

CS 4350: Fundamentals of Software Engineering

Lesson 5.2 Evaluating Tests

Jonathan Bell, Adeel Bhutta, Ferdinand Vesely, Mitch Wand
Khoury College of Computer Sciences



Learning Objectives for this Lesson

- By the end of this lesson, you should be able to:
 - Explain what makes a good test, and give examples and counter examples
 - Explain different things a test suite might accomplish, and sketch how one might judge how well a test suite accomplishes those goals

Review: Three Purposes of Tests

- Test Driven Development
- Regression Test
 - Prevent bugs from (re-)entering during maintenance.
- Acceptance Test
 - Customer-level requirement testing
 - Validation: Are we building the right system ?

*These purposes are
copied from Lesson 5.1*

What makes a Test Good

- Tests should be **hermetic**
 - Reduce flakiness.
- Tests should be **clear**
 - After failure, should be clear what went wrong.
- Tests should be scoped as **small** as possible
 - Faster and more reliable.
- Tests should make calls against **public** APIs
 - Or they become brittle.

For a fuller treatment:

https://learning.oreilly.com/library/view/software-engineering-at/9781492082781/ch12.html#unit_testing

What makes a Test Bad?

- *Flaky* tests are those that fail intermittently:
 - Nondeterminism (e.g., hash codes, random numbers);
 - Timing issues (e.g., threads, network).
 - Availability of Resources
- *Brittle* tests are those that are not self-contained:
 - Ordering of tests (e.g., assume prior state)
- *Mystery* tests aren't clear why they fail:
 - Too complicated
 - Not enough information provided
 - Tests too much code at a time
- All these impede maintenance:
 - A capricious, rigid or incomprehensible gatekeeper impedes the ability to make progress.

These are sometimes referred to as "test smells"

Example of a *Flaky* Test

```
it('writes right', () => {  
  const w = fs.createWriteStream('test.txt');  
  const t = createBigTree();  
  t.write(w);  
  w.end();  
  const d = fs.readFileSync('test.txt');  
  /* ... check result ... */  
})
```

Problem:

- Here we are assuming “test.txt” is writable and not being used by something else (e.g., this same test being run in parallel).
- Test may fail for reasons unrelated to the code being tested.

What else is wrong with this test?

Example of a *Mystery* Test

```
it('remove only removes one', () =>{
  const tree = makeBST();
  for (let i = 0; i < 1000; ++i) {
    tree.add(i);
  }
  for (let j = 0; j < 1000; ++j) {
    for (let i = 0; i < 1000; ++i) {
      if (i !== j) tree.remove(i);
    }
    expect(tree.contains(j)).
      toBe(true);
  }
})
```

Problem:

- Test is hard to understand
- Testing too much code in one test
- If it fails, no clue as to what went wrong:
 - “false is not true”
- Test code has conditionals/loops

(Incidentally, also suffers from hard-coding 1000 in the test.)

Example of a *Brittle* test

```
it('removes max', ()=>{  
  tree.remove(31);  
  expect(tree.size()).  
    toBe(4);  
})
```

Problem

- Assumes (mutable) context.
- Uses information unknown to test;
- Test will mis-behave if tests/tree are reordered

What makes a Test Suite good?

- Depends on the **purpose** of the test suite.
- Test Driven Development
 - Does the SUT satisfy its specification? (“functional testing”)
- Regression Test
 - Did something change since some previous version?
 - Prevent bugs from (re-)entering during maintenance.
- Acceptance Test
 - Does the SUT satisfy the customer (requirement testing)
 - Validation: Are we building the right system ?

Does the SUT satisfy its specification?

- Test behavior without regard to the implementation (“black-box testing” or “functional testing”).
- What’s a specification?:
 - A precise definition of all acceptable behaviors of a SUT (outputs, state mutation, other effects) in **all** situations (state and inputs)
 - A specification may be formal (mathematical), informal (natural language) or implicit (“I know it when I see it”).
- A test suite is an approximation to an unwritten specification
 - That’s the “T” in TDD
 - Adequacy of test suite is likelihood that an implementation passing all the tests actually fulfills the (unwritten) specification.

Not often seen in the wild

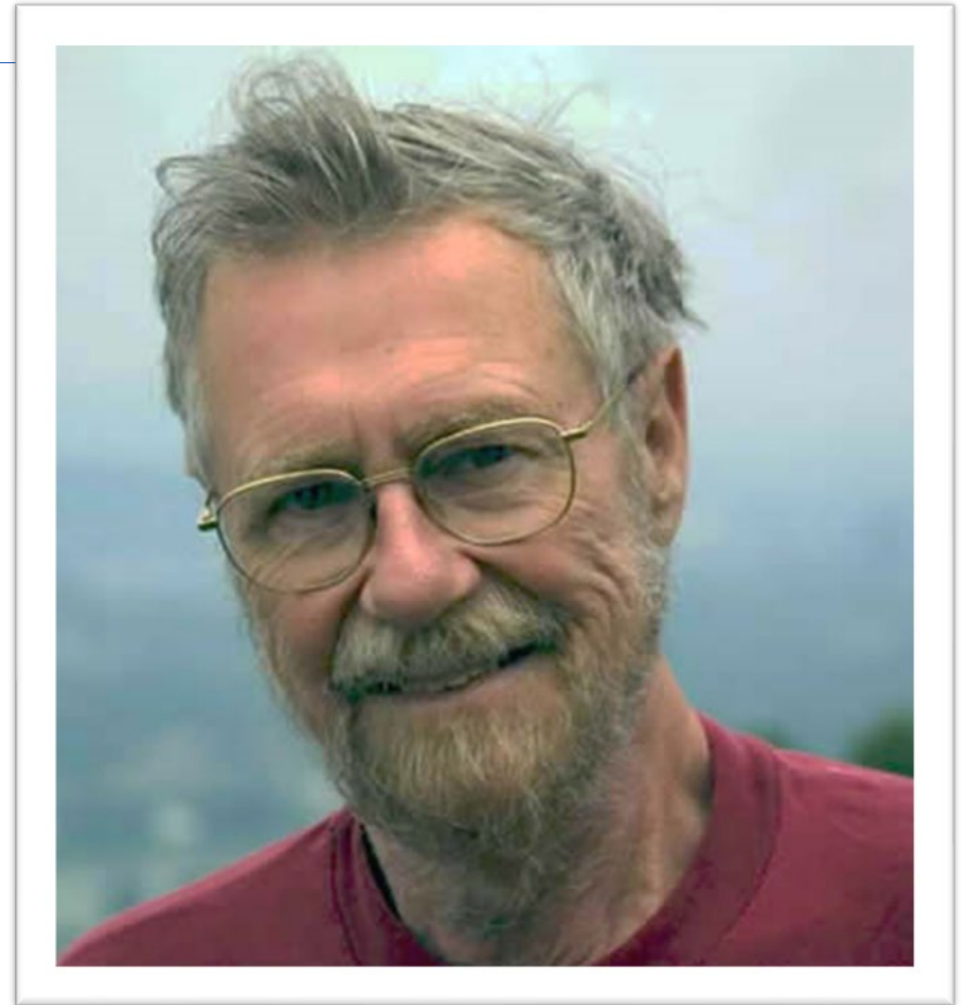
Remember Dijkstra?

