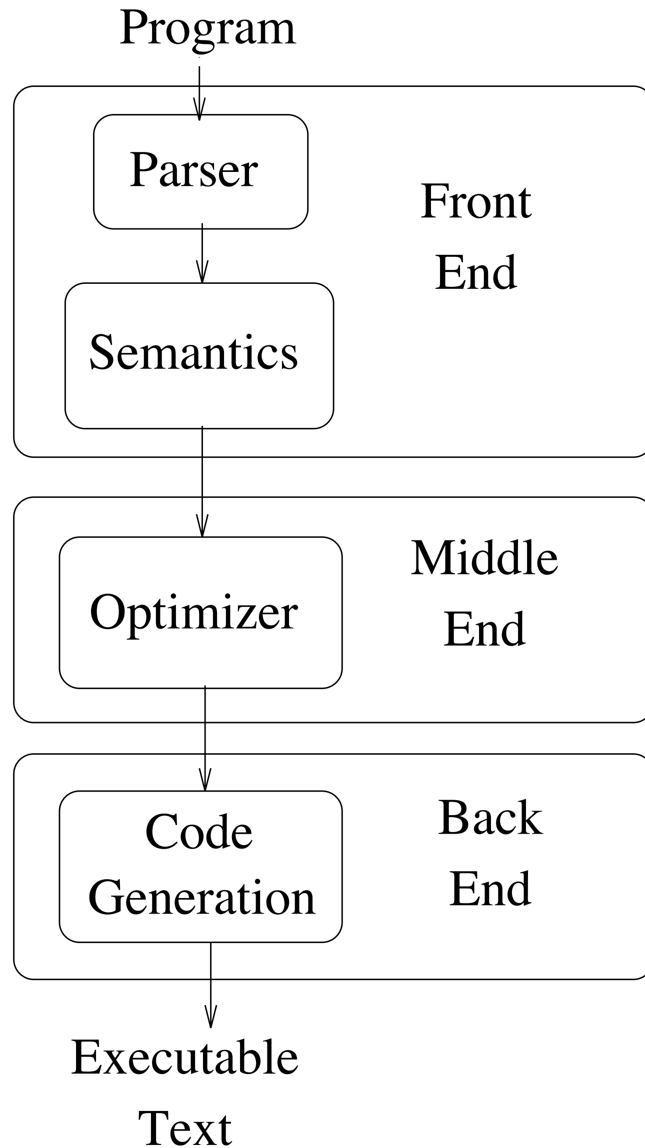
Compiler organizations



Front end: Operator and storage abstractions, alias mechanisms.

Middle end:

- Dead code elimination
- Code motion
- Reduction in strength
- Constant propagation
- Common subexpression elimination
- Fission
- Fusion
- Strip mining
- Jamming
- Splitting
- Collapsing

Back end: Finite resource issues and code generation.

Copyright ©1994 Ron K. Cytron. All rights reserved

Some thoughts

Misconceptions

Optimization optimizes your program. There's probably a better algorithm or sequence of program transformations. While optimization hopefully improves your program, the result is usually not optimal.

Optimization requires (much) more compilation time. For example, dead code elimination can reduce the size of program text such that overall compile time is also reduced.

A clever programmer is a good substitute for an optimizing compiler. While efficient coding of an algorithm is essential, programs should not be obfuscated by "tricks" that are architecture- (and sometimes compiler-) specific.

All too often. . .

Optimization is disabled by default. Debugging optimized code can be treacherous (45, 23). Optimization is often the primary suspect of program misbehavior—sometimes deservedly so. "No, not the *third* switch!"

Optimization is slow. Transformations are often applied to too much of a program. Optimizations are often textbook recipes, applied without proper thought.

Optimization produces incorrect code. Although recent work is encouraging (42), optimizations are usually developed *ad hoc.*

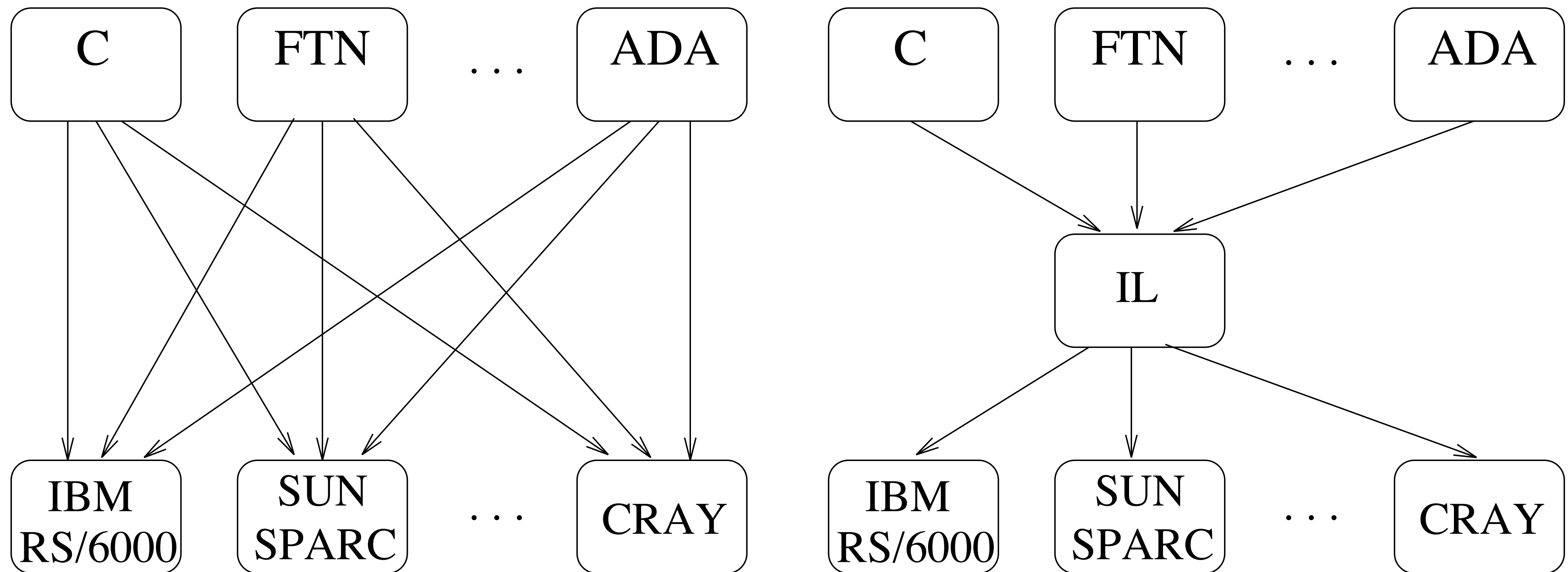
Programmers are trained by their compilers. A style is inevitably developed that is conducive to optimization.

Optimization is like *sex:*

- Everybody claims to get good results using *exotic* techniques;
- Nobody is willing to provide the details.

Copyright ©1994 Ron K. Cytron. All rights reserved

Multilingual systems



Architecting an *intermediate language* reduces the incremental cost of accommodating new source languages or target architectures (5). Moreover, many optimizations can be performed directly on the intermediate language text, so that source- and machine-independent optimizations can be performed by a common middle-end.

Copyright ©1994 Ron K. Cytron. All rights reserved

Intermediate languages

It's very easy to devote much time and effort toward choosing the "right" IL. Below are some guidelines for choosing or developing a useful intermediate language:

- **The IL should be a** *bona fide* **language, and not just an aggregation of data structures.**
- **The semantics of the IL should be cleanly defined and readily apparent.**
- **The IL's representation should not be overly verbose:**
  - **Although some expansion is inevitable, the IL-to-source token ratio should be as low as possible.**
  - **It's desirable for the IL to have a verbose, human-readable form.**
- **The IL should be easily and cleanly extensible.**
- **The IL should be sufficiently general to represent the important aspects of multiple front-end languages.**
- **The IL should be sufficiently general to support efficient code generation for multiple back-end targets.**

A sampling of difficult issues:

- **How should a string operation be represented (intact or as a "loop")?**
- **How much detail of a procedure's behavior is relevant?**

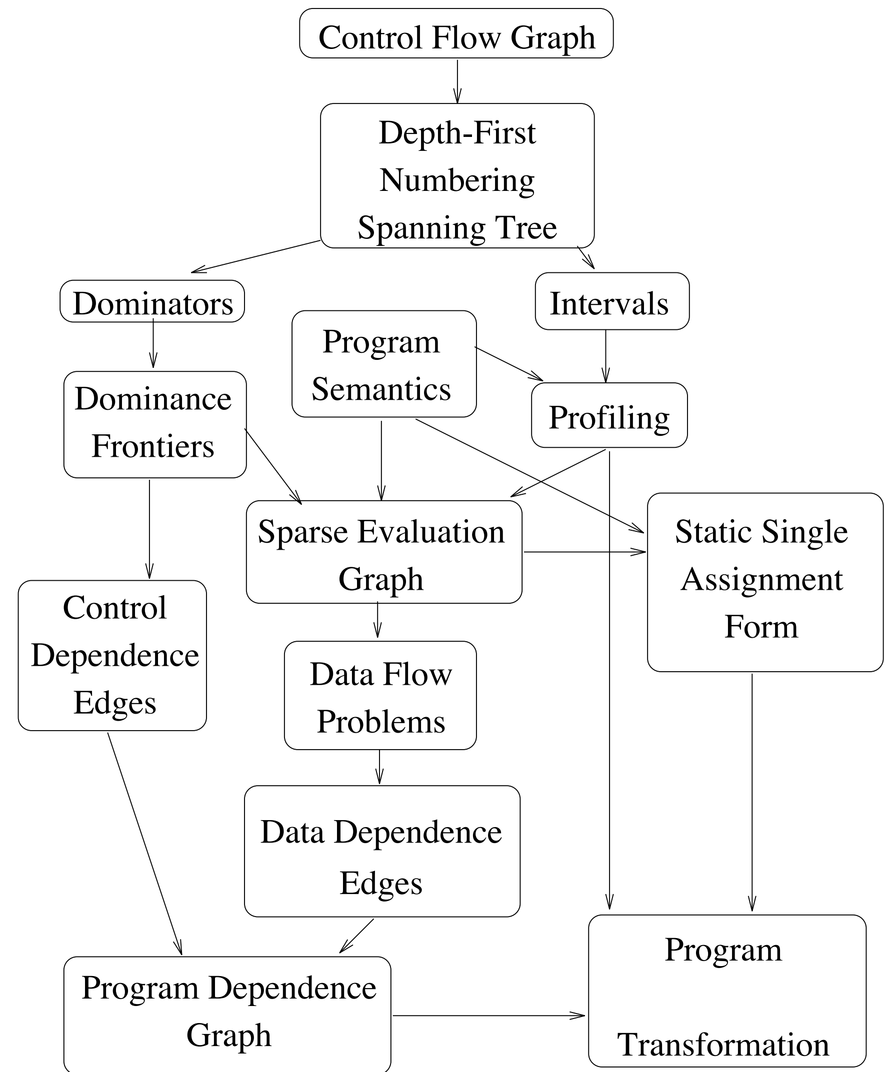
Ideally, an IL has *fractal* **characteristics: optimization can proceed at a given level; the IL can be "lowered"; optimization is then applied to the freshly exposed description.**

Copyright ©1994 Ron K. Cytron. All rights reserved

What happens in the middle end?

Essentially, the program is transformed into an observably equivalent while less resource-consumptive program. Such transformation is often based on:

- **Assertions provided by the program author or benefactor.**
- **The program dependence graph (29, 15, 6).**
- **Static single assignment (SSA) form (8, 3, 44, 9).**
- **Static information gathered by solving data flow problems (25, 34, 35, 36, 22, 37, 38, 27).**
- **Run-time information collected by** *profiling* **(40).**



Let's take a look at an example that benefits greatly from optimization. . .

