

Why Optimization?

- **Avoid redundancy:** something already computed need not be computed again
- **Smaller code:** less work for CPU, cache, and memory!
- **Less jumps:** jumps interfere with code pre-fetch. Can be reduced by inlining or loop unrolling.
- **Code locality:** Code and data that are accessed closely together in time should be placed close together in memory to increase spatial locality of reference.
- **Extract more information about code:** More info – better code generation

Classical Optimization

- **Analysis:** Compute information about a program
- **Semantics-preserving Transformations:** Use that information to perform program transformations
(with the goal of improving some metric, e.g. performance)

Criteria of code optimization

- Must preserve the semantic equivalence of the programs
- The algorithms should not be modified
- Transformation, on average should speed up the execution of the program
- Transformations should be simple enough to have a good effect

Redundancy elimination

- Constant Folding
- Constant/Copy propagation
- Common sub-expression elimination
- Dead-code elimination
- Code Motion
- Induction Variable and reduction in strength
- Etc.

Constant folding

- Process of simplifying constant expressions at compile time.
- Evaluate constant expressions at compile time and replace them by single value
- Goal: eliminates unnecessary operation

Constant folding: example

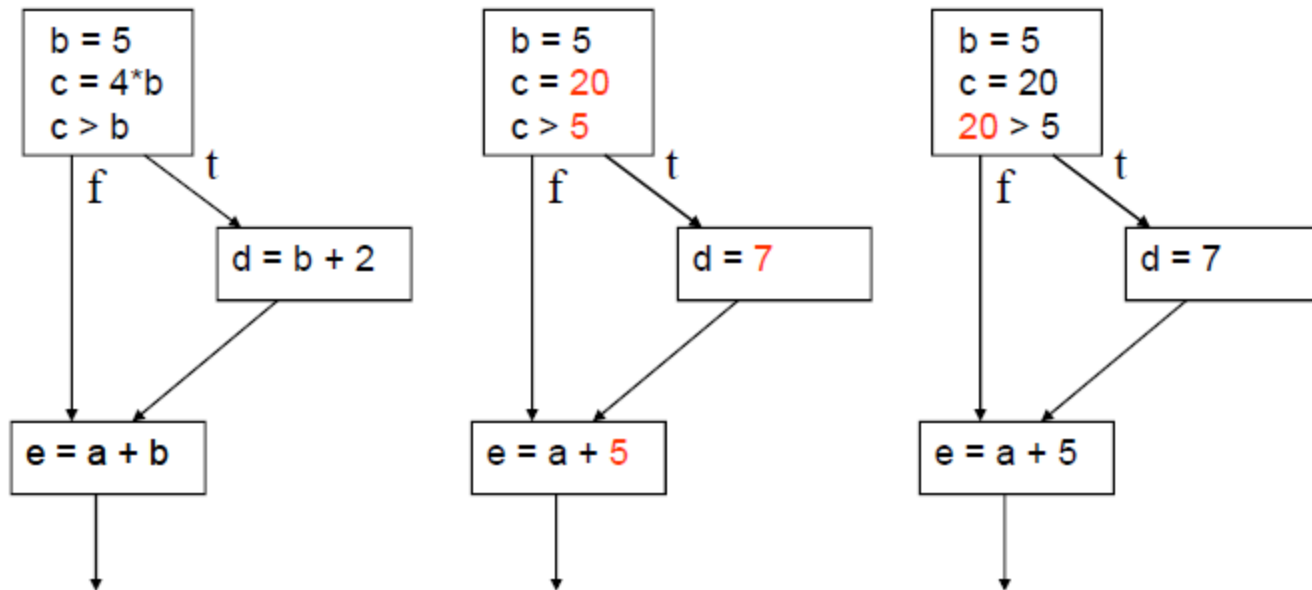
```
area := (22.0/7.0) * r ** 2
```



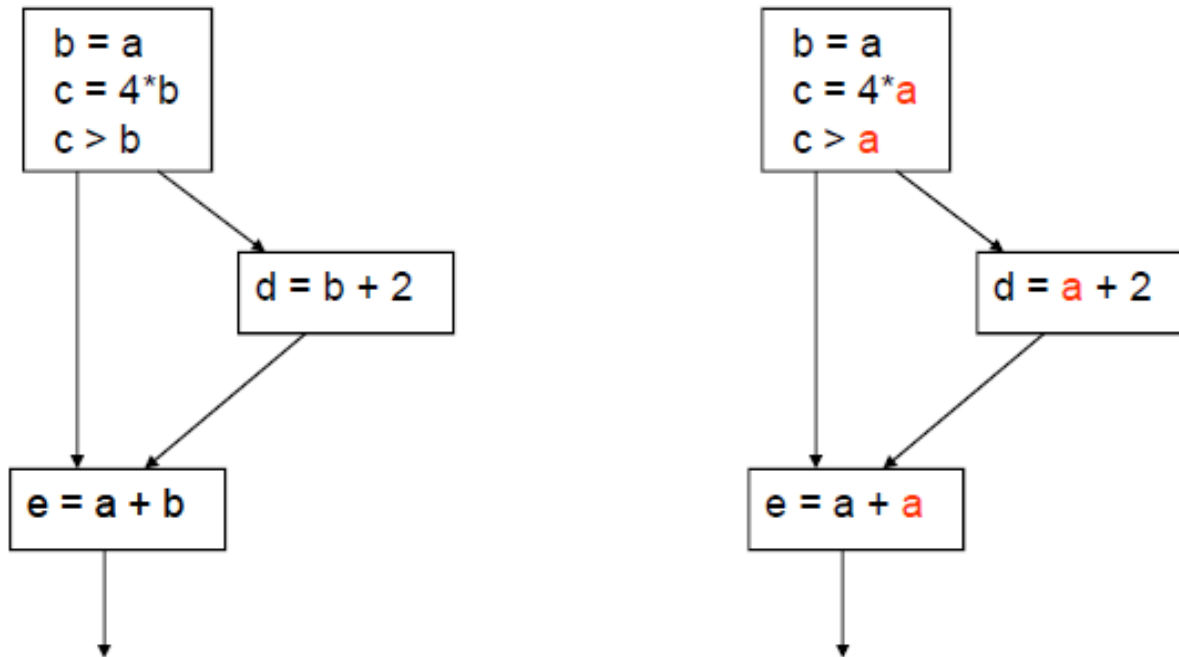
```
area := 3.14286 * r ** 2
```

Constant propagation

- 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 .



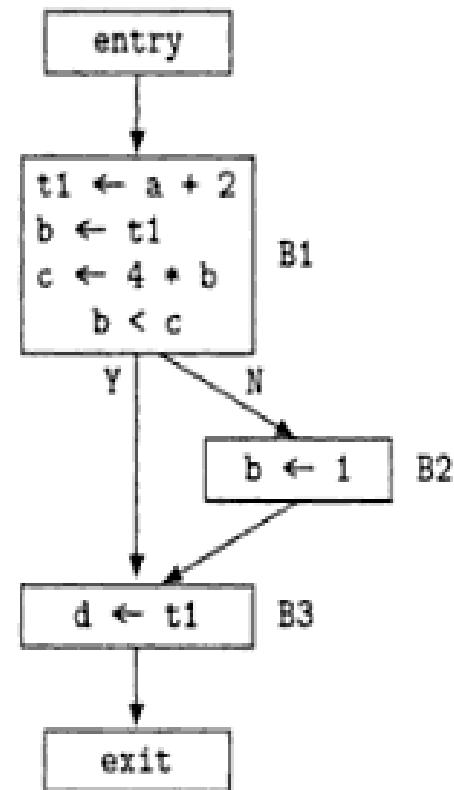
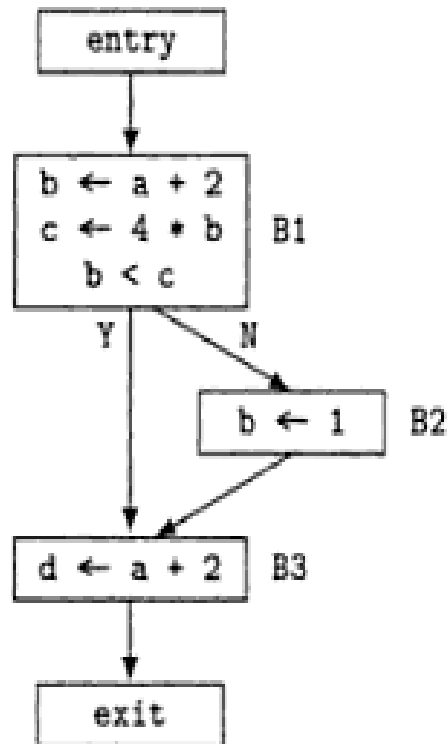
Copy Propagation



