Programm- & Systemverifikation

Coverage Criteria

Georg Weissenbacher

184.741

What happened so far

- How bugs come into being:
  - **Fault** – cause of an error (e.g., mistake in coding)
  - **Error** – *incorrect* state that may lead to failure
  - **Failure** – deviation from *desired* behaviour
- We specified *intended* behaviour using **assertions**
- We proved (simple) programs correct.
- We learned about black-box testing
  - equivalence partitioning
  - boundary testing

Black-box Testing

- Mainly applicable to higher levels of testing
  - Acceptance Testing
  - System Testing
- Focus on <u>what</u> the software does (not how it does it)
- Derive input equivalence classes by speculating on *behaviour*

Black-box Testing

```
float sqrt (float x);
```

pre: $\texttt{x} \geq 0$

post: $|\texttt{result}^2 - \texttt{x}| < \varepsilon$

Test cases from valid equivalence classes:

- $+0$, -0, `FLT_MAX`, `FLT_EPSILON`, 15.3

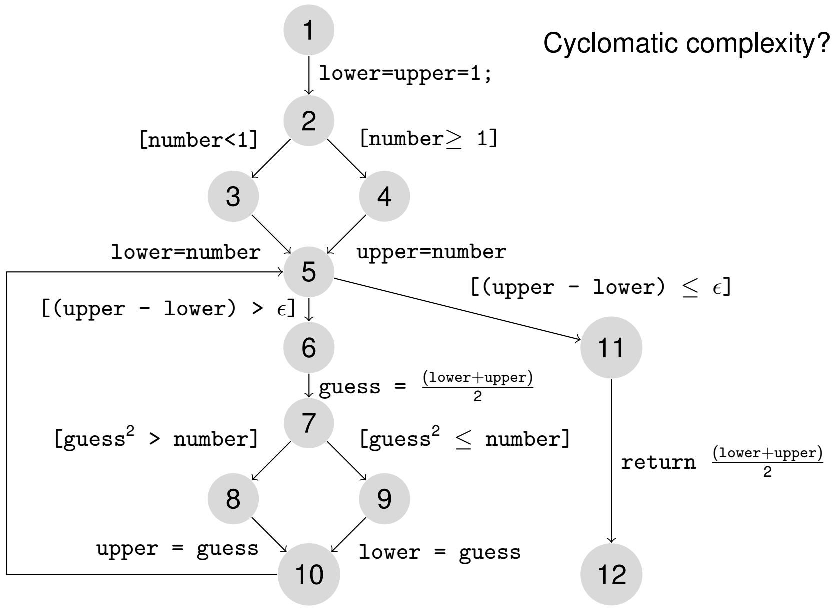
Test cases from invalid equivalence classes:

- `FLT_MIN`, `-FLT_EPSILON`, -7.9
- $-\infty$, $+\infty$
- `NaN`

Testing our Square-Root Implementation

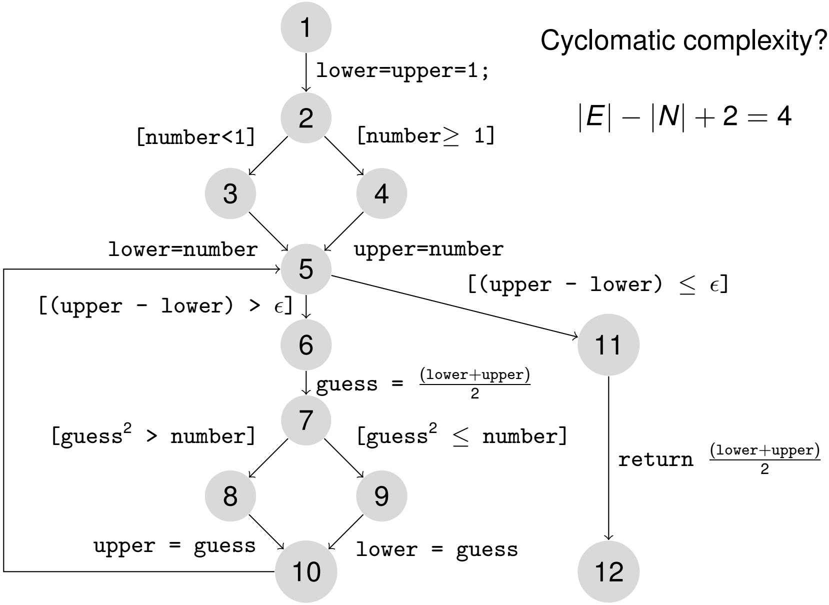
```
float sqrt (float number) {
  float lower = 1, upper = 1, guess;

  if (number < 1)
    lower = number;
  else
    upper = number;

  while ((upper - lower) > EPSILON) {
    guess = (lower + upper) / 2;
    if (guess*guess > number)
      upper = guess;
    else
      lower = guess;
  }
  return (lower + upper) / 2;
}
```

Testing our Square-Root Implementation



Cyclomatic complexity?

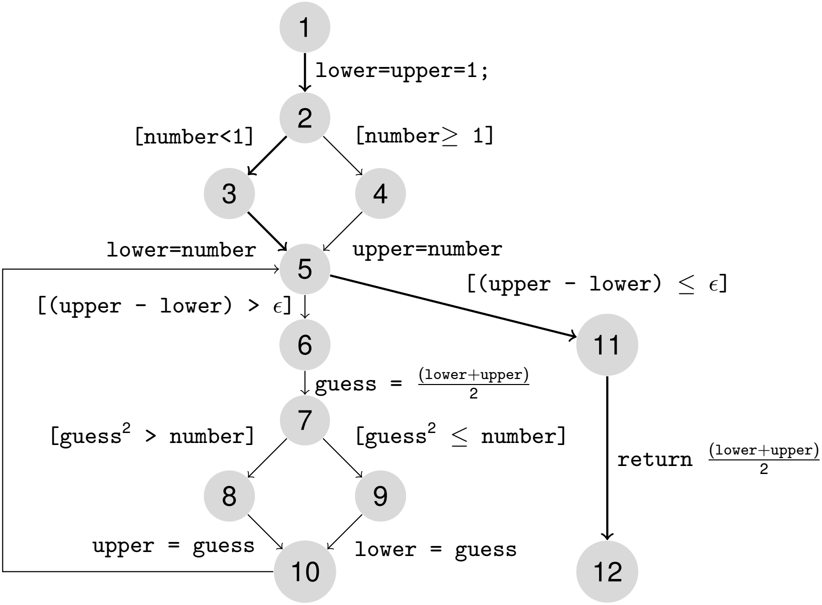
Testing our Square-Root Implementation



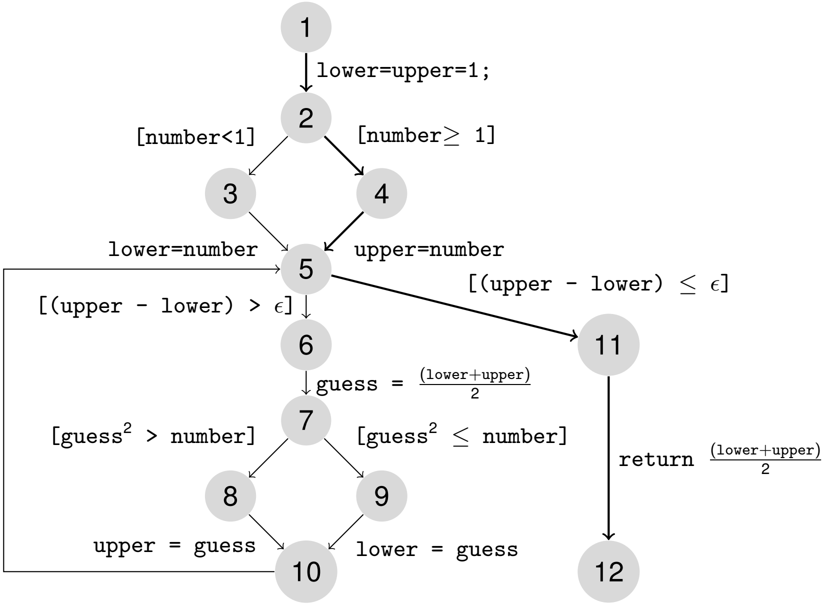
Cyclomatic complexity?

