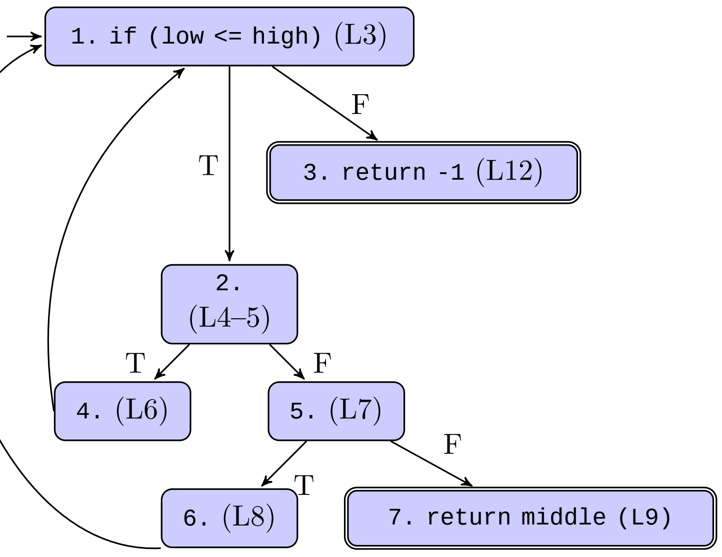
**Software Testing, Quality Assurance and Maintenance** Winter 2017

# Lecture 8 — January 20, 2017

*Patrick Lam* *version 1*

**Larger CFG example.** You can draw a 7-node CFG for this program:

```java
/** Binary search for target in sorted subarray a[low..high] */
int binary_search(int[] a, int low, int high, int target) {
  while (low <= high) {
    int middle = low + (high-low)/2;
    if (target < a[middle)
      high = middle - 1;
    else if (target > a[middle])
      low = middle + 1;
    else
      return middle;
  }
  return -1; /* not found in a[low..high] */
}
```



Here are more exercise programs that you can draw CFGs for.

```java
/* effects: if x==null, throw NullPointerException
            otherwise, return number of elements in x that are odd, positive or both. */
int oddOrPos(int[] x) {
  int count = 0;
  for (int i = 0; i < x.length; i++) {
    if (x[i]%2 == 1 || x[i] > 0) {
      count++;
    }
  }
  return count;
}

// example test case: input: x=[-3, -2, 0, 1, 4]; output: 3
```

Finally, we have a really poorly-designed API (I'd give it a D at most, maybe an F) because it's impossible to succinctly describe what it does. **Do not design functions with interfaces like this.** But we can still draw a CFG, no matter how bad the code is.

```
/** Returns the mean of the first maxSize numbers in the array,
    if they are between min and max. Otherwise, skip the numbers. */
double computeMean(int[] value, int maxSize, int min, int max) {
  int i, ti, tv, sum;

  i = 0; ti = 0; tv = 0; sum = 0;
  while (ti < maxSize) {
    ti++;
    if (value[i] >= min && value[i] <= max) {
      tv++;
      sum += value[i];
    }
    i++;
  }
  if (tv > 0)
    return (double)sum/tv;
  else
    throw new IllegalArgumentException();
}
```

## Statement and Branch Coverage

We defined Control-Flow Graphs so that we can give principled definitions of statement and branch coverage. We can start with the definition of a test path:

**Definition 1** *A test path is a path $p$ (possibly of length 0) that starts at some initial node (i.e. in $N_0$) and ends at some final node (i.e. in $N_f$).*

Here's a definition of coverage for graphs:

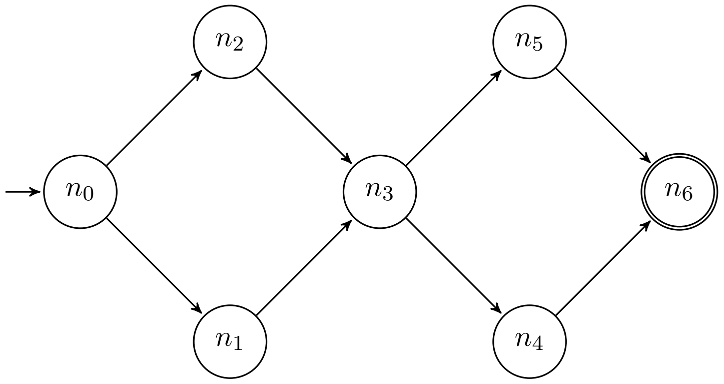
**Definition 2** *Given a set of test requirements TR for a graph criterion $C$, a test set $T$ satisfies $C$ on graph $G$ iff for every test requirement tr in TR, at least one test path $p$ in path$(T)$ exists such that $p$ satisfies tr.*

We'll use this notion to define a number of standard testing coverage criteria. But first, what are test paths?

**Test cases and test paths.** We connect test cases and test paths with a mapping $\text{path}_G$ from test cases to test paths; e.g. $\text{path}_G(t)$ is the set of test paths corresponding to test case $t$.

- usually we just write path since $G$ is obvious from the context.
- we can lift the definition of path to test sets $T$ by defining $\text{path}(T) = \{\text{path}(t) | t \in T\}$.
- each test case gives at least one test path. If the software is deterministic, then each test case gives exactly one test path; otherwise, multiple test cases may arise from one test path.