Common subexpression

- An occurrence of an expression in a program is a common subexpression if there is another occurrence of the expression whose evaluation always precedes this one in the execution order and if the operands of the expression remain unchanged between the two evaluations.

Common subexpression: example



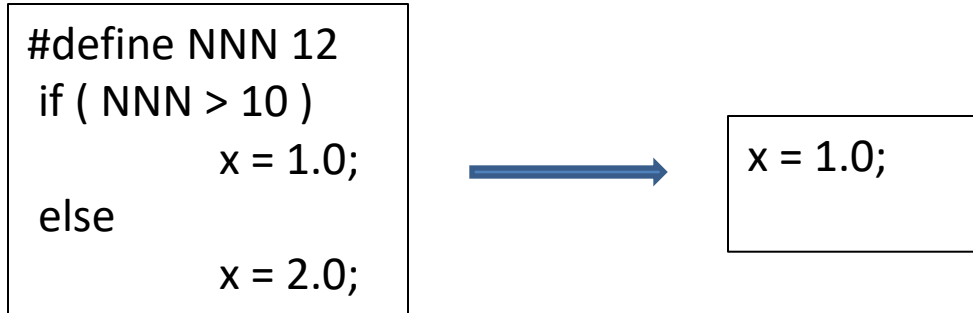
Common subexpression: example

a := b * c		temp := b * c
...		a := temp
...	→	...
x := b * c + 5		x := temp + 5

Dead-code elimination

- Dead code is code that is either never executed, or if it is executed, its result is never used by the program.

Dead code elimination: example



Code Motion

- Decrease the number of code in a loop
- Identify the loop-invariant computation

While($i \leq \text{limit}-2$)



$t = \text{limit}-2$;

While ($i \leq t$)

Induction variable and reduction in strength

- A basic induction variable is
 - a variable X whose only definitions within the loop are assignments of the form:
$$X = X + c \quad \text{or} \quad X = X - c,$$
where c is either a constant or a loop-invariant variable.
- An induction variable is
 - a basic induction variable, or
 - a variable defined once within the loop, whose value is a linear function of some induction variable at the time of the definition: $A = c_1 * B + c$

Induction variable and reduction in strength

Ex. do i = 1,99

 a(i) := 2*i-1

enddo



t := -1

do i = 1,99

 t := t+2

 a(i) := t

enddo

i is inductive

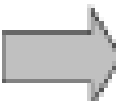
*2*i-1 is inductive*

Strength Reduction

*Replaces expensive ops (like *) with less expensive ops (like +)*

Induction variable and reduction in strength

```
while (i<10) {  
    j = 3*i+1; //<i,3,1>  
    a[j] = a[j] -2;  
    i = i+2;  
}  
  
s = 3*i+1;  
while (i<10) {  
    j = s;  
    a[j] = a[j] -2;  
    i = i+2;  
    s= s+6;  
}
```



```
foo(z) {  
    x := 3 + 6;  
    y := x - 5  
    return z * y  
}
```

```
foo(z) {  
    return z << 2;  
}
```



Constant Folding
Constant Propagation
Strenght Reduction
Dead Assignment Elimination

Basic Blocks and Flow Graph

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 and Flow Graph

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.

