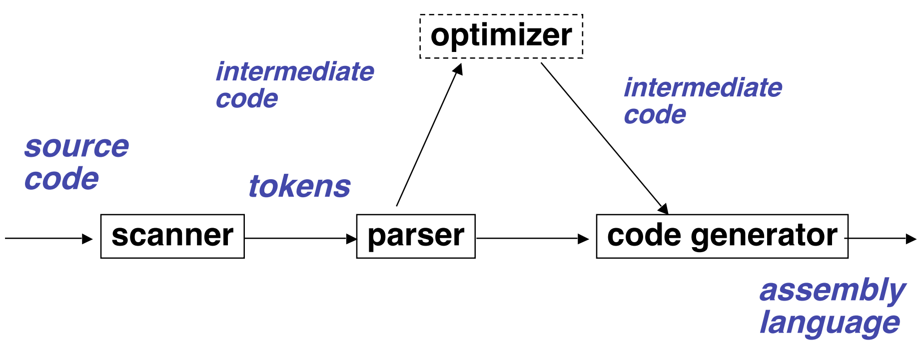
Machine Independent Compiler Optimization

- What is classical machine independent optimization?
- Control flow graph, basic blocks, local opts
- Control flow abstractions: loops, dominators
- Four classical dataflow problems
  - Reaching definitions
  - Live variables
  - Available expressions
  - Very busy expressions

Phases of Compilation



Optimization is a semantics-preserving transformation

Example

- To define classical optimizations using an example loop from Fortran scientific code
- Opportunities for these optimizations result from table-driven code generation

```
   …
   sum = 0
   do 10 i = 1, n
10  sum = sum + a(i) * a(i)
   …
```


Three Address Code

```
1.  sum = 0
2.  i = 1
3.  if i > n goto 15
4.  t1 = addr(a) - 4
5.  t2 = i * 4
6.  t3 = t1[t2]
7.  t4 = addr(a) - 4
8.  t5 = i * 4
9.  t6 = t4[t5]
10. t7 = t3 * t6
11. t8 = sum + t7
12. sum = t8
13. i = i + 1
14. goto 3
15.
```

sum = 0; initialize loop counter (lines 1–2)

loop test, check for limit (line 3)

a[i] (lines 4–6)

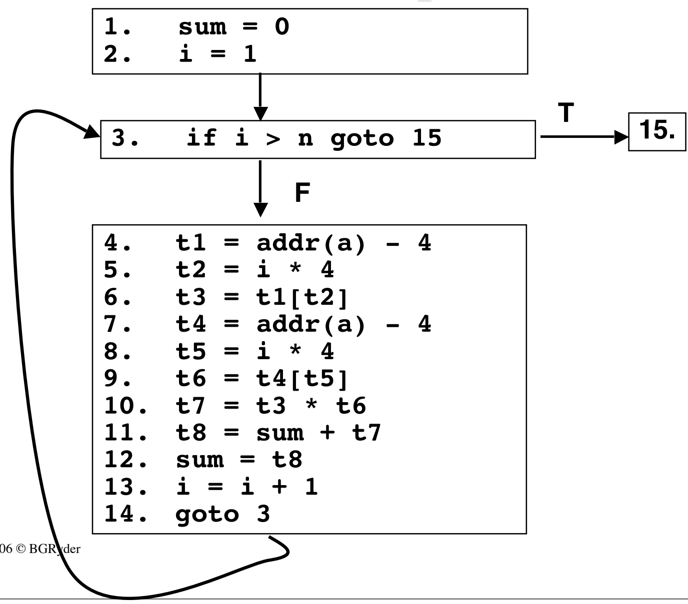
a[i] (lines 7–9)

a[i] * a[i] (lines 10–11)

increment sum (line 12)

increment loop counter (line 13)

Control Flow Graph (CFG)



