

Better code generation

Goal is to produce more efficient code for expressions

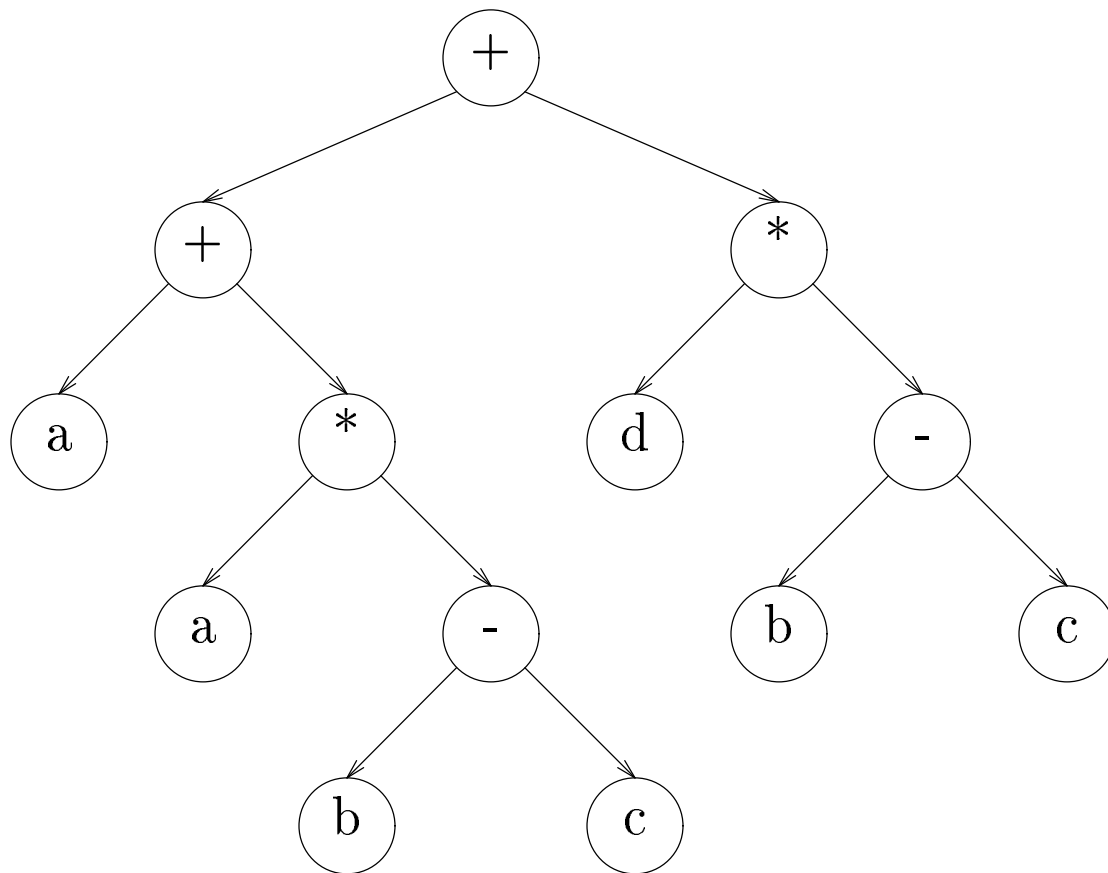
We consider

- directed acyclic graphs (DAG)
- “optimal” register allocation for trees
Sethi-Ullman
- “more optimal” register allocation for trees
Proebsting-Fischer

Common subexpressions

Consider the tree for the expression

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



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

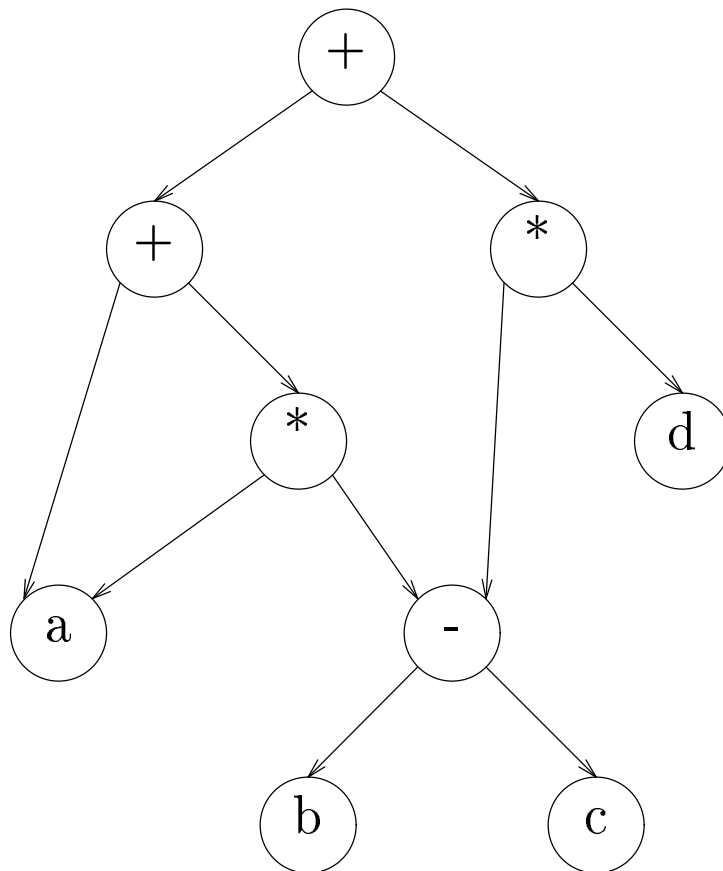
- compute the same value
- should compute the value once

