## CSE 501: Compiler Construction

Topic: implementing languages efficiently on stock hardware
- analysis & optimizations
- imperative, functional, and object-oriented languages

Prerequisites:
- CSE 401 or equivalent
- CSE 505 or equivalent

Read Appel's "Modern Compiler Implementation"
+ ~20 papers from literature
- "Compilers: Principles, Techniques, & Tools", a.k.a. the Dragon Book, as a reference

Periodic homework assignments + course project

Midterm + final



## Course outline

Basic compilation techniques:
- intermediate representations, analysis frameworks, optimizations

Fancier techniques:
- lattice-theoretic analysis frameworks & tools
- better intermediate representations

Analyzing and optimizing procedure calls:
- inlining
- interprocedural analysis
  - with first-class functions
  - with dynamically dispatched messages
- procedure specialization & partial evaluation

Analyzing and optimizing use of pointers

Analyzing and optimizing arrays in loops

Register allocation

Instruction scheduling

Garbage collection algorithms

Compiling functional & object-oriented languages



## Goals for language implementation

Correctness

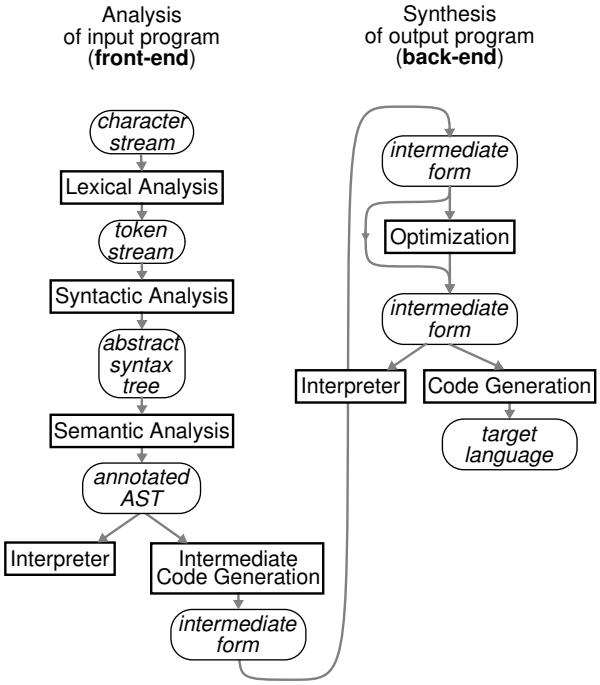
Efficiency
- of: time, data space, code space
- at: compile-time, run-time

Support expressive language features
- first-class, higher-order functions
- dynamic dispatching
- exceptions, continuations
- ...

Support desirable programming environment features
- fast turnaround
- separate compilation, shared libraries
- source-level debugging
- profiling
- garbage collection
- ...



## Standard compiler organization



## Key questions

How are programs represented in the compiler?

How are analyses organized/structured?
What analysis algorithms are used?
Over what region of the program are analyses performed?
When are optimizations performed?

What kinds of optimizations can be performed?
Which are profitable in practice?
How should analyses/optimizations be sequenced/combined?

What optimizations can be performed to handle:

- pointers, arrays
- parallel target machines
- first-class functions
- inheritance & message passing

Other issues:

- speeding compilation
- making compilers portable, table-driven
- supporting tools like debuggers, profilers, garbage collect'rs



## Overview of optimizations

First **analyze** program to learn things about it
Then **transform** the program based on info
Repeat...

Requirement: don't change the semantics!

- transform input program into
  semantically equivalent but better output program

Analysis determines when transformations are:

- legal
- profitable

Caveat: "optimize" a misnomer

- almost never optimal
- sometimes slow down some programs on some inputs
  (although hope to speed up most programs on most inputs)



## Semantics

Exactly what are the semantics that are to be preserved?
Subtleties:

- evaluation order
- arithmetic properties like associativity, commutativity
- behavior in "error" cases

Some languages very precise

- programmers always know what they're getting

Others weaker

- allow better performance (but how much?)

Semantics selected by compiler option?