Local Common Subexpression Elimination (CSE)

```
1.  sum = 0
2.  i = 1
3.  if i > n goto 15
4.  t1 = addr(a) - 4
5.  t2 = i * 4
6.  t3 = t1[t2]
7.  t4 = addr(a) - 4
8.  t5 = i * 4
9.  t6 = t4[t5]
10. t7 = t3 * t6
11. t8 = sum + t7
12. sum = t8
13. i = i + 1
14. goto 3
15.
```

→

```
1.  sum = 0
2.  i = 1
3.  if i > n goto 15
4.  t1 = addr(a) - 4
5.  t2 = i * 4
6.  t3 = t1[t2]
7.  t4 = addr(a) - 4
8.  t5 = i * 4
9.  t6 = t4[t5]
10. t7 = t3 * t6
10a t7 = t3 * t3
11. t8 = sum + t7
11a sum = sum + t7
12. sum = t8
13. i = i + 1
14. goto 3
15.
```

Blue code eliminated; Red code added

Invariant Code Motion

```
1.   sum = 0
2.   i = 1
3.   if i > n goto 15
4.   t1 = addr(a) - 4
5.   t2 = i * 4
6.   t3 = t1[t2]
10a t7 = t3 * t3
11a sum = sum + t7
13. i = i + 1
14. goto 3
15.
```

→

```
1.   sum = 0
2.   i = 1
2a  t1 = addr(a) - 4
3.   if i > n goto 15
4.   t1 = addr(a) - 4
5.   t2 = i * 4
6.   t3 = t1[t2]
10a t7 = t3 * t3
11a sum = sum + t7
13. i = i + 1
14. goto 3
15.
```

MachineIndepOpt-1, Sp06 © BGRyder 7

Reduction in Strength

```
1.   sum = 0
2.   i = 1
2a  t1 = addr(a) - 4
3.   if i > n goto 15
5.   t2 = i * 4
6.   t3 = t1[t2]
10a t7 = t3 * t3
11a sum = sum + t7
13. i = i + 1
14. goto 3
15.
```

→

```
1.   sum = 0
2.   i = 1
2a  t1 = addr(a) - 4
2b  t2 = i * 4
3.   if i > n goto 15
5.   t2 = i * 4
6.   t3 = t1[t2]
10a t7 = t3 * t3
11a sum = sum + t7
11b t2 = t2 + 4
13. i = i + 1
14. goto 3
15.
```

MachineIndepOpt-1, Sp06 © BGRyder 8

Test Elision and Induction Variable Elimination

```
1.   sum = 0
2.   i = 1
2a   t1 = addr(a) - 4
2b   t2 = i * 4
3.   if i > n goto 15
6.   t3 = t1[t2]
10a t7 = t3 * t3
11a sum = sum + t7
11b t2 = t2 + 4
13. i = i + 1
14. goto 3
15
```

→

```
1.   sum = 0
2.   i = 1
2a   t1 = addr(a) - 4
2b   t2 = i * 4
2c   t9 = 4 * n
3.   if i > n goto 15
3a   if t2 > t9 goto 15
6.   t3 = t1[t2]
10a t7 = t3 * t3
11a sum = sum + t7
11b t2 = t2 + 4
13. i = i + 1
14. goto 3a
15
```

