中国科学技术大学
University of Science and Technology of China

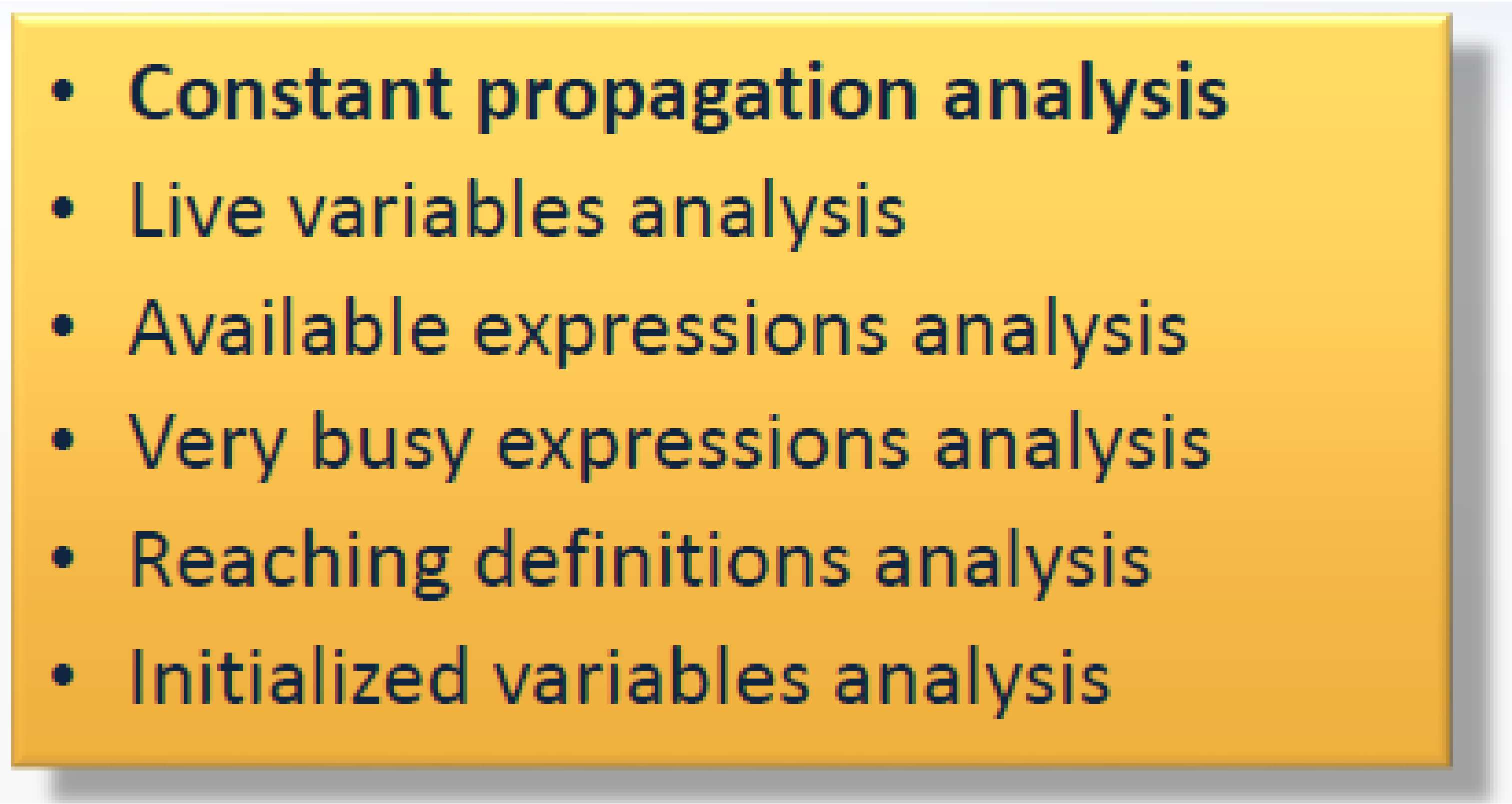
Flow Sensitive Analysis

Most content comes from http://cs.au.dk/~amoeller/spa/

张昱

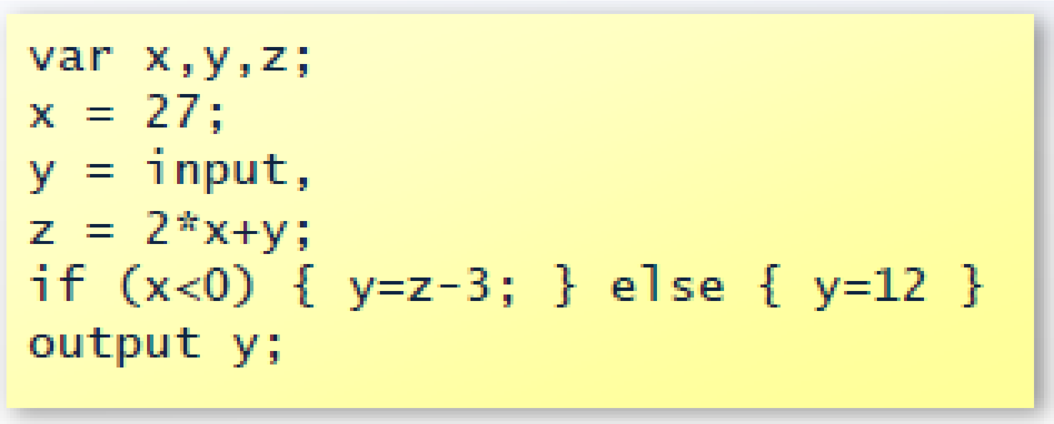
yuzhang@ustc.edu.cn
中国科学技术大学
计算机科学与技术学院

Agenda

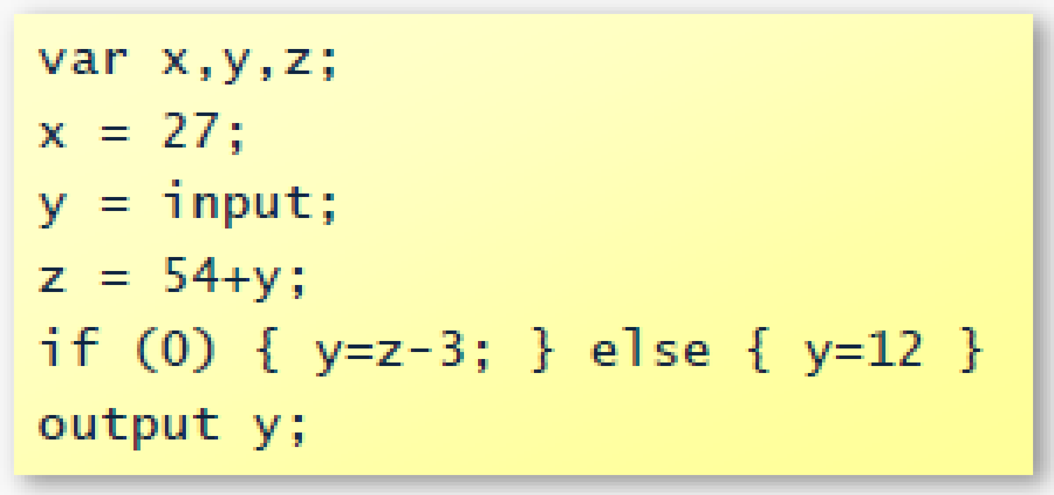


- **Constant propagation analysis**
- Live variables analysis
- Available expressions analysis
- Very busy expressions analysis
- Reaching definitions analysis
- Initialized variables analysis

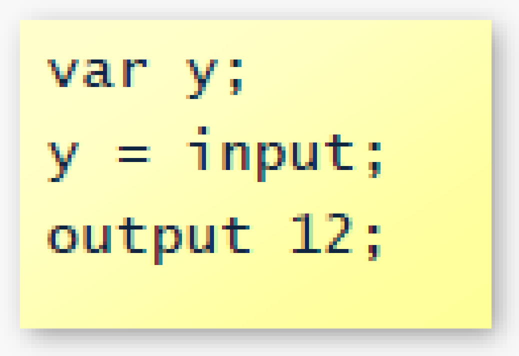
Constant Propagation Optimization



```
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;
```

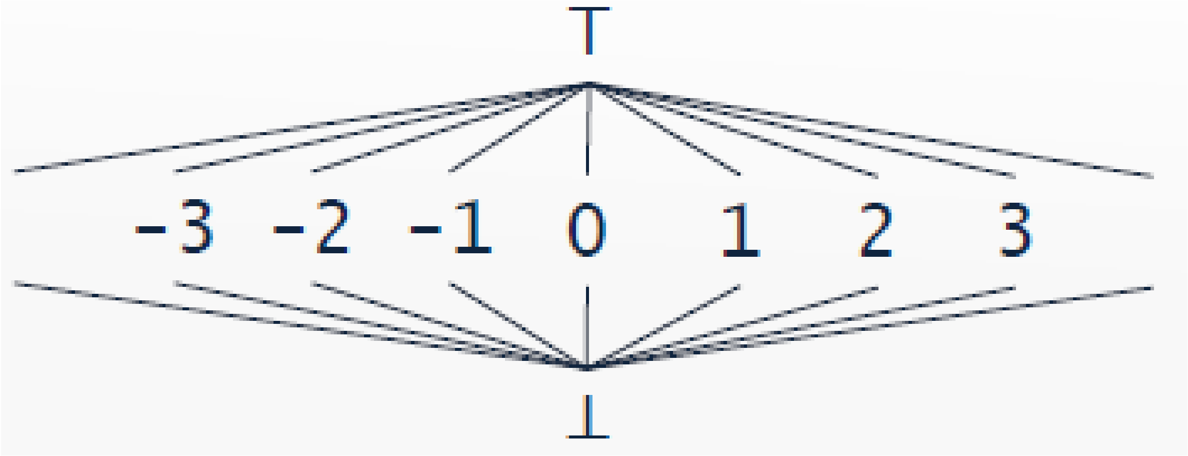


```
var y;
y = input;
output 12;
```

https://github.com/cs-au-dk/TIP/blob/master/src/tip/analysis/ConstantPropagationAnalysis.scala

Constant Propagation Analysis

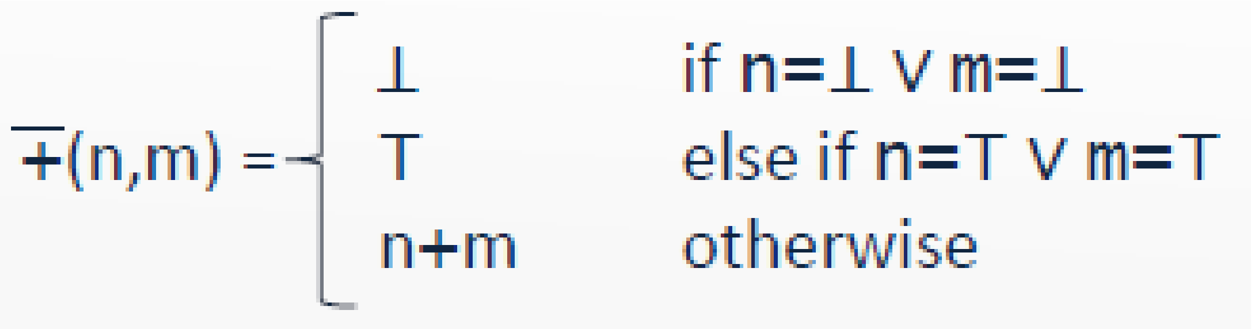
- Determine variables with a constant value
- Flat lattice:



Constraints for Constant Propagation

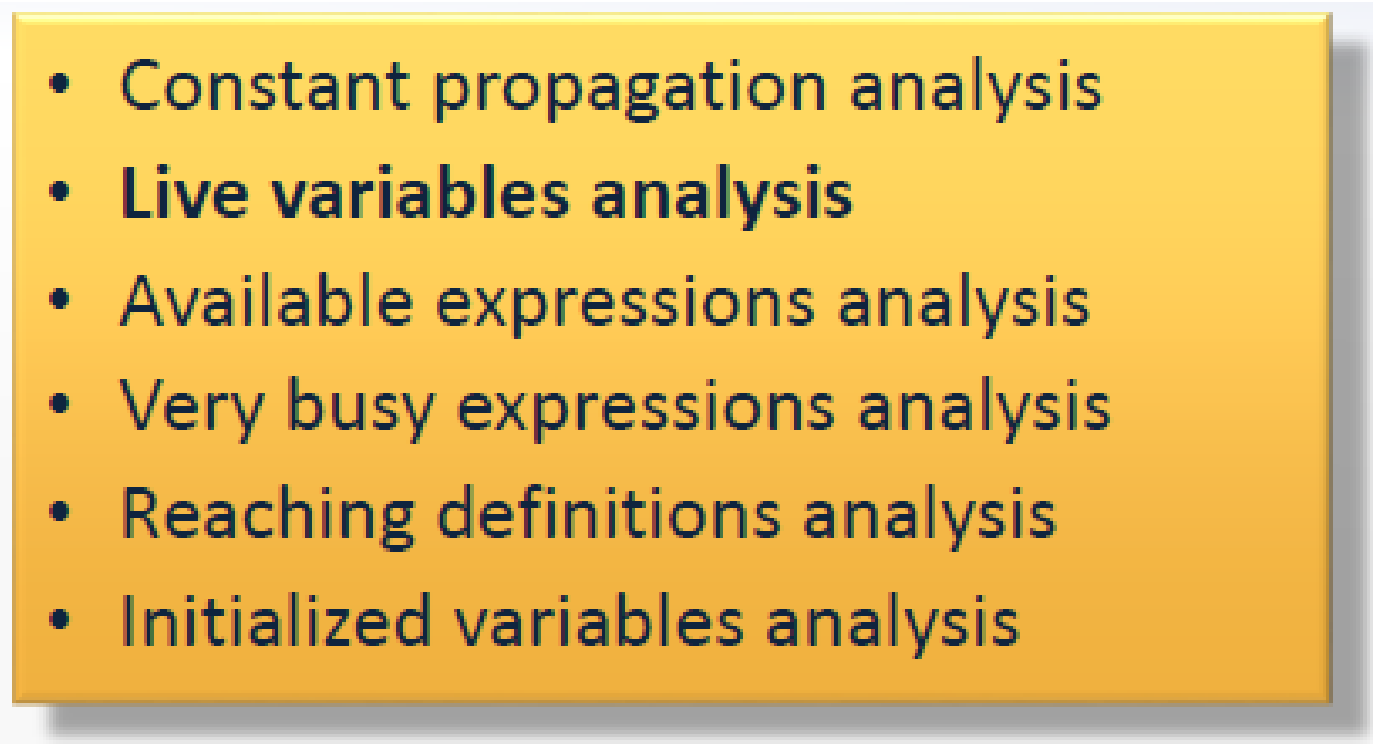
- □ **Essentially as for the Sign analysis…**

- □ **Abstract operator for addition:**



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

Agenda



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

Live Variables Analysis

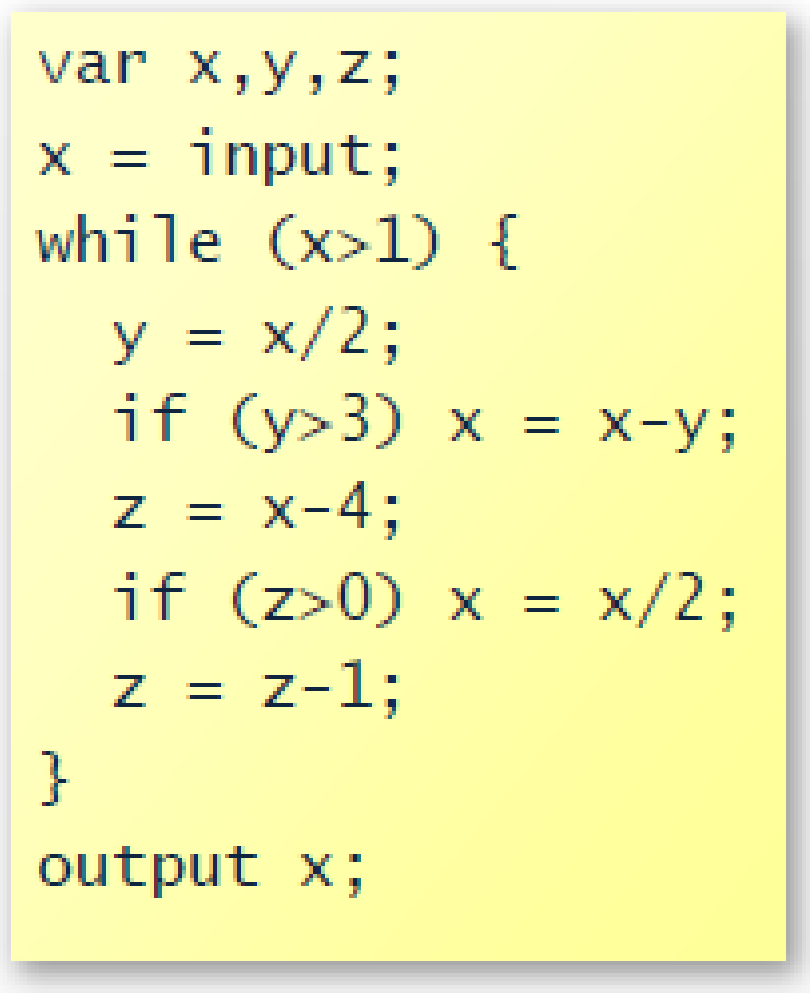
- ☐ **A variable is *live* at a program point its value may be read later in the remaining execution**

- ☐ **Undecidable, but the property can be conservatively approximated**

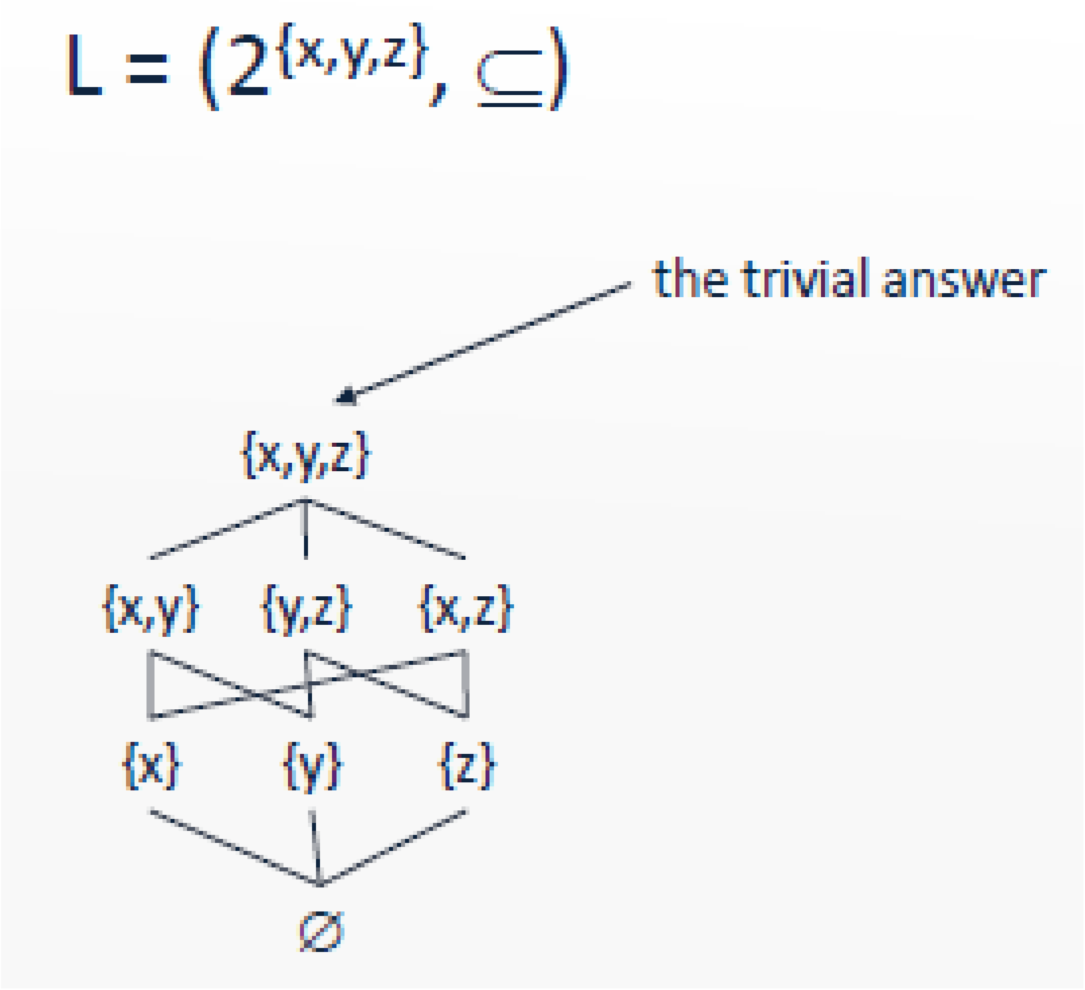
- ☐ **The analysis must only reply *dead* if the variable is really dead**
  - No need to store the values of dead variables

A Lattice for Liveness

- ☐ **A powerset lattice of program variables**



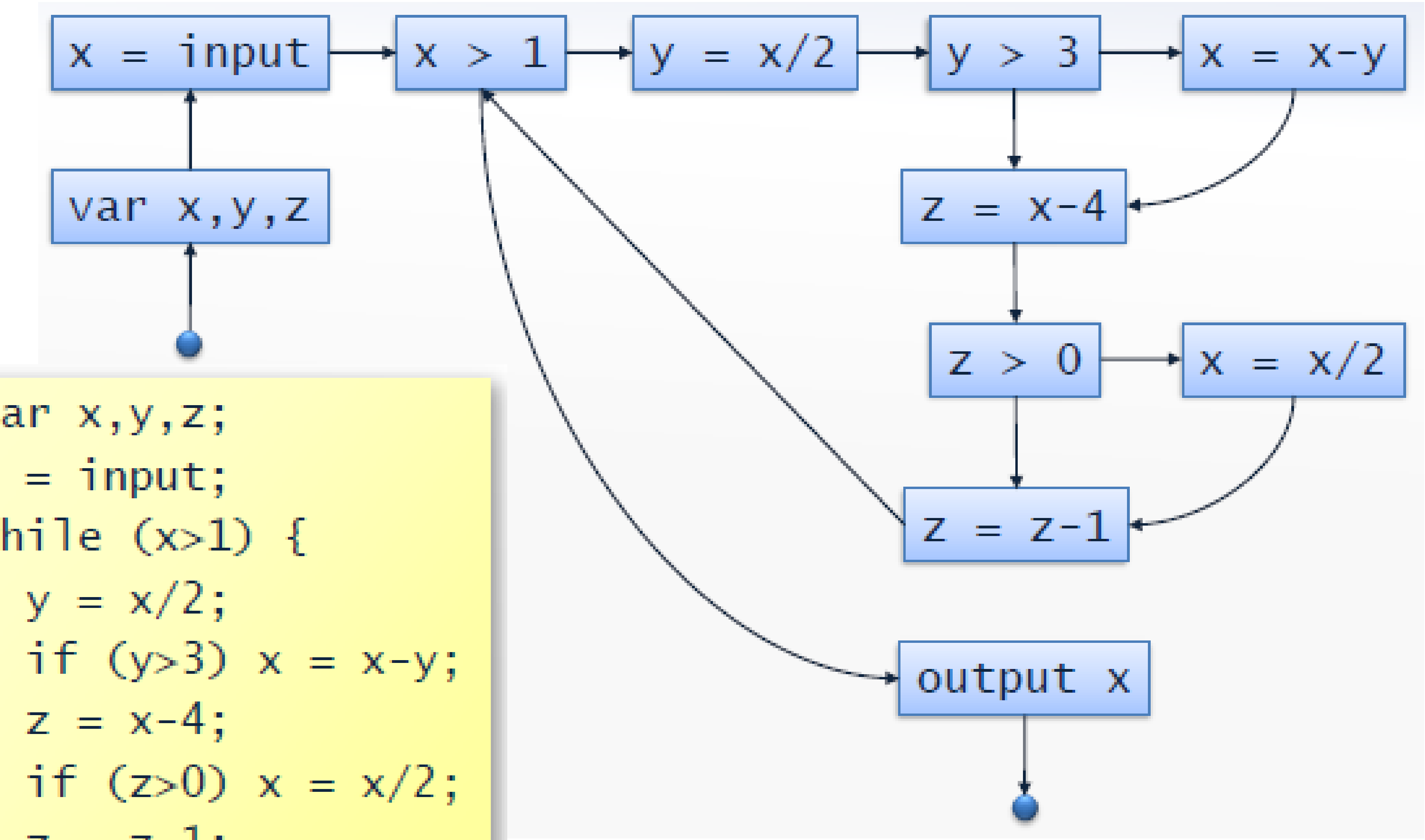
```
var x,y,z;
x = input;
while (x>1) {
  y = x/2;
  if (y>3) x = x-y;
  z = x-4;
  if (z>0) x = x/2;
  z = z-1;
}
output x;
```



$$L = (2^{\{x,y,z\}}, \subseteq)$$

the trivial answer

The Control Flow Graph



```
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;
```

