CSE P 501 – Compilers

Dataflow Analysis

Hal Perkins

Spring 2018

Agenda

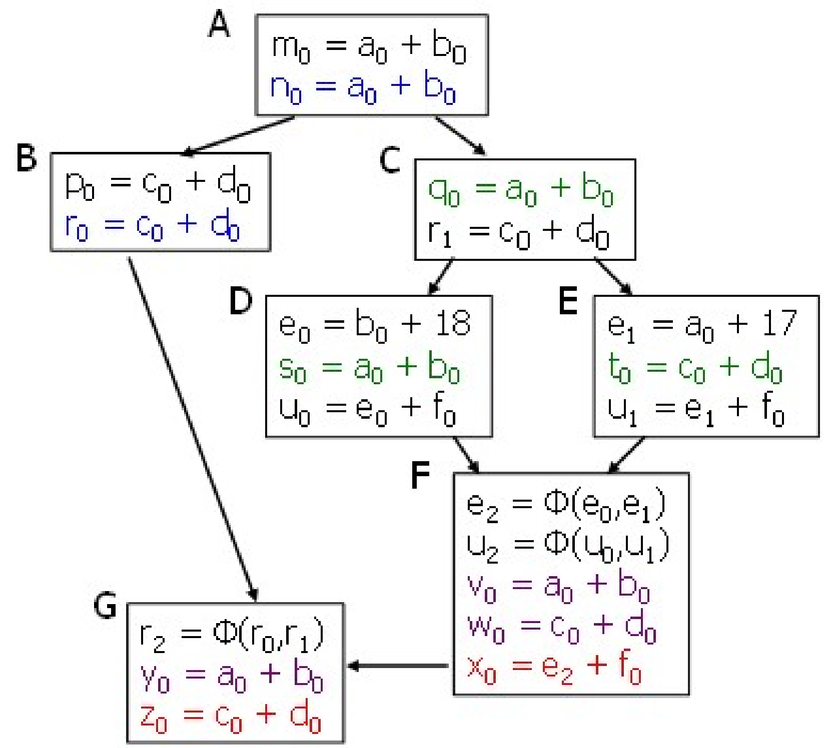
- Dataflow analysis: a framework and algorithm for many common compiler analyses
- Initial example: dataflow analysis for common subexpression elimination
- Other analysis problems that work in the same framework
- Some of these are optimizations we've seen, but now more formally and with details

The Story So Far…

- Redundant expression elimination
  - Local Value Numbering
  - Superlocal Value Numbering
    - Extends VN to EBBs
    - SSA-like namespace
  - Dominator VN Technique (DVNT)
- All of these propagate along forward edges
- None are global
  - In particular, can't handle back edges (loops)

Dominator Value Numbering

- Most sophisticated algorithm so far
- Still misses some opportunities
- Can't handle loops

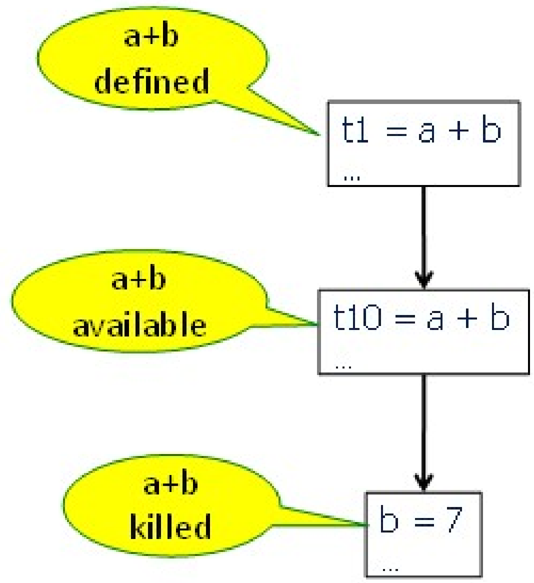


Available Expressions

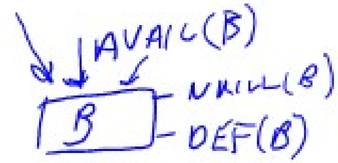
- Goal: use dataflow analysis to find common subexpressions whose range spans basic blocks
- Idea: calculate *available expressions* at beginning of each basic block
- Avoid re-evaluation of an available expression
  – use a copy operation

“Available” and Other Terms

- An expression *e* is *defined* at point *p* in the CFG if its value is computed at *p*
  - Sometimes called *definition site*
- An expression *e* is *killed* at point *p* if one of its operands is defined at *p*
  - Sometimes called *kill site*
- An expression *e* is *available* at point *p* if every path leading to *p* contains a prior definition of *e* and *e* is not killed between that definition and *p*

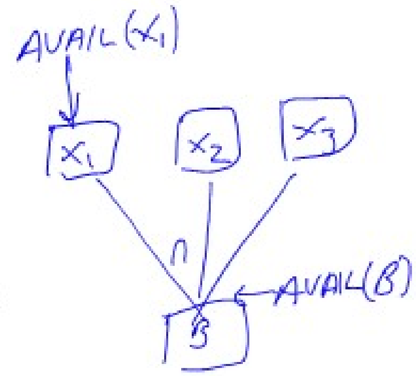


Available Expression Sets



- To compute available expressions, for each block *b*, define
  - AVAIL(b) – the set of expressions available on entry to *b*
  - NKILL(b) – the set of expressions not killed in *b*
    - i.e., all expressions in the program *except* for those killed in *b*
  - DEF(b) – the set of expressions defined in *b* and not subsequently killed in *b*

Computing Available Expressions



- AVAIL(b) is the set
  - $\text{AVAIL}(\mathbf{b}) = \bigcap_{x \in \text{preds}(b)} (\text{DEF}(x) \cup (\text{AVAIL}(x) \cap \text{NKILL}(x)))$
  - preds(b) is the set of b's predecessors in the CFG
  - The set of expressions available on entry to *b* is the set of expressions that were available at the end of *every* predecessor basic block *x*
  - The expressions available on exit from block *b* are those defined in *b* or available on entry to *b* and not killed in *b*
- This gives a system of simultaneous equations – a dataflow problem

Name Space Issues

- In previous value-numbering algorithms, we used a SSA-like renaming to keep track of versions
- In global dataflow problems, we use the original namespace
  - we require a+b have the same value along *all* paths to its use
  - If a or b is updated along *any* path to its use, then a+b has the "wrong" value
  - so original names are exactly what we want
- The KILL information captures when a value is no longer available

Computing Available Expressions

- Big Picture
  - Build control-flow graph
  - Calculate initial local data – DEF(b) and NKILL(b)
    - This only needs to be done once for each block b and depends only on the statements in b
  - Iteratively calculate AVAIL(b) by repeatedly evaluating equations until nothing changes
    - Another fixed-point algorithm

Computing DEF and NKILL (1)

- For each block *b* with operations o_1, o_2, …, o_k

  KILLED = ∅   // killed *variables*, not expressions

  DEF(b) = ∅

  for i = k to 1   // note: working back to front

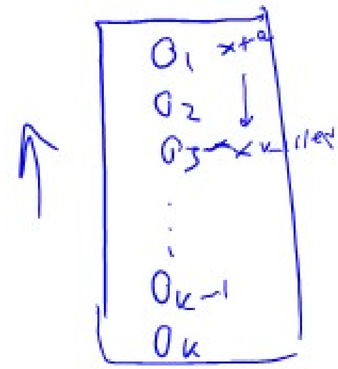
    assume o_i is "x = y + z"

    if (y ∉ KILLED and z ∉ KILLED)

      add "y + z" to DEF(b)

    add x to KILLED

  …

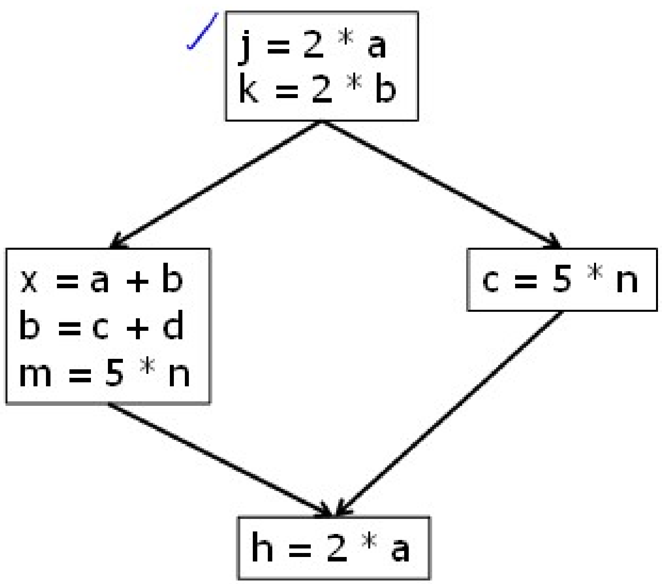


Computing DEF and NKILL (2)

