# Static Program Analysis

## Part 4 – flow sensitive analyses

http://cs.au.dk/~amoeller/spa/

Anders Møller & Michael I. Schwartzbach

Computer Science, Aarhus University

# Agenda

- **Constant propagation analysis**
- Live variables analysis
- Available expressions analysis
- Very busy expressions analysis
- Reaching definitions analysis
- Initialized variables analysis

# Constant propagation optimization

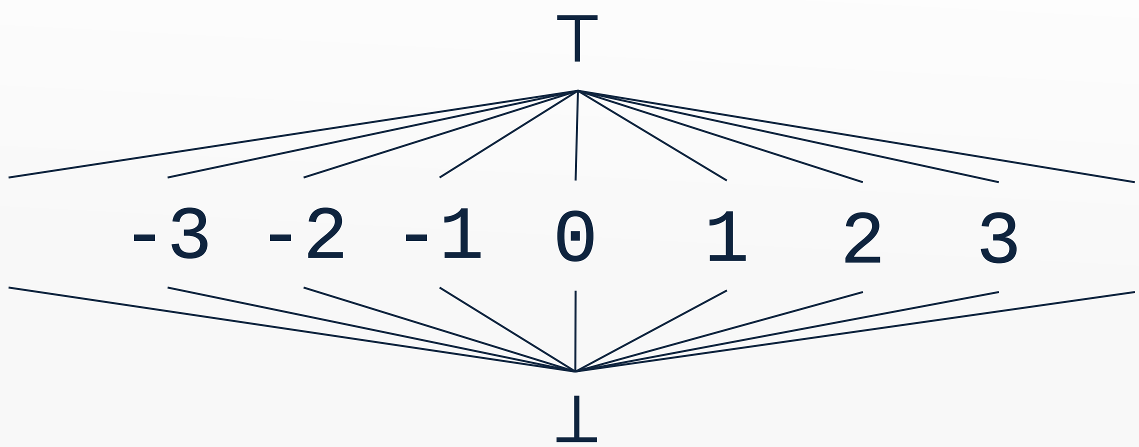
```
var x,y,z;
x = 27;
y = input,
z = 2*x+y;
if (x<0) { y=z-3; } else { y=12; }
output y;
```

```
var x,y,z;
x = 27;
y = input;
z = 54+y;
if (0) { y=z-3; } else { y=12; }
output y;
```

```
var y;
y = input;
output 12;
```

# Constant propagation analysis

- Determine variables with a constant value
- Flat lattice:



# Constraints for constant propagation

- Essentially as for the Sign analysis…

- Abstract operator for addition:

$$\overline{+}(n,m) = \begin{cases} \bot & \text{if } n=\bot \vee m=\bot \\ \top & \text{else if } n=\top \vee m=\top \\ n+m & \text{otherwise} \end{cases}$$

# Agenda

- Constant propagation analysis
- **Live variables analysis**
- Available expressions analysis
- Very busy expressions analysis
- Reaching definitions analysis
- Initialized variables analysis

# Liveness analysis

- A variable is *live* at a program point if its current value may be read in the remaining execution

- This is clearly undecidable, but the property can be conservatively approximated

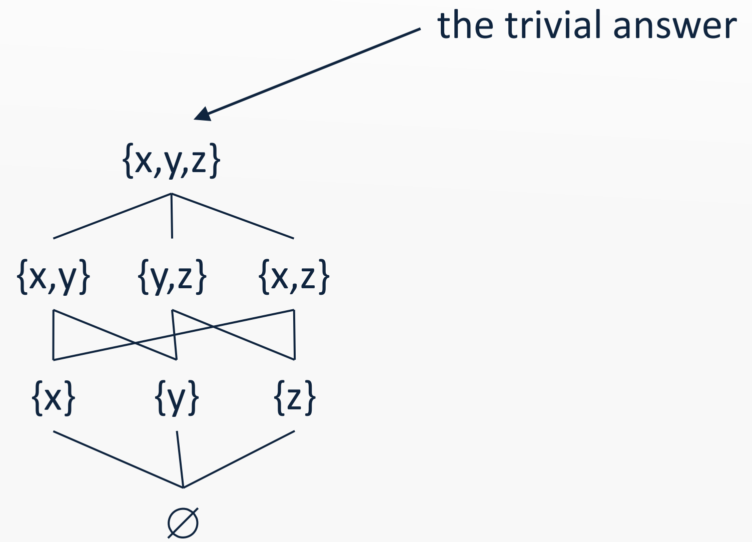
- The analysis must only answer *"dead"* if the variable is really dead
  – no need to store the values of dead variables

