# Contents of Lecture 5

- Constant Folding
- Earlier Constant Propagation Algorithms
- Constant Propagation with Conditional Branches

# Constant Folding

```
#define KB (1024)
#define MB (KB * KB)

char buffer[8 * MB]

double f(void)
{
    double        a = 1.0/3.0;
    static double b = 1.0/3.0;

}
```

- C/C++ compilers are required to perform a simple form of constant propagation called **constant folding**.
- Floating point expressions must be evaluated as if the rounding mode is taken into account (which can be set at runtime).
- In static initializers, the default rounding mode may be used.

# Constant Propagation with Iterative Dataflow Analysis

```
a = 1;
b = 2;
if (a < b)
        c = 3;
else
        c = 4;
put(c);
```

- Invented Gary Kildall 1973.
- Every variable can be either
  - Undef
  - Constant
  - Non-constant
- Iterative dataflow analysis is performed to determine whether a variable is constant and in that case which constant.
- All branches (i.e. paths in a function) are assumed to be executable.
- Since c cannot be both 3 and 4 it's assumed to be nonconstant.

# Constant Propagation with Conditional Branches

```
a = 1;
b = 2;
if (a < b)
        c1 = 3;
else
        c2 = 4;
c3 = phi(c1, c2);
put(c3);
```

- Based on SSA Form.
- Invented at IBM Research and published 1991.
- Recall Kildall's algorithm assumed every branch was executable.
- This algorithm assumes that nothing is executable except the start vertex.
- The function is interpreted and the constant expressions are propagated.
- The interpretation proceeds until no new knowledge about constants can be found.

# Key Idea with $\phi$-functions

```
a = 1;
b = 2;
if (a < b)
        c1 = 3;
else
        c2 = 4;
c3 = phi(c1, c2);
put(c3);
```

- Thanks to SSA Form, one statement and variable is analyzed at a time.
- At a $\phi$-function, if any operand is nonconstant the result is nonconstant, and if any two constants have different values the result also is nonconstant.
- However, operands corresponding to branches which we don't think will be executed can be ignored for the moment.
- While interpreting the program we may later realize that the branch in fact might be executed and then the $\phi$-function will be re-evaluated.
- We can ignore $c_2$ and let $c_3$ be 3.

# Result from Two $\phi$-operands

| $x$ | $y$ | $x \wedge y$ |
|---|---|---|
| *nonconst* | — | *nonconst* |
| — | *nonconst* | *nonconst* |
| *undef* | *undef* | *undef* |
| *undef* | $m \in \mathbb{Z}$ | $m$ |
| $m \in \mathbb{Z}$ | *undef* | $m$ |
| $m \in \mathbb{Z}$ | $n \in \mathbb{Z}, n \neq m$ | *nonconst* |
| $m \in \mathbb{Z}$ | $n \in \mathbb{Z}, n = m$ | $m$ |

# Interpreting Unconditional Branches

```
a = 1;
b = 2;
goto L;
/* ... */
L:
```

- In the vcc compiler, an unconditional branch is called a branch-always and has mnemonic `BA`.
- The name branch-always comes from the SPARC instruction.
- A branch-always should simply tell the interpreter that the target basic block should be interpreted in the future.
- Actually we don't have a list of basic blocks waiting for interpretation but rather a list of edges.

# Interpreting Conditional Branches 1(2)

```
label  U
mov    1,a
mov    2,b
bgt    a,b,V
ba     W
```

- When the branch condition can be evaluated only one of the successors should be put on the list of edges to be interpreted.
- In this case it is the edge $(u, w)$ that is put on the list.

# Interpreting Conditional Branches 2(2)

```
label  U
mov    x,a
mov    2,b
bgt    a,b,V
ba     W
```

- Assume `x` is nonconstant.
- Both edges $(u, v)$ and $(u, w)$ are put on the list.

# Uses of a Variable on SSA Form

- Every variable has a list of instructions (three-address statements) in which it is used.
- This list is called the **uselist** of a variable and some compilers maintain it while others don't.
- With it, algorithms can be somewhat simpler but they obviously need some memory.
- For example SGI's compiler doesn't use it, while lmpcc and vcc do.
- When we have determined that the value of a variable has been lowered from Undef or Constant we must re-evaluate all executable instructions in which the variable is used.
- An instruction in vertex $v$ is executable if there is an executable edge $(u, v)$ in the control flow graph

# Two Worklists are Maintained during Interpretation

- The **edge-worklist** of new edges to interpret.
- The **ssa-worklist** of uses which need to be re-evaluated.
- The algorithm can take an object from the lists in any order and perform interpretation. The result will always be the same.
- The algorithm terminates when both lists are empty.
- The statements are modified **after** the interpretation is complete.

# Visiting a Basic Block

- Only the first time a basic block is processed are all its statements interpreted.
- On subsequent processing of $v$ due to an edge $(u, v)$ only the $\phi$-functions in $v$ must be re-evaluated.
- They have to be re-evaluated since the previous times $v$ was processed we ignored the operand corresponding to the edge $(u, v)$.
- The other statements will be re-evaluated if they enter the **ssa-worklist** and are executable.