Copyright ©1994 Ron K. Cytron. All rights reserved

Unoptimized matrix multiply

for $i = 1$ **to** N **do**
    for $j = 1$ **to** N **do**
        $A[i,j] \leftarrow 0$
        for $k = 1$ **to** N **do**
            $A[i,j] \leftarrow A[i,j] + B[i,k] \times C[k,j]$
        od
    od
od

Note that $A[i,j]$ is really

$$Addr(A) + ((i-1) \times K_1 + (j-1)) \times K_2$$

which takes 6 integer operations.

The innermost loop of this "textbook" program takes

24	integer ops
3	loads
1	floating add
1	floating mpy
1	store
30	instructions

Copyright ©1994 Ron K. Cytron. All rights reserved

Optimizing matrix multiply

for $i = 1$ **to** N **do**
  for $j = 1$ **to** N **do**
    $a \leftarrow \&(A[i,j])$
    for $k = 1$ **to** N **do**
      $\star a \leftarrow \star a + B[i,k] \times C[k,j]$
    od
  od
od

The expression $A[i,j]$ is *loop-invariant* with respect to the k loop. Thus, *code motion* can move the address arithmetic for $A[i,j]$ out of the innermost loop.

The resulting innermost loop contains only 12 integer operations.

for $i = 1$ **to** N **do**
  $b \leftarrow \&(B[i,1])$
  for $j = 1$ **to** N **do**
    $a \leftarrow \&(A[i,j])$
    for $k = 1$ **to** N **do**
      $\star a \leftarrow \star a + \star b \times C[k,j]$
      $b \leftarrow b + K_B$
    od
  od
od

As loop k iterates, addressing arithmetic for B changes from $B[i,k]$ to $B[i,k+1]$. *Induction variable analysis* detects the constant difference between these expressions.

The resulting innermost loop contains only 7 integer operations.

Similar analysis for C yields only 2 integer operations in the innermost loop, for a speedup of nearly 5. We can do better, especially for large arrays.

Copyright ©1994 Ron K. Cytron. All rights reserved

If optimization is...

so great because:

A good compiler can sell (even a slow) machine. Optimizing compilers easily provide a factor of two in performance. Moreover, the analysis performed during program optimization can be incorporated into the "programming environment" (29, 7, 43).

New languages and architectures motivate new program optimizations. Although some optimizations are almost universally beneficial, the advent of functional and parallel programming languages has increased the intensity of research into program analysis and transformation.

Programs can be written with attention to clarity, rather than performance. There is no substitute for a good algorithm. However, the expression of an algorithm should be as independent as possible of any specific architecture.

then:

Why does it take so long? Compilation time is usually 2–5 times slower, and programs with large procedures often take longer. Often this is the result of poor engineering: better data structures or algorithms can help in the optimizer.

Why does the resulting program sometimes exhibit unexpected behavior? Sometimes the source program is at fault, and a bug is uncovered when the optimized code is executed; sometimes the optimizing compiler is itself to blame.

Why is "no-opt" the default? Most compilations occur during the software development cycle. Unfortunately, most debuggers cannot provide useful information when the program has been optimized (45, 23). Even more unfortunately, optimizing compilers sometimes produce incorrect code. Often, insufficient time is spent testing the optimizer, and with no-opt the default, bugs in the optimizer may remain hidden.

Copyright ©1994 Ron K. Cytron. All rights reserved

Ingredients in a data flow framework

Data flow graph

$$\mathcal{G}_{df} = (\mathcal{N}_{df}, \mathcal{E}_{df})$$

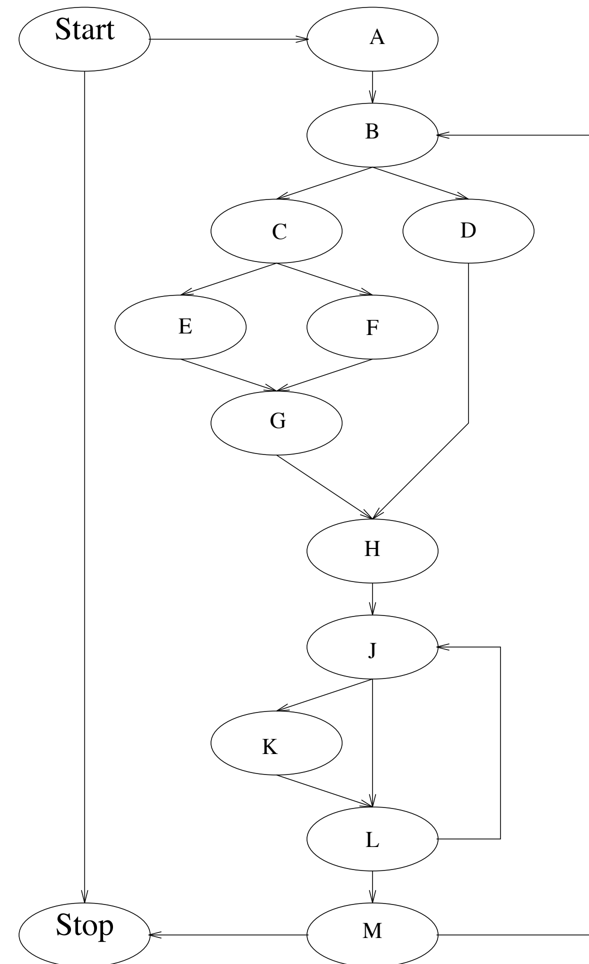
which is based on a directed *flow graph* $\mathcal{G}_f = (\mathcal{N}_f, \mathcal{E}_f)$, **typically the** *control flow graph* **of a procedure.**

A data flow problem is

forward if the solution at a node may depend only on the program's past behavior;

backward if the solution at a node may depend only on a program's future behavior;

bidirectional if both past and future behavior is relevant (12, 13, 14).



- We'll assume the data flow graph is augmented with a $Start$ and $Stop$ node, and an edge from $Start$ to $Stop$.
- We'll limit our discussion to non-bidirectional problems, and assume that edges in the data flow graph are oriented in the direction of the data flow problem.

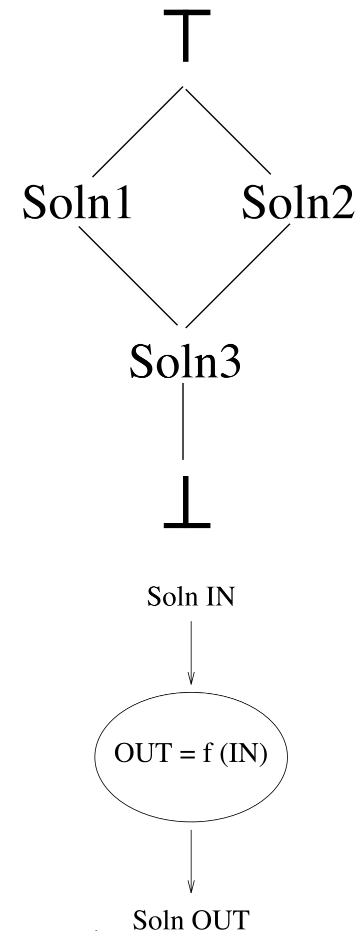
Copyright ©1994 Ron K. Cytron. All rights reserved

Ingredients in a data flow framework (cont'd)

Meet lattice which determines the outcome when disparate solutions combine. The lattice is specified with distinguished elements

- $\top$ which represents the best possible solution, and
- $\bot$ which represents the worst possible solution.

Transfer Functions which transform one solution into another.



We'll use the meet lattice to summarize the effects of convergent paths in the data flow graph, and transfer functions to model the effects of a data flow graph path on the data flow solution.

We'll begin with some simple *bit-vectoring* data flow problems, classically solved as operations on bit-vectors. For ease of exposition, we'll associate data flow solutions with the edges, rather than the nodes, of the data flow graph.

Copyright ©1994 Ron K. Cytron. All rights reserved

Available expressions

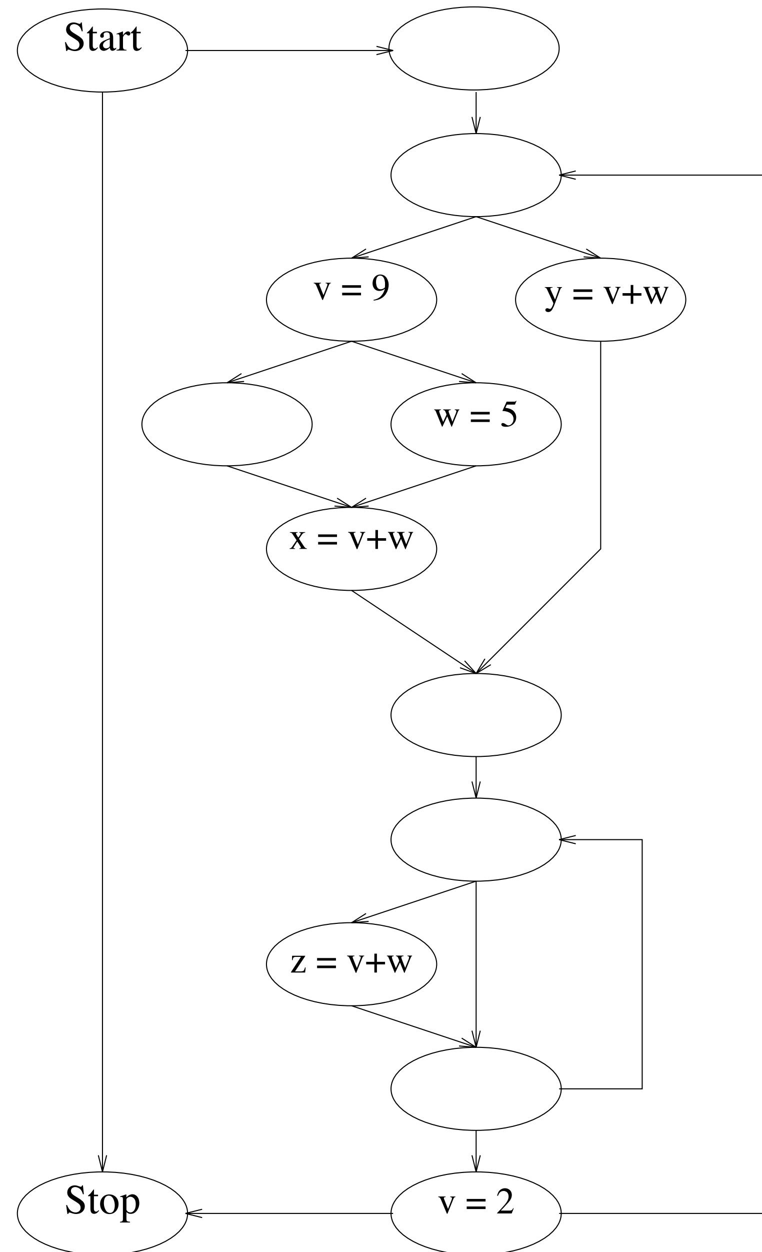
An expression $expr$ is *available* ($Avail$) at flow graph edge e if any past behavior of the program includes a computation of the value of $expr$ at e.

Consider the expression $(v + w)$ in the flow graph shown to the right. If the expression is available at the assignment to z, then it need not be recomputed.

- **This is a forward problem, so the data flow graph will have the same edges and $Start$ and $Stop$ nodes as the flow graph.**
- **The solution for any given $expr$ is either $Avail$ or $\overline{Avail}$.**
- **The "best" solution for an expression is $Avail$. We thus obtain the two-level lattice:**

  $\top$ **is** $Avail$.

  $\bot$ **is** $\overline{Avail}$.