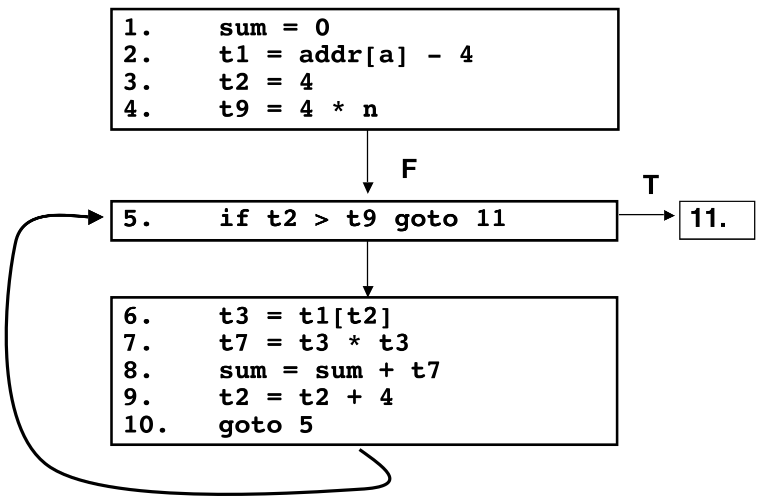

Constant Propagation and Dead Code Elimination

```
1.   sum = 0
2.   i = 1
2a   t1 = addr(a) - 4
2b   t2 = i * 4
2c   t9 = 4 * n
3a   if t2 > t9 goto 15
6.   t3 = t1[t2]
10a t7 = t3 * t3
11a sum = sum + t7
11b t2 = t2 + 4
14. goto 3a
15
```

→

```
1.   sum = 0
2.   i = 1
2a   t1 = addr(a) - 4
2b   t2 = i * 4
2d   t2 = 4
2c   t9 = 4 * n
3a   if t2 > t9 goto 15
6.   t3 = t1[t2]
10a t7 = t3 * t3
11a sum = sum + t7
11b t2 = t2 + 4
14. goto 3a
15
```

New Control Flow Graph



How to build CFG?

- **Need to find basic blocks and possible branches between them**
- **Basic block *leader* statements**
  - **First program statement**
  - **Targets of conditional or unconditional goto's**
  - **Any statement following a conditional goto**
- **For each *leader s*, construct basic block B_s as all statements *t* reachable from *s* through straight-line code**
- **Eventually, any statements not included in some basic block are unreachable from program entry *dead code***

Leader Statements

```
1.  sum = 0
2.  i = 1
3.  if i > n goto 15
4.  t1 = addr(a) - 4
5.  t2 = i * 4
6.  t3 = t1[t2]
7.  t4 = addr(a) - 4
8.  t5 = i * 4
9.  t6 = t4[t5]
10. t7 = t3 * t6
11. t8 = sum + t7
12. sum = t8
13. i = i + 1
14. goto 3
15.
```

first program statement

conditional goto statement

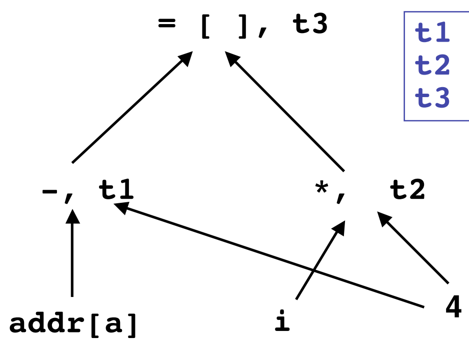
statement following conditional goto

branch target

Local CSE

- Accomplished while translating into three address code
- For each statement, form expression DAGs (for operand sharing)
  - Operands are children of operator nodes
  - Operand nodes can be used by more than one operator node
  - Intermediate results that must be stored cause creation of compiler temporaries
  - Multiple labels on same node mean CSE

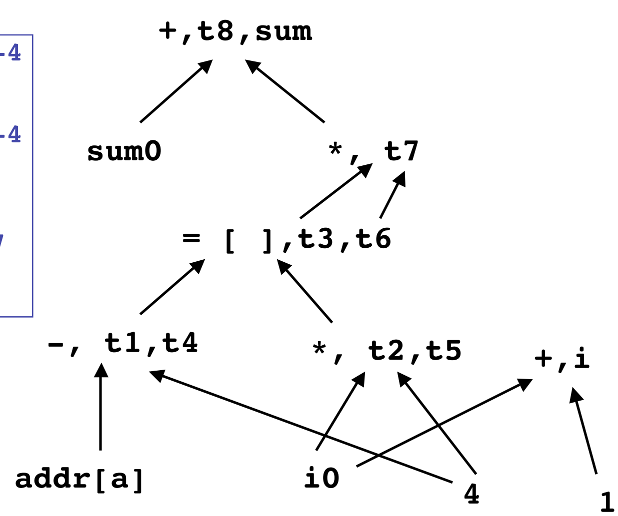
Expression DAG construction




```
t1 = addr[a]-4
t2 = i * 4
t3 = t1[t2]
```


