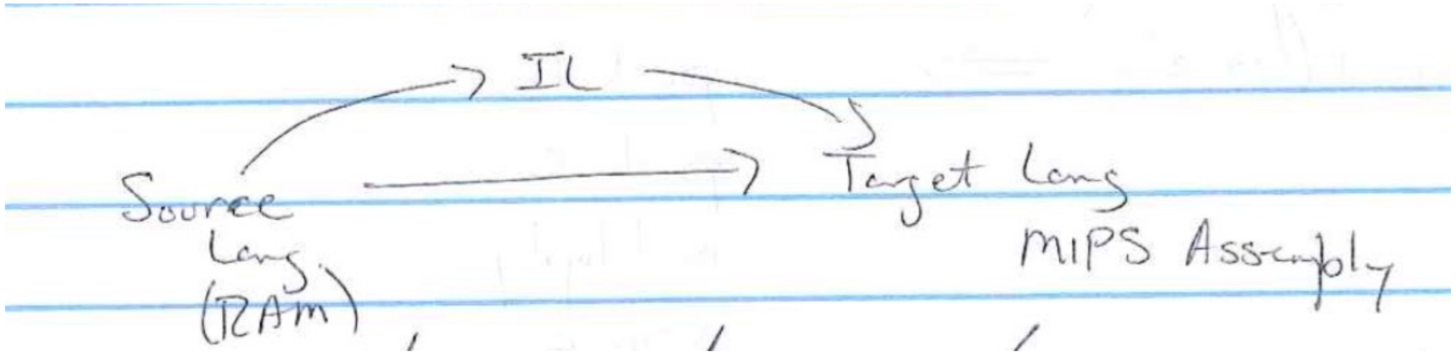
CSC 416/565: DESIGN AND CONSTRUCTION OF COMPILERS

West Chester University
Dr. Richard Burns
Fall 2023

Intermediate Code/Language



Stepping back a bit from *code generation.*

What is an IL?

- A language between the *source* and *target*
- Provides an intermediate level of abstraction

1. More details than the *source*
    - MiniJava/Ram23 has no notion of registers
    - Optimization algorithms may depend on registers
2. Fewer details than the *target*
    - IL doesn't have all the details of a *target* machine
    - Easier to re-translate IL to some other *target*

Compiler design my include a sequence of IL's.

IL as ”High-level” Assembly

Very similar to the assembly code from the last few weeks.

Some notable differences:

- Unlimited number of registers
- IL opcodes corresponding directly to assembly opcodes
  - More pseudoinstructions, like `push`

Example

Operands of IL instructions are registers or constants.

Cannot evaluate expression `a + (b * c)` directly.

Translated to IL:

```
t1 := b * c
t2 := a + t1
```

```
t1 := b * c
t2 := a + t1
```

- <u>Three-address code IL:</u> every subexpression is given a name

- In Proj #5 code generation assignment, rather than use temporaries/registers, we will *exploit the stack.*

```
a + (b * c)
```

```
push b
push c
multiply
push a
add
```

Demo Project #5

Local Optimization

Compiler layout:

Lexical Analysis ⇒ Parsing ⇒ Semantic Analysis ⇒ Optimization ⇒ Code Generation

- In modern compilers, optimization phase is that largest and most complex.

Going to <u>optimize</u> after translation into IL:

- Exposes more optimization opportunities
  - (we now have <u>temporaries</u>, not as high-level as AST)

- *Q:* Why not just optimize at the assembly language level?
  - While this also exposes optimization opportunities, this is machine dependent.
  - Would have to re-implement optimizations when re-targeting to a different machine.

Optimization Seeks to Improve

1. Execution time (most often)
2. Code size
3. Can also imagine optimizations for:
    - Network messages sent
    - Power
    - Disk lookups
    - etc.

Key Rule:

Optimization should not alter what program computes!

