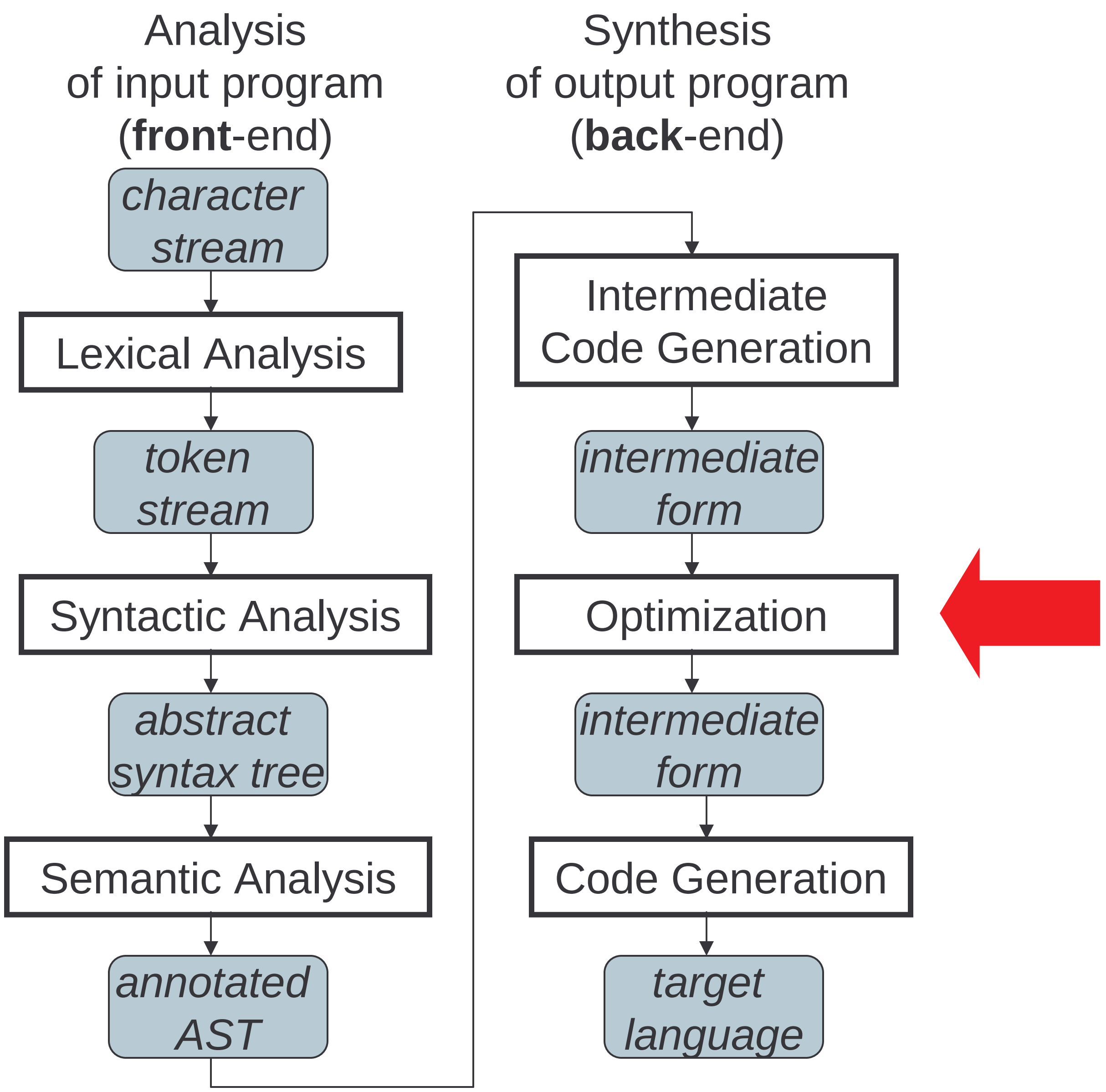
Optimization

Before and after generating machine code, devote one or more passes over the program to “improve” code quality

Compiler Passes



Optimizations

Identify inefficiencies in intermediate or target code

Replace with equivalent but better sequences

- equivalent = "has the same externally visible behavior"

