CS 526

Advanced Compiler Construction

http://misailo.cs.Illinois.edu/courses/cs526

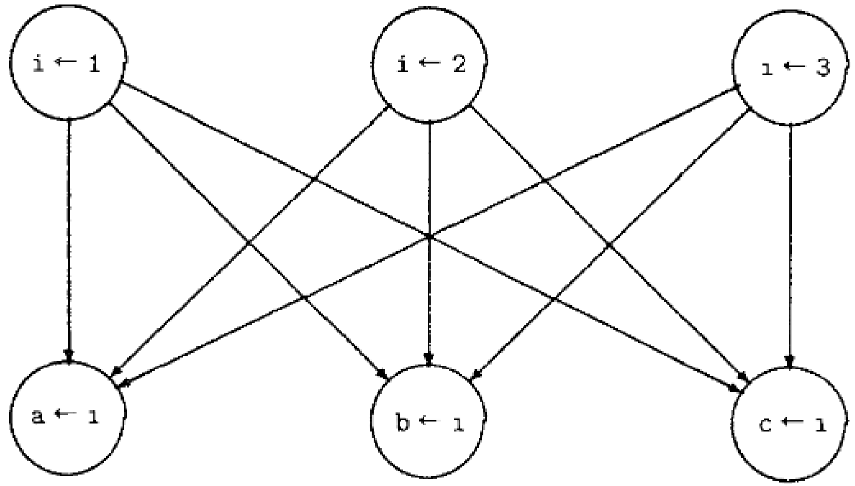
STATIC SINGLE ASSIGNMENT

The slides adapted from Vikram Adve

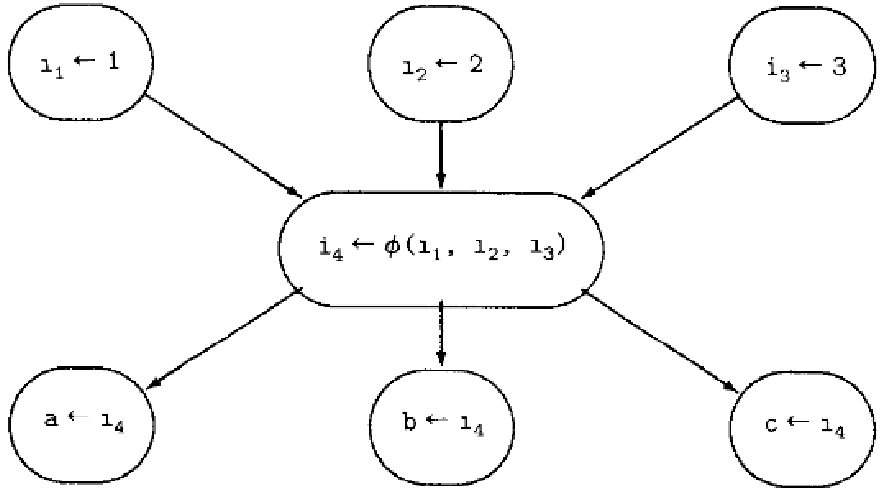
Def-use and Use-def (SSA vs no-SSA)

```
select j
  when x {i ← 1}
  when y {i ← 2}
  when z {i ← 3}
end
select k
  when x {a ← i}
  when y {b ← i}
  when z {c ← i}
end
```

Original Program



Def-Use Chains for Previous Program



SSA Graph for Previous Program

From Wegman et al.

SSA-Based Optimizations

- Dead Code Elimination (DCE)
- Sparse Conditional Constant Propagation (SCCP)
- Loop-Invariant Code Motion (LICM)
- Global Value Numbering (GVN)
- Strength Reduction of Induction Variables
- Live Range Identification in Register Allocation

Constant Propagation

Goals

Whenever there is a statement of the form v = Const, the uses of v can be replaced by Const.

Safety

Analysis: Explicit propagation of constant expressions

Transformation: Most languages allow removal of computations

Profitability

Fewer computations, almost always

Opportunity

Symbolic constants, conditionally compiled code, …

Simple Constant Propagation

```
Worklist = All statements in the SSA program

While Worklist ≠ ∅
      remove a statement S from Worklist

      if S is “v = φ(c1,…cn)” and c1=…=cn=c (Const),
            replace S with v = c

      if S is “v = c” (c is Const)
            Delete S from the program
            For each Statement T ∈ Uses (v)
                  substitute v with c in T
                  Worklist = Worklist ∪ {T}
```

Extensions of the Algorithm

Copy propagation:

- Assignments `x = y` or `x = φ(y)` can be replaced by a simple use of y.

Constant folding:

- Assignments of the form x = a © b can be immediately evaluated if a and b are constants, and the statement replaced with x = c (c = a © b)

Constant conditions:

- If a condition if (x © y) always evaluate to true or false, then keep only one branch.

Conditional Constant Propagation: SCCP

Goals

Identify and replace SSA variables with constant values

Delete infeasible branches due to discovered constants

Safety

Analysis: Explicit propagation of constant expressions

Transformation: Most languages allow removal of computations

Profitability

Fewer computations, almost always (except pathological cases)

Opportunity

Symbolic constants, conditionally compiled code, …

Example I

```
J = 1;
...
if (J > 0)
    I = 1; // Always produces 1
else
    I = 2;
```

Example 2

```
I = 1;
...
while (...) {
    J = I;
    I = f(...);
    ...
    I = J; // Always produces 1
}
```

We need to proceed with the assumption that everything is constant until proved otherwise.

Example 3

```
I = 1;
...
while (...) {
    J = I;
    I = f(...);
    ...
    if (J > 0)
        I = J; // Always produces 1
}
```

For Ex. 1, we could do constant propagation and condition evaluation separately, and repeat until no changes. This separate approach is not sufficient for Ex. 3.

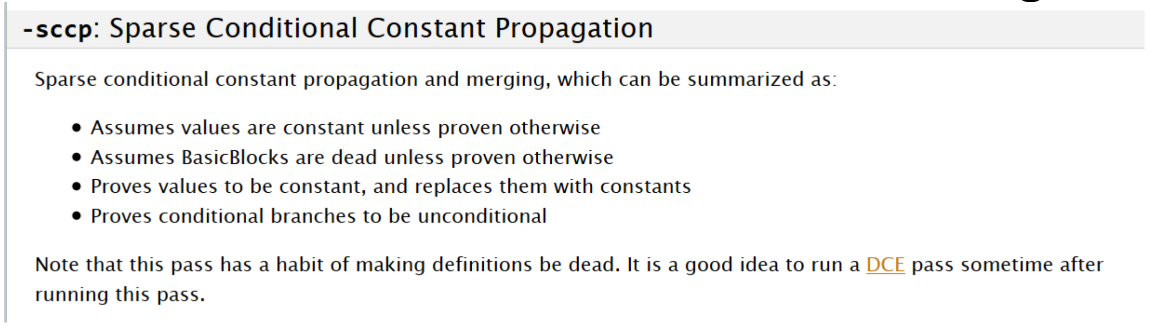
Conditional Constant Propagation*

Advantage:
Simultaneously finds constants + eliminates infeasible branches.