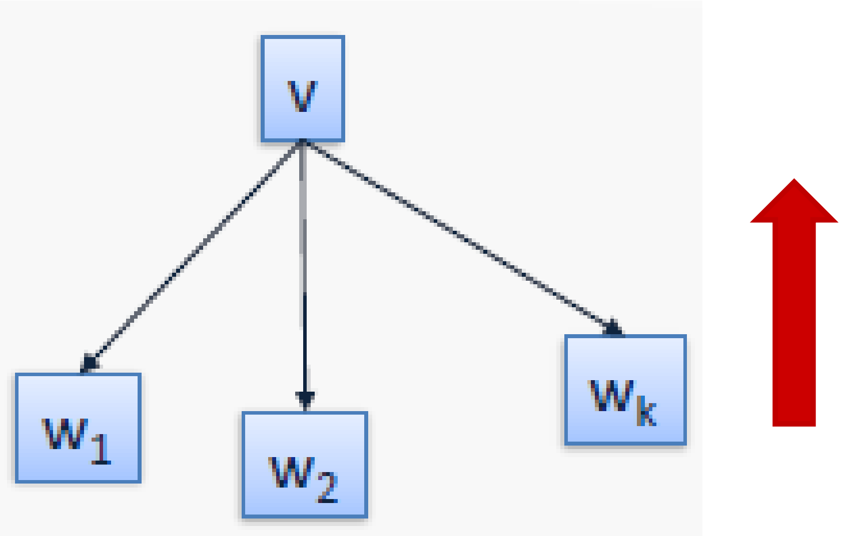
Setting Up

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

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

分析出的是**可能**的活跃变量集合

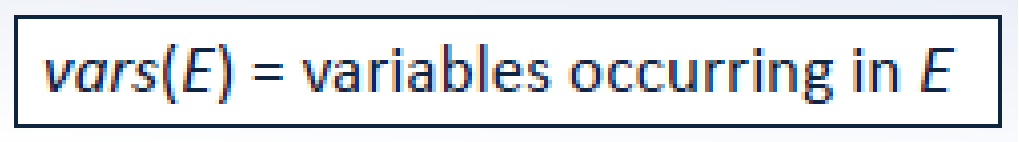
- **Auxiliary definition**
  - $\mathrm{JOIN}(v) = \bigcup_{w \in succ(v)} [\![w]\!]$



Liveness Constraints

- **For the exit node**

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



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

- **For conditions and output**

$[\![\text{if } (E)]\!] = [\![\text{while } E]\!] = [\![\text{output } E]\!] = JOIN(v) \cup vars(E)$

- **For assignments**

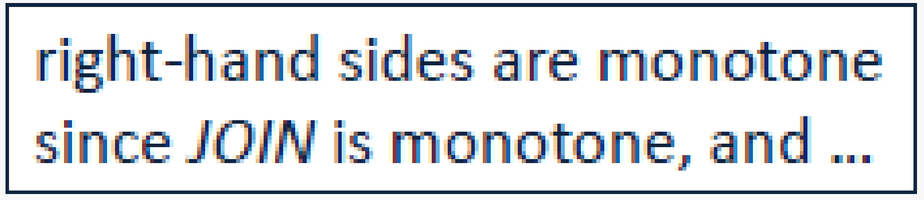
$[\![x = E]\!] = JOIN(v) \backslash \{x\} \cup vars(E)$

- **For variable declarations**

$[\![\text{var } x_1, \cdots, x_n]\!] = JOIN(v) \backslash \{x_1, \cdots, x_n\}$

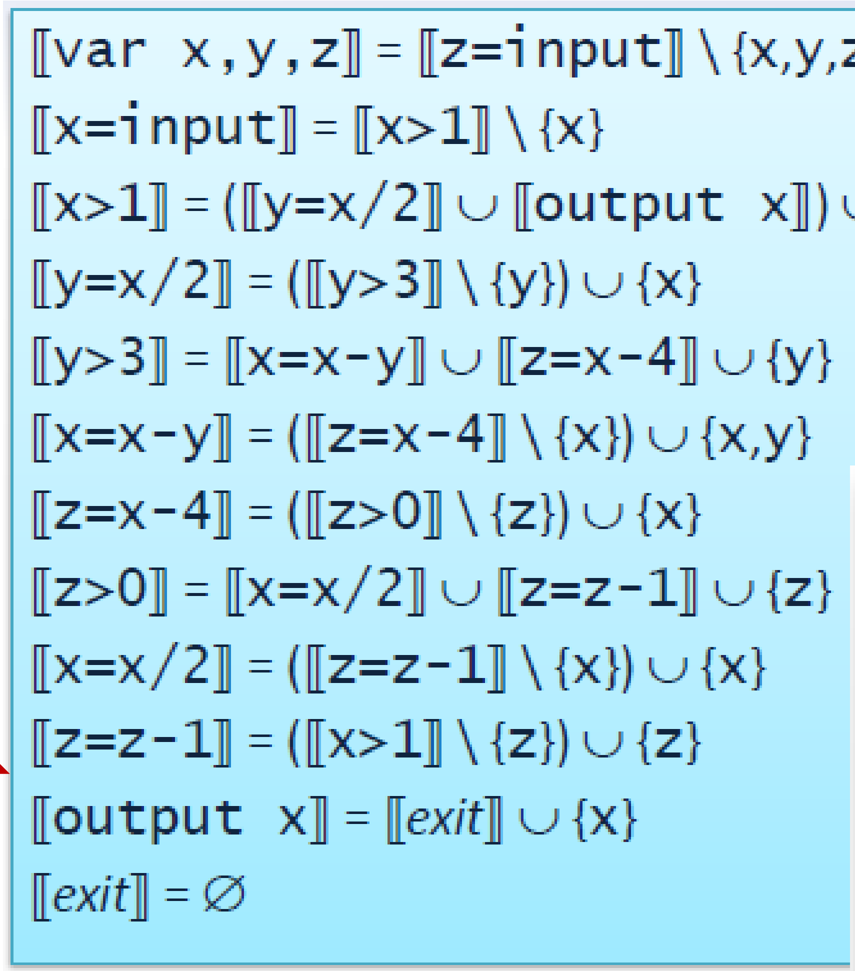
- **For all other nodes**

$[\![v]\!] = JOIN(v)$



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

Generated Constraints



$\llbracket\texttt{var x,y,z}\rrbracket = \llbracket\texttt{z=input}\rrbracket \setminus \{x,y,z\}$

$\llbracket\texttt{x=input}\rrbracket = \llbracket\texttt{x>1}\rrbracket \setminus \{x\}$

$\llbracket\texttt{x>1}\rrbracket = (\llbracket\texttt{y=x/2}\rrbracket \cup \llbracket\texttt{output x}\rrbracket) \cup \{x\}$

$\llbracket\texttt{y=x/2}\rrbracket = (\llbracket\texttt{y>3}\rrbracket \setminus \{y\}) \cup \{x\}$

$\llbracket\texttt{y>3}\rrbracket = \llbracket\texttt{x=x-y}\rrbracket \cup \llbracket\texttt{z=x-4}\rrbracket \cup \{y\}$

$\llbracket\texttt{x=x-y}\rrbracket = (\llbracket\texttt{z=x-4}\rrbracket \setminus \{x\}) \cup \{x,y\}$

$\llbracket\texttt{z=x-4}\rrbracket = (\llbracket\texttt{z>0}\rrbracket \setminus \{z\}) \cup \{x\}$

$\llbracket\texttt{z>0}\rrbracket = \llbracket\texttt{x=x/2}\rrbracket \cup \llbracket\texttt{z=z-1}\rrbracket \cup \{z\}$

$\llbracket\texttt{x=x/2}\rrbracket = (\llbracket\texttt{z=z-1}\rrbracket \setminus \{x\}) \cup \{x\}$

$\llbracket\texttt{z=z-1}\rrbracket = (\llbracket\texttt{x>1}\rrbracket \setminus \{z\}) \cup \{z\}$

$\llbracket\texttt{output x}\rrbracket = \llbracket exit\rrbracket \cup \{x\}$

$\llbracket exit\rrbracket = \varnothing$

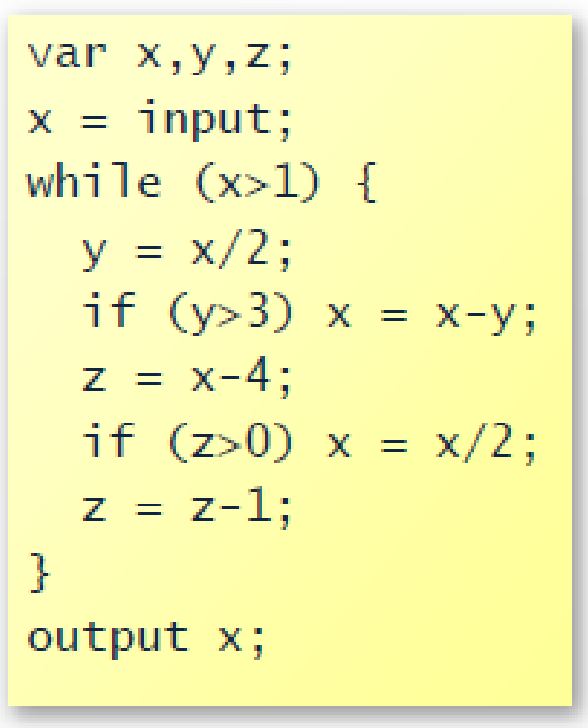
$\llbracket exit\rrbracket = \emptyset$

$\llbracket\text{if }(E)\rrbracket = \llbracket\text{while }E\rrbracket = \llbracket\text{output }E\rrbracket = JOIN(v) \cup vars(E)$

$\llbracket x = E\rrbracket = JOIN(v)\setminus\{x\} \cup vars(E)$

$\llbracket\text{var }x_1, \cdots, x_n\rrbracket = JOIN(v)\setminus\{x_1, \cdots, x_n\}$

$\llbracket v\rrbracket = JOIN(v)$



```
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;
```

Least Solution

$\llbracket \texttt{var x,y,z} \rrbracket = \llbracket \texttt{z=input} \rrbracket \setminus \{x,y,z\}$

$\llbracket \texttt{x=input} \rrbracket = \llbracket \texttt{x>1} \rrbracket \setminus \{x\}$