“Program testing can be used to show the presence of bugs, but never to show their absence!”

– Edsger Dijkstra

- The state space of a SUT is (usually) infinite, but testing can only execute a finite number of tests.
- Even if the state space is finite, it may still be too large to make exhaustive testing feasible.

And this ignores the fallibility of tests.
What if the tests are in error?



Needles in a Haystack

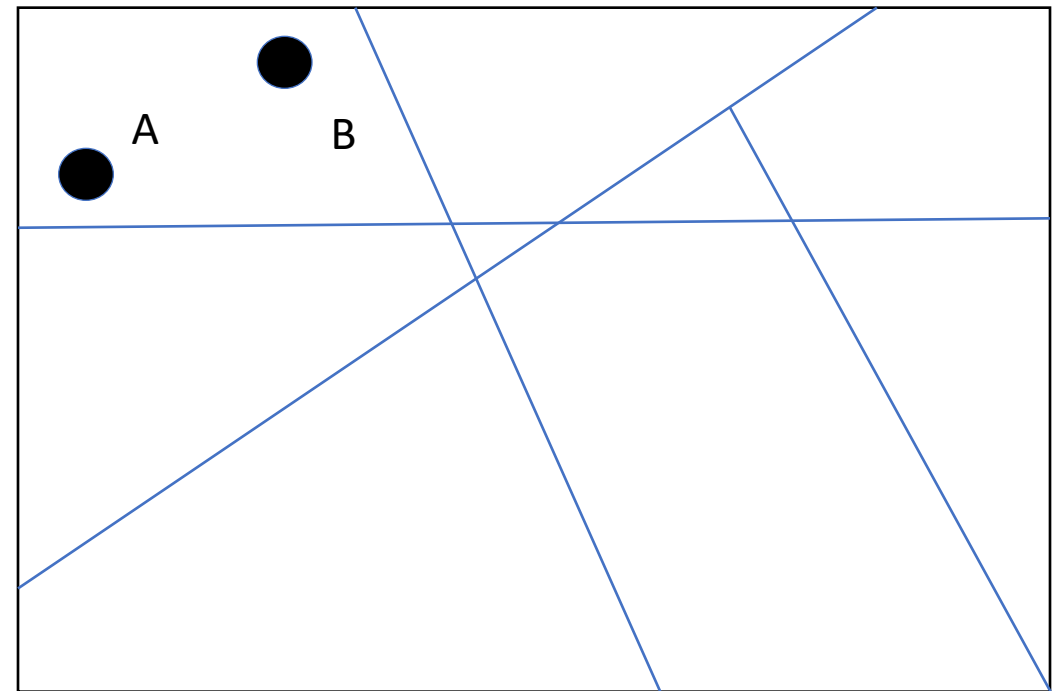
- To find needles, look systematically
- We need to find out *what makes needles special*



So which of these infinitely many behaviors should we check?

- Divide the behaviors into regions of similar behavior called *equivalence classes*
- Which points in a region should we choose?
- It's fair to assume that if one point in the region works, then the others will.

=> ***partition testing***



If the program works for input A, it will probably work for input B

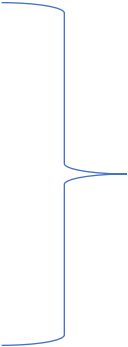
Make sure the regions have the right boundaries.

- Select “special” values of a range
 - Boundary values;
 - Barely legal, barely illegal inputs;

=> ***boundary testing***
- Integer overflow a serious problem: may be implicit
 - ComAir problem due to a list getting more than 32767 elems
- <https://arstechnica.com/uncategorized/2004/12/4490-2/>



Do our tests check all of the code?

- This is the question of test *coverage*.
 - Statement coverage
 - Branch coverage
 - Path coverage
 - ...

“*Structural testing*”
- Quantitative measurement is possible.

How do you compute Coverage?

- Coverage is computed automatically while the tests execute
- jest --coverage
 - Does it all for you

```
calculator/add
  ✓ should return a number when parameters are passed to `add()`
  ✓ should return sum of `2` when 1 + 1 is passed to `add()`

calculator/subtract
  ✓ should return a number when parameters are passed to `subtract()`
  ✓ should return sum of `1` when 2 - 1 is passed to `subtract()`

4 passing (4ms)
```

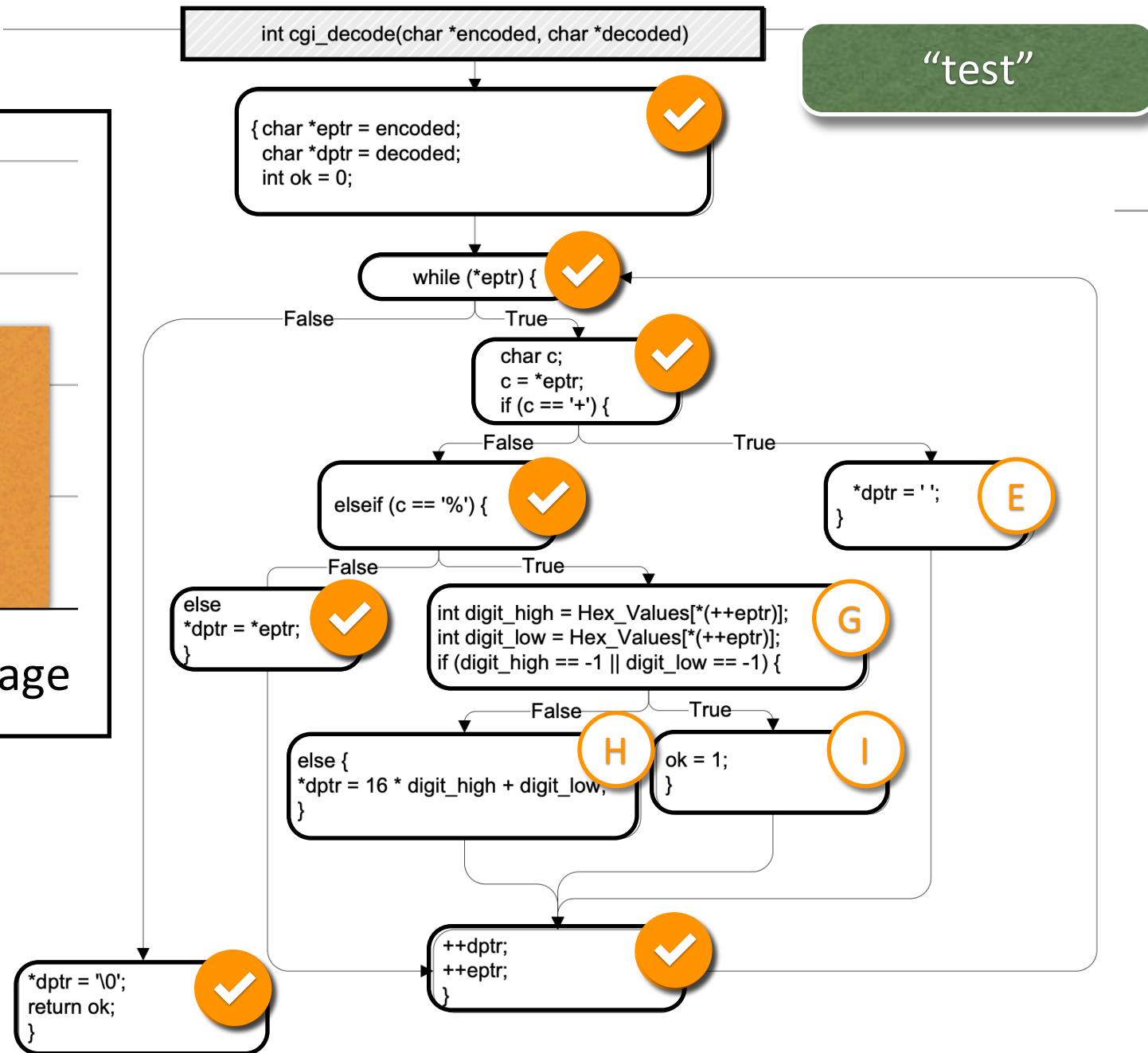
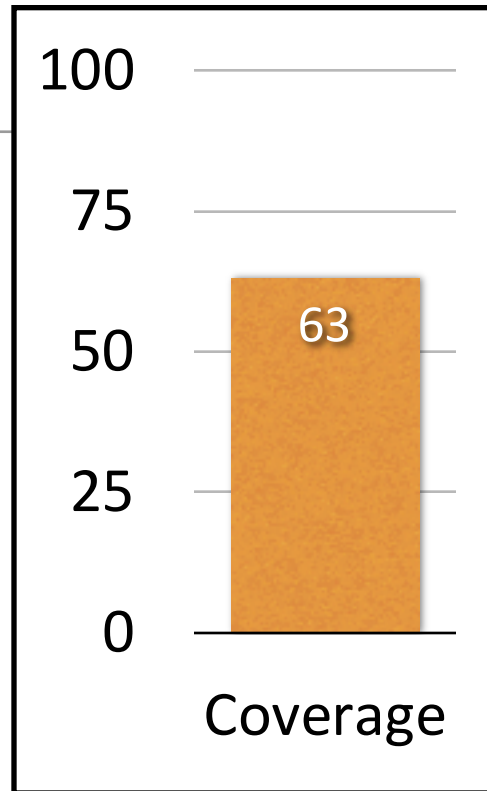
File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	100	100	100	100	
Add.ts	100	100	100	100	
Subtract.ts	100	100	100	100	