Optimistic
Assume every variable may be constant, until proven otherwise.
(Pessimistic ≡ initially assume nothing is constant.)

Sparse: Only propagates variable values where they are actually used or defined (using def-use chains in SSA form).

Iterative:
Build the list of constant definitions and uses using a worklist algorithm.



-sccp: Sparse Conditional Constant Propagation

Sparse conditional constant propagation and merging, which can be summarized as:

- Assumes values are constant unless proven otherwise
- Assumes BasicBlocks are dead unless proven otherwise
- Proves values to be constant, and replaces them with constants
- Proves conditional branches to be unconditional

Note that this pass has a habit of making definitions be dead. It is a good idea to run a DCE pass sometime after running this pass.

* Constant Propagation with Conditional Branches; M. Wegman and K. Zadeck, TOPLAS'91

Dead Code Elimination

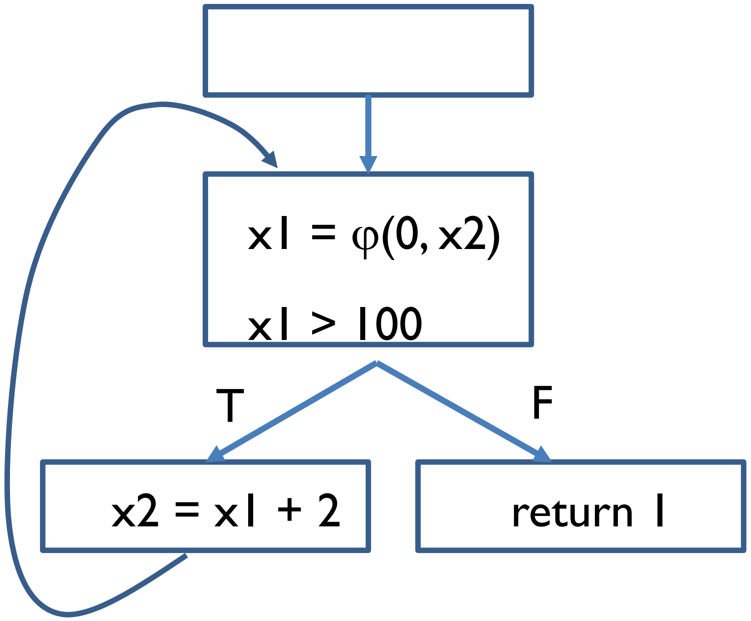
The results of the computation are visible through return values or output statements

- We can remove the instructions that do not contribute to the visible outputs

A simple algorithm:

- Compute (or maintain) the def-use chains
- Iterate over the instructions:
    - For v = x op z, if v has at least one use, mark as live, otherwise mark as dead
- Remove the instructions marked as dead

Aggressive Dead Code Elimination

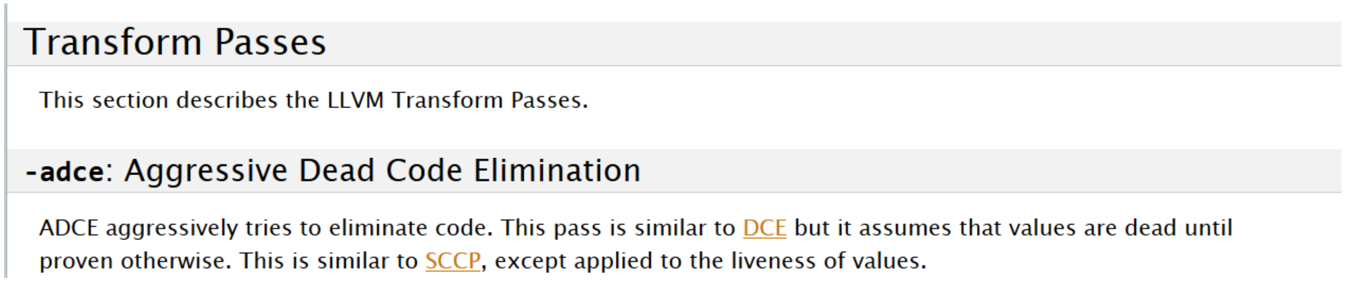


Ordinary DCE:

- x2 is live because used in def. of x1.
- x1 is live because used in def. of x2.

Yet…

Idea: Analogous to SCCP, be optimistic and assume a statement is dead unless proven otherwise (i.e., contributes to the output)



Transform Passes

This section describes the LLVM Transform Passes.

-adce: Aggressive Dead Code Elimination

ADCE aggressively tries to eliminate code. This pass is similar to DCE but it assumes that values are dead until proven otherwise. This is similar to SCCP, except applied to the liveness of values.

Aggressive Dead Code Elimination

In each step, mark a statement as live if:

1. It is the **output** statement (e.g., return)
2. It has **known side effects** (e.g., assignment to global variable or calling a function with side effects)
3. It **defines a variable x** <u>used by an already live statement</u>
4. It is a **conditional branch**, and some other, already live statement is <u>control dependent</u> on the branch (and its block)

The algorithm then converges to a set of live variables

- Caveat: the algorithm may remove "empty" infinite loops

Loop-Invariant Code Motion

Example:

```
x = 1; y = 0

while ( y < 10 ) {
  t = min(x,2)
  y = y + x
}
```

Becomes:

```
x = 1; y = 0

t = min(x,2)
while ( y < 10 ) {
   y = y + x
}
```

Pattern:

```
loop {

  v = a op b

  … v is used here
}
```

Becomes:

```
v = a op b
loop {
  … v is used here
}
```

What conditions does the code need to satisfy for this transformation to be sound?

Loop-Invariant Code Motion

Analysis

Conditions for the analysis (`v = a op b`):

- Both `a, b` are constants

```
    while (b) { v = 2 + 3;  /* … */ }
```

- Both `a, b` are defined before the loop (SSA ensures there is only a single dominating definition for each)

```
    x = …;
    while (b) { v = x + 1;  /* … */ }
```

- Both `a, b` are referring to the variables that are in the loop but already determined to be loop invariant

```
    x = …;
    while (b) { v = x + 1; t = v * 2; /* … */ }
```

If curious about what complications arise if the program is not in the SSA form, take a look at http://www.cs.cmu.edu/~aplatzer/course/Compilers11/17-loopinv.pdf

Loop-Invariant Code Motion

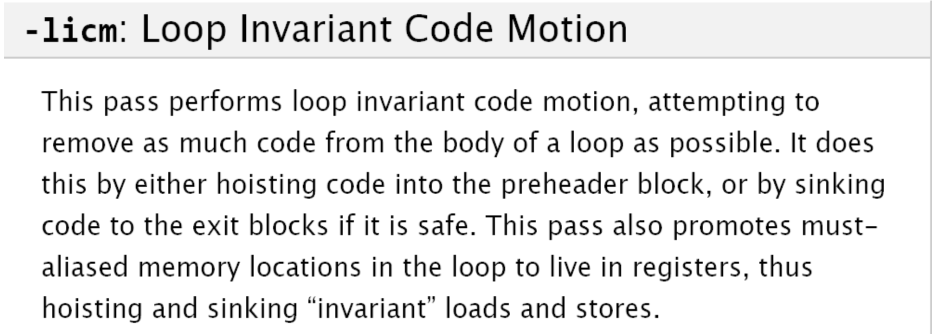
Transformation

Version 1: Since the computation is in SSA form, just move it to the node before the header

- What if there is no single such node? (Make it!)
- ***Loop preheader***: a single node that dominates the loop header
- How do we ensure there are no side effects?
- What if the loop-invariant computation is expensive?