$\llbracket \texttt{x>1} \rrbracket = (\llbracket \texttt{y=x/2} \rrbracket \cup \llbracket \texttt{output x} \rrbracket) \cup \{x\}$

$\llbracket \texttt{y=x/2} \rrbracket = (\llbracket \texttt{y>3} \rrbracket \setminus \{y\}) \cup \{x\}$

$\llbracket \texttt{y>3} \rrbracket = \llbracket \texttt{x=x-y} \rrbracket \cup \llbracket \texttt{z=x-4} \rrbracket \cup \{y\}$

$\llbracket \texttt{x=x-y} \rrbracket = (\llbracket \texttt{z=x-4} \rrbracket \setminus \{x\}) \cup \{x,y\}$

$\underline{\llbracket \texttt{z=x-4} \rrbracket = (\llbracket \texttt{z>0} \rrbracket \setminus \{z\}) \cup \{x\}}$

$\llbracket \texttt{z>0} \rrbracket = \llbracket \texttt{x=x/2} \rrbracket \cup \llbracket \texttt{z=z-1} \rrbracket \cup \{z\}$

$\llbracket \texttt{x=x/2} \rrbracket = (\llbracket \texttt{z=z-1} \rrbracket \setminus \{x\}) \cup \{x\}$

$\llbracket \texttt{z=z-1} \rrbracket = (\llbracket \texttt{x>1} \rrbracket \setminus \{z\}) \cup \{z\}$

$\llbracket \texttt{output x} \rrbracket = \llbracket exit \rrbracket \cup \{x\}$

$\llbracket exit \rrbracket = \varnothing$

$\llbracket entry \rrbracket = \varnothing$

$\llbracket \texttt{var x,y,z} \rrbracket = \varnothing$

$\llbracket \texttt{x=input} \rrbracket = \varnothing$

$\llbracket \texttt{x>1} \rrbracket = \{x\}$

$\llbracket \texttt{y=x/2} \rrbracket = \{x\}$

$\llbracket \texttt{y>3} \rrbracket = \{x,y\}$

$\llbracket \texttt{x=x-y} \rrbracket = \{x,y\}$

$\llbracket \texttt{z=x-4} \rrbracket = \{x\}$

$\llbracket \texttt{z>0} \rrbracket = \{x,z\}$

$\llbracket \texttt{x=x/2} \rrbracket = \{x,z\}$

$\llbracket \texttt{z=z-1} \rrbracket = \{x,z\}$

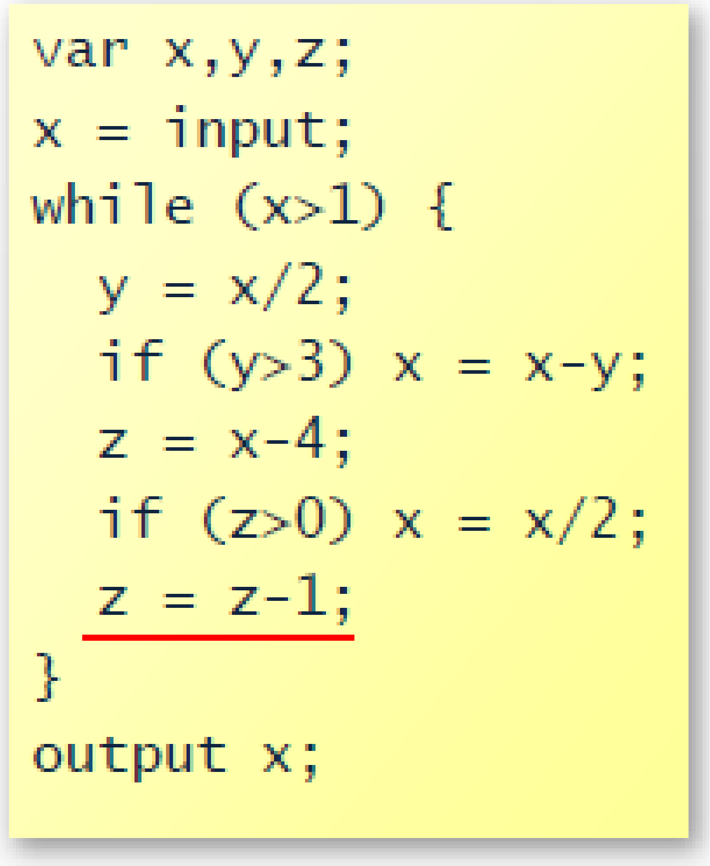
$\llbracket \texttt{output x} \rrbracket = \{x\}$

$\llbracket exit \rrbracket = \varnothing$

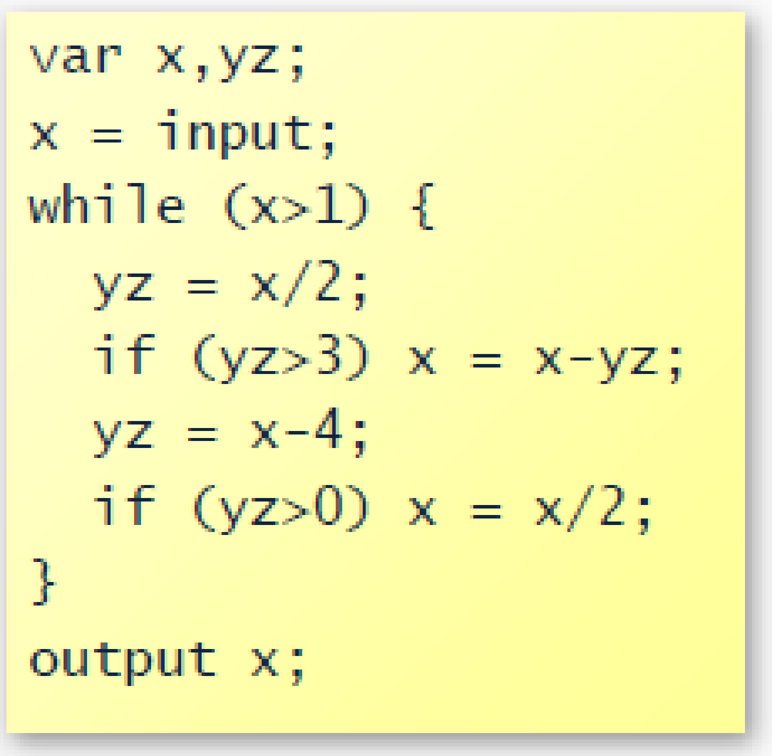
Many non-trivial answers!