Types of Optimizations

1. Local optimizations: basic block level
2. “Global” optimizations: single procedure level using control-flow graphs
3. Inter-procedural optimizations across method boundaries

Most compilers do many *local optimizations* and a few *global optimizations.*

Some optimizations are not implemented by a compiler.

Why not?

1. If they're hard to implement
2. Costly in compilation time
3. Have a low payoff
4. etc.

Basic Block

Sequence (grouping) of three-address instructions

Properties of basic blocks:

- Maximal sequence of instructions with
  - no labels (except at first instruction)
  - no jumps (except in last instruction)
- Always enter a basic block at the beginning and exit at the end.

Big Idea: execution of a basic block is completely predictable

- guaranteed to flow from the first statement in the block to the last statement
- (furthermore, no way to jump into the middle of the block)

Example: Local Optimization with a Basic Block

```
1    Label1:
2        x := 2 * y
3        w := x + y
4        if w > 0  j Label2
```

Prompt: is there a potential optimization?

- Change line 3 to `w := 3 * y`
- Correct optimization.

Prompt: Can line 2 be eliminated?

- Depends if `x` has any other uses outside of this basic block.

Other Types of Local Optimizations

(without needing to analyze the entire procedure body)

Some statements can be deleted:

```
x := x + 0
```

Any others?

```
x := x * 1
```

Some statements can be simplified:

```
x := x + 0    →    x := 0
```

But only is an optimization if instruction on the right runs faster…

- Depends on the machine.
- Right instruction doesn't have `ADD` nor second memory access.
- However, still worthwhile to do on *all* machines, because assigning an identifier to a constant can open the door for other optimizations to be performed.

Replacing Exponentiation Operator

```
y := y ** 2        →        y := y * y
```

- Left instruction probably links to some built-in math library
- Will have overhead in performing this call, looping 2 times, etc.

Multiplying by a Power of 2

```
x := x * 8        →        x := x << 3
```

- Operation of *left bit shift* is faster than multiply
  - *(more so in historical machines)*

Also applicable if multiplication is not by a power of 2.

```
x := x * 15       →        t1 := x << 4
                            x := t1 - x
```

Constant-Folding class of optimizations: Computing operations on constants at compile-time (rather than run-time) if operands are literals in instruction.

Example 1:

`x := 2 + 2` → `x := 4`

Example 2: (Conditional constant)

`if 2 < 0 j Label` → *(deleted because condition is false)*

Example 3: (Conditional constant)

`if 2 > 0 j Label` → `j Label`

(conditional jump to unconditional jump)

Unreachable Code/Basic Blocks

- Eliminate basic blocks that are *not* the target of any jump or conditional.
- May happen if a predecessor to a basic block is deleted (see Conditional Constants, previous slide).
- *Effect:* will make program size smaller
  - And sometimes faster (increasing spatial locality)

What is another scenario that causes unreachable blocks?

1.
2. Libraries – reduction of size of final binary
   - Only the methods that are called are linked

Static Single Assignment Form

Some optimizations are simplified if each register only occurs once on LHS of an assignment.

- Rewrite intermediate code into static single assignment form.
- Each variable is assigned exactly once and defined before it is used.
- *Example:*

```
x := y + z    →    t := y + z
w := x        →    w := t
x := 4 * x    →    x := 4 * t
```

Can be complicated due to loops.

Static Single Assignment Form - Optimization

An optimization that depends on <u>static single assignment form.</u> Given:

1. Some basic block is in single assignment form.
2. Variable is assigned exactly once.
3. Every variable is defined before it is used.

When two assignments have the same RHS, they <u>will</u> compute the save value.

