Better code generation

Goal is to produce more efficient code for expression

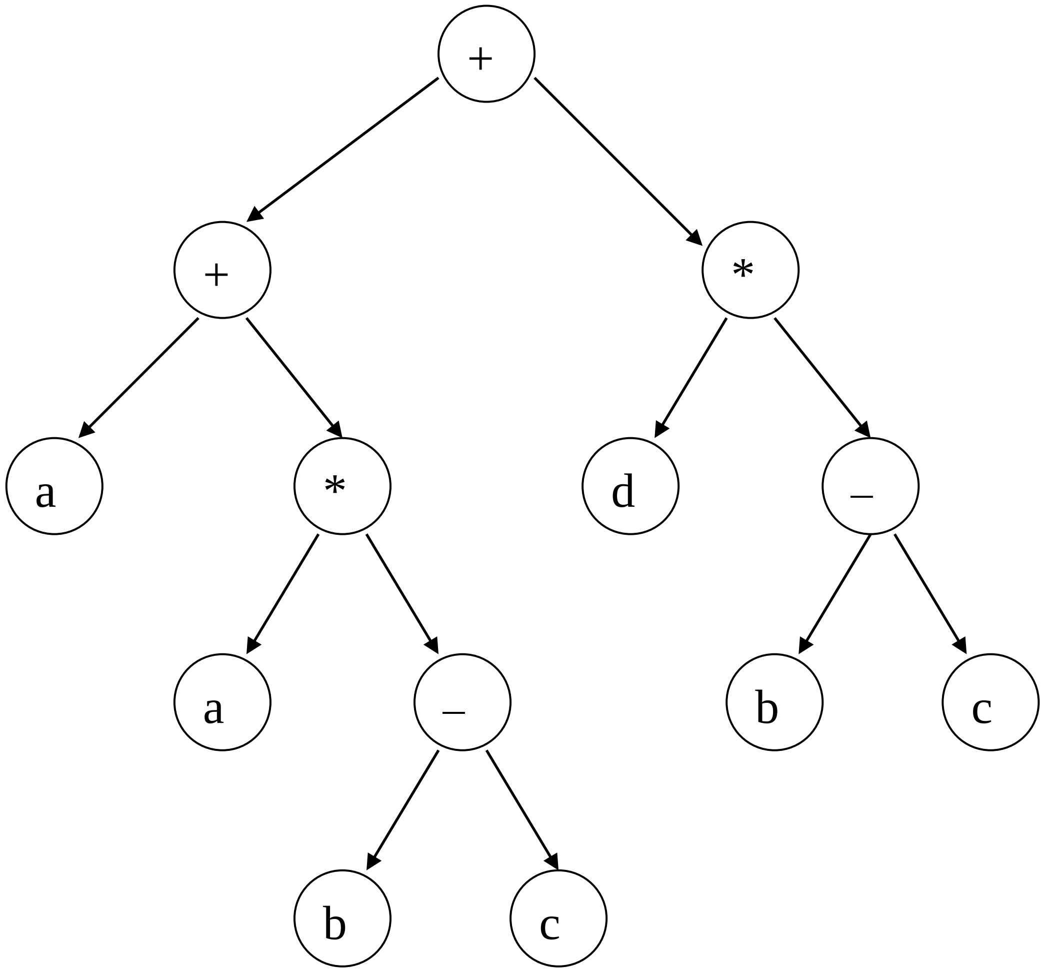
We consider

- 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)