Expression DAG construction

```
t1 = addr[a]-4
t2 = i * 4
t3 = t1[t2]
t4 = addr[a]-4
t5 = i * 4
t6 = t4 [t5]
t7 = t3 * t6
t8 = sum + t7
sum = t8
i = i + 1
```

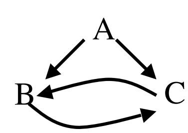


DAG construction

- How to add a subexpression into a partially constructed DAG? A = B + C
- Is there a node already for B + C?
  - If so, add A to its list of labels
  - If not,
    - Is there a node labeled B already? If not, create a leaf labeled B
    - Is there a node labeled C already? If not, create a leaf labeled C
  - Create a node labeled A for + with left child B and right child C

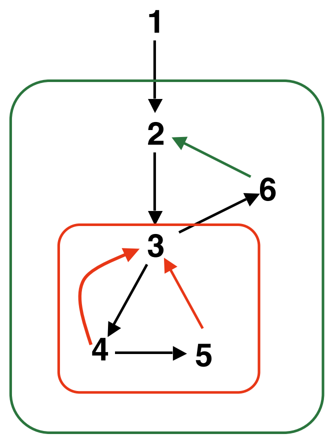
Flow of Control Abstractions

- ***Dominator*** **A node *x*** ***dominates*** **a node *y* if and only if all paths from the control flow graph (CFG) entry node to *y* pass through *x*.**
- ***(Natural) Loop*** **Let (*y*,*x*) be a CFG edge such that *x* dominates *y*. Then all nodes on paths from *x* to *y* are in the loop defined by (*y*,*x*).**
  - (*y*,*x*) is called a *back edge*
  - For *reducible* graphs, the set of back edges is unique
  - CFG is reducible if each loop can be entered through a single node
  - *Irreducible* means contains a subgraph



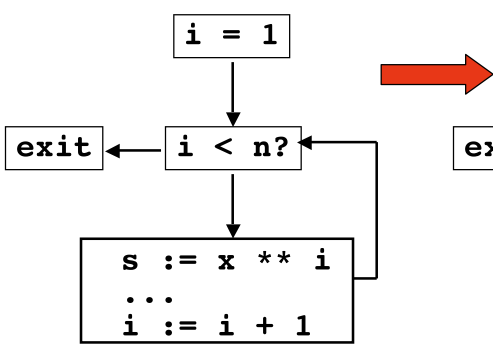
Loops

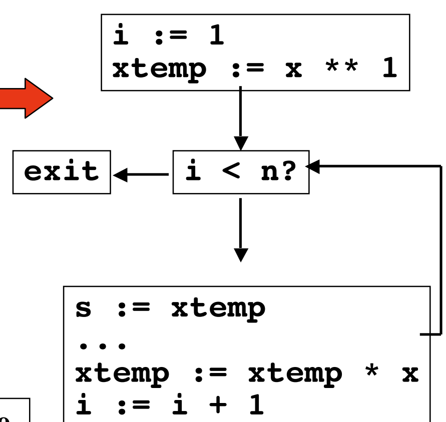
Back edges: (5,3), (4,3), (6,2)
Loop (5,3) = {3, 4, 5}
Loop (4,3) = {3, 4}
} **combined**
Loop (6,2) = {2, 3, 4, 5, 6}



CFG

General Step in Strength Reduction





transforms an repeated expensive operation into a less expensive one

General Code Motion

```
n := 1; k := 0; m := 3; read x;
while n ≤ 10 do
  if 2 + x < 5 then k := 5;
  if 3 + k = 3 then m := m + 2;
  n := n + k + m;
endwhile;
```

definitions within loop are barriers to code motion

General Code Motion

```
n := 1; k := 0; m := 3; read x;
if 2 + x < 5 then k := 5;//move first
t1 := 3 + k = 3 //move second
while n ≤ 10 do
  if 2 + x < 5 then k := 5;
  if 3 + k = 3 then m := m + 2;
  if t1 then m := m + 2;
  n := n + k + m;
endwhile;
```

Why can't we move any more code out of the loop?