Common Subexpression Elimination:

```
                x := y + z          →   x := y + z
                         …          →   …
// SSA: values of x,y,z will not    →   …
// change in between these lines    →   …
                         …          →   …
                  w := y + z        →   w := x
```

Copy Propagation:

- "Propagating copies through the code"
- *Given:* static single assignment form
- Useful for enabling other optimizations
  - (constant folding, dead code elimination)
  - **Dead Code Elimination:** statement that does not contribute to the program's result
    - Statement is <u>dead</u> and can be eliminated
- *Example:*

```
t := y + z     →     t := y + z
w := t         →     w := t
x := 4 * w     →     x := 4 * t
```

- Initially, no improvement in the code.
- Perhaps `w := t` can be deleted?

Conclusion

- Each local optimization does little by itself.
- But, typically optimizations interact.
  - Performing one optimization may enable another
- Optimizing compilers repeat optimizations until no improvement is possible...
- ...or compiler stops optimization to limit compilation time.

Example 1

```
a := 5
x := 2 * a
y := x + 6
t := x * y
```

Example 2

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

Example 3

(given that only g and x are referenced outside of this basic block)

```
a := 1
b := 3
c := a + x
d := a * 3
e := b * 3
f := a + b
g := e - f
```

After the Break

Global Optimization

Global Optimization

Topics:

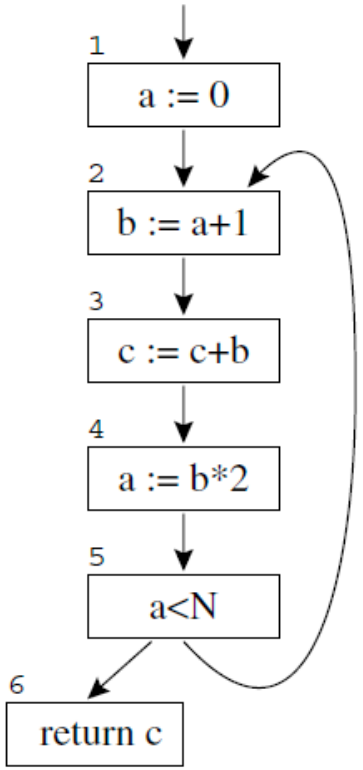
- Data flow analysis
- Control flow graph

Control Flow Graphs

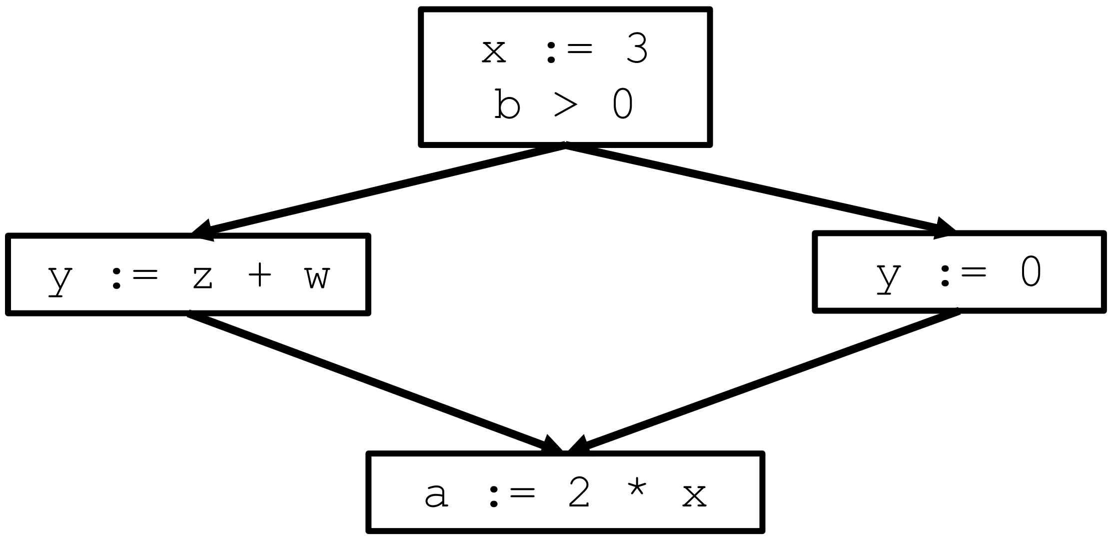
One Definition:

- Each statement in the program is a node in the graph
- If statement x can be followed by statement y, then there is an edge
  x → y

$$
\begin{aligned}
&a \leftarrow 0 \\
L_1 : \; &b \leftarrow a + 1 \\
&c \leftarrow c + b \\
&a \leftarrow b * 2 \\
&\text{if } a < N \text{ goto } L_1 \\
&\text{return } c
\end{aligned}
$$

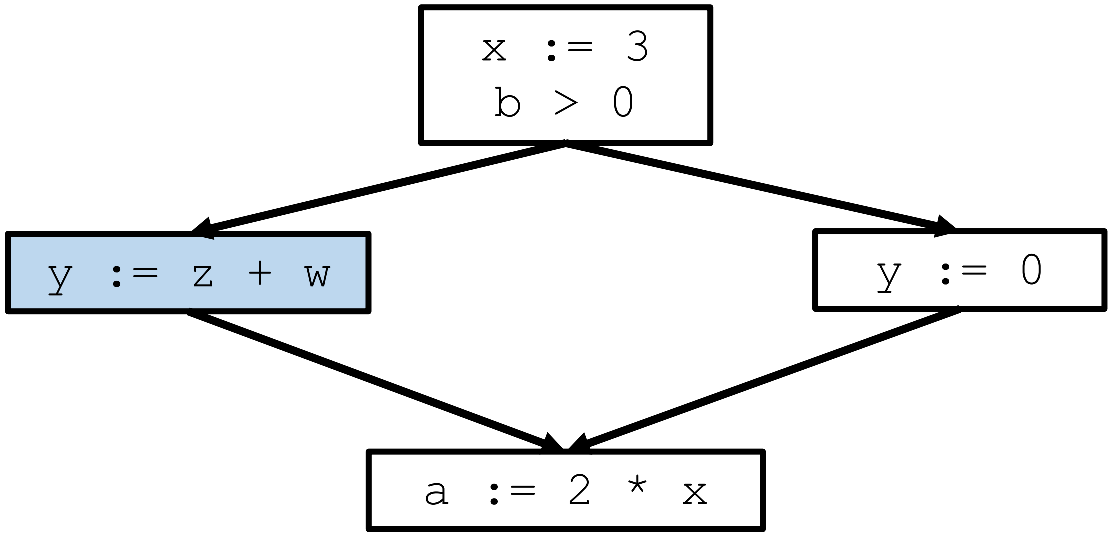


Control Flow Graphs



Another Definition:

- Each node in a control flow graph is a basic block
- *Prompt:* What is the program that this control flow graph represents?



- Notice that the *constant propagation* optimization could be performed on this graph – and it would be ok!
- What if the blue node was block `y := z + w ; x := 4` ?
  - Would the constant propagation optimization still be safe?

- *Big Question:* **When is it ok to *globally* propagate constants?**

Criteria Needed for *Global* Constant Propagation

Recall a use:

- statement i assigns a value to variable x
- statement j uses x as an operand
- if there is a path from statement i to j and x is not reassigned along the way, then statement j uses the value of x computed at statement i
- x is live at statement i

Can only replace a use of x by a constant n, when the last assignment to x is x := n

- Checking every path of all paths in a control flow graph is not trivial… must consider loops and conditionals.
- Data flow analysis analyses entire control flow graph to find/compute all program points where this criteria holds.