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 *cses*

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 *mknnode()*
 - 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 *cses* 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

$$a \leftarrow b + c$$

$$b \leftarrow a - d$$

$$c \leftarrow b + c$$

$$d \leftarrow a - d$$

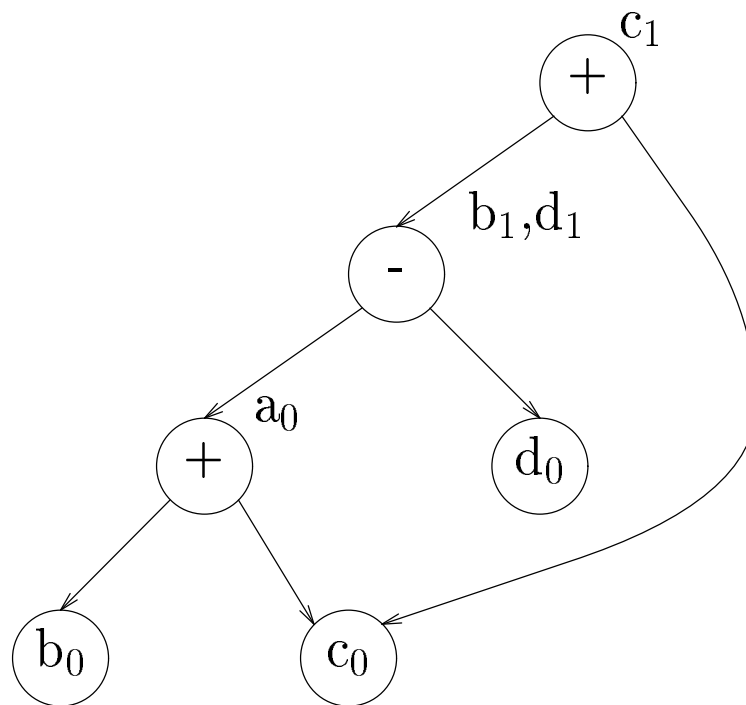
After Renaming

$$a_0 \leftarrow b_0 + c_0$$

$$b_1 \leftarrow a_0 - d_0$$

$$c_1 \leftarrow b_1 + c_0$$

$$d_1 \leftarrow a_0 - d_0$$



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

Directed acyclic graphs

Reality

Do compilers really use this stuff?

The *dag* construction algorithm is fast enough

A compilers that uses quads will (*often*)

- build a *dag* to find *cses*
- convert back to quads for later passes

Are there many *cses*? *Yes!*

- they arise in addressing
- array subscript code
- field access in records
- expressions based on loop indices
- access to parameters

Optimal code

A comment on the word “optimal”

- Aho, Sethi, and Ullman use *optimal* a lot
- particularly in regard to code generation
- look closely at the underlying assumptions
- look for simplifications, like “no sharing”

There can't be that many

- *optimal* code sequences, or
- ways of generating them

Machine model

For code generation, Aho, Sethi, and Ullman propose a simple machine model.

- byte-addressable machine with four byte words
- n general purpose registers
- two-address instructions — $op\ src, dest$

<i>Mode</i>	<i>Form</i>	<i>Address</i>	<i>Added cost</i>
absolute	M	M	1
register	R	R	0
indexed	$off(R)$	$off + c(R)$	1
ind. register	*R	$c(R)$	0
ind. indexed	* $off(R)$	$c(off + c(R))$	1

Aho, Sethi, and Ullman, §9.2

Code generation for trees

Overview of Sethi-Ullman schemes

Phase 1

- compute number of registers required to evaluate a subtree without storing values to memory
- label each interior node with that number

Phase 2

- walk the tree and generate code
- evaluation order guided by labels

Phase 1

```
if n is a leaf then
  if n is the leftmost child then
    label(n) ← 1
  else label(n) ← 0
else begin /* n is an interior node */
  let  $n_1, n_2, \dots, n_k$  be the children of n,
    ordered so that
    label( $n_1$ ) ≥ label( $n_2$ ) ≥  $\dots$  ≥ label( $n_k$ )
  label(n) ←  $\max_{1 \leq i \leq k} (\text{label}(n_i) + i - 1)$ 
```

Can compute labels in postorder

label is defined recursively as:

$$\text{label}(n) = \begin{cases} \max(l_1, l_2) & \text{if } l_1 \neq l_2 \\ l_1 + 1 & \text{if } l_1 = l_2 \end{cases}$$