# Main Algorithm

```
procedure cprop
    for each definition d do
        value(d) ← ⊤
    for each vertex w do
        visited(w) ← false
    visit_vertex(s)
    while (not empty(edge_worklist) or not empty(ssa_worklist)) do
        if (not empty(edge_worklist))
            edge ←take edge from edge_worklist
            if (not executable(edge))
                set_executable(edge)
                visit_vertex(head(edge))
        if (not empty(ssa_worklist))
            t ←take statement from ssa_worklist
            v ← vertex(t)
            if (any edge (u, v) is executable)
                visit_stmt(t)
```

# Visiting a Basic Block

**procedure** *visit_vertex*($w$)
    **bool** *onlyphi*

    *onlyphi* $\leftarrow$ *visited*($w$)
    *set_visited*($w$)
    **for** each statement $t$ in $w$ **do**
        **if** (*onlyphi* **and** $t$ is **not** $\phi$)
            **return**
        *visit_stmt*($t$)

# Visiting a Statement 1(3)

**procedure** $visit\_stmt(t)$
  $w \leftarrow vertex(t)$
  **switch** $(stmt\_type(t))$ {
  **case** unconditional_branch:
    $add\_edge(w, succ(w))$
    **break**

  **case** conditional_branch:
    add appropriate edges depending on what is known
    about the operands
    **break**

# Visiting a Statement 2(3)

**case** add:

  *left* $\leftarrow$ value of first source operand
  *right* $\leftarrow$ value of second source operand
  *result* $\leftarrow$ what can be determined from *left* and *right*
  **if** $(\mathit{result} < \mathit{value}(t))$
    add uses of destination of $t$ to *ssa_worklist*
    $\mathit{value}(t) \leftarrow \mathit{result}$
  **break**

# Visiting a Statement 3(3)

**case** $\phi$:

 *result* $\leftarrow \top$

 **for** each $p \in \mathit{pred}(w)$ **do**

  **if** (the edge $(p, w)$ is marked *executable*)

   *value* $\leftarrow$ value of $\phi$-function operand for $p$

   *result* $\leftarrow \mathit{result} \wedge \mathit{value}$

 **if** $(\mathit{result} < \mathit{value}(t))$

  add uses of destination of $t$ to *ssa_worklist*

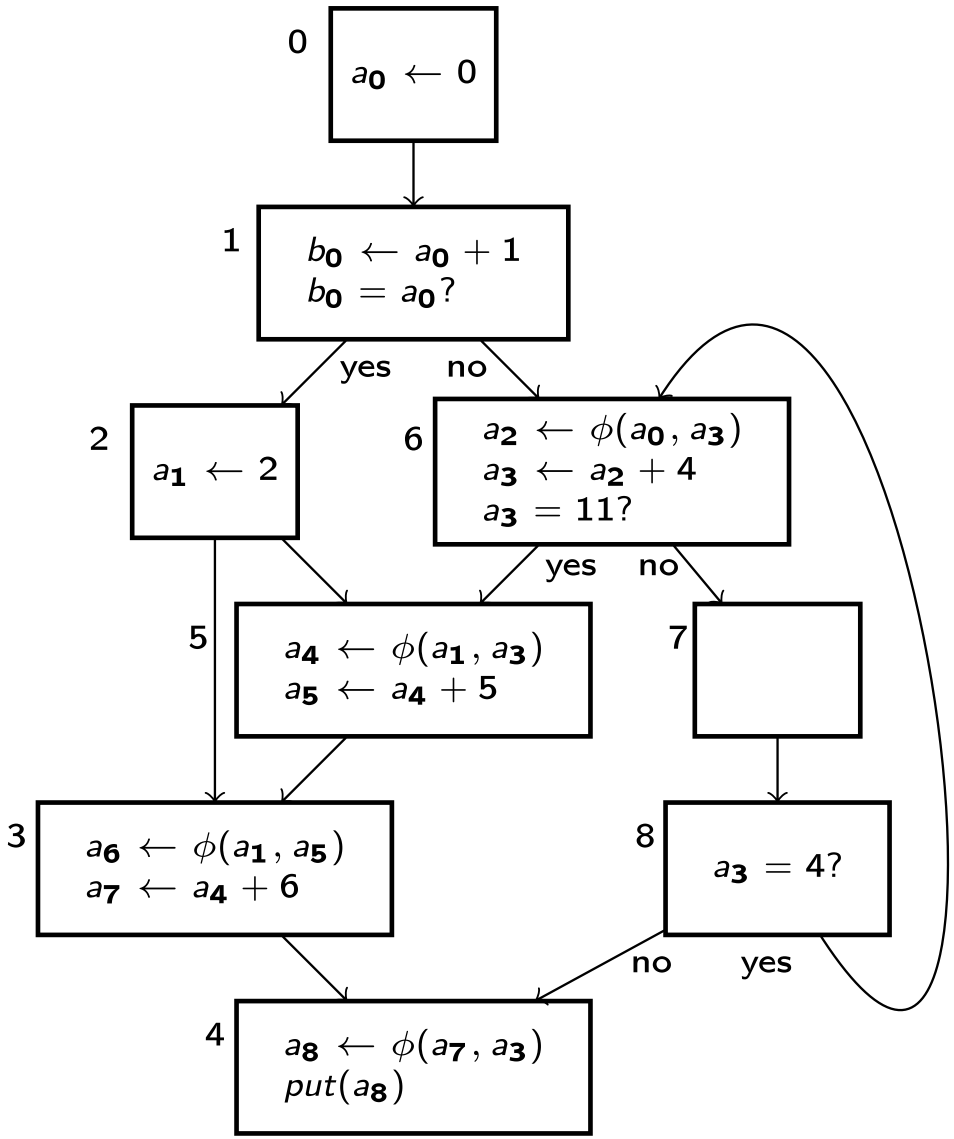
  $\mathit{value}(t) \leftarrow \mathit{result}$