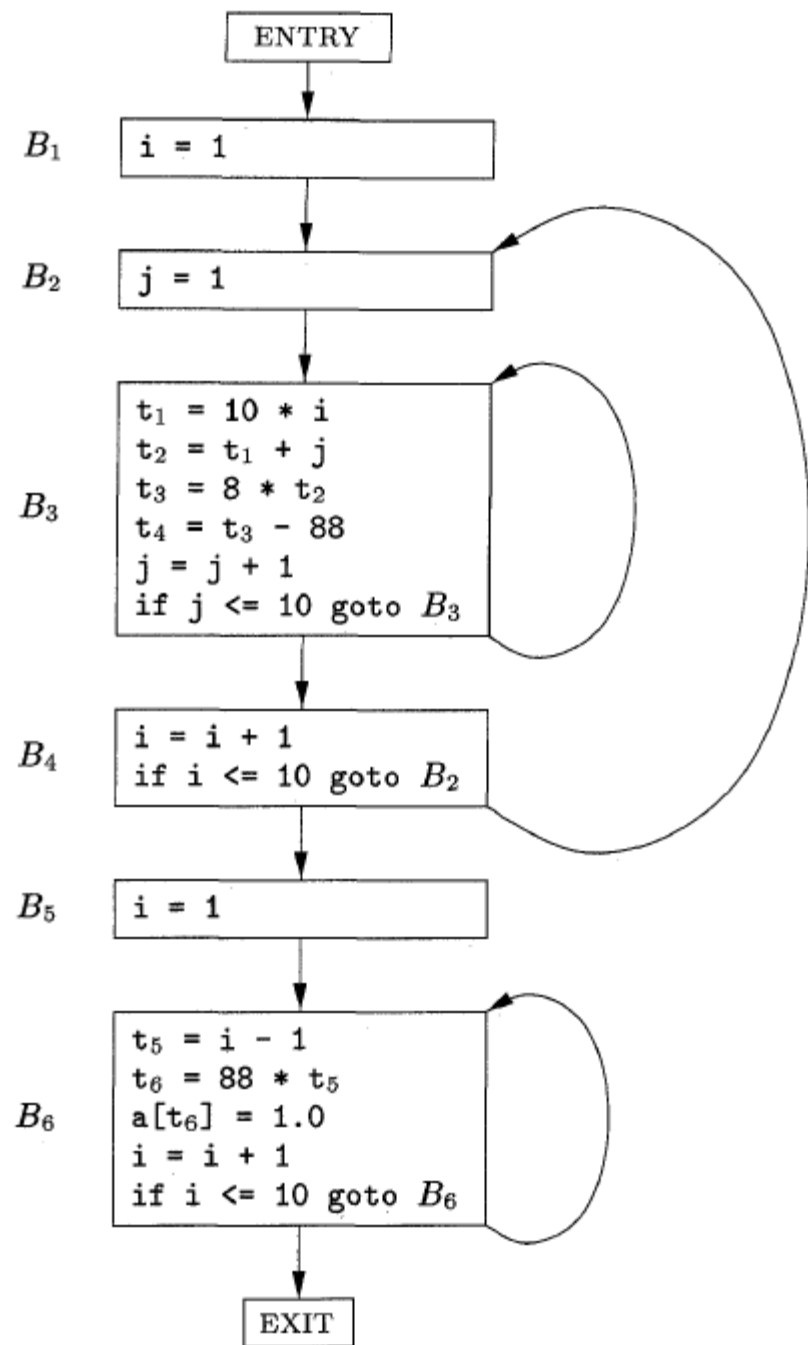
```
for i from 1 to 10 do
    for j from 1 to 10 do
         $a[i, j] = 0.0;$ 
for i from 1 to 10 do
     $a[i, i] = 1.0;$ 
```

```
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)
```

```

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)

```



Levels of Optimizations

Types of classical optimizations

- **Operation level:** *one operation in isolation*
- **Local:** *optimize pairs of operations in same basic block (with or without dataflow analysis)*
- **Global:** *optimize pairs of operations spanning multiple basic blocks and must use dataflow analysis in this case, e.g. reaching definitions.*
- **Loop:** *optimize loop body and nested loops*