# A lattice for liveness

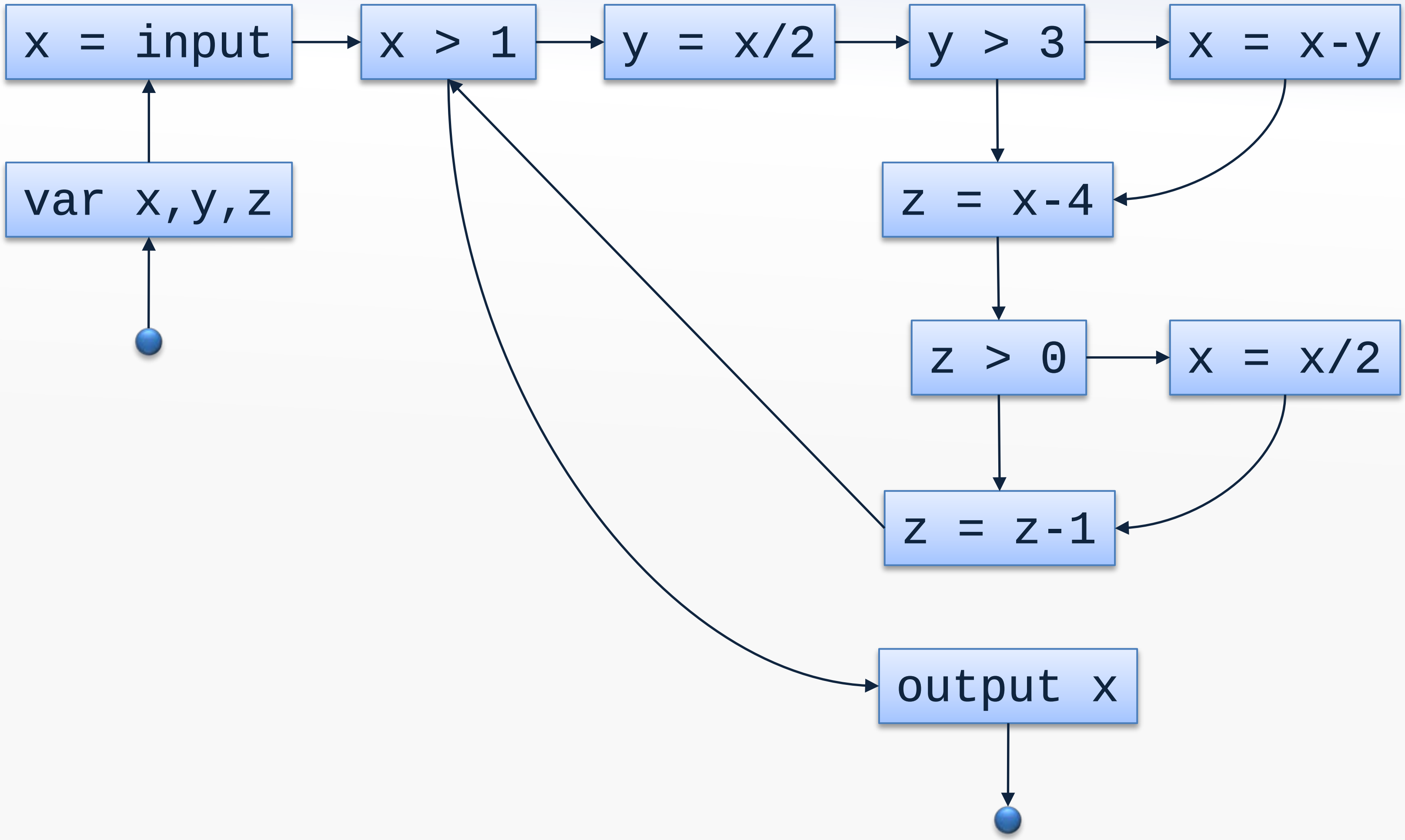
A powerset lattice of program variables

```
var x,y,z;
x = input;
while (x>1) {
  y = x/2;
  if (y>3) x = x-y;
  z = x-4;
  if (z>0) x = x/2;
  z = z-1;
}
output x;
```

$$L = (\mathcal{P}(\{x,y,z\}), \subseteq)$$



# The control flow graph



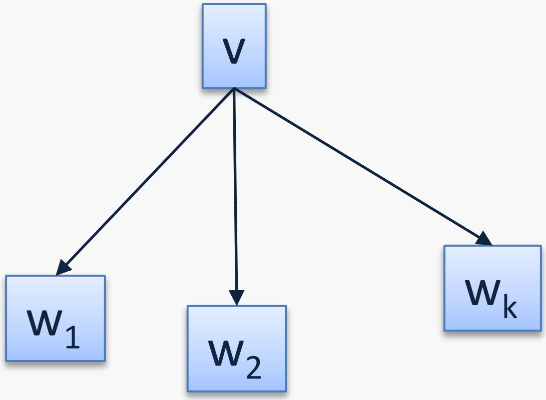
# Setting up

- For every CFG node, v, we have a variable ⟦v⟧:
  - the set of program variables that are live at the program point *before* v