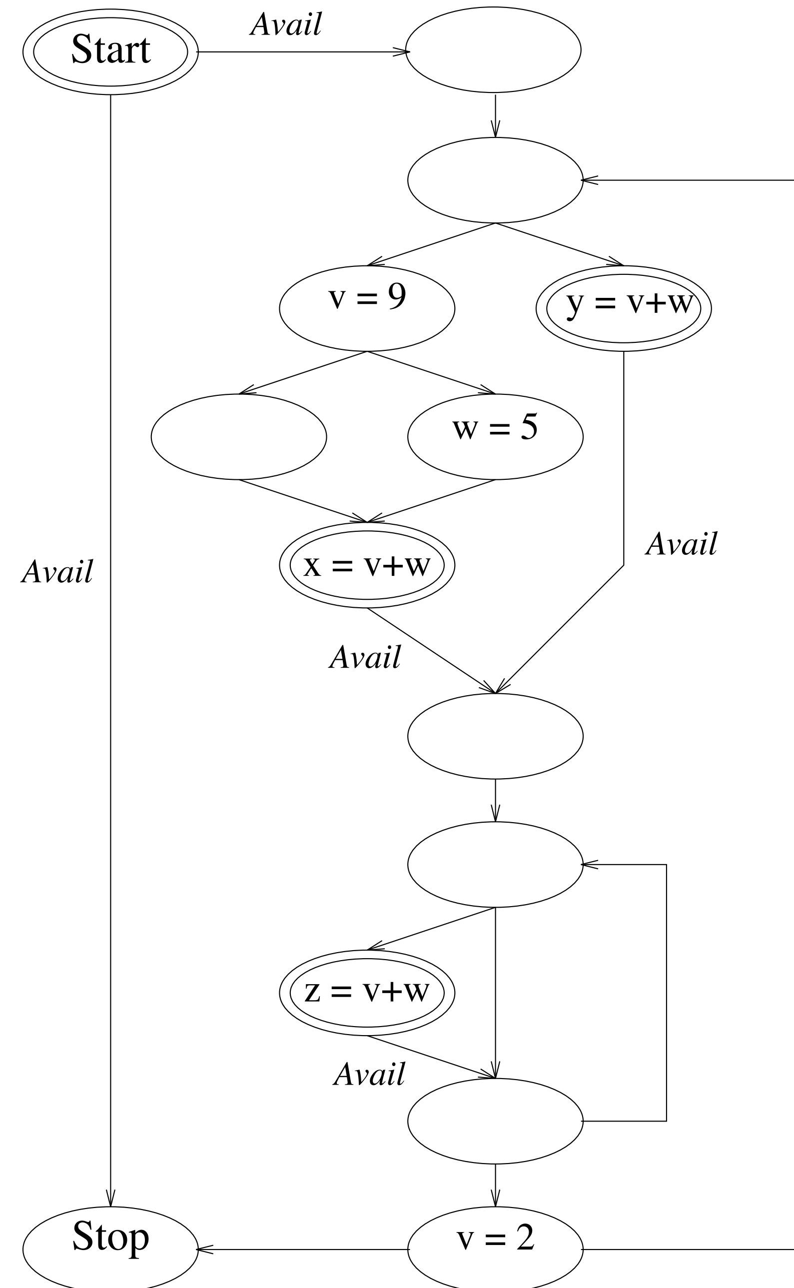


Copyright ©1994 Ron K. Cytron. All rights reserved

Available expressions(cont'd)

Nodes that compute an expression make that expression available. We also assume that every expression is available from $Start$.

The transfer function for each highlighted node makes the expression $(v + w)$ $Avail$, regardless of the solution present at the node's input.

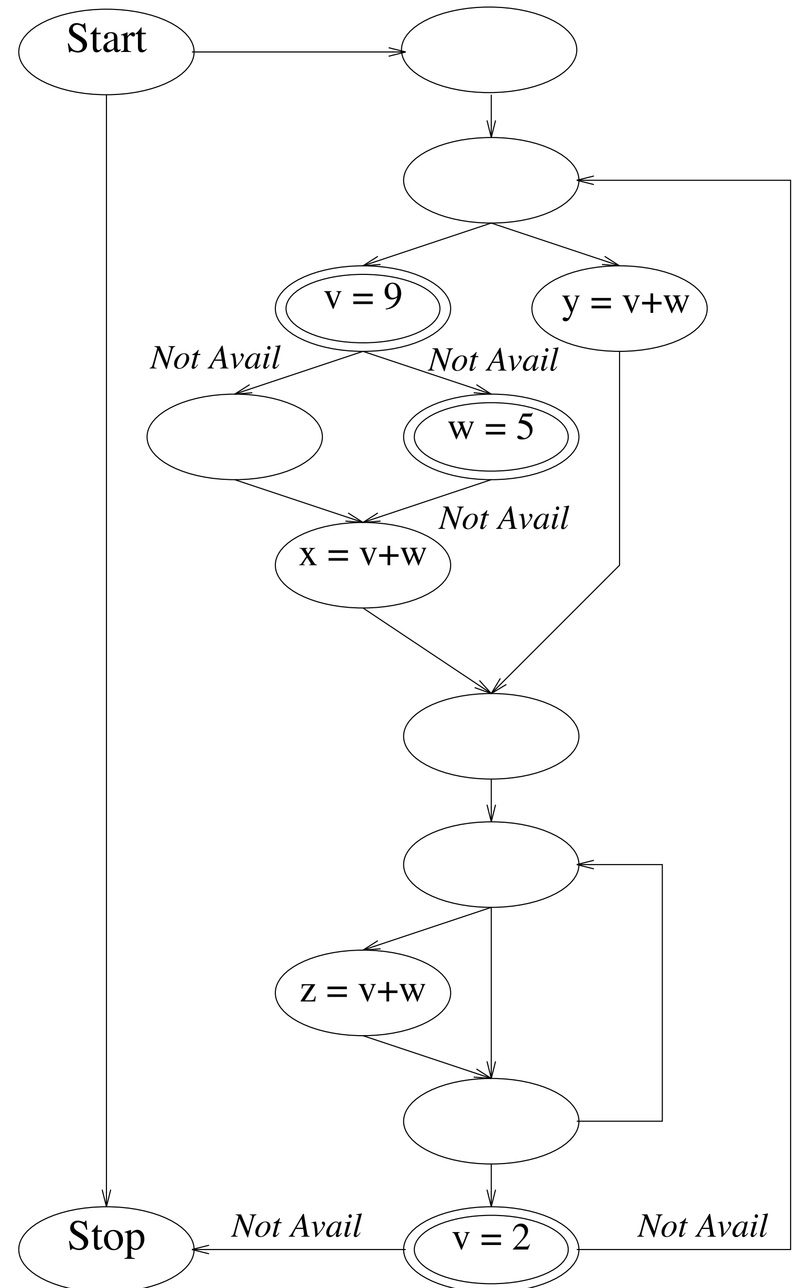


Copyright ©1994 Ron K. Cytron. All rights reserved

Available expressions(cont'd)

Nodes that assign to any variable in an expression make that expression not available, even if the variable's value is unchanged.

The transfer function for each highlighted node makes the expression $(v+w)$ $\overline{Avail}$, regardless of the solution present at the node's input.



Copyright ©1994 Ron K. Cytron. All rights reserved

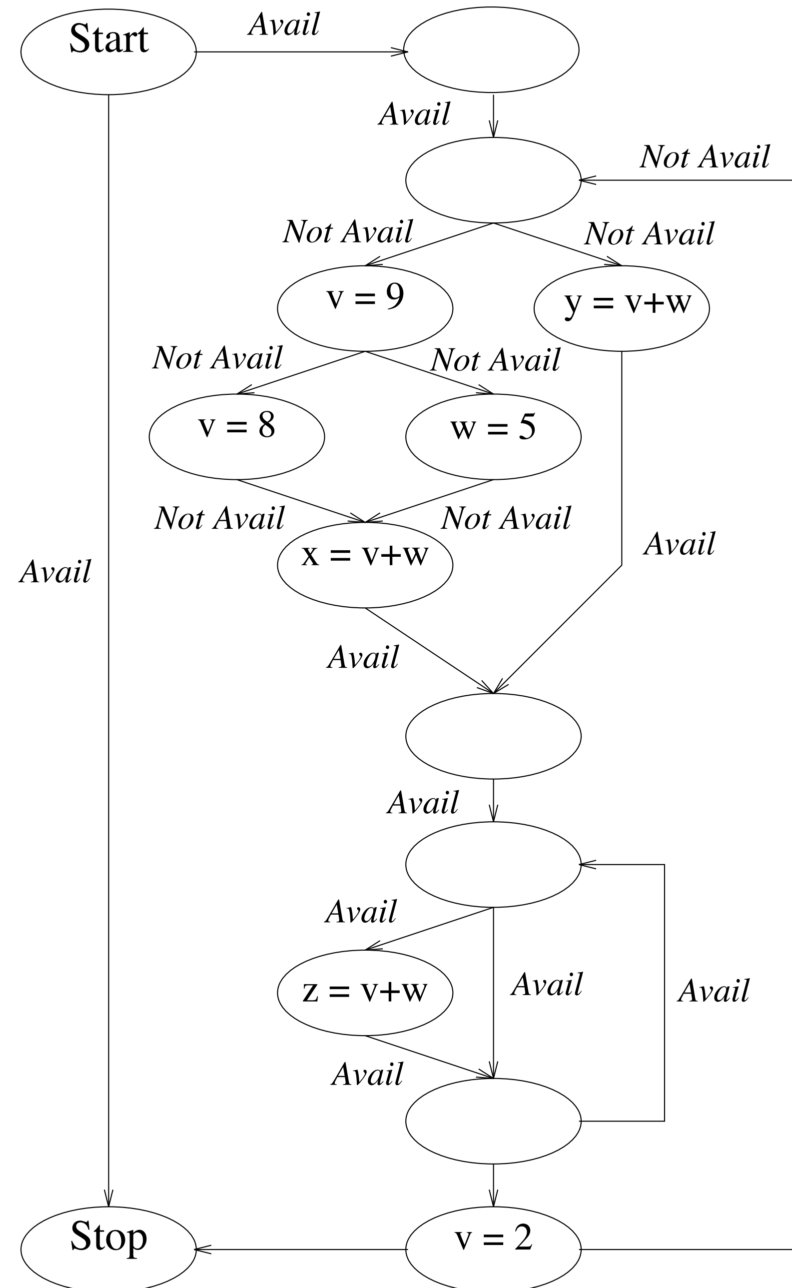
Available expressions (cont'd)

Here we see the global solution for availability of the expression $(v + w)$.

Each of the highlighted nodes shown previously asserts a solution on its output edge(s). It's the job of global data flow analysis to assign the best possible solution to every edge in the data flow graph, consistent with the asserted solutions.

The expression $(v + w)$ need not be computed in the assignment to z. The relevant value is held either in x, or y, depending on program flow.

To solve this problem using bit-vectors, assign each expression a position in the bit-vector. When an expression is available, its associated bit is 1.