 **break**

 $\vdots$

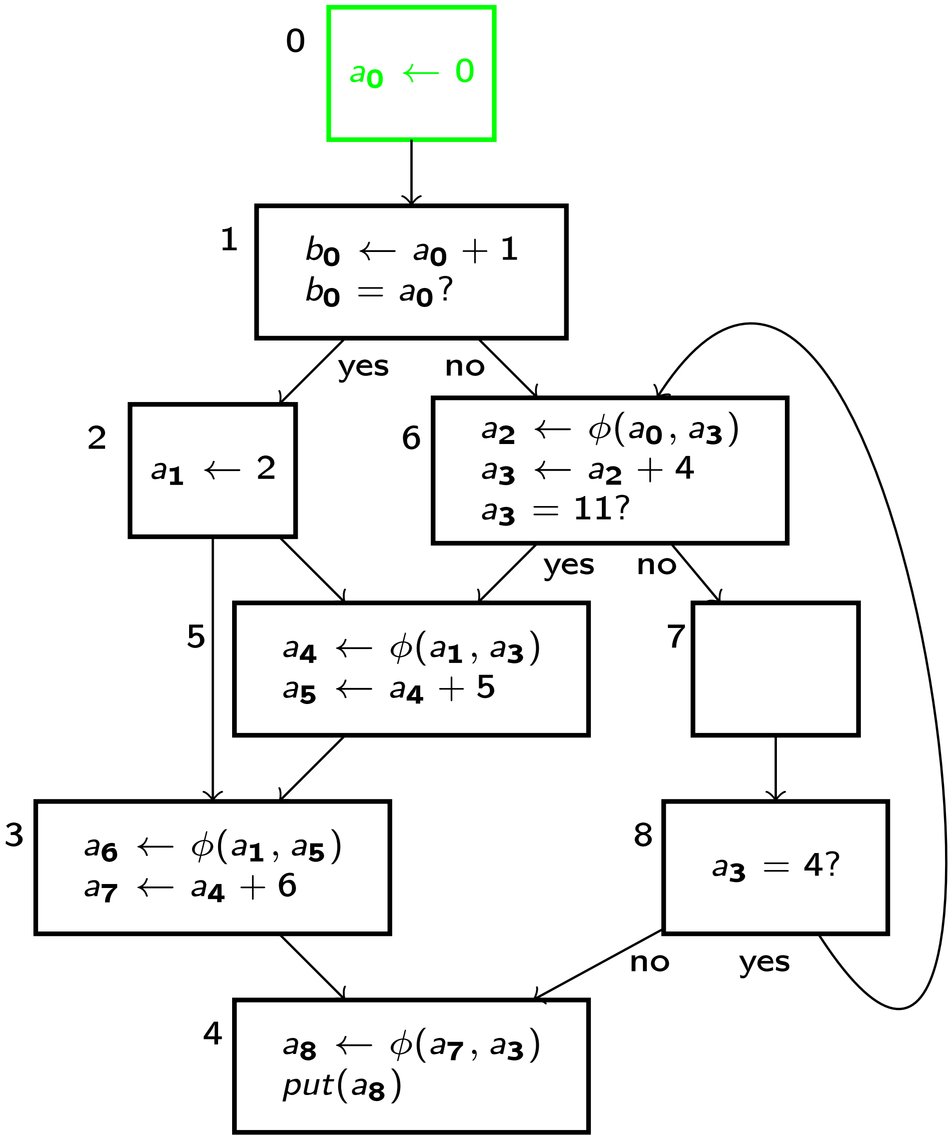
}

# An Example 1(10)



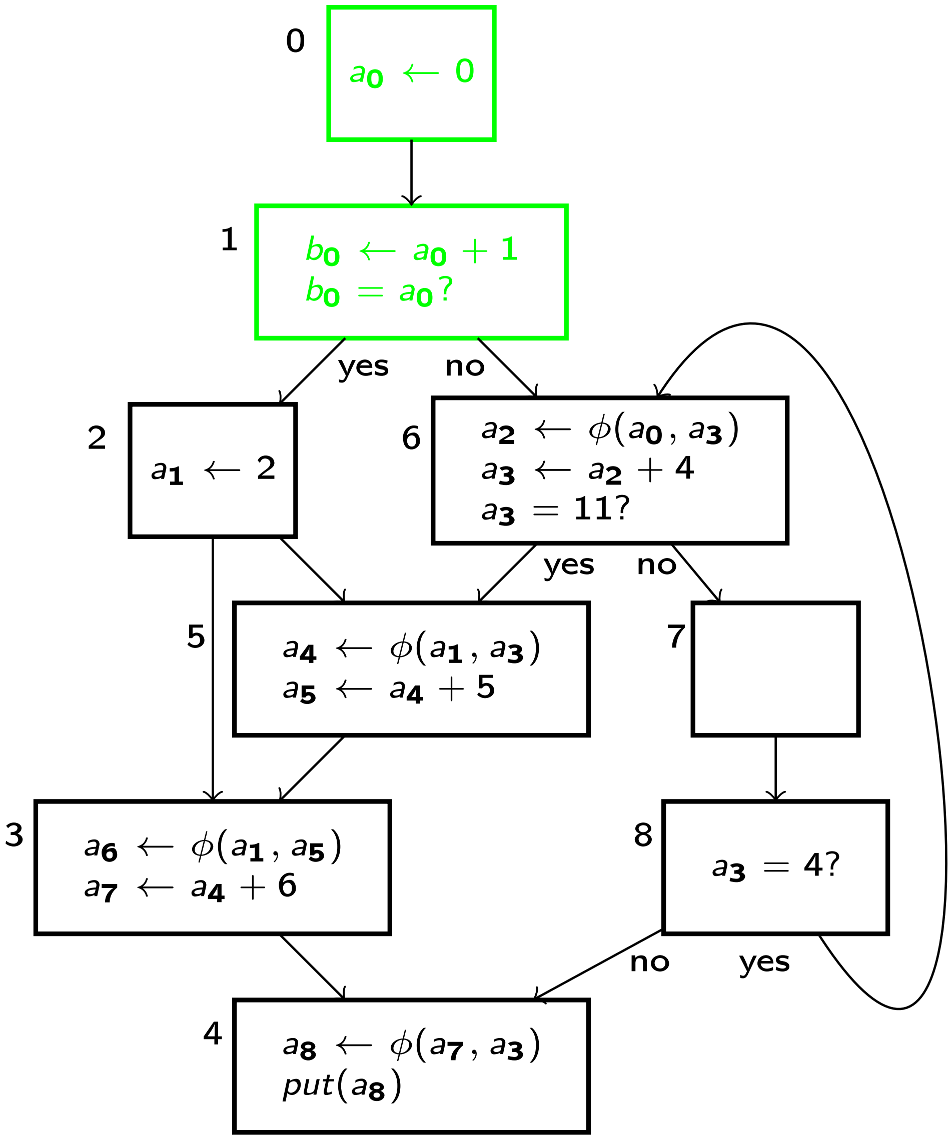

- $edge\_worklist = \emptyset$
- $ssa\_worklist = \emptyset$

# An Example 2(10): Visit 0

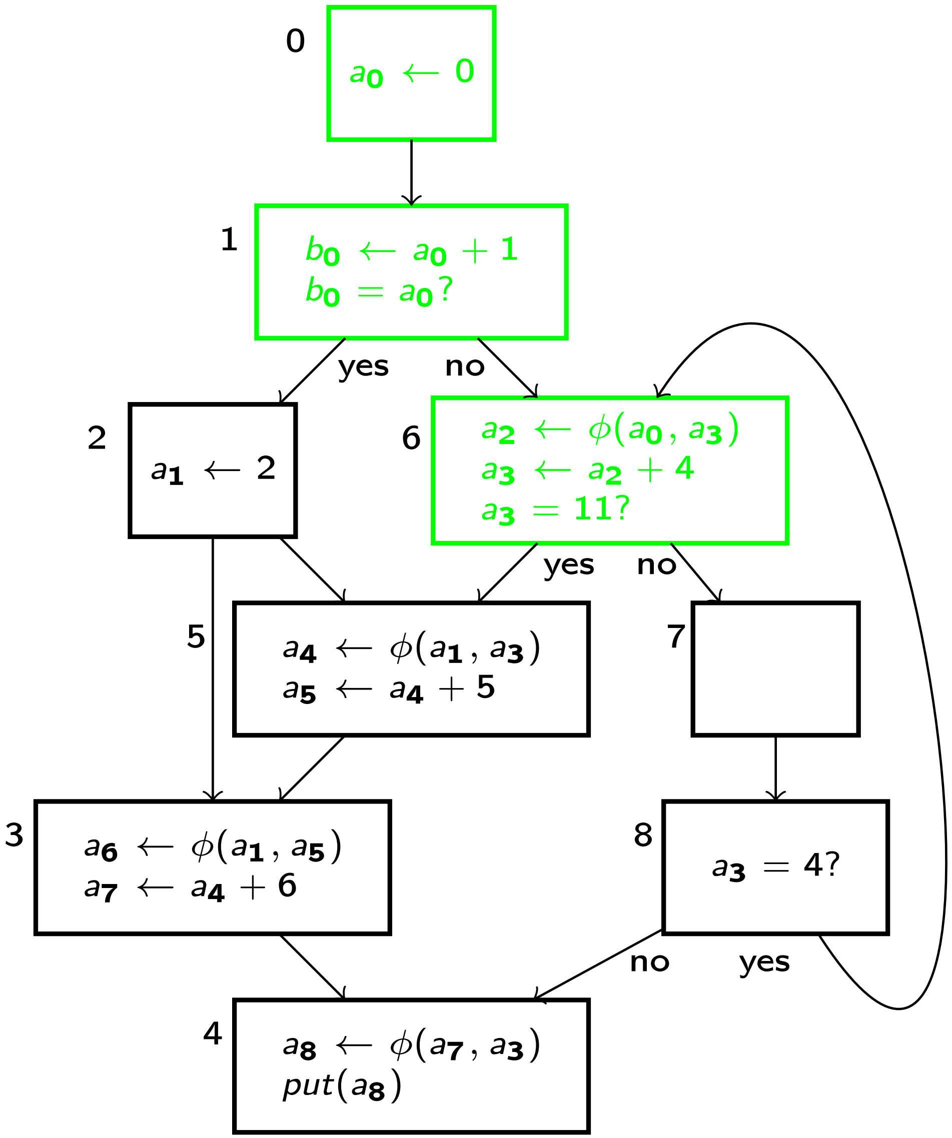