$$|E| - |N| + 2 = 4$$

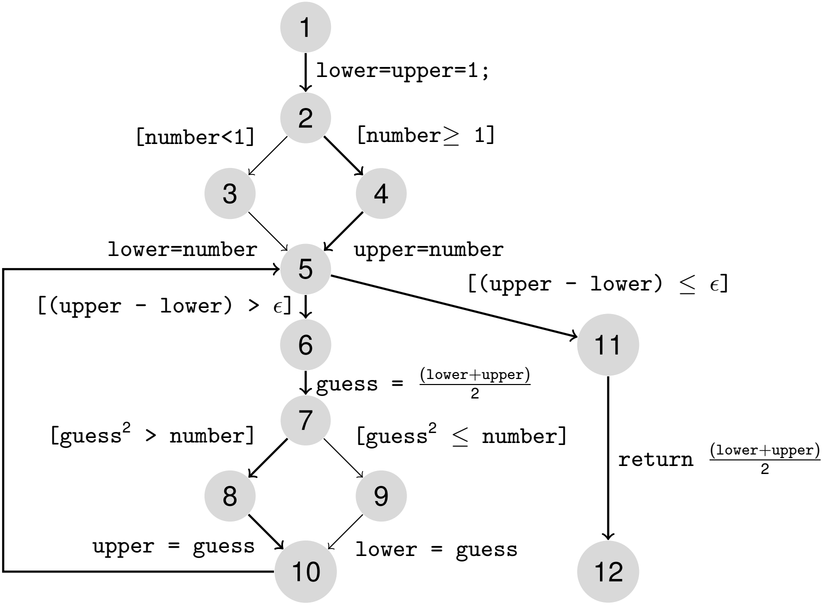
Testing our Square-Root Implementation



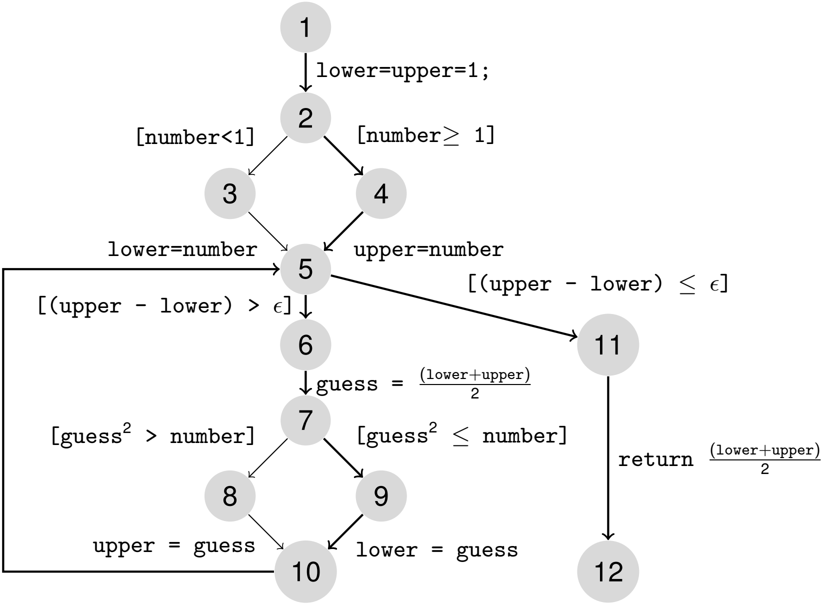
Testing our Square-Root Implementation



Testing our Square-Root Implementation



Testing our Square-Root Implementation



Cyclomatic Complexity of Square-Root Implementation

- *cyclomatic complexity* = max # linearly independent paths
- linearly independent $\stackrel{\text{def}}{=}$
  contains (at least) one edge not covered by other paths

$$1 \to 2 \to 3 \to 5 \to 11 \to 12$$

$$1 \to 2 \to 4 \to 5 \to 6 \to 7 \to 8 \to 10 \to 5 \to 11 \to 12$$

$$1 \to 2 \to 4 \to 5 \to 6 \to 7 \to 9 \to 10 \to 5 \to 11 \to 12$$

- think of linear algebra and linearly independent equations

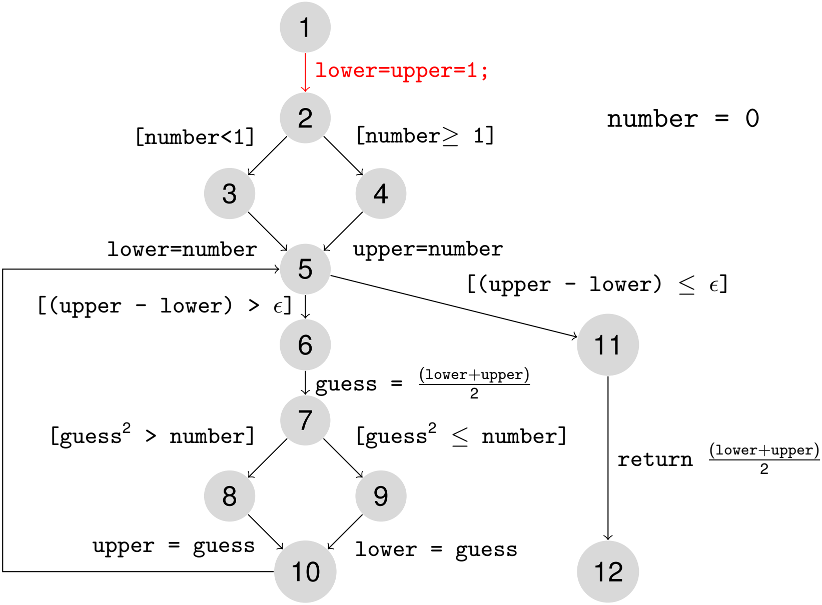
Cyclomatic Complexity

- upper bound of test-cases necessary to test all *branches*
- in our case, 2 paths are enough:
  - $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 9 \rightarrow 10 \rightarrow 10 \rightarrow 5 \rightarrow 11 \rightarrow 12$
  - $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 10 \rightarrow 10 \rightarrow 5 \rightarrow 11 \rightarrow 12$

Cyclomatic Complexity

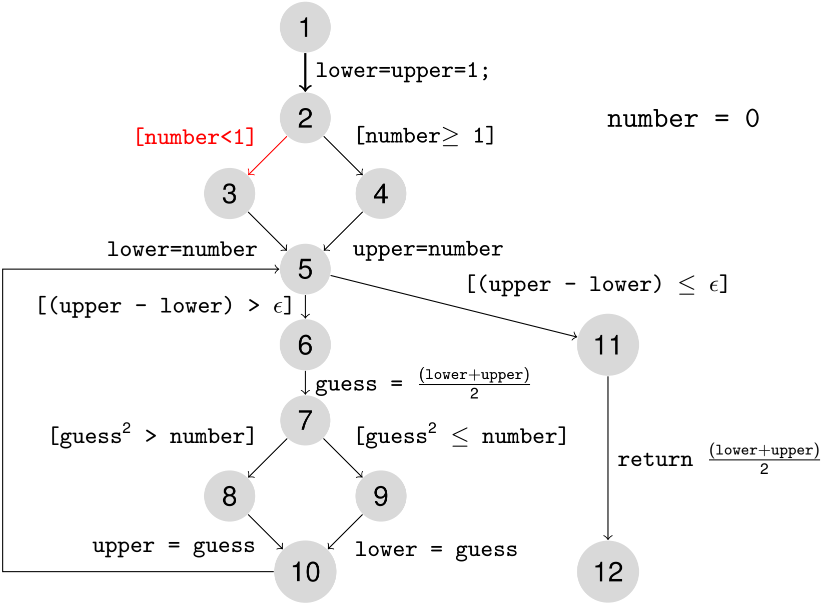
- upper bound of test-cases necessary to test all *branches*
- in our case, 2 paths are enough:
  - $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 9 \rightarrow 10 \rightarrow 10 \rightarrow 5 \rightarrow 11 \rightarrow 12$
  - $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 10 \rightarrow 10 \rightarrow 5 \rightarrow 11 \rightarrow 12$
- Do our test-cases cover all branches?