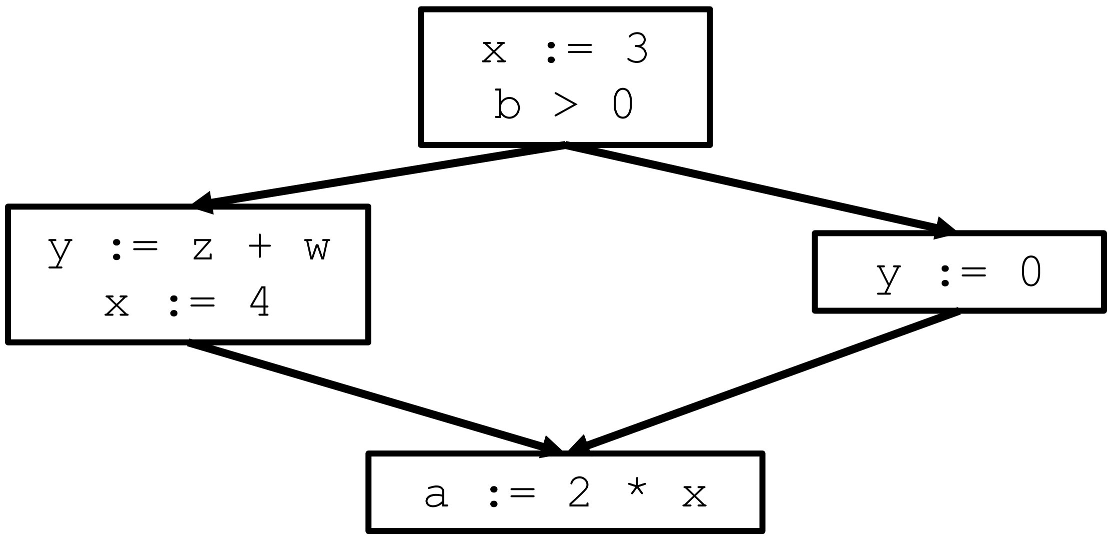
Data Flow Analysis

Values

At every program point, associate that x is either:

1. Not a constant, or we don't know
   - "TOP"
   - x = $\top$
2. Constant
   - "CONSTANT" c
   - x = c (*some constant*)
3. The statement at the program point never executes
   - "BOTTOM"
   - x = $\perp$

Example



Given global constant information, it is easy to perform the optimization.

- Inspect value of x at *each <u>program point</u>*.
- If x is a constant at that point, we can replace use of x with that constant.
- <u>Program points</u> are *in between* statements.

Building a Systematic Algorithm

- *Big Idea:* “transfer/push” information from one statement to the next

<u>Two functions:</u>

1. Constant information
   - C(s,x,in) = value of x *before* statement s
   - C(s,x,out) = value of x *after* statement s
2. Transfer function
   - Will transfer information from one statement to another

Statement s has some set of immediate predecessor statements $p_1 \ldots p_n$

Rules 1-4: Transfer info from OUT of one statement to IN of the *next* statement.

Transfer Function, Rule 1:

$$\text{if } C(p_i, x, out) = \top \text{ for any } i, \text{then } C(s, x, in) = \top$$

Rule 2:

$$\text{if } C(p_i, x, out) = m \;\&\; C\left(p_j, x, out\right) = n \;\&\; m \,!= n, \text{then } C(s, x, in) = \top$$

Rule 3:

$$\text{if } C(p_i, x, out) = c \text{ or } \perp \text{ for all } i, \text{then } C(s, x, in) = c$$

Rule 4:

$$\text{if } C(p_i, x, out) = \perp \text{ for all } i, \text{then } C(s, x, in) = \perp$$

Statement s has some set of immediate predecessor statements $p_1 \ldots p_n$

Rules 5-8: Transfer info from IN of one statement to OUT of the *same* statement.

