Software Testing, Quality Assurance and Maintenance Winter 2015

Lecture 15 — February 6, 2015

Patrick Lam *version 1*

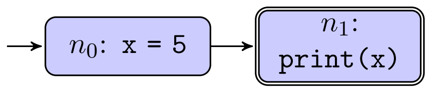
Data flow Criteria

So far we've seen structure-based criteria which imposed test requirements solely based on the nodes and edges of a graph. These criteria have been oblivious to the contents of the nodes.

However, programs mostly move data around, so it makes sense to propose some criteria based on the flow of data around a program. We'll be talking about *du*-pairs, which connect definitions and uses of variables.

```
         [ x = 5           // def(x)
du-pair  | ⋮
         [ print(x)        // use(x)
```

Let's look at some graphs.



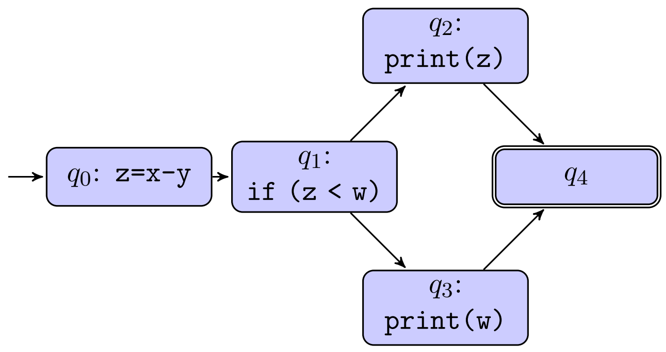
We write

$$\text{def}(n_0) = \text{use}(n_1) = \{x\}.$$

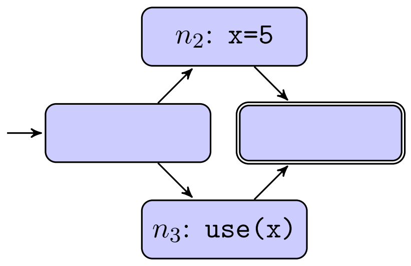
Note that edges can also have defs and uses, for instance in a graph corresponding to a finite state machine. In that case, we could write:

$$\text{use}(n_0, n_1) = \{\}.$$

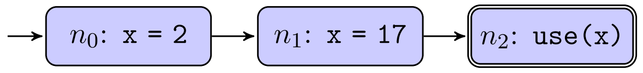
Here's another example.



A particular def d of variable x may (or may not) *reach* a particular use u. If a def may reach a particular use, then there exists a path from d to u which is free of redefinitions of x. In the following graph, the def at n_2 does not reach the use at n_3, since no path goes from n_2 to n_3.



Another example of a definition which does not reach:



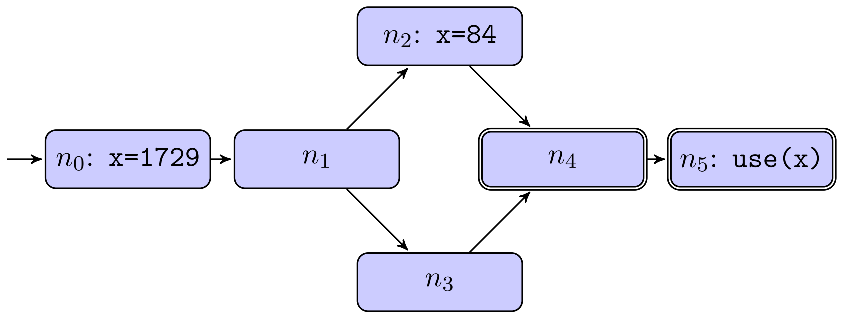
We say that the definition at n_1 *kills* the definition at n_0, so that def(n_0) does not reach n_2. We are therefore looking for def-clear paths.

Definition 1 *A path p from ℓ_1 to ℓ_m is* def-clear *with respect to variable v if for every node n_k and every edge e_k on p from ℓ_1 to ℓ_m, where $k \neq 1$ and $k \neq m$, then v is not in def(n_k) or in def(e_k).*

That is, nothing on the path p from location ℓ_1 to location ℓ_m redefines v. (Locations are edges or nodes.)

Definition 2 *A def of v at ℓ_i reaches a use of v at ℓ_2 if there exists a def-clear path from ℓ_i to ℓ_j with respect to v.*

Quick poll: does the def at n_0 reach the use at n_5?