Aho, Sethi, and Ullman, §9.10

Phase 2

Assumptions

- input tree is labeled by Phase 1
- `rstack` is a stack of registers
 - initialize to `r0, r1, ..., rk`
- `swap(rstack)` interchanges top two registers
 - ensures left child and parent in same register
- `tstack` is a stack of temporary locations

Phase 2

Code for phase 2

```
procedure gencode(n)
  /* case 0 — just load it */
  if n is leaf “name” and leftmost child
    gen(mov, name, top(rstack))
  else if n is interior node “op n1 n2” then
    /* case 1 — n1 in reg, n2 in RAM */
    if label(n2) = 0 then
      gencode(n1)
      gen(op, name of n2, top(rstack))
    /* case 2 — n1 needs no stores */
    /* but n2 needs more registers */
    else if 1 ≤ label(n1) ≤ label(n2)
      and label(n1) < r then
        swap(rstack)
        gencode(n2)
        R ← pop(rstack)
        gencode(n1)
        gen(op, R, top(rstack))
        push(rstack, R)
        swap(rstack)
```

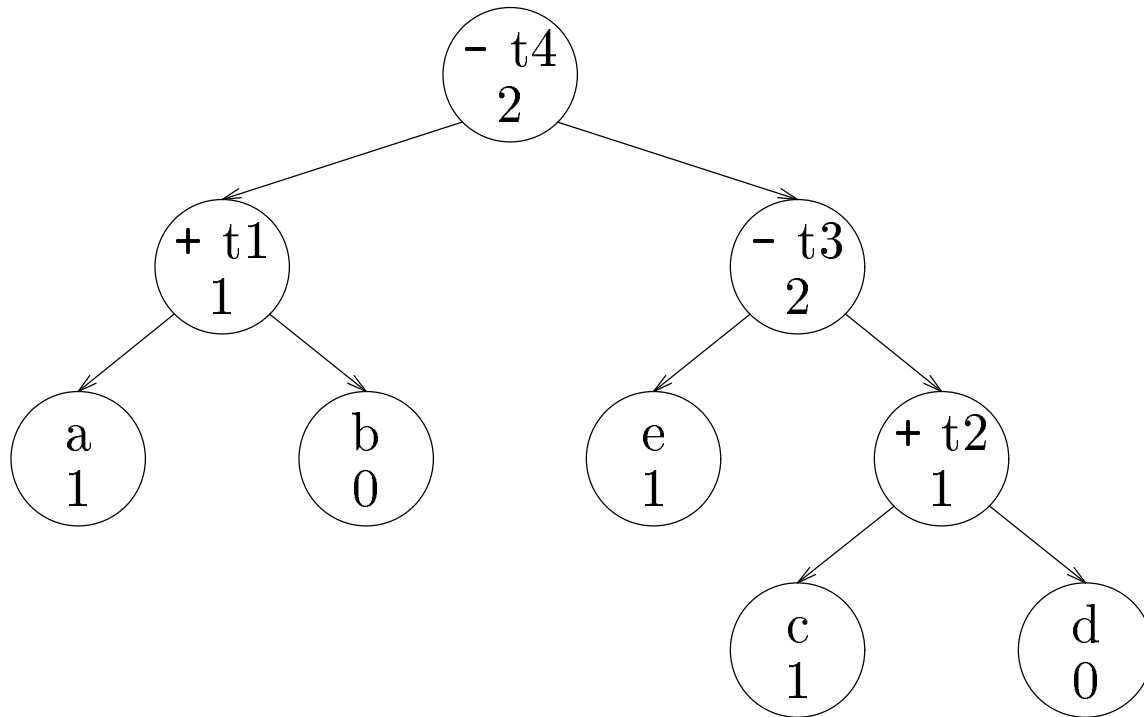
Aho, Sethi, & Ullman, §9.10

Phase 2

```
/* case 3 — symmetric to case 2 */
else if  $1 \leq \text{label}(n_2) \leq \text{label}(n_1)$  and
       $\text{label}(n_2) < r$  then
    gencode( $n_1$ )
    R = pop(rstack)
    gencode( $n_2$ )
    gen(op, top(rstack), R)
    push(rstack, R)

/* case 4 — need a temporary */
else
    gencode( $n_2$ )
    T ← pop(tstack)
    gen(MOV, top(rstack), T)
    gencode( $n_1$ )
    push(tstack, T)
    gen(op, T, top(rstack))
```

Example



```
gencode(t4)                                case 2
  gencode(t3)                                case 3
    gencode(e)                                case 0
      mov e, r1
    gencode(t2)                                case 1
      gencode(c)                                case 0
        mov c, r0
      add d, r0
    sub r0, r1
  gencode(t1)                                case 1
    gencode(a)                                case 0
      mov a, r0
    add b, r0
  sub r1, r0
```

Extensions to the labeling scheme

Multiple register operations

- increase base case to reserve registers
- paired registers may require triples

Algebraic properties

- commutativity, associativity to lower labels
- deep, narrow, left-biased trees

Common subexpressions (*dags*)

- increases complexity of code generation
(NP-Complete)
- partition into subtrees that have *cses* as roots
- order trees and apply Sethi-Ullman