POLITECNICO
MILANO 1863

Software Engineering 2

V&V terminology

Static vs Dynamic Analysis

Data Flow Analysis

M Camilli, E Di Nitto, M Rossi

Verification & Validation

Terminology

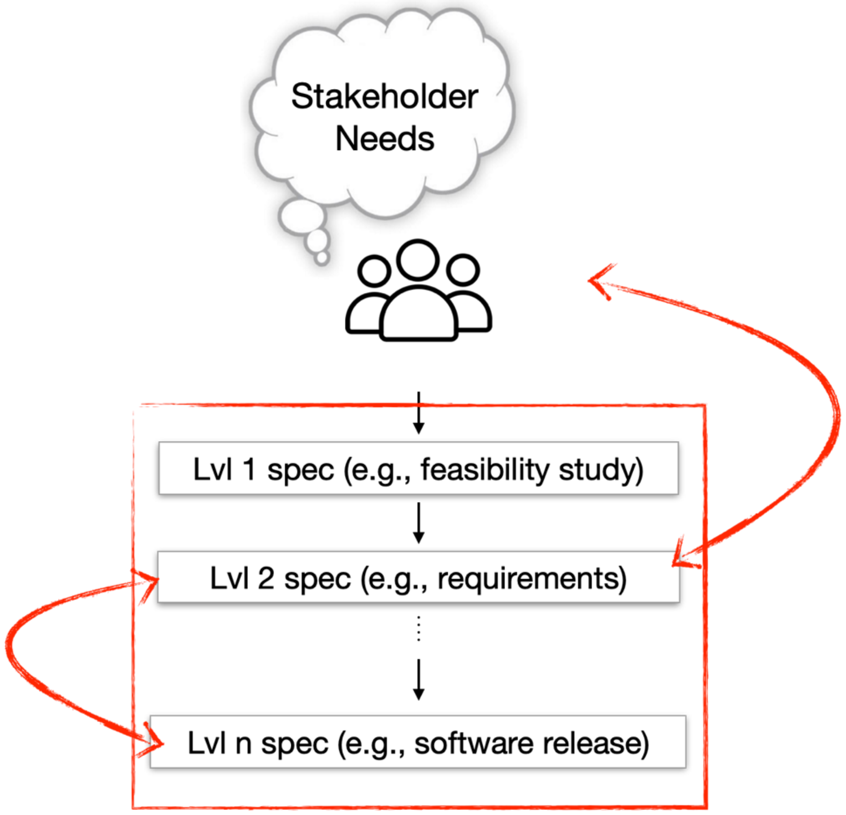
Verification & Validation

- Verification is internal
  - *Are we building the software right (w.r.t. a specification)?*
- Validation is external
  - *Are we building the right software (w.r.t. stakeholder needs)?*

Verification & Validation

Verification
"is level i consistent with level i+1?"

internal consistency check



(abstract) process

Validation
"does level i conform to needs?"

external consistency check

Quality Assurance (QA)

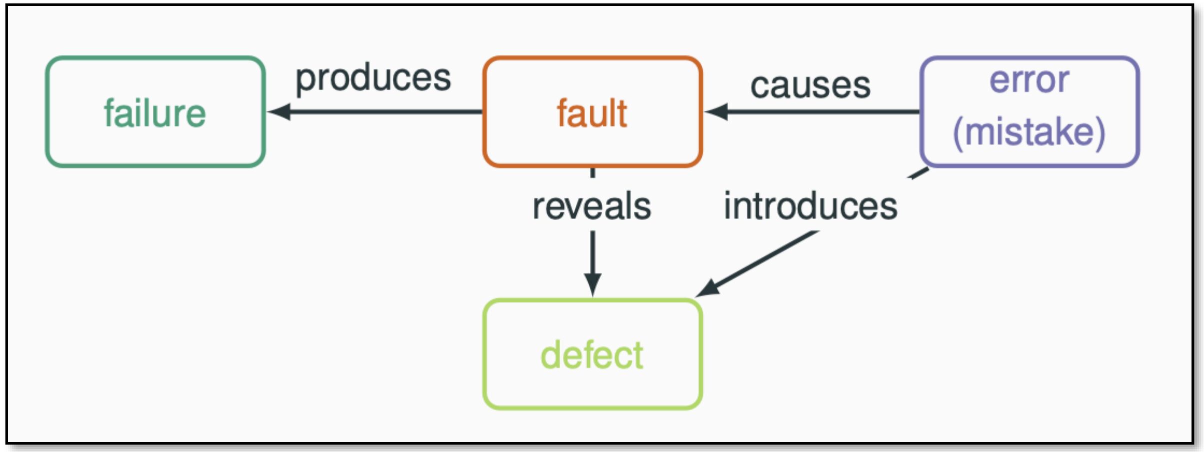
POLITECNICO MILANO 1863

- Define policies and processes to achieve **quality**
- Assess quality and find **defects** (through V&V techniques)
- Improve quality

- Quality = general term, may refer to
  - Absence of defects (or bugs)
  - Absence of other issues that prevent the fulfilment of non-functional requirements or the degradation of some software qualities
    - external qualities (e.g., performance, availability) or internal ones (e.g., maintainability)

Terminology

- Classification from the IEEE Computer Society



- **Failure**: (A) Termination of the ability of a product to perform a required function or its inability to perform within previously specified limits. (B) An event in which a system or system component does not perform a required function within specified limits.
- **Fault**: A manifestation of a defect
- **Defect**: an imperfection or deficiency in a program (e.g., function should always return a positive value, but returns a negative value in this case)
- **Error**: human action that introduced an incorrect result (any programming mistakes)

"IEEE Standard Classification for Software Anomalies," in IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993) , vol., no., pp.1-23, 7 Jan. 2010, doi: 10.1109/IEEESTD.2010.5399061.

Windows

An error has occurred. To continue:

Press Enter to return to Windows, or

Press CTRL+ALT+DEL to restart your computer. If you do this, you will lose any unsaved information in all open applications.

Error: 0E : 016F : BFF9B3D4

Press any key to continue _

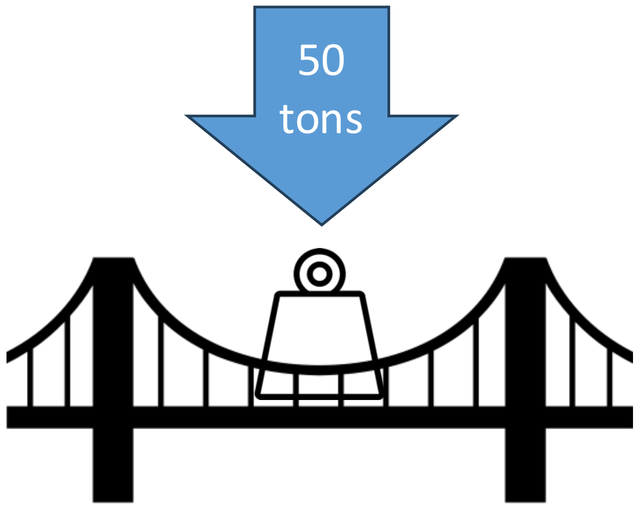
Failure? Fault? Defect? Error?