- $edge\_worklist = \{(0, 1)\}$

# An Example 3(10): Visit 1

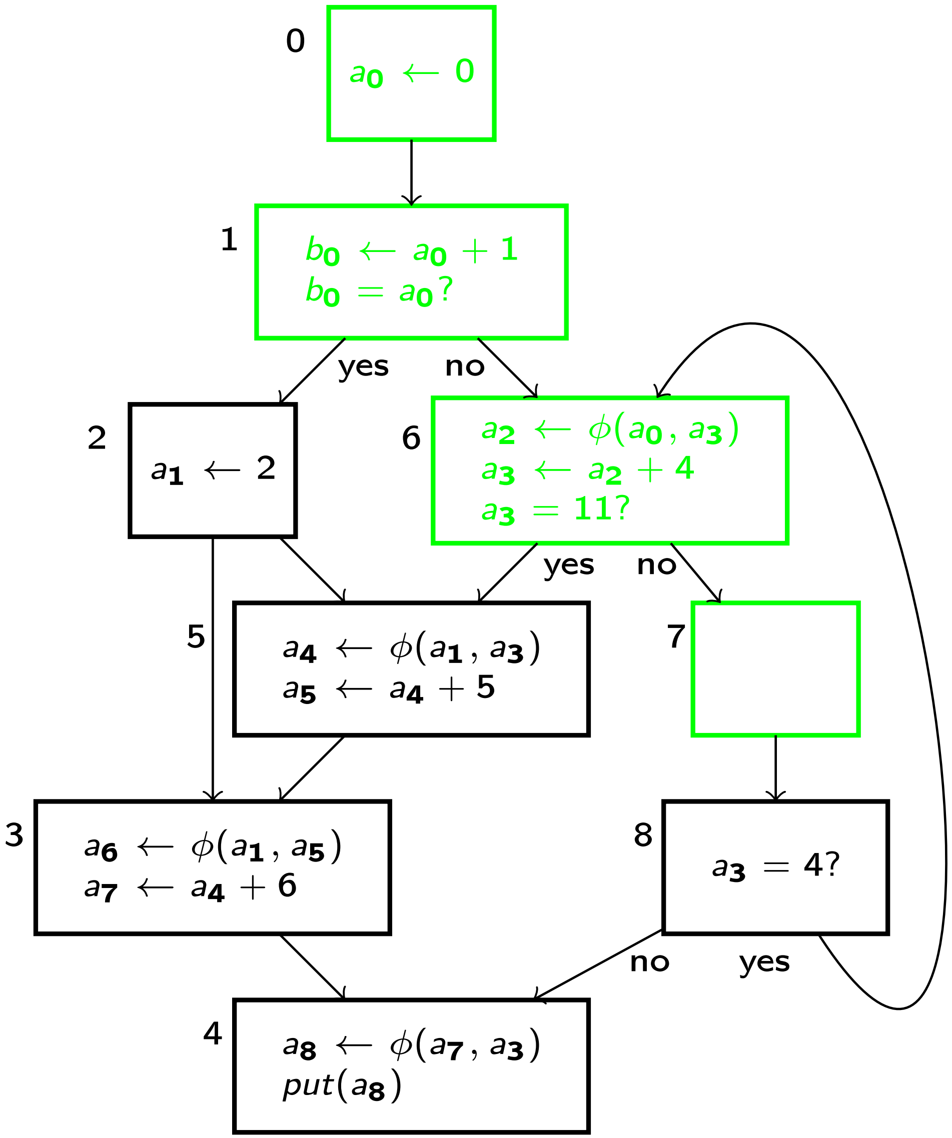



- $edge\_worklist = \{(1,6)\}$



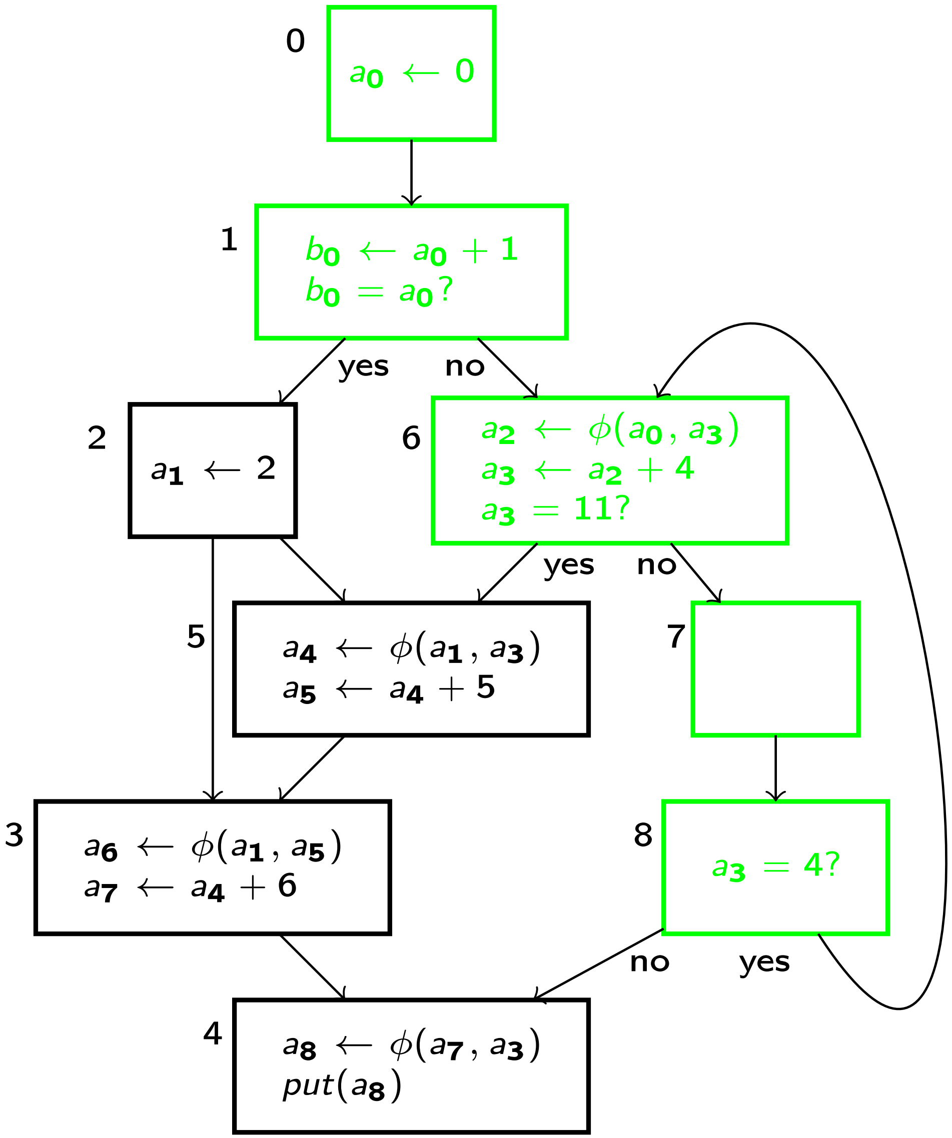