Testing our Square-Root Implementation

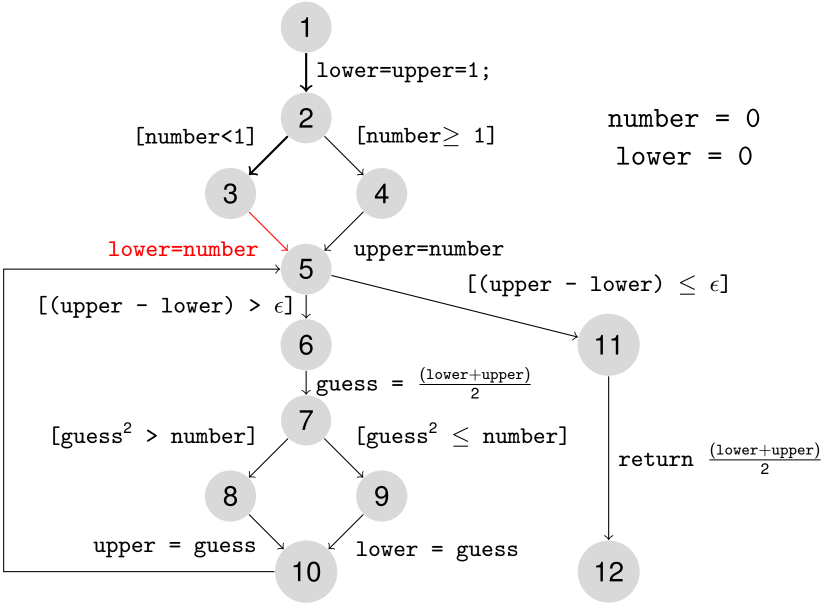


number = 0

Testing our Square-Root Implementation



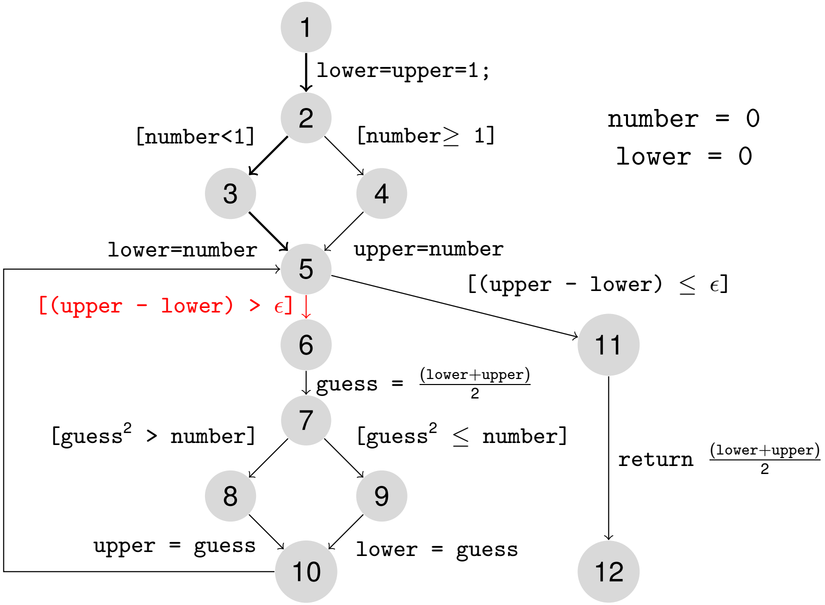
Testing our Square-Root Implementation



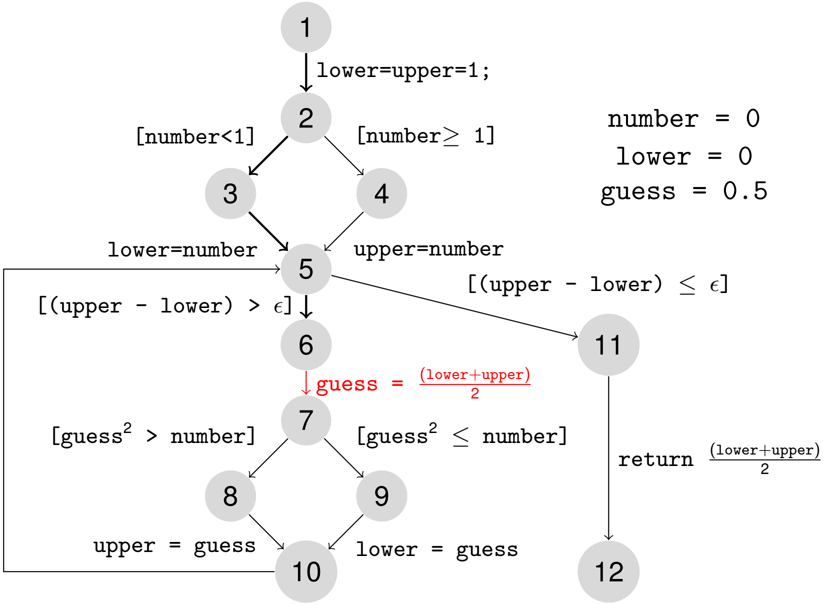
number = 0

lower = 0

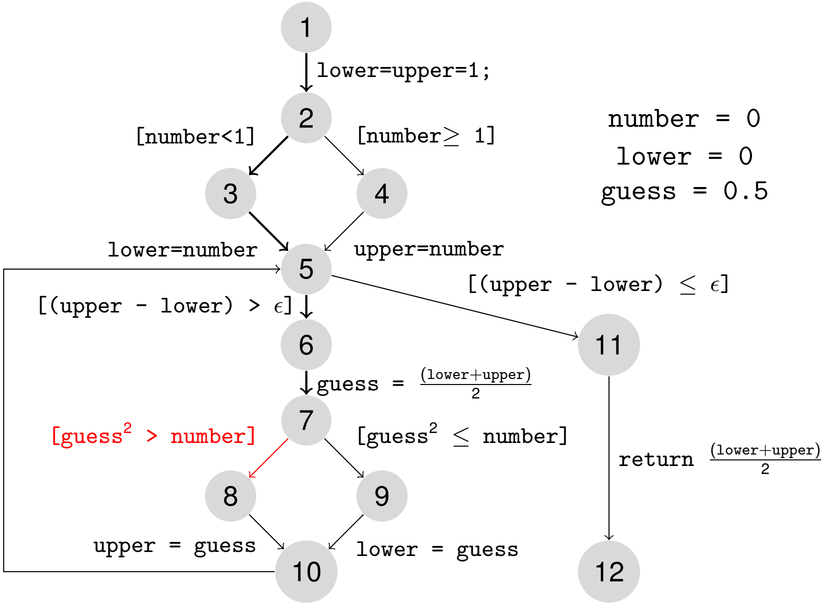
Testing our Square-Root Implementation



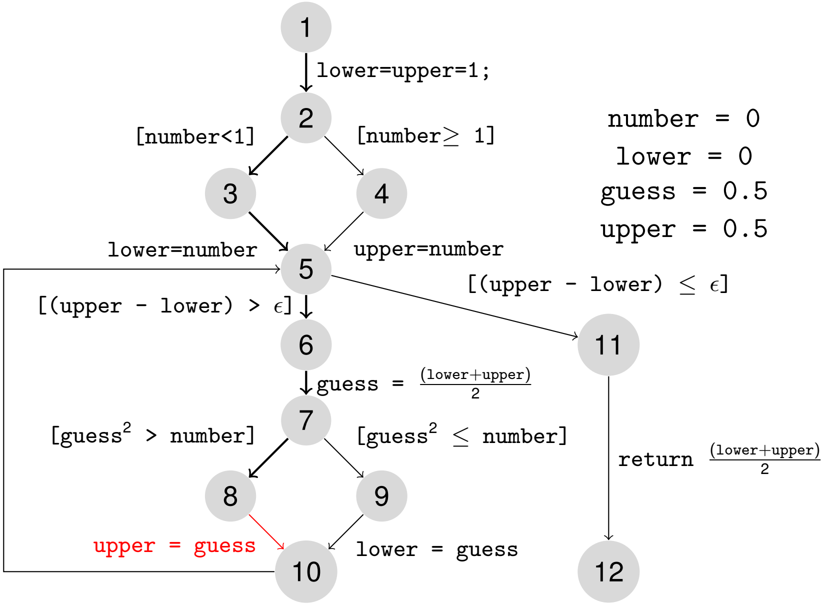
Testing our Square-Root Implementation



Testing our Square-Root Implementation



Testing our Square-Root Implementation



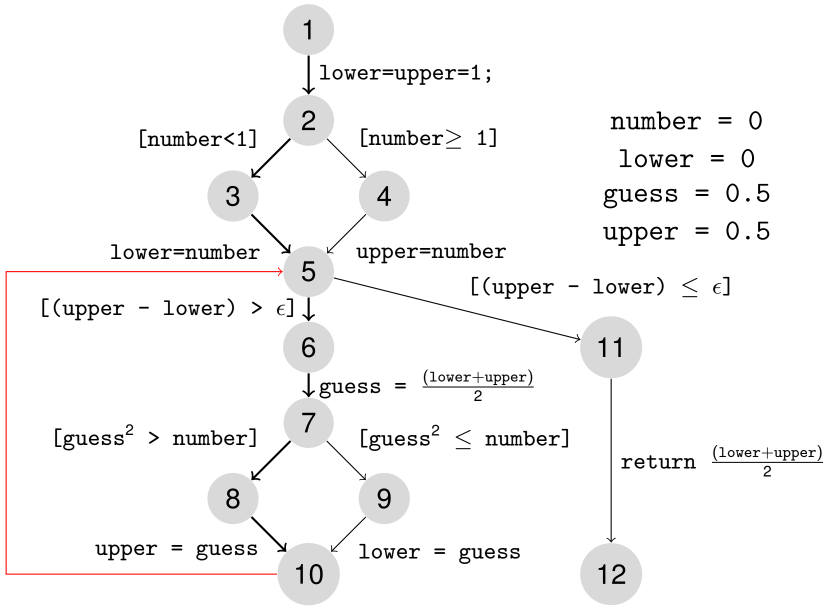
number = 0

lower = 0

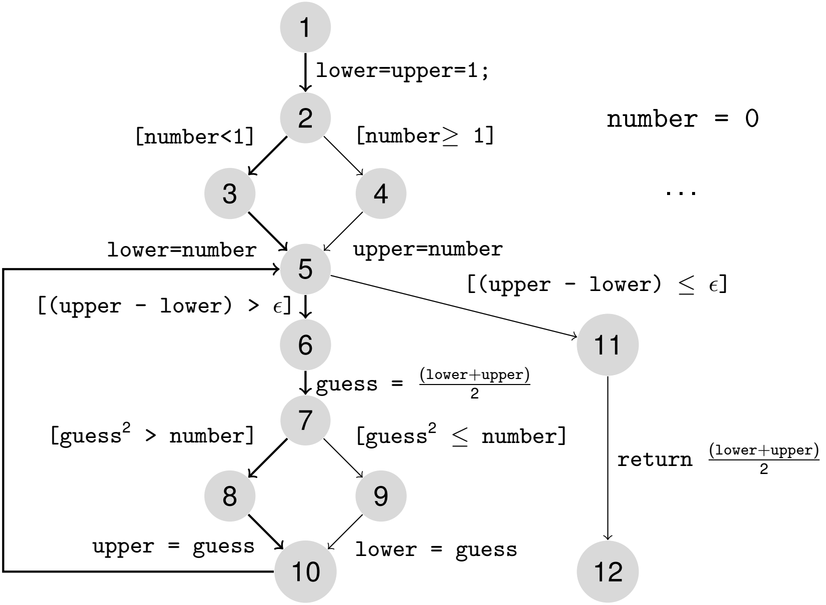
guess = 0.5

upper = 0.5

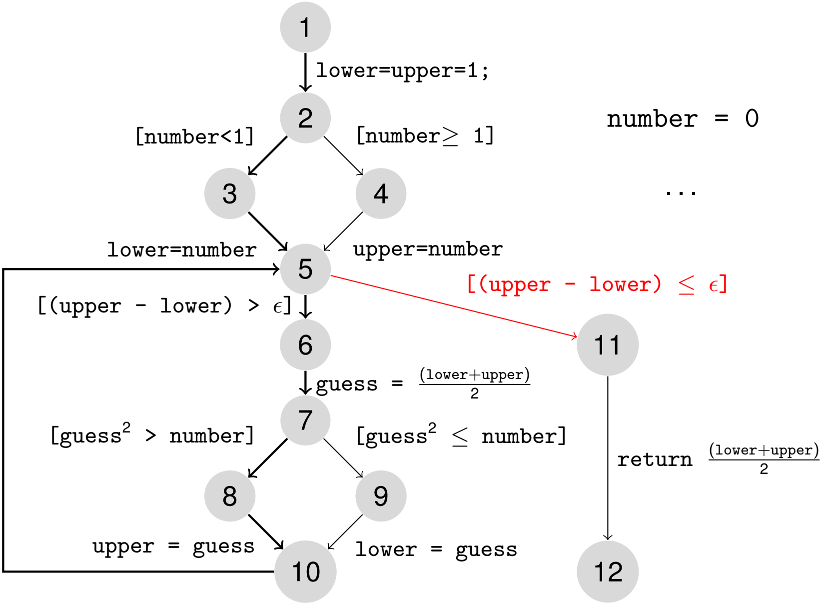
Testing our Square-Root Implementation



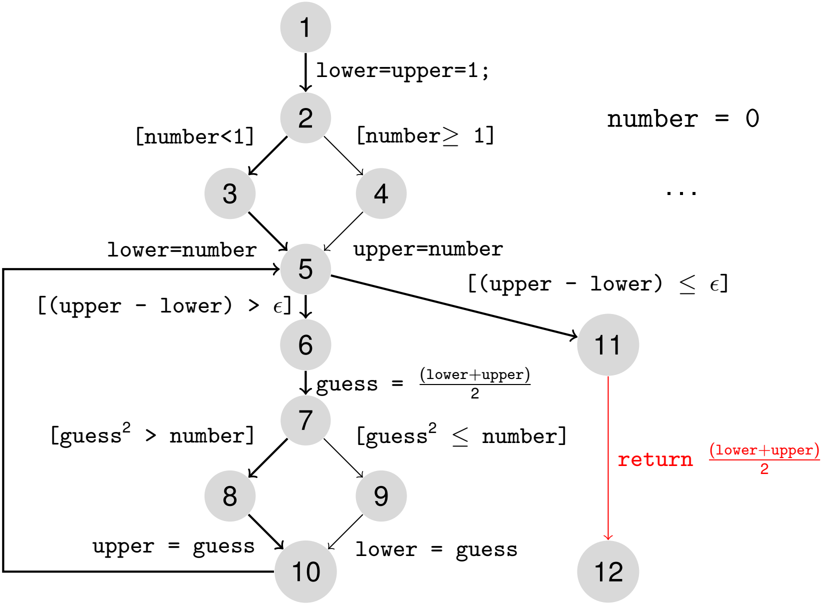
Testing our Square-Root Implementation



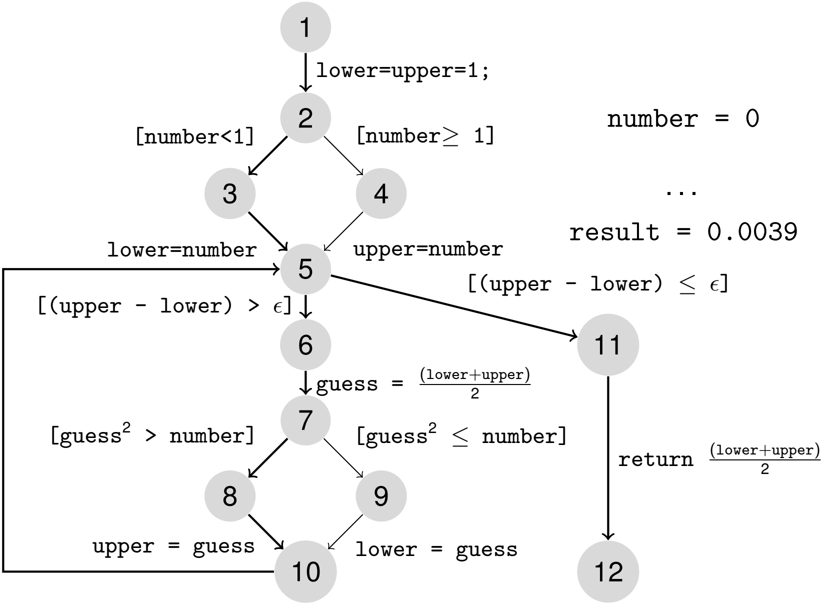
Testing our Square-Root Implementation



Testing our Square-Root Implementation



Testing our Square-Root Implementation



number = 0

...

result = 0.0039

Which "independent" path fragments were executed?

- Test case 0 traversed
  - $2 \rightarrow 3 \rightarrow 5$,
  - $7 \rightarrow 8 \rightarrow 10$, and
  - $5 \rightarrow 11 \rightarrow 12$
- It did *not* traverse
  - $2 \rightarrow 4 \rightarrow 5$ and
  - $7 \rightarrow 8 \rightarrow 10$
- Could we have predicted that one test case is not enough?

Which “independent” path fragments were executed?

- Test case 0 traversed
  - $2 \rightarrow 3 \rightarrow 5$,
  - $7 \rightarrow 8 \rightarrow 10$, and
  - $5 \rightarrow 11 \rightarrow 12$
- It did *not* traverse
  - $2 \rightarrow 4 \rightarrow 5$ and
  - $7 \rightarrow 8 \rightarrow 10$
- Could we have predicted that one test case is not enough?
  - Not without knowing the implementation!

Have we done “enough” testing?

- Reasonable to assume that “all of the code” should be tested!
- We need *at least one* additional test cases!