## Scope of optimizations

**Peephole**: across a small number of "adjacent" instructions
[adjacent in space or time]

- trivial analysis

**Local**: within a **basic block**

- simple analysis

**Intraprocedural** (a.k.a. **global**):
across basic blocks, within a procedure

- analysis more complex:
  branches, merges, loops

**Interprocedural**:
across procedures, within a whole program

- analysis even more complex:
  calls, returns
- sometimes useful
  - more useful for higher-level languages
- hard with separate compilation

## Catalog of optimizations/transformations

arithmetic simplifications

- constant folding

```
x := 3 + 4   ⇒   x := 7
```

- strength reduction

```
x := y * 4   ⇒   x := y << 2
```

constant propagation

```
x := 5       ⇒   x := 5       ⇒   x := 5
y := x + 2       y := 5 + 2       y := 7
```

copy propagation

```
x := y       ⇒   x := y
w := w + x       w := w + y
```

common subexpression elimination (CSE)

```
x := a + b   ⇒   x := a + b
...              ...
y := a + b       y := x
```

partial redundancy elimination (PRE)

- CSE, but with earlier expression only available along subset of possible paths



integer range analysis

```
index := 5
~~if index >= 10 goto _error~~
a[index] := 0
```

pointer/alias analysis

```
p  := &x     ⇒   p  := &x      ⇒   p  := &x
*p := 5          *p := 5           *p := 5
y  := x + 1      y  := 5 + 1       y  := 6
```

dead (unused) assignment elimination

```
~~x := 5~~
... // no use of x
x := 6
```

dead (unreachable) code elimination

```
  ~~if false goto _else~~
  ...
  goto _done
~~_else:~~
  ~~...~~
_done:
```



## Loop optimizations

loop-invariant code motion (a.k.a. code motion, code hoisting)

```
for j := 1 to 10      ⇒  for j := 1 to 10
  for i := 1 to 10           t := b[j]
    a[i] := a[i] + b[j]      for i := 1 to 10
                               a[i] := a[i] + t
```

induction variable elimination

```
for i := 1 to 10   ⇒  for p := &a[1] to &a[10]
  a[i] := a[i] + 1       *p := *p + 1
```

- `a[i]` is several instructions, `*p` is one

loop unrolling

```
for i := 1 to N      ⇒  for i := 1 to N by 4
  a[i] := a[i] + 1         a[i]   := a[i]   + 1
                           a[i+1] := a[i+1] + 1
                           a[i+2] := a[i+2] + 1
                           a[i+3] := a[i+3] + 1
```



parallelization

```
for i := 1 to 1000   ⇒  forall i := 1 to 1000
  a[i] := a[i] + 1         a[i] := a[i] + 1
```

loop interchange, skewing, reversal, ...

blocking

- restructuring loops for better data cache locality

## Call optimizations

inlining

```
l := ...          ⇒  l := ...      ⇒  l := ...
w := 4               w := 4           w := 4
a := area(l,w)       a := l * w       a := l << 2
```

interprocedural constant propagation, alias analysis, etc.

procedure specialization

- more generally, partial evaluation



## Machine-dependent optimizations

register allocation

instruction selection

```
p1 := p + 4     ⇒    ld %g3, [%g1 + 4]
x := *p1
```

instruction scheduling

```
ld  %g2, [%g1 + 0]  ⇒    ld  %g2, [%g1 + 0]
add %g3, %g2, 1          ld  %g5, [%g1 + 4]
ld  %g2, [%g1 + 4]       add %g3, %g2, 1
add %g4, %g2, 1          add %g4, %g5, 1
```



## The phase ordering problem

Typically, want to perform a number of optimizations;
in what order should the transformations be performed?

some optimizations create opportunities for other optimizations
⇒ order optimizations using this dependence

- some optimizations simplified
  if can assume another opt will run later & “clean up”

but what about cyclic dependencies?

- constant folding ⇔ constant propagation

what about adverse interactions?