Program Analysis

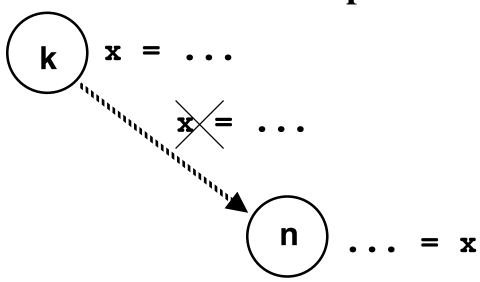
- **Performed at compile-time, deriving something about semantics of program**
- **Termed *flow analysis***
  - **– *Control flow analysis* reveals possible execution paths**
    - **Cannot tell actual feasibility of path. Why not?**
  - **– *Dataflow analysis* determines information about modification, preservation, and use of data entities in a program**

Four Classical Data Flow Problems

- **Reaching definitions, Live uses of variables, Available expressions, Very Busy Expressions**
- **Def-use and Use-def chains, built from Reach and Live, used for many optimizations**
- **Avail enables global common subexpression elimination**
- **VeryB was used for conservative code motion**

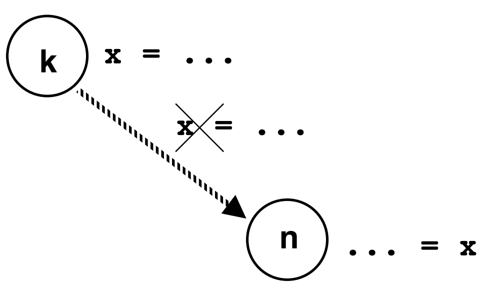
Reaching Definitions

- *Definition* A statement which may change the value of a variable
- A definition of a variable *x* at node *k* *reaches* node *n* if there is a definition-clear path from *k* to *n*.



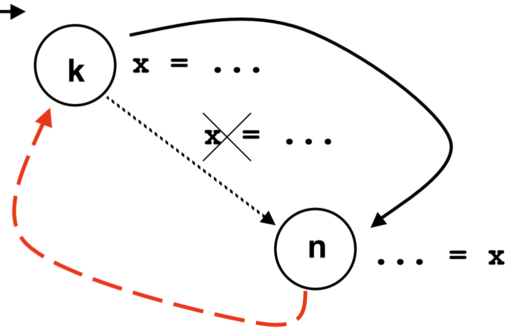
Live Uses of Variables

- *Use* Appearance of a variable as an operand of a 3 address statement
- A use of a variable *x* at node n is *live on exit* from node *k* if there is a definition-clear path for *x* from *k* to *n*.



Def-use Relations

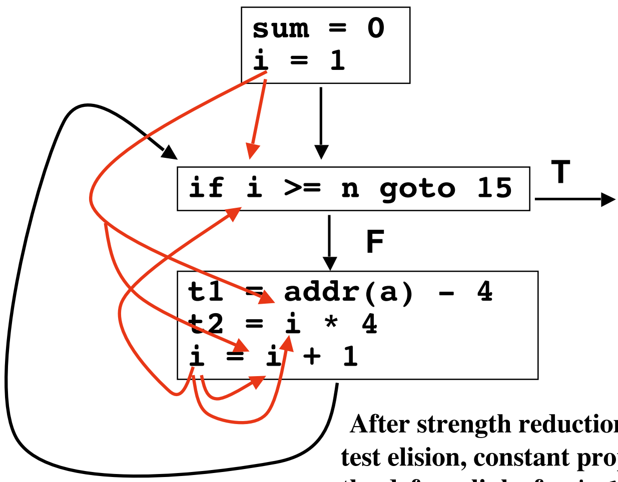
- Use-def chain links an use to a definition that reaches that use
- Def-use chain links a definition to an use that it reaches