QA challenges

- **Zero defect** software practically impossible to achieve, so…
  - Careful and continuous QA needed
  - Ideally, every artifact shall be subject of QA (spec documents, design documents, test data, …)
    - even the verification artifacts must be verified!
- QA along the entire development process, not just at the end
- … in this course, focus on **verification** and not on validation

Verification in engineering disciplines

- Structural engineering (bridge)
  - Requirement example: the bridge shall support heavy trucks (40 tons)
  - Test example: load the bridge with 50 tons
    - One test covers infinite cases



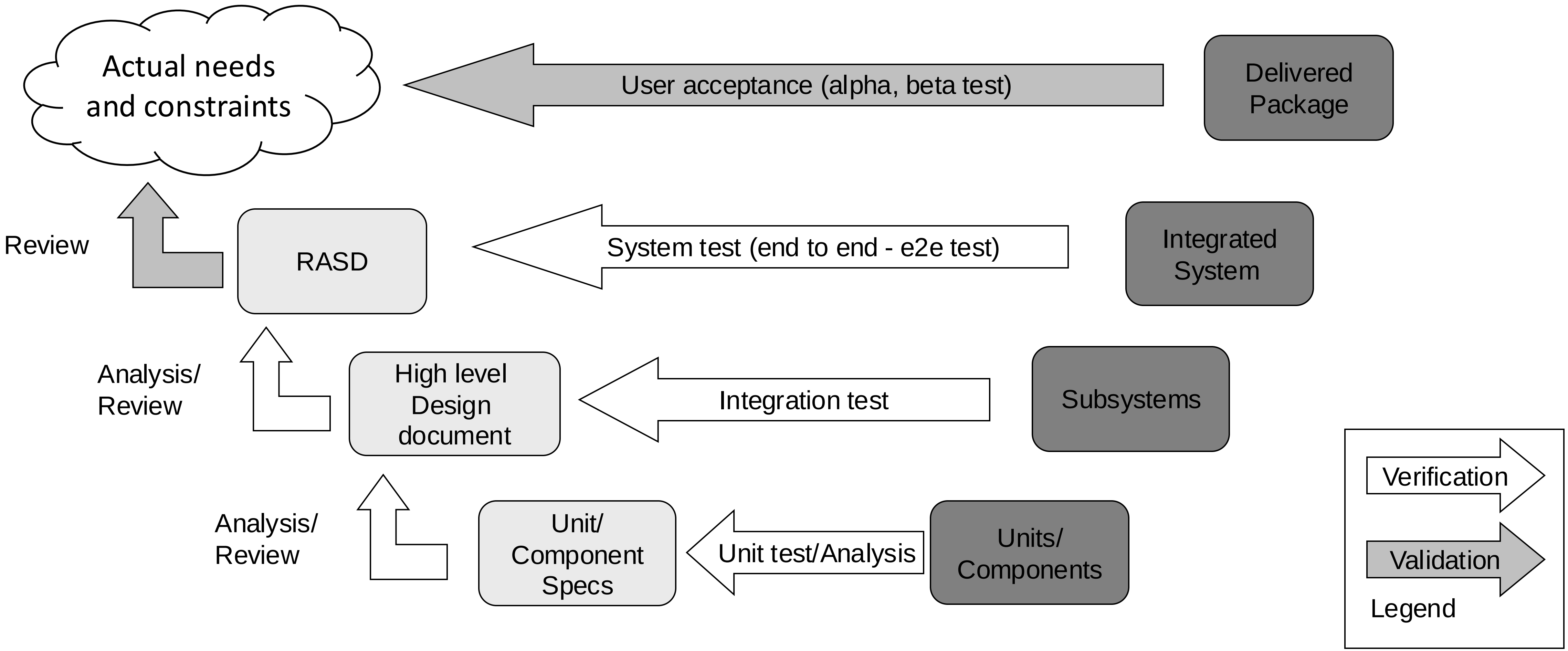
Verification in engineering disciplines

- Software engineering (program)
  - Programs do not display a “continuous” behavior
  - Verifying a program for a single data point does not tell us anything about other points
  - Example:

```
…
a =  y / (x + 20) …
…
```

  - Any value of *x* is ok but one (*x* = -20)

Verification at which level? (V model)



Main approaches: static vs dynamic analysis

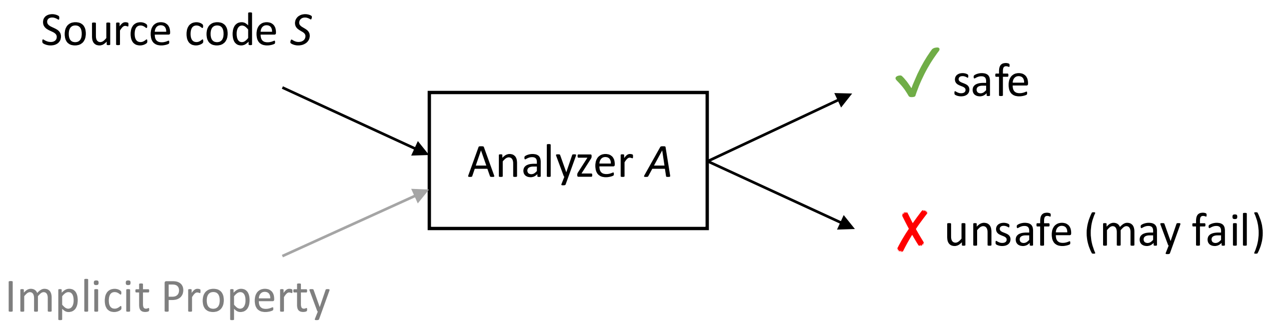
- Static Analysis
  - Done on source code without execution
  - Analysis is static but properties are dynamic
- Testing (dynamic analysis)
  - Done by executing the sources (usually by sampling)
  - Analysis of the actual behavior compared to an expected one

Static Analysis

Introduction

Static Analysis

- The very idea
  - Analyzes the source code
  - Each analyzer targets a fixed set of **hard-coded** (pre-defined, not custom) properties
  - Completely **automatic**
  - The output reports
    - **Safe** = no issues
    - **Unsafe** = potential issues



Static Analysis: properties

- Checked properties are often general safety properties (absence of certain conditions that may yield errors)
- Examples:
  - No overflow for integer variables
  - No type errors
  - No null-pointer dereferencing
  - No out-of-bound array accesses
  - No race conditions
  - No useless assignments
  - No usage of undefined variables

Static analysis: succesful stories

[2017] *"Our strategy at Uber has been to use static code analysis tools to prevent null pointer exception crashes."*

Engineering NullAway, Uber's Open Source Tool for Detecting NullPointerExceptions on Android
https://www.uber.com/en-IT/blog/nullaway/

[2013] *"Each month, hundreds of potential bugs identified by Facebook Infer are fixed [. . .] before they are [. . .] deployed to people's phones."*

Facebook buys code-checking Silicon Roundabout startup Monoidics
https://www.theguardian.com/technology/2013/jul/18/facebook-buys-monoidics