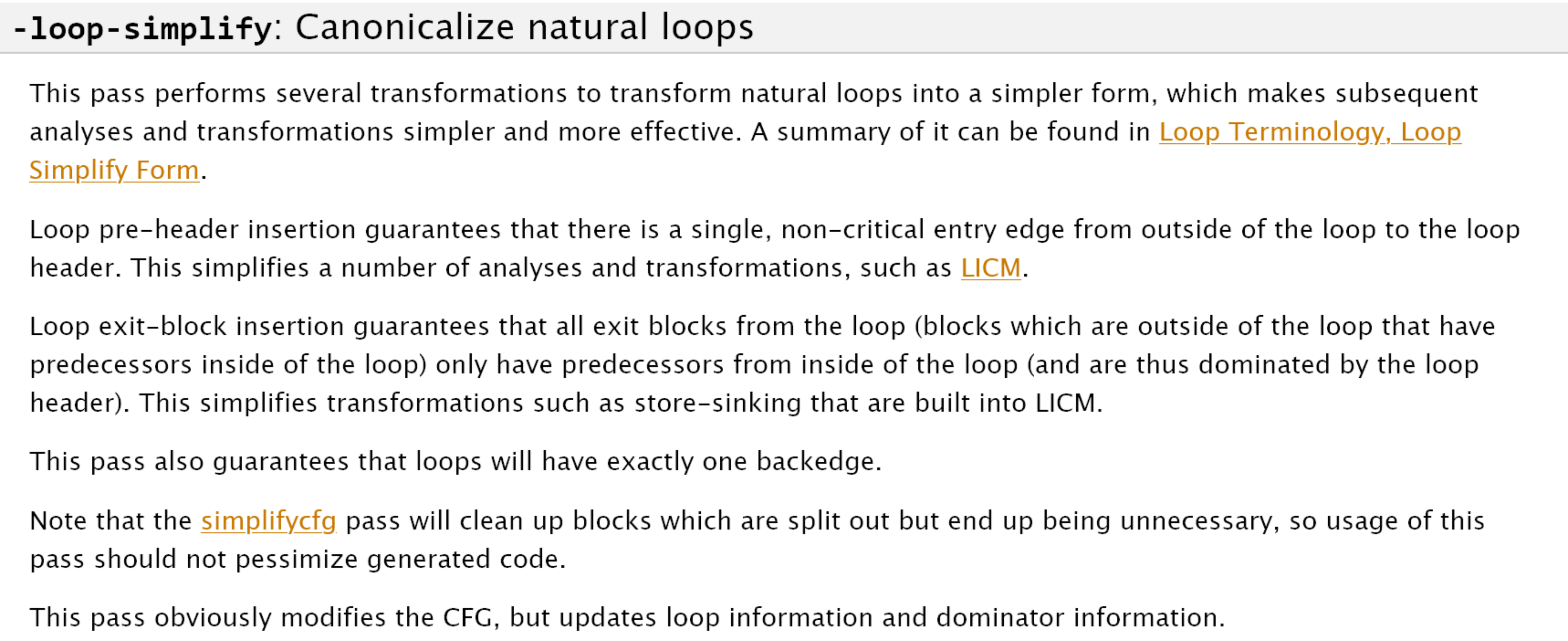
Version 2: Like candidate 1, but add loop's condition to avoid unnecessary execution of a op b:

```
if (cond) {
    t = a op b;
    while (cond) { /* … */ }
}
```


-licm: Loop Invariant Code Motion

This pass performs loop invariant code motion, attempting to remove as much code from the body of a loop as possible. It does this by either hoisting code into the preheader block, or by sinking code to the exit blocks if it is safe. This pass also promotes must-aliased memory locations in the loop to live in registers, thus hoisting and sinking “invariant” loads and stores.

Another Handy One…



`-loop-simplify`: Canonicalize natural loops

This pass performs several transformations to transform natural loops into a simpler form, which makes subsequent analyses and transformations simpler and more effective. A summary of it can be found in Loop Terminology, Loop Simplify Form.

Loop pre-header insertion guarantees that there is a single, non-critical entry edge from outside of the loop to the loop header. This simplifies a number of analyses and transformations, such as LICM.

Loop exit-block insertion guarantees that all exit blocks from the loop (blocks which are outside of the loop that have predecessors inside of the loop) only have predecessors from inside of the loop (and are thus dominated by the loop header). This simplifies transformations such as store-sinking that are built into LICM.

This pass also guarantees that loops will have exactly one backedge.

Note that the simplifycfg pass will clean up blocks which are split out but end up being unnecessary, so usage of this pass should not pessimize generated code.

This pass obviously modifies the CFG, but updates loop information and dominator information.

See also: https://llvm.org/docs/LoopTerminology.html

Induction Variable Substitution

Auxiliary Induction Variable

An auxiliary induction variable in a loop

```
      for (int i = 0; i < n; i++) { … }
```

is any variable `j` that can be expressed as

```
      c × i + m
```

at every point where it is used in the loop, where `c` and `m` are loop-invariant values, but `m` may be different at each use.

Optimization Goals

Identify linear expression for each auxiliary induction variable

- More effective dependence analysis, and loop transformations
- Substitute linear expression in place of every use
- Eliminate expensive or loop-invariant operations from loop

Induction Variable Substitution

Auxiliary Induction Variable

```
for (int i = 0; i < n; i++) {
    j = 2*i + 1;
    k = -i;
    l = 2*i*i + 1;
    c = c + 5;
}
```

Induction Variable Substitution

Auxiliary Induction Variable

```
for (int i = 0; i < n; i++) {
   j = 2*i + 1;        // Y
   k = -i;             // Y
   l = 2*i*i + 1;      // N
   c = c + 5;          // Y*
}
```