Building on the notion of a def-clear path:

Definition 3 *A* du-path *with respect to v is a simple path that is def-clear with respect to v from a node n_i, such that v is in def(n_i), to a node n_j, such that v is in use(n_j).*

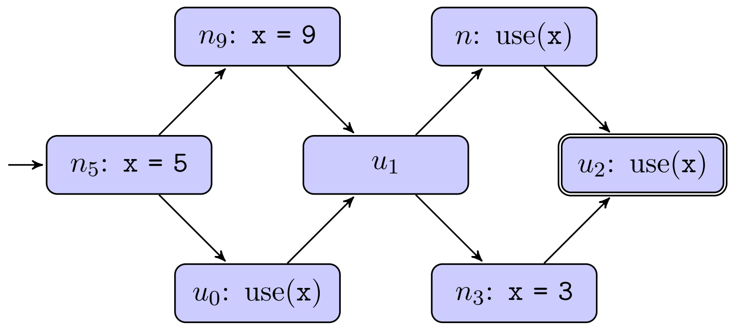
(This definition could be easily modified to use edges e_i and e_j).

Note the following three points about *du*-paths:

- associated with a variable
- simple (otherwise there are too many)
- may be any number of uses in a du-path

Coverage criteria using du-paths

We next create groups of *du*-paths. Consider again the following double-diamond graph D:



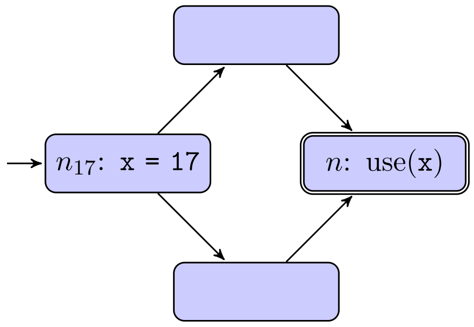
We will define two sets of *du*-paths:

- def-path sets: fix a def and a variable, e.g.
  - du$(n_5, x) =$
  - du$(n_3, x) =$
- def-pair sets: fix a def, a use, and a variable, e.g. du$(n_5, n, x) =$

These sets will give the notions of all-defs coverage (tour at least one *du*-path from each def-path set—a weak criterion) and all-uses coverage (tour at least one *du*-path from each def-pair set).

How can there be multiple elements in a def-pair set?

Here's an example with two *du*-paths in a def-pair set.



We then have

$$\text{du}(n_{17}, n, x) =$$

Note the general relation

$$\text{du}(n_i, v) = \bigcup_{n_j} \text{du}(n_i, n_j, v)$$

There are more def-pair sets than def-path sets. Cycles are always allowed as *du*-paths, as long as the *du*-path is simple; you can always tour a *du*-path with a non-simple path, of course.

Useful exercise. Create an example where one def-path set splits into several def-pair sets; you can get a smaller example than the one in the book.

We can use the above definitions to provide coverage criteria.

Criterion 1 All-Defs Coverage *(ADC). For each def-path set $S = du(n, v)$, TR contains at least one path d in S.*

Criterion 2 All-Uses Coverage *(AUC). For each def-pair set $S = du(n_i, n_j, v)$, TR contains at least one path d in S.*

What do these criteria mean? For each def,

- ADC: reach at least one use;
- AUC: reach every use somehow;

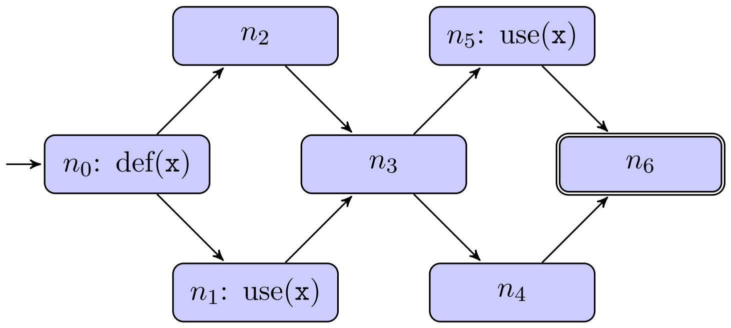
In the context of the earlier example,

- ADC requires:
- AUC requires:

Nodes versus edges. So far, we've assumed definitions and uses occur on nodes.

- uses on edges ("*p*-uses") work as well;
- defs on edges are trickier, because a *du*-path from an edge to an edge may not be simple. (We could make things work out with more work.)

Another example.



Some test sets that meet these criteria:

- ADC:
- AUC: