Optimization:
GCSE, GDFA, SSA, …

COMP 412

Fall 2005

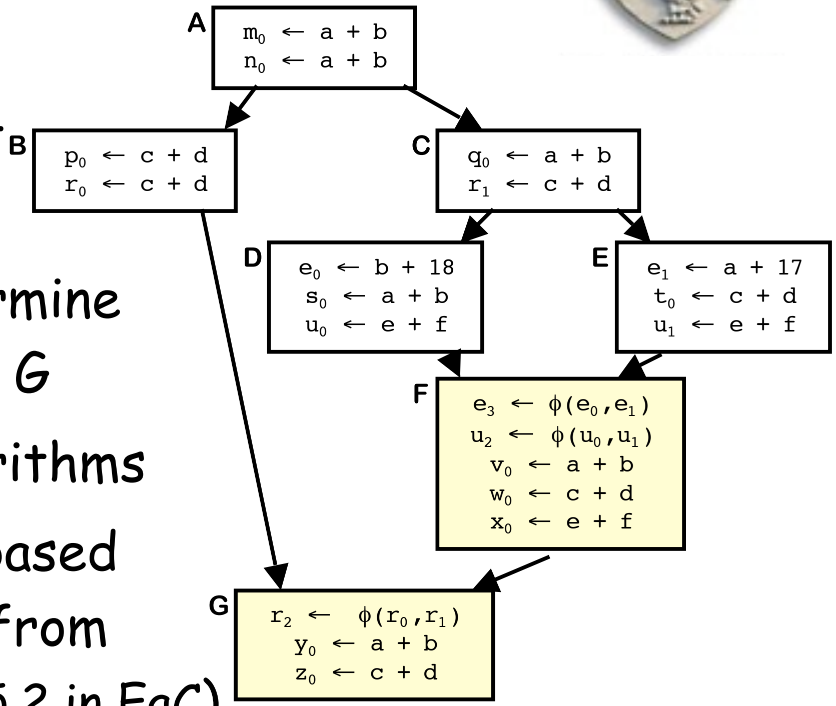
What About Larger Scopes?

Two interesting approaches

- Change IR to represent context in an explicit way (SSA form)
- Perform global analysis to determine what facts hold on entry to F & G

Approaches lead to different algorithms

- SSA form leads to fast, value-based technique using strong notions from control-flow analysis (DVNT, §8.5.2 in EaC)
- Global analysis leads to classic formulation of redunancy analysis as a problem in *global data-flow analysis*
  - Syntactic equivalence rather than value equivalence



A

$m_0 \leftarrow a + b$

$n_0 \leftarrow a + b$

B

$p_0 \leftarrow c + d$

$r_0 \leftarrow c + d$

C

$q_0 \leftarrow a + b$

$r_1 \leftarrow c + d$

D

$e_0 \leftarrow b + 18$

$s_0 \leftarrow a + b$

$u_0 \leftarrow e + f$

E

$e_1 \leftarrow a + 17$

$t_0 \leftarrow c + d$

$u_1 \leftarrow e + f$

F

$e_3 \leftarrow \phi(e_0, e_1)$

$u_2 \leftarrow \phi(u_0, u_1)$

$v_0 \leftarrow a + b$

$w_0 \leftarrow c + d$

$x_0 \leftarrow e + f$

G

$r_2 \leftarrow \phi(r_0, r_1)$

$y_0 \leftarrow a + b$

$z_0 \leftarrow c + d$

Global Common Subexpression Elimination

The goal

Find common subexpressions whose range spans basic blocks, *and* eliminate unnecessary re-evaluations

Safety

- Available expressions proves that the replacement value is current
- Transformation must ensure right name→value mapping

Profitability

- Don't add any evaluations
- Add some copy operations →

- Copies are inexpensive
- Many copies coalesce away
- Copies can shrink or stretch live ranges

Global Common Subexpression Elimination

The Big Picture

Assume, wlog, that we can annotate each block b with a set AVAIL(*b*) such that AVAIL(*b*) contains all the expressions that have been previously computed, on every path reaching b, and would produce the same result on entry to b