More on Static vs Dynamic

POLITECNICO
MILANO 1863

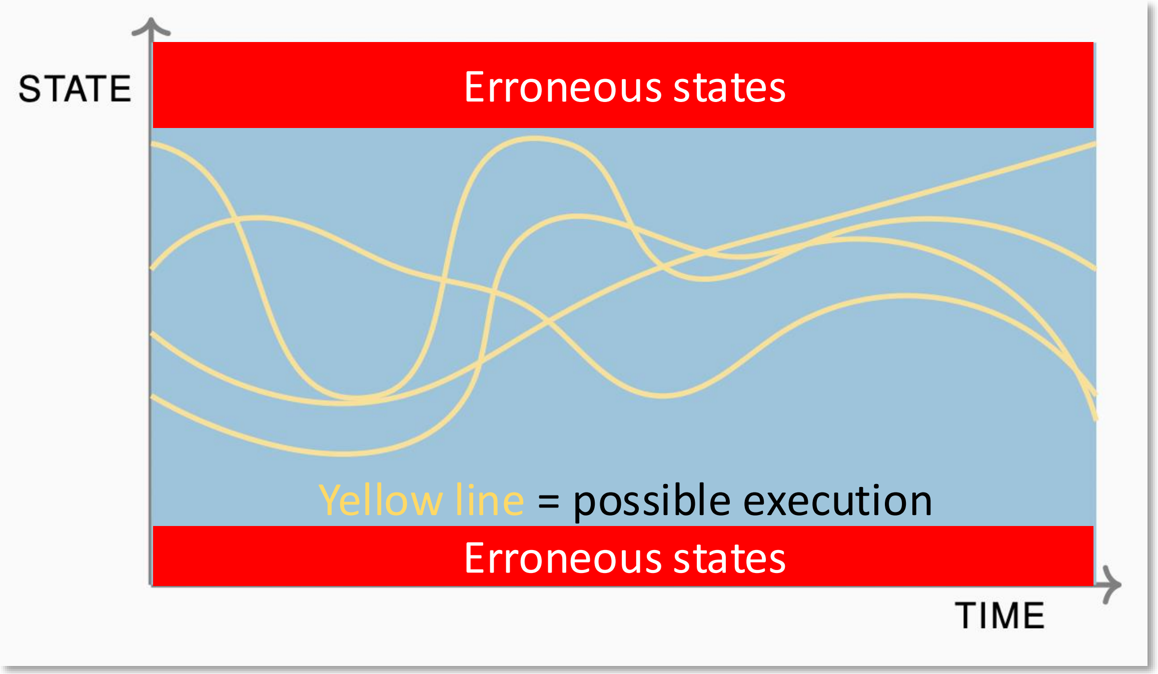
- Static
  - at compile time – before execution
  - related to source code (or any other model of the software)
  - without execution of the software
  - on generic (or symbolic) inputs

- Dynamic
  - at runtime – during execution
  - related to software behavior
  - while executing the software
  - on specific inputs

- Static analysis: techniques, methods, tools used to infer properties of the dynamic behavior without explicitly running the software
  - As such, properties may (or may not) hold at runtime…
  - ***What does this mean?***

Static analysis and program behavior

- Program behavior: all possible executions as sequences of states
- Static analysis allows us to find erroneous states
  - … but program behavior may not reach any erroneous state
- Thus, static analysis is pessimistic!



Precision/efficiency trade-off

- Static analysis is based on over-approximations to be sound
- Degree of precision is often traded-off against efficiency
  - Perfect precision is often impossible due to undecidability
  - High precision may still be too computationally expensive
  - Low precision is cheaper but leads to many false positives that must be verified manually

- Designing a static analysis technique requires to balance precision and efficiency in a way that is practical

Static Analysis

Data-flow analysis

Preliminaries: Control-Flow Graphs (CFG)

POLITECNICO MILANO 1863

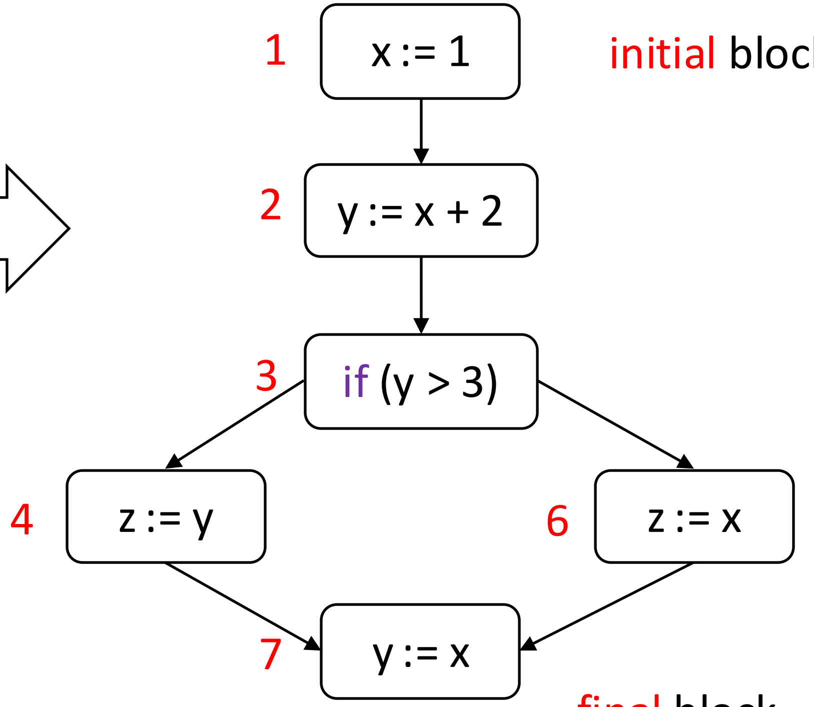
- CFG is a directed graph representing possible execution paths
  - CFG block = program statement
  - CFG edge = connects two consecutive statements
  - We ignore declarations since they do not affect the program state

CFG: example

- We label statements to use precise references

Example of source S (pseudocode)

```
1: x := 1
2: y := x + 2
3: if (y > 3)
4:    z := y
5: else
6:    z := x
7: y := x
```



initial block = execution begins

final block = execution terminates

Data-flow analysis

- Works on the Control-Flow Graph (CFG) of a program
- Extracts information about the data flow:
  - What values are read (used) and written (defined)