As an example, consider the double-diamond graph $D$.



Here are the four test paths in $D$:

$$[n_0, n_1, n_3, n_4, n_6]$$
$$[n_0, n_1, n_3, n_5, n_6]$$
$$[n_0, n_2, n_3, n_4, n_6]$$
$$[n_0, n_2, n_3, n_5, n_6]$$

For the *statement coverage* criterion, we get the following test requirements:

$$\{n_0, n_1, n_2, n_3, n_4, n_5, n_6\}$$

That is, any test set $T$ which satisfies statement coverage on $D$ must include test cases $t$; the cases $t$ give rise to test paths path$(t)$, and some path must include each node from $n_0$ to $n_6$. (No single path must include all of these nodes; the requirement applies to the set of test paths.)

Let's formally define statement coverage.

**Definition 3** *Statement coverage: For each node $n \in reach_G(N_0)$, TR contains a requirement to visit node $n$.*

For our example,

$$TR = \{n_0, n_1, n_2, n_3, n_4, n_5, n_6\}.$$

Let's consider an example of a test set which satisfies statement coverage on $D$.

Start with a test case $t_1$; assume that executing $t_1$ gives the test path

$$\text{path}(t_1) = p_1 = [n_0, n_1, n_3, n_4, n_6].$$

Then test set $\{t_1\}$ does not give statement coverage on $D$, because no test case covers node $n_2$ or $n_5$. If we can find a test case $t_2$ with test path

$$\text{path}(t_2) = p_2 = [n_0, n_2, n_3, n_5, n_6],$$

then the test set $T = \{t_1, t_2\}$ satisfies statement coverage on $D$.

What is another test set which satisfies statement coverage on $D$?

Here is a more verbose definition of statement coverage.

**Definition 4** *Test set $T$ satisfies statement coverage on graph $G$ if and only if for every syntactically reachable node $n \in N$, there is some path $p$ in path($T$) such that $p$ visits $n$.*

A second standard criterion is that of branch coverage.

**Criterion 1 Branch Coverage**. *TR contains each reachable path of length up to 1, inclusive, in $G$.*

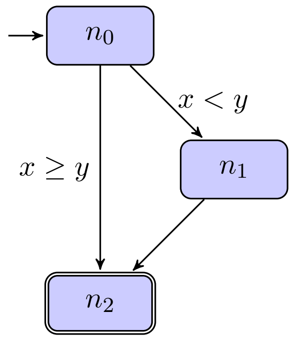
Here are some examples of paths of length $\leq 1$:

Note that since we're not talking about *test paths*, these reachable paths need not start in $N_0$.

In general, paths of length $\leq 1$ consist of nodes and edges. (Why not just say edges?)

Saying "edges" on the above graph would not be the same as saying "paths of length $\leq 1$".

Here is a more involved example:



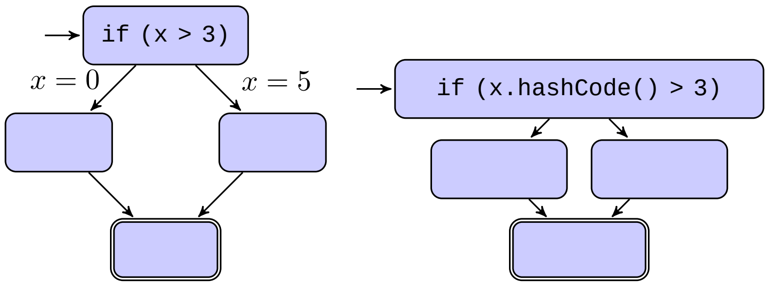
Let's define

$$\begin{aligned} \text{path}(t_1) &= [n_0, n_1, n_2] \\ \text{path}(t_2) &= [n_0, n_2] \end{aligned}$$

Then

$$\begin{aligned} T_1 &= \qquad\qquad \text{satisfies statement coverage} \\ T_2 &= \qquad\qquad \text{satisfies branch coverage} \end{aligned}$$

**Nondeterminism.** I mentioned the mapping between test cases and test paths above. The mapping is not one-to-one for nondeterminstic code. Here's an example of deterministic and non-deterministic control-flow graphs:

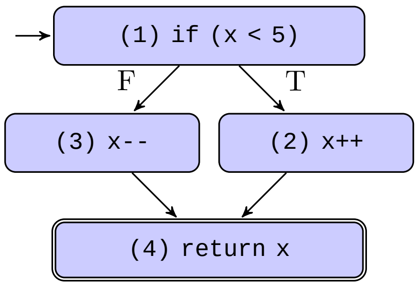


Causes of nondeterminism include dependence on inputs; on the thread scheduler; and on memory addresses, for instance as seen in calls to the default Java `hashCode()` implementation.

Nondeterminism makes it hard to check test case output, since more than one output might be a valid result of a single test input.

**Example.** Here is a short method, the associated control-flow graph, and some test cases and test paths.

```
int foo(int x) {
  if (x < 5) {
    x ++;
  } else {
    x --;
  }
  return x;
}
```



- Test case: $x = 5$; test path: $[(1), (3), (4)]$.
- Test case: $x = 2$; test path: $[(1), (2), (4)]$.

Note that (1) we can deduce properties of the test case from the test path; and (2) in this example, since our method is deterministic, the test case determines the test path.