Reminder: Strength Reduction

Goal: Replace expensive operations by cheaper ones

Primitive Operations: Many Examples

n * 2 → n << 1 (similarly, n/2)

n ** 2 → n * n

Recurrences

Example: `x = a[i]` to `x = (base(a) + (i-1) * 4)`

Such recurrences are common in array address calculations

Induction Variable Substitution

Strategy

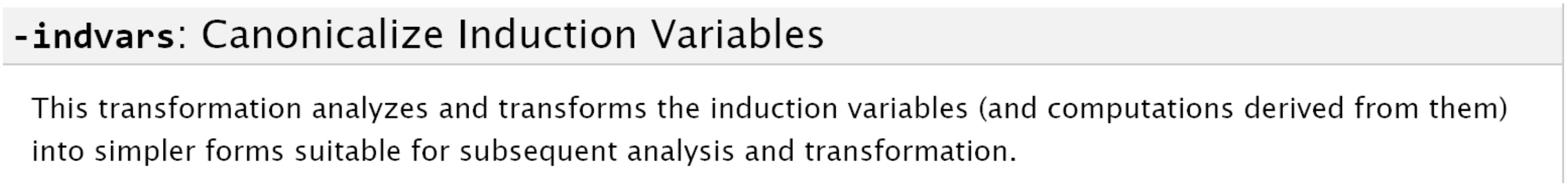
- Identify operations of the form:

  $x \leftarrow iv \times c$ or $x \leftarrow iv \pm c$

  iv: induction variable or another recurrence

  c : loop-invariant variable

- Eliminate **multiplications** from the loop body
- Eliminate induction variable if the **only remaining use** is in the loop **termination test**



-indvars: Canonicalize Induction Variables

This transformation analyzes and transforms the induction variables (and computations derived from them) into simpler forms suitable for subsequent analysis and transformation.

Induction Variable Substitution

```
do i = 1 to 100
   sum = sum + a(i)
enddo
```

Source code

```
    sum = 0.0
    i = 1
L:  t1 = i - 1
    t2 = t1 * 4
    t3 = t2 + a
    t4 = load t3
    sum = sum + t4
    i = i + 1
    if (i <= 100) goto L
```

Intermediate code

$$
\begin{array}{ll}
 & \texttt{sum}_0 = 0.0 \\
 & \texttt{i}_0 = 1 \\
\texttt{L:} & \texttt{sum}_1 = \phi(\texttt{sum}_0, \texttt{sum}_2) \\
 & \texttt{i}_1 = \phi(\texttt{i}_0, \texttt{i}_2) \\
 & \texttt{t1}_0 = \texttt{i}_1 - 1 \\
 & \texttt{t2}_0 = \texttt{t1}_0 * 4 \\
 & \texttt{t3}_0 = \texttt{t2}_0 + \texttt{a} \\
 & \texttt{t4}_0 = \texttt{load t3}_0 \\
 & \texttt{sum}_2 = \texttt{sum}_1 + \texttt{t4}_0 \\
 & \texttt{i}_2 = \texttt{i}_1 + 1 \\
 & \texttt{if (i}_2 \texttt{ <= 100) goto L}
\end{array}
$$

SSA form

Induction Variable Substitution

```
      sum_0 = 0.0
      i_0 = 1
L:    sum_1 = φ(sum_0,sum_2)
      i_1 = φ(i_0,i_2)
      t1_0 = i_1 - 1
      t2_0 = t1_0 * 4
      t3_0 = t2_0 + a
      t4_0 = load t3_0
      sum_2 = sum_1 + t4_0
      i_2 = i_1 + 1
      if (i_2 <= 100) goto L
```

SSA form

```
      sum_0 = 0.0
      i_0 = 1
      t5_0 = a
L:    sum_1 = φ(sum_0,sum_2)
      i_1 = φ(i_0,i_2)
      t5_1 = φ(t5_0,t5_2)
      t4_0 = load t5_0
      sum_2 = sum_1 + t4_0
      i_2 = i_1 + 1
      t5_2 = t5_1 + 4
      if (i_2 <= 100) goto L
```

After strength reduction

Induction Variable Substitution

```
      sum_0 = 0.0
      i_0 = 1
      t5_0 = a
L:    sum_1 = φ(sum_0,sum_2)
      i_1 = φ(i_0,i_2)
      t5_1 = φ(t5_0,t5_2)
      t4_0 = load t5_0
      sum_2 = sum_1 + t4_0
      i_2 = i_1 + 1
      t5_2 = t5_1 + 4
      if (i_2 <= 100) goto L
```

After strength reduction

```
      sum_0 = 0.0
      t5_0 = a
L:    sum_1 = φ(sum_0,sum_2)
      t5_1 = φ(t5_0,t5_2)
      t4_0 = load t5_0
      sum_2 = sum_1 + t4_0
      t5_2 = t5_1 + 4
      if (t5_2 <= 396 + a) goto L
```

After induction variable substitution

Induction Variable Substitution (recap)

```
      sum_0 = 0.0
      t5_0 = a
L:    sum_1 = φ(sum_0,sum_2)
      t5_1 = φ(t5_0,t5_2)
      t4_0 = load t5_0
      sum_2 = sum_1 + t4_0
      t5_2 = t5_1 + 4
      if (t5_2 <= 396 + a) goto L
```

After induction variable substitution

```
      sum_0 = 0.0
      i_0 = 1
L:    sum_1 = φ(sum_0,sum_2)
      i_1 = φ(i_0,i_2)
      t1_0 = i_1 - 1
      t2_0 = t1_0 * 4
      t3_0 = t2_0 + a
      t4_0 = load t3_0
      sum_2 = sum_1 + t4_0
      i_2 = i_1 + 1
      if (i_2 <= 100) goto L
```

Before induction variable substitution

References

Cocke and Kennedy, CACM 1977 (superseded by the next one).

Allen, Cocke and Kennedy, “Reduction of Operator Strength,” In Program Flow Analysis: Theory and Applications, 1981.

Classical Approach

- ACK: Classic algorithm, widely used.
- works on “loops” (Strongly Connected Regions) of flow graph
- uses def-use chains to find induction variables and recurrences

Cooper, Simpson & Vick, 2001, “Operator Strength Reduction,” Trans. Prog. Lang. Sys. 23(5), Sept. 2001.

SSA-based algorithm

- Same effectiveness as ACK, but faster and simpler
- Identify induction variables from SCCs in the SSA graph

Value Numbering

Code:

```
a = x + y

b = x + y

a = 1
c = x + y
d = y + x
e = d - 1
f = e + 1
```

- **Analysis:** Determining equivalent computations (variables, expressions, consts)
- **Transformation:** Eliminates duplicates with a semantics-preserving optimization
- Form of redundancy elimination

Value Numbering

- Assign an **identifying number** to each variable / expression / constant:

  x and y have same **id number**

  ⇔ x = y **for all** inputs

- Use algebraic identities to simplify expressions
- Discover redundant computations & replace them
- Discover constant values, fold & propagate them

Value Numbering

- Use algebraic identities to simplify expressions
    - Commutativity (a+b = b+a), a+b+c = c+b+a, (a+b)^2 = a^2+2ab+b^2…
- Discover redundant computations and replace them
    - E.g., `y=2*x; z=2*x+1 => y=2*x; z=y+1`
- Discover constant values, fold & propagate them
    - After SCCP: e.g., `x=1; y = x+1 => y = 1+1`
    - Evaluate constant expression (`y = 2`) then propagate

Local Value Numbering

- Each variable, expression, & constant gets a **"value number"** (hash code)

  Same value number ⇒ same value

- **Prerequisites:** low-level intermediate code and existing basic blocks
- Equivalence based solely on facts from within the **single basic block**
- If an instruction's value number is already defined, instr. can be eliminated & subsequent references subsumed
- Constant folding is simple

Local Value Numbering

```
a = x + y
```

$V1 \leftarrow$ hash(+,VN[x],VN[y]),
Name[V1] $\leftarrow$ a

```
b = x + y
```

hash(+,VN[x],VN[y]) == V1
So, replace x+y with a. Transformed: b = a

```
a = 1
```

Name[V1] $\leftarrow \emptyset$ (can we be more precise?)

```
c = x + y
d = y + x
e = d - 1
f = e + 1
```

$\}$ Can we replace?