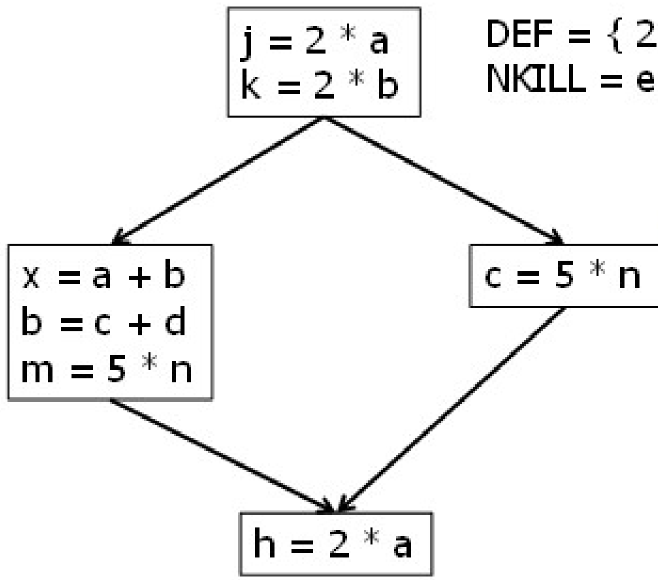
- After computing DEF and KILLED for a block *b*, compute set of all expressions in the program not killed in *b*

NKILL(*b*) = { all expressions }
for each expression *e*
  for each variable $v \in$ e
    if $v \in$ KILLED then
      NKILL(*b*) = NKILL(*b*) - *e*