- Ignore $a_3$ in $\phi$-function in vertex 6.
- $edge\_worklist = \{(6, 7)\}$

# An Example 5(10): Visit 7



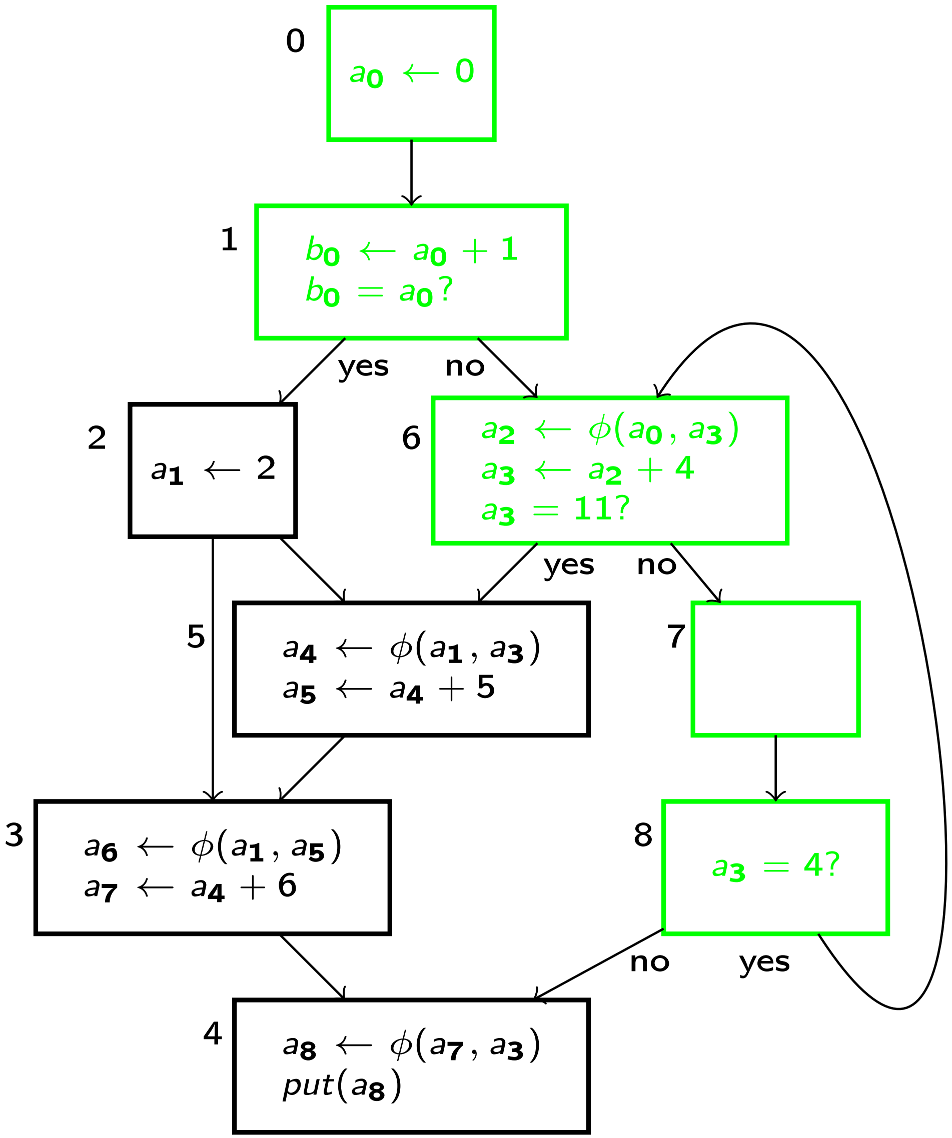

- Ignore $a_3$ in $\phi$-function in vertex 6.
- $edge\_worklist = \{(7, 8)\}$

