b** is not live on exit from the block, then we do not need to compute **b** variable, and can use **d** to receive the value
- If both **b** and **d** are live on exit, then a fourth statement must be used to copy the value from one to the other

Reassembling Basic Blocks From DAG 's

If both b and d are live on exit, or if we are not sure whether or not they are live on exit, then we need to compute b as well as d . We can do so with the sequence

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

This basic block is still more efficient than the original. Although the number of instructions is the same, we have replaced a subtraction by a copy, which tends to be less expensive on most machines. Further, it may be that by doing a global analysis, we can eliminate the use of this computation of b outside the block by replacing it by uses of d . In that case, we can come back to this basic block and eliminate $b = d$ later. Intuitively, we can eliminate this copy if wherever this value of b is used, d is still holding the same value. That situation may or may not be true, depending on how the program recomputes d . \square

Peephole Optimization

- Improve the quality of the target code by applying "**optimizing transformations**" to the target program
- Peephole optimization is done by **examining** a **sliding window** of target instructions (called the peephole) and
- **Replacing instruction** sequences within the peephole by a **shorter or faster sequence**,

Eliminating Redundant Loads and Stores

```
LD a, R0  
ST R0, a
```

- Redundant loads and stores of this nature **would not be generated** by the **simple code generation** algorithm

However, a **naive code generation** algorithm would generate redundant sequences

- Note that **if the store instruction had a label**, we could not be sure that the **first instruction is always executed before the second**, so we could not remove the store instruction.

Instruction Selection – Example

example, every three-address statement of the form $x = y + z$, where x , y , and z are statically allocated, can be translated into the code sequence

```
LD  R0, y      // R0 = y      (load y into register R0)
ADD R0, R0, z   // R0 = R0 + z (add z to R0)
ST  x, R0      // x = R0      (store R0 into x)
```

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

would be translated into

```
LD  R0, b      // R0 = b
ADD R0, R0, c   // R0 = R0 + c
ST  a, R0      // a = R0
LD  R0, a      // R0 = a
ADD R0, R0, e   // R0 = R0 + e
ST  d, R0      // d = R0
```

Redundant

Eliminating Unreachable Code

An **unlabeled instruction immediately following an unconditional jump** may be removed.

One obvious peephole optimization is to eliminate jumps over jumps

```
    if debug == 1 goto L1
    goto L2
L1: print debugging information
L2:
```

After optimization

```
    if debug != 1 goto L2
    print debugging information
L2:
```

If `debug` is set to 0 at the beginning of the program, constant propagation would transform this sequence into

```
    if 0 != 1 goto L2
    print debugging information
L2:
```

Flow-of-Control Optimizations

- Simple intermediate code-generation algorithms frequently produce
 - jumps to jumps, jumps to conditional jumps, or conditional jumps to jumps.

```
    goto L1
    ...
L1: goto L2
```

by the sequence

```
    goto L2
    ...
L1: goto L2
```



Remove this, if no other jump at L1

```
    if a < b goto L1
    ...
```

```
L1: goto L2
```

can be replaced by the sequence

```
    if a < b goto L2
    ...
```

```
L1: goto L2 ←
```

```
goto L1
```

```
...
```

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

```
L3:
```

may be replaced by the sequence

```
    if a < b goto L2
```

```
    goto L3
```

```
...
```

```
L3:
```

- While the **number of instructions** in the two sequences is the **same**,
 - we **sometimes skip** the **unconditional jump** in the **second** sequence,
 - but never in the first sequence .

Optimization – Beyond basic blocks

```
void quicksort(int m, int n)
/* recursively sorts a[m] through a[n] */
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /* swap a[i], a[j] */
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

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

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



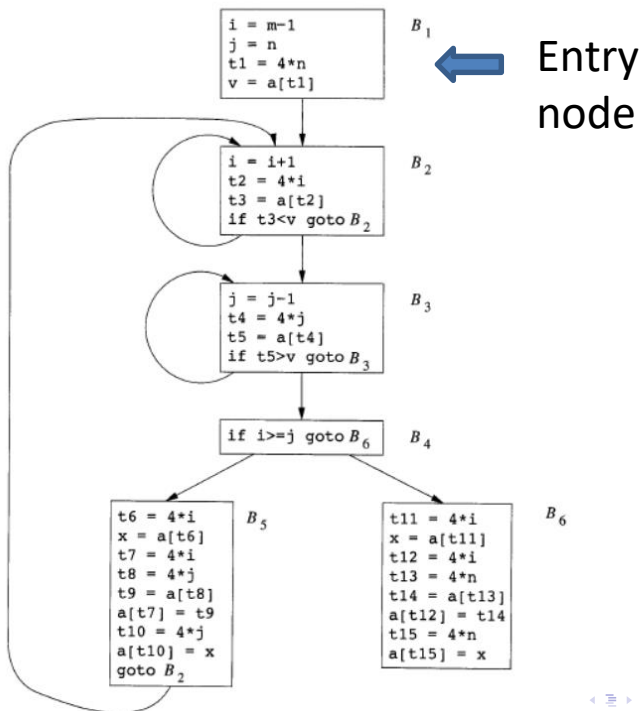
`x = a[i]` is translated as