Live variables

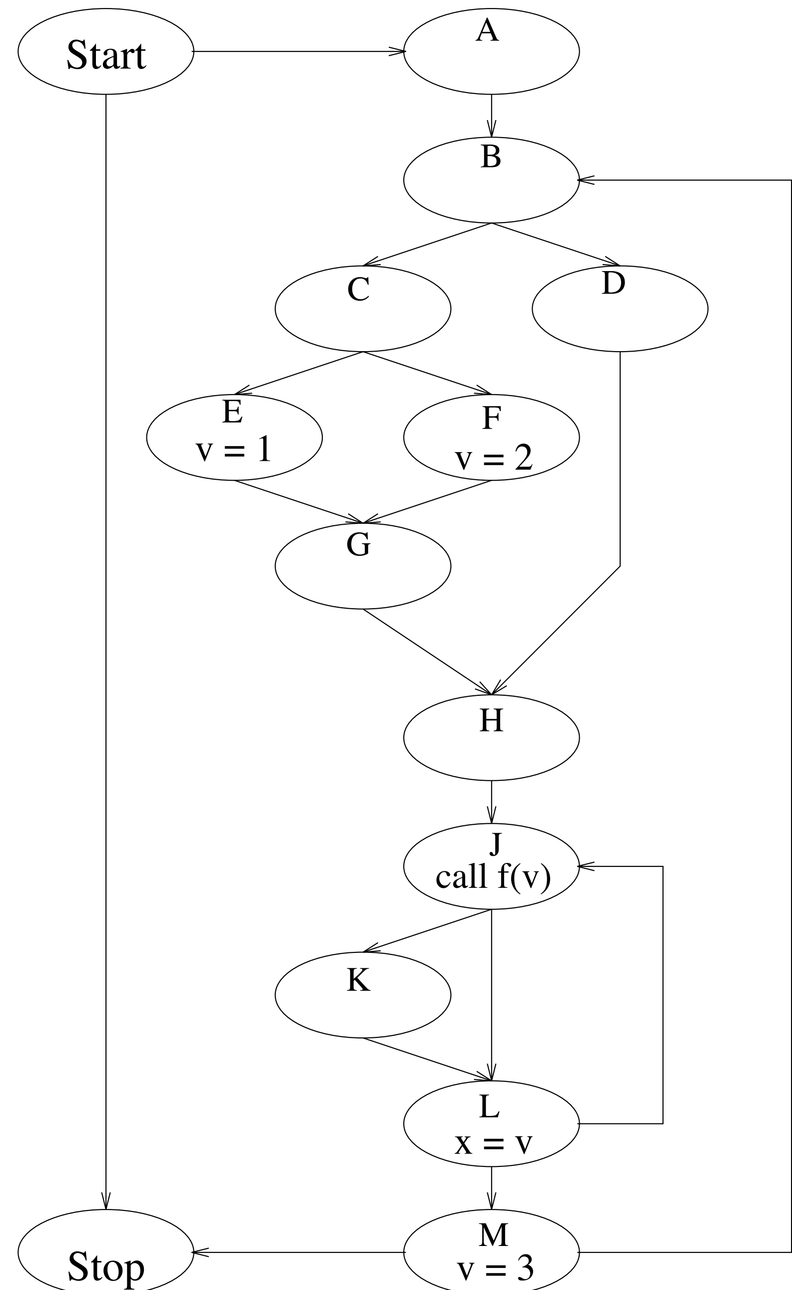
A variable v is *live* at edge e if the future behavior of the program may reference the value of v at e.

If a variable v is not live, then any resources associated with v (registers, storage, etc.) may be reclaimed.

- **This is a backward problem.**
- **In the bit-vector representation, each variable is associated with a bit.**
- **The "best" solution is $\overline{Live}$, so we obtain the two-level lattice:**

  $\top$ **is** $\overline{Live}$.

  $\bot$ **is** $Live$.

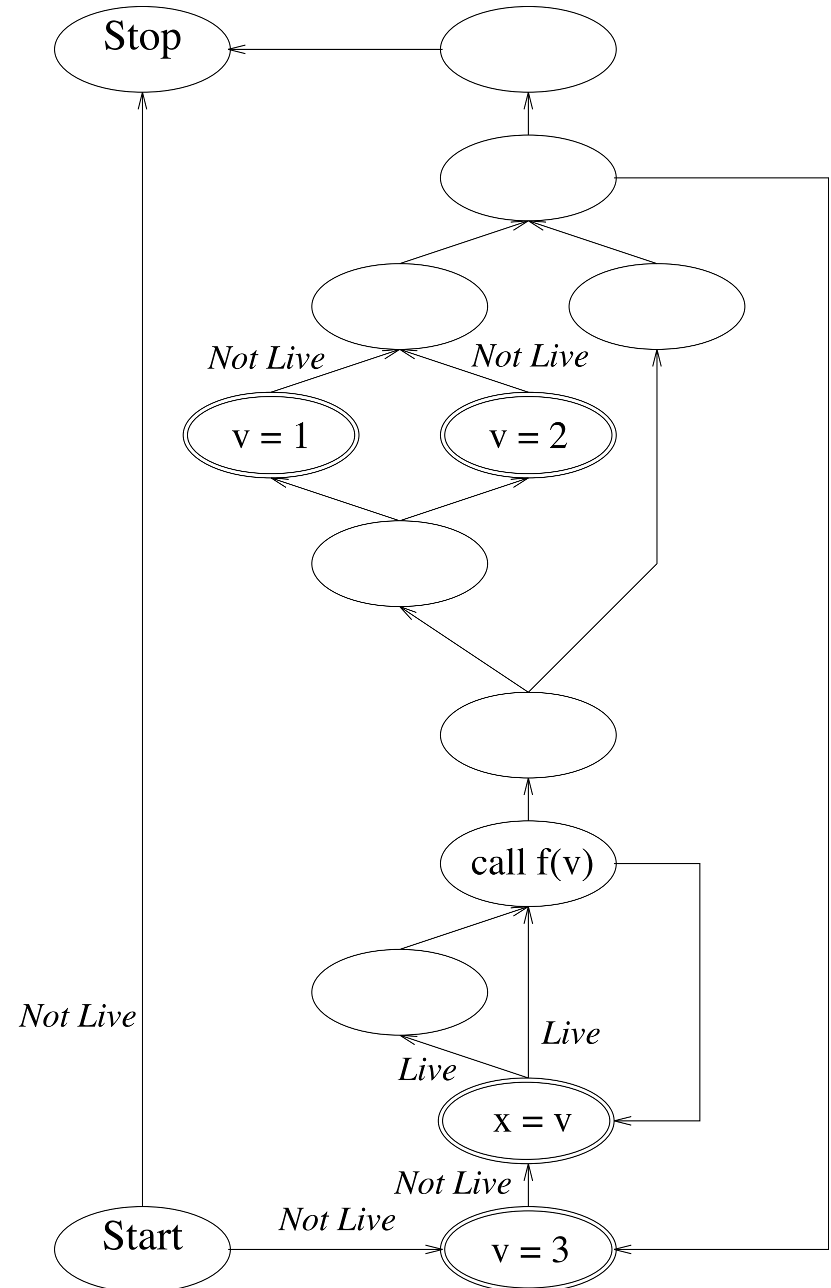


Copyright ©1994 Ron K. Cytron. All rights reserved

Live variables (cont'd)

Each of the highlighted nodes affects the data flow solution:

- **If a node uses v, then the node's asserts that v is $Live$.**
- **If a node kills v, then the node's output asserts that v is $\overline{Live}$.**



Copyright ©1994 Ron K. Cytron. All rights reserved

Live variables (cont'd)

If a node Y preserves v (as might a procedure call), then the node does not affect the solution.

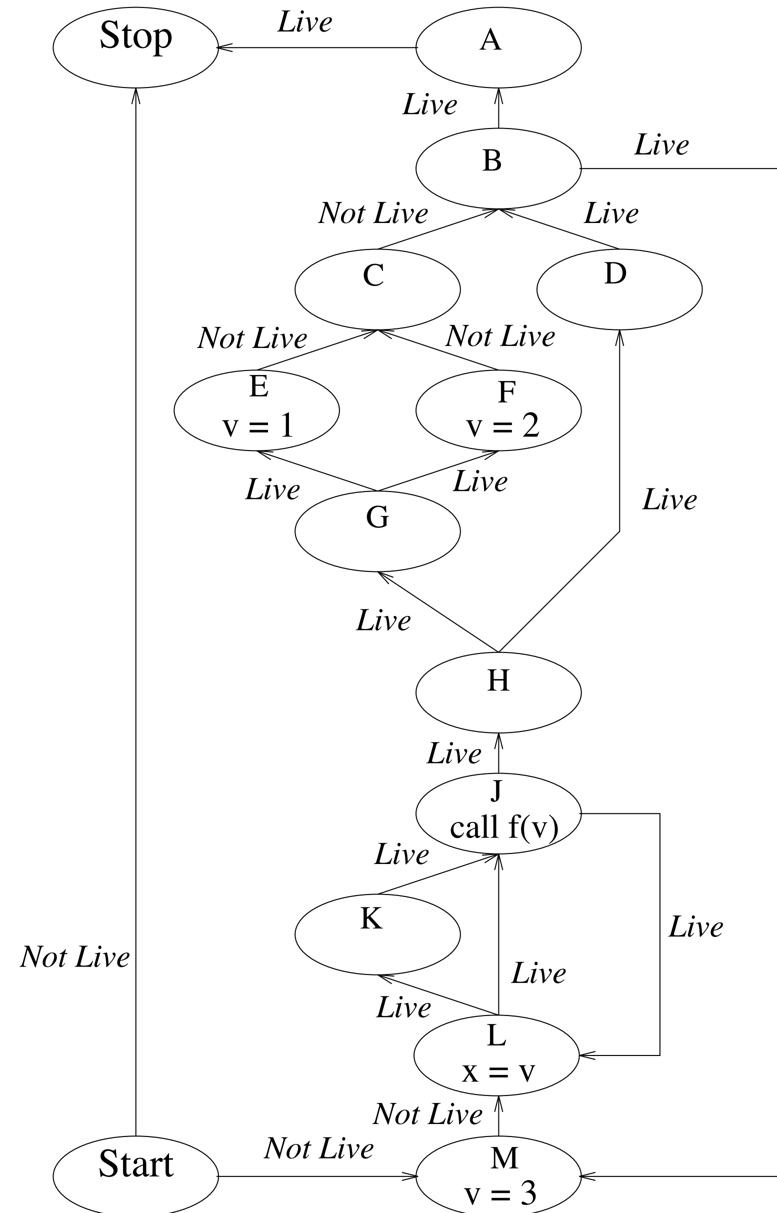
- **If v is $Live$ on "input" to Y, then Y cannot make v $\overline{Live}$.**
- **If v is $\overline{Live}$ on "input" to Y, then Y does not make v $Live$.**

Node Y's transfer function is therefore the *identity* function:

$$f_Y(IN) = IN$$

assuming node Y does not use v.

Global solution: Live variables



Formal specification of a data flow framework

The *data flow graph*

$$\mathcal{G}_{df} = (\mathcal{N}_{df}, \mathcal{E}_{df})$$

has been described previously:

- its edges are oriented in the direction of the data flow problem;
- $\mathcal{G}_{df}$ is augmented with nodes $Start$ and $Stop$ and an edge $(Start, Stop)$, suitably inserted with respect to the direction of the data flow problem.

Successors and predecessors are also defined with respect to the direction of the data flow problem:

$$\begin{aligned} Succs(Y) &= \{ Z \mid (Y, Z) \in \mathcal{E}_{df} \} \\ Preds(Y) &= \{ X \mid (X, Y) \in \mathcal{E}_{df} \} \end{aligned}$$

The *meet semilattice* is

$$L = (A, \top, \bot, \preceq, \wedge)$$

A is a set (usually a powerset), whose elements form the domain of the data flow problem,

$\top$ and $\bot$ are distinguished elements of A, usually called "top" and "bottom", respectively,

$\preceq$ is a reflexive partial order, and

$\wedge$ is the associative and commutative *meet* operator, such that for any $a, b \in A$,

$$\begin{aligned} a \preceq b &\iff a \wedge b = a \\ a \wedge a &= a \\ a \wedge b &\preceq a \\ a \wedge b &\preceq b \\ a \wedge \top &= a \\ a \wedge \bot &= \bot \end{aligned}$$

These rules allow formal reasoning about $\top$ and $\bot$ in a framework.