Target-independent optimizations best done on IL code

Target-dependent optimizations best done on target code

“Optimize” overly optimistic

- “usually improve” is better

An example

Source code:

```
x = a[i] + b[2];
c[i] = x - 5;
```

Intermediate code (if array indexing calculations explicit):

```
t1 = *(fp + ioffset);  // i
t2 = t1 * 4;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t5 = 2;
t6 = t5 * 4;
t7 = fp + t6;
t8 = *(t7 + boffset);  // b[2]
t9 = t4 + t8; *(fp + xoffset) = t9; // x = ...
t10 = *(fp + xoffset); // x
t11 = 5;
t12 = t10 - t11;
t13 = *(fp + ioffset); // i
t14 = t13 * 4;
t15 = fp + t14;
*(t15 + coffset) = t15; // c[i] := ...
```

Kinds of optimizations

Scope of study for optimizations:

- **peephole**:
  - look at adjacent instructions
- **local**:
  - look at straight-line sequence of statements
- **global** (intraprocedural):
  - look at whole procedure
- **interprocedural**:
  - look across procedures

Larger scope => better optimization but more cost and complexity

Peephole Optimization

After target code generation, look at adjacent instructions (a “peephole” on the code stream)

– try to replace adjacent instructions with something faster

Example:

```
sw $8, 12($fp)
lw $12, 12($fp)
=>
sw $8, 12($fp)
mv $12, $8
```

More Examples

On 68K:

```
sub sp, 4, sp
mov r1, 0(sp)
  =>
mov r1, -(sp)

mov 12(fp), r1
add r1, 1, r1
mov r1, 12(fp)
  =>
inc 12(fp)
```

Do complex instruction selection through peep hole optimization

Peephole Optimization of Jumps

Eliminate jumps to jumps

Eliminate jumps after conditional branches

“Adjacent” instructions = “adjacent in control flow”

Source code: IL:

```
if (a < b) {
   if (c < d) {
      // do nothing
   } else {
      stmt1;
   }
 } else {
    stmt2;
 }
```

Algebraic Simplification

“constant folding”, “strength reduction”

```
z = 3 + 4;

z = x + 0;
z = x * 1;

z = x * 2;
z = x * 8;

z = x / 8;

double x, y, z;
z = (x + y) - y;
```

Can be done by peephole optimizer, or by code generator

Local Optimizations

Analysis and optimizations within a basic block

Basic block: straight-line sequence of statements

- no control flow into or out of middle of sequence

Better than peephole

Not too hard to implement

Machine-independent, if done on intermediate code

Local Constant Propagation

If variable assigned a constant, replace downstream uses of the variable with constant

Can enable more constant folding

Example:

```
final int count = 10;
...
x = count * 5;
y = x ^ 3;
```

Unoptimized intermediate code:

```
    t1 = 10;
    t2 = 5;
    t3 = t1 * t2;
___ x = t3;
    t4 = x;
    t5 = 3;
    t6 = exp(t4, t5);
    y = t6;
```

Local Dead Assignment Elimination

If l.h.s. of assignment never referenced again before being overwritten, then can delete assignment

Example:

```
final int count = 10;
...
x = count * 5;
y = x ^ 3;
x = 7;
```

E.g. clean-up after previous optimizations

Intermediate code after constant propagation:

```
t1 = 10;
t2 = 5;
t3 = 50;
x = 50;
t4 = 50;
t5 = 3;
t6 = 125000;
y = 125000;
x = 7;
```

Local Common Subexpression Elimination

Avoid repeating the same calculation

- CSE of repeated loads: redundant load elimination

Keep track of available expressions

Source:

```
... a[i] + b[i] ...
```

Unoptimized intermediate code:

```
t1 = *(fp + ioffset);
t2 = t1 * 4;
t3 = fp + t2;
t4 = *(t3 + aoffset);

t5 = *(fp + ioffset);
t6 = t5 * 4;
t7 = fp + t6;
t8 = *(t7 + boffset);
t9 = t4 + t8;
```

Intraprocedural (“global”) optimizations

Enlarge scope of analysis to whole procedure

- more opportunities for optimization
- have to deal with branches, merges, and loops

Can do constant propagation, common subexpression elimination, etc. at global level