Example: Compute DEF and NKILL

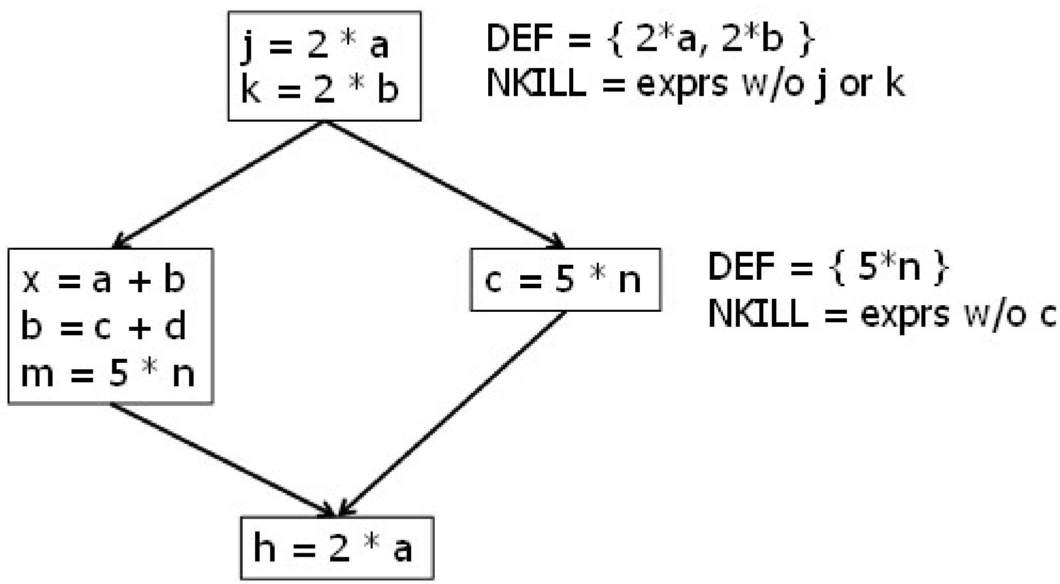


Example: Compute DEF and NKILL

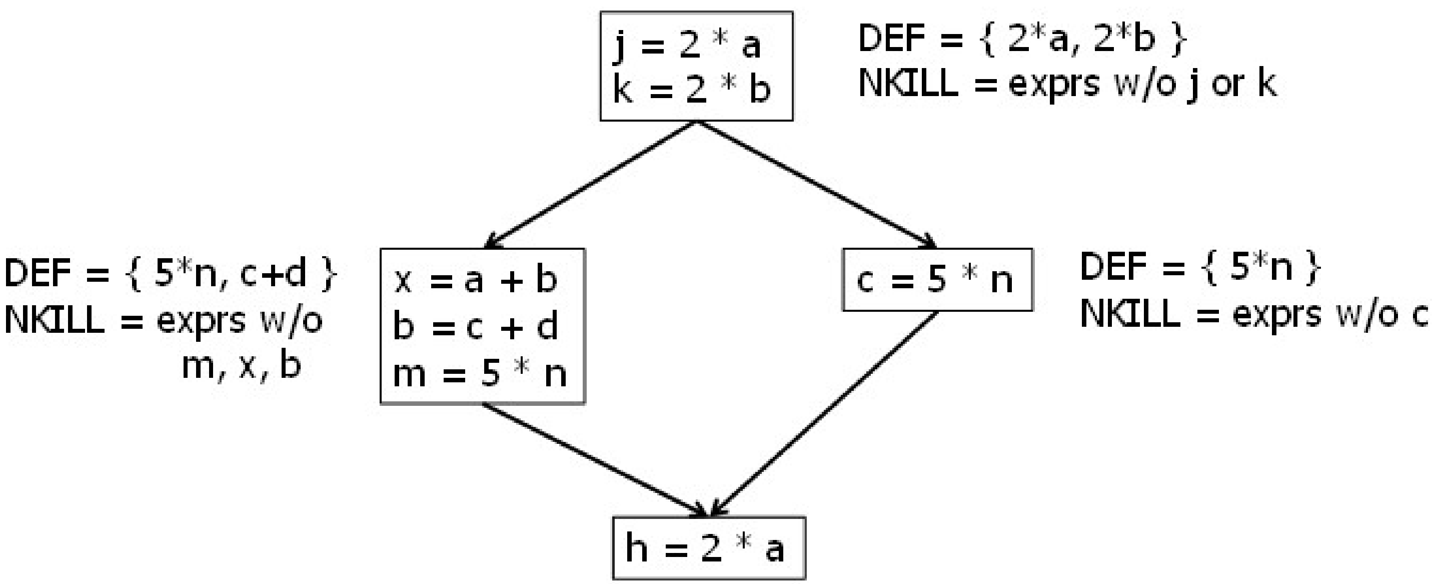


DEF = { 2*a, 2*b }
NKILL = exprs w/o j or k

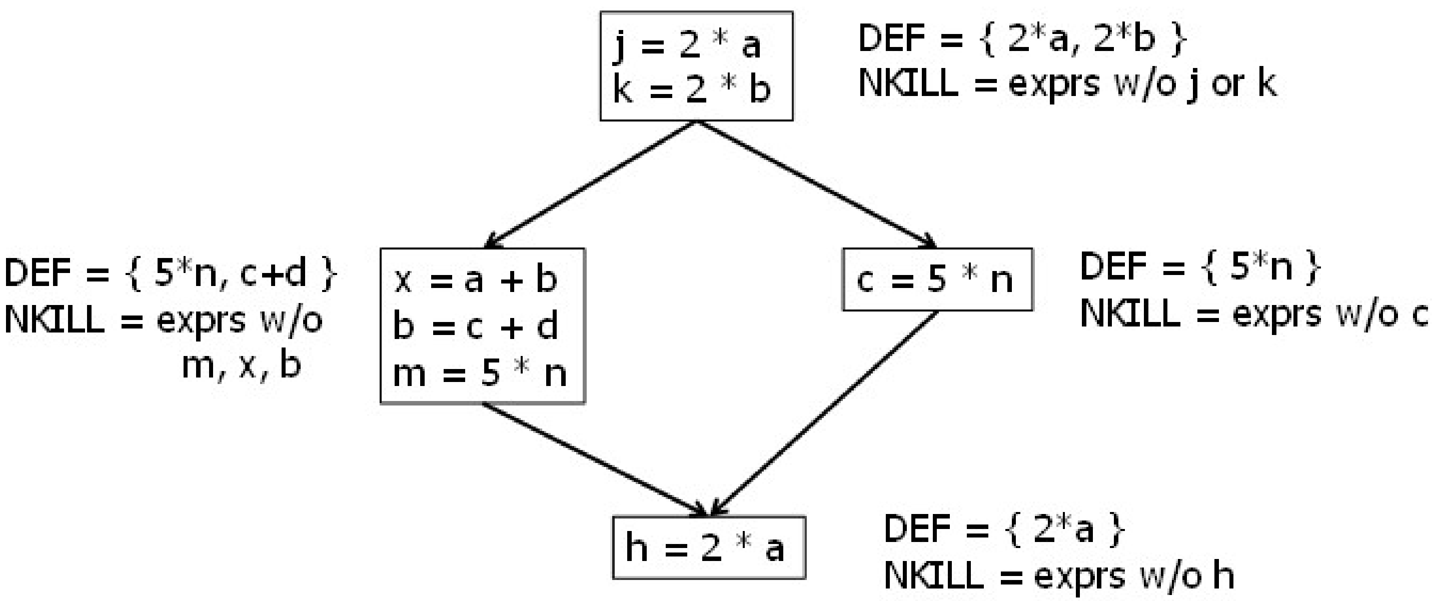
Example: Compute DEF and NKILL



Example: Compute DEF and NKILL



Example: Compute DEF and NKILL



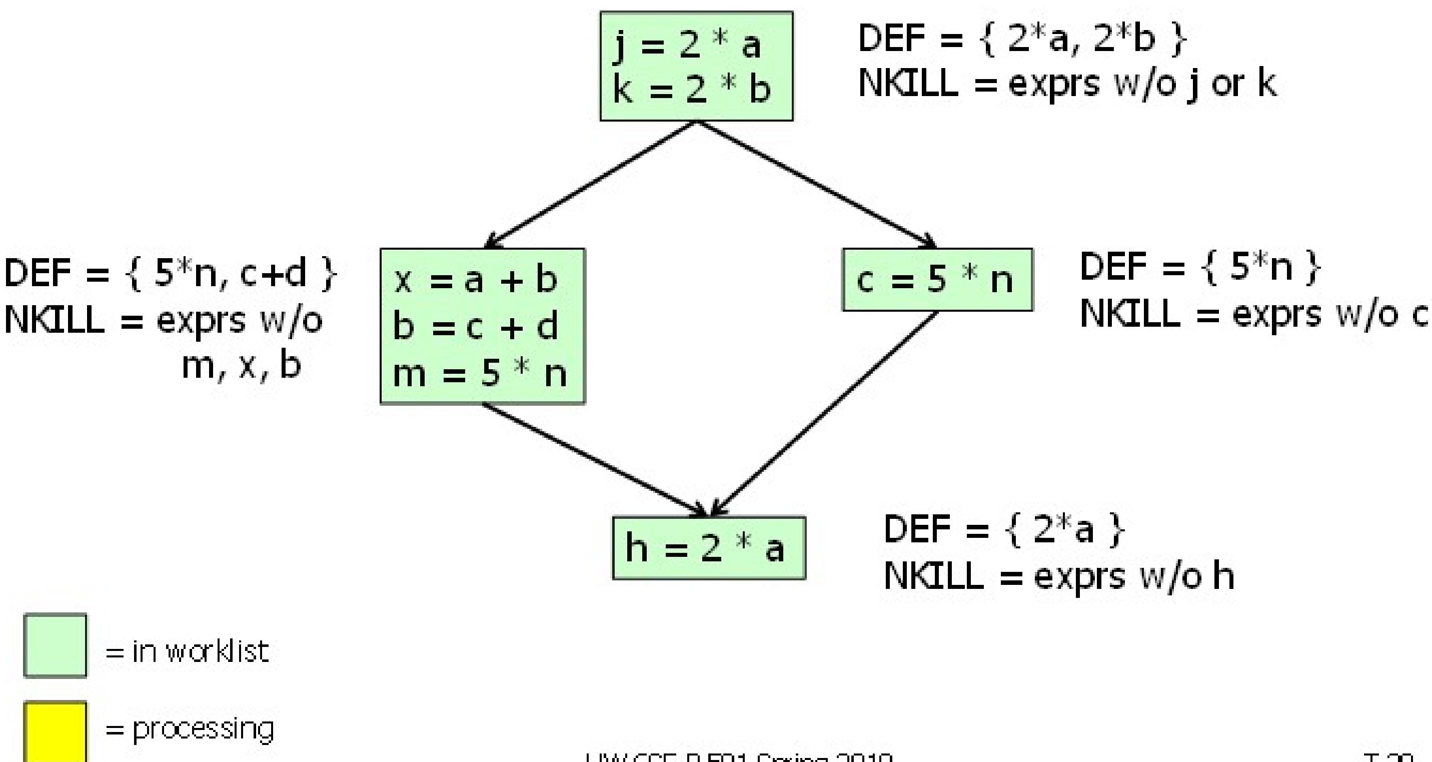
Computing Available Expressions

Once DEF(b) and NKILL(b) are computed for all blocks b

Worklist = { all blocks b_i }
while (Worklist $\neq \emptyset$)
    remove a block *b* from Worklist
    recompute AVAIL(*b*)
    if AVAIL(*b*) changed
        Worklist = Worklist $\cup$ successors(*b*)

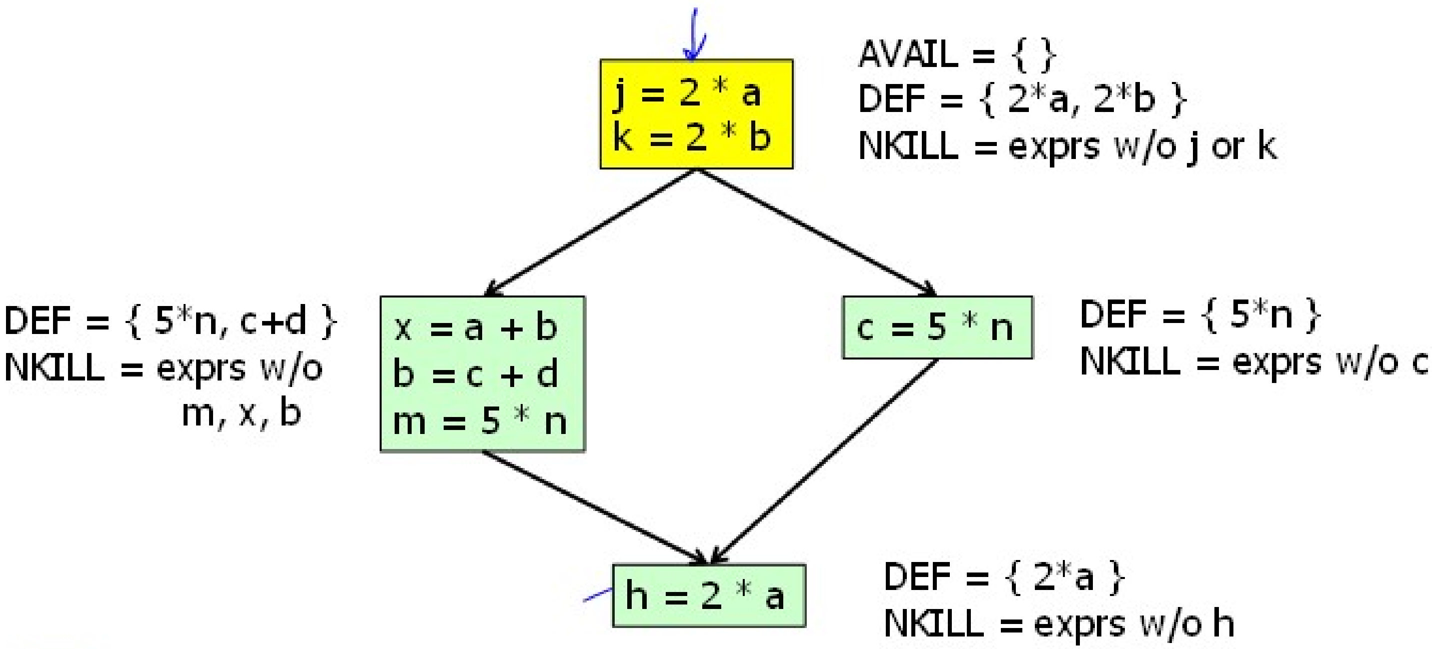
Example: Find Available Expressions

– AVAIL(b) = $\cap_{x \in preds(b)}$ (DEF(x) $\cup$ (AVAIL(x) $\cap$ NKILL(x)))



Example: Find Available Expressions

$$\text{AVAIL}(b) = \bigcap_{x \in preds(b)} \left(\text{DEF}(x) \cup (\text{AVAIL}(x) \cap \text{NKILL}(x))\right)$$