- Since the analysis is conservative, the computed sets may be *too large*

- Auxiliary definition:

$$JOIN(v) = \bigcup_{w \in succ(v)} [\![w]\!]$$



# Liveness constraints

- For the exit node:

  $[\![exit]\!] = \emptyset$

- For conditions and output:

  $[\![$ `if (E)` $]\!] = [\![$ `output` $E\,]\!] = JOIN(\mathrm{v}) \cup vars(E)$

- For assignments:

  $[\![\, x = E\,]\!] = JOIN(\mathrm{v}) \setminus \{x\} \cup vars(E)$

- For variable declarations:

  $[\![$ `var` $x_1,\ \ldots,\ x_n\,]\!] = JOIN(\mathrm{v}) \setminus \{x_1, \ldots, x_n\}$

- For all other nodes:

  $[\![\mathrm{v}]\!] = JOIN(\mathrm{v})$

*vars*(*E*) = variables occurring in *E*

right-hand sides are monotone since *JOIN* is monotone, and …

# Generated constraints

$[\![\texttt{var x,y,z}]\!] = [\![\texttt{x=input}]\!] \setminus \{x,y,z\}$

$[\![\texttt{x=input}]\!] = [\![\texttt{x>1}]\!] \setminus \{x\}$

$[\![\texttt{x>1}]\!] = ([\![\texttt{y=x/2}]\!] \cup [\![\texttt{output x}]\!]) \cup \{x\}$

$[\![\texttt{y=x/2}]\!] = ([\![\texttt{y>3}]\!] \setminus \{y\}) \cup \{x\}$

$[\![\texttt{y>3}]\!] = [\![\texttt{x=x-y}]\!] \cup [\![\texttt{z=x-4}]\!] \cup \{y\}$

$[\![\texttt{x=x-y}]\!] = ([\![\texttt{z=x-4}]\!] \setminus \{x\}) \cup \{x,y\}$

$[\![\texttt{z=x-4}]\!] = ([\![\texttt{z>0}]\!] \setminus \{z\}) \cup \{x\}$

$[\![\texttt{z>0}]\!] = [\![\texttt{x=x/2}]\!] \cup [\![\texttt{z=z-1}]\!] \cup \{z\}$

$[\![\texttt{x=x/2}]\!] = ([\![\texttt{z=z-1}]\!] \setminus \{x\}) \cup \{x\}$

$[\![\texttt{z=z-1}]\!] = ([\![\texttt{x>1}]\!] \setminus \{z\}) \cup \{z\}$

$[\![\texttt{output x}]\!] = [\![\textit{exit}]\!] \cup \{x\}$

$[\![\textit{exit}]\!] = \emptyset$

# Least solution

$[\![\mathit{entry}]\!] = \emptyset$

$[\![$`var x,y,z`$]\!] = \emptyset$

$[\![$`x=input`$]\!] = \emptyset$

$[\![$`x>1`$]\!] = \{x\}$

$[\![$`y=x/2`$]\!] = \{x\}$

$[\![$`y>3`$]\!] = \{x,y\}$

$[\![$`x=x-y`$]\!] = \{x,y\}$

$[\![$`z=x-4`$]\!] = \{x\}$

$[\![$`z>0`$]\!] = \{x,z\}$

$[\![$`x=x/2`$]\!] = \{x,z\}$

$[\![$`z=z-1`$]\!] = \{x,z\}$

$[\![$`output x`$]\!] = \{x\}$

$[\![\mathit{exit}]\!] = \emptyset$

Many non-trivial answers!