# An Example 6(10): Visit 8



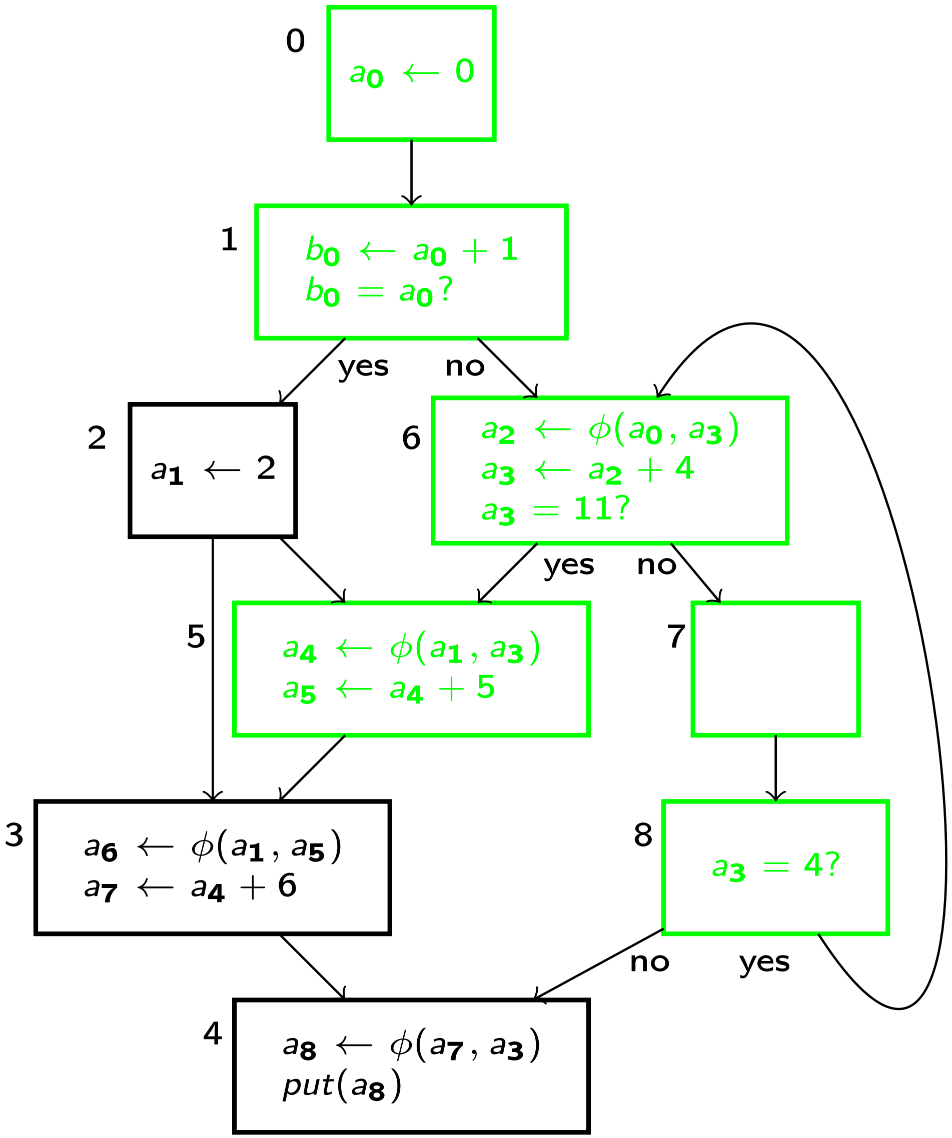

- $edge\_worklist = \{(8, 6)\}$

# An Example 7(10): Revisit 6



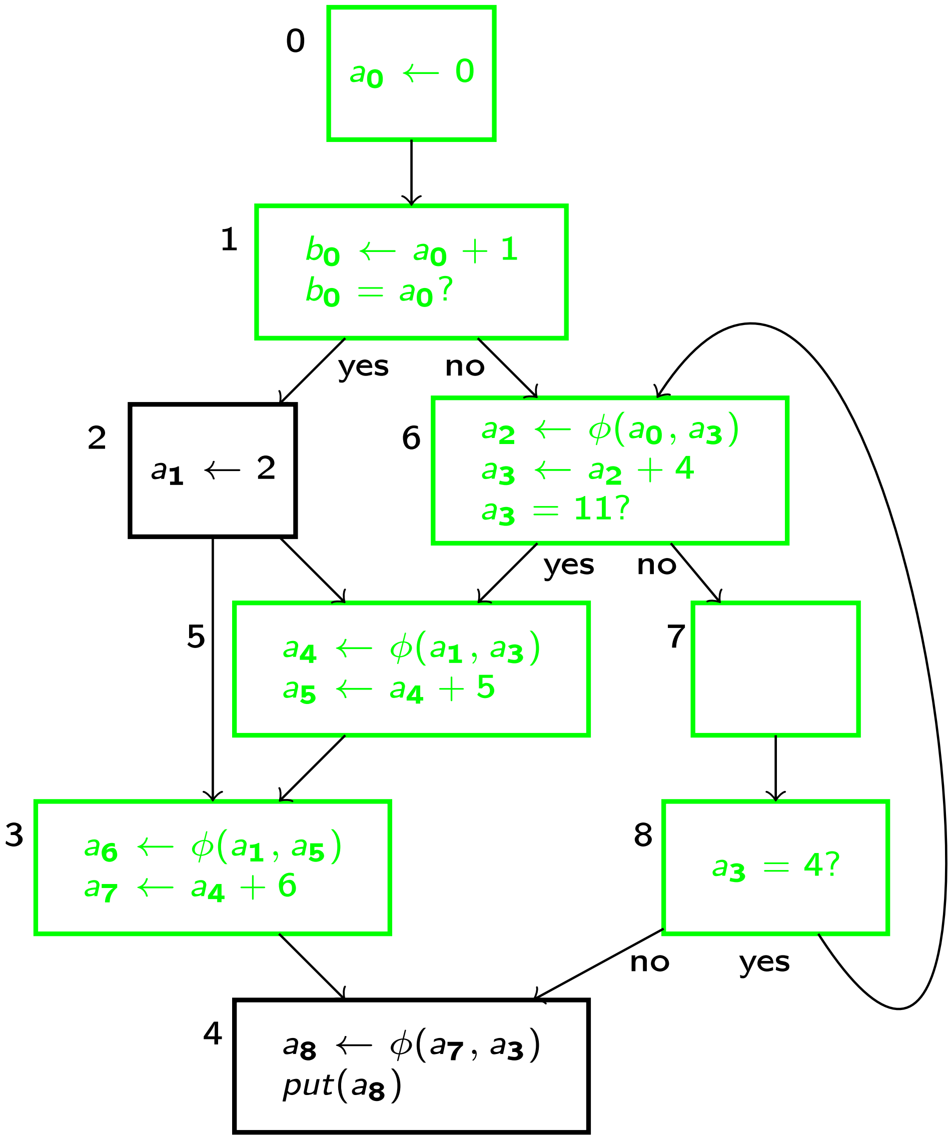

- Now only the $\phi$-function is re-evaluated at first.
- This time $a_2$ is classified as a nonconstant.
- Then use of $a_2$ is put in the ssa-worklist.
- Then use of $a_3$ in the branch is put in the ssa-worklist.
- Since $a_3$ is nonconstant also $(6, 5)$ will be interpreted.
- $edge\_worklist = \{(6, 5)\}$