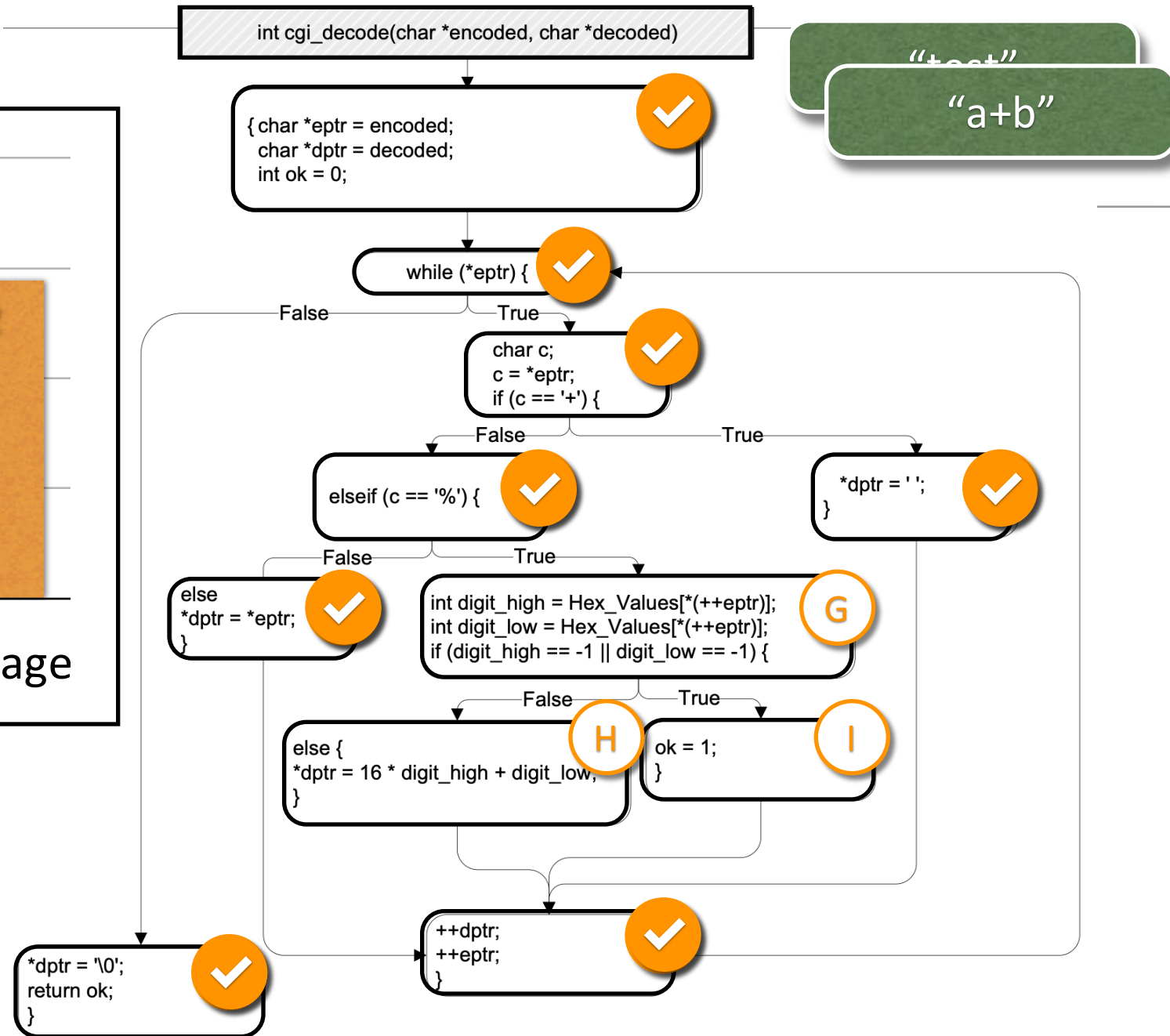
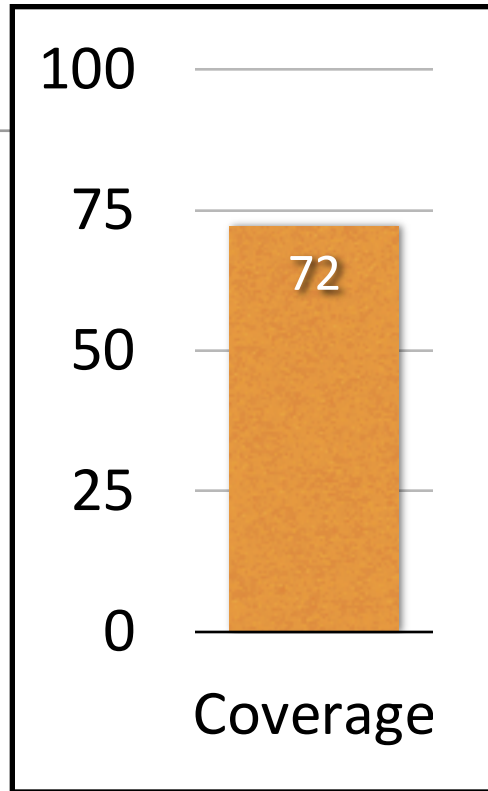
*see example at <https://github.com/philipbeel/example-typescript-nyc-mocha-coverage>