# Optimizations

- Variables `y` and `z` are never simultaneously live
  - ⇒ they can share the same variable location
- The value assigned in `z=z-1` is never read
  - ⇒ the assignment can be skipped

```
var x,yz;
x = input;
while (x>1) {
  yz = x/2;
  if (yz>3) x = x-yz;
  yz = x-4;
  if (yz>0) x = x/2;
}
output x;
```

- better register allocation
- a few clock cycles saved

# Time complexity (for the naive algorithm)

- With $n$ CFG nodes and $k$ variables:
  - the lattice $L^n$ has height $k \cdot n$
  - so there are at most $k \cdot n$ iterations
- Subsets of Vars (the variables in the program) can be represented as bitvectors:
  - each element has size $k$
  - each $\cup$, $\backslash$, $=$ operation takes time $O(k)$
- Each iteration uses $O(n)$ bitvector operations:
  - so each iteration takes time $O(k \cdot n)$
- Total time complexity: $O(k^2 n^2)$

- Exercise: what is the complexity for the worklist algorithm?

# Agenda

- Constant propagation analysis
- Live variables analysis
- **Available expressions analysis**
- Very busy expressions analysis
- Reaching definitions analysis
- Initialized variables analysis

# Available expressions analysis

- A (nontrivial) expression is *available* at a program point if its current value has already been computed earlier in the execution

- The approximation generally includes *too few* expressions
  - the analysis can only report *"available"* if the expression is definitely available
  - no need to re-compute available expressions (e.g. common subexpression elimination)