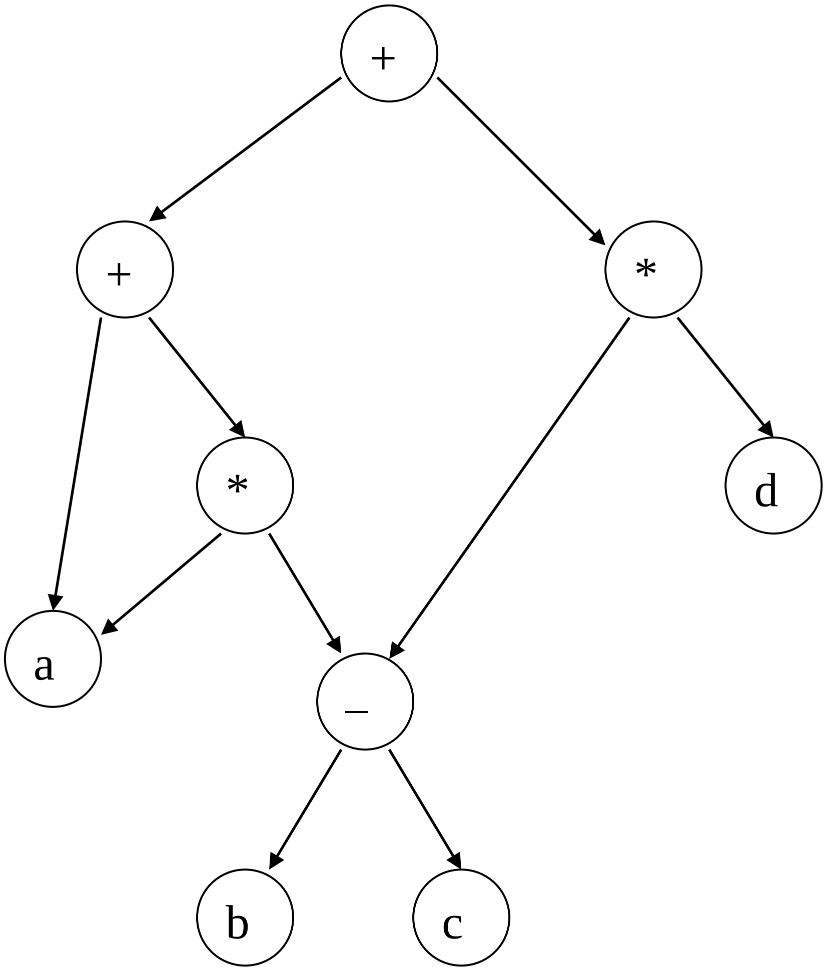
- compute the same value
- should compute the value once

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

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

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 <op, l, r>
- 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

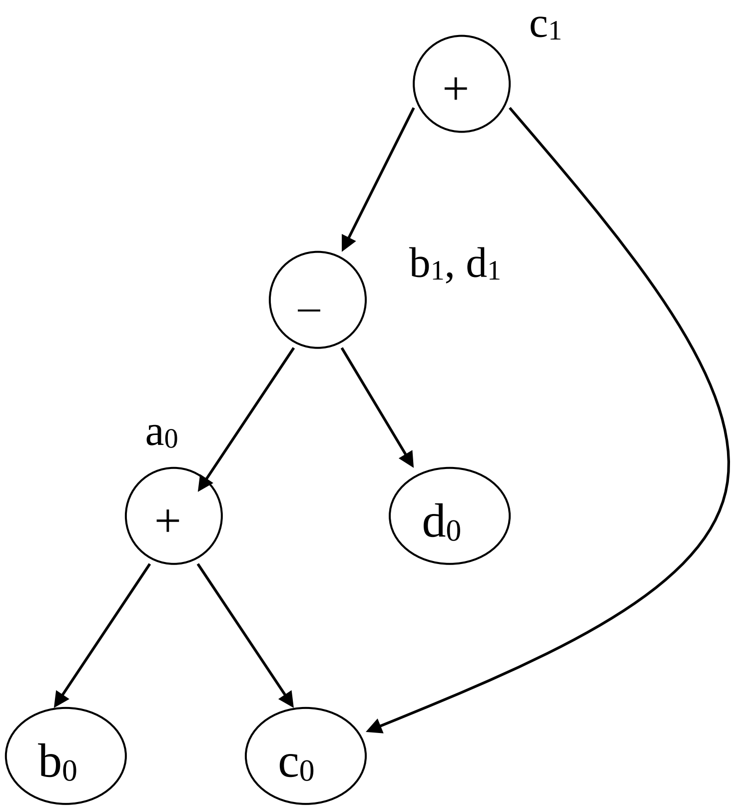
A dag for a basic block has labeled nodes

