Both $a$ and $p-c$ are common subexpressions (case)

A simple and general form of code improvement:
- Should compute the value once
- Compute the same value

Consider the tree for the expression $a + a * (p - c) + (p - c) * d$.

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

Directed acyclic graphs
Directed cyclic graphs

A directed cyclic graph is a tree with sharing.

Directed acyclic graphs

To find common subexpressions (within a statement),

generate code from the dag
build the dag

generate code from the dag
build the dag

both a tree and a dag have a distinguished root

A directed acyclic graph is a tree with sharing.

This should lead to faster evaluation.

Directed acyclic graphs

catch only cases within expression
initialize hash table on each expression

anywhere that we build a tree, we could build a dag

unique name for each node — its value number

hash on do{i+},do{r,i} —

implies() and mknode()

teach primitives to catch cases

use construction primitives for building tree

How do we build a dag for an expression?

This should lead to faster evaluation.
Directed acyclic graphs

Assignments complicate case detection.

Each variable has a unique node.

What about assignment?

A dag for a basic block has labeled nodes.

Leaves are labeled with values or constants.

Variable names or constants.

I. Leaves are labeled with unique identifiers.

2. Interior nodes are labeled with operators.

Leaves represent values on entry, $x_0$.

Obvious by context.

A dag for an entire basic block.

Use a single dag for an entire basic block.

Can we go beyond a single statement?

Example

$4 \leftarrow 4a + b$

Kills all nodes built from $x_i$.

Creates new node for this $x_i$.

While building the dag, an assignment

adds substitutions to variables.
Example

Building a Def

Directed acyclic graphs

Directed a cyclic graphs

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

set node(u) to u
append x to the list of labels for u
2. delete x from the list of labels for node(x)
create a leaf for node(x)
 create it and let u point to that node
4. if op(x, op(y)) node(z) doesn't exist,
do the same for z
set node(z) to the new node
create a leaf for node(z)

3. if node(u) is undeclared,
    1. set node(u) to undeclared for each symbol <p>
   (current dag for <p>)
   2. for each statement x → y op z, repeat steps 2.

    2. for each statement x → y op z, repeat steps 2.
   1. set node(u) to undeclared for each symbol <p>
   (current dag for <p>)
   2. for each statement x → y op z, repeat steps 2.
   3. if node(u) is undeclared,
      1. set node(u) to undeclared for each symbol <p>
    (current dag for <p>)

Code

Example

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

We generalize these conditions as data-flow analysis.

\[
\begin{align*}
D &= A + B \\
C &= A + B \\
\text{else} \\
C &= A + B
\end{align*}
\]

What can we do?

\[A = \cdots (\cdots)\]

Possible use

\[D = A + B\]

Interference kill

Possible gen

\[D = A + B\]

Use

Examples

must consider control flow

can no longer build DAGs

can no longer build basic blocks