Have we done “enough” testing?

- Reasonable to assume that “all of the code” should be tested!
- We need *at least one* additional test cases!
  - Let’s have a look at 15.3, ok?

GCOV Usage

This is tedious, can't we automate this?

- `gcc -g -fprofile-arcs -ftest-coverage -o sqrt sqrt.c`
  (use `clang` instead of `gcc` on newer Macs)
- `gcov sqrt`
- `cat sqrt.c.gcov`
- `./sqrt ; gcov sqrt`
- `cat sqrt.c.gcov`

Coverage information for `sqrt(0.0)`

```
        1:    6:float sqrt (float number) {
        1:    7:  float lower = 1, upper = 1, guess;
        -:    8:
        1:    9:  if (number < 1)
        1:   10:    lower = number; // sqrt < 1, but > number
        -:   11:  else
    #####:   12:    upper = number; // sqrt > 1, but < number
        -:   13:
        9:   14:  while ((upper - lower) > EPSILON) {
        7:   15:    guess = (lower + upper) / 2;
        7:   16:    if (guess*guess > number)
        7:   17:      upper = guess;
        -:   18:    else
    #####:   19:      lower = guess;
        7:   20:  }
        1:   21:  return (lower + upper) / 2;
        -:   22:}
```

Coverage information for `sqrt(15.3)`

```
        1:    6:float sqrt (float number) {
        1:    7:  float lower = 1, upper = 1, guess;
        -:    8:
        1:    9:  if (number < 1)
    #####:   10:    lower = number; // sqrt < 1, but > number
        -:   11:  else
        1:   12:    upper = number; // sqrt > 1, but < number
        -:   13:
       13:   14:  while ((upper - lower) > EPSILON) {
       11:   15:    guess = (lower + upper) / 2;
       11:   16:    if (guess*guess > number)
        8:   17:      upper = guess;
        -:   18:    else
        3:   19:      lower = guess;
       11:   20:  }
        1:   21:  return (lower + upper) / 2;
        -:   22:}
```

Have we done “enough” testing?

What is “enough”?

- Does executing all statements guarantee correctness?

Have we done “enough” testing?

What is “enough”?