The Plan

Many ways to achieve this goal

1. Compute AVAIL sets
2. Assign each expression in AVAIL a unique name
3. Replace redundant uses of expressions in AVAIL
   - → x+y ∈ some AVAIL set, at each evaluation of x+y, assign the newly computed value to its unique name
   - → x+y ∈ AVAIL(*b*), and x+y is evaluated before either x or y is redefined in *b*, replace x+y with a reference to its unique name

Computing AVAIL

Initial information

- DEExpr(*b*) — expressions defined in *b* and available on exit
  - Downward Exposed Expressions
- ExprKill(*b*) — expressions that are killed in *b*
  - An expression is killed one of its inputs is assigned a value

Now,

$$AVAIL(b) = \bigcap_{p \text{ in } Pred(b)} \left(DEExpr(p) \cup \left(AVAIL(p) \cap \overline{ExprKill(i)}\right)\right)$$

- What is the starting value for AVAIL(*b*)? AVAIL(b_0)?
- How do we solve this set of simultaneous equations?

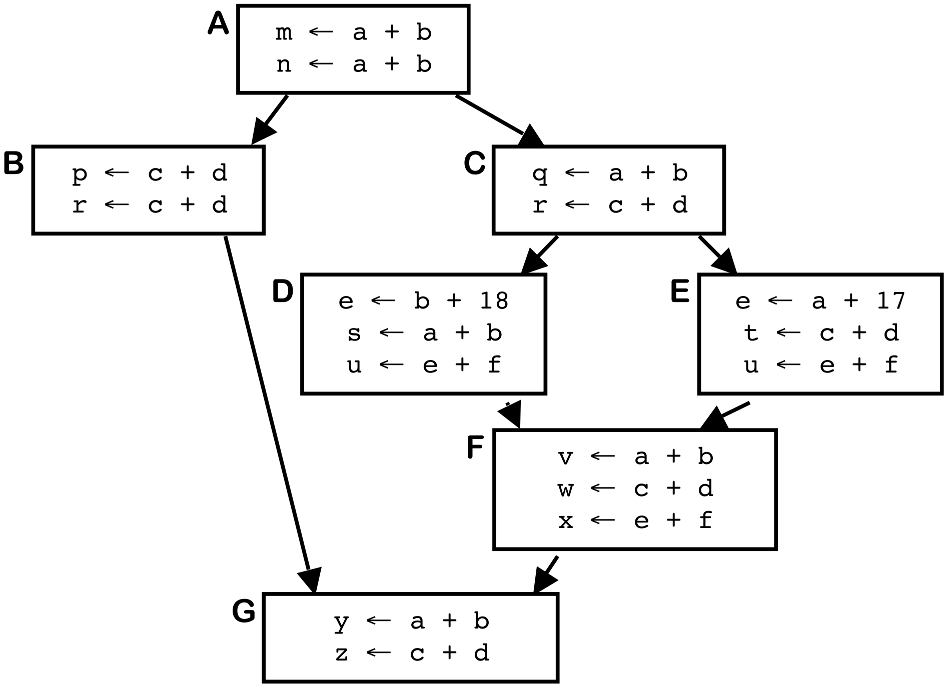
Round-robin Iterative Algorithm

```
$\textit{Avail}(b_0) \leftarrow \emptyset$
for i $\leftarrow$ 1 to N
    $\textit{Avail}(b_i) \leftarrow \{\textit{ all expressions }\}$
change $\leftarrow$ true
while (change)
    change $\leftarrow$ false
    for i $\leftarrow$ 0 to N
        $\text{Temp} \leftarrow \bigcap_{x \in pred(b_i)} (\textit{DEExpr}(x) \cup (\textit{Avail}(x) \cap \overline{\textit{ExprKill}(x)}))$
    if $\textit{Avail}(b_i) \neq \text{Temp}$ then
        change $\leftarrow$ true
         $\text{Avail}(b_i) \leftarrow \text{Temp}$
```

The round-robin solver is easier to analyze than the more efficient worklist solver.