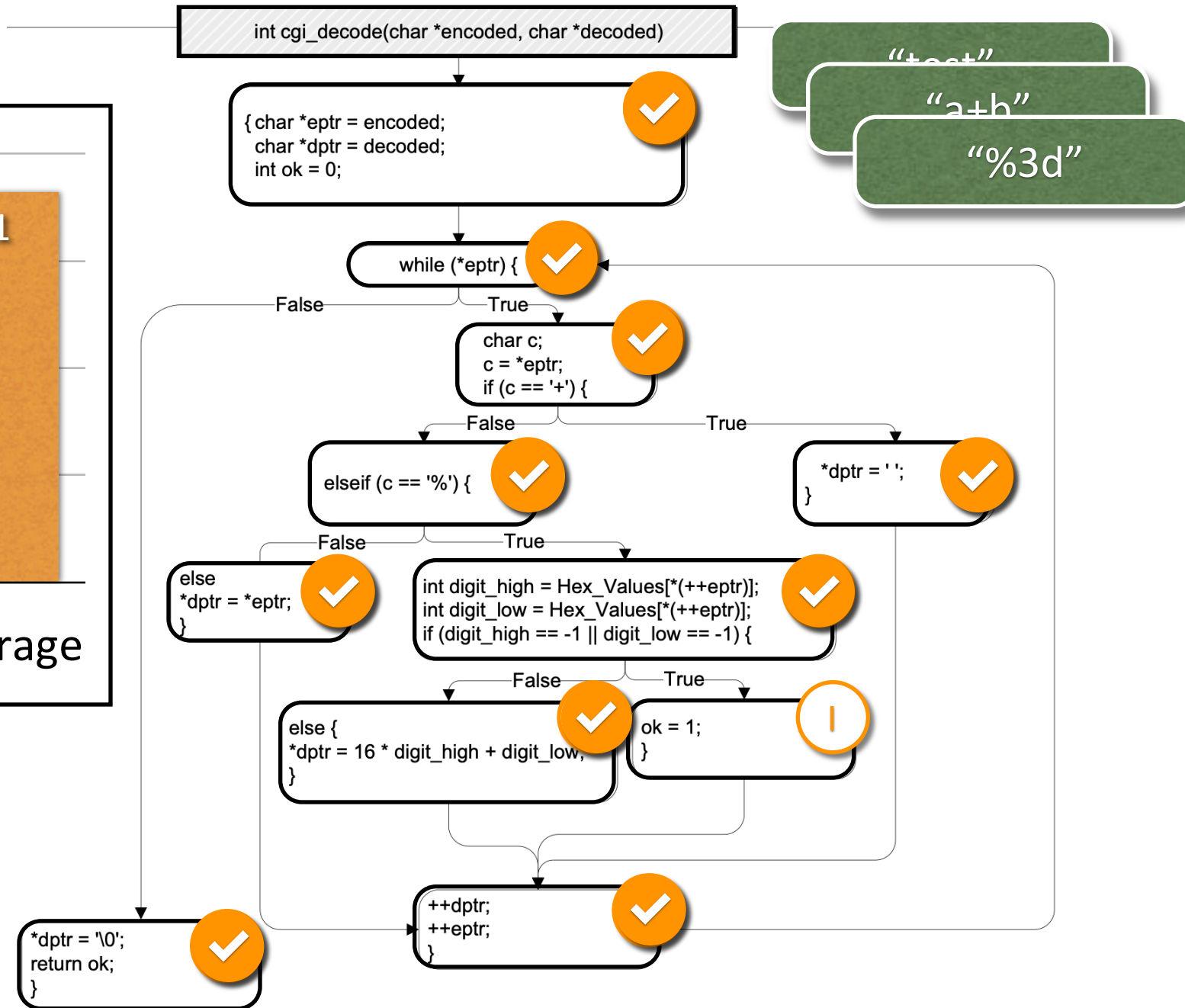
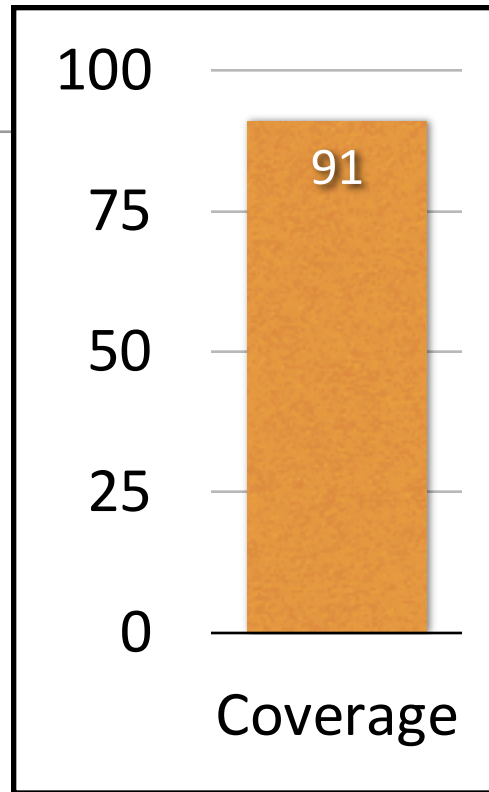
Statement Coverage