CFG S → Data-flow analyzer A →

Line 1: $L_1 = \{x, y\}$
Line 2: $L_2 = \{z\}$
…
Line n: $L_n = \{\ \}$

output

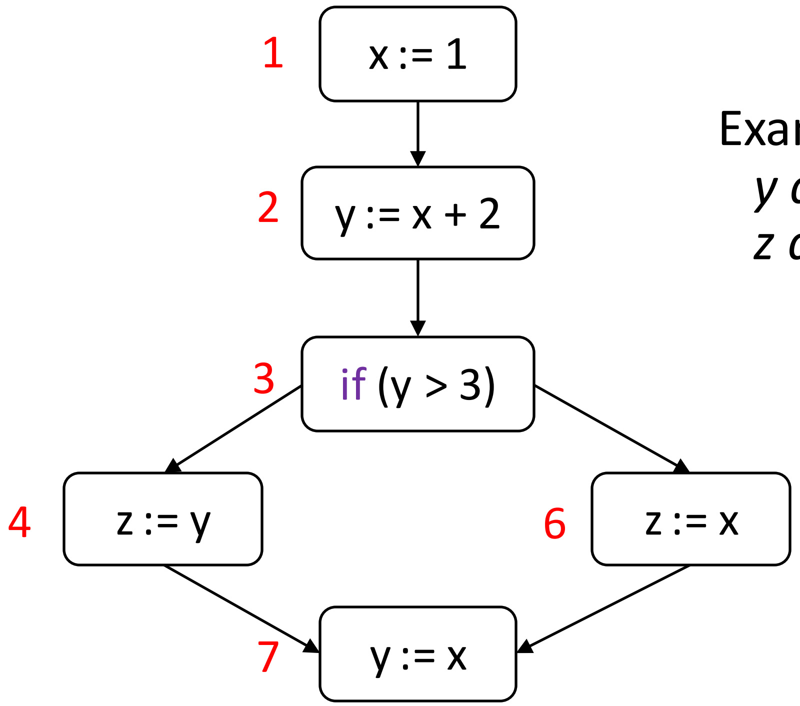
- Properties can be checked by analyzing the output
- Example: live variable analysis — is there a useless variable assignment in the code?

Data-flow Analysis

Live variables

Live variables

- Given a CFG, a variable *v* is live at the exit of a block *b* if there is some path (on the CFG) from block *b* to a use of *v* that does not redefine *v*.



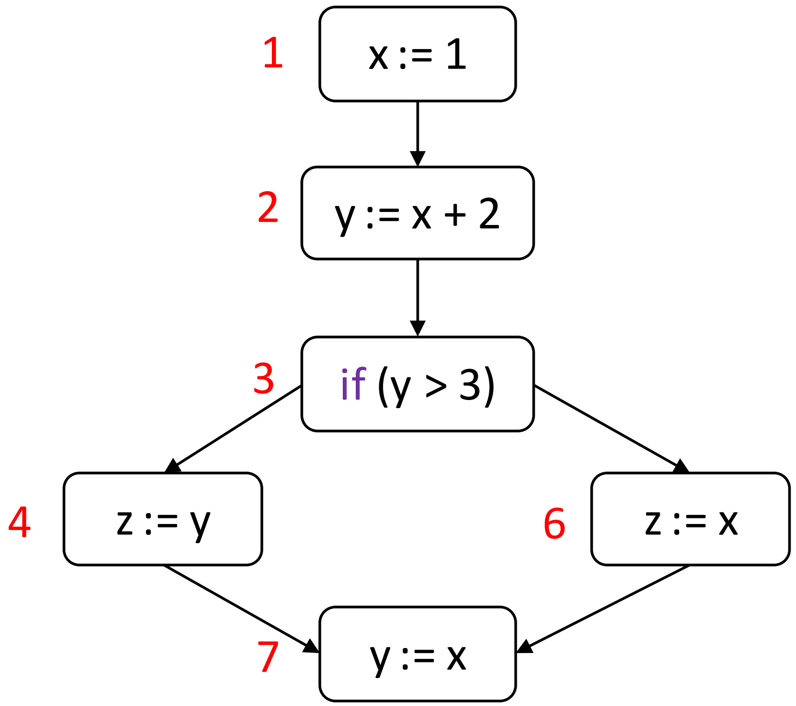
Examples

y at 2? *LIVE*

z at 4? *NOT LIVE*

Live variables

- Live variable analysis: for each block determine which variables may be live (at the exit of the block)
  - Output
    - LV(1) = {x}
      LV(2) = {x, y}
      LV(3) = {x, y}
      LV(4) = {x}
      LV(6) = {x}
      LV(7) = { }



Live variables

- Live variable analysis: for each block determine which variables may be live (at the exit of the block)
  - Note “may be live” = over-approximation
    - LV(k) is a superset of the live variables at k
    - If $x \notin LV(k)$ → definitely not live at k
    - If $x \in LV(k)$ → still, may not be live at k (e.g., live along certain paths only)

Live variables: applications

- Dead assignment elimination
  - If a variable is not live after it is defined by an assignment, the assignment is useless and can be removed without changing the program behavior

- Any block *k* s.t.
  - *k* (re)defines a variable *v*
  - *v* is not live at *k*, $v \notin LV(k)$
- → *k* can be eliminated without affecting the behavior

x not live here (useless assignment)

x may be live here
(possibly useful assignment)

```
x := 4
x := 7
if (z > y)
    y := x
```

Live variables analysis

- We did it manually, but…
- Live variable analysis reduces to an equation system
- The solution of the equation system identifies live variables
  - Can be computed automatically using standard algorithms
    - Fixed point, worklist algorithm, …

Data-flow Analysis

Reaching definitions analysis

Reaching definitions

- A definition (v,k) is an assignment to variable v occurring at block k
- A definition (v,k) reaches block r if there is a path from k to r that does not redefine v

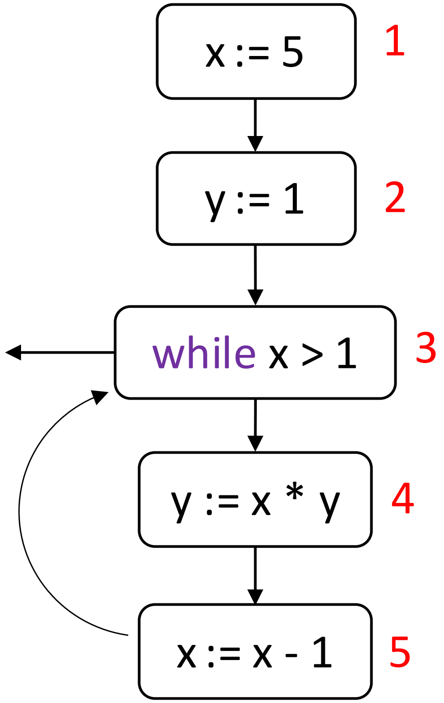
Example: Which definition(s) reach the entry of block 5?

```
1: x := 5
2: y := 1
3: while (x > 1)
4:   y := x * y
5:   x := x - 1
```

- 1^{st} loop iteration: (x, 1) and (y, 4)
- Following iterations: (y, 4) and (x, 5)

Reaching definitions analysis

- Reaching definition analysis: for each block, determine which definitions may reach the block