- common subexpression elimination ⇔ register allocation
- register allocation ⇔ instruction scheduling



## Engineering

Building a compiler is an engineering activity

- balance
  complexity of implementation,
  speed-up of “typical” programs,
  compilation speed,
  ...

Near infinite number of special cases for optimization
can be identified

- can’t implement them all

Good compiler design, like good language design, seeks
small set of powerful, general analyses and transformations,
to minimize implementation complexity while
maximizing effectiveness

- reality isn’t always this pure...

## Representation of programs

Primary goals:

- analysis is easy & effective
  - just a few cases to handle
  - provide support for linking things of interest
- transformations are easy
- general, across input languages & target machines

Additional goals:

- compact in memory
- easy to translate to and from
- tracks info for source-level debugging, profiling, etc.
- extensible (new optimizations, targets, language features)
- displayable



## High-level syntax-based representation

Represent source-level control structures & expressions directly

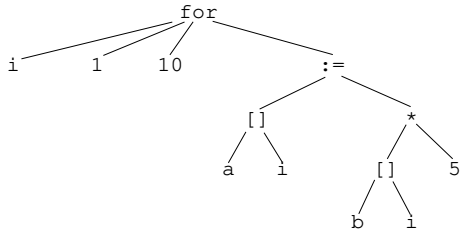
Examples

- (Attributed) AST
- Lisp S-expressions

Source:

```
for i := 1 to 10 do
  a[i] := b[i] * 5;
end
```

AST:






## Low-level representation

Translate input programs into low-level primitive chunks, often close to the target machine

Examples

- assembly code, virtual machine code (e.g. stack machine)
- three address code, register transfer language (RTLs)
- lambda calculus

Standard RTL operators:

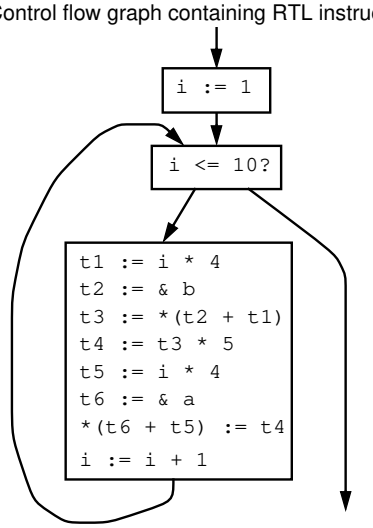
| | |
|---|---|
| assignment | `x := y;` |
| unary op | `x := op y;` |
| binary op | `x := y op z;` |
| address-of | `p := &y;` |
| load | `x := *(p + o);` |
| store | `*(p + o) := x;` |
| call | `x := f(...);` |
| unary compare | `op x ?` |
| binary compare | `x op y ?` |



Source:

```
for i := 1 to 10 do
  a[i] := b[i] * 5;
end
```

Control flow graph containing RTL instructions:



## Comparison

Advantages of high-level rep:
- analysis can exploit high-level knowledge of constructs
- easy to map to source code terms for debugging, profiling
- may be more compact
- supports semantics-based reasoning about correctness etc. of analysis

Advantages of low-level rep:
- can do low-level, machine-specific optimizations (if target-based representation)
  - high-level rep may not be able to express some transformations
- supports good analysis (?), transformations
- can have relatively few kinds of instructions to analyze
- can be language-independent
  - can be used e.g. at link-time

High-level rep suitable for a source-to-source optimizer

Can mix multiple representations in single compiler, can sequence compilers using different reps



## Components of representation

Operations

Control flow: sequencing of operations
- evaluation of then & else arms depends on result of test
- side-effects of statements occur in right order

Data flow: flow of values from **definitions** to **uses**
- operands computed before operation

Also called control & data **dependence**

Ideal: represent just those dependences that matter
- dependences constrain transformations
- fewest dependences ⇒ most flexibility in implementation



## Representing control dependences

Option 1: high-level representation
- control flow implicit in semantics of AST nodes

Option 2: control flow graph
- nodes are basic blocks
  - instructions in basic block sequence side-effects
- edges represent branches (control flow between basic blocks)