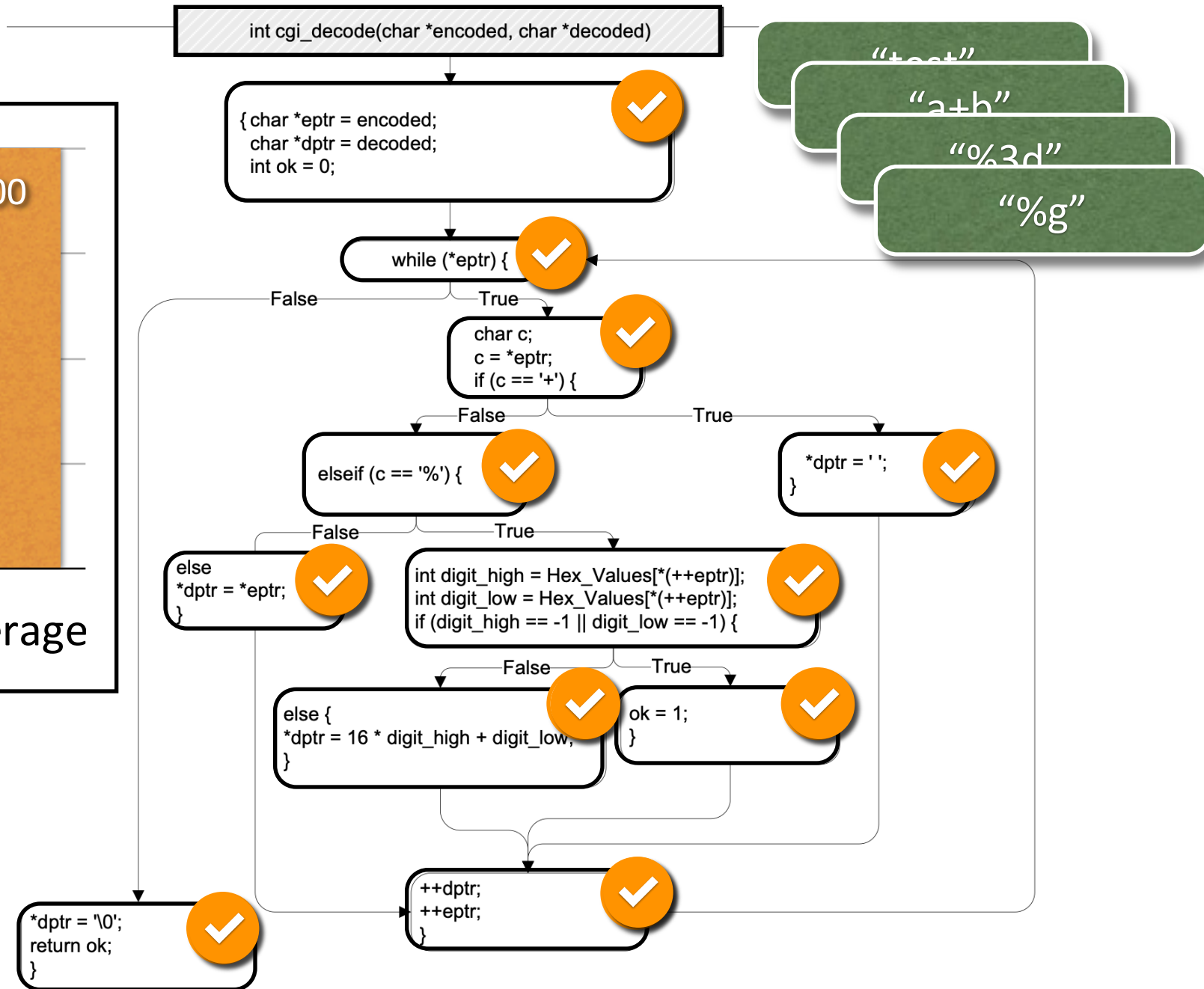
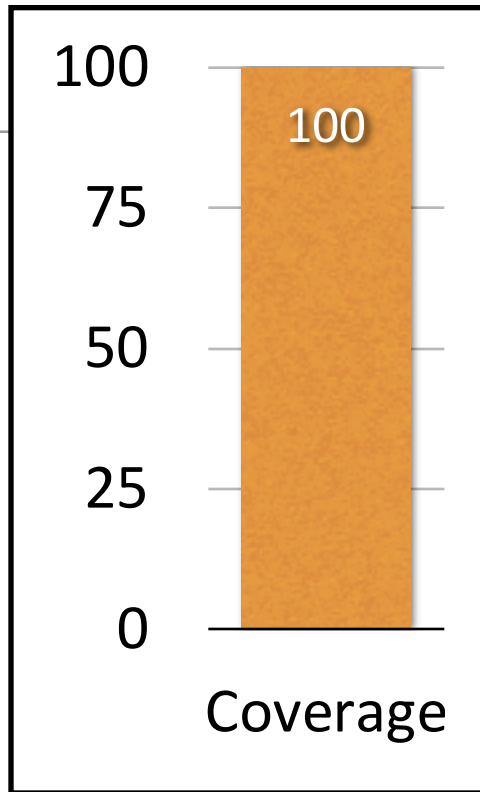
- Each line (or part of) the code (or node in CFG) should be executed at least once in the test suite
- There are good tools for measuring how many lines were executed or not executed