- Does executing all statements guarantee correctness?
  - What about the code `f(int x) { return (1/x);}`

Have we done "enough" testing?

What is "enough"?

- Does executing all statements guarantee correctness?
  - What about the code `f(int x) { return (1/x);}`
- Do we have to test *all inputs*?
  - How many different inputs are there to `sqrt(float)`?

Have we done "enough" testing?

What is "enough"?

- Does executing all statements guarantee correctness?
  - What about the code `f(int x) { return (1/x);}`
- Do we have to test *all inputs*?
  - How many different inputs are there to `sqrt(float)`?
    - `sizeof(float)` = 4 bytes, so roughly 2^{32}

Have we done “enough” testing?

What is “enough”?

- Does executing all statements guarantee correctness?
  - What about the code `f(int x) { return (1/x);}`
- Do we have to test *all inputs*?
  - How many different inputs are there to `sqrt(float)`?
    - `sizeof(float)` = 4 bytes, so roughly 2^{32}
  - How many different inputs are there to our `AVL` implementation?

Have we done "enough" testing?

What is "enough"?

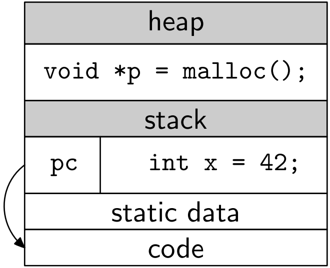
- Does executing all statements guarantee correctness?
  - What about the code `f(int x) { return (1/x);}`
- Do we have to test *all inputs*?
  - How many different inputs are there to `sqrt(float)`?
    - `sizeof(float)` = 4 bytes, so roughly 2^{32}
  - How many different inputs are there to our `AVL` implementation?
- Maybe visit all possible *states*?

Have we done “enough” testing?

What is a *state*?

Values of

- global variables

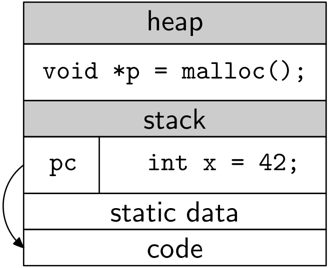


Have we done “enough” testing?

What is a *state*?

Values of

- global variables
- stack variables

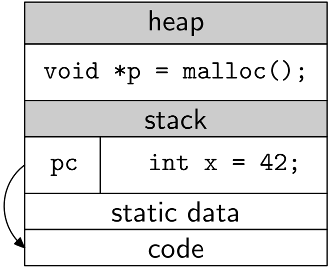


Have we done "enough" testing?

What is a *state*?

Values of

- global variables
- stack variables
- heap…

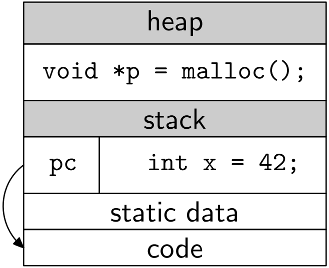


Have we done "enough" testing?

