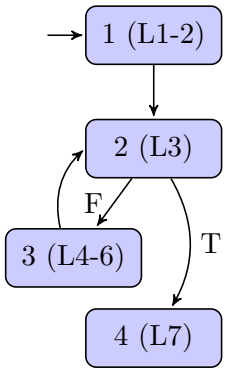
Software Testing, Quality Assurance and Maintenance Winter 2017

Lecture 7 — January 18, 2017

Patrick Lam *version 0*

Basic Blocks. We can simplify a CFG by grouping together statements which always execute together (in sequential programs):



We use the following definition:

Definition 1 *A basic block has one entry point and one exit point.*

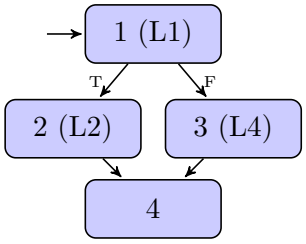
Note that a basic block may have multiple successors. However, there may not be any jumps into the middle of a basic block (which is why statement `l0` has its own basic block.)

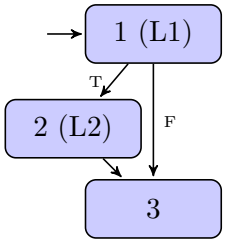
Some Examples

We'll now see how to construct control-flow graph fragments for various program constructs.

if statements: One can put the conditions (and hence uses) on the control-flow edges, rather than in the `if` node. I prefer putting the condition in the node.

```
1  if (z < 17)
2    print (x);
3  else
4    print (y);
```



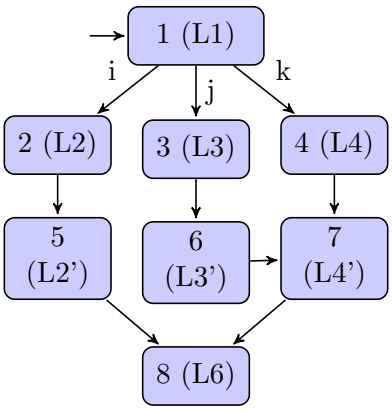



```
1  if (z < 17)
2    print(x);
```

Short-circuit `if` evaluation is more complicated; I recommend working it out yourself.

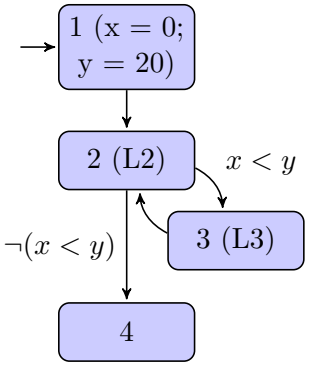
case / switch statements:

```
1  switch (n) {
2    case 'I': ...; break;
3    case 'J': ...; // fall thru
4    case 'K': ...; break;
5  }
6  // ...
```



while statements:

```
1  x = 0; y =
       20;
2  while (x < y)
       {
3    x ++; y --;
4  }
```



Note that arbitrarily complicated structures may occur inside the loop body.

for statements:

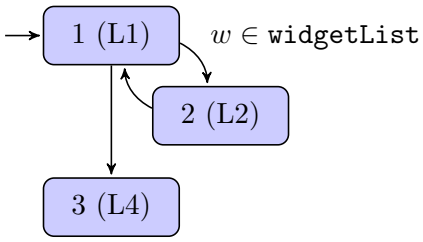
```
1  for (int i = 0; i < 57; i++) {
2    if (i % 3 == 0) {
3      print (i);
4    }
5  }
```

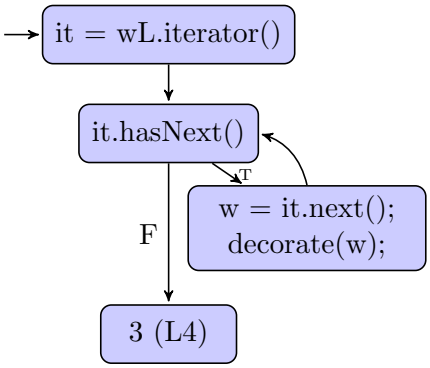
(an exercise for the reader;
we saw one earlier!)

This example uses Java's enhanced for loops, which iterates over all of the elements in the `widgetList`:

```
1  for (Widget w : widgetList) {
2    decorate(w);
3  }
4  // ...
```

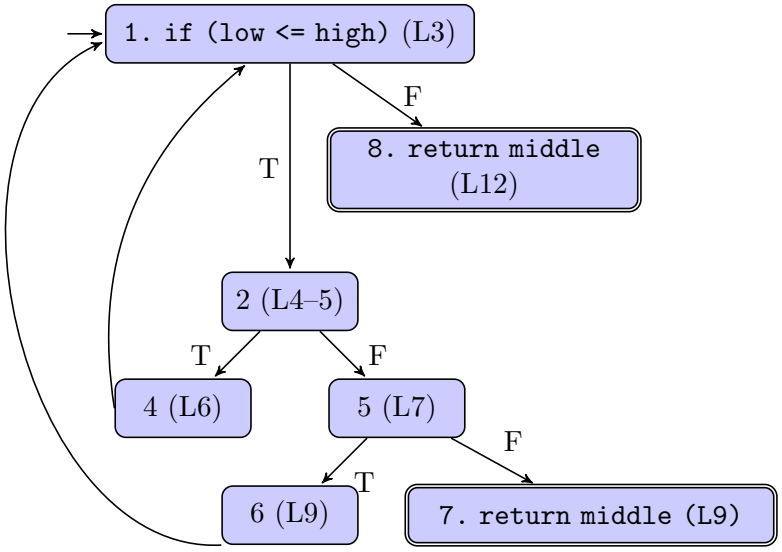
I will accept the simplified CFG or the more useful one on the right:





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

```
1   /** Binary search for target in sorted subarray a[low..high] */
2   int binary_search(int[] a, int low, int high, int target) {
3     while (low <= high) {
4       int middle = low + (high-low)/2;
5       if (target < a[middle)
6         high = middle - 1;
7       else if (target > a[middle])
8         low = middle + 1;
9       else
10        return middle;
11    }
12    return -1; /* not found in a[low..high] */
13  }
```



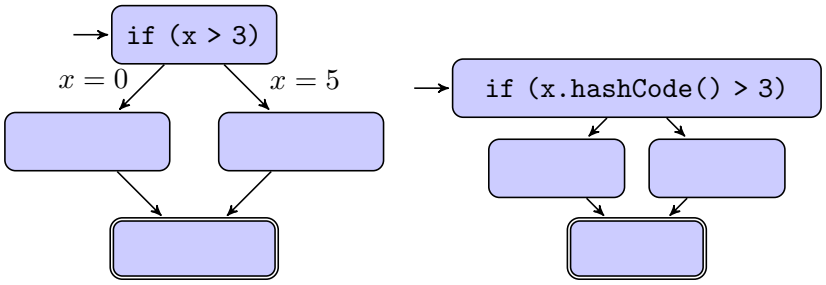
Here are more exercise programs that you can draw CFGs for.

```
1  /* effects: if x==null, throw NullPointerException
2              otherwise, return number of elements in x that are odd, positive or both. */
3  int oddOrPos(int[] x) {
4    int count = 0;
5    for (int i = 0; i < x.length; i++) {
6      if (x[i]%2 == 1 || x[i] > 0) {
7        count++;
8      }
9    }
10   return count;
11 }
12
13 // 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.

```
1   /** Returns the mean of the first maxSize numbers in the array,
2       if they are between min and max. Otherwise, skip the numbers. */
3   double computeMean(int[] value, int maxSize, int min, int max) {
4     int i, ti, tv, sum;
5
6     i = 0; ti = 0; tv = 0; sum = 0;
7     while (ti < maxSize) {
8       ti++;
9       if (value[i] >= min && value[i] <= max) {
10        tv++;
11        sum += value[i];
12      }
13      i++;
14    }
15    if (tv > 0)
16      return (double)sum/tv;
17    else
18      throw new IllegalArgumentException();
19  }
```

Here's an example of deterministic and nondeterministic control-flow graphs:



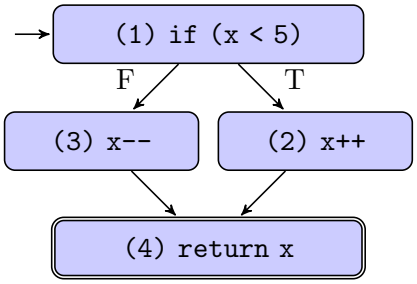
Causes of nondeterminism include dependence on inputs; on the thread scheduler; and on memory addresses, for instance as seen in calls to the default Java `hashCode()` implementation.

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

Indirection. Note that we will describe coverage criteria with respect to *test paths*, but we always run *test cases*.

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

```
1  int foo(int x)
       {
2    if (x < 5) {
3      x ++;
4    } else {
5      x --;
6    }
7    return x;
8  }
```



- 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.

Graph Coverage

Having defined all of the graph notions we'll need for now, we apply them to graphs. Recall our previous definition of coverage:

Definition 2 *Given a set of test requirements* TR *for a coverage criterion C, a test set T* satisfies *C iff for every test requirement* tr *in* TR*, at least one t in T exists such that t satisfies* tr*.*

We apply this definition to graph coverage:

Definition 3 *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. (At this point, the textbook defines predicates, but mostly ignores them afterwards. I'll just ignore them right away.)

Recall the double-diamond graph D which we saw earlier. For the *node 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 node 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 node coverage.

Definition 4 *Node coverage: For each node $n \in reach_G(N_0)$,* TR *contains a requirement to visit node n.*

We will state all of the coverage criteria in the following form:

Criterion 1 *Node Coverage (NC):* TR *contains each reachable node in G.*

We can then write

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

Let's consider an example of a test set which satisfies node coverage on D, the double-diamond graph from last time.

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 node 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 node coverage on D.

What is another test set which satisfies node coverage on D?

Here is a more verbose definition of node coverage.

Definition 5 *Test set T satisfies* node 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 edge coverage.

Criterion 2 Edge Coverage *(EC). TR contains each reachable path of length up to 1, inclusive, in G.*

We describe edge coverage this way so that, as far as possible, new criteria in a series will subsume previous criteria.

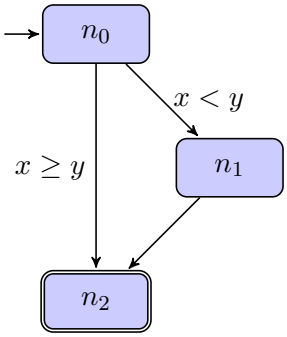
Here are some examples of paths of length ≤ 1:

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

In general, paths of length ≤ 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 ≤ 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

$$\begin{aligned} T_1 &= \text{satisfies node coverage} \\ T_2 &= \text{satisfies edge coverage} \end{aligned}$$