 - Jest -- coverage
- Adequacy criterion: *each statement must be executed at least once*

Coverage: $\frac{\# \text{ executed statements}}{\# \text{ statements}}$







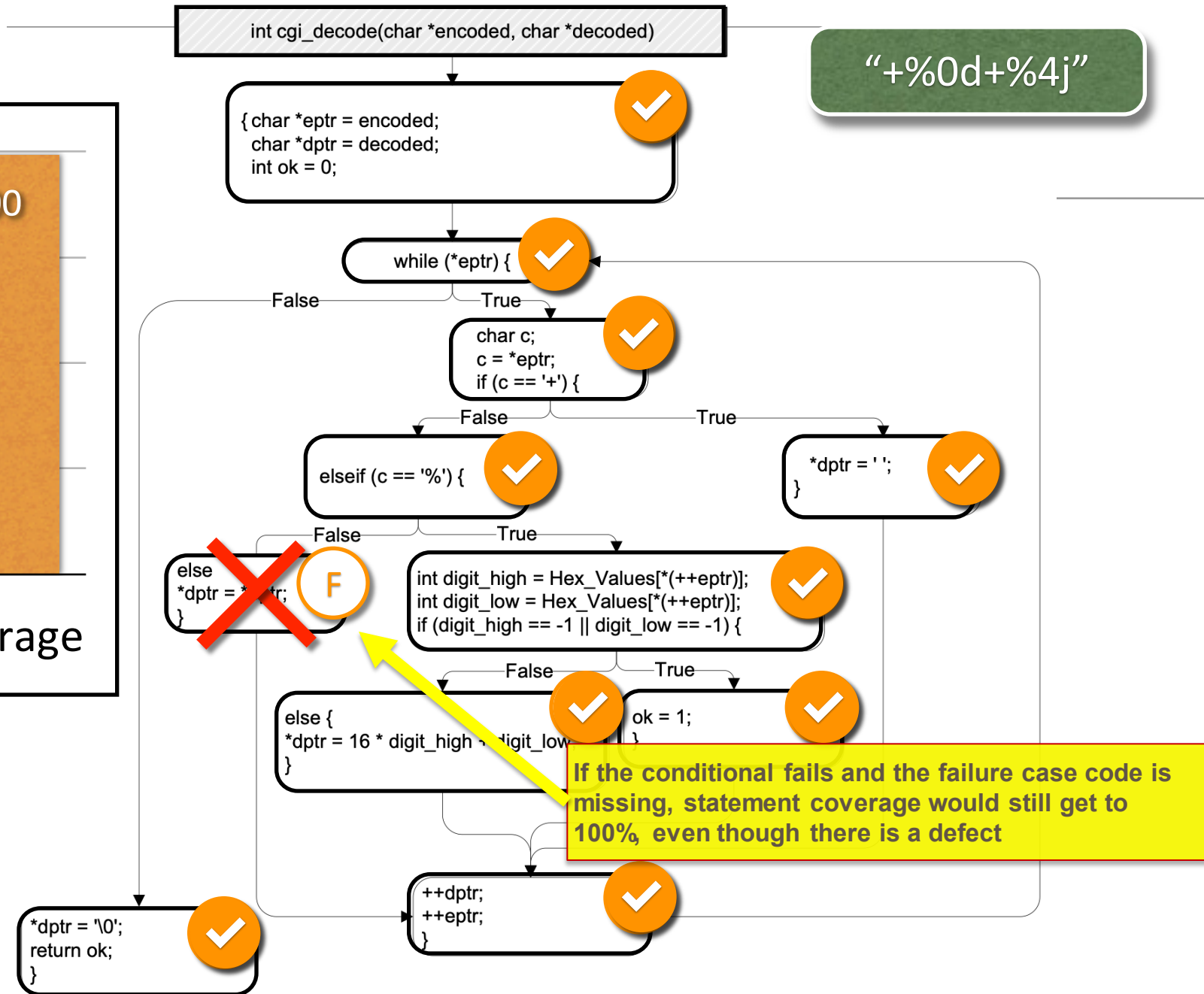
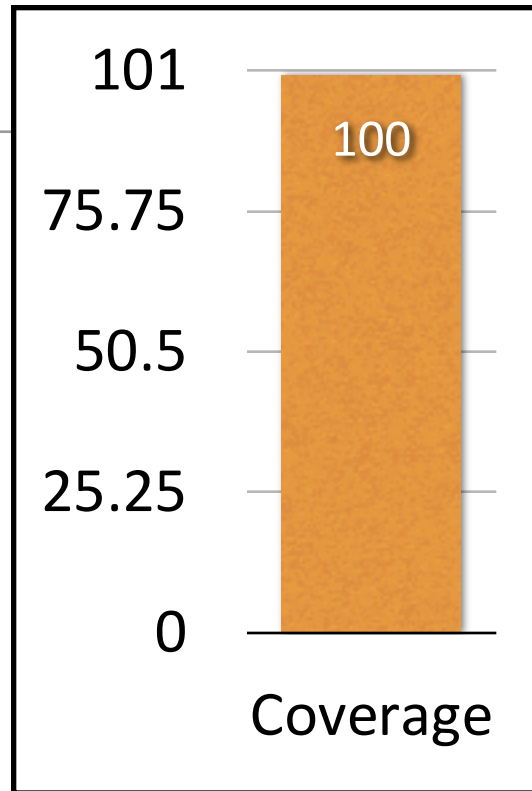