# An Example 8(10): Visit 5



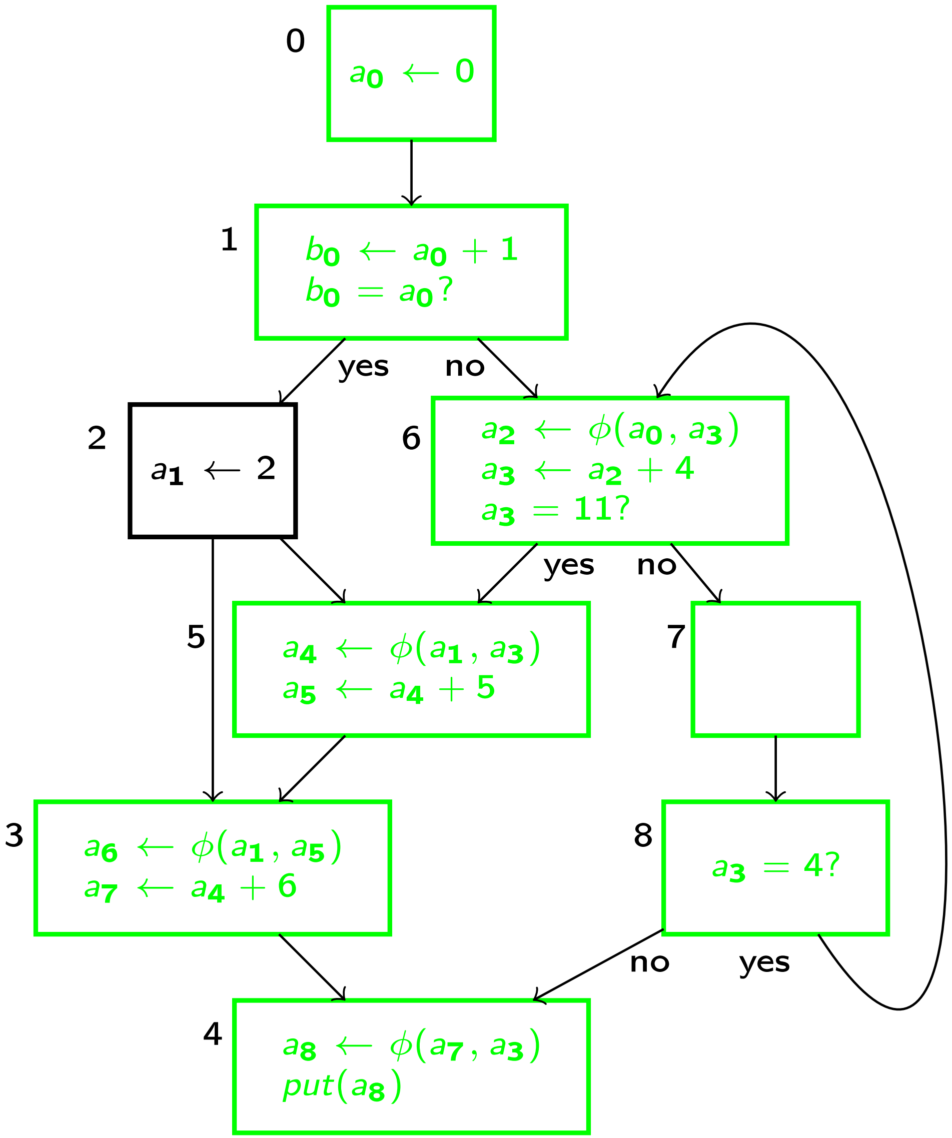

- Now $a_1$ is ignored but $a_3$ is nonconstant.
- $a_4$ and $a_5$ become nonconstant as well.
- $edge\_worklist = \{(5, 3)\}$

# An Example 9(10): Visit 3




- Again $a_1$ is ignored but $a_5$ is nonconstant.
- $a_6$ and $a_7$ become nonconstant as well.
- $edge\_worklist = \{(3, 4)\}$

# An Example 10(10): Visit 4




- $a_8$ will be read from memory.
- Vertex 2 and the branch to it can be deleted.
- In this example, for simplicity, we have not included the contents of the ssa-worklist.

# Simple Extensions 1

```
if (a != 44)
        b = a + 1;
else {
        b = a + 2;
        f(b);
}
```

- The parameter must have the value 46.
- By inserting `a = 44` in the else-clause, the constant propagation algorithm is helped.

# Simple Extensions 2

```
if (x != y) {
        a = 1;
        b = 2;
} else {
        a = 2;
        b = 1;
}

c = a + b;
```

- Clearly the sum is 3 but the present algorithm cannot find this.
- It's a rather trivial extension to "enhance" the algorithm to cover such codes as well.
- Is it worth it? No, only in very rare codes is it beneficial while all compilations would be somewhat slower.
- But see next slide!

# A Remark About Rarely Used Optimizations

- And a more important point than making the compiler slightly slower: **never** include optimizations in a compiler which are rarely useful because then they are much more likely to contain obscure bugs than if they are used millions of times every day!
- There was a famous bug in a Bell Labs FORTRAN compiler which was an "optimization" which had never been useful for years.
- Once it was but it resulted in incorrect code and a lot of confusion for the programmer!
- It is said to have costed the compiler writer several days to implement for no use and then additional application debugging time!