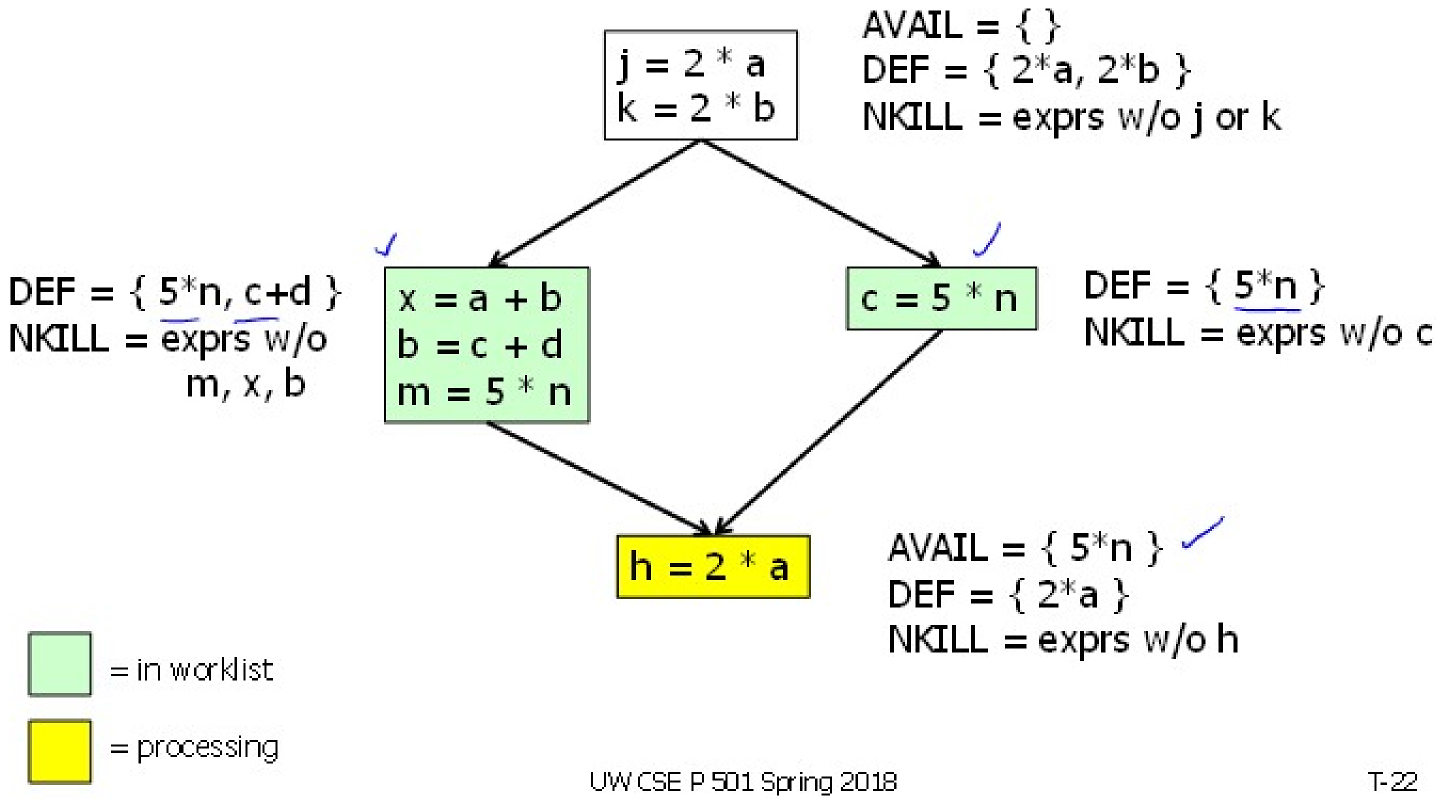


= in worklist

= processing

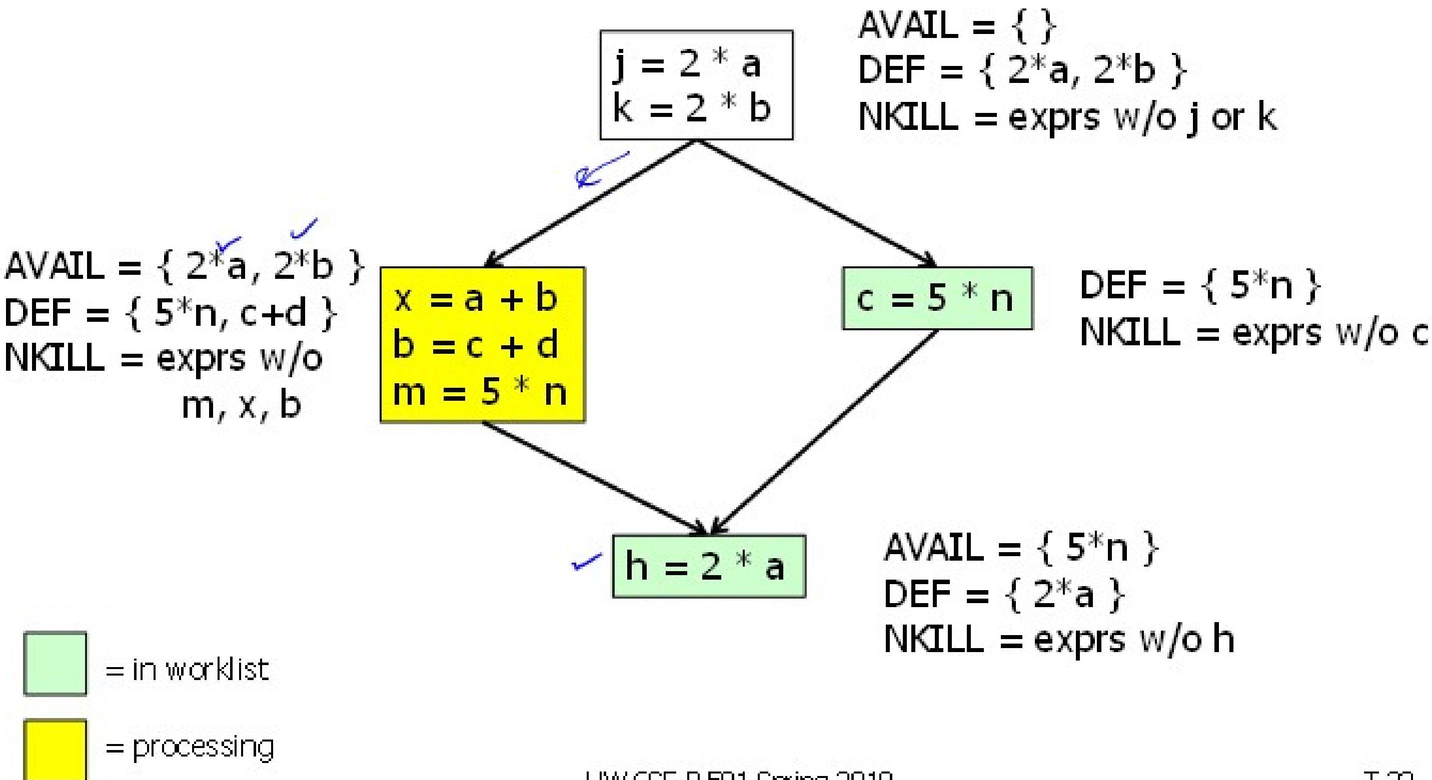
Example: Find Available Expressions

$\text{AVAIL}(b) = \bigcap_{x \in preds(b)} (\text{DEF}(x) \cup (\text{AVAIL}(x) \cap \text{NKILL}(x)))$