Copyright ©1994 Ron K. Cytron. All rights reserved

Formal specification (cont'd)

The set F of *transfer functions*

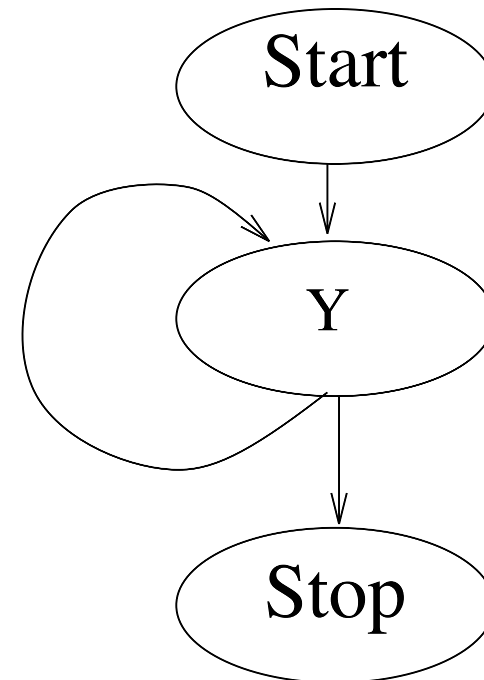
$$\mathcal{F} \subseteq \{f : L \mapsto L\}$$

has elements for describing the behavior of any flow graph node with respect to the data flow problem.

To obtain a stable solution, we'll require the functions in $\mathcal{F}$ to be *monotone*:

$$(\forall f \in \mathcal{F})(\forall x, y \in L)$$
$$x \preceq y \rightarrow f(x) \preceq f(y)$$

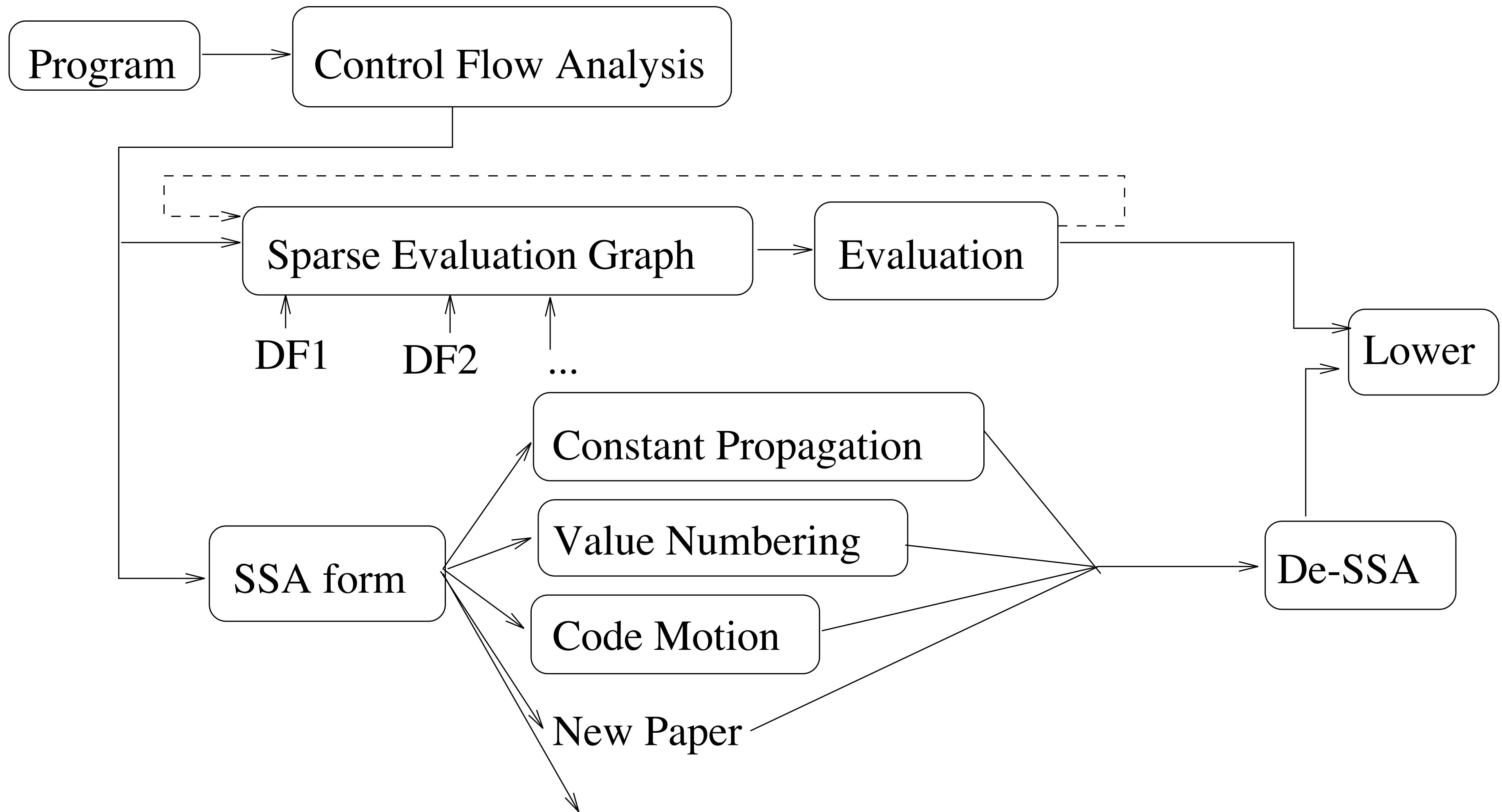
In other words, a node cannot produce a "better" solution when given "worse" input. Given a two-level lattice, evaluation of the data flow graph shown to the right oscillates between solutions and never reaches a fixed point.



$$f_Y(IN) = \begin{cases} \top & \textbf{if } IN = \bot \\ \bot & \textbf{if } IN = \top \end{cases}$$

Copyright ©1994 Ron K. Cytron. All rights reserved

Big picture

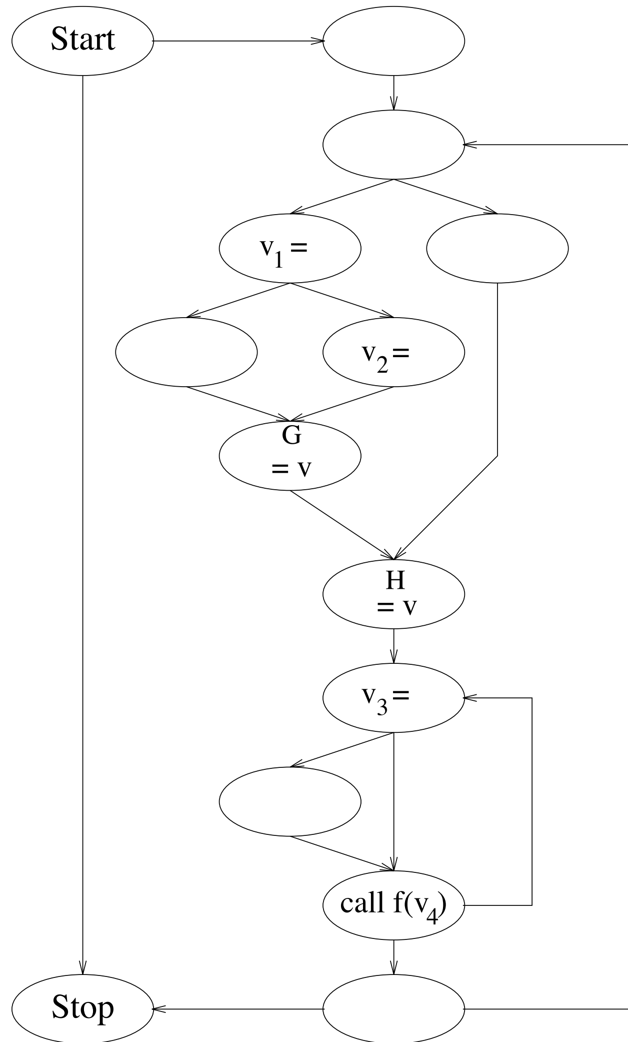


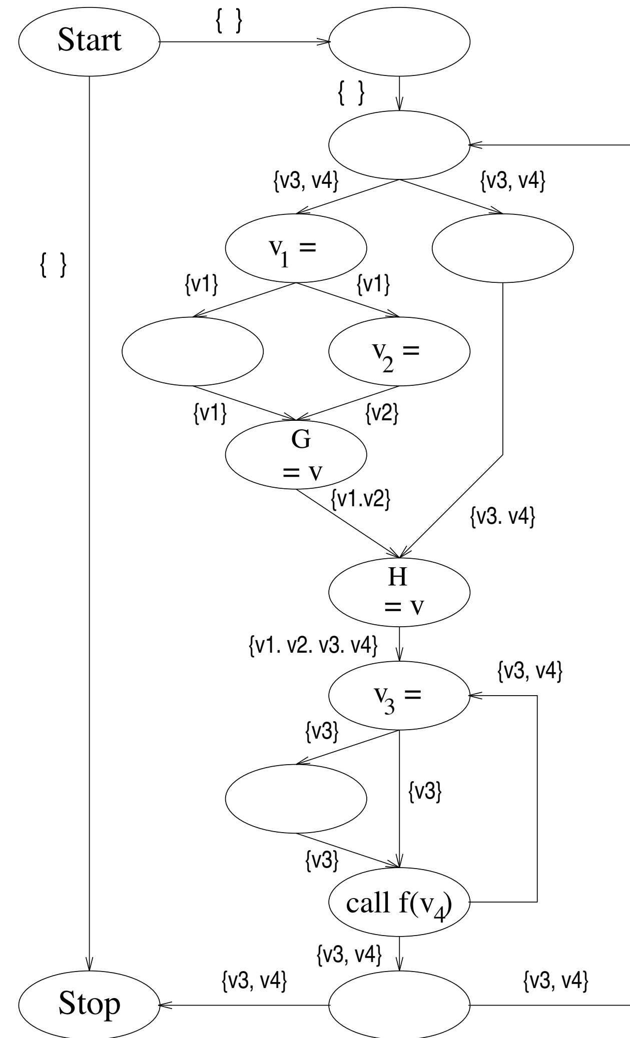
We'll now examine some special algorithms for optimization, based on a single assignment representation.

Copyright ©1994 Ron K. Cytron. All rights reserved

Static Single Assignment (SSA) form

Below are shown a program and its reaching definitions.