Some fancier options:
- control dependence graph, part of **program dependence graph** (PDG) [Ferrante *et al.* 87]
- convert into data dependences on explicit state, in **value dependence graph** (VDG) [Weise *et al. 94*]



## Kinds of data dependences

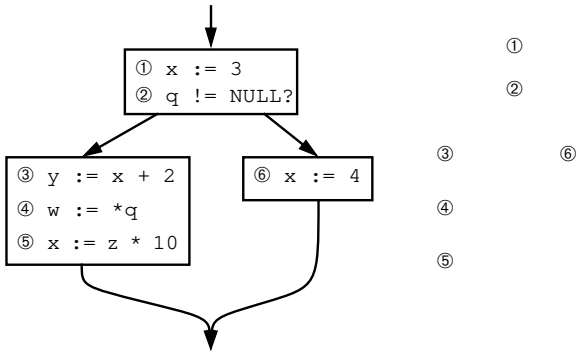
read-after-write (RAW): **true dependence**, **flow dependence**
- reflects real data flow, operands to operation

write-after-read (WAR): **anti-dependence**
write-after-write (WAW): **output dependence**
- reflects reuse of memory, not real data flow ⇒ can sometimes be eliminated by optimization

read-after-read (RAR): no dependence
- can occur in any order

## Example

```
x := 3
if q != NULL then
  y := x + 2
  w := *q
  x := z * 10
else
  x := 4
endif
```






## Representing data dependences

Within basic block:

Option 1: sequence of instructions
(represent data flow as a kind of control dependence)
- \+ simple, source-like
- \+ fixed ordering supports easy analysis
- – may overconstrain order of operations

Option 2: expression tree/DAG
- \+ natural, abstract
- \+ directly captures data dependences within basic block
- \+ DAG supports local CSE
- \+ can be compact
- – conceptually harder to analyze, transform
- – must linearize eventually



## Example

Source:

```
x := (z/y) + (y*4)
y := x + (y*4)
z := z + (y*4)
```

Linear RTL:

```
t1 := z/y
t2 := y*4
x := t1 + t2

t3 := y*4
y := x + t3

t4 := y*4
z := z + t4
```



## Representing data dependences, cont.

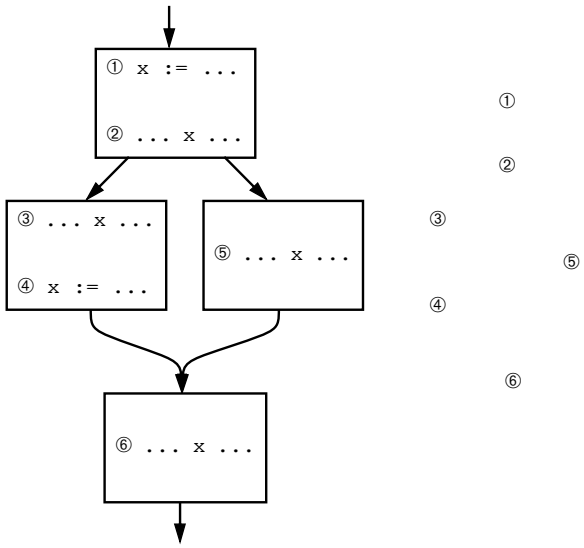
Across basic blocks:

Option 1: implicitly through variable defs/uses
- \+ simple
- – analysis wants important things explicit

Option 2: def/use chains, linking each def with each use
- \+ explicit
- – must be computed, maintained after transformations
- – space consuming

Fancier options:
- **static single assignment** (SSA) form [Alpern *et al.* 88]
- value dependence graphs (VDGs)
- **dependence flow graphs** (DFGs)
- ...

## Example






## Data flow analysis

Want to compute some info about program

- at **program points**
- to determine opportunities for improving transformations

Can model data flow analysis as solving a system of constraints

- each node in CFG places a constraint relating info at predecessor and successor points
- solution to constraints is result of analysis

Solution must be **safe**/**sound**

Solution can be **conservative**

Key issues:

- how to know if constraint system defines the analysis correctly?
- how to represent info efficiently?
- how to represent & solve constraints efficiently?
  - how long does constraint solving take? finite time?
- what if multiple solutions are possible?
- how to synchronize transformations with analysis?