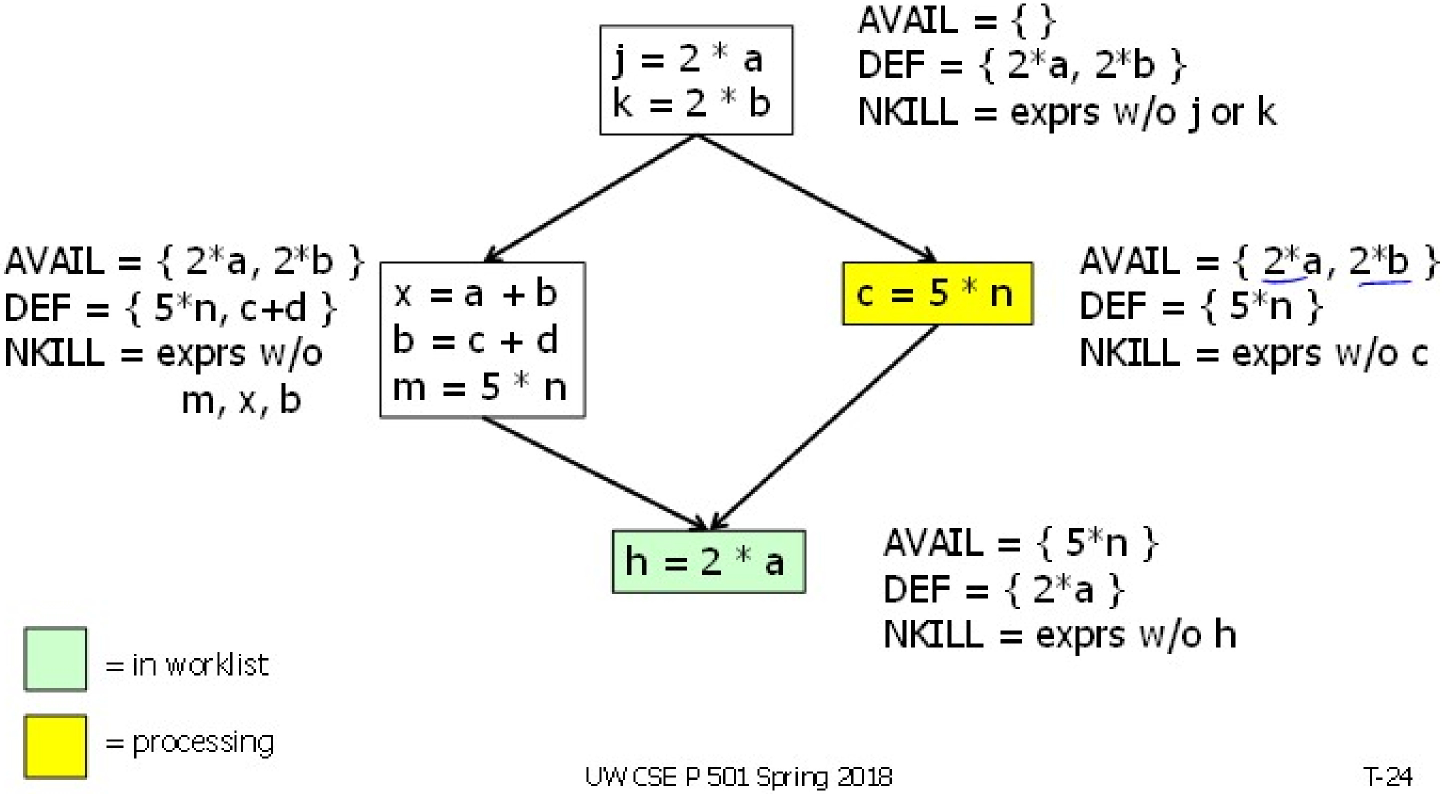
Example: Find Available Expressions

$$AVAIL(b) = \bigcap_{x \in preds(b)} (DEF(x) \cup (AVAIL(x) \cap NKILL(x)))$$



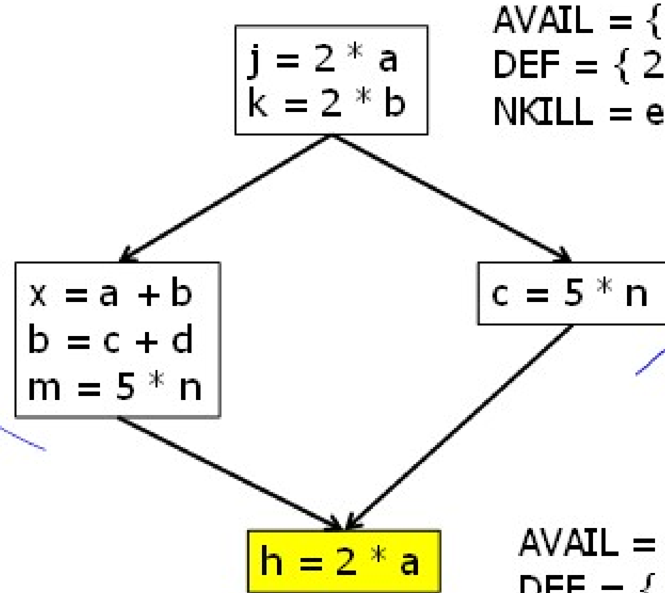
Example: Find Available Expressions

AVAIL(b) = $\cap_{x \in preds(b)}$ (DEF(x) $\cup$ (AVAIL(x) $\cap$ NKILL(x)))



Example: Find Available Expressions

$\text{AVAIL}(b) = \bigcap_{x \in preds(b)} (\text{DEF}(x) \cup (\text{AVAIL}(x) \cap \text{NKILL}(x)))$



AVAIL = { }
DEF = { 2*a, 2*b }
NKILL = exprs w/o j or k

AVAIL = { 2*a, 2*b }
DEF = { 5*n, c+d }
NKILL = exprs w/o m, x, b

AVAIL = { 2*a, 2*b }
DEF = { 5*n }
NKILL = exprs w/o c

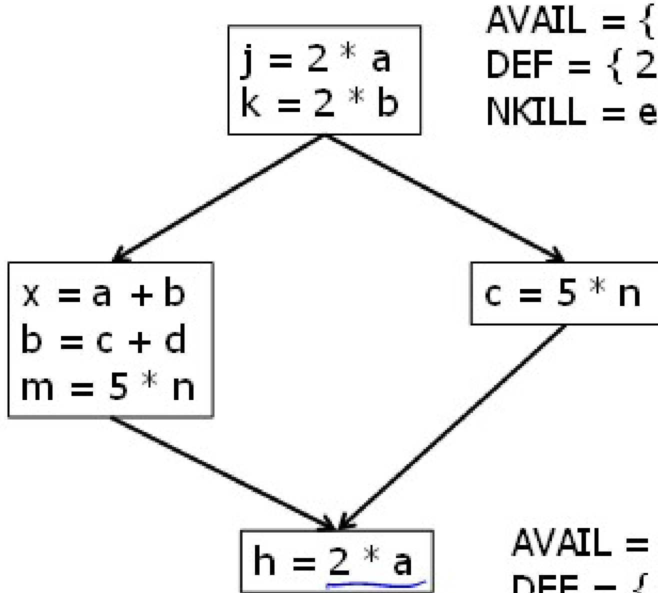
AVAIL = { 5*n, 2*a }
DEF = { 2*a }
NKILL = exprs w/o h

= in worklist

= processing

Example: Find Available Expressions

AVAIL(b) = $\cap_{x \in preds(b)}$ (DEF(x) $\cup$ (AVAIL(x) $\cap$ NKILL(x)))



AVAIL = { }
DEF = { 2*a, 2*b }
NKILL = exprs w/o j or k

AVAIL = { 2*a, 2*b }
DEF = { 5*n, c+d }
NKILL = exprs w/o m, x, b

AVAIL = { 2*a, 2*b }
DEF = { 5*n }
NKILL = exprs w/o c

AVAIL = { 5*n, 2*a }
DEF = { 2*a }
NKILL = exprs w/o h

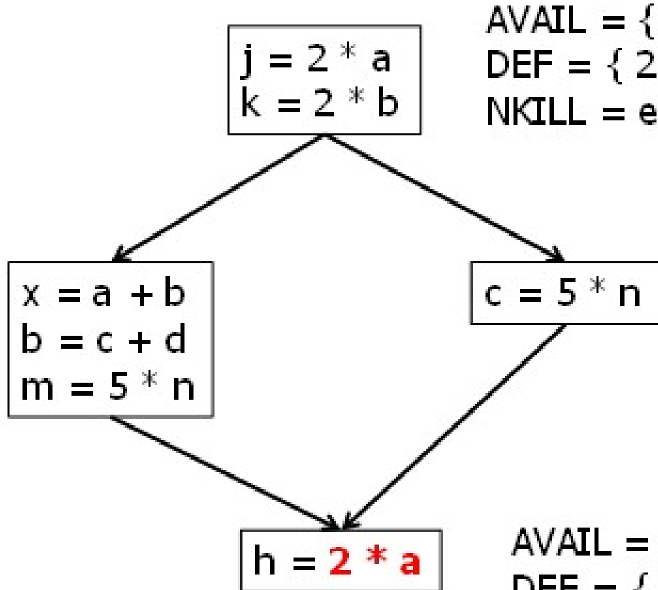
= in worklist

= processing

And the common subexpression is???

Example: Find Available Expressions

$AVAIL(b) = \bigcap_{x \in preds(b)} (DEF(x) \cup (AVAIL(x) \cap NKILL(x)))$



AVAIL = { }
DEF = { 2*a, 2*b }
NKILL = exprs w/o j or k

AVAIL = { 2*a, 2*b }
DEF = { 5*n, c+d }
NKILL = exprs w/o m, x, b

AVAIL = { 2*a, 2*b }
DEF = { 5*n }
NKILL = exprs w/o c

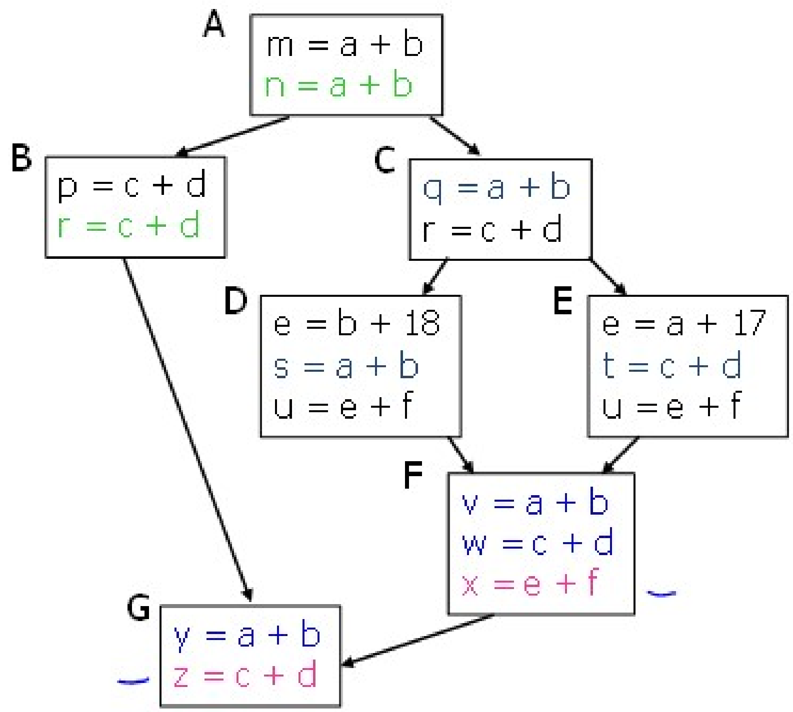
AVAIL = { 5*n, **2*a** }
DEF = { 2*a }
NKILL = exprs w/o h

= in worklist

= processing

Comparing Algorithms

- LVN – Local Value Numbering
- SVN – Superlocal Value Numbering
- DVN – DominatoT-based Value Numbering
- GRE – Global Redundancy Elimination



Comparing Algorithms (2)

- LVN => SVN => DVN form a strict hierarchy – later algorithms find a superset of previous information
- Global RE finds a somewhat different set
  - Discovers e+f in F (computed in both D and E)
  - Misses identical values if they have different names (e.g., a+b and c+d when a=c and b=d)
    - Value Numbering catches this

Scope of Analysis

- Larger context (EBBs, regions, global, interprocedural) sometimes helps
  - More opportunities for optimizations
- But not always
  - Introduces uncertainties about flow of control
  - Usually only allows weaker analysis
  - Sometimes has unwanted side effects
    - Can create additional pressure on registers, for example