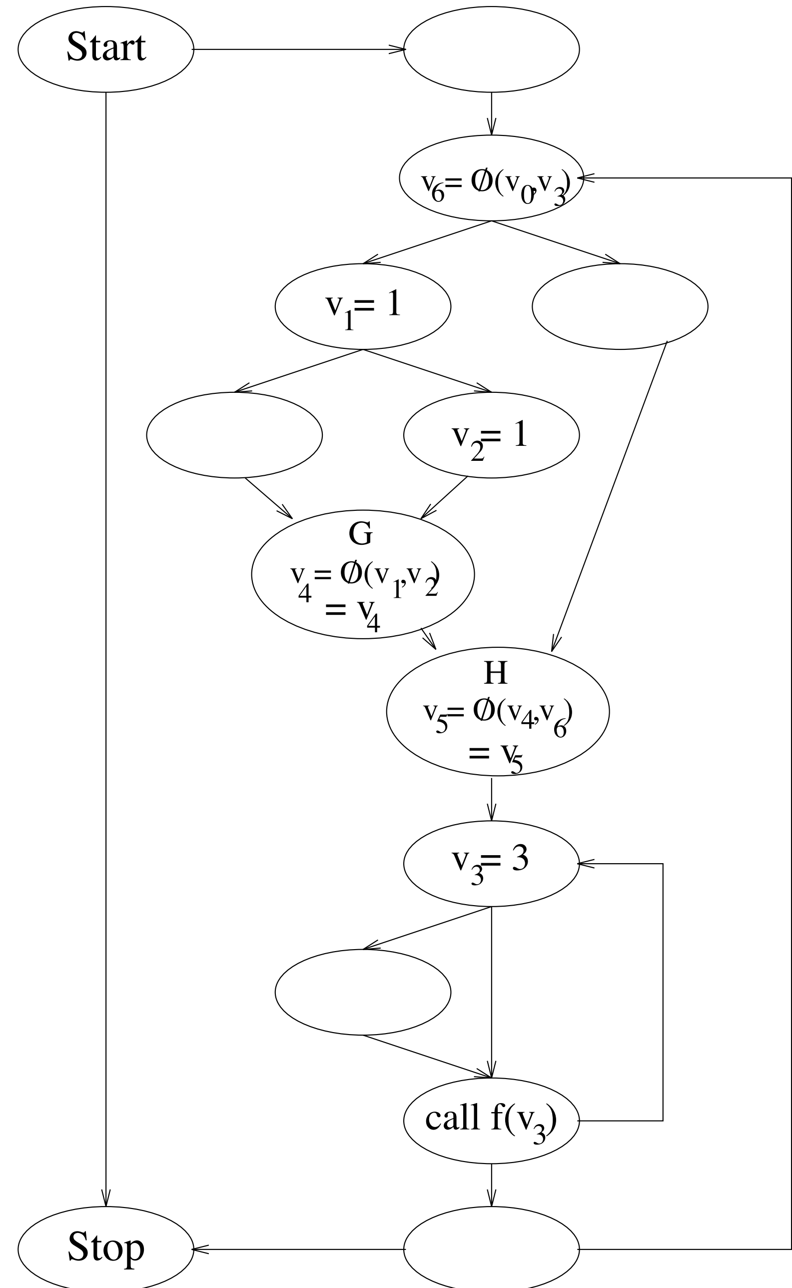
Notice how the use of v at G is reached by two definitions, and the use at H is reached by four definitions. If each use were reached by just a single definition, data flow analysis based on definitions could consult one definition per use.

Copyright ©1994 Ron K. Cytron. All rights reserved

SSA form (cont'd)

Here we see the SSA form of the program.

- Each definition of v is with respect to a distinct symbol: v_1 is as different from v_2 as x would be from y.
- Where multiple definitions reach a node, a ϕ-function is inserted, with arguments sufficient to receive a different "name" for v on each in-edge.
- Each use is appropriately renamed to the distinct definition that reaches it.
- Although ϕ-functions could have been placed at every node, the program shown has exactly the right number and placement of ϕ-functions to combine multiple defs from the original program.
- Our example assumes that procedure f does not modify v.



Copyright ©1994 Ron K. Cytron. All rights reserved

SSA form (cont'd)

Each def is now regarded as a "killing" def, even those usually regarded as preserving defs. For example, if v is *potentially* modified by the call site, then the old value for v must be passed into the called procedure, so that its value can be assigned to the name for v that *always* emerges from the procedure.

Procedure $foo(v)$

> **if** (c) **then**
> > $v \leftarrow 7$
>
> **else**
> > /⋆ Do nothing ⋆/
>
> **fi**

end

Procedure $foo(v_{out}, v_{in})$

> $v_0 \leftarrow v_{in}$
>
> **if** (c) **then**
> > $v_1 \leftarrow 7$
>
> **else**
> > /⋆ Do nothing ⋆/
>
> **fi**
>
> $v_2 \leftarrow \phi(v_0, v_1)$
>
> $v_{out} \leftarrow v_2$

end

SSA form can be computed by a data flow framework, in which the transfer function for a node with multiple reaching defs of v generates its own def of v. Uses are then named by the solution in effect at the associated node.

Copyright ©1994 Ron K. Cytron. All rights reserved

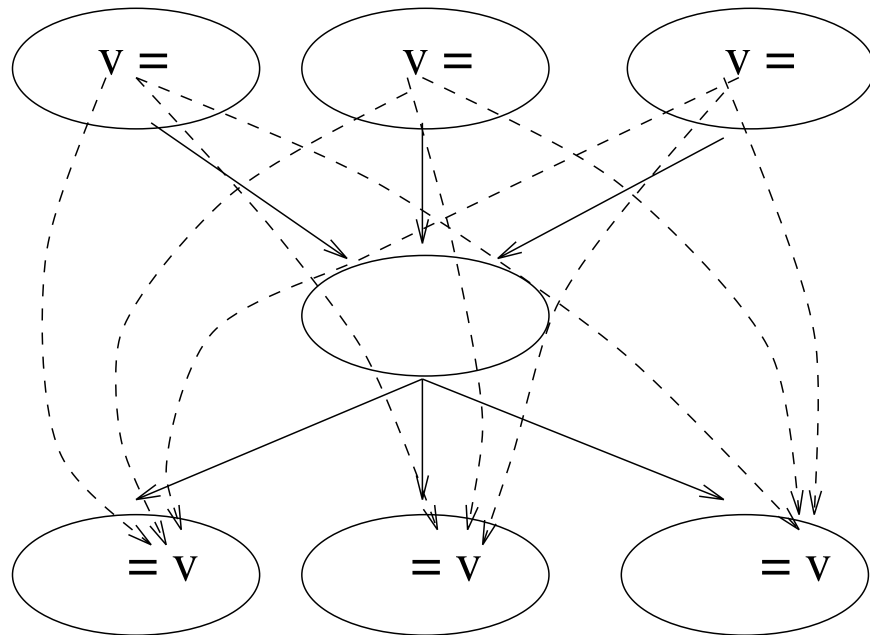
SSA form construction [9]

1. **Every preserving def is turned into a killing def, by copying potentially unmodified values (at subscripted defs, call sites, aliased defs, etc.).**
2. **Each ordinary definition of v defines a new name.**
3. **At each node in the flow graph where multiple definitions of v *meet*, a ϕ-function is introduced to represent yet another new name for v.**
4. **Uses are renamed by their dominating definition (where uses at a ϕ-function are regarded as belonging to the appropriate predecessor node of the ϕ-function).**

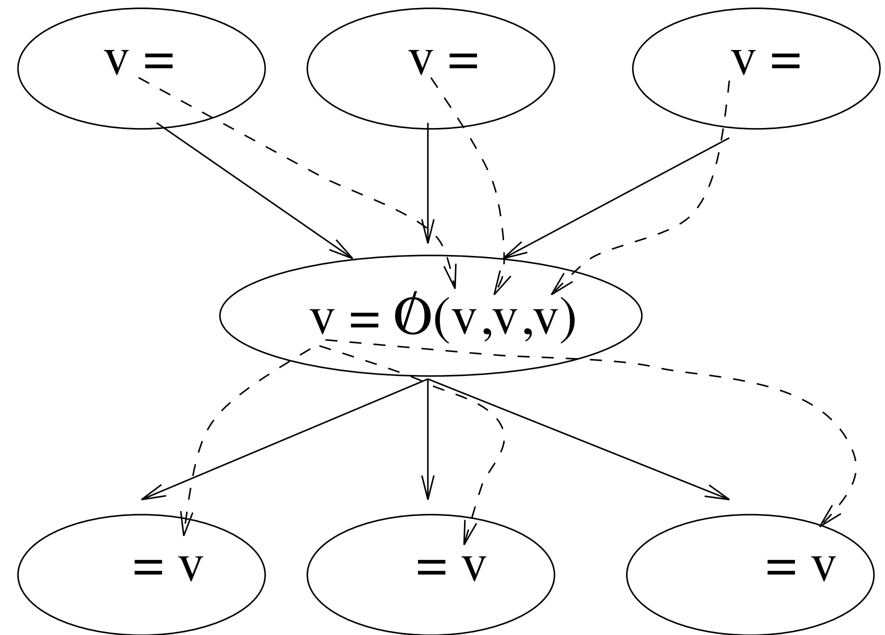
Copyright ©1994 Ron K. Cytron. All rights reserved

Why is SSA good?

Data flow algorithms built on def-use chains gain asymptotic efficiency as shown below:



Quadratic def-use chains



Linear def-use chains

With each use reached by a unique def, program transformations such as code motion are simplified: motion of a use depends primarily on motion of its unique reaching def. Intuitively, the program has been transformed to represent directly the flow of values. We'll now look at some optimizations that are simplified by SSA form.

Copyright ©1994 Ron K. Cytron. All rights reserved

SSA constant propagator [44]

Original Program

$i \leftarrow 6$
$j \leftarrow 1$
$k \leftarrow 1$
repeat
    if $(i = 6)$ **then**
        $k \leftarrow 0$
    else
        $i \leftarrow i + 1$
    fi
    $i \leftarrow i + k$
    $j \leftarrow j + 1$
until $(i = j)$

SSA form

$i_1 \leftarrow 6$
$j_1 \leftarrow 1$
$k_1 \leftarrow 1$
repeat
    $i_2 \leftarrow \phi(i_1, i_5)$
    $j_2 \leftarrow \phi(j_1, j_3)$
    $k_2 \leftarrow \phi(k_1, k_4)$
    if $(i_2 = 6)$ **then**
        $k_3 \leftarrow 0$
    else
        $i_3 \leftarrow i_2 + 1$
    fi
    $i_4 \leftarrow \phi(i_2, i_3)$
    $k_4 \leftarrow \phi(k_3, k_2)$
    $i_5 \leftarrow i_4 + k_4$
    $j_3 \leftarrow j_2 + 1$
until $(i_5 = j_3)$

Each name is initialized to the lattice value $\top$. Propagation proceeds only along edges marked *executable*. Such marking takes place when the associated condition reaches a non-$\top$ value. The value $\top$ propagates along unexecutable edges.

Copyright © 1994 Ron K. Cytron. All rights reserved.

SSA constant propagator (cont'd)

SSA Form

$i_1 \leftarrow 6$
$j_1 \leftarrow 1$
$k_1 \leftarrow 1$
repeat
  $i_2 \leftarrow \phi(i_1, i_5)$
  $j_2 \leftarrow \phi(j_1, j_3)$
  $k_2 \leftarrow \phi(k_1, k_4)$
  if $(i_2 = 6)$ **then**
    $k_3 \leftarrow 0$
  else
    $i_3 \leftarrow i_2 + 1$
  fi
  $i_4 \leftarrow \phi(i_2, i_3)$
  $k_4 \leftarrow \phi(k_3, k_2)$
  $i_5 \leftarrow i_4 + k_4$
  $j_3 \leftarrow j_2 + 1$
until $(i_5 = j_3)$

Pass 1

$i_1 \leftarrow 6$
$j_1 \leftarrow 1$
$k_1 \leftarrow 1$
repeat
  $i_2 \leftarrow \phi(i_1, i_5) = (6 \wedge \top) = 6$
  $j_2 \leftarrow \phi(j_1, j_3) = (1 \wedge \top) = 1$
  $k_2 \leftarrow \phi(k_1, k_4) = (1 \wedge \top) = 1$
  if $(i_2 = 6)$ **then**
    $k_3 \leftarrow 0$
  else
    /$\star$ Not executed $\star$/
  fi
  $i_4 \leftarrow \phi(i_2, i_3) \Rightarrow (6 \wedge \top) = 6$
  $k_4 \leftarrow \phi(k_3, k_2) \Rightarrow (0 \wedge \top) = 0$
  $i_5 \leftarrow i_4 + k_4 \Rightarrow (6 + 0) = 6$
  $j_3 \leftarrow j_2 + 1 \Rightarrow (1 + 1) = 2$
until $(i_5 = j_3 \Rightarrow (6 = 2) =$ **false**$)$

Copyright ©1994 Ron K. Cytron. All rights reserved

SSA constant propagator (cont'd)