Peephole 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

Peephole Optimization

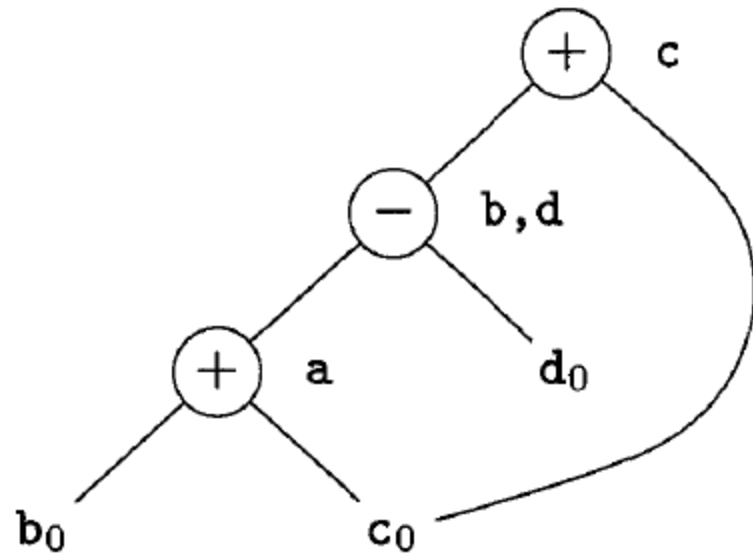
- **Peephole optimization** is a kind of optimization performed over a very small set of instructions in a segment of generated code.
- The set is called a "peephole" or a "window". It works by recognizing sets of instructions that can be replaced by shorter or faster set of instructions.

Local Optimization

- a) We can eliminate *local common subexpressions*, that is, instructions that compute a value that has already been computed.
- b) We can eliminate *dead code*, that is, instructions that compute a value that is never used.
- c) We can reorder statements that do not depend on one another; such reordering may reduce the time a temporary value needs to be preserved in a register.
- d) We can apply algebraic laws to reorder operands of three-address instructions, and sometimes thereby simplify the computation.

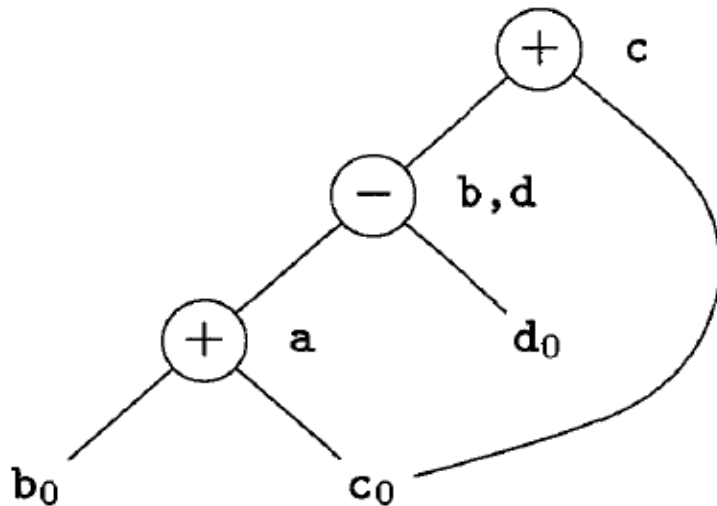
DAG Representation of Basic Block

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



Common Subexpression Elimination

- Suppose, b is not live



$$a = b + c$$

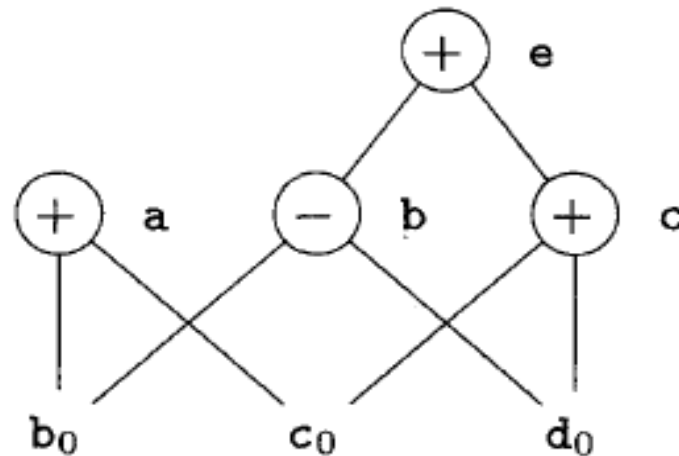
$$d = a - d$$

$$c = d + c$$

Common Subexpression Elimination

- Fails to eliminate common subexpression..
- Miss the fact that 1st and 4th computing same expression.