Optimizations Enabled by Def-use

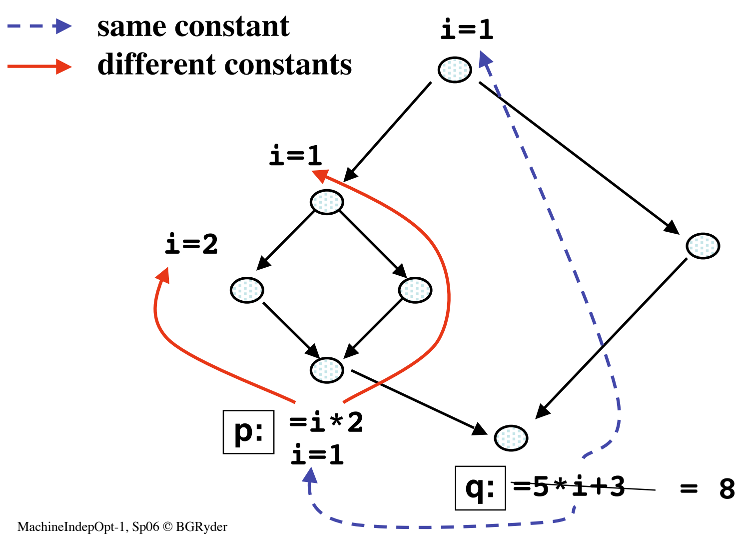
- Dead code elimination (Def-use)
- Code motion (Use-def)
- Strength reduction (Use-def)
- Test elision (Use-def)
- Constant propagation (Use-def)
- Copy propagation (Def-use)

Dead Code Elimination



After strength reduction, test elision, constant propagation the def-use links for `i=1` disappear and it becomes dead code.

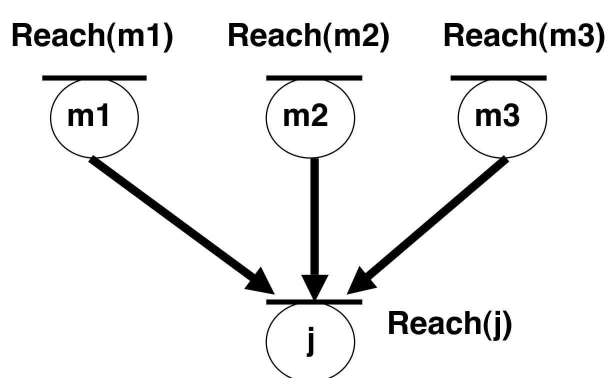
Constant Propagation



Classical Dataflow Problems

- How to formulate analysis from CFG to dataflow equations?
- *Forward* and *backward* dataflow problems
- *May* and *must* dataflow problems

Reaching Definitions



forward, may dataflow problem

Reaching Definitions Equations

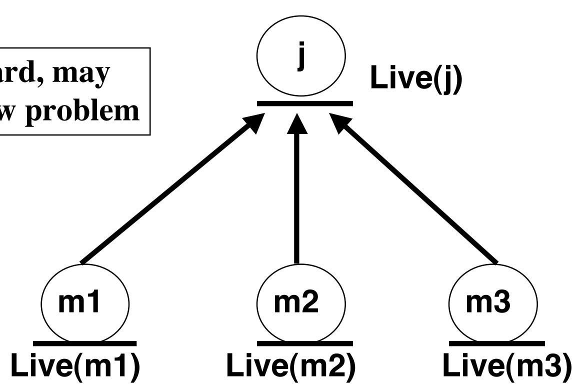
$$Reach(j) = \bigcup_{m \in Pred(j)} \{ Reach(m) \cap pres(m) \cup dgen(m) \}$$

where:

pres(m) is the set of defs preserved through node *m*
dgen(m) is the set of defs generated at node *m*
Pred(j) is the set of immediate predecessors of node *j*