What is a *state*?

Values of

- global variables
- stack variables
- heap...



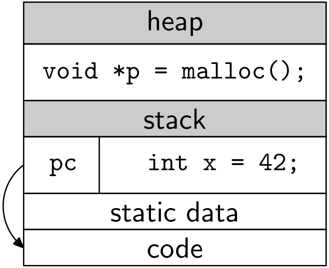
How many possible states are there?

Have we done "enough" testing?

What is a *state*?

Values of

- global variables
- stack variables
- heap...



How many possible states are there?

- ∞, in theory

Have we done “enough” testing?

How about finite state programs?

- Assume that there are only n different elements that we can insert into our AVL tree.

Element	1	2	...	n
Inserted	✓	✗	...	✓

- Finitely many *states* for infinitely test scenarios

Have we done "enough" testing?

How about finite state programs?

- Assume that there are only *n* different elements that we can insert into our AVL tree.

Element	1	2	…	n
Inserted	✓	✗	…	✓

- Finitely many *states* for infinitely test scenarios
  - But still 2^n possible sets (and even more trees)!

Have we done "enough" testing?

But aren't many trees "*similar*"?

Elements	1	2	3	4	5
State 1	✓	✓	✗	✗	✗
State 2	✗	✓	✓	✗	✗

- Maybe, we don't need to "cover" all of them?
- What is the problem with this argument?

Have we done "enough" testing?

But aren't many trees "*similar*"?

Elements	1	2	3	4	5
State 1	✓	✓	✗	✗	✗
State 2	✗	✓	✓	✗	✗

- Maybe, we don't need to "cover" all of them?
- What is the problem with this argument?
  - it is not *formally proven* (maybe even wrong)
  - it is specific to one program

Coverage Criteria

- Common agreement on what "sufficiently tested" means
  - coverage criteria are about *confidence*, trust
  - required for *certification* (according to industry standards)
- Important: achieving coverage is not a goal in itself
  - "The journey is the reward:" Testing until coverage is reached
  - Test-cases should be generated from *requirements*

Coverage Criteria

Coverage criteria define equivalence classes with respect to program behaviour

- Control flow-based coverage
  - Path coverage
  - Statement/basic block coverage
  - Branch coverage
  - Decision coverage
  - Condition coverage
  - Condition/Decision coverage
  - Modified condition/decision coverage (MC/DC)
  - Multiple decision coverage
- Data flow-based coverage
  - Definition/use pairs
- Mutation testing
- ...

Coverage Criteria: Path coverage

- Goal: Execute every *path* of the program
  - Independently of the variable values along that path
  - Every path is an equivalence class
- What's the number of paths through the following program?

```
while (1) {
  if (getchar() == EOF)
     break;
}
```

Coverage Criteria: Path coverage

- Goal: Execute every *path* of the program
  - Independently of the variable values along that path
  - Every path is an equivalence class
- What's the number of paths through the following program?

```
while (1) {
  if (getchar() == EOF)
     break;
}
```

- In general, path coverage can't be achieved

Coverage Criteria: Statement Coverage

- Goal: Execute every program statement at least once
  - All paths visiting that statement build equivalence class
- Bad criterion:
  - consider test case `x = 5` for following code fragment:

```
if (x > 1) {
  x++;
}
int y = x/y;
```

Coverage Criteria: Statement Coverage

- Goal: Execute every program statement at least once
  - All paths visiting that statement build equivalence class
- Bad criterion:
  - consider test case `x = 5` for following code fragment:

```
if (x > 1) {
  x++;
}
int y = x/y;
```

- All statements executed, but `else` branch never taken
- May not exercise all outcomes of a conditional statement

Coverage Criteria: Branch Coverage

- Goal: Execute all branches in a program
  - Equivalence class: paths execute a certain branch
- Usually implies statement coverage (but see comments later)
- C.f. cyclomatic complexity

Coverage Criteria: Decision Coverage

- Goal: Exercise every decision outcome at least once
  - decision is a "Boolean expression composed of conditions and zero or more Boolean operators"
  - Equivalence class: paths in which decision evaluates to same value
- Subtly different from "branch coverage"
  - *Vacuously* true for the following program:

```
x = y;
x++;
```

  - *all* decisions covered even *without testing*
  - Therefore, does *not* imply statement coverage

Coverage Criteria: Decision Coverage vs. Branch Coverage

Danger, Will Robinson:

branch coverage $\neq$ condition coverage

At least not in general!

- *Numerous* subtle differences
- *Inconsistent* definitions (in industry standards)
- In particular, neither metric subsumes the other

Coverage Criteria: Definition of “Branch”

- **branch (1) (software).** (A) A computer program construct in which one of two or more alternative sets of programs statements is selected for execution. (B) A point in a computer program at which one of two or more alternative sets of program statements is selected for execution. Syn: *branchpoint*. [. . .]
- **branch testing.** Testing designed to execute each outcome of each decision point in a computer program. Contrast with: path testing; statement testing.

IEEE Std 100-1992 Standard Dictionary of Electrical and Electronic Terms

Coverage Criteria: Definition of “Branch”

imprecise definitions of “branch”

- Some definitions may or may not include
  - unconditional branches (`goto`)
  - function calls
  - fall-throughs (in `switch`/`case` constructs
-
  - Example:

```
    x++;
    goto L;
    ...
L:  y++;
```

  - Contains *no decisions*
  - But: contains a *non-conditional* branch

expressions with side effects

- Consider the following example:

```
if ((y > 1) && (( z > 1) || foo())
  x = y;
else
  x = z;
```

  - “Decision” evaluates to true if `y > 1` and `z > 1`
  - “Decision” evaluates to false if `z <= 1`
  - `foo` is never executed (short-circuited evaluation!)
  - covered by branch coverage, *if* function call a is branch

Coverage Criteria: Definition of “Decision”

- “Decision” is defined as “Boolean expression”
  - *not necessarily only* at branching points!

```
x = x + 1;
b = (x>0);
y = x;
```

  - Strictly speaking, have to cover every outcome of `x>0`
  - E.g., enforced in DO-178B standard
  - This code doesn’t contain any branch!

Coverage Criteria: Branch Coverage vs Decision Coverage

Branch coverage implies decision coverage

- if “decision” means Boolean expressions at branching points only

Decision coverage is stronger than branch coverage

- if “branch” doesn’t include unconditional jumps
- if “decision” refers to *all* Boolean expressions

Often branch and decision outcome are used synonymously

Coverage Criteria: Branch Coverage vs Decision Coverage

Branch coverage implies decision coverage

- if “decision” means Boolean expressions at branching points only

Decision coverage is stronger than branch coverage

- if “branch” doesn’t include unconditional jumps
- if “decision” refers to *all* Boolean expressions

Often branch and decision outcome are used synonymously

Meaning varies, depending on industry standard that applies

- Also: interpreted differently by different coverage tools

Coverage Criteria: Condition Coverage

- Goal: Exercise every sub-expression/atom/condition outcome
  - atom is a Boolean expression not containing Boolean operators (e.g., `&&`, `||`)
  - Equivalence class: paths in which condition evaluates to same value
- Does not imply decision coverage!
  - Consider the following program fragment:

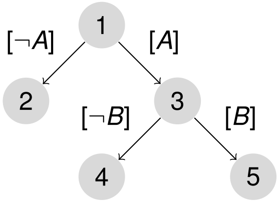
```
if ((x > 0) && (y > 0))
  x++;
```

  - Inputs: x = 5, y = -3 and x = -1 and y = 2
  - All condition outcomes considered, but decision always false

Coverage Criteria: Condition Coverage

- Goal: Exercise every sub-expression/atom/condition outcome
  - atom is a Boolean expression not containing Boolean operators (e.g., `&&`, `||`)
  - Equivalence class: paths in which condition evaluates to same value
- Can be considered as path partitioning if evaluation follows some order
  - think of generated intermediate representation

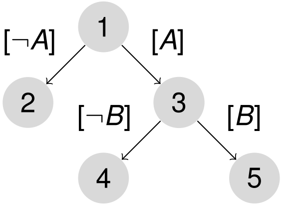
```
if (A && B)
```



Coverage Criteria: Condition/Decision Coverage

- Combination of decision and condition coverage
  - Cover all condition outcomes
  - Cover all decision outcomes
- not all branches in intermediate code might be executed!
- Consider the following cases:

A	B	A && B
0	0	0
1	1	1



- Coverage criterion is satisfied; $1 \rightarrow 3 \rightarrow 4$ never executed!