Challenges:
tracking where each value resides
commutativity $\Rightarrow$???
identities (e.g., Vx OR Vx × 1): $\Rightarrow$
instr. gets value number of operand (Vx)

Local Value Numbering

```
a1 = x + y
```

V1 ← hash(+,VN[x],VN[y]),
Name[V1] ← a

```
b = x + y
```

hash(+,VN[x],VN[y]) == V1
So, replace x+y with a. Transformed: b = a1

```
a2 = 1
c = x + y
d = y + x
e = d - 1
f = e + 1
```

Name[V1] ←∅ (don't need anymore)

b = a1

c = a1

d = a1

Challenges:
tracking where each value resides
commutativity ⇒ ???
identities (e.g., Vx OR Vx × 1): ⇒
instr. gets value number of operand (Vx)

Local Value Numbering

```
a1 = x + y
```

V1 ← hash(+,VN[x],VN[y]),
Name[V1] ← a

```
b1 = a1
```

hash(+,VN[x],VN[y]) == V1
So, replace x+y with a. Transformed: b = a1

```
a2 = 1
```

Name[V1] ←∅ (don't need anymore)

```
c1 = a1
d1 = a1
e = d1 - 1
f = e + 1
```

b = a1

c = a1

d = a1

Challenges:
What happens with e and f ?

Local Value Numbering

```
For each instruction **i : x ← y op z** in the block
        V1 ← VN[y]
        V2 ← VN[z]

        let v = hash(op,V1,V2)
        if (v exists in hash table)
                replace RHS with Name[v]
        else
                enter v in hash table
                VN[x] ← v
                Name[v] ← ti (new temporary)
                replace instruction with: “ti ← y op z; x ← ti”
```

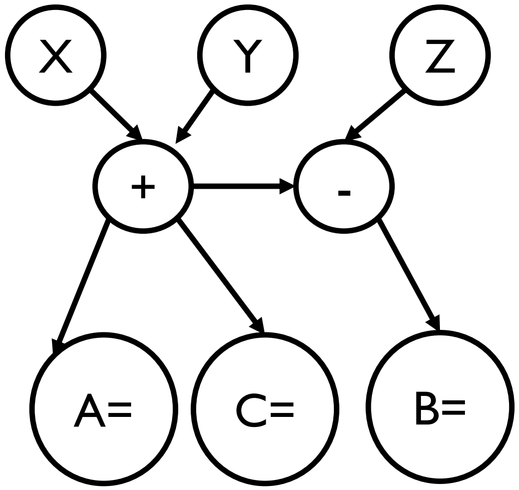
Local VN *Simplifications*

- If the operands have the same value number i.e. z=x op y, and VN[x] = VN[y]
    - if *op* is MAX, MIN, AND, OR, . . replace *op* with a copy operation (z=x)
    - if *op* tests equality, replace it with z=true
    - if *op* tests inequality replace it with z=false
- if all operands (x,y) are constants and we haven't already simplified the expression, then immediately evaluate the resulting constant and propagate constants down
- if one operand is constant and we haven't yet simplified the expression:
    - if a constant operand is zero, replace ADD and OR with another operand; replace MULT, AND with zero
    - if constant operand is one, replace MULT with assignment of another operand
- If *op* commutes, reorder its operands into **ascending order by value number** (canonical form)

Local VN *Analogy*

- Constructing a DAG from a forest (set of trees)
- Each expression is a node in a dag, edges are uses of the expression in the instructions
- Start from the leading instruction of the basic block
- Collapse nodes that are repeated into a single node and connect the edges to all uses

```
a = x + y
b = (x + y) - z
c = y + x
```



Global Value Numbering

```
W = X + Y;
if (...) {
  Z = X + Y;
  X = 1;
} else {
  Z = X + Y - 1;
}


U = X + Y - 1;   // ??
```

Global Value Numbering

```
W1 = X1 + Y1;
if (...) {
  Z1 = X1 + Y1;
  X2 = 1;
} else {
  Z2 = X1 + Y1 – 1;
}
X3 = Phi(X1, X2)
Z3 = Phi(Z1, Z2)
U1 = X3 + Y1 – 1;   // ??
```

Global Value Numbering

```
T1 = X1 + Y1; W1 = T1;
if (...) {
  Z1 = T1;
  X2 = 1;
} else {
  Z2 = T1 – 1;
}
X3 = Phi(X1, X2)
Z3 = Phi(Z1, Z2)
U1 = X3 + Y1 – 1;   // ??
```

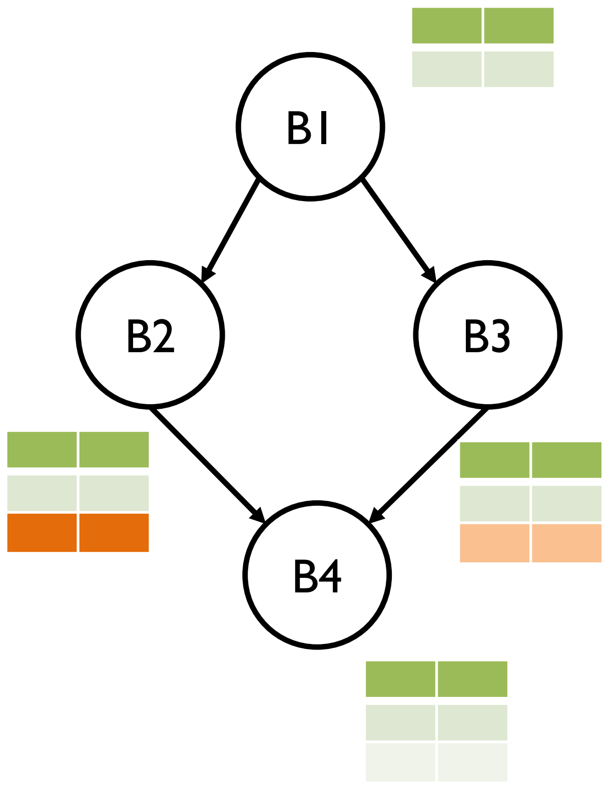
Yet another example

```
X0 = 1
Y0 = 1
do {
     X1 = ϕ(X0, X2)
     Y1 = ϕ(Y0, Y2)
     X2 = X1 + 1
     Y2 = Y1 + 1
} while (. . .)
```