Reaching definitions analysis output:

$$RD_{IN}(5) = RD_{OUT}(4) = \{ (x,1), (x,5), (y,4) \}$$

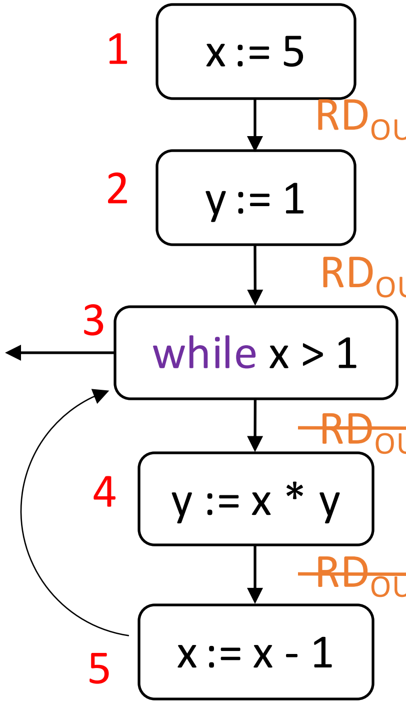
Note: this analysis results in an over-approximation since RD(*k*) is a superset of the reaching definitions at *k*

- if $(x,h) \in RD_{IN}(k)$ → def of *x* at *h* may (or may not) reach *k* (e.g., may be overwritten along certain paths and not along others)
- if $(x,h) \notin RD_{IN}(k)$ → def of *x* at *h* definitely does not reach *k*

Reaching definitions analysis

- Working forward: record the reaching definitions at the entry and exit of every block

$RD_{IN}(1) = \{ (x,?), (y,?)\}$ Implicit initialization



$RD_{OUT}(1) = \{ (x,1), (y,?)\}$ $RD_{IN}(2)$ same as $RD_{OUT}(1)$ in this case

$RD_{OUT}(2) = \{ (x,1), (y,2)\}$

~~$RD_{OUT}(3) = \{ (x,1), (y,2) \}$~~ → $RD_{OUT}(3) = \{ (x,1), (y,2), (y,4), (x,5)\}$

~~$RD_{OUT}(4) = \{ (x,1), (y,4) \}$~~ → $RD_{OUT}(4) = \{ (x,1), (y,4), (x,5)\}$

$RD_{OUT}(5) = \{ (y,4), (x,5) \}$

Reaching definitions analysis

- We can define the following equations
- For each block k:
  - $RD_{IN}(k) = \bigcup_{h \rightarrow k} (RD_{OUT}(h))$
  - $RD_{OUT}(k) = (RD_{IN}(k) \setminus kill_{RD}(k)) \cup gen_{RD}(k)$

For all block h s.t. h prev k

$kill_{RD}(k)$= other definitions of same variables redefined at k

$gen_{RD}(k)$= variables defined at k

Use-def and def-use chains

- Applications: information about which statements define values and which use them is useful for program optimizations (e.g., parallelization of multiple uses with no def) or avoid potential errors (e.g., use with no def)

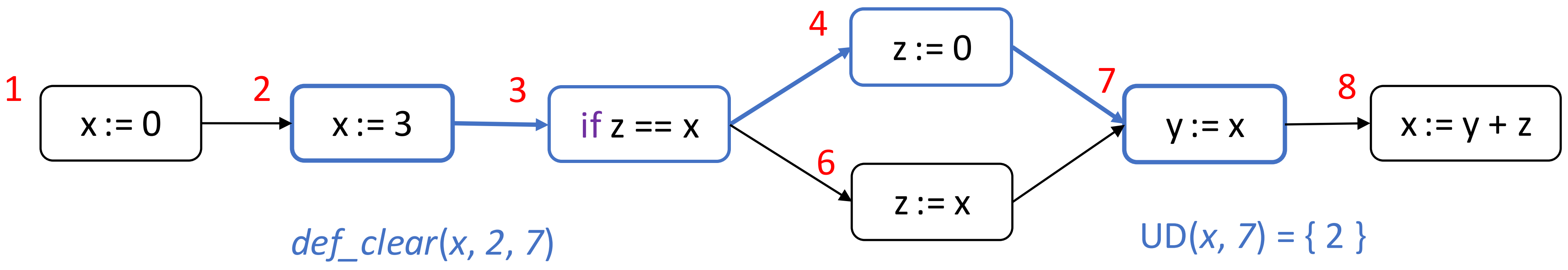
```
1: x := 0
2: x := 3
3: if (z == x)
4:    z := 0
5: else
6:    z := x
7: y := x
8: x := y + z
```

- Use-def (UD) chains: link from use to all def that may reach it
  - Example 1: UD chain for *x* at block 7
- Def-use (DU) chains: link from def to all use s.t. def may reach them
  - Example 2: DU chain for *x* at block 2

UD chains

Location of implicit initialization

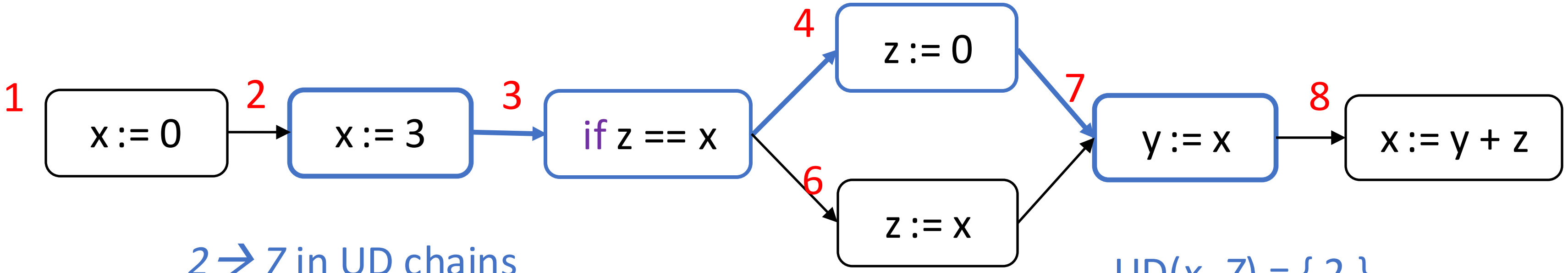
- Link from use to all def that may reach it
  - UD(v, k) = { q | "q: v := E" and *def_clear*(v, q, k) } ∪ { ? | *def_clear*(v, ?, k) }
    - "q: v := E" assignment for v at line q
    - *def_clear*(v, q, k) holds iff there is a definition-clear path from q to k (i.e., no block between q and k redefines v)



UD chains

Location of implicit initialization

- Link from use to all def that may reach it
  - $UD(v, k) = \{ q \mid \text{"}q\text{: } v := E\text{"} \text{ and } def\_clear(v, q, k) \} \cup \{ ? \mid def\_clear(v, ?, k) \}$
    - "q: v := E" assignment for v at line q
    - *def_clear*(v, q, k) holds iff there is a definition-clear path from q to k (i.e., no block between q and k redefines v)
- Definition: all $q \rightarrow k$ s.t.
  - k uses some v
  - $q \in UD(v, k)$



2 → 7 in UD chains

UD(x, 7) = { 2 }

UD chains from reaching definitions

- Set UD(v, k) can be computed from reaching definitions analysis

$$UD(v, k) = \begin{cases} \{ q \mid (v, q) \in RD_{IN}(k) \} & \text{if } v \text{ used in block } k \\ \{\ \} & \text{otherwise} \end{cases}$$