Optimizations

- ☐ **Variables y and z are never live at the same time**
  - ➔they can share the same variable location
- ☐ **The value assigned in z=z-1 is never read**
  - ➔the assignment can be skipped



```
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;
```



```
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)

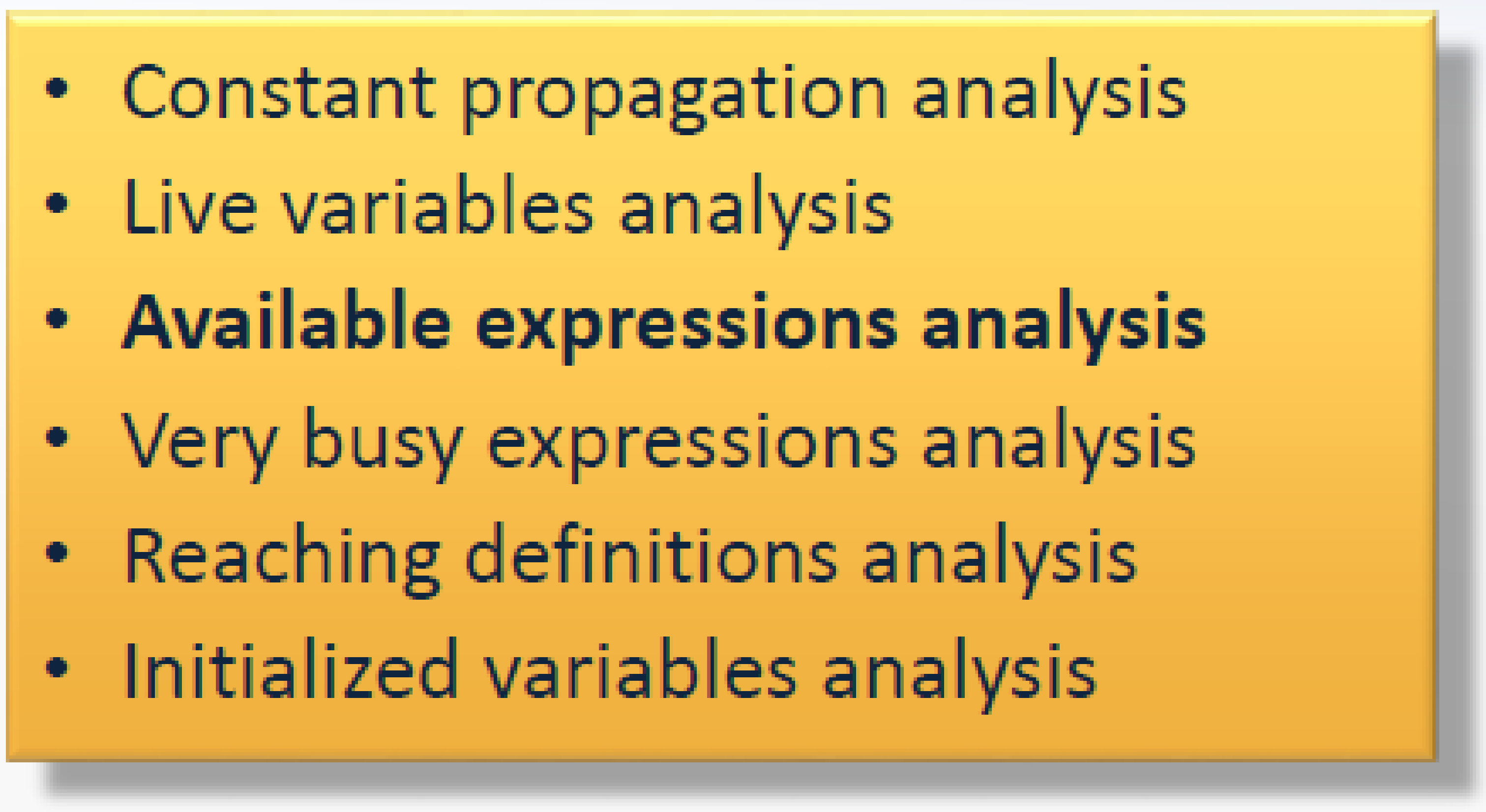
L^n是CFG中n个node要计算的程序点状态的取值的范围

一次迭代的状态转移函数f: $L^n \rightarrow L^n$

- ☐ **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 ∪, \, = 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

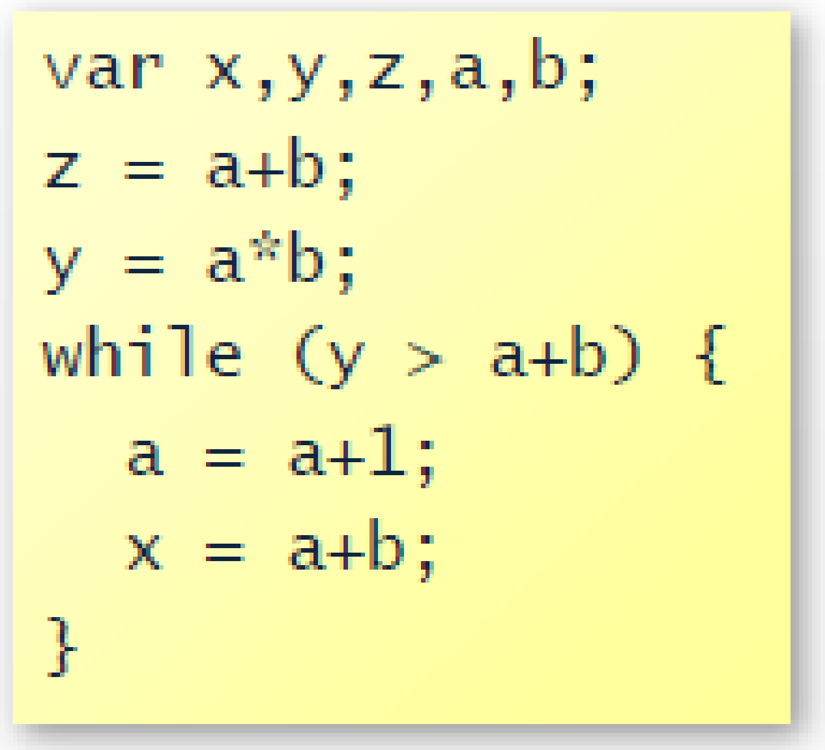
Available Expressions Analysis

- **A (non-trivial) 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

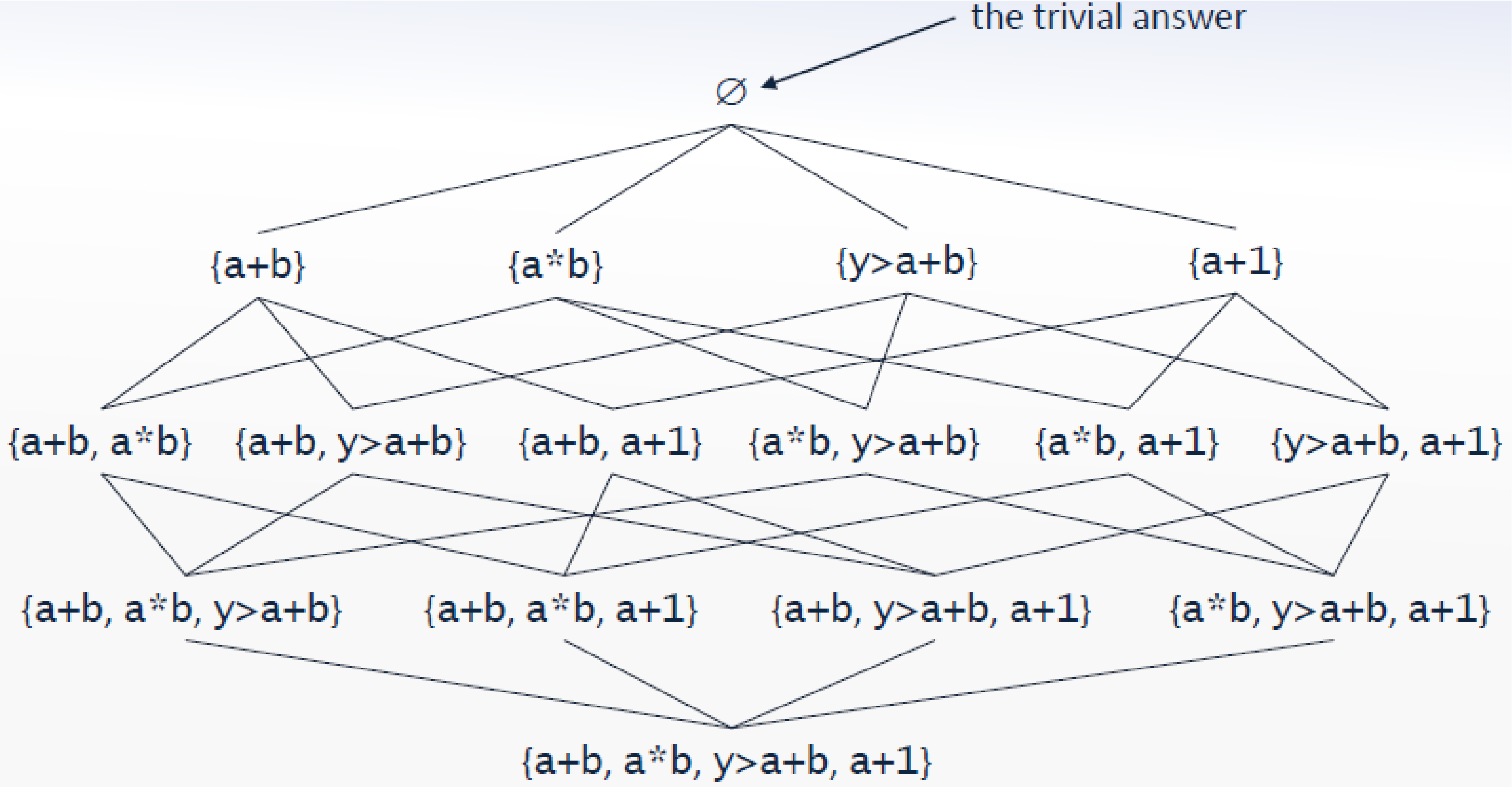
A reverse powerset lattice of nontrivial expressions



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

$$L = (2^{\{a+b,\ a*b,\ y>a+b,\ a+1\}}, \supseteq)$$

Reverse Powerset Lattice



the trivial answer

$\emptyset$

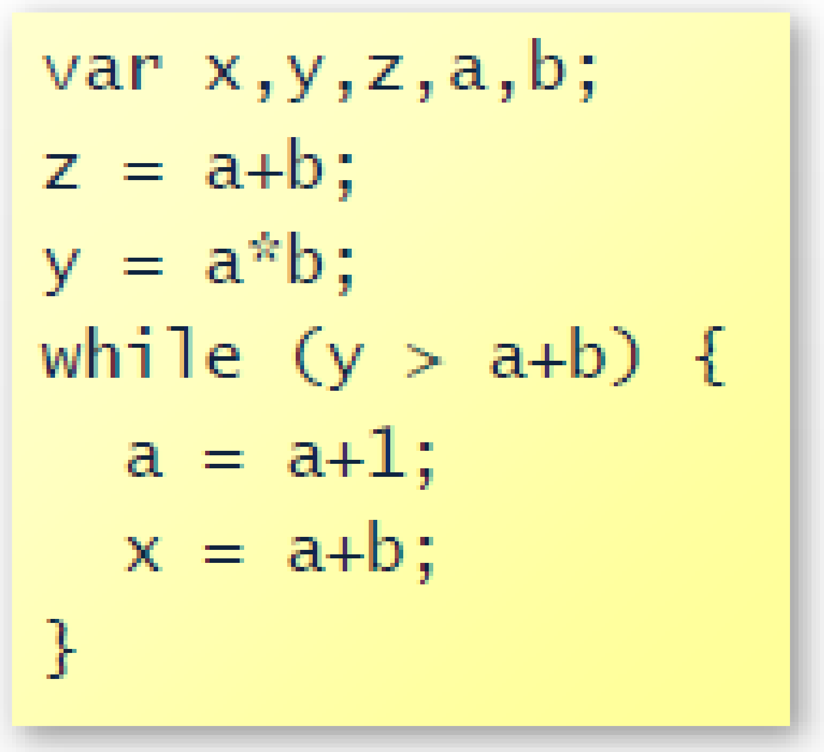
{a+b} {a*b} {y>a+b} {a+1}

{a+b, a*b} {a+b, y>a+b} {a+b, a+1} {a*b, y>a+b} {a*b, a+1} {y>a+b, a+1}

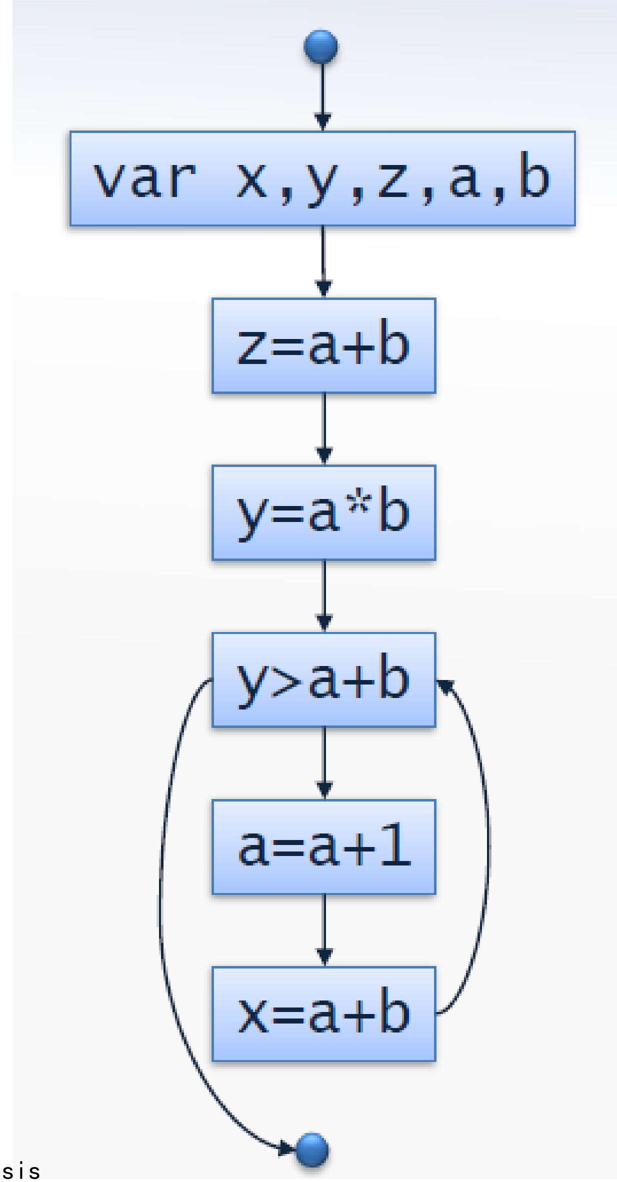
{a+b, a*b, y>a+b} {a+b, a*b, a+1} {a+b, y>a+b, a+1} {a*b, y>a+b, a+1}

{a+b, a*b, y>a+b, a+1}

Flow Graph



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



Setting Up

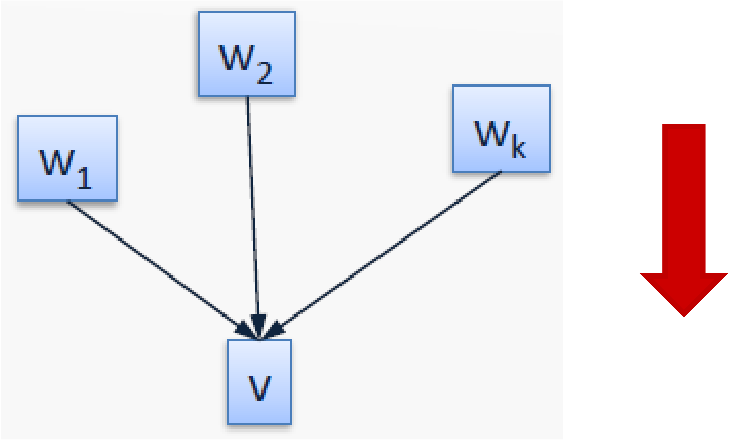
□ **For every CFG node v we have a variable ⟦v⟧**

- the subset of expressions that are available at the program point *after* v

□ **Since the analysis is conservative, the computed set may be *too small***

□ **Auxiliary definition**

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



Auxiliary Functions

- ☐ **The function X↓*x* removes all expressions from X that contain a reference to the variable *x***

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

Availablity Constraints

- ☐ **For the entry node**

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

- ☐ **For conditions and output**

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

- ☐ **For assignments**

  $$[\![x = E]\!] = (\,JOIN(v) \cup exps(E)) \downarrow x$$

- ☐ **For all other nodes**

  $$[\![v]\!] = JOIN(v)$$

Generated Constraints

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

$[\![\text{if } (E)]\!] = [\![\text{while } E]\!] = [\![\text{output } E]\!]$

$= JOIN(v) \cup exps(E)$

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

$[\![v]\!] = JOIN(v)$

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

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

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

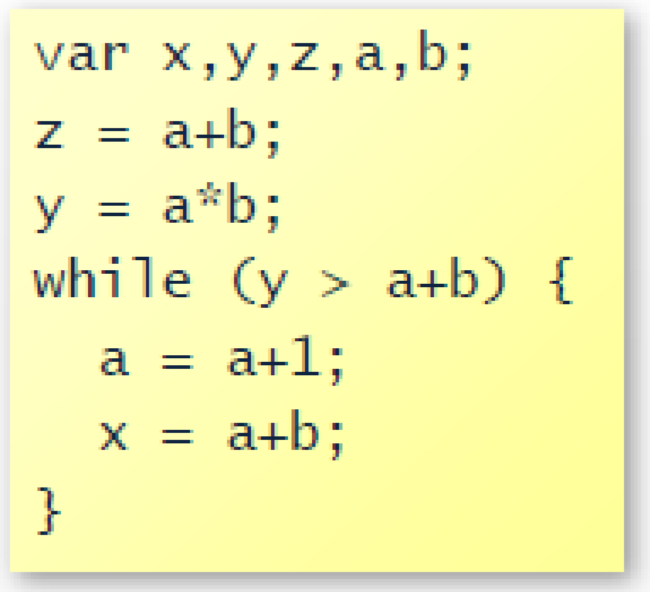
$[\![\text{y=a*b}]\!] = ([\![\text{z=a+b}]\!] \cup exps(\text{a*b})) \downarrow \text{y}$

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

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

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

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



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

Least Solution

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

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

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

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

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

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

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

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

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

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

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

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

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

$[\![\mathrm{a{=}a{+}1}]\!] = \emptyset$

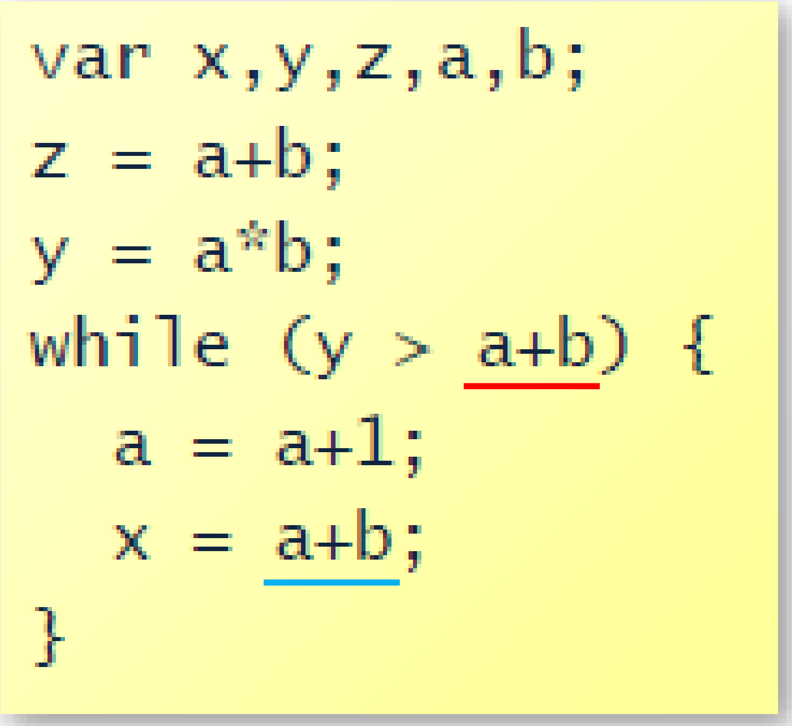
$[\![\mathrm{x{=}a{+}b}]\!] = \{\mathrm{a{+}b}\}$

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

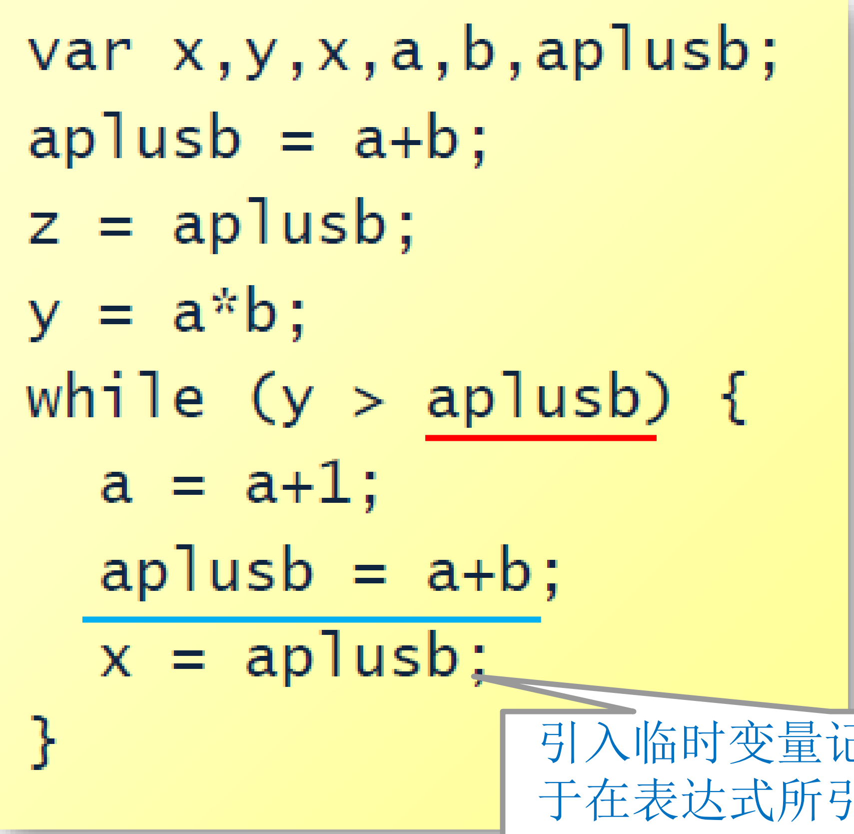
Many non-trivial answers!

Optimizations

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



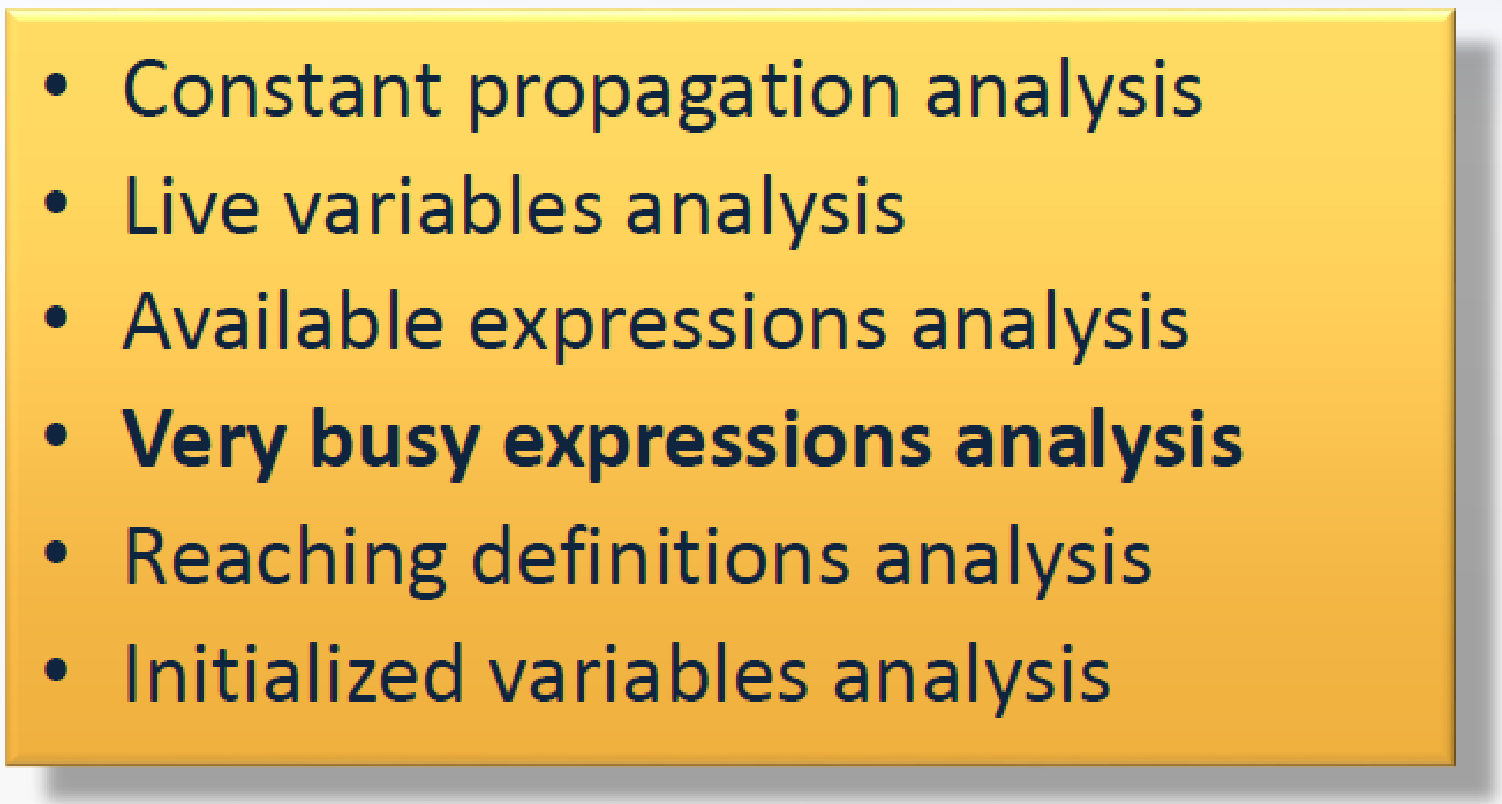
```
var x,y,z,a,b;
z = a+b;
y = a*b;
while (y > a+b) {
   a = a+1;
   x = a+b;
}
```