Pass 1

$i_1 \leftarrow 6$
$j_1 \leftarrow 1$
$k_1 \leftarrow 1$
repeat
    $i_2 \leftarrow \phi(i_1, i_5) = (6 \wedge \top) = 6$
    $j_2 \leftarrow \phi(j_1, j_3) = (1 \wedge \top) = 1$
    $k_2 \leftarrow \phi(k_1, k_4) = (1 \wedge \top) = 1$
    if $(i_2 = 6)$ **then**
        $k_3 \leftarrow 0$
    else
        /⋆ Not executed ⋆/
    fi
    $i_4 \leftarrow \phi(i_2, i_3) \Rightarrow (6 \wedge \top) = 6$
    $k_4 \leftarrow \phi(k_3, k_2) \Rightarrow (0 \wedge \top) = 0$
    $i_5 \leftarrow i_4 + k_4 \Rightarrow (6 + 0) = 6$
    $j_3 \leftarrow j_2 + 1 \Rightarrow (1 + 1) = 2$
until $(i_5 = j_3 \Rightarrow (6 = 2) =$ **false**$)$

Pass 2

$i_1 \leftarrow 6$
$j_1 \leftarrow 1$
$k_1 \leftarrow 1$
repeat
    $i_2 \leftarrow \phi(i_1, i_5) = (6 \wedge 6) = 6$
    $j_2 \leftarrow \phi(j_1, j_3) = (1 \wedge 2) = \bot$
    $k_2 \leftarrow \phi(k_1, k_4) = (1 \wedge \top) = \bot$
    if $(i_2 = 6)$ **then**
        $k_3 \leftarrow 0$
    else
        /⋆ Not executed ⋆/
    fi
    $i_4 \leftarrow \phi(i_2, i_3) \Rightarrow (6 \wedge \top) = 6$
    $k_4 \leftarrow \phi(k_3, k_2) \Rightarrow (0 \wedge \top) = 0$
    $i_5 \leftarrow i_4 + k_4 \Rightarrow (6 + 0) = 6$
    $j_3 \leftarrow j_2 + 1 \Rightarrow (\bot + 1) = \bot$
until $(i_5 = j_3 \Rightarrow (6 = \bot) = \bot)$

Our solution has stabilized. Even though k_2 is $\bot$, that value is never transmitted along the unexecuted edge to the ϕ for k_4.

Copyright ©1994 Ron K. Cytron. All rights reserved

SSA value numbering [3, 39]

$a \leftarrow read()$
$v \leftarrow a + 2$
$c \leftarrow a$
$w \leftarrow c + 2$
$t \leftarrow a + 2$
$x \leftarrow t - 1$

For the above program, constant propagation will fail to determine a compile-time value for v and w, because the behavior of the $read()$ function must be captured as $\perp$ at compile-time.

Nonetheless, we can see that v and w will hold the same *value*, even though we cannot determine at compile-time exactly what that value will be. Such knowledge helps us replace the computation of $(c + 2)$ by a simple copy from v.

Value numbering attempts to label each computation of the program with a number, such that identical computations are identically labeled.

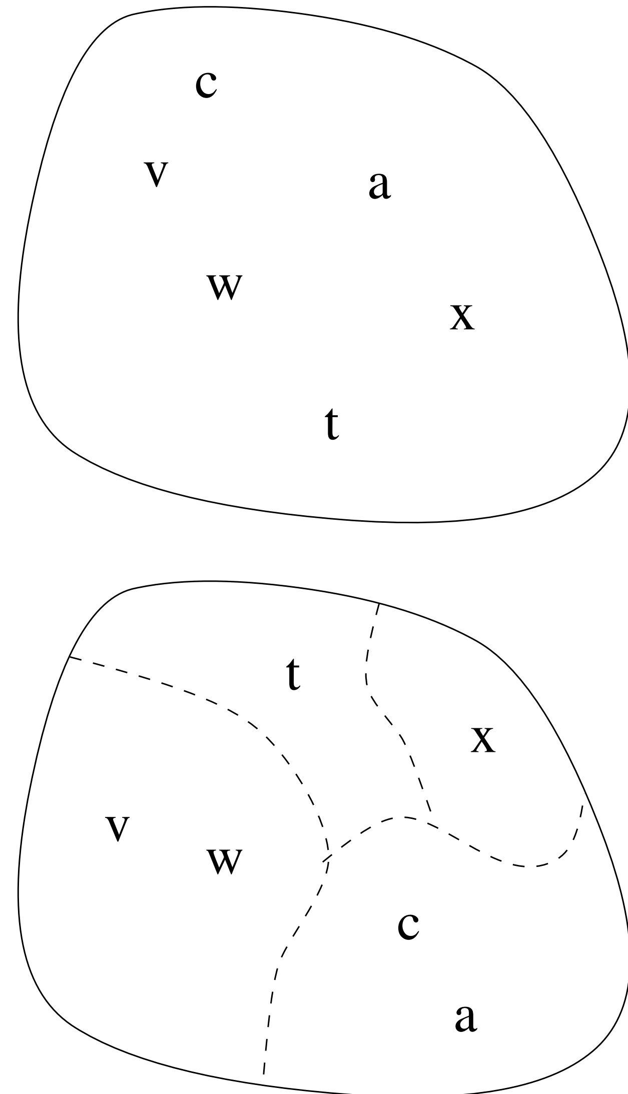
- **Prior to SSA form, value numbering algorithms were applied only within basic blocks (i.e., no branching) (2).**
- **Early value numbering algorithms relied on textual equivalence to determine value equivalence. The text of each expression (and perhaps subexpression) was *hashed* to a value number. Intervening defs of variables contained in an expression would kill the expression. This approach could not detect equivalence of v and w in the example to the left, since $(a + 2)$ is not textually equivalent to $(c + 2)$.**

It seems that x ought to have the same value as v and w, but our algorithm won't discover this, because the "function" that computes x ($\lambda n.n - 1$) differs from the "function" that computes v and w ($\lambda n.n + 2$).

Copyright ©1994 Ron K. Cytron. All rights reserved

SSA value numbering (cont'd)

- **We essentially seek a *partition* of SSA names by value equivalence, since value equivalence is reflexive, symmetric, and transitive.**
- **We'll initially assume that all SSA names have the same value.**
- **When evidence surfaces that a given block may contain disparate values (names), we'll talk about *splitting* the block. Generally, the algorithm will only split a block in two. However, the first split is more severe, in that names are split by the functional form of the expressions that compute their value.**



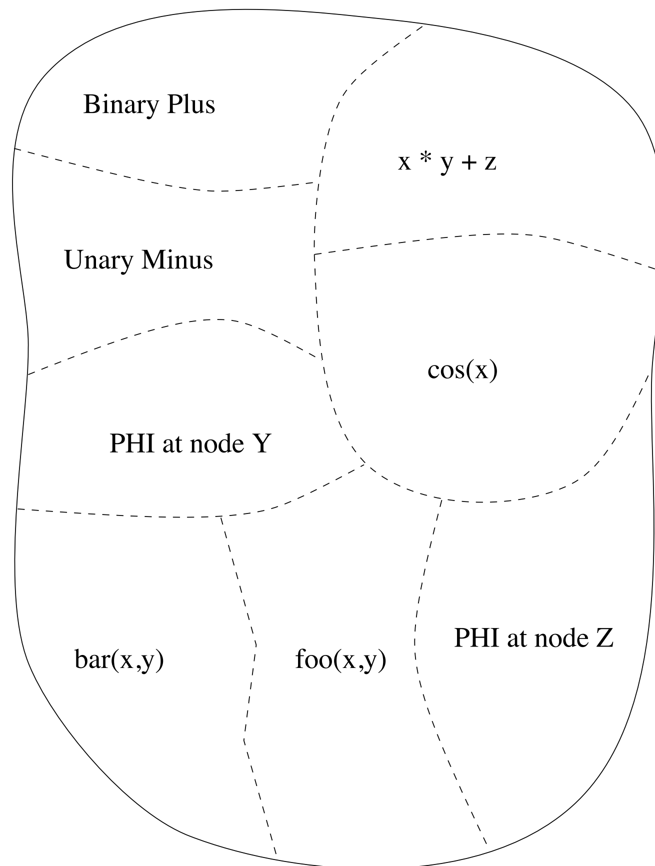
Above are shown the initial and final partitions for the example on the previous page.

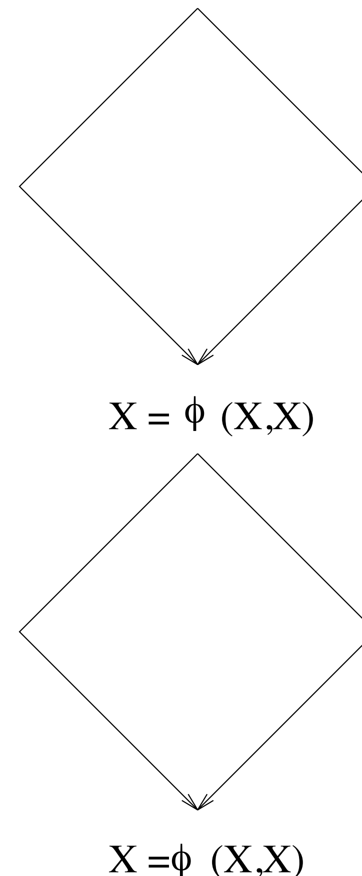
Copyright ©1994 Ron K. Cytron. All rights reserved

SSA value numbering (cont'd)

After construction of SSA form, we split by the function name that computes values for the assigned variables. We thus distinguish between binary addition, multiplication, etc.

One further point is that ϕ-functions at different nodes must also be distinguished, even though their function form appears the same. This is necessary because a branch taken into one ϕ-function is not necessarily the same branch taken into another, unless the two functions reside in the same node.





Copyright ©1994 Ron K. Cytron. All rights reserved

SSA value numbering example

if $(condA)$ **then**

  $a_1 \leftarrow \alpha$

  if $(condB)$ **then**

    $b_1 \leftarrow \alpha$

  else

    $a_2 \leftarrow \beta$

    $b_2 \leftarrow \beta$

  fi

  $a_3 \leftarrow \phi(a_1, a_2)$

  $b_3 \leftarrow \phi(b_1, b_2)$

  $c_2 \leftarrow \star a_3$

  $d_2 \leftarrow \star b_3$

else

  $b_4 \leftarrow \gamma$

fi

$a_5 \leftarrow \phi(a_1, a_0)$

$b_5 \leftarrow \phi(b_0, b_4)$

$c_3 \leftarrow \star a_5$

$d_3 \leftarrow \star b_5$

$e_3 \leftarrow \star a_5$

For brevity, symbols α, β, and γ represent syntactically distinct function classes in the program shown to the left.

In the figures that follow, we'll see that c_2 and d_2 have the same value, while c_3 and d_3 do not. Thus, program optimization will save a memory fetch by using the value of c_2 for d_2.

Note that if b is declared *volatile* in the language C, then this optimization would be incorrect, because each reference to b should be realized. How can one account for volatility in this optimization? Perhaps by assuming that volatile variables cannot have the same value.