Can do new things, e.g. loop optimizations

Optimizing compilers usually work at this level

Code Motion

Goal: move loop-invariant calculations out of loops

Can do at source level or at intermediate code level

Source:

```
for (i = 0; i < 10; i = i+1) {
  a[i] = a[i] + b[j];
  z = z + 10000;
}
```

Transformed source:

```
t1 = b[j];
t2 = 10000;
for (i = 0; i < 10; i = i+1) {
  a[i] = a[i] + t1;
  z = z + t2;
}
```

Code Motion at Intermediate Code Level

Source:

```
for (i = 0; i < 10; i = i+1) {
  a[i] = b[j];
}
```

Unoptimized intermediate code:

```
  *(fp + ioffset) = 0;
label top;
  t0 = *(fp + ioffset);
  iffalse (t0 < 10) goto done;
  t1 = *(fp + joffset);
  t2 = t1 * 4;
  t3 = fp + t2;
  t4 = *(t3 + boffset);
  t5 = *(fp + ioffset);
  t6 = t5 * 4;
  t7 = fp + t6; *(t7 + aoffset) = t4;
  t9 = *(fp + ioffset);
  t10 = t9 + 1;
  *(fp + ioffset) = t10;
  goto top;
label done;
```

Code Motion at Intermediate Code Level

Source:

```
        for (i = 0; i < 10; i = i+1) {
          a[i] = b[j];
        }
```

Unoptimized intermediate code:

```
          *(fp + ioffset) = 0;
        label top;
          t0 = *(fp + ioffset);
          iffalse (t0 < 10) goto done;
          t1 = *(fp + joffset);
          t2 = t1 * 4;
          t3 = fp + t2;
          t4 = *(t3 + boffset);
          t5 = *(fp + ioffset);
          t6 = t5 * 4;
          t7 = fp + t6; *(t7 + aoffset) = t4;
          t9 = *(fp + ioffset);
          t10 = t9 + 1;
          *(fp + ioffset) = t10;
          goto top;
        label done;
```

Loop Induction Variable Elimination

For-loop index is **induction variable**

- incremented each time around loop
- offsets & pointers calculated from it

If used only to index arrays, can rewrite with pointers

- compute initial offsets/pointers before loop
- increment offsets/pointers each time around loop
- no expensive scaling in loop

Source:

```
for (i = 0; i < 10; i = i+1) {
  a[i] = a[i] + x;
}
```

- Transformed source:

```
for (p = &a[0]; p < &a[10]; p = p+4) {
  *p = *p + x;
}
```

  - then do loop-invariant code motion

Global Register Allocation

Try to allocate local variables to registers

If life times of two locals don't overlap, can give to same register

Try to allocate most-frequently-used variables to registers first

Example:

```
int foo(int n, int x) {
  int sum; int i; int t;
  sum = x;
  for (i = n; i > 0; i=i-1) {
    sum = sum + i;
  }
  t = sum * sum;
  return t;
}
```

Interprocedural Optimization

Expand scope of analysis to procedures calling each other

Can do local & intraprocedural optimizations at larger scope

Can do new optimizations, e.g. inlining

Inlining

Replace procedure call with body of called procedure

Source:

```
final double pi = 3.1415927;
double circle_area(double radius) {
   return pi * (radius * radius);
}
...
double r = 5.0;
...
double a = circle_area(r);
```

After inlining:

```
...
double r = 5.0;
...
double a = pi * r * r;
```

(Then what?)

Summary

Enlarging scope of analysis yields better results

- today, most optimizing compilers work at the intraprocedural (a\k\a global) level

Optimizations organized as collections of passes, each rewriting IL in place into better version

Presence of optimizations makes other parts of compiler (e.g. intermediate and target code generation) easier to write

Intraprocedural (Global) Optimizations

Construct convenient representation of procedure body

Control flow graph (CFG) captures flow of control

- nodes are IL statements, or whole basic blocks
- edges represent control flow
- node with multiple successors = branch/switch
- node with multiple predecessors = merge
- loop in graph = loop

Data flow graph (DFG) capture flow of data

E.g. def/use chains:

- nodes are def(inition)s and uses
- edge from def to use
- a def can reach multiple uses
- a use can have multiple reaching defs