## Example: reaching definitions

For each program point,
want to compute set of definitions (statements) that *may reach* that point

- reach: are the last definition of some variable
- can use info to build def-use chains, do constant propagation, etc.

Info ≡ set of *var*→*rtl* bindings

E.g.:
$\{x \to s_1, y \to s_5, y \to s_8\}$

Safety rule:
can have more bindings than the "true" answer, but can't miss any



## Reaching definitions, continued

Main constraints:

A simple assignment removes any old reaching defs for the lhs and replaces them with this stmt:

- **strong update**

*s:* `x := ...`:
$\text{info}_{succ} = \text{info}_{pred} - \{x \to *\} \cup \{x \to s\}$

A pointer assignment may modify anything, but doesn't definitely replace anything

- **weak update**

*s:* `*p := ...`:
$\text{info}_{succ} = \text{info}_{pred} \cup \{* \to s\}$

Other statements: do nothing
$\text{info}_{succ} = \text{info}_{pred}$

## Reaching definitions, still continued

Branches pass through reaching defs to both successors

$info_{succ[i]} = info_{pred}$

Merges take the union of all incoming reaching defs

- we don't know which path is being taken at run-time
  ⇒ be conservative

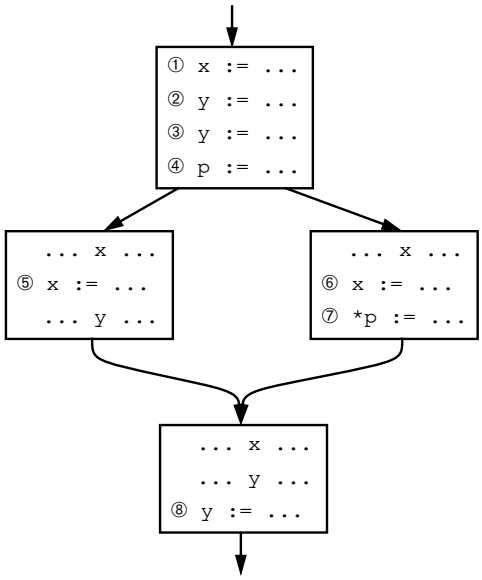
$info_{succ} = \cup_i info_{pred[i]}$

Conditions at entry to CFG: {}

- or maybe bindings for formals



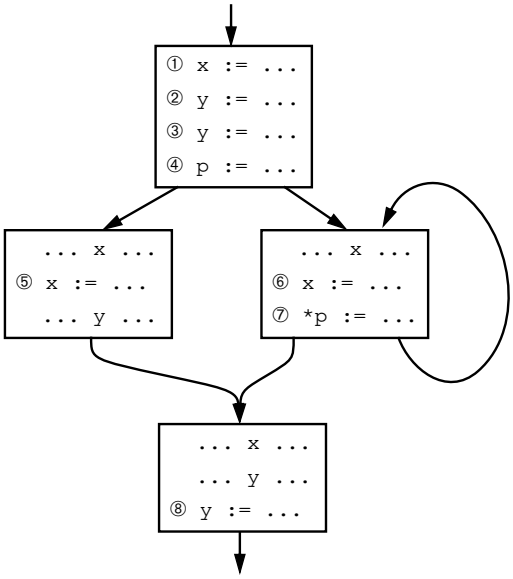
## Example




Simple constraint-solving algorithm:
traverse instructions in forward topological order,
computing successor info from predecessor info



## Another example




Topological order not defined!