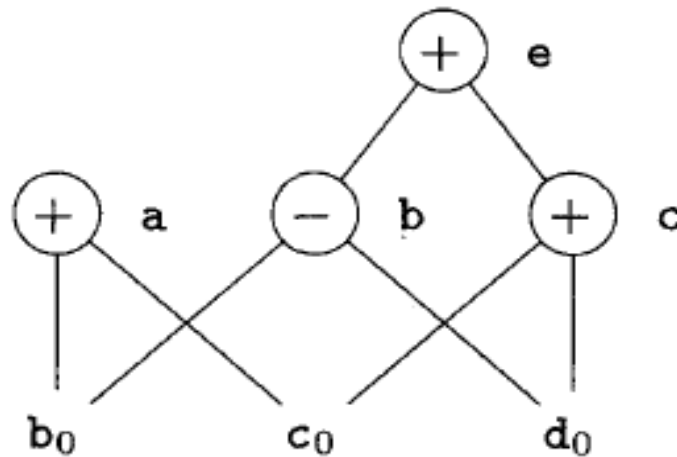
$a = b + c;$
 $b = b - d$
 $c = c + d$
 $e = b + c$



Dead Code Elimination

- If e and c are not live..

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



Algebraic Optimization

- Constant-folding
- Algebraic Identities

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

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

$$x - 0 = x$$

$$x/1 = x$$

- Strength Reduction

EXPENSIVE

CHEAPER

$$x^2$$

=

$$x \times x$$

$$2 \times x$$

=

$$x + x$$

$$x/2$$

=

$$x \times 0.5$$

Algebraic Transformation

- Following Commutativity $x * y = y * x$

Check the existence of the node in DAG during creation....

- Following Associativity and Commutativity of +

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

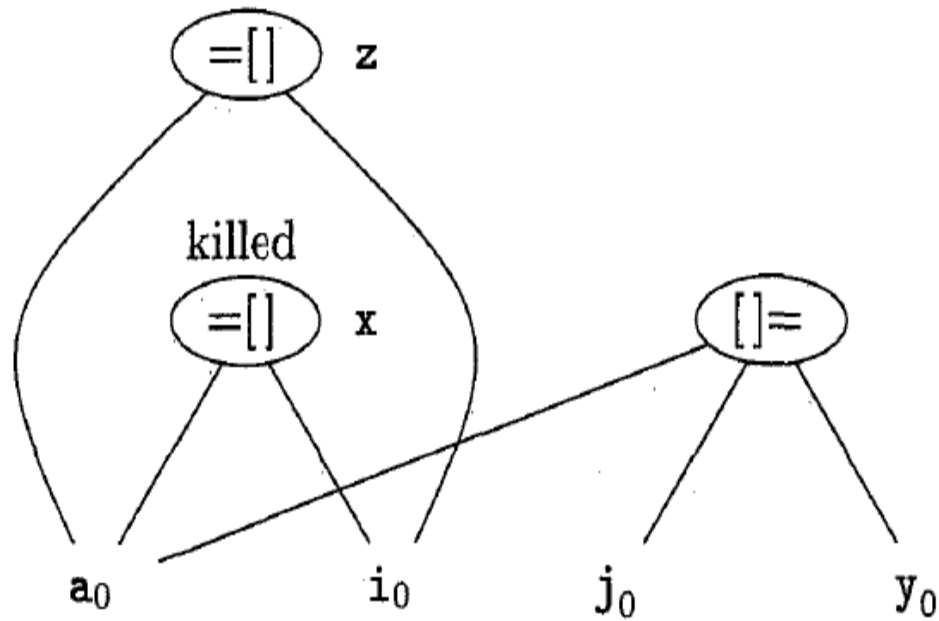
```
a = b + c  
t = c + d  
e = t + b
```



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


Optimization in Presence of Array

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



Optimization in Presence of Array

- A node can kill if it has a descendant that is an array, even though none of its children are array nodes.

```
b = 12 + a  
x = b[i]  
b[j] = y
```

