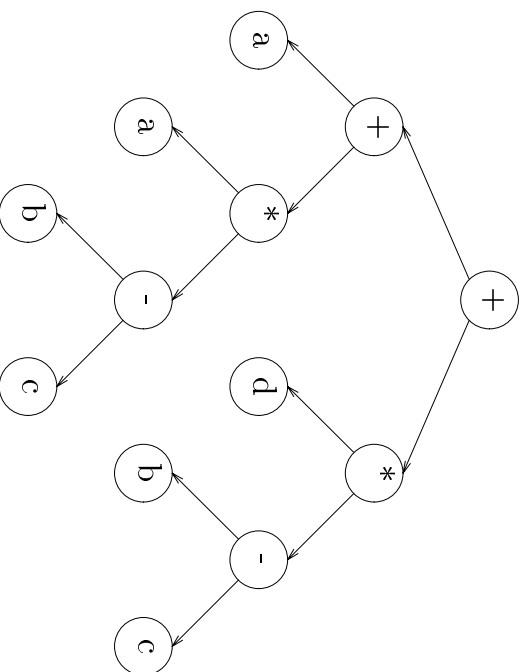


## Common subexpressions

---

Consider the tree for the expression

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



Both  $a$  and  $b-c$  are common subexpressions (*cse*)

- compute the same value
- should compute the value once

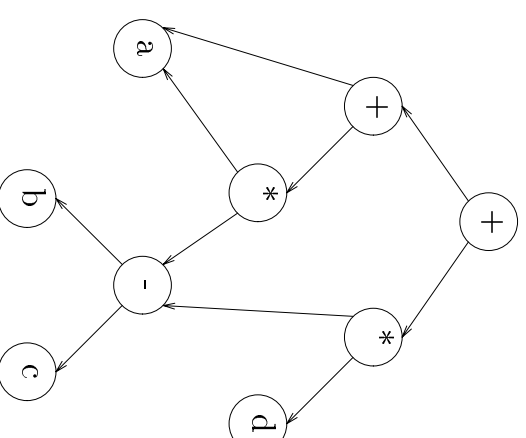
A simple and general form of code improvement

## Directed acyclic graphs

---

The *directed acyclic graph* is a useful representation for such expressions

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



The *dag* clearly exposes the *cse*s

Aho, Sethi, and Ullman, §5.2, §9.8, ...

## Directed acyclic graphs

---

A *directed acyclic graph* is a tree with sharing

- a tree is a directed acyclic graph where each node has at most one parent
- a *dag* allows multiple parents for each node
- both a tree and a *dag* have a distinguished *root*
- no cycles in the graph!

To find common subexpressions (*within a statement*)

- build the *dag*
- generate code from the *dag*

*This should lead to faster evaluation*

## Directed acyclic graphs

---

How do we build a *dag* for an expression?

- use construction primitives for building tree
- teach primitives to catch *cse*'s
  - *mkleaf* () and *mknode* ()
  - hash on  $\langle op, l, r \rangle$
- unique name for each node — its *value number*

Anywhere that we build a tree, we could build a *dag*

- initialize hash table on each expression
- catch only *cse*s within expression

## Directed acyclic graphs

---

What about *assignment* ?

- complicates *cse* detection
- each *value* has a unique node
- add subscripts to variables

While building the *dag*, an assignment

- creates new node for *lhs* — a new  $x_i$
- kills all nodes built from  $x_{i-1}$

Example

$$a_1 \leftarrow a_0 + b$$

*Can we go beyond a single statement?*

## Directed acyclic graphs

---

*Use a single dag for an entire basic block*

A *dag* for a *basic block* has labeled nodes

1. *leaves are labeled with unique identifier*
  - either variable names or constants
  - *lvalues* or *rvalues* (*obvious by context*)
  - leaves represent values on entry,  $x_0$

2. *interior nodes are labeled with operators*

3. *nodes have optional identifier labels*

- interior nodes represent computed values
- identifier label represents assignment

## Directed acyclic graphs

---

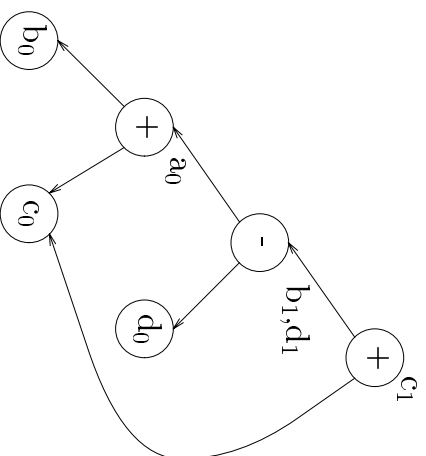
Example

Code

```
a ← b + c
b ← a - d
c ← b + c
d ← a - d
```

After Renaming

```
a0 ← b0 + c0
b1 ← a0 - d0
c1 ← b1 + c0
d1 ← a0 - d0
```



## Directed acyclic graphs

---

Building a dag

node(  $\langle id \rangle$  )  $\rightarrow$  current *dag* for  $\langle id \rangle$

1. set node( $y$ ) to undefined, for each symbol  $y$
2. for each statement  $x \leftarrow y \text{ op } z$ , repeat steps 3, 4, and 5
3. if node( $y$ ) is undefined,
  - create a leaf for  $y$
  - set node( $y$ ) to the new node
  - do the same for*  $z$
4. if  $\langle \text{op}, \text{node}(y), \text{node}(z) \rangle$  doesn't exist, create it and let  $n$  point to that node
5. delete  $x$  from the list of labels for node( $x$ )  
append  $x$  to the list of labels for  $n$   
set node( $x$ ) to  $n$

Aho, Sethi, and Ullman, Algorithm 9.2, in §9.8

## Common subexpressions

---

Going beyond basic blocks

- can no longer build DAGs
- must consider control flow

Examples

- use

$C = A+B$

$D = A+B$

- intervening kill

$C = A+B$

$A = \dots$

$D = A+B$

- possible use

$C = A+B$

if (...)

$D = A+B$

## Common subexpressions

---

More examples

- possible kill

$C = A+B$

if (...)

$A = \dots$

$D = A+B$

- possible gen

if (...)

$C = A+B$

$D = A+B$

- multiple gen

if (...)

$C = A+B$

else

$C = A+B$

$D = A+B$

We generalize these conditions as data-flow analysis