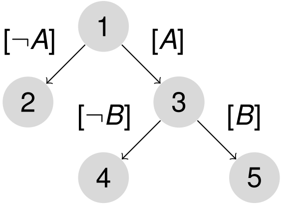
Coverage Criteria: *Modified* Condition/Decision Coverage (MC/DC)

- Each condition outcome must affect the decision outcome independently
  - “fix” the value of all conditions in a decision except for one
  - flipping that one decision must change the decision outcome
  - each outcome of the condition must influence the outcome of the decision at least once

Coverage Criteria: *Modified* Condition/Decision Coverage (MC/DC)

- Each condition outcome must affect the decision outcome independently
  - "fix" the value of all conditions in a decision except for one
  - flipping that one decision must change the decision outcome
  - each outcome of the condition must influence the outcome of the decision at least once

A	B	A && B
0	0	0
1	1	1



- MC/DC not satisfied: neither $A = 0$ nor $B = 0$ influence outcome 0 of A && B independently!
  - need to add $A = 0, B = 1$ and $A = 1, B = 0$

Coverage Criteria: MC/DC as defined in DO-178B

1. Every entry and exit point in the program has to be visited
2. Every conditional statement (i.e., branchpoint) has to take all possible outcomes (i.e., branches)
3. Every non-constant Boolean expression has to evaluate at least once to 1 and at least once to 0
4. Every non-constant condition in a Boolean expression has to evaluate at least once to 1 and at least once to 0
5. Every non-constant condition in a Boolean expression has to affect that expression's outcome independently

- Decision coverage requires (1, 2, 3)
- Decision/Condition coverage requires (1, 2, 3, 4)
- MC/DC requires 1 through 5
  - Note: equating branch and decision coverage violates MC/DC definition in DO-178B

DO-178B (Software Considerations in Airborne Systems and Equipment Certification)

- Safety standard
- Used for certification of safety critical software
- defines levels of criticality depending on potential damage of fault:
  - catastrophic
  - hazardrous/sever-major
  - major
  - minor
- Defines corresponding criticality levels *A*, *B*, *C*, *D*

Coverage Criteria in DO-178B

- For certification, following coverage criteria apply:

A	MC/DC
B	Decision and Statement coverage
C	Statement coverage
D	None

- (also specifies other criteria, e.g., documentation, traceability of requirements to test-cases, etc.)

Coverage Criteria: Multiple Condition Coverage

- All combinations of conditions in each decision have to be tested
- Consider the expression `(A || B) && C`
  - Condition/Decision coverage:

A	B	C
1	1	1
0	0	0

  - MC/DC (bold values influence decision outcome):

A	B	C
0	**0**	1
1	0	**1**
0	**1**	**1**
1	1	**0**

  - Multiple condition overage: *all* 2^3 combinations!

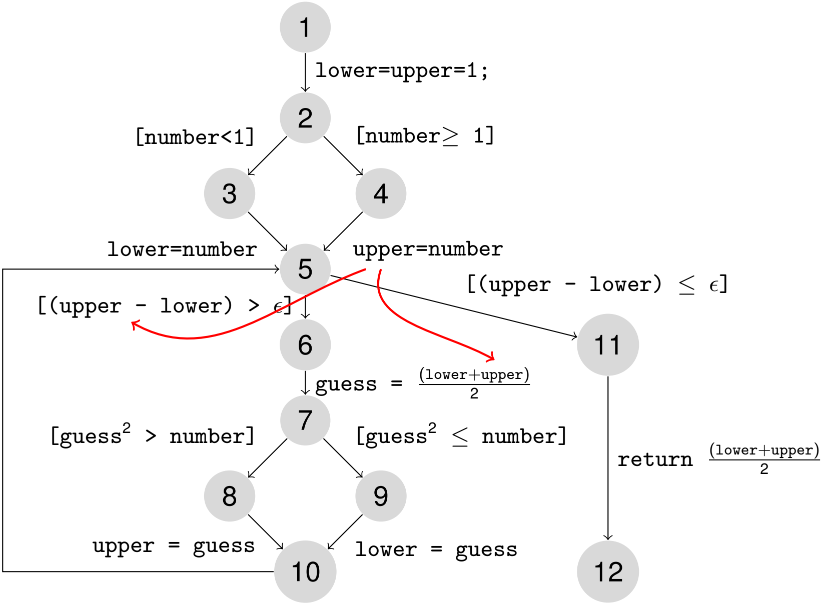
Coverage Criteria

- Control flow-based coverage
  - Path coverage
  - Statement/basic block coverage
  - Branch coverage
  - Decision coverage
  - Condition coverage
  - Condition/Decision coverage
  - Modified condition/decision coverage (MC/DC)
  - Multiple decision coverage
- **Data flow-based coverage**
  - Definition/use pairs
- Mutation testing
- ...

Coverage Criteria: Data-Flow

- Data flow: how do values propagate through program?
  - **definition:** assignment of a value to a variable
  - **use:** statement where the value is read
  - **def-use chain:** cycle-free path, first statement defines value, last statement uses value; value not re-defined in between

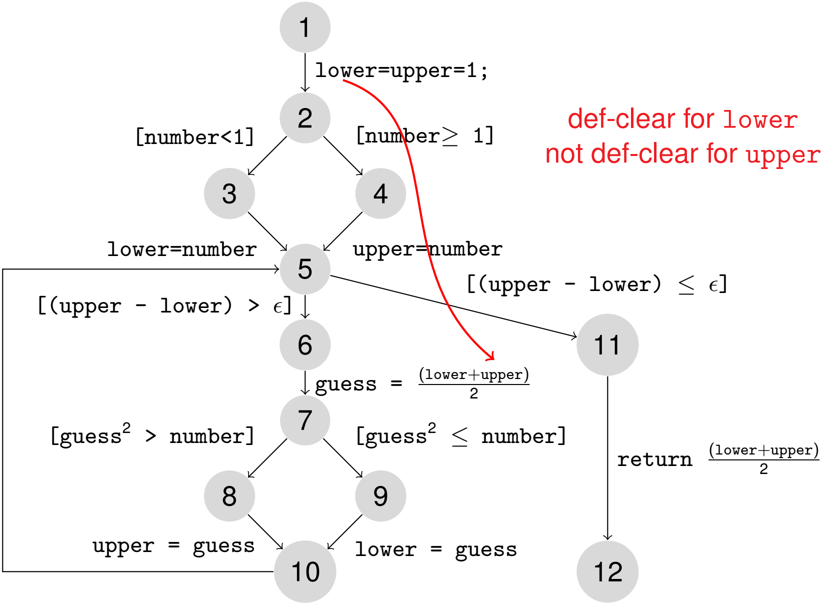
Coverage Criteria: Data-Flow



Coverage Criteria: Data-Flow

- Definitions can "flow into"
  - Boolean expressions ("predicates") in conditional statements
  - variables used to define ("compute") other values (right-hand-side of assignment)
- Some notation:
  - defs(`x`): locations where `x` is defined
  - p-use(`x`): locations where `x` is used in predicate
  - c-use(`x`): locations where `x` is used to compute other value
- A path is def-clear for `x` if
  - it traverses no location twice (exception: if first = last location)
  - `x` may not be re-defined between first and last node