$RD_{IN}(1) = \{ (x,?), (y,?), (z,?) \}$

$RD_{IN}(2) = RD_{OUT}(1) = \{ (x,1), (y,?), (z,?) \}$

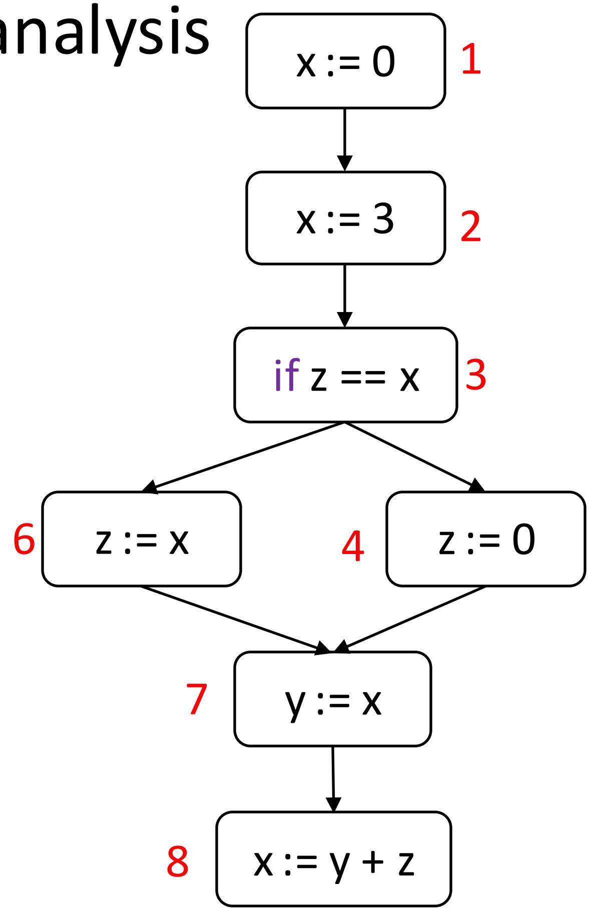
$RD_{IN}(3) = RD_{OUT}(2) = \{ (x,2), (y,?), (z,?) \}$ => UD(x, 3) = { 2 }, **UD(z, 3) = {?}**

$RD_{IN}(4) = RD_{OUT}(3) = \{ (x,2), (y,?), (z,?) \}$

$RD_{IN}(6) = RD_{OUT}(3) = \{ (x,2), (y,?), (z,?) \}$ => UD(x, 6) = { 2 }

$RD_{IN}(7) = RD_{OUT}(4) \cup RD_{OUT}(6) = \{ (x,2), (y,?), (z,4), (z,6) \}$ => UD(x, 7) = { 2 }

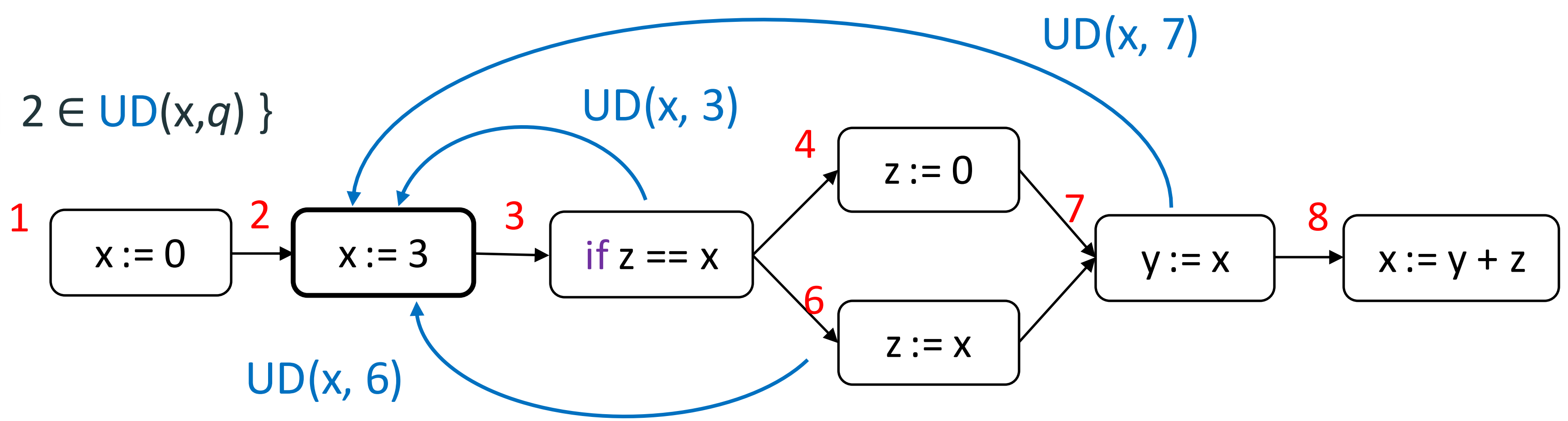
$RD_{IN}(8) = RD_{OUT}(7) = \{ (x,2), (y,7), (z,4), (z,6) \}$ => UD(z, 8) = { 4, 6 }, UD(y, 8) = {7}



DU chains and def-use pairs

- Link from def to all use s.t. def may reach them
  - DU(v, k) = { q | q uses v and (v,k) reaches q }
  - DU chains: all $k \rightarrow q$ s.t. k defines some v and $q \in$ DU(v, k)
- Set DU(v, k) can be computed as the inverse of UD
  - DU(v,k) = { q | $k \in$ UD(v,q) }
- Given DU(v,k), if some $q \in$ DU(v, k), we say that <k, q> is a def-use pair for v

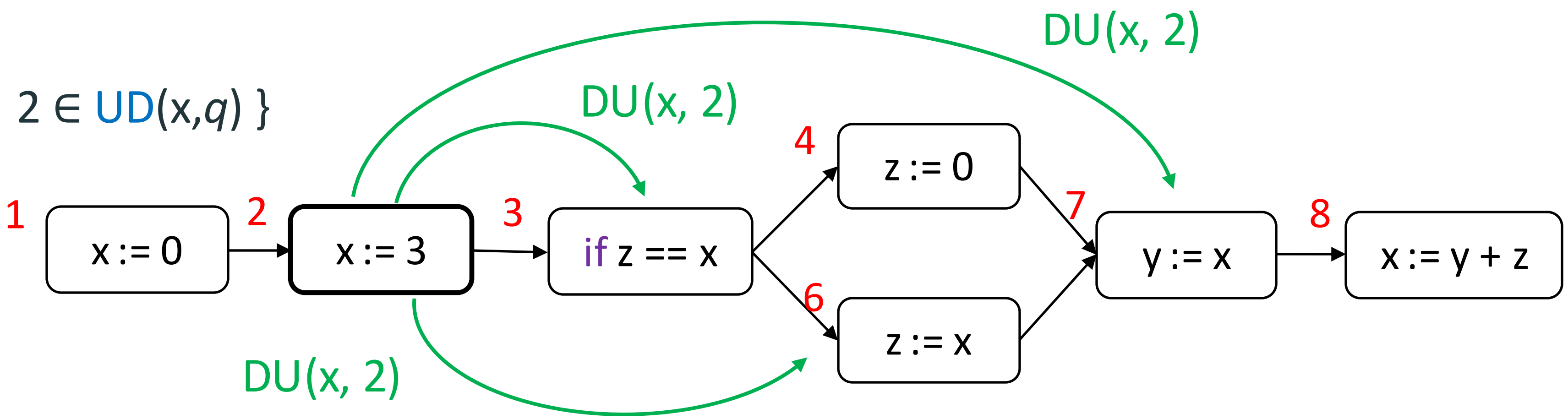
DU(x,2) = { q | 2 $\in$ UD(x,q) }



DU chains and def-use pairs

- Link from def to all use s.t. def may reach them
  - DU(v, k) = { q | q uses v and (v,k) reaches q }
  - DU chains: all $k \rightarrow q$ s.t. k defines some v and $q \in$ DU(v, k)
- Set DU(v, k) can be computed as the inverse of UD
  - DU(v,k) = { q | $k \in$ UD(v,q) }
- Given DU(v,k), if some $q \in$ DU(v, k), we say that <k, q> is a def-use pair for v