Live Uses of Variables

backward, may dataflow problem



Live Uses Equations

$$Live(j) = \bigcup_{m \in Succ(j)} \{ Live(m) \cap pres(m) \cup ugen(m) \}$$

where

pres(m) is the set of uses preserved through node *m*

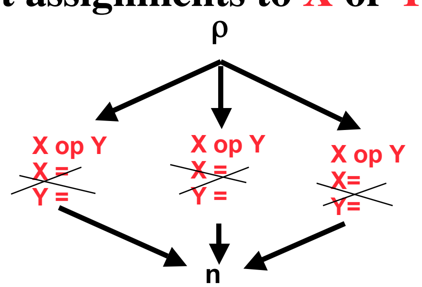
(these will correspond to variables whose defs are preserved)

ugen(m) is the set of uses generated at node *m*

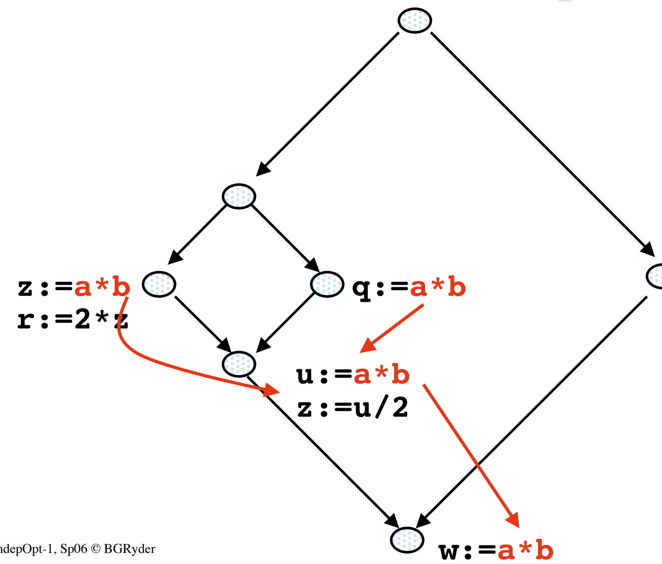
succ(j) is the set of immediate successors of node *j*

Available Expressions

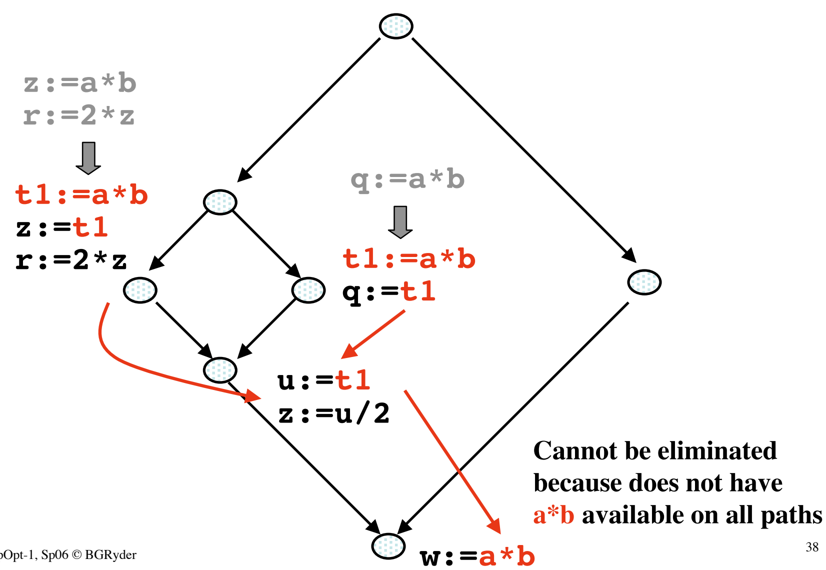
- **An expression X op Y is *available* at program point *n* if EVERY path from program entry to *n* evaluates X op Y, and after every evaluation prior to reaching *n*, there are NO subsequent assignments to X or Y.**