## Loop terminology

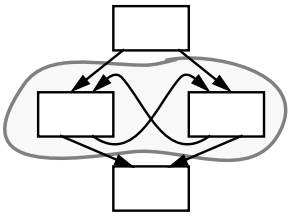
**loop**: strongly-connected component in CFG with single entry

**loop entry edge**: source not in loop, target in loop
**loop exit edge**: the reverse
**back edge**: target is loop head node

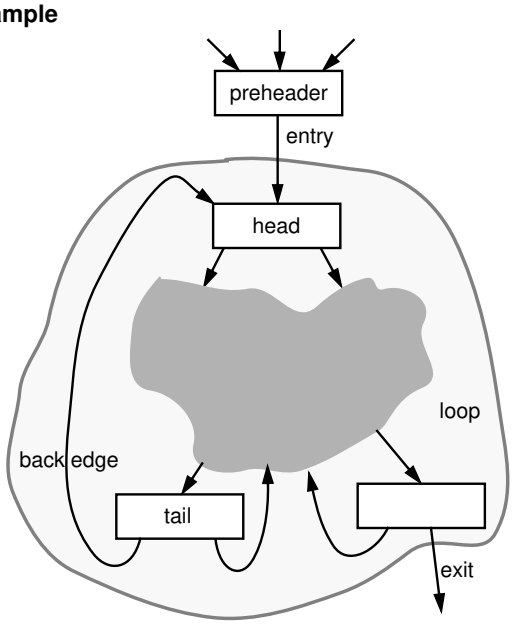
**loop head node**: target of loop entry edge
**loop tail node**: source of back edge
**loop preheader node**:
single node that's source of loop entry edge

**nested loop**: loop whose head is inside another loop

**reducible flow graph**: all SCC's have single entry



## Example






## Analysis of loops

If CFG has a loop, data flow constraints are recursively defined:

$info_{loop\text{-}head} = info_{loop\text{-}entry} \cup info_{back\text{-}edge}$

$info_{back\text{-}edge} = ... info_{loop\text{-}head} ...$

Summarizing set of constraints in loop body as *F*:

$info_{loop\text{-}head} = F(info_{loop\text{-}head})$

Legal solution to constraints is a **fixpoint** of *F*

- want **least**/**greatest** fixpoint, corresponding to most precise answer

How to find fixpoint?

- for restricted CFGs can use specialized methods
  - e.g. **interval analysis** for **reducible** CFGs
- for arbitrary CFGs, can use **iterative** approximation



## Iterative data flow analysis

1. Start with initial guess of info at end of loop:

   $info_{back\text{-}edge} = guess$

   $info_{loop\text{-}head} = info_{loop\text{-}entry} \cup info_{back\text{-}edge}$

2. Solve equations for loop:

   $info_{back\text{-}edge}' = F_{body}(info_{loop\text{-}head})$

   $info_{loop\text{-}head}' = info_{loop\text{-}entry} \cup info_{back\text{-}edge}'$

3. Test if found fixpoint:

   $info_{loop\text{-}head}' = info_{loop\text{-}head}$ ?

   A. if so, then done

   B. if not, then adopt result as (better) guess and repeat:

   $info_{back\text{-}edge}'' = F_{body}(info_{loop\text{-}head}')$

   $info_{loop\text{-}head}'' = info_{loop\text{-}entry} \cup info_{back\text{-}edge}''$

   $info_{loop\text{-}head}'' = info_{loop\text{-}head}'$ ?

   ...



## When does iterating work?

1. need to be able to make an initial guess
2. $info^{n+1}$ must be closer to the fixpoint than $info^{n}$ (constraints must be **monotonic**)
3. must eventually reach the fixpoint in a finite # of iterations (info must be drawn from a **finite-height domain**)

To reach best fixpoint, initial guess for loop head should be **optimistic**

- standard choice: $info_{loop\text{-}head} = info_{loop\text{-}entry}$

To speed iterative analysis, want to test guess ASAP

- ideal: solve constraints along shortest path from loop head to loop tail
- practical: avoid solving constraints outside of loop until fixpoint is reached within loop

## Direction of dataflow analysis

In what order are constraints solved, in general?