Global Value Numbering (DVTN)

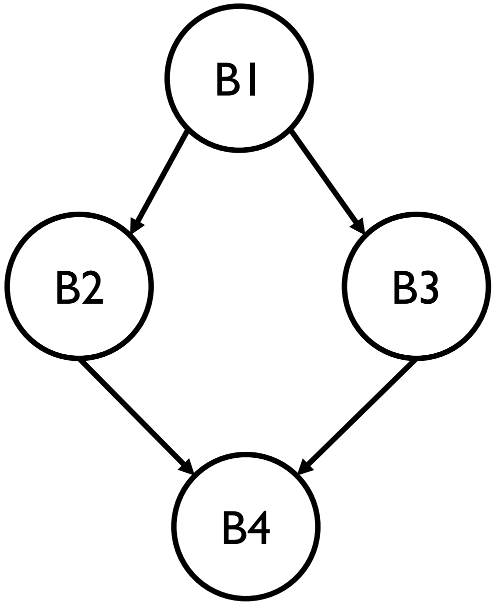
The Dominator-based VN Technique **(DVNT)**

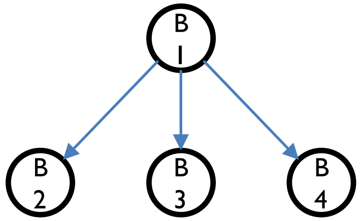
- B2, B3 can be value-numbered using B1's table
- How about B4? Yes, can use the expressions from B1 (dominator node) but needs to invalidate the expressions killed in B2, B3
- Still based on hashing
- **BUT:** difficult to merge these tables
    - A variable may be redefined in B2, B3, or both



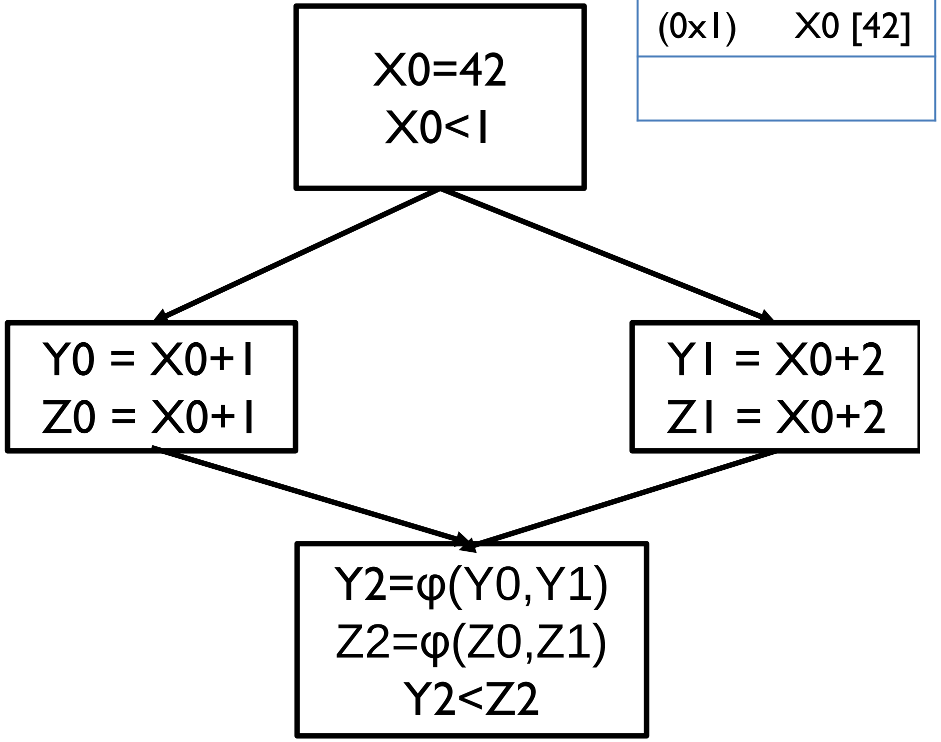
Global Value Numbering (DVTN)

```
procedure $DVNT$(Block $B$)
    Mark the beginning of a new scope
    for each $\phi$-function $p$ of the form "$n \leftarrow \phi(\ldots)$" in $B$
        if $p$ is meaningless or redundant
            Put the value number for $p$ into $VN[n]$
            Remove $p$
        else
            $VN[n] \leftarrow n$
            Add $p$ to the hash table
    for each assignment $a$ of the form "$x \leftarrow y$ op $z$" in $B$
        Overwrite $y$ with $VN[y]$ and $z$ with $VN[z]$
        $expr \leftarrow \langle y \text{ op } z \rangle$
        if $expr$ can be simplified to $expr'$
            Replace $a$ with "$x \leftarrow expr'$"
            $expr \leftarrow expr'$
        if $expr$ is found in the hash table with value number $v$
            $VN[x] \leftarrow v$
            Remove $a$
        else
            $VN[x] \leftarrow x$
            Add $expr$ to the hash table with value number $x$
    for each successor $s$ of $B$
        Adjust the $\phi$-function inputs in $s$
    for each child $c$ of $B$ in the dominator tree
        $DVNT(c)$
    Clean up the hash table after leaving this scope
```


$$\begin{tikzcd} & B1 \arrow[dl] \arrow[dr] & \\ B2 \arrow[dr] & & B3 \arrow[dl] \\ & B4 & \end{tikzcd}$$


$$\begin{tikzcd} & B1 \arrow[dl] \arrow[d] \arrow[dr] & \\ B2 & B3 & B4 \end{tikzcd}$$