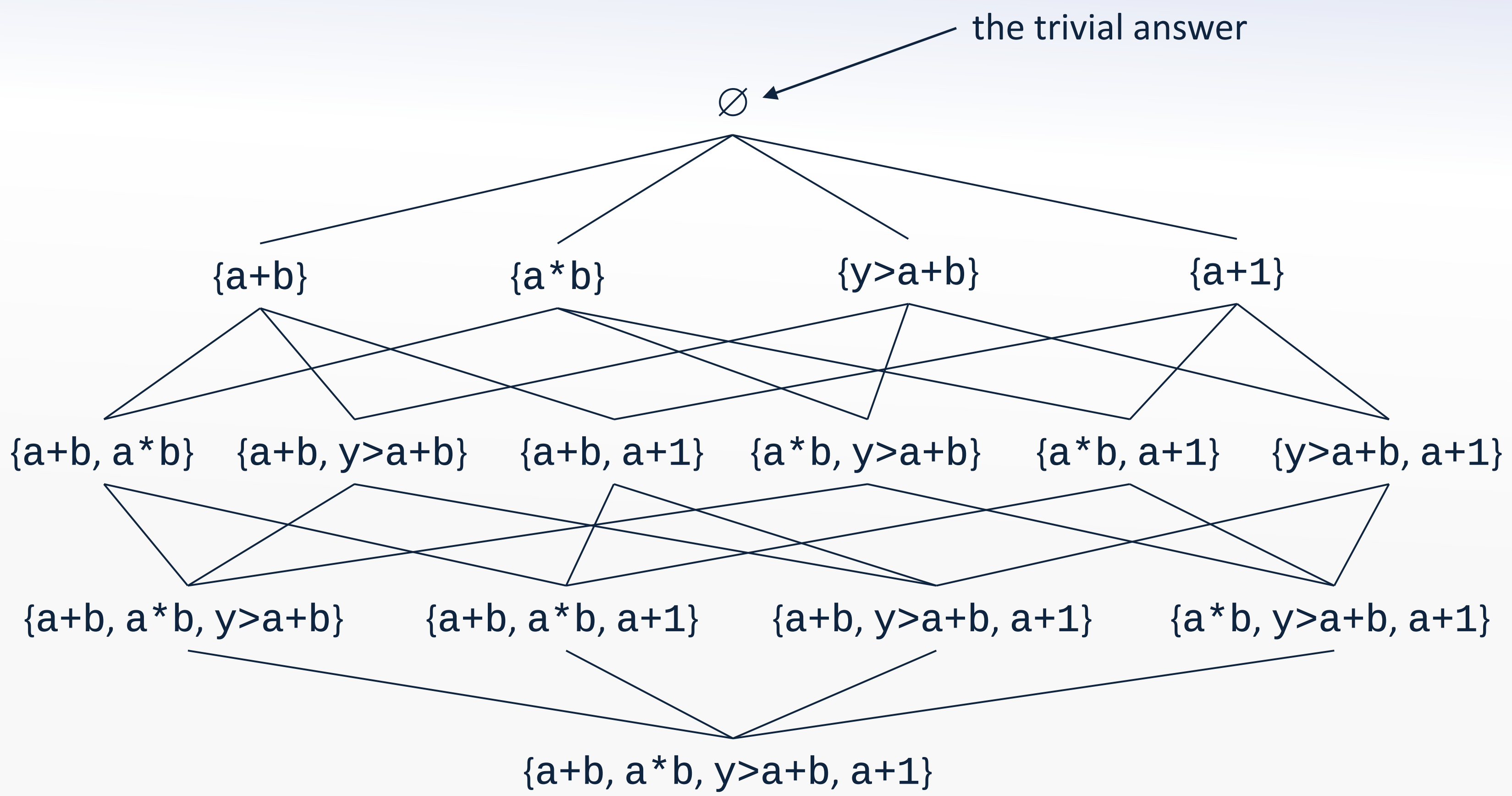
# A lattice for available expressions

A reverse powerset lattice of nontrivial expressions

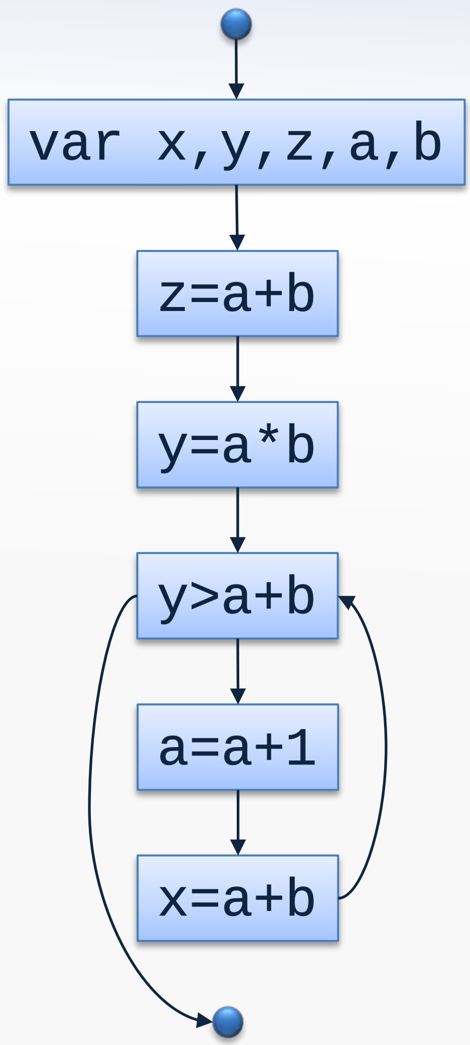
```
var x,y,z,a,b;
z = a+b;
y = a*b;
while (y > a+b) {
  a = a+1;
  x = a+b;
}
```

$$L = (\mathcal{P}(\{\texttt{a+b}, \texttt{a*b}, \texttt{y>a+b}, \texttt{a+1}\}), \supseteq)$$

# Reverse powerset lattice



# The control flow graph



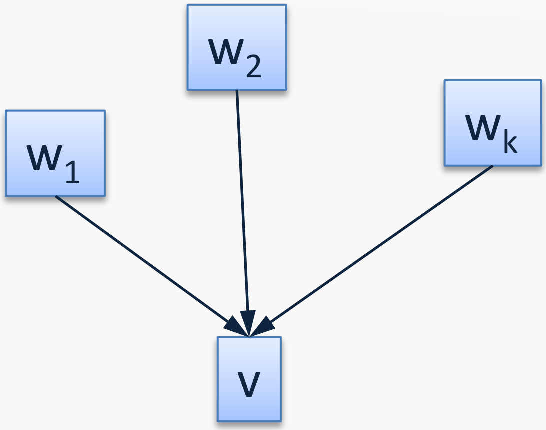
# Setting up

- For every CFG node, v, we have a variable ⟦v⟧:
  - the set of expressions that are available at the program point *after* v

- Since the analysis is conservative, the computed sets may be *too small*

- Auxiliary definition:

$$JOIN(v) = \bigcap_{w \in pred(v)} [\![w]\!]$$



# Auxiliary functions

- The function $S{\downarrow}x$ removes all expressions that contain the variable $x$ from the set S

- The function *exps*(*E*) is defined as:
  - $exps(intconst) = \emptyset$
  - $exps(x) = \emptyset$
  - $exps(\texttt{input}) = \emptyset$
  - $exps(E_1\ op\ E_2) = \{E_1\ op\ E_2\} \cup exps(E_1) \cup exps(E_2)$
    but don’t include expressions containing `input`

# Availability constraints

- For the *entry* node:

  $[\![entry]\!] = \emptyset$

- For conditions and output:

  $[\![$ `if (`$E$`)` $]\!] = [\![$ `output` $E\ ]\!] = JOIN(\mathrm{v}) \cup exps(E)$

- For assignments:

  $[\![\ x = E\ ]\!] = (JOIN(\mathrm{v}) \cup exps(E))\!\downarrow\! x$

- For any other node v:

  $[\![\mathrm{v}]\!] = JOIN(\mathrm{v})$

# Generated constraints

$[\![ \mathit{entry} ]\!] = \emptyset$

$[\![ \texttt{var x,y,z,a,b} ]\!] = [\![ \mathit{entry} ]\!]$

$[\![ \texttt{z=a+b} ]\!] = \mathit{exps}(\texttt{a+b}){\downarrow}\texttt{z}$

$[\![ \texttt{y=a*b} ]\!] = ([\![ \texttt{z=a+b} ]\!] \cup \mathit{exps}(\texttt{a*b})){\downarrow}\texttt{y}$

$[\![ \texttt{y>a+b} ]\!] = ([\![ \texttt{y=a*b} ]\!] \cap [\![ \texttt{x=a+b} ]\!]) \cup \mathit{exps}(\texttt{y>a+b})$

$[\![ \texttt{a=a+1} ]\!] = ([\![ \texttt{y>a+b} ]\!] \cup \mathit{exps}(\texttt{a+1})){\downarrow}\texttt{a}$

$[\![ \texttt{x=a+b} ]\!] = ([\![ \texttt{a=a+1} ]\!] \cup \mathit{exps}(\texttt{a+b})){\downarrow}\texttt{x}$

$[\![ \mathit{exit} ]\!] = [\![ \texttt{y>a+b} ]\!]$

# Least solution

$[\![\mathit{entry}]\!] = \emptyset$

$[\![$`var x,y,z,a,b`$]\!] = \emptyset$

$[\![$`z=a+b`$]\!] = \{$`a+b`$\}$

$[\![$`y=a*b`$]\!] = \{$`a+b`, `a*b`$\}$

$[\![$`y>a+b`$]\!] = \{$`a+b`, `y>a+b`$\}$

$[\![$`a=a+1`$]\!] = \emptyset$

$[\![$`x=a+b`$]\!] = \{$`a+b`$\}$

$[\![\mathit{exit}]\!] = \{$`a+b`$\}$

Again, many nontrivial answers!

# Optimizations

- We notice that a+b is available before the loop
- The program can be optimized (slightly):

```
var x,y,x,a,b,aplusb;
aplusb = a+b;
z = aplusb;
y = a*b;
while (y > aplusb) {
  a = a+1;
  aplusb = a+b;
  x = aplusb;
}
```

# Agenda

- Constant propagation analysis
- Live variables analysis
- Available expressions analysis
- **Very busy expressions analysis**
- Reaching definitions analysis
- Initialized variables analysis

# Very busy expressions analysis

- A (nontrivial) expression is *very busy* if it will definitely be evaluated before its value changes

- The approximation generally includes *too few* expressions
  – the answer "*very busy*" must be the true one
  – very busy expressions may be pre-computed (e.g. loop hoisting)

- Same lattice as for available expressions

# An example program

```
var x,a,b;
x = input;
a = x-1;
b = x-2;
while (x > 0) {
  output a*b-x;
  x = x-1;
}
output a*b;
```

The analysis shows that `a*b` is very busy right before the while loop

# Code hoisting

```
var x,a,b;
x = input;
a = x-1;
b = x-2;
while (x > 0) {
  output a*b-x;
  x = x-1;
}
output a*b;
```

```
var x,a,b,atimesb;
x = input;
a = x-1;
b = x-2;
atimesb = a*b;
while (x > 0) {
  output atimesb-x;
  x = x-1;
}
output atimesb;
```

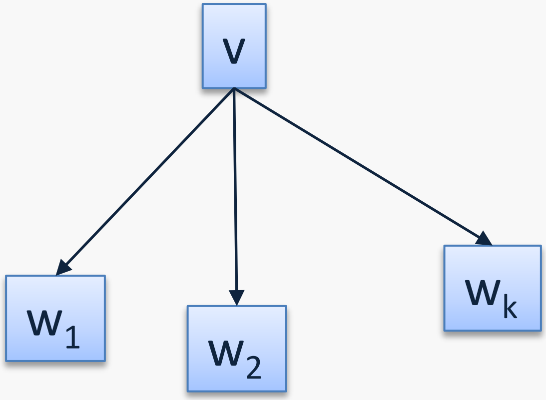
# Setting up

- For every CFG node, v, we have a variable ⟦v⟧:
  - the set of expressions that are very busy at the program point *before* v