DU(x,2) = { q | 2 ∈ UD(x,q) }



Exercise

- Consider the following fragment of code in *C* language:

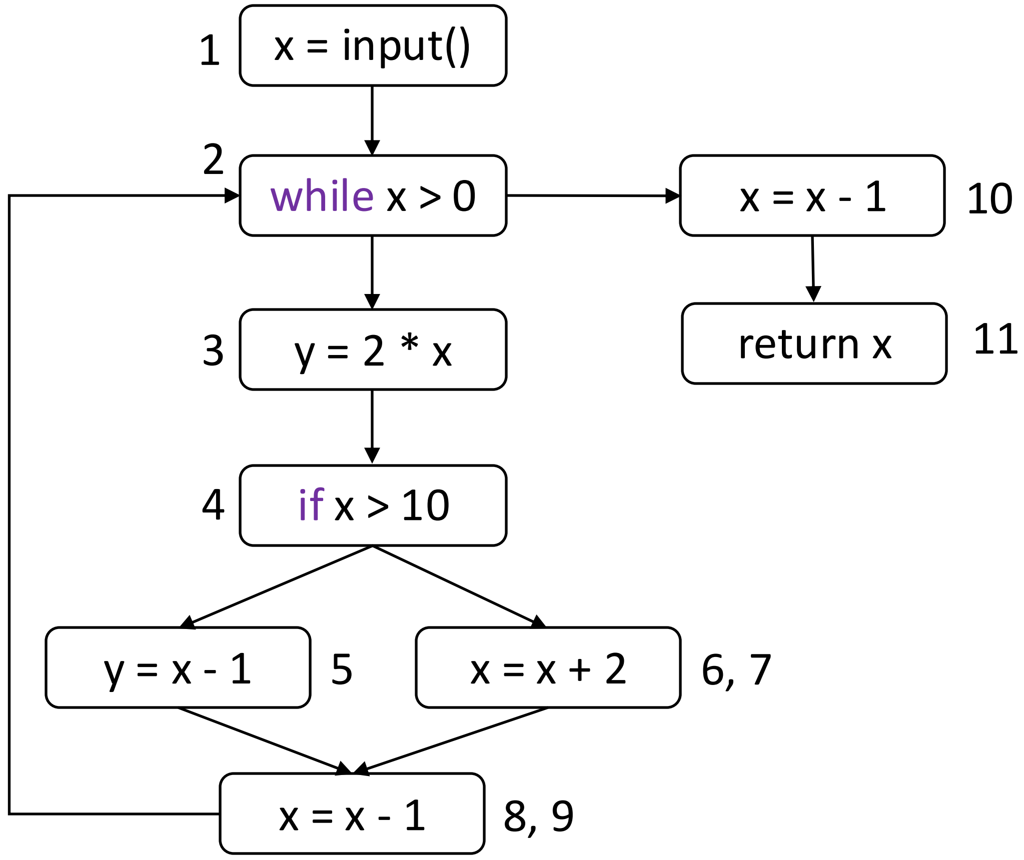
```c
int foo() {
  x = input();
  while (x > 0) {
    y = 2 * x;
    if (x > 10)
      y = x - 1;
    else
      x = x + 2;
    x = x - 1;
  }
  x = x - 1;
  return x;
}
```

You have to accomplish the following:

1. draw the control flow graph of the program;
2. apply the live variable analysis
3. point out a potential issue with this code that live variable analysis would be able to spot;
4. provide the def-use pairs for variables x and y;

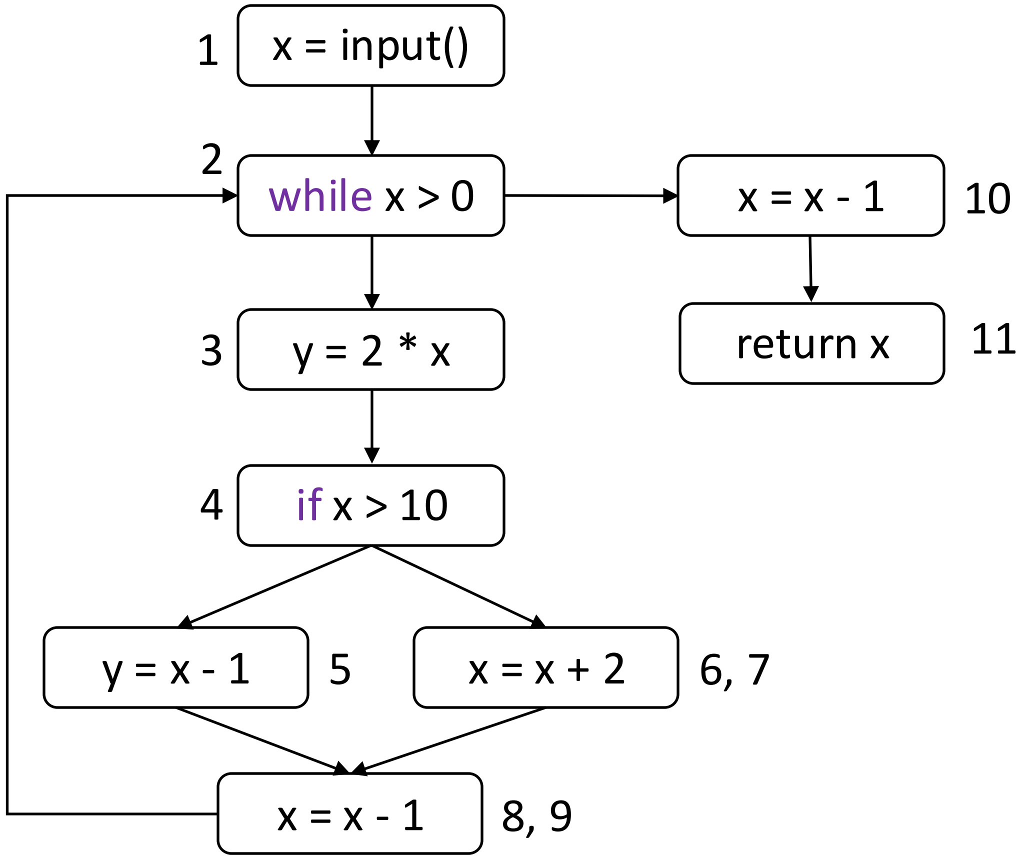
1. Control flow graph

```
0: int foo() {
1:   x = input();
2:   while (x > 0) {
3:     y = 2 * x;
4:     if (x > 10)
5:       y = x - 1;
6:     else
7:       x = x + 2;
8:     x = x - 1;
9:   }
10:  x = x - 1;
11:  return x;
12:}
```