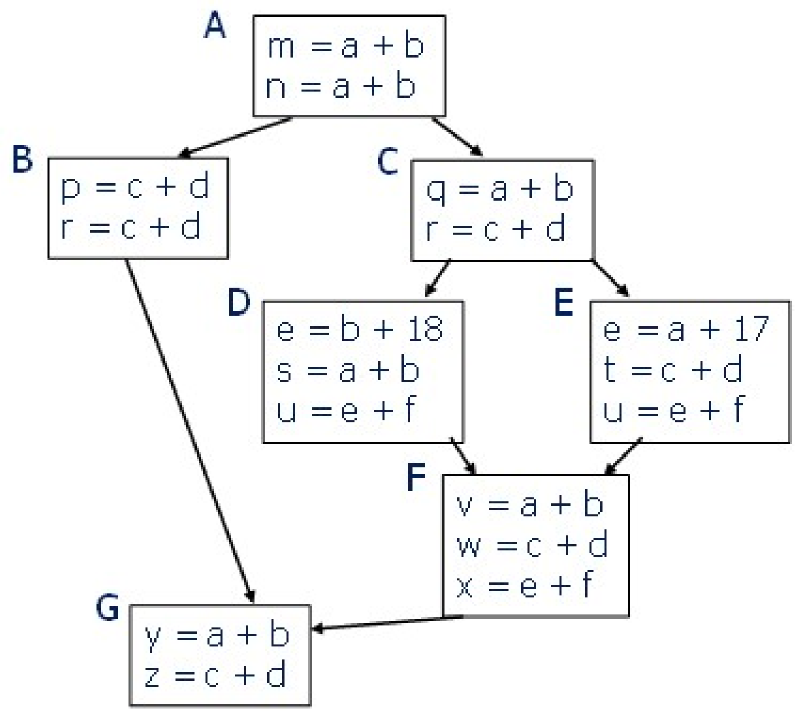
Code Replication

- Sometimes replicating code increases opportunities – modify the code to create larger regions with simple control flow
- Two examples
  – Cloning
  – Inline substitution

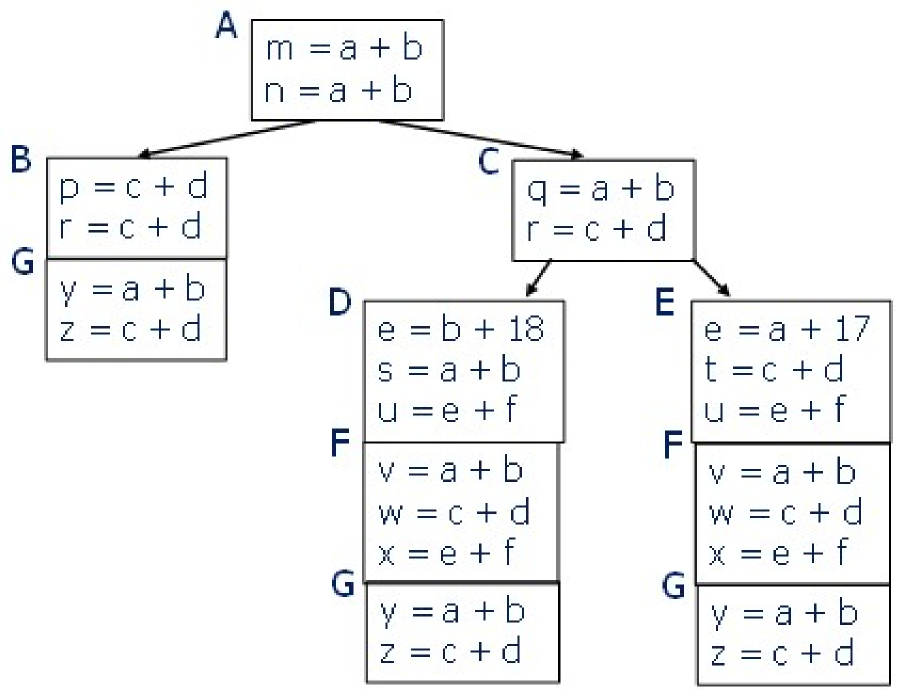
Cloning

- Idea: duplicate blocks with multiple predecessors
- Tradeoff
  - More local optimization possibilities – larger blocks, fewer branches
  - But: larger code size, may slow down if it interacts badly with cache

Original VN Example



Example with cloning



Inline Substitution

- Problem: an optimizer has to treat a procedure call as if it (could have) modified all globally reachable data
  - Plus there is the basic expense of calling the procedure
- Inline Substitution: replace each call site with a copy of the called function body

Inline Substitution Issues

- Pro
  - More effective optimization – better local context and don't need to invalidate local assumptions
  - Eliminate overhead of normal function call
- Con
  - Potential code bloat
  - Need to manage recompilation when either caller or callee changes

Dataflow analysis

- Available expressions are an example of a *dataflow analysis* problem
- Many similar problems can be expressed in a similar framework
- Only the first part of the story – once we've discovered facts, we then need to use them to improve code

Characterizing Dataflow Analysis

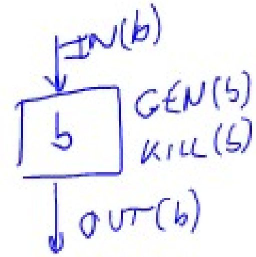
- All of these algorithms involve sets of facts about each basic block b

  IN(b) – facts true on entry to b

  OUT(b) – facts true on exit from b

  GEN(b) – facts created and not killed in b

  KILL(b) – facts killed in b



- These are related by the equation

  OUT(b) = GEN(b) ∪ (IN(b) – KILL(b))

  – Solve this iteratively for all blocks

  – Sometimes information propagates forward; sometimes backward

Dataflow Analysis (1)

- A collection of techniques for compile-time reasoning about run-time values
- Almost always involves building a graph
  - Trivial for basic blocks
  - Control-flow graph or derivative for global problems
  - Call graph or derivative for whole-program problems

Dataflow Analysis (2)

- Usually formulated as a set of *simultaneous equations* (dataflow problem)
  - Sets attached to nodes and edges
  - Need a lattice (or semilattice) to describe values
    - In particular, has an appropriate operator to combine values and an appropriate "bottom" or minimal value

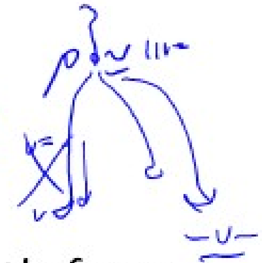
Dataflow Analysis (3)

- Desired solution is usually a *meet over all paths* (MOP) solution
  - "What is true on every path from entry"
  - "What can happen on any path from entry"
  - Usually relates to safety of optimization

Dataflow Analysis (4)

- Limitations
  - Precision – "up to symbolic execution"
    - Assumes all paths taken
  - Sometimes cannot afford to compute full solution
  - Arrays – classic analysis treats each array as a single fact
  - Pointers – difficult, expensive to analyze
    - Imprecision rapidly adds up
    - But gotta do it to effectively optimize things like C/C++
- For scalar values we can quickly solve simple problems

Example:Live Variable Analysis



- A variable *v* is *live* at point *p* iff there is *any* path from *p* to a use of *v* along which *v* is not redefined
- Some uses:
  - Register allocation – only live variables need a register
  - Eliminating useless stores – if variable not live at store, then stored variable will never be used
  - Detecting uses of uninitialized variables – if live at declaration (before initialization) then it might be used uninitialized
  - Improve SSA construction – only need Φ-function for variables that are live in a block (later)

Liveness Analysis Sets

- For each block b, define
  - use[*b*] = variable used in *b* before any def
  - def[*b*] = variable defined in *b* & not killed
  - in[*b*] = variables live on entry to *b*
  - out[*b*] = variables live on exit from *b*

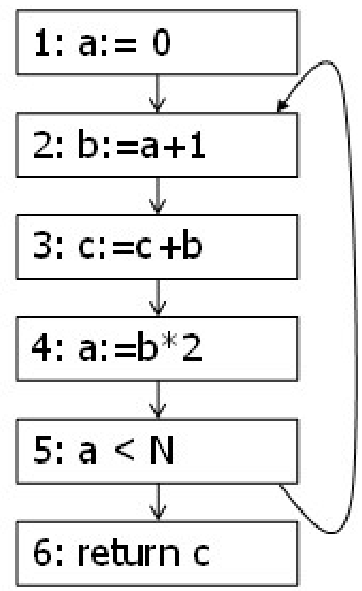
Equations for Live Variables

- Given the preceding definitions, we have
  - ✓ $in[b] = use[b] \cup (out[b] - def[b])$
  - $out[b] = \cup_{s \in succ[b]} in[s]$
- Algorithm
  - Set $in[b] = out[b] = \emptyset$
  - Update in, out until no change

Example (1 stmt per block)