Rule 5:

$$\text{if } C(s,x,in) = \bot, \text{then } C(s,x,out) = \bot$$

Rule 6:

$$\text{if } s \text{ is an assignement statement } x \coloneqq n \;\&\; C(s,x,in) \neq \bot \text{ (so } s \text{ can be reached),}$$
$$\text{then } C(x \coloneqq n, x, out) = n$$

Rule 7:

$$\text{if } s \text{ (RHS of } s\text{) is more complicated than assignment,}$$
$$\text{then } C(s,x,out) = \top$$

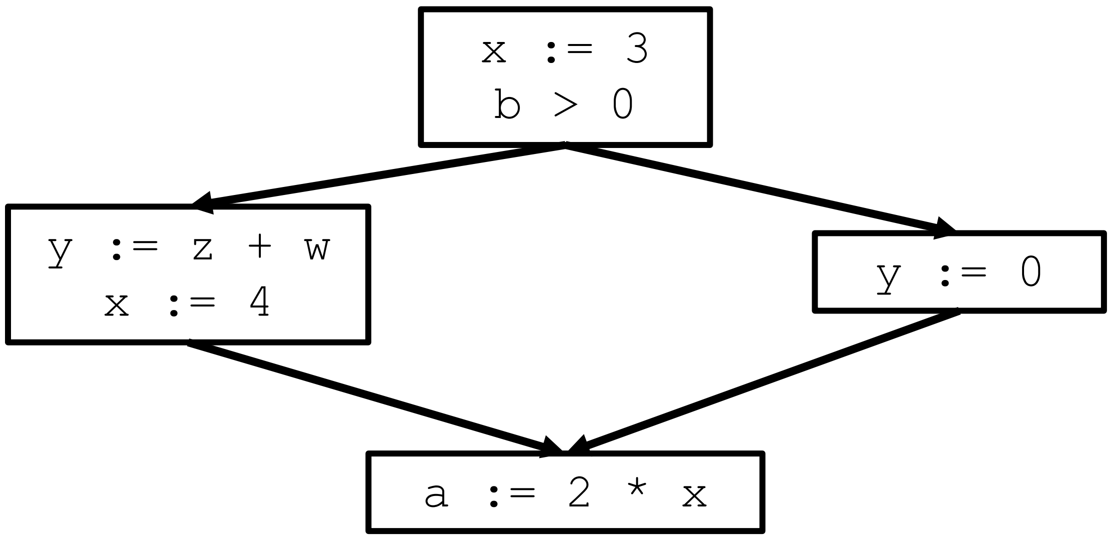
Rule 8:

$$C(y \coloneqq \ \ldots, x, out) = \ C(y \coloneqq \ \ldots, x, in), \text{if } x \,!= y$$

Algorithm

1. For every entry s into the program, set $C(s,x,in) = \top$
2. Everywhere else, set $C(s,x,in) = C(s,x,out) = \bot$
3. Repeat until all points satisfy one of the 8 rules

Example 1



Ideas:

- Look where the information is inconsistent and update it.
- At the beginning, information is consistent everywhere except at the first statement.

Example 2

Analysis of Loops

The need for $\perp$ is because of loops.

Example 3

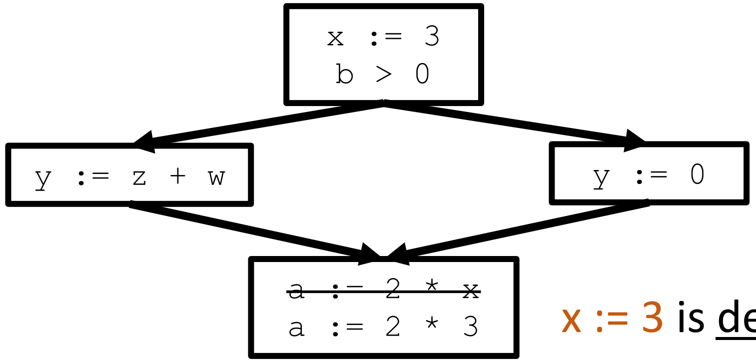
Can prove that constant propagation algorithm is linear to program size.