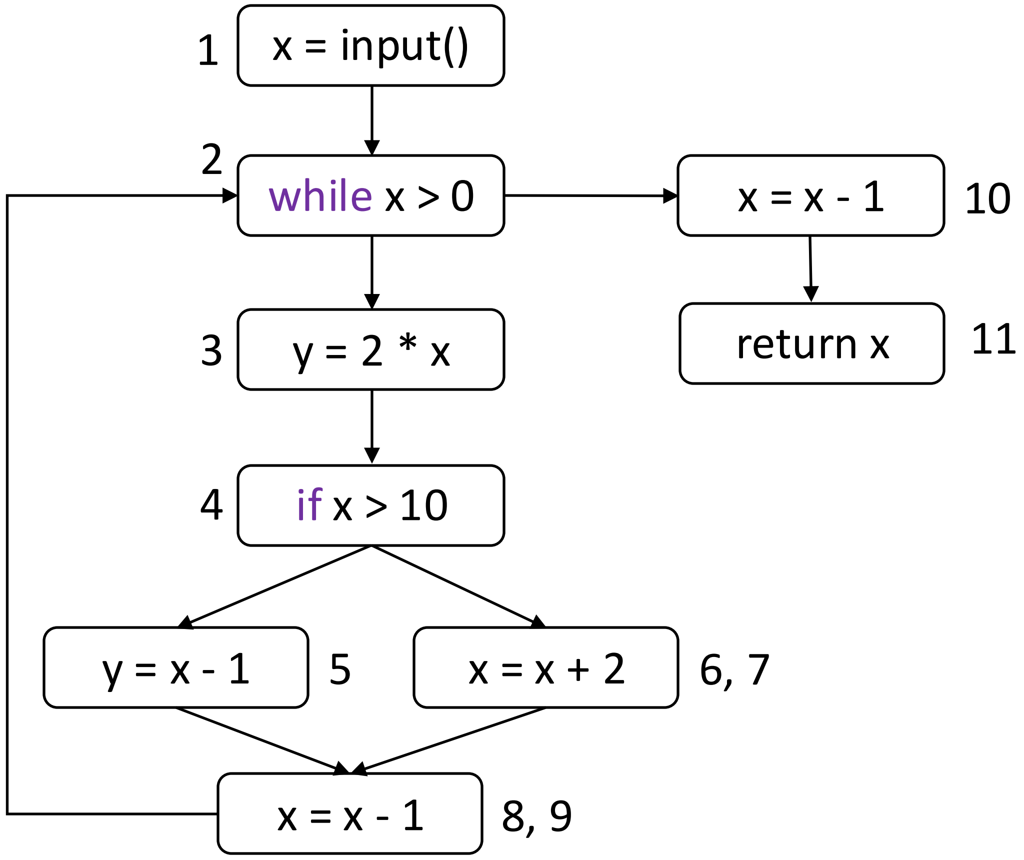
2. Live variable analysis

- LV(1) = {x}
- LV(2) = {x}
- LV(3) = {x} -> we have a problem, y is defined here and not live!
- LV(4) = {x}
- LV(5) = {x} -> same issue as before
- LV(6, 7) = LV(8, 9) = LV(10) = {x}
- LV(11) = {}



3. def-use pairs – Reaching definitions

- $RD_{IN}(1)$ = { (x,?), (y,?) }
- $RD_{IN}(2) = RD_{OUT}(1) \cup RD_{OUT}(8,9)$ = { (x,1), (y,?), (x, 8), (y, 5), (y, 3) }
- $RD_{IN}(3) = RD_{OUT}(2)$ = { (x,1), (y,?) , (x, 8), (y, 5), (y, 3)}
- $RD_{IN}(4) = RD_{OUT}(3)$ = { (x,1), (x, 8), (y, 3)}
- $RD_{IN}(5) = RD_{OUT}(4)$ = { (x,1), (x, 8), (y, 3)}
- $RD_{IN}(6,7) = RD_{OUT}(4)$ = { (x,1), (x, 8), (y, 3)}
- $RD_{IN}(8,9) = RD_{OUT}(5) \cup RD_{OUT}(6,7)$ = { (x,1), (y, 5), (x, 7), (x, 8), (y, 3) }
- $RD_{IN}(10) = RD_{OUT}(2)$ = { (x,1), (y,?) , (x, 8), (y, 5), (y, 3)}
- $RD_{IN}(11) = RD_{OUT}(10)$ = { (x,10), (y,?) , (y, 5), (y, 3)}



Def-use pairs – UD chains identification

- $RD_{IN}(1) = \{ (x,?), (y,?) \}$
- $RD_{IN}(2) = RD_{OUT}(1) \cup RD_{OUT}(8,9) = \{ (x,1), (y,?), (x, 8), (y, 5), (y, 3) \}$
- $RD_{IN}(3) = RD_{OUT}(2) = \{ (x,1), (y,?) , (x, 8), (y, 5), (y, 3)\}$
- $RD_{IN}(4) = RD_{OUT}(3) = \{ (x,1), (x, 8), (y, 3) \}$
- $RD_{IN}(5) = RD_{OUT}(4) = \{ (x,1), (x, 8), (y, 3) \}$
- $RD_{IN}(6,7) = RD_{OUT}(4) = \{ (x,1), (x, 8), (y, 3)\}$
- $RD_{IN}(8,9) = RD_{OUT}(5) \cup RD_{OUT}(6,7) = \{ (x,1), (y, 5), (x, 7), (x, 8), (y, 3) \}$
- $RD_{IN}(10) = RD_{OUT}(2) = \{ (x,1), (y,?) , (x, 8), (y, 5), (y, 3) \}$
- $RD_{IN}(11) = RD_{OUT}(10) = \{ (x,10), (y,?) , (y, 5), (y, 3) \}$

- x defined in 1, 7, 8, 10 and used in 2, 3, 4, 5, 7, 8, 10, 11
- y defined in 3, 5 and not used
- UD(x, 2) = UD(x, 3) = UD(x, 4) = UD(x, 5) = UD(x, 7) = UD(x, 10) = {1, 8}
- UD(x, 8) = {1, 7, 8}
- UD(x, 11) = {10}
- Def-use pairs for x
  <1, 2> <1, 3> <1, 4> <1, 5> <1, 7> <1, 8> <1, 10>
  <7,8>
  <8, 2> <8, 3> <8, 4> <8, 5> <8, 7> <8, 8> <8, 10>
  <10, 11>

References

- Carlo A. Furia. Material for the Software Analysis course. https://github.com/bugcounting/software-analysis/tree/master