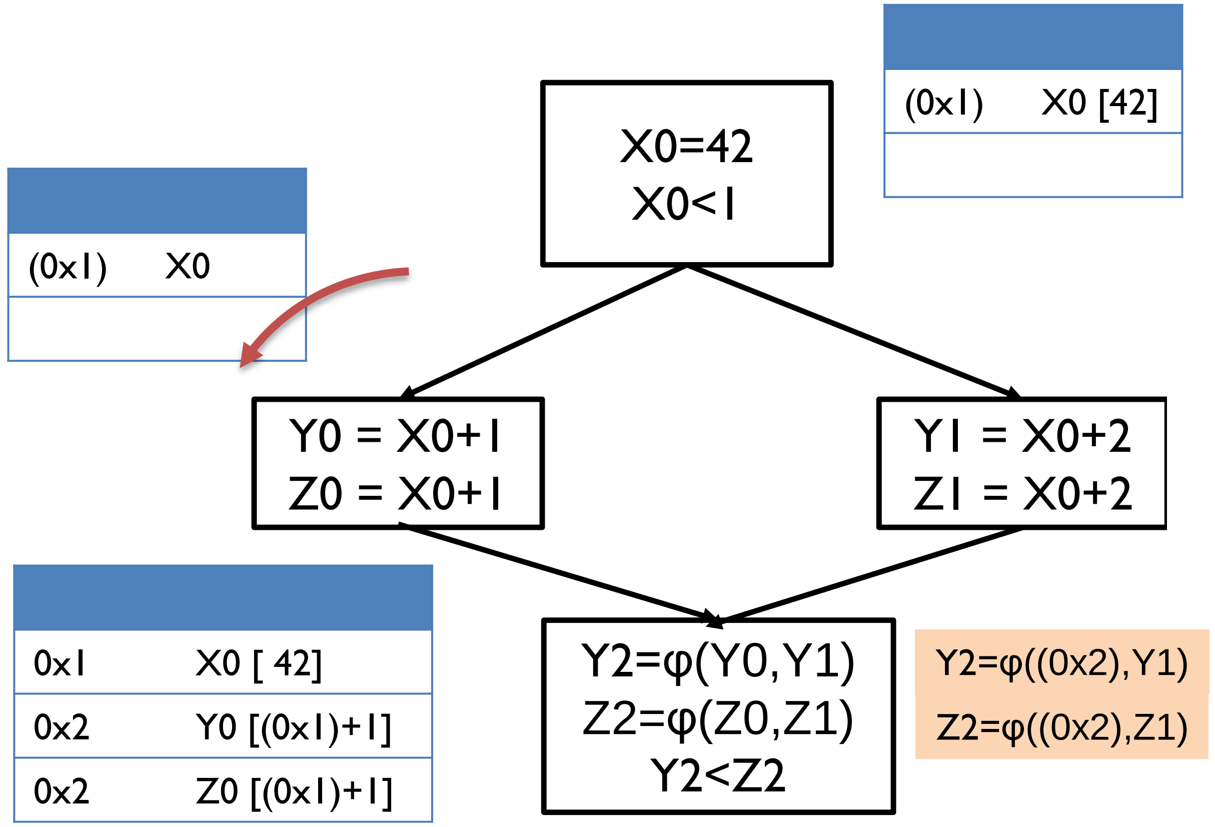
Example



(0x1)	X0 [42]

```
procedure DVNT(Block $B$)
    Mark the beginning of a new scope
    for each $\phi$-function $p$ of the form "$n \leftarrow \phi(\ldots)$" in $B$
        if $p$ is meaningless or redundant
            Put the value number for $p$ into $VN[n]$
            Remove $p$
        else
            $VN[n] \leftarrow n$
            Add $p$ to the hash table
    for each assignment $a$ of the form "$x \leftarrow y$ op $z$" in $B$
        Overwrite $y$ with $VN[y]$ and $z$ with $VN[z]$
        $expr \leftarrow \langle y$ op $z\rangle$
        if $expr$ can be simplified to $expr'$
            Replace $a$ with "$x \leftarrow expr'$"
            $expr \leftarrow expr'$
        if $expr$ is found in the hash table with value number $v$
            $VN[x] \leftarrow v$
            Remove $a$
        else
            $VN[x] \leftarrow x$
            Add $expr$ to the hash table with value number $x$
    for each successor $s$ of $B$
        Adjust the $\phi$-function inputs in $s$
    for each child $c$ of $B$ in the dominator tree
        DVNT($c$)
    Clean up the hash table after leaving this scope
```

Example



(0x1)	X0 [42]

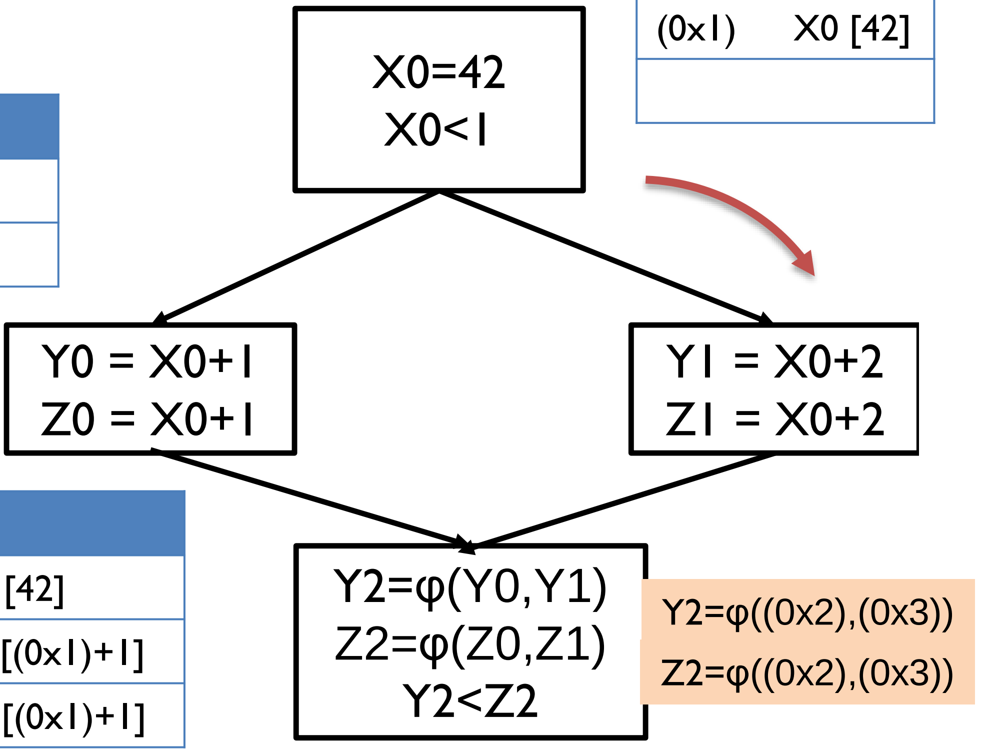
(0x1)	X0

0x1	X0 [42]
0x2	Y0 [(0x1)+1]
0x2	Z0 [(0x1)+1]

```
procedure DVNT(Block B)
    Mark the beginning of a new scope
    for each φ-function p of the form "n ← φ(...)" in B
        if p is meaningless or redundant
            Put the value number for p into VN[n]
            Remove p
        else
            VN[n] ← n
            Add p to the hash table
    for each assignment a of the form "x ← y op z" in B
        Overwrite y with VN[y] and z with VN[z]
        expr ← ⟨y op z⟩
        if expr can be simplified to expr′
            Replace a with "x ← expr′"
            expr ← expr′
        if expr is found in the hash table with value number v
            VN[x] ← v
            Remove a
        else
            VN[x] ← x
            Add expr to the hash table with value number x
    for each successor s of B
        Adjust the φ-function inputs in s
    for each child c of B in the dominator tree
        DVNT(c)
    Clean up the hash table after leaving this scope
```

Example

(0x1)	X0



(0x1)	X0 [42]

0x1	X0 [42]
0x2	Y0 [(0x1)+1]
0x2	Z0 [(0x1)+1]