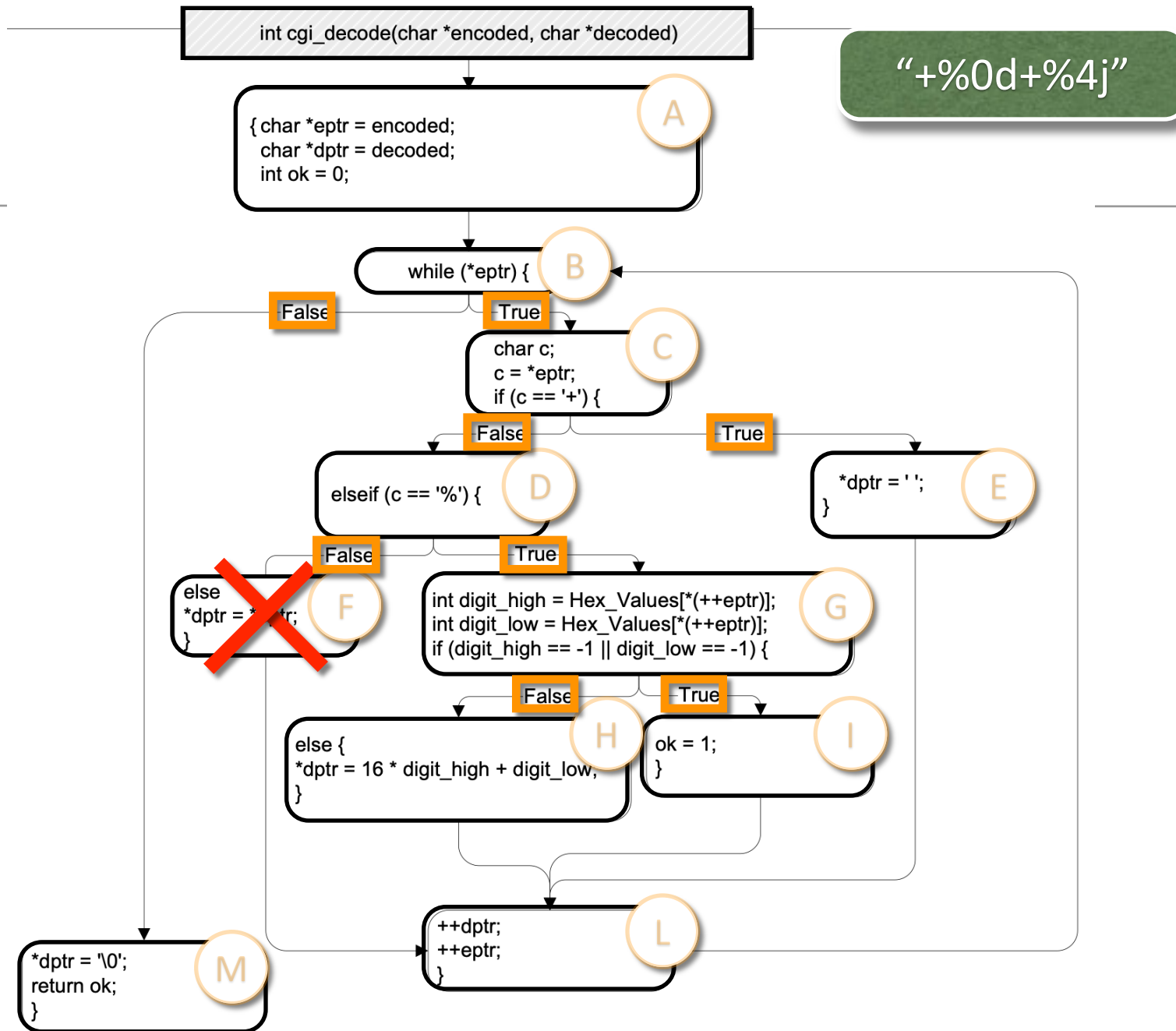
Branch Coverage

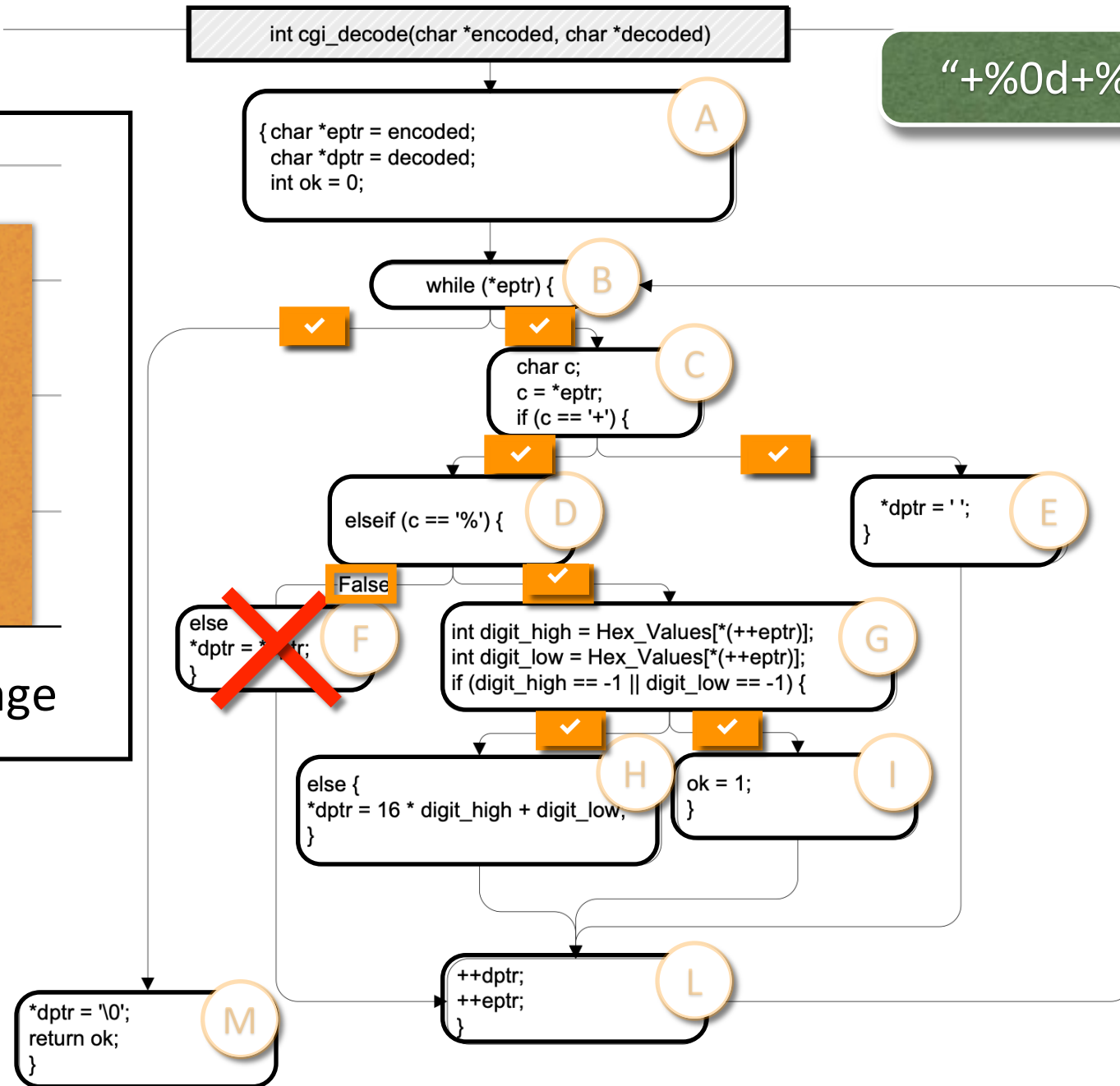
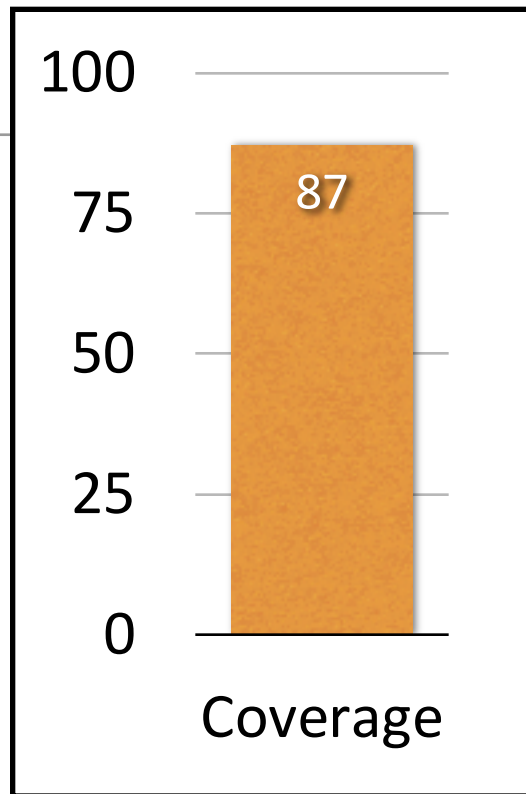
- Adequacy criterion: *each branch in the CFG must be executed at least once*