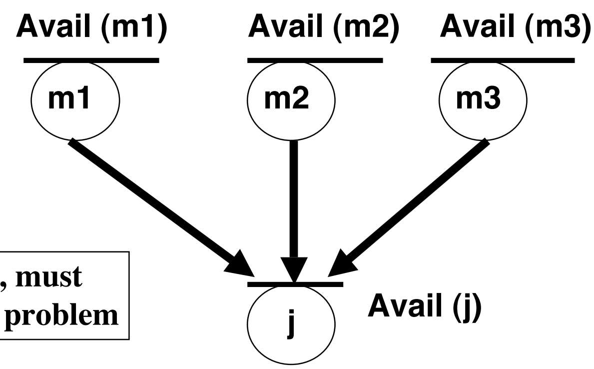
Global Common Subexpressions



Global CSE



Available Expressions



Forward, must dataflow problem

Available Expressions Equations

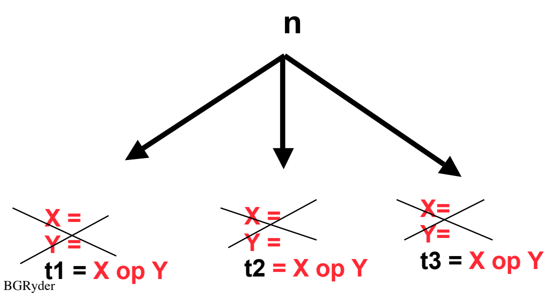
$$Avail(j) = \bigcap_{m \in Pred(j)} \{ Avail(m) \cap epres(m) \cup egen(m) \}$$

where:

- **epres(m) is the set of expressions preserved through node *m***
- **egen(m) is the set of (downwards exposed) expressions generated at node *m***
- **pred(j) is the set of immediate predecessors of node *j***

Very Busy Expressions

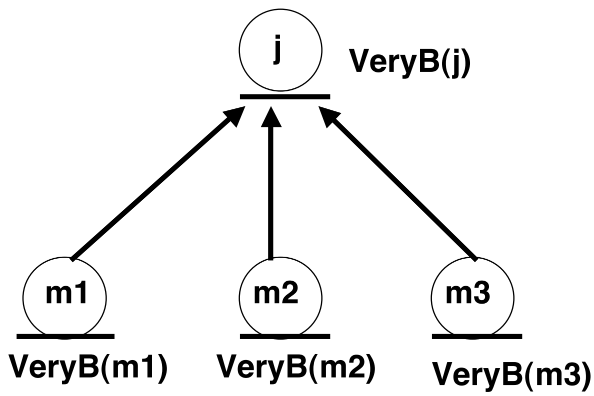
- An expression X op Y is very busy at program point *n*, if along EVERY path from *n*, we come to a computation of X op Y BEFORE any redefinition of X or Y.



Code Hoisting

- SAFETY: Assume X op Y is in VeryB(n) and *n* dominates all expression calculations that are hoisting candidates *p*.
- For every X op Y at program point *p*, trace backwards from *p* to *n* to ensure there is a path from *n*--> *p* without any definitions of X, Y, X op Y
- Hoist (Calculate t=X op Y) at exit of node *n*; change candidate calculations from s = X op Y to s = t.
- PROFITABILITY: Check that copy propagation can eliminate all copies introduced in the previous step. If not, undo the hoist.

Very Busy Expressions



Very Busy Equations

$$VeryB(j) = \bigcap_{m \in Succ(j)} \{ VeryB(m) \cap epres(m) \cup vgen(m) \}$$

where:

epres(m) is the set of expressions preserved through node *m*

vgen(m) is the set of (upwards exposed) expressions generated at node *m*

succ(j) is the set of immediate successors of node *j*

Dataflow Problems

	May Problems	Must Problems
Forward Problems	Reaching Defs	Available Exprs
Backward Problems	Live Uses of Variables	Very Busy Expressions