```
t6 = 4*i
x = a[t6]
```

`a[j] = x` becomes


```
t10 = 4*j
a[t10] = x
```

- Notice that every array access in the original program translates into a pair of steps, consisting of a multiplication and an array subscription operation.
- Short program fragment translates into a rather long sequence of three-address operations.



Global Common Subexpressions

- An occurrence of an expression E is called a **common subexpression**
 - If E was previously computed and the values of the variables in E have not changed since the previous computation.
- We avoid recomputing E if we can use its previously computed value



```
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2
```

(a) Before.

B_5

```
t6 = 4*i
x = a[t6]
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2
```

B_5

(b) After.

Compute the common subexpressions $4 * i$ and $4 * j$, respectively

Global Common Subexpressions



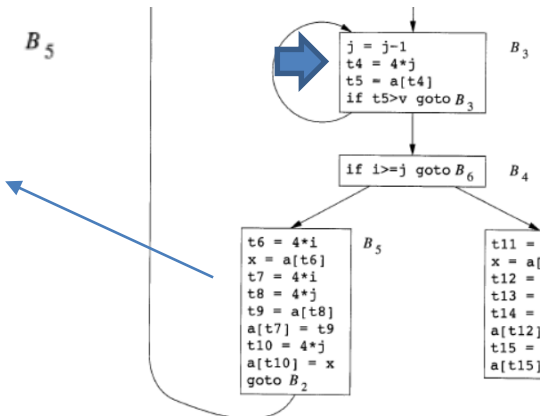
```
t6 = 4*i
x = a[t6]
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2
```

B_5

```
t8 = 4*j
t9 = a[t8]
a[t8] = x
```

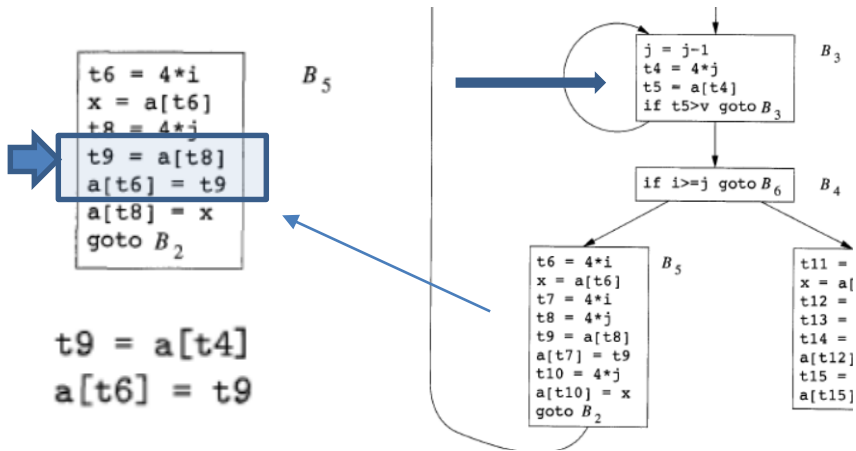
in B_5 can be replaced by