- Since the analysis is conservative, the computed sets may be *too small*

- Auxiliary definition:

$$JOIN(v) = \bigcap_{w \in succ(v)} [\![w]\!]$$



# Very busy constraints

- For the *exit* node:

  $[\![exit]\!] = \emptyset$

- For conditions and output:

  $[\![\texttt{if (}E\texttt{)}]\!] = [\![\texttt{output}\ E]\!] = JOIN(\mathrm{v}) \cup exps(E)$

- For assignments:

  $[\![x = E]\!] = JOIN(\mathrm{v}){\downarrow}x \cup exps(E)$

- For all other nodes:

  $[\![\mathrm{v}]\!] = JOIN(\mathrm{v})$

same ↓ operator as for available expressions analysis

# Agenda

- Constant propagation analysis
- Live variables analysis
- Available expressions analysis
- Very busy expressions analysis
- **Reaching definitions analysis**
- Initialized variables analysis

# Reaching definitions analysis

- The *reaching definitions* for a program point are those assignments that may define the current values of variables

- The conservative approximation may include *too many* possible assignments

# A lattice for reaching definitions

The powerset lattice of assignments

$$L = (\mathcal{P}(\{\texttt{x=input}, \texttt{y=x/2}, \texttt{x=x-y}, \texttt{z=x-4}, \texttt{x=x/2}, \texttt{z=z-1}\}), \subseteq)$$

```
var x,y,z;
x = input;
while (x > 1) {
  y = x/2;
  if (y>3) x = x-y;
  z = x-4;
  if (z>0) x = x/2;
  z = z-1;
}
output x;
```

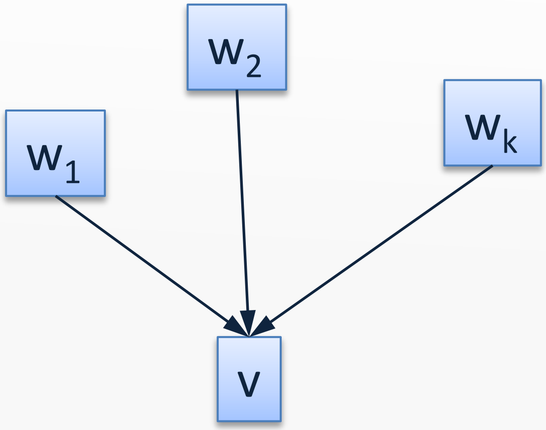
# Reaching definitions constraints

- For assignments:

$$[\![ x = E ]\!] = JOIN(\mathrm{v}){\downarrow}x \cup \{ x = E \}$$

- For all other nodes:

$$[\![\mathrm{v}]\!] = JOIN(\mathrm{v})$$




- Auxiliary definition:

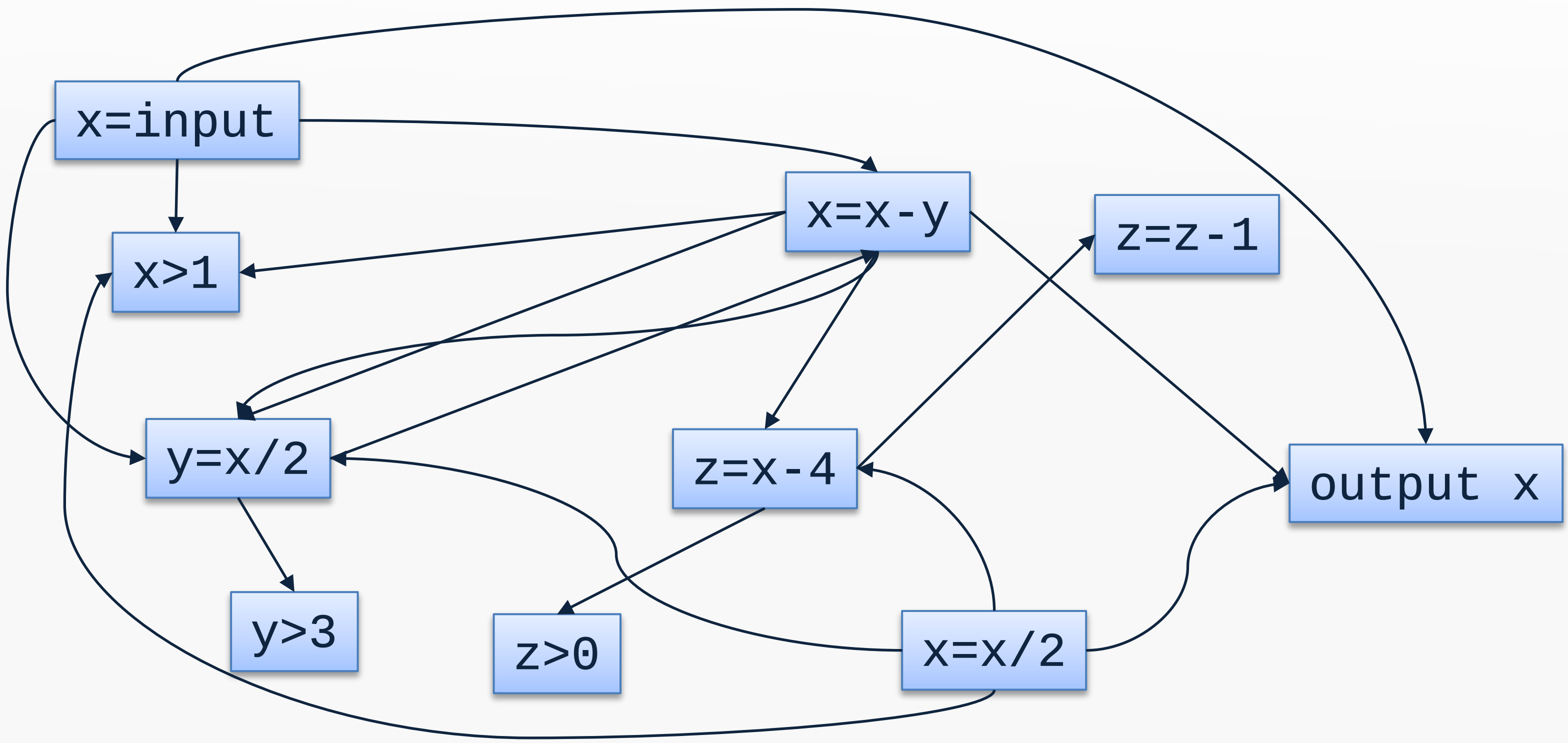
$$JOIN(\mathrm{v}) = \bigcup_{\mathrm{w} \in pred(\mathrm{v})} [\![\mathrm{w}]\!]$$

- The function S↓*x* removes assignments to *x* from the set S

# Def-use graph

Reaching definitions define the def-use graph:

- like a CFG but with edges from *def* to *use* nodes
- basis for *dead code elimination* and *code motion*



# Forward vs. backward

- A *forward* analysis:
  - computes information about the *past* behavior
  - examples: available expressions, reaching definitions
- A *backward* analysis:
  - computes information about the *future* behavior
  - examples: liveness, very busy expressions

# May vs. must

- A *may* analysis:
  - describes information that is *possibly* true
  - an *over*-approximation
  - examples: liveness, reaching definitions

- A *must* analysis:
  - describes information that is *definitely* true
  - an *under*-approximation
  - examples: available expressions, very busy expressions

# Classifying analyses

| | forward | backward |
|---|---|---|
| **may** | example: reaching definitions<br><br>$[\![v]\!]$ describes state after v<br><br>$\text{JOIN}(v) = \bigsqcup_{w \in pred(v)} [\![w]\!] = \bigcup_{w \in pred(v)} [\![w]\!]$ | example: liveness<br><br>$[\![v]\!]$ describes state before v<br><br>$\text{JOIN}(v) = \bigsqcup_{w \in succ(v)} [\![w]\!] = \bigcup_{w \in succ(v)} [\![w]\!]$ |
| **must** | example: available expressions<br><br>$[\![v]\!]$ describes state after v<br><br>$\text{JOIN}(v) = \bigsqcup_{w \in pred(v)} [\![w]\!] = \bigcap_{w \in pred(v)} [\![w]\!]$ | example: very busy expressions<br><br>$[\![v]\!]$ describes state before v<br><br>$\text{JOIN}(v) = \bigsqcup_{w \in succ(v)} [\![w]\!] = \bigcap_{w \in succ(v)} [\![w]\!]$ |

# Agenda

- Constant propagation analysis
- Live variables analysis
- Available expressions analysis
- Very busy expressions analysis
- Reaching definitions analysis
- **Initialized variables analysis**

# Initialized variables analysis

- Compute for each program point those variables that have *definitely* been initialized in the *past*
- (Called *definite assignment* analysis in Java and C#)
- $\Rightarrow$ *forward must analysis*
- Reverse powerset lattice of all variables

$$JOIN(v) = \bigcap_{w \in pred(v)} [\![w]\!]$$

- For assignments: $[\![\ x = E\ ]\!] = JOIN(v) \cup \{x\}$
- For all others: $[\![v]\!] = JOIN(v)$