Constraints are declarative, not directional/procedural, so may require mixing forward & backward solving, or other more global solution methods

But often constraints can be solved by (directional) propagation & iteration

- may be **forward** or **backward** propagation of info
- topological traversals of acyclic subgraphs minimize analysis time

Constraints of simple statements called **flow functions**

- often written as functions on input info to compute output

$RD_{s:\ \texttt{x := ...}}(in) = in - \{\texttt{x}\rightarrow *\} \cup \{\texttt{x}\rightarrow s\}$

$RD_{s:\ \texttt{*p := ...}}(in) = in \cup \{*\rightarrow s\}$



## GEN and KILL sets

Can often think of flow functions in terms of each's GEN set and KILL set

- GEN = new information added
- KILL = old information removed

Then

$F_{instr}(in) = in - KILL_{instr} \cup GEN_{instr}$

E.g., for reaching defs:

$RD_{s:\ \texttt{x := ...}}(in) = in - \{\texttt{x}\rightarrow *\} \cup \{\texttt{x}\rightarrow s\}$

$RD_{s:\ \texttt{*p := ...}}(in) = in \cup \{*\rightarrow s\}$



## Bit vectors

Can sometimes represent info/KILL/GEN sets as **bit vectors**

- if can express abstractly as set of things (e.g. statements, vars), each thing getting a bit position

Bit vector operations are used for set difference & union

Flow function may be able to be represented simply by a pair of bit vectors, if they don't depend on input bit vector

- can merge the KILL and GEN bit vectors of a whole block of instructions into a single overall KILL and GEN set, for faster iterating



## Another example: constant propagation

Goal: data flow analysis that implements constant propagation

What info computed for each program point?

*I* is a conservative approximation to true info $I_{true}$ iff:

$CP_{\texttt{x := }N}$:

$CP_{\texttt{x := y + z}}$:

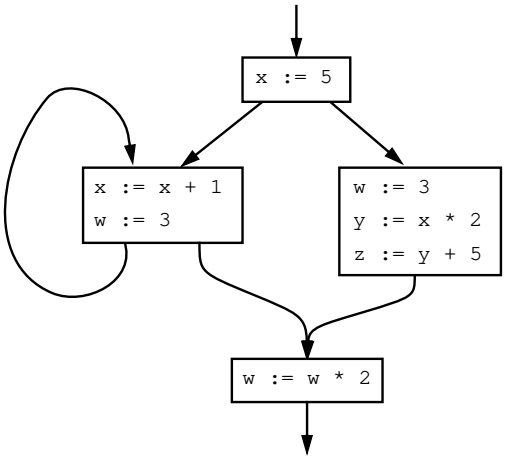
$CP_{\texttt{*p := *q + *r}}$:

Merge function?

Initial info at program points?

Direction of analysis?

## Example






## May vs. must info

Other kinds of info imply guarantees: **must** info

Some kinds of info imply possibilities: **may** info

- the complement of **may** info is **must not** info

| | May | Must |
|---|---|---|
| desired info | small set | big set |
| safe | overly big set | overly small set |
| GEN | add everything that might be true | add only if guaranteed true |
| KILL | remove only if guaranteed wrong | remove everything possibly wrong |
| MERGE | $\cup$ | $\cap$ |
| guess for back edge (optimistic) | $\varnothing$ | universal set |



## Another example: live variables

Want the set of variables that are **live** at each pt. in program

- live: might be used later in the program
- supports dead assignment elimination, register allocation

What info computed for each program point?

*I* is a conservative approximation to true info $I_{true}$ iff:

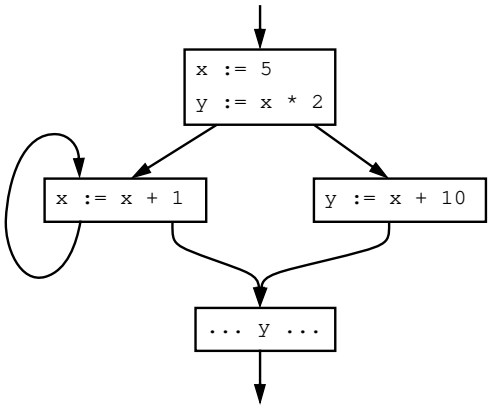
$LV_{\texttt{x := y + z}}$:

$LV_{\texttt{*p := *q + *r}}$:

Merge function?

Initial info at program points?

Direction of analysis?



## Example