- Code

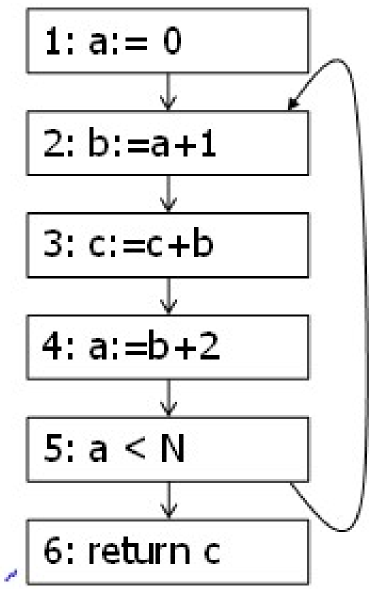
```
    a := 0
L:  b := a+1
    c := c+b
    a := b*2
    if a < N goto L
    return c
```



$$\begin{cases} in[b] = use[b] \cup (out[b] - def[b]) \\ out[b] = \cup_{s \in succ[b]}\, in[s] \end{cases}$$

Calculation

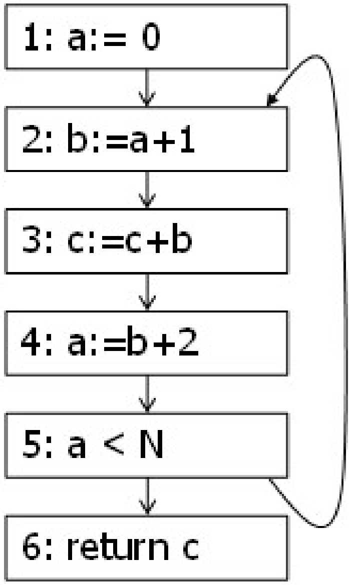
			I		II		III	
block	use	def	out	in	out	in	out	in
6	c	—	—	c	—	c		
5	a	—	c	a, c	c, a	a, c		
4	b	a	a, c	b, c	a, c	b, c	same	
3	b, c	c	b, c	b, c	b, c	b, c		
2	a	b	b, c	a, c	b, c	a, c		
1	—	a	a, c	c	a, c	c		



$$in[b] = use[b] \cup (out[b] - def[b])$$
$$out[b] = \cup_{s \in succ[b]} in[s]$$

Calculation

			I		II		III	
block	use	def	out	in	out	in	out	in
6	c	--	–	c	--	c		
5	a	--	c	a,c	a,c	a,c		
4	b	a	a,c	b,c	a,c	b,c		
3	b,c	c	b,c	b,c	b,c	b,c		
2	a	b	b,c	a,c	b,c	a,c		
1	--	a	a,c	c	a,c	c		



$$in[b] = use[b] \cup (out[b] - def[b])$$
$$out[b] = \cup_{s \in succ[b]} in[s]$$

Equations for Live Variables v2

- Many problems have more than one formulation. For example, Live Variables…
- Sets
  - USED(b) – variables used in b before being defined in b
  - NOTDEF(b) – variables not defined in b
  - LIVE(b) – variables live on *exit* from b
- Equation

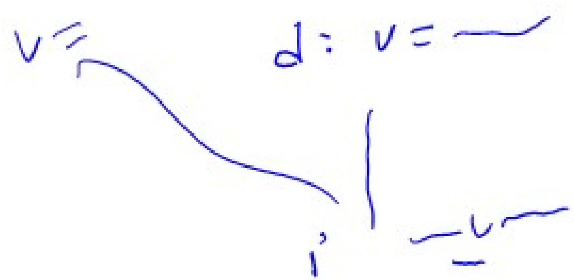
$$\text{LIVE}(b) = \cup_{s \in \text{succ}(b)} \text{USED}(s) \cup (\text{LIVE}(s) \cap \text{NOTDEF}(s))$$

Efficiency of Dataflow Analysis

- The algorithms eventually terminate, but the expected time needed can be reduced by picking a good order to visit nodes in the CFG
  - Forward problems – reverse postorder
  - Backward problems – postorder

Example: Reaching Definitions

- A definition *d* of some variable *v* *reaches* operation *i* iff *i* reads the value of *v* and there is a path from *d* to *i* that does not define *v*
- Uses
  - Find all of the possible definition points for a variable in an expression



Equations for Reaching Definitions

- Sets
  - DEFOUT(b) – set of definitions in b that reach the end of b (i.e., not subsequently redefined in b)
  - SURVIVED(b) – set of all definitions not obscured by a definition in b
  - REACHES(b) – set of definitions that reach b
- Equation

$$\text{REACHES}(b) = \cup_{p \in \text{preds}(b)} \text{DEFOUT}(p) \cup (\text{REACHES}(p) \cap \text{SURVIVED}(p))$$

Example: Very Busy Expressions

- An expression *e* is considered *very busy* at some point *p* if *e* is evaluated and used along every path that leaves *p*, and evaluating *e* at *p* would produce the same result as evaluating it at the original locations

- Uses
  - Code hoisting – move *e* to *p* (reduces code size; no effect on execution time)

Equations for Very Busy Expressions

- Sets
  - USED(b) – expressions used in b before they are killed
  - KILLED(b) – expressions redefined in b before they are used
  - VERYBUSY(b) – expressions very busy on exit from b
- Equation

$$\text{VERYBUSY}(b) = \bigcap_{s \in succ(b)} \text{USED}(s) \cup (\text{VERYBUSY}(s) - \text{KILLED}(s))$$

Using Dataflow Information

- A few examples of possible transformations…

Classic Common-Subexpression Elimination (CSE)

- In a statement s: t := x op y, if x op y is *available* at s then it need not be recomputed
- Analysis: compute *reaching expressions* i.e., statements n: v := x op y such that the path from n to s does not compute x op y or define x or y

Classic CSE Transformation

- If x op y is defined at n and reaches s
  - Create new temporary w
  - Rewrite n: v := x op y as

    n: w := x op y

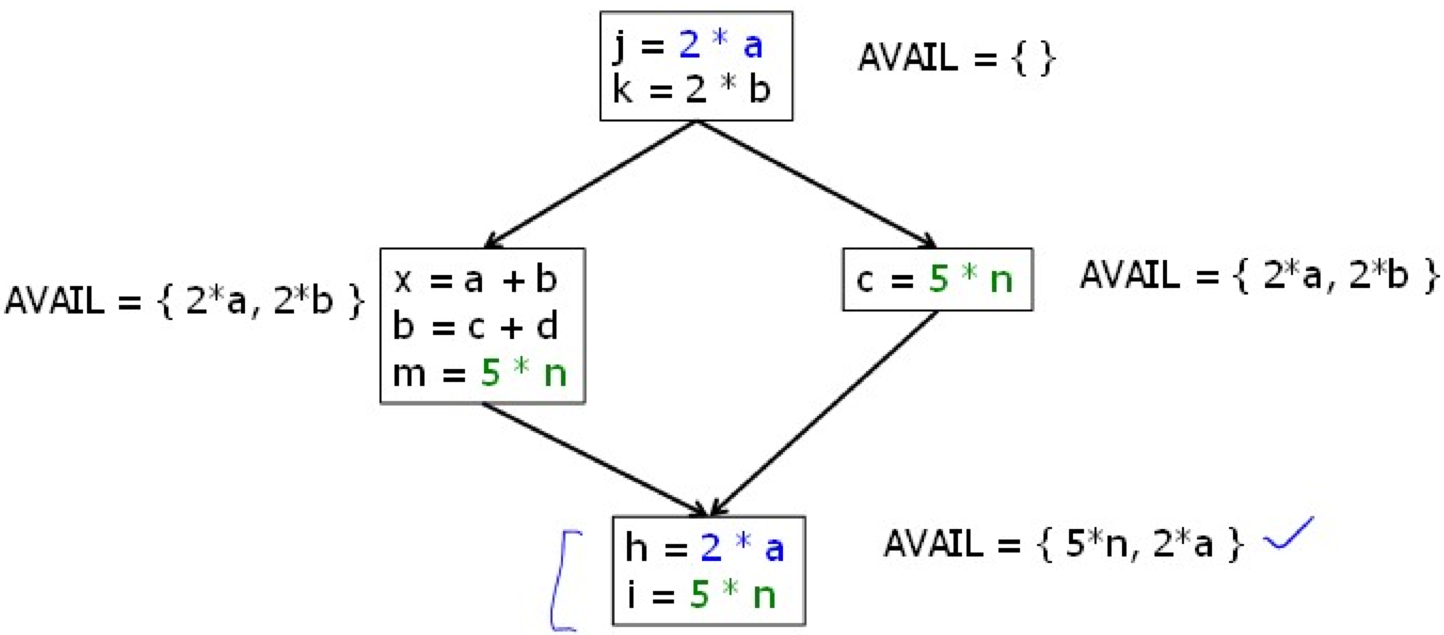
    n': v := w
  - Modify statement s to be

    s: t := w
  - (Rely on copy propagation to remove extra assignments that are not really needed)