```
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;
}
```

引入临时变量记录表达式的值，便于在表达式所引用的变量修改后重新计算并记录新值，也便于后面实施复写传播，发现更多优化机会

Agenda



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

Very Busy Expressions Analysis

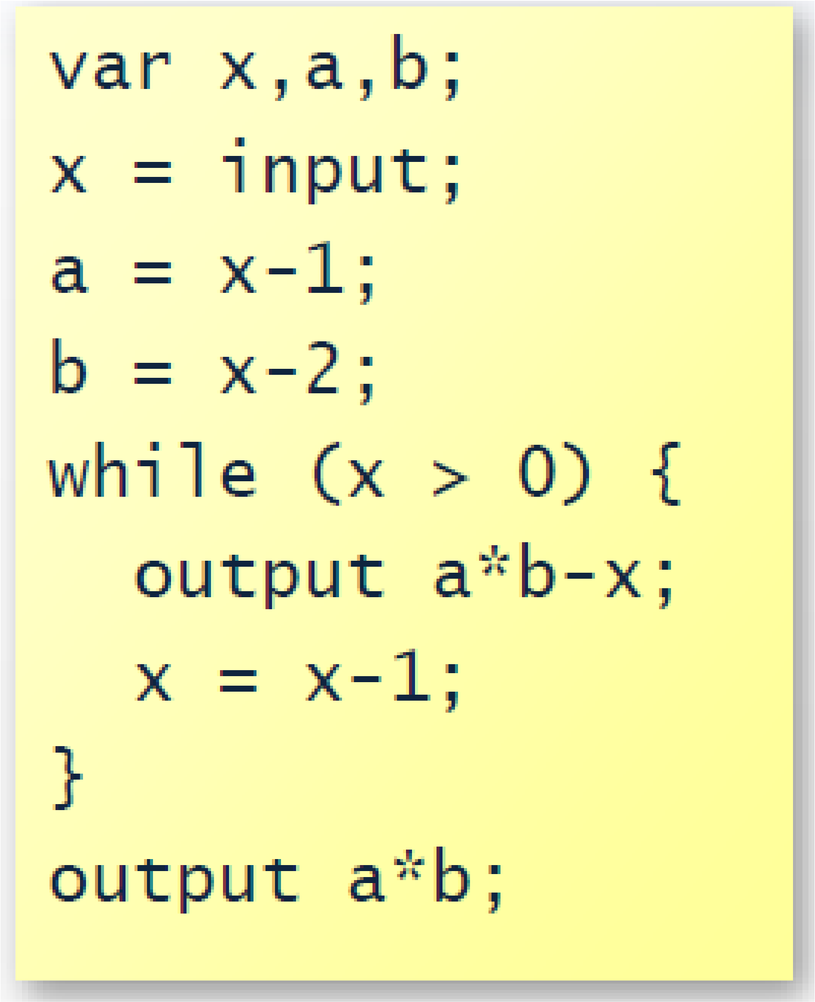
- ☐ **A (nontrivial) expression is *very busy* if it will definitely be evaluated before its value changes**

一个表达式在程序点非常忙当它无论沿哪条路径从那个点到终止点都会被计算

- ☐ **The approximation generally includes *too few* expressions**
  - the answer “*verybusy*” must be the true one
  - Very busy expressions may be pre-computed (e.g. loop hoisting)

- ☐ **Same lattice as for available expressions**

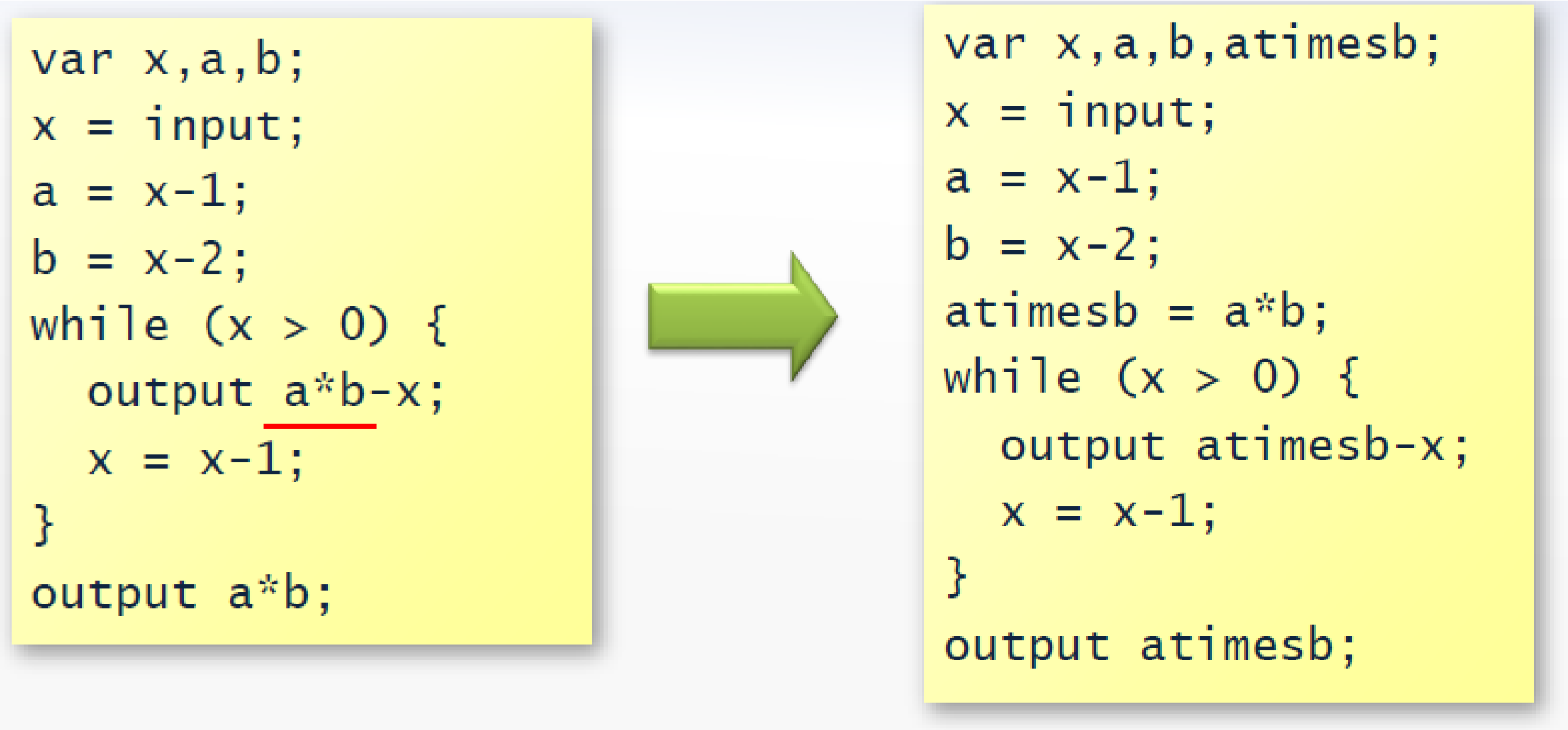
An Example Program