- Termination: does it halt?
- Correctness: what answer does it produce?
- Speed: how quickly does it find that answer?

Concrete Example: Available Expressions



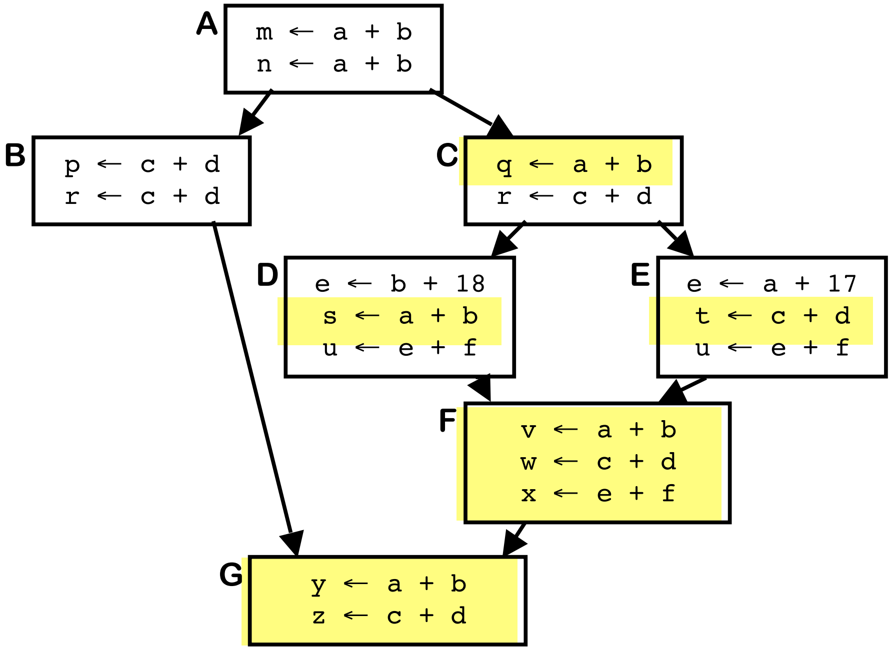
***E* = {**`a+b,c+d,e+f,a+17,b+18`**}**

2^E is the set of all subsets of *E*

2^E = [{a+b,c+d,e+f,a+17,b+18},
{a+b,c+d,e+f,a+17},
{a+b,c+d,e+f,b+18},
{a+b,c+d,a+17,b+18},
{a+b,e+f,a+17,b+18},
{c+d,e+f,a+17,b+18}, {a+b,c+d,e+f},
{a+b,c+d,b+18}, {a+b,c+d,a+17},
{a+b,e+f,a+17},
{a+b,e+f,b+18},{a+b,a+17,b+18},
{c+d,e+f,a+17}, {c+d,e+f,b+18},
{c+d,a+17,b+18},{e+f,a+17,b+18},
{a+b,c+d},{a+b,e+f},{a+b,a+17},
{a+b,b+18},{c+d,e+f},{c+d,a+17},
{c+d,b+18},{e+f,a+17},{e+f,b+18},
{a+17,b+18},{a+b}, {c+d}, {e+f}, {a+17},
{b+18}, {}]

Making Theory Concrete

Computing AVAIL for the example



	A	B	C	D	E	F	G
DEExpr	a+b	c+d	a+b,c+d	b+18,a+b,e+f	a+17,c+d,e+f	a+b,c+d,e+f	a+b,c+d
ExprKill	{ }	{ }	{ }	e+f	e+f	{ }	{ }

Avail(A) = Ø

Avail(B) = {a+b} ∪ (ø ∩ all)

= {a+b}

Avail(C) = {a+b}

Avail(D) = {a+b,c+d} ∪ ({a+b} ∩ all)

= {a+b,c+d}

Avail(E) = {a+b,c+d}

Avail(F) = [{b+18,a+b,e+f} ∪

({a+b,c+d} ∩ {all - e+f})]

∩ [{a+17,c+d,e+f} ∪

({a+b,c+d} ∩ {all - e+f})]

= {a+b,c+d,e+f}

Avail(G) = [{c+d} ∪ ({a+b} ∩ all)]