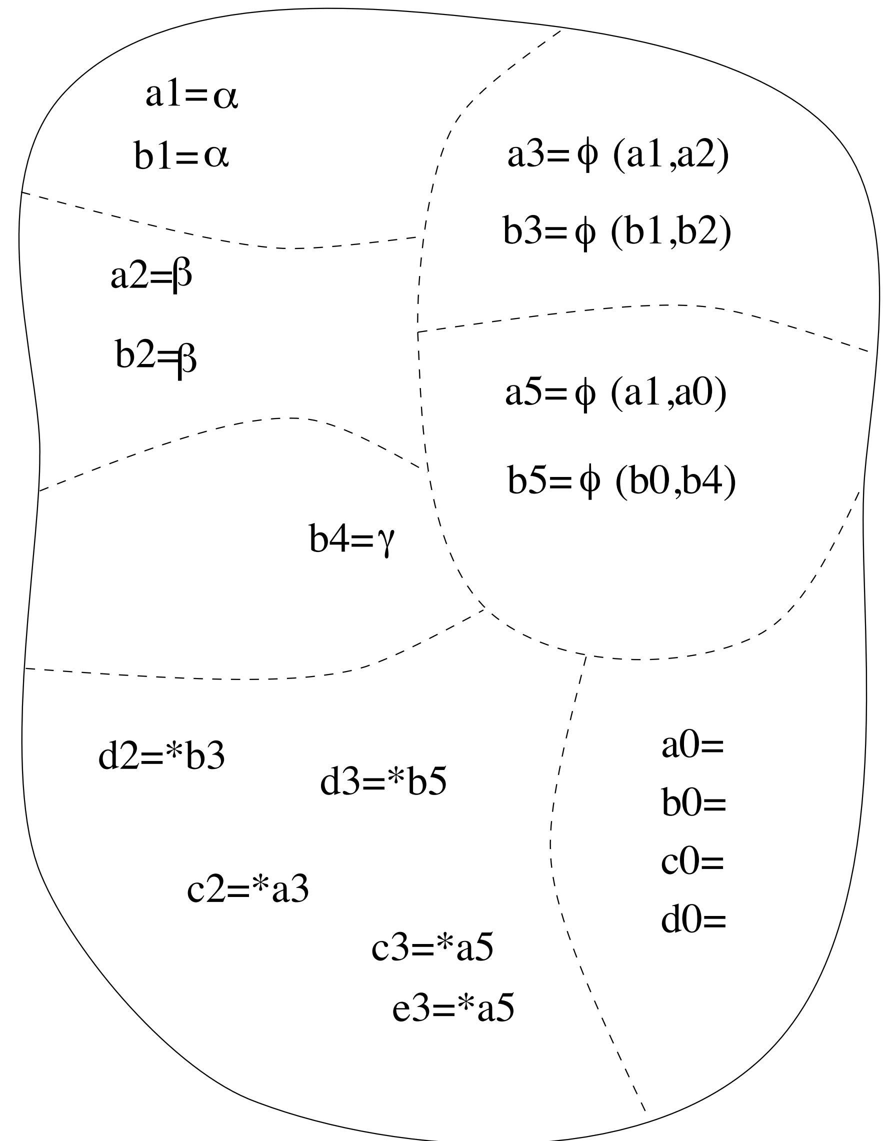
It would be difficult and expensive to express all possible defs of a volatile variable in SSA form.

Copyright ©1994 Ron K. Cytron. All rights reserved

SSA value numbering example (cont'd)

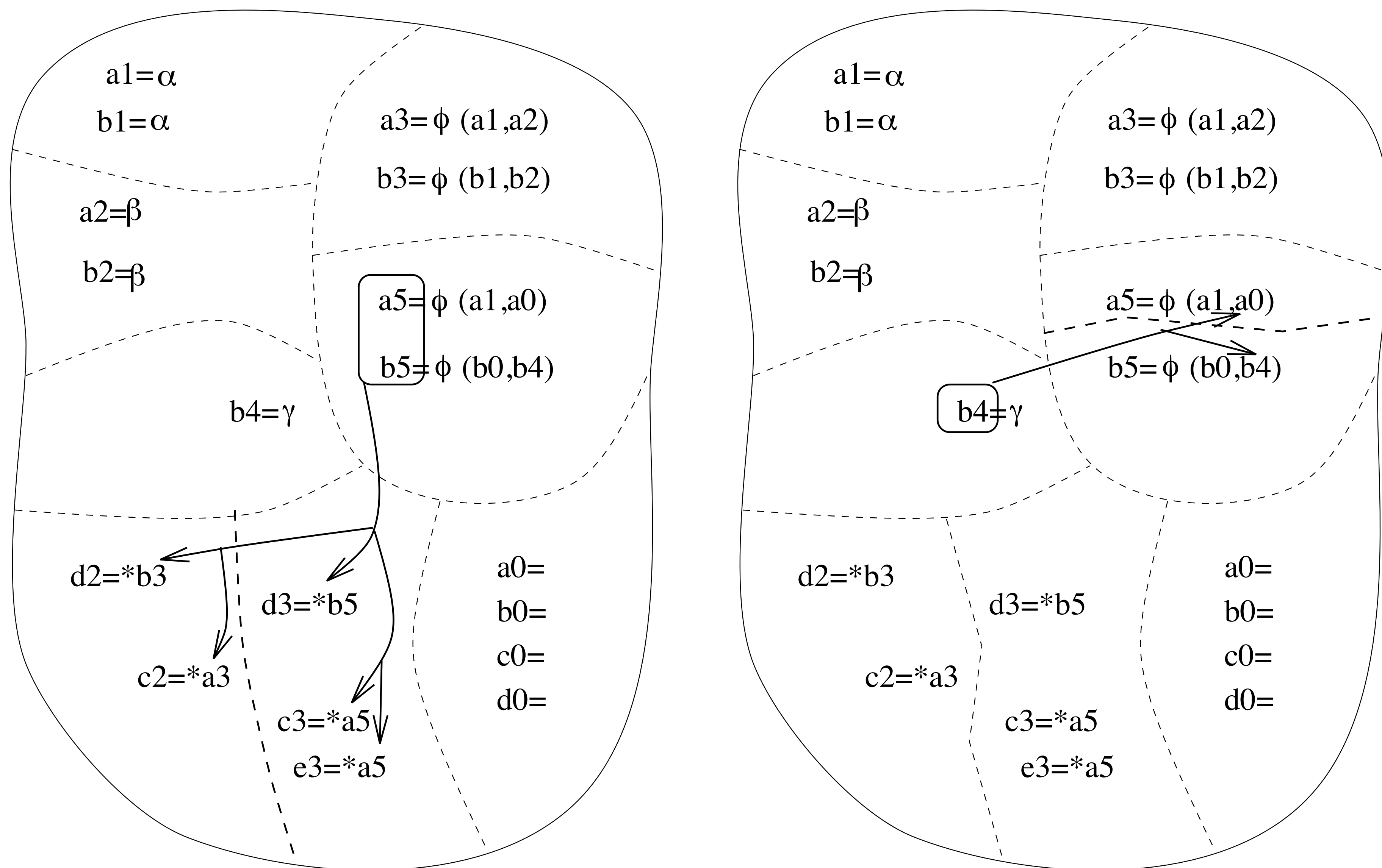
Here we see the initial partition of SSA names:

- **The syntactic classes α, β, and γ are distinguished;**
- **ϕ-functions at different nodes are distinguished;**
- **The initial value of each variable v_0 is considered identical;**
- **Within each syntactic class, values are considered identical.**



Copyright ©1994 Ron K. Cytron. All rights reserved

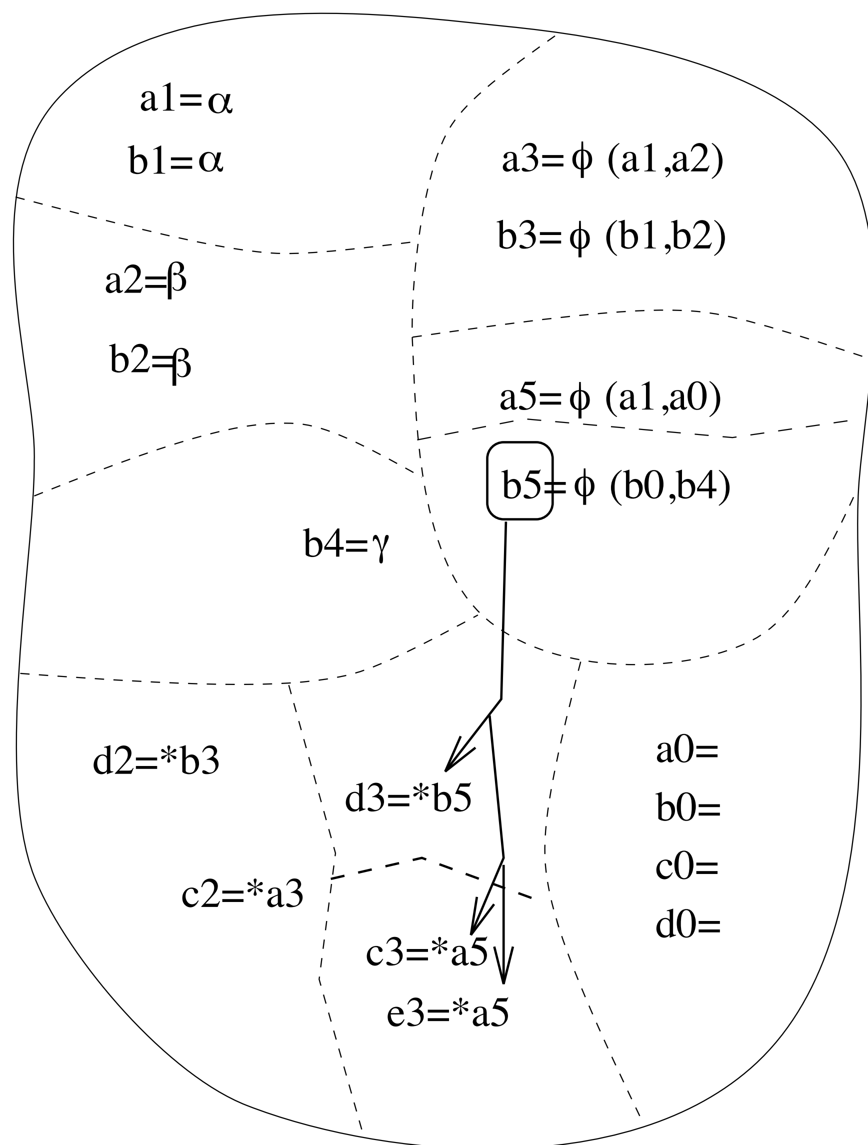
SSA value numbering example (cont'd)



On the left, the block with a_5 splits the five names shown into two subblocks; on the right, b_4 splits a_5 from b_5.

Copyright ©1994 Ron K. Cytron. All rights reserved

SSA value numbering example (cont'd)



Finally, b_5 splits c_3 from d_3. Here, note that we could have used either a_5 or b_5 to do the job. Asymptotic efficiency is gained by choosing b_5, because there are fewer uses of that name than of a_5.

In summary, the algorithm is as follows:

1. **Let W be a worklist of blocks to be used for further splitting.**
2. **Pick and remove (arbitrary) block D from W.**
3. **For each block C properly split by D,**
   - **(a) If C is on W, then remove C and enqueue its splits by D;**
   - **(b) Otherwise, enqueue the split with the fewest uses.**
4. **Loop to step 2 until W is empty.**

Copyright ©1994 Ron K. Cytron. All rights reserved

Register allocation

- **Optimal register allocation is NP-hard.**
- **Trivial approaches can be really bad: using the most recently freed register is provably worst for pipelined machines.**
- **Many approaches begin by assuming an infinite number of** *virtual registers* **for assignment to values. These are then covered by actual registers during allocation.**

Chaitin-Chandra

Each variable (or expression) is assigned a virtual register for the duration of a procedure. Actual registers are allocated by coloring an interference graph, using Chandra's heuristic. Where allocation fails, some expressions care chosen for *spilling*: **these are not kept in registers but loaded on demand and immediately stored afterwards.**

Chow-Hennessey

The maximum number of live variables is computed. Some register allocation can clearly succeed if there are sufficient registers to cover max-live. However, this may involve allocating the same variable to two different registers in different live ranges. This necessitates swapping registers for a given variable where control flow merges.

Knobe and Zadeck give a method that sloshes rather than spills: variables are kept intermittently in registers.

Copyright ©1994 Ron K. Cytron. All rights reserved