Liveness Analysis

(Another Global Analysis)

Liveness Analysis

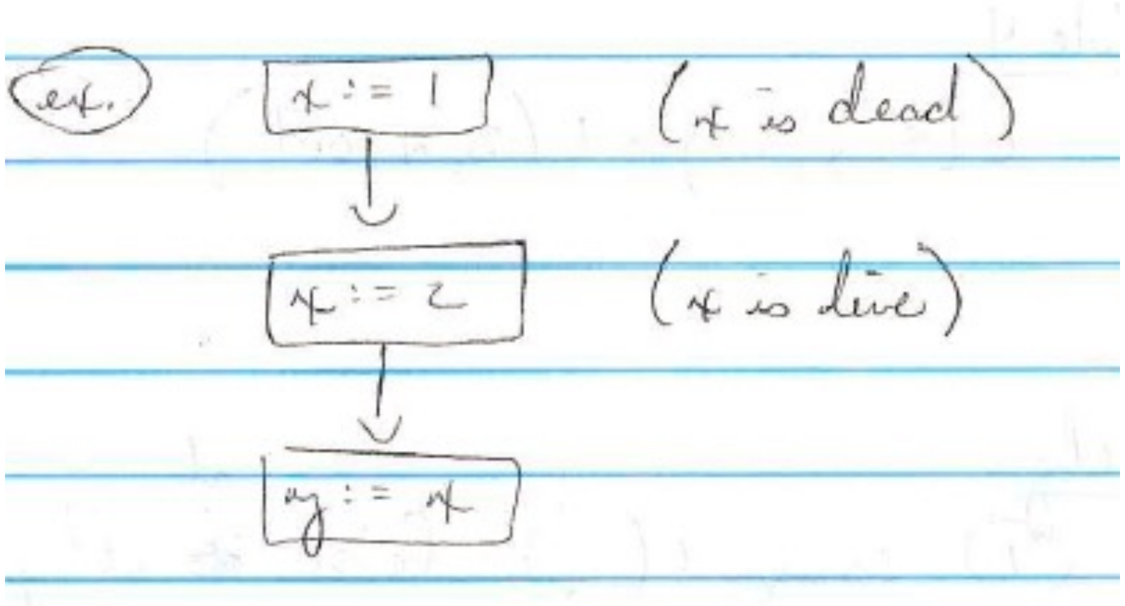
- *Motivation:* once constants have been globally propagated, we would like to eliminate dead code



x := 3 is dead (assuming x is not used elsewhere)

- Liveness: a variable is live on an edge if there is a path from an edge to the use of that variable, that does not go through any reassignment.

- Dead statements can be deleted from the program
- Need liveness information
- Similar to constant propagation algorithm:
  - We will transfer information between adjacent program points



- Rather than a constant information function, we need a liveness information function.
- Will return a Boolean (liveness is simpler):
  - L(s,x,out)
  - L(s,x,in)

Statement s has some set of immediate successor statements $t_1 \ldots t_n$

Rule 1:

$$L(s, x, out) = \bigcup L(t_i, x, in), \text{where } t_i \text{ is a successor of } s$$

Rule 2:

$$L(s, x, in) = true, \text{if } s \text{ refers to } x \text{ on RHS}$$

Rule 3:

$$L(s_{x:=e}, x, in) = false, \text{if } e \text{ does not refer to } x$$

Rule 4:

$$L(s, x, in) = L(s, x, out)$$

$$\text{(statements that do not refer to } x\text{)}$$

Algorithm

1. Assign L(…)=false, where … is some variable, at all program points
2. Repeat constraint satisfaction alg until all statements s satisfy rules 1-4 (there are no inconsistencies)
   - Pick an s not satisfying 1-4 and update using the appropriate rule.

Example

- Loop that counts to 10 and exits.

Observe:

- Values can change from false to true, but not the other way around.
- Each value can only change once.
  - Termination guaranteed.

Conclusion

- Two kinds of global analysis:
  1. Constant propagation
  2. Liveness
- Constant propagation is a *forward* analysis.
- Liveness is *backwards.*
- There are other types of global flow analysis.
- Most can be classified as either forward or backward.

Example 2