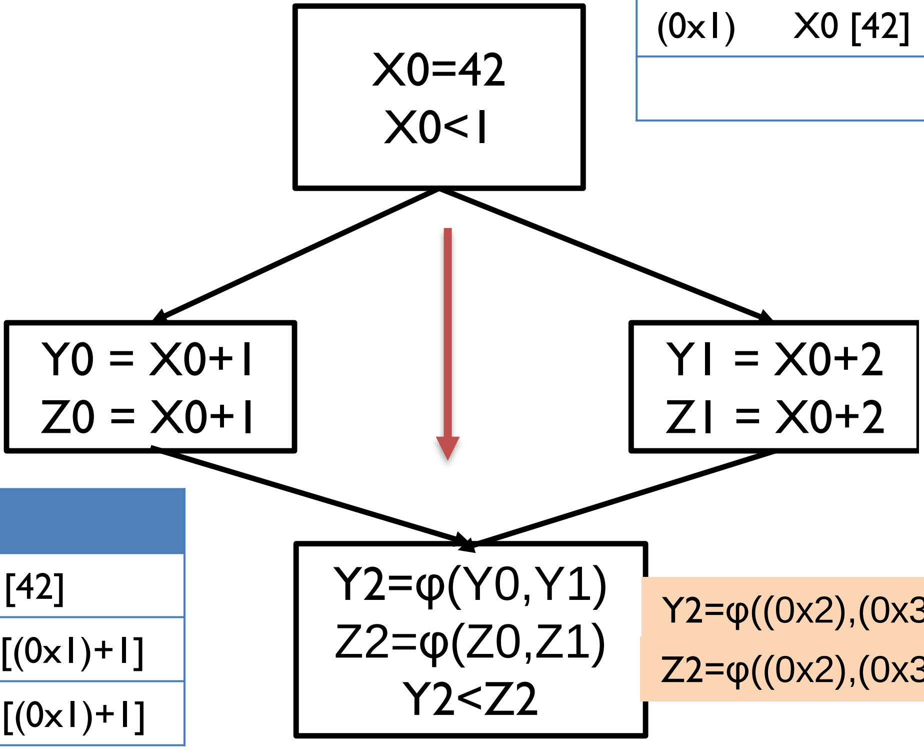
0x1	X0 [42]
0x3	Y0 [(0x1)+1]
0x3	Z0 [(0x1)+1]

```
procedure DVNT(Block B)
    Mark the beginning of a new scope
    for each φ-function p of the form "$n \leftarrow \phi(\ldots)$" in B
        if p is meaningless or redundant
            Put the value number for p into $VN[n]$
            Remove p
        else
            $VN[n] \leftarrow n$
            Add p to the hash table
    for each assignment a of the form "$x \leftarrow y$ op $z$" in B
        Overwrite y with $VN[y]$ and z with $VN[z]$
        expr ← ⟨y op z⟩
        if expr can be simplified to expr'
            Replace a with "x ← expr'"
            expr ← expr'
        if expr is found in the hash table with value number v
            $VN[x] \leftarrow v$
            Remove a
        else
            $VN[x] \leftarrow x$
            Add expr to the hash table with value number x
    for each successor s of B
        Adjust the φ-function inputs in s
    for each child c of B in the dominator tree
        DVNT(c)
    Clean up the hash table after leaving this scope
```

Example

```
procedure $DVNT$(Block $B$)
    Mark the beginning of a new scope
    for each $\phi$-function $p$ of the form "$n \leftarrow \phi(\ldots)$" in $B$
        if $p$ is meaningless or redundant
            Put the value number for $p$ into $VN[n]$
            Remove $p$
        else
            $VN[n] \leftarrow n$
            Add $p$ to the hash table
    for each assignment $a$ of the form "$x \leftarrow y$ op $z$" in $B$
        Overwrite $y$ with $VN[y]$ and $z$ with $VN[z]$
        $expr \leftarrow \langle y$ op $z\rangle$
        if $expr$ can be simplified to $expr'$
            Replace $a$ with "$x \leftarrow expr'$"
            $expr \leftarrow expr'$
        if $expr$ is found in the hash table with value number $v$
            $VN[x] \leftarrow v$
            Remove $a$
        else
            $VN[x] \leftarrow x$
            Add $expr$ to the hash table with value number $x$
    for each successor $s$ of $B$
        Adjust the $\phi$-function inputs in $s$
    for each child $c$ of $B$ in the dominator tree
        $DVNT(c)$
    Clean up the hash table after leaving this scope
```

(0x1)	X0 [42]



Y2=φ((0x2),(0x3))
Z2=φ((0x2),(0x3))

0x1	X0 [42]
0x2	Y0 [(0x1)+1]
0x2	Z0 [(0x1)+1]

0x1	X0 [42]
0x3	Y0 [(0x1)+1]
0x3	Z0 [(0x1)+1]

(0x1)	X0 [42]
(0x4)	Y2 [φ((0x2),(0x3))]
(0x4)	Z2 [φ((0x2),(0x3))]

Instruction Congruence

Instructions **i** and **j** are **congruent** iff:

1. They are the same instruction, or
2. They are assignments of constants, which are equal (e.g. $x:=c_i$, $y:=c_j$ and $c_i==c_j$), or
3. They have one or multiple operands, e.g.,

   $z_i = x_i \text{ op } y_i$

   $z_j = x_j \text{ op } y_j$

   same operator and their operands are **congruent** (x_i congruent to x_j and y_i congruent to y_j), taking into consideration commutativity of op.

References

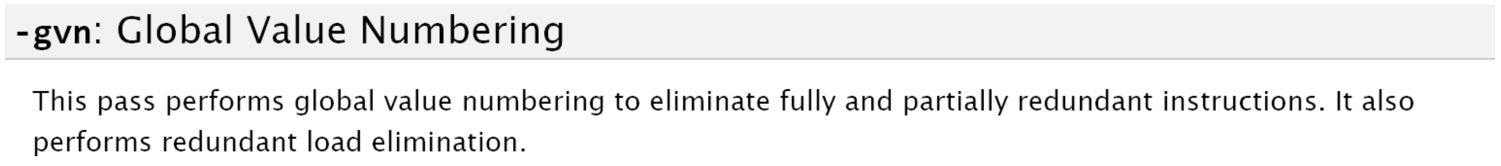
Long history in literature

- form of redundancy elimination (compare CSE)
- local version using hashing: late 60's Cocke & Schwartz, 1969
- algorithms for blocks, extended blocks, dominator regions, entire procedures, and (maybe) whole programs
- easy to understand algorithm for single block
- larger scopes cause more complex algorithms

1. Alpern, Wegman & Zadeck, "Detecting Equality of Variables in Programs," *Proceedings POPL* 1988

2. Cooper & Simpson, "SCC-Based Value Numbering," *Rice University TR CRPC-TR95636-S*, 1995.

3. Briggs, Cooper, Simpson, "Value Numbering," *Software–Practice and Experience*, June 1997 (for explanations).



-gvn: Global Value Numbering

This pass performs global value numbering to eliminate fully and partially redundant instructions. It also performs redundant load elimination.

Optimizations where we will need more information

- Global Common Subexpression Elimination (GCSE)
- Partial Redundancy Elimination (PRE)
- Redundant Load Elimination
- Dead or Redundant Store Elimination
- Code Placement Optimizations