Example Program

```
x = 3;
y = x * x;
if (y > 10) {
  x = 5;
  y = y + 1;
} else {
  x = 6;
  y = x + 4;
}
w = y / 3;
while (y > 0) {
  z = w * w;
  x = x - z;
  y = y - 1;
}
System.out.println(x);
```

Example Program

```
x = 3;
y = x * x;
if (y > 10) {
  x = 5;
  y = y + 1;
} else {
  x = 6;
  y = x + 4;
}
w = y / 3;
while (y > 0) {
  z = w * w;
  x = x - z;
  y = y - 1;
}
System.out.println(x);
```

Analysis and Transformation

Each optimization is made up of

- some number of **analyses**
- followed by a **transformation**

Analyze CFG and/or DFG by propagating info forward or backward along CFG and/or DFG edges

- edges called **program points**
- merges in graph require combining info
- loops in graph require **iterative approximation**

Perform improving transformations based on info computed

- have to wait until any iterative approximation has converged

Analysis must be **conservative/safe/sound** so that transformations preserve program behavior

Example: Constant Propagation, Folding

Can use either the CFG or the DFG

CFG analysis info:

table mapping each variable in scope to one of

- a particular constant
- *NonConstant*
- *Undefined*

- Transformation: at each instruction:
  - if reference a variable that the table maps to a constant, then replace with that constant (constant propagation)
  - if r.h.s. expression involves only constants, and has no side-effects, then perform operation at compile-time and replace r.h.s. with constant result (constant folding)

For best analysis, do constant folding as part of analysis, to learn all constants in one pass

Example Program

```
x = 3;
y = x * x;
v = y - 2;
if (z > 10) {
   x = 5;
   y = y + 1;
} else {
   y = x + 4;
}
w = y / v;
if (v > 20) {
   v = x - 1;
}
u = x + v;
```

Merging data flow analysis info

How to merge analysis info?

Constraint: merge results must be sound

- if something is believed true after the merge, then it must be true no matter which path we took into the merge
- only things true along all predecessors are true after the merge

To merge two maps of constant info, build map by merging corresponding variable infos

To merge two variable infos:

- if one is Undefined, keep the other
- if both same constant, keep that constant
- otherwise, degenerate to NonConstant

Example Merges

$$\begin{tikzcd}
 & {\begin{matrix}\texttt{int x} \\ \texttt{x := 5}\end{matrix}} \arrow[dl] \arrow[dr] & \\
{} \arrow[dr] & & {} \arrow[dl] \\
 & \texttt{x ==?} &
\end{tikzcd}$$

$$\begin{tikzcd}
 & \texttt{int x} \arrow[dl] \arrow[dr] & \\
\texttt{x := 5} \arrow[dr] & & \texttt{x := 5} \arrow[dl] \\
 & \texttt{x ==?} &
\end{tikzcd}$$

$$\begin{tikzcd}
 & \texttt{int x} \arrow[dl] \arrow[dr] & \\
\texttt{x := 5} \arrow[dr] & & \texttt{x := 4} \arrow[dl] \\
 & \texttt{x ==?} &
\end{tikzcd}$$

Example Merges

$$\begin{tikzcd}
& \texttt{int x} \arrow[dl] \arrow[dr] & \\
{} \arrow[dr] & & \texttt{x := 5} \arrow[dl] \\
& \texttt{x ==?} &
\end{tikzcd}$$

$$\begin{tikzcd}
& \texttt{int x} \arrow[dl] \arrow[dr] & \\
\texttt{x := 5} \arrow[dr] & & \texttt{x := f(…)} \arrow[dl] \\
& \texttt{x ==?} &
\end{tikzcd}$$

Analysis of Loops

- How to analyze a loop?

```
i = 0;
x = 10;
y = 20;
while (...) {
  // what’s true here?
  ...
  i = i + 1;
  y = 30;
}
// what’s true here?
... x ... i ... y ...
```

A safe but imprecise approach:

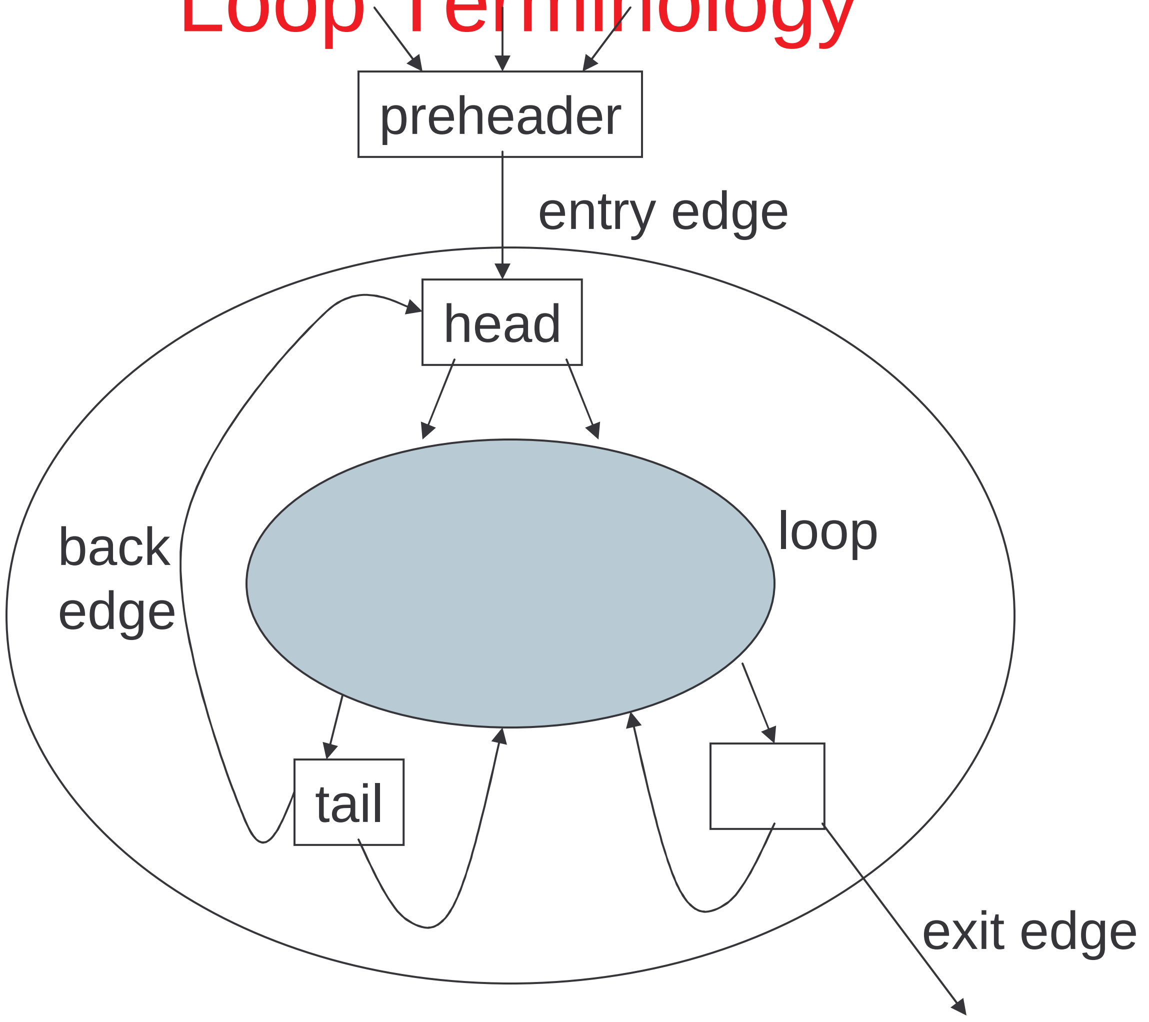
- forget everything when we enter or exit a loop

A precise but unsafe approach:

- keep everything when we enter or exit a loop

Can we do better?

Loop Terminology



Optimistic Iterative Analysis

1. Assuming info at loop head is same as info at loop entry
2. Then analyze loop body, computing info at back edge
3. Merge infos at loop back edge and loop entry
4. Test if merged info is same as original assumption
    a. If so, then we’re done
    b. If not, then replace previous assumption with merged info, and goto step 2

Example

```
i = 0;
x = 10;
y = 20;
while (...) {
   // what’s true here?
   ...
   i = i + 1;
   y = 30; }
// what’s true here?
... x ... i ... y ...
```

Why does optimistic iterative analysis work?