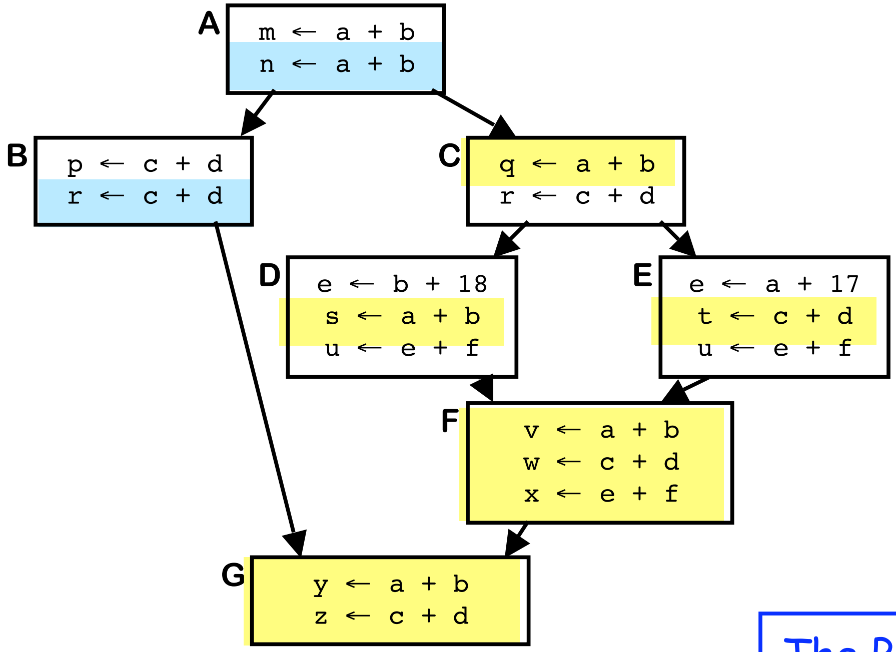
∩ [{a+b,c+d,e+f} ∪

({a+b,c+d,e+f} ∩ all)]

= {a+b,c+d}

Making Theory Concrete

Computing AVAIL for the example



Using AVAIL information in conjunction with local value numbering (LVN) can find all of the redundancy in this example.

In fact, if we initialize the hash table with the AVAIL set for the block, we can use LVN to perform all of our replacements.

recall

The Plan

1. Compute AVAIL Sets
2. Assign a unique name to each expr. that appears in an AVAIL set
3. Replace evaluations with references, as legal

SSA Name Space (in general)

Two principles

- Each name is defined by exactly one operation
- Each operand refers to exactly one definition

To reconcile these principles with real code

- Add subscripts to variable names for uniqueness
- Insert ϕ-functions at merge points to reconcile name space

$x \leftarrow \ldots$

$x \leftarrow \ldots$

$\ldots \leftarrow x + \ldots$

becomes

$x_0 \leftarrow \ldots$

$x_1 \leftarrow \ldots$

$x_2 \leftarrow \phi(x_0, x_1)$

$\leftarrow x_2 + \ldots$

SSA Name Space

About these ϕ-functions …

- A ϕ-function occurs at the start of a block
- A ϕ-function has one argument for each CFG edge entering the block
- A ϕ-function returns the argument that corresponds to the edge along which control flow entered the block
  - All ϕ-functions in the block execute concurrently
  - Since machines do not support ϕ-functions, must translate back out of SSA form before we produce executable code
- Using SSA form leads to simpler or better formulations of many optimizations (*alternative to global data-flow analysis*)

Building SSA

SSA Form

- Each name is defined exactly once
- Each use refers to exactly one name

What's Hard?

- Straight-line code is easy
- Split points are easy
- Merge points are hard

(Sloppy) Construction Algorithm

- Insert a ϕ-function for each variable at each merge point
- Rename all values for uniqueness (*using subscripts*)

This approach

- Inserts too many ϕ-functions
- Inserts ϕ-functions in too many places

The rest, however, is optimization & beyond the scope of today's lecture. (See §9 in EaC)