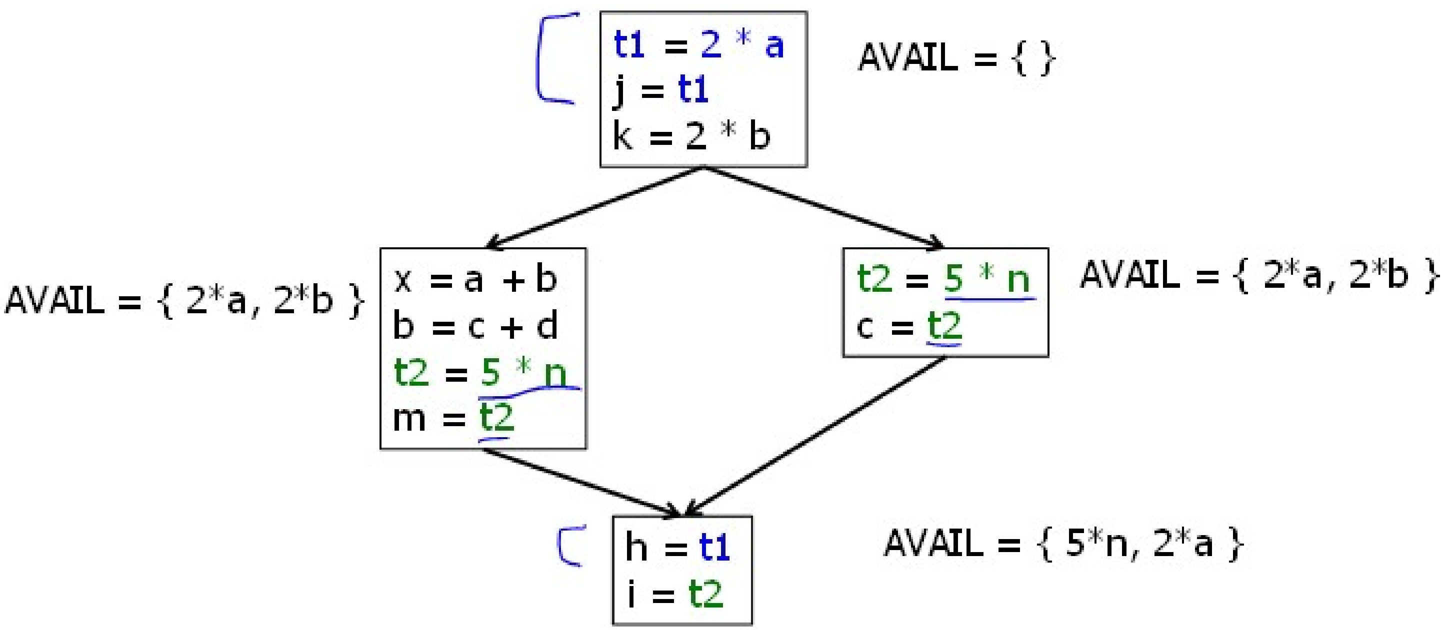
n: v= x op y

s: t= x op y

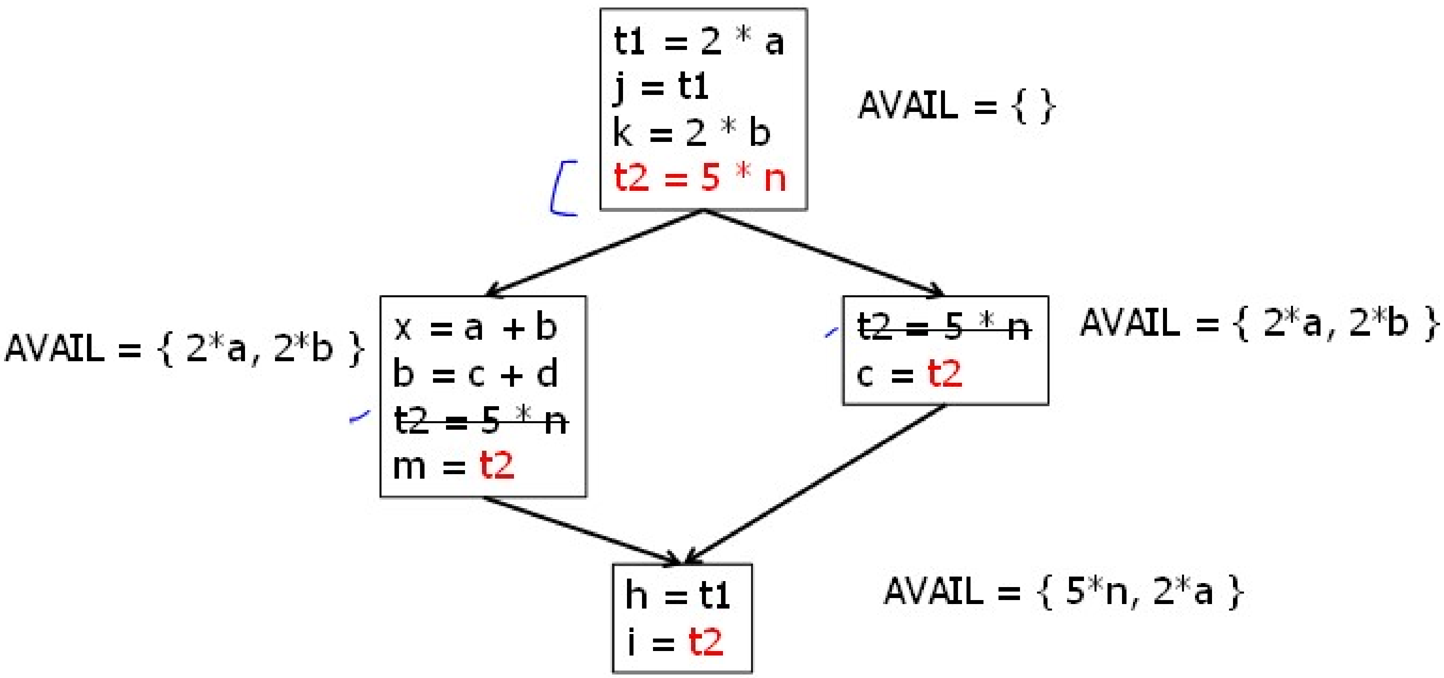
Revisiting Example (w/slight addition)



Revisiting Example (w/slight addition)



Then Apply Very Busy…



Constant Propagation

- Suppose we have
  - Statement d: t := c, where c is constant
  - Statement n that uses ~~t~~ c
- If d reaches n and no other definitions of t reach n, then rewrite n to use c instead of t

Copy Propagation

- Similar to constant propagation
- Setup:
  - Statement d: t := z
  - Statement n uses ~~t~~ z
- If d reaches n and no other definition of t reaches n, and there is no definition of z on any path from d to n, then rewrite n to use z instead of t
  - Recall that this can help remove dead assignments

Copy Propagation Tradeoffs

- Downside is that this can increase the lifetime of variable z and increase need for registers or memory traffic
- But it can expose other optimizations, e.g.,

  ✓ a := y + z

  ✓ ~~u := y~~

  ✓ c := ~~u~~ y + z   // copy propagation makes this y + z

  – After copy propagation we can recognize the common subexpression

Dead Code Elimination

- If we have an instruction

  s: a := b op c ?

  and a is not live-out after s, then s can be eliminated
  - Provided it has no implicit side effects that are visible (output, exceptions, etc.)
    - If b or c are function calls, they have to be assumed to have unknown side effects unless the compiler can prove otherwise

Aliases

- A variable or memory location may have multiple names or *aliases*
  - Call-by-reference parameters
  - Variables whose address is taken (&x)
  - Expressions that dereference pointers (p.x, *p)
  - Expressions involving subscripts (a[i])
  - Variables in nested scopes

Aliases vs Optimizations

- Example:
  - p.x := 5; q.x := 7; a := p.x;

  – Does reaching definition analysis show that the definition of p.x reaches a?

  – (Or: do p and q refer to the same variable/object?)

  – (Or: *can* p and q refer to the same thing?)

Aliases vs Optimizations

- Example

```
void f(int *p, int *q) {
  *p = 1; *q = 2;
  return *p;
}
```

  - How do we account for the possibility that p and q might refer to the same thing?
  - Safe approximation: since it's possible, assume it is true (but rules out a lot)
    - C programmers can use "restrict" to indicate no other pointer is an alias for this one

Types and Aliases (1)

- In Java, ML, MiniJava, and others, if two variables have incompatible types they cannot be names for the same location
  - Also helps that programmer cannot create arbitrary pointers to storage in these languages

Types and Aliases (2)

- Strategy: Divide memory locations into *alias classes* based on type information (every type, array, record field is a class)
- Implication: need to propagate type information from the semantics pass to optimizer
  - Not normally true of a minimally typed IR
- Items in different alias classes cannot refer to each other

Aliases and Flow Analysis

- Idea: Base alias classes on points where a value is created
  - Every new/malloc and each local or global variable whose address is taken is an alias class
  - Pointers can refer to values in multiple alias classes (so each memory reference is to a set of alias classes)
  - Use to calculate "may alias" information (e.g., p "may alias" q at program point s)

Using “may-alias” information

- Treat each alias class as a “variable” in dataflow analysis problems
- Example: framework for available expressions
  - Given statement   s: M[a]:=b,

    gen[s] = { }

    kill[s] = { M[x] | a may alias x at s }

May-Alias Analysis

- Without alias analysis, #2 kills M[t] since x and t might be related
- If analysis determines that “x may-alias t” is false, M[t] is still available at #3; can eliminate the common subexpression and use copy propagation

- Code

```
1:  u := M[t]
2:  M[x] := r
3:  w := M[t]
4:  b := u+w
```

Where are we now?

- Dataflow analysis is the core of classical optimizations
  - Although not the only possible story
- Still to explore:
  - Discovering and optimizing loops
  - SSA – Static Single Assignment form