$$\text{coverage: } \frac{\# \text{ executed branches}}{\# \text{ branches}}$$

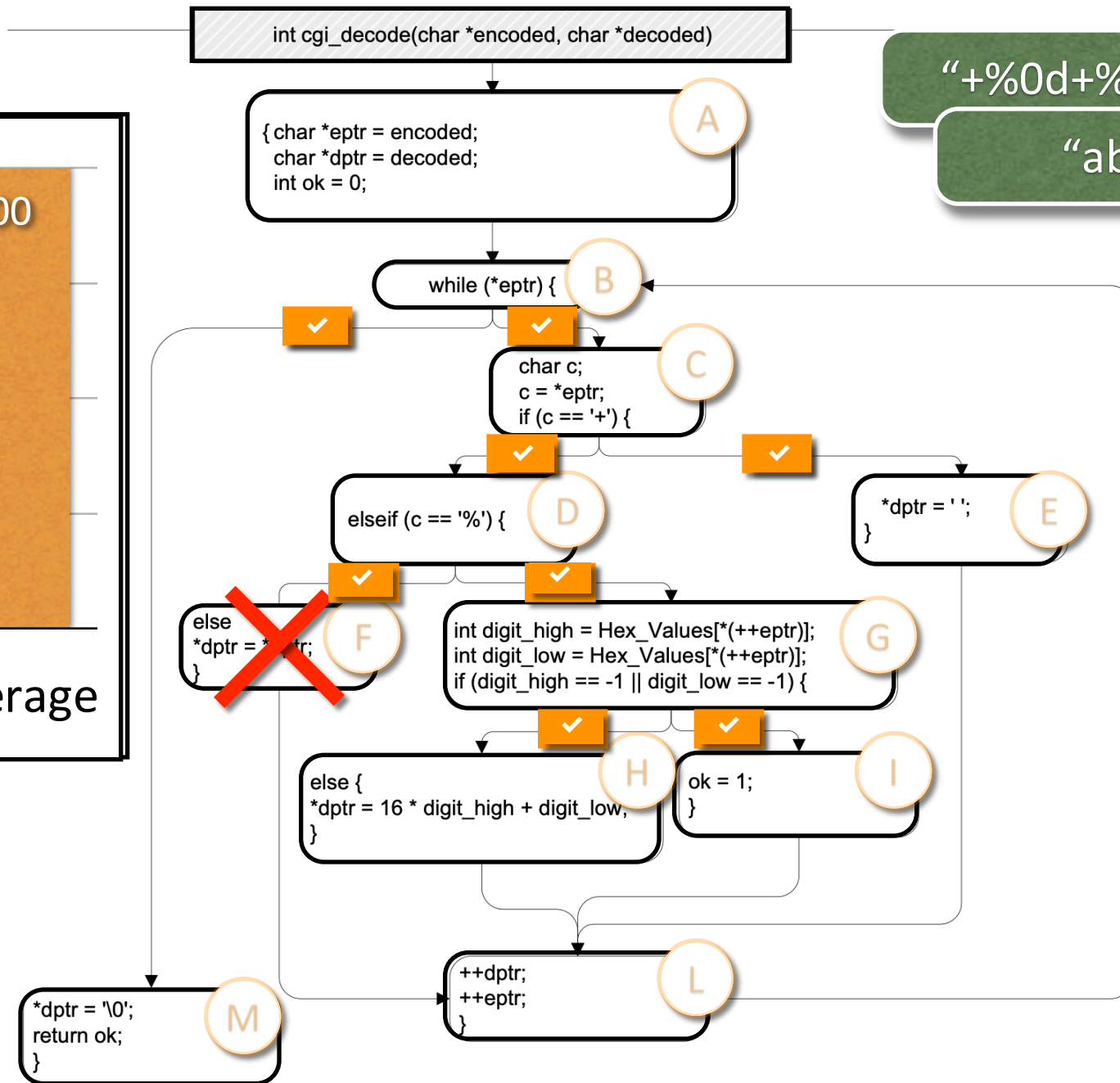
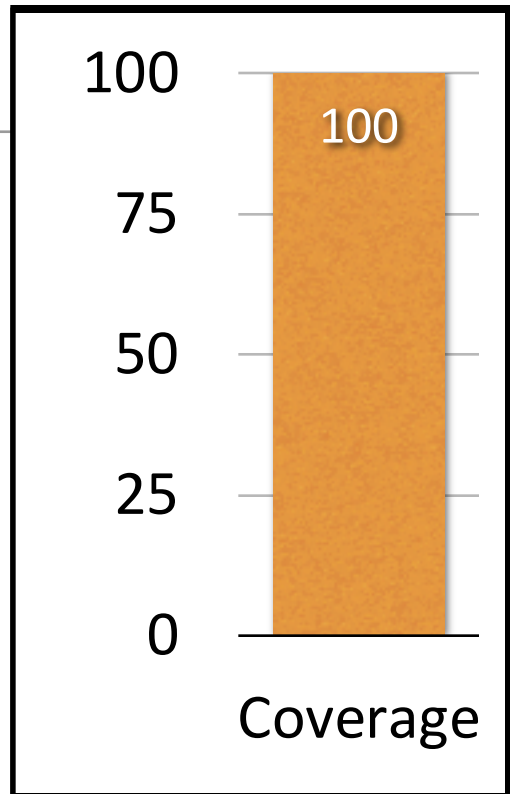
- Subsumes statement testing criterion because traversing all edges implies traversing all nodes
- Most **widely used criterion in industry**







“+%0d+%4j”

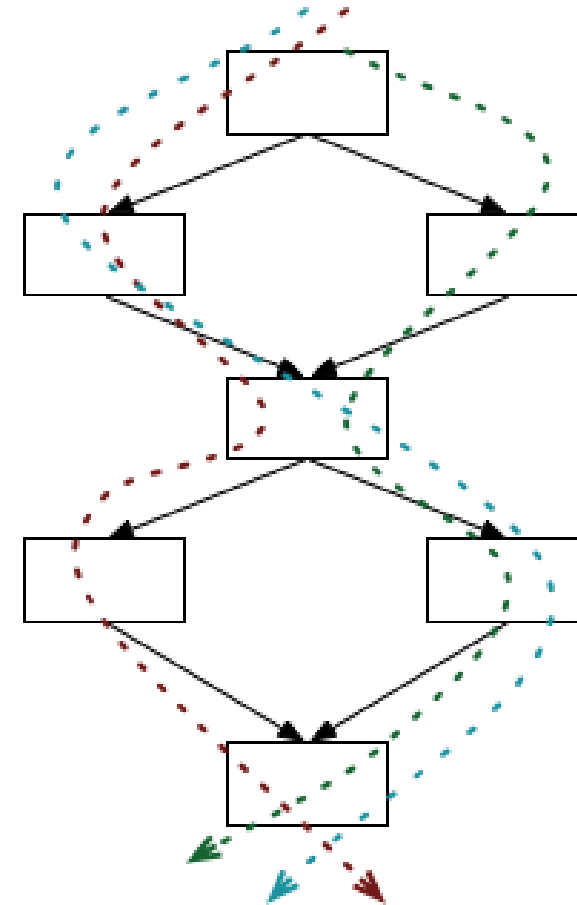


“+ %0d+ %4j”

“abc”

Path Coverage

- Sometimes a fault is only manifest on a particular path
 - E.g., choosing the left branch and then choosing the right branch. (dashed blue path)
- But the number of paths can be infinite
 - E.g., if there is a loop.
- There are ways to bound the number of paths to cover.



Mutation Testing is a way of checking to see whether you've tested "enough" paths.

- Test framework mutates the code in the SUT
 - E.g., replacing “&&” with “||” in an “if” statement.
- Then we see if the test suite fails.
- If the test suite still passes, that means we don't have enough tests.
- Difficult in practice:
 - Too many mutants possible (time)
 - Too many mutants are equivalent or uninteresting:
 - `rpc.set_deadline(10);` → `rpc.set_deadline(20);`

But possible!
<https://research.google/pubs/pub46584/>

100% Coverage may be Impossible

- Path coverage (even without loops)
 - Dependent conditions: if (x) A; B; if (x) C;
- Branch coverage
 - Dead Branches e.g., if (x < 0) A; else if (x == 0) B; else if (x > 0) C;
 - (x > 0) test will always succeed
- Statement coverage
 - Dead code (e.g., defensive programming)

There are many other ways to judge the Adequacy of Structural Tests

1. Path coverage (usually impossible) *implies*
 2. Branch Coverage *implies*
 3. Block Coverage = Statement coverage.
- (Other coverage criteria exist, some incomparable)

See https://en.wikipedia.org/wiki/White-box_testing

What if the purpose of your test suite is regression testing?

- Regression tests control maintenance:
 - A change cannot be committed until “all” tests pass.
 - Often “all tests” means “all *small automated unit* tests”
- Adequacy includes whether tests cover all *uses*:
 - Uses may include unspecified behavior:
 - e.g., Users may assume that a hash result is non-negative;
 - Hyrum’s law: any visible behavior may have dependents.
- Users are responsible to add tests:
 - **Beyoncé rule**: “If you liked it you should have put a ~~ring~~ **test** on it” (SoftEng @ Google)

Adequacy of Acceptance Tests



- Crucial: meet with prospective customers.
- This is difficult, time-consuming and expensive.
- But building the wrong product is much worse!

Supplement to Acceptance Evaluation

- *Dogfooding* (“Eat your own dogfood”)
- Be your own customer.
- Weaknesses:
 - Employees unrepresentative of customers
 - Whether someone can be compelled to use a product does not say whether they would purchase it.



Review

- Now that you've studied this lesson, you should be able to:
 - Explain some properties of good tests.
 - Distinguish flaky, brittle or Mystery tests;
 - Describe measures of test suite adequacy, and to know their limitations;