Coverage Criteria: Data-Flow



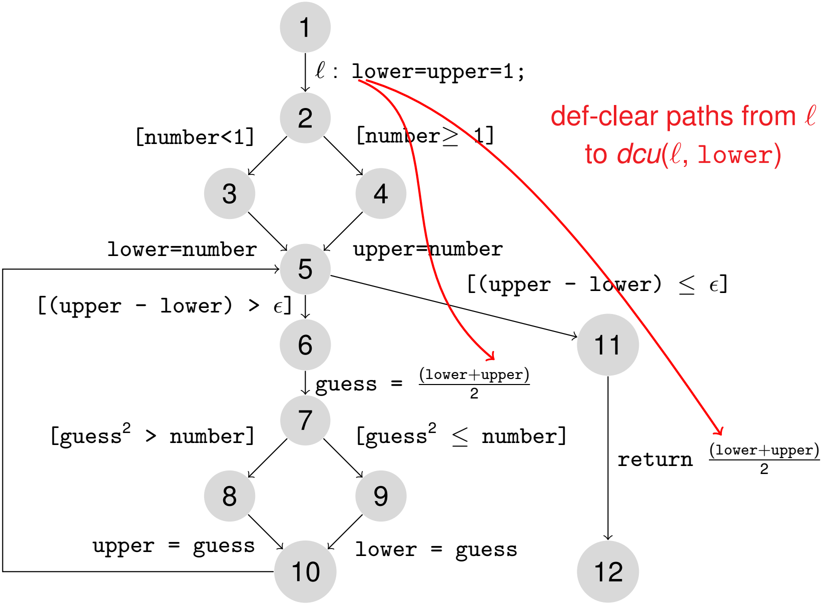
Coverage Criteria: Data-Flow

- dpu(ℓ, $\mathtt{x}$) locations $\ell' \in$ p-use($\mathtt{x}$) such that there is a def-clear path from ℓ to ℓ'

  This are the locations which use $\mathtt{x}$ in a predicate and can potentially be influenced by the definition of $\mathtt{x}$ at ℓ
- dcu(ℓ, $\mathtt{x}$) locations $\ell' \in$ c-use($\mathtt{x}$) such that there is a def-clear path from ℓ to ℓ'

  This are the locations which use $\mathtt{x}$ in a computations and can potentially be influenced by the definition of $\mathtt{x}$ at ℓ

Coverage Criteria: Data-Flow, Example for dcu



Frankl & Weyuker's Data-Flow Coverage Criteria

For each definition of variable $\mathtt{x}$ and for every $\ell \in \text{defs}(\mathtt{x})$, the test suite traverses:

- all-defs: one path to *some* $\ell' \in (\text{dpu}(\ell, \mathtt{x}) \cup \text{dcu}(\ell, \mathtt{x}))$

  $\Rightarrow$ all definitions get used
- all-c-uses: one path to *each* $\ell' \in \text{dcu}(\ell, \mathtt{x})$

  $\Rightarrow$ all computations affected by each definition are executed
- all-p-uses: one path to *each* $\ell' \in \text{dpu}(\ell, \mathtt{x})$

  $\Rightarrow$ all decisions affected by each definition are executed

Frankl & Weyuker's Data-Flow Coverage Criteria

- all-c-uses/some-p-uses: one path to *each* $\ell' \in \text{dcu}(\ell, \mathtt{x})$, but if $\text{dcu}(\ell, \mathtt{x}) = \emptyset$, then at least one path to $\ell' \in \text{dpu}(\ell, \mathtt{x})$

  $\Rightarrow$ all definitions used, and if they affect computations, then all affected computations are executed
- all-p-uses/some-c-uses: one path to *each* $\ell' \in \text{dpu}(\ell, \mathtt{x})$, but if $\text{dpu}(\ell, \mathtt{x}) = \emptyset$, then at least one path to $\ell' \in \text{dcu}(\ell, \mathtt{x})$

  $\Rightarrow$ all definitions used, and if they affect decisions, then all affected decisions are executed

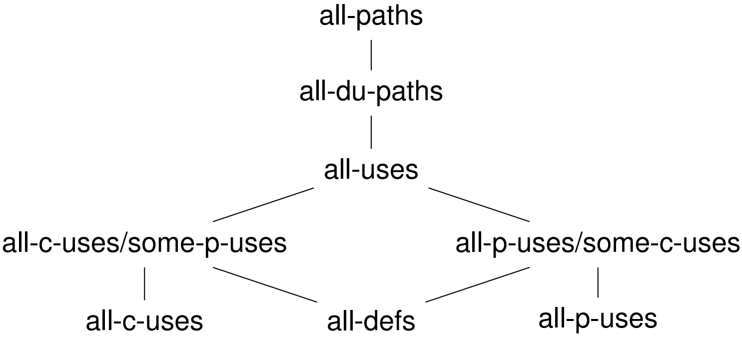
Frankl & Weyuker's Data-Flow Coverage Criteria

- all-uses: one path to *each* node $\ell' \in (\text{dpu}(\ell, \mathtt{x}) \cup \text{dcu}(\ell, \mathtt{x}))$

  $\Rightarrow$ every computation and decision affected by definition executed
- all-du-paths: *all paths* to each node $\ell' \in (\text{dpu}(\ell, \mathtt{x}) \cup \text{dcu}(\ell, \mathtt{x}))$

  $\Rightarrow$ like above, but *all* def-use paths

Subsumption Lattice



Coverage Criteria: Data-Flow

- Data-flow criteria track dependencies between variables
- Set of all pairs can be approximated by static analysis
  - typically covered in course on compiler design

Coverage Criteria

- Control flow-based coverage
  - Path coverage
  - Statement/basic block coverage
  - Branch coverage
  - Decision coverage
  - Condition coverage
  - Condition/Decision coverage
  - Modified condition/decision coverage (MC/DC)
  - Multiple decision coverage
- Data flow-based coverage
  - Definition/use pairs
- **Mutation testing**
- ...

Coverage Criteria: Mutation Testing

- Can we test the ability of a test-suite to detect bugs?

Coverage Criteria: Mutation Testing

- Can we test the ability of a test-suite to detect bugs?
- Idea: *inject* bugs into program

Coverage Criteria: Mutation Testing

- Can we test the ability of a test-suite to detect bugs?
- Idea: *inject* bugs into program Mutation Testing
  - Uses a set of *program mutations* (“mutants”)
  - After designing a test-suite, mutants are applied one-by-one
  - Each mutant should be caught (*killed*) by one of the test cases!

Coverage Criteria: Mutation Testing

- Can we test the ability of a test-suite to detect bugs?
- Idea: *inject* bugs into program Mutation Testing
  - Uses a set of *program mutations* (“mutants”)
  - After designing a test-suite, mutants are applied one-by-one
  - Each mutant should be caught (*killed*) by one of the test cases!
- Typical mutations: simple syntactic modifications
  - Delete a statement
  - Change `&&` to `||`, `-` to `+`, `<` to `<=`, ...
  - Replace variables with others in scope

- **Weak mutation testing**
  Test case must trigger the injected fault and result in an error
- **Strong mutation testing** Test case must trigger the injected fault and result in a failure

- Obstacles:
  - *Equivalent mutants:* Some faults can't be triggered (e.g., changing == to <= in `for (i=10; i==0; i--)`)
  - Also, most "real world" bugs aren't that simple (does mutation testing evaluate the ability of a test-suite to catch "real" bugs?)

Coverage Criteria: Mutation Testing – Fuzzing

- *Fuzzing:* a variation of Mutation Testing
  - "mutate" (or randomly vary) input data
  - monitor program for resulting crashes, failed assertions, memory leaks
  - c.f. fault injection

Summary: Coverage Metrics

- Coverage criteria for when program is “sufficiently” tested
- Widely used, also in certification of safety critical systems
- Are effectively a confidence measure
  - do not guarantee that program is bug-free
  - also, some of the definitions are *ambiguous*
- Never forget:
  - Test-case generation driven by <u>specification</u>, *not* by coverage!