Why are the results always conservative?

Because if the algorithm stops, then

- the loop head info is at least as conservative as both the loop entry info and the loop back edge info
- the analysis within the loop body is conservative, given the assumption that the loop head info is conservative

Why does the algorithm terminate?

It might not!

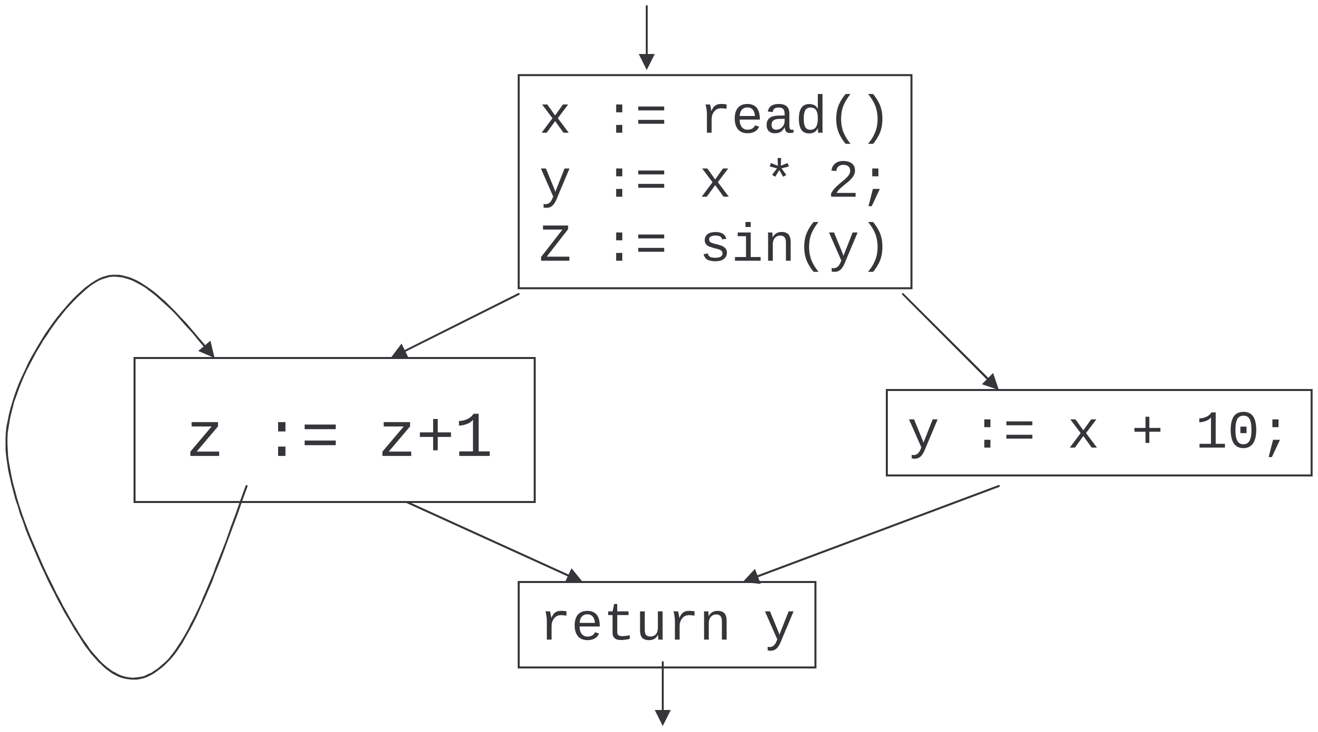
But it does if:

- there are only a finite number of times we could merge values together without reaching the worst case info (e.g. NotConstant)

Another example: live variable analysis

- Want the set of live variables 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?
- What is the requirement for this info to be conservative?
- How to merge two infos conservatively?
- How to analyze an assignment, e.g. X := Y + Z?
  - given *liveVars* before (or after?), what is computed after (or before?)
- What is live at procedure entry (or exit?)?

Example


x := read()
y := x * 2;
Z := sin(y)
z := z+1
y := x + 10;
return y