```
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

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;
```

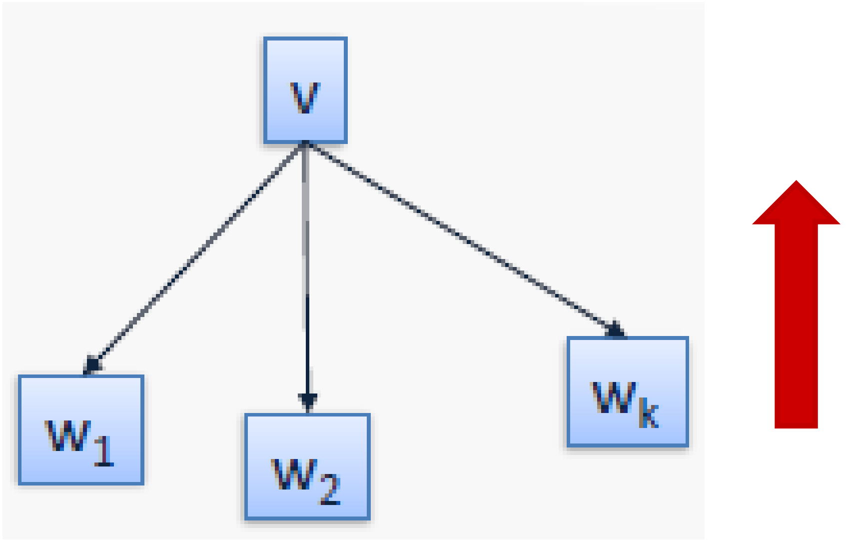
Setting Up

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

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

必须其后的每条路径上都very busy才能称为very busy

- ☐ **Auxiliary definition**
  - $\text{JOIN}(v) = \bigcap_{w \in succ(v)} [\![w]\!]$



Very Busy Constraints

- ☐ **For the exit node**

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

- ☐ **For conditions and output**

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

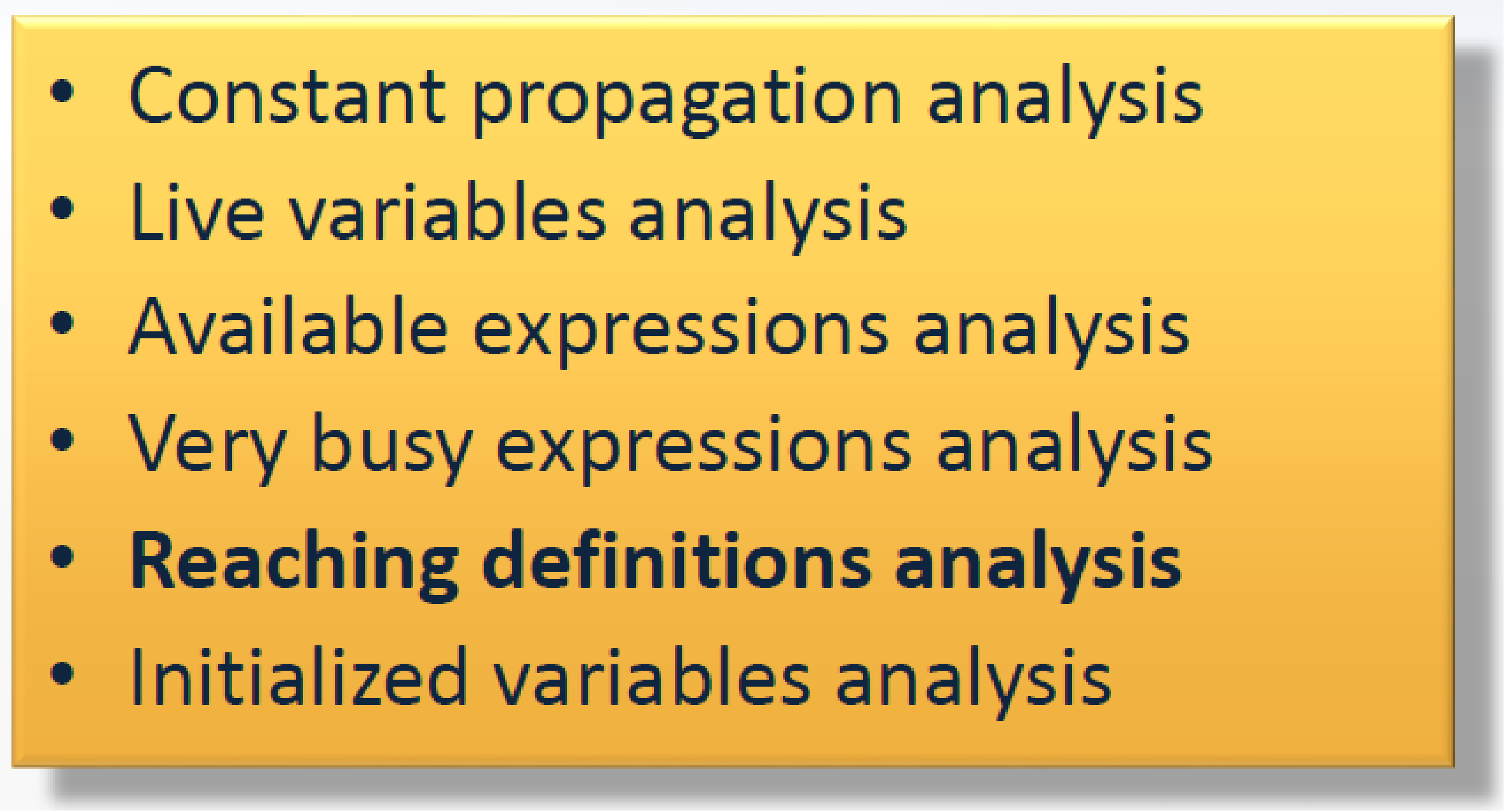
- ☐ **For assignments**

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

- ☐ **For all other nodes**

  $[\![v]\!] = JOIN(v)$

Agenda



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

Reaching Definitions Analysis

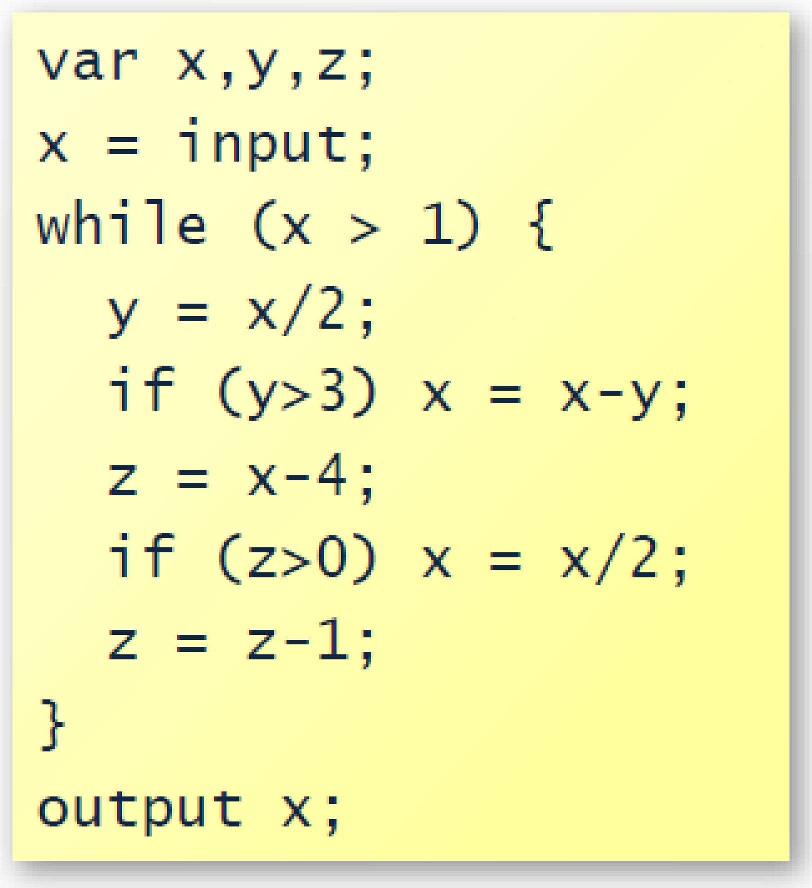
- The ***reaching definitions*** for a program point are those assignments that may define the current values of variables

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

A Lattice for Reaching Definitions

The powerset lattice of assignments

$$L = (2^{\{x=\text{input},\, y=x/2,\, x=x-y,\, z=x-4,\, x=x/2,\, z=z-1\}}, \subseteq)$$