```
t9 = a[t4]
a[t4] = x
```



- Control passes from the evaluation of $4 * j$ in B_3 to B_5 ,
- No change to j and no change to t_4 , so t_4 can be used if $4 * j$ is needed.

Global Common Subexpressions

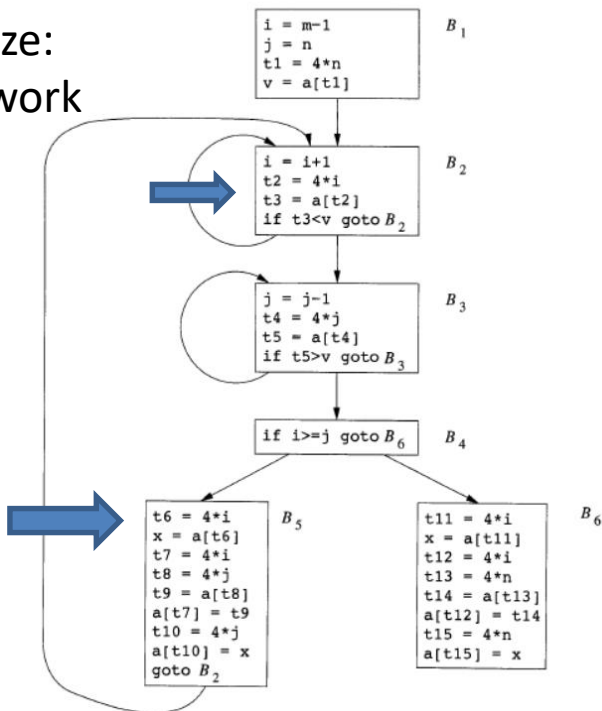


in B_5 therefore can be replaced by

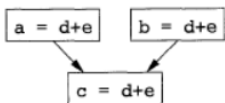
$a[t6] = t5$

$a[t4]$, a value computed into a temporary $t5$, retains its value as control leaves B_3 and then enters B_5

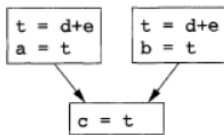
Optimize: Homework



Copy Propagation

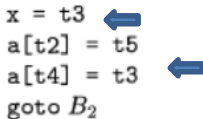
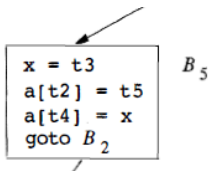


(a)




(b)

We must use a new variable t to hold the value of $d + e$
After the copy statement $u = v$, **use v for u**



Dead-Code Elimination

```
x = t3  
a[t2] = t5  
a[t4] = t3  
goto B2
```



```
a[t2] = t5  
a[t4] = t3  
goto B2
```

if (debug) print ...

debug = FALSE ← Copy propagation

- Drop this code segment
- Constant folding

Code Motion

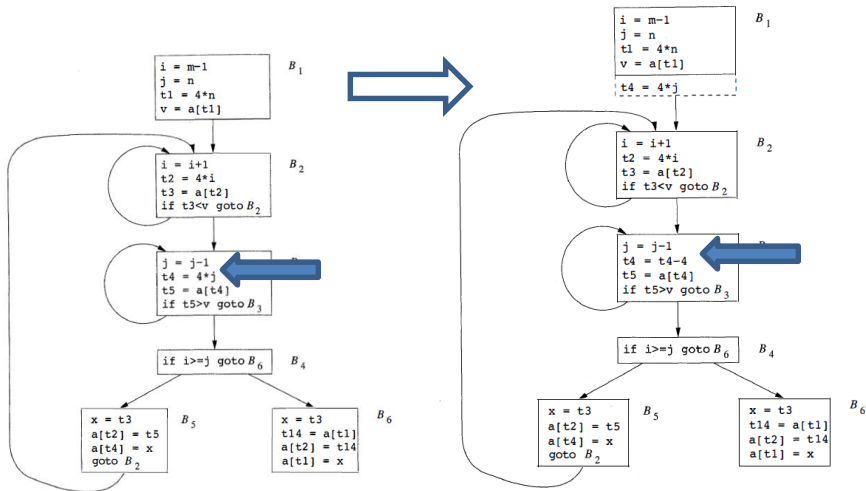
- Loops are a very important place for optimizations, especially the inner loops where programs tend to spend the bulk of their time.
- The running time of a program may be improved if we decrease the number of instructions in an inner loop,
 - even if we increase the amount of code outside that loop.
- Code Motion takes an expression
 - That yields the **same result** independent of the number of times a loop is executed (**a loop-invariant computation**) and
 - Evaluates the expression before the loop

```
while (i <= limit-2) /* statement does not change limit */
```

Code motion will result in the equivalent code

```
t = limit-2
while (i <= t) /* statement does not change limit or t */
```


Induction variable and reduction in strength



Induction variable and reduction in strength