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	After renaming
$a \leftarrow b + c$	$a_0 \leftarrow b_0 + c_0$
$b \leftarrow a - d$	$b_1 \leftarrow a_0 - d_0$
$c \leftarrow b + c$	$c_1 \leftarrow b_1 + c_0$
$d \leftarrow a - d$	$d_1 \leftarrow a_0 - d_0$



Directed acyclic graphs

Building a dag

node(<id>) → current dag for <id>

: returns the most recently created node associated with id

1. set node(y) to undefined, for each symbol y
2. for each statement x ← y 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 < op, node(y), node(z) > 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* found in step 4

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

Code generator generators

Automating the process

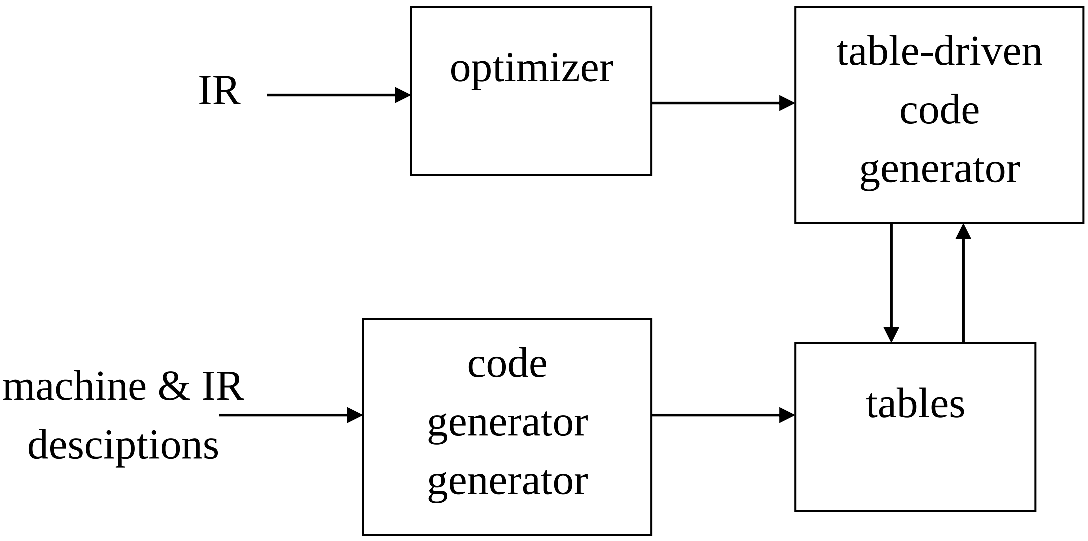
- would like a description-based tool
- machine description + IR description give code generator(cg)
- resulting cg should produce great code
- resulting cg should run quickly

Two major schools

- tree pattern matching
- instruction matching

Code generator generators

The big picture



This scheme should look familiar

Tree pattern matching

Assume that the program is represented as a set of trees.

Tree rewriting schemes (BURS)

- machine description is
  1. mapping of subtree into single node
  2. associated code(to be emitted)

- example pattern:
  - $r_i \leftarrow + \; a \; b$
  - {load r1, a; load r2, b; add r1, r2, r3}

- paradigm is
  - find a pattern to match subtree
  - replace rhs pattern with lhs node
  - emit the associated code

Tree pattern matching

Several basic techniques

- work from a simple tree walk
  depth-first traversal
  simple local choice criterion

- adopt Aho & Corasick string matching(TWIG)
  matches multiple string patterns
  translate to/from linear form

- adopt Aho & Johnson(dynamic programming)
  run rewriting and cost computation concurrently
  choose low-cost alternative at each point

- use a real tree pattern matching algorithm
  generate all subtree matches concurrently
  pick the best overall match

Tree parsing scheme

Use LR parsers

- encode pattern matching into parsing problem
  - use well understood technology
  - write grammar to describe target machine

- reductions emit code
  - attributed-style specification
  - lots of contextual knowledge available

- grammars are very ambiguous
  - reduce/reduce => pick longer reduction
  - shift/reduce => shift

- linear time scheme!