```
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

- ☐ The function $X\downarrow x$ removes assignments to *x* from *X*

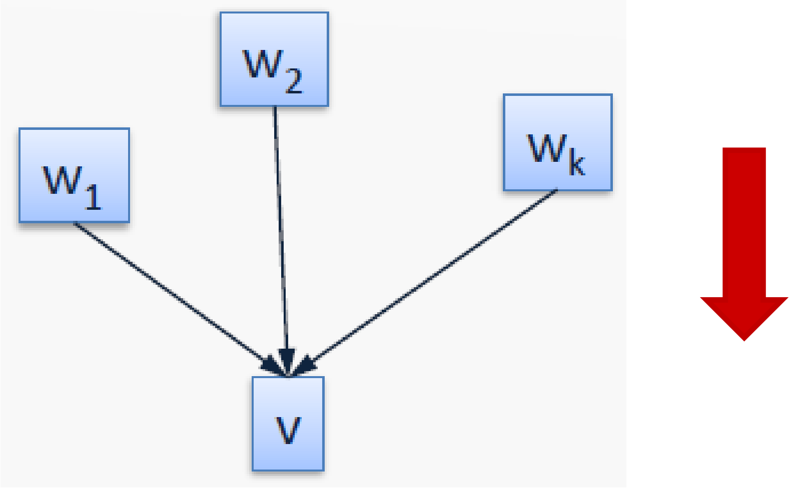
- ☐ For assignments

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

- ☐ For all other nodes

$$[\![v]\!] = JOIN(v)$$

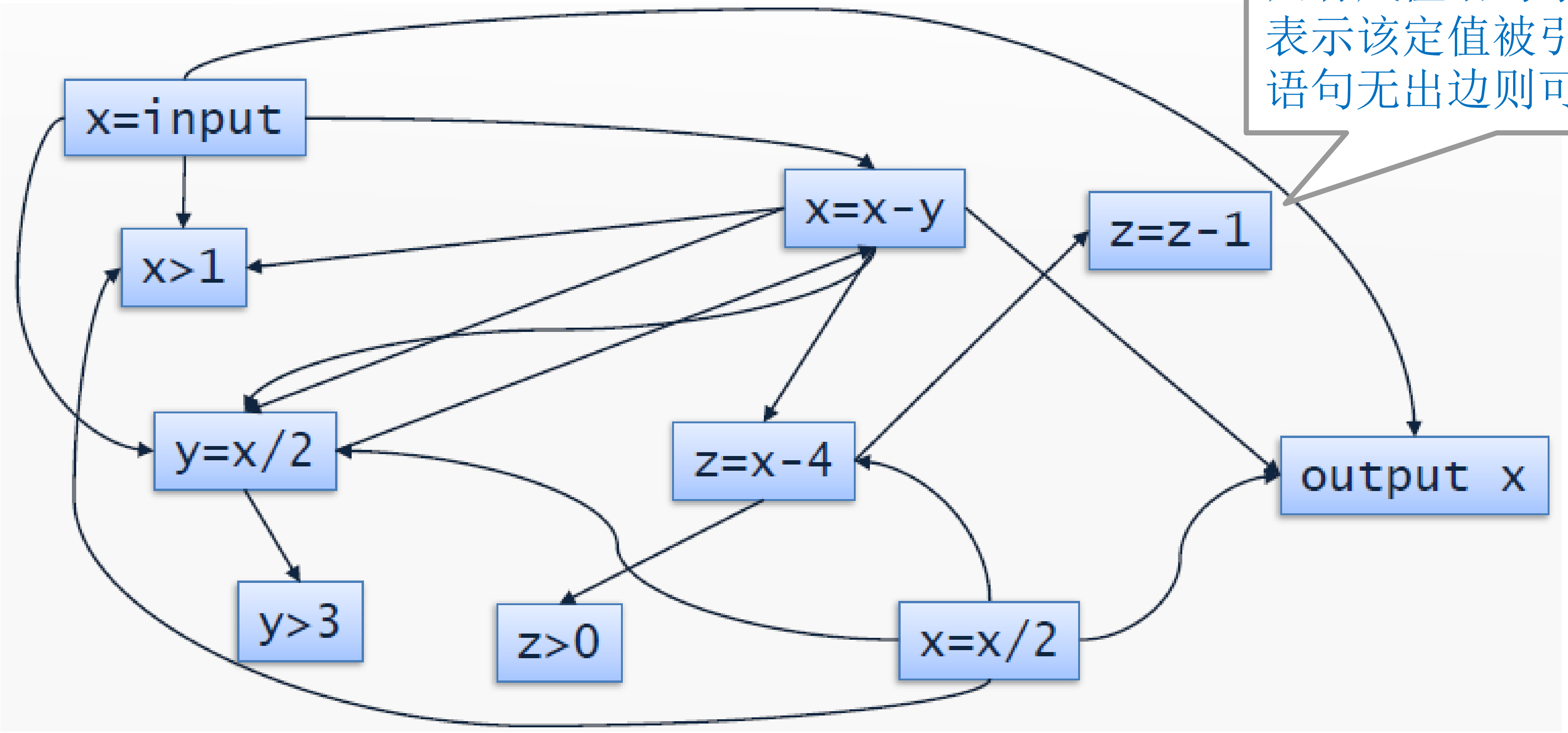
- ☐ Auxiliary definition
  - $\mathrm{JOIN}(v) = \bigcup_{w \in pred(v)} [\![w]\!]$



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*



只有赋值语句才可能有出边，表示该定值被引用；若定值语句无出边则可删除

Forward vs. Backward

- **A *forward* analysis:**
  - computes information about the *past* behavior
  - examples: available expressions, reaching definitions

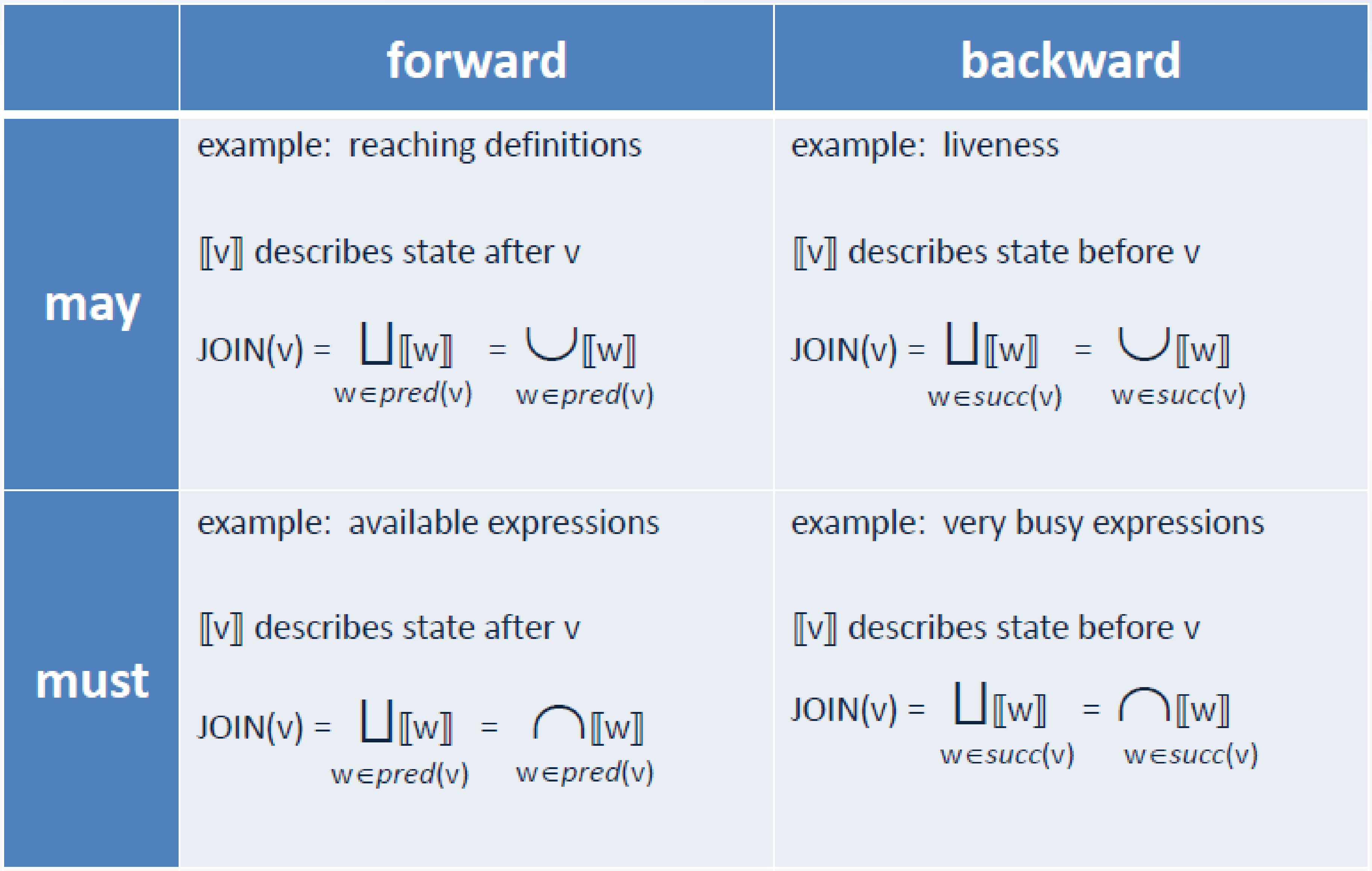
- **A *backward* analysis:**
  - computes information about the *future* behavior
  - examples: liveness, very busy expressions

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 $[\![v]\!]$ describes state after v $\mathrm{JOIN}(v) = \bigsqcup_{w\in pred(v)} [\![w]\!] = \bigcup_{w\in pred(v)} [\![w]\!]$	example: liveness $[\![v]\!]$ describes state before v $\mathrm{JOIN}(v) = \bigsqcup_{w\in succ(v)} [\![w]\!] = \bigcup_{w\in succ(v)} [\![w]\!]$
must	example: available expressions $[\![v]\!]$ describes state after v $\mathrm{JOIN}(v) = \bigsqcup_{w\in pred(v)} [\![w]\!] = \bigcap_{w\in pred(v)} [\![w]\!]$	example: very busy expressions $[\![v]\!]$ describes state before v $\mathrm{JOIN}(v) = \bigsqcup_{w\in succ(v)} [\![w]\!] = \bigcap_{w\in succ(v)} [\![w]\!]$

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#)
- ☐ ➔*forward must analysis*
- ☐ Reverse powerset lattice of all variables

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

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

Thanks