**SE 465 Final Review**

**Faults, Errors & Failures**

* **Fault (bug)** – static defect in software; e.g. incorrect lines of code
* **Error** – an incorrect internal state that is the manifestation of some fault; not necessarily observed yet
* **Failure** – external, incorrect behaviour with respect to the expected behaviour; must be visible
* RIP fault model – for a fault to become a failure:
  + Fault must be reachable
  + Program state after reaching fault must be incorrect (i.e. infection)
  + Infected state must propagate to output to cause a visible failure
* Dealing with faults:
  + **Fault avoidance** – not programming in a vulnerable language; better system design, e.g. by making an error state unreachable
  + **Fault detection** – testing; software verification; and repairing detected faults
  + **Fault tolerance** – redundancy (e.g. extra hardware); isolation (e.g. checking preconditions)

**Testing**

* To test a program, we can:
  + Execute every statement in the program (statement coverage)
  + Feed random inputs
  + Check different values for output conditions (logic coverage)
  + Analyze possible inputs & cover all interesting combinations of input (input space coverage)
* **Static testing** – a.k.a. “ahead of time”
  + E.g. static analysis – runs at compile time, automated
  + E.g. code review
* **Dynamic testing** – a.k.a. “at run-time”
  + Observe program behaviour by executing it
  + E.g. **black-box** (not looking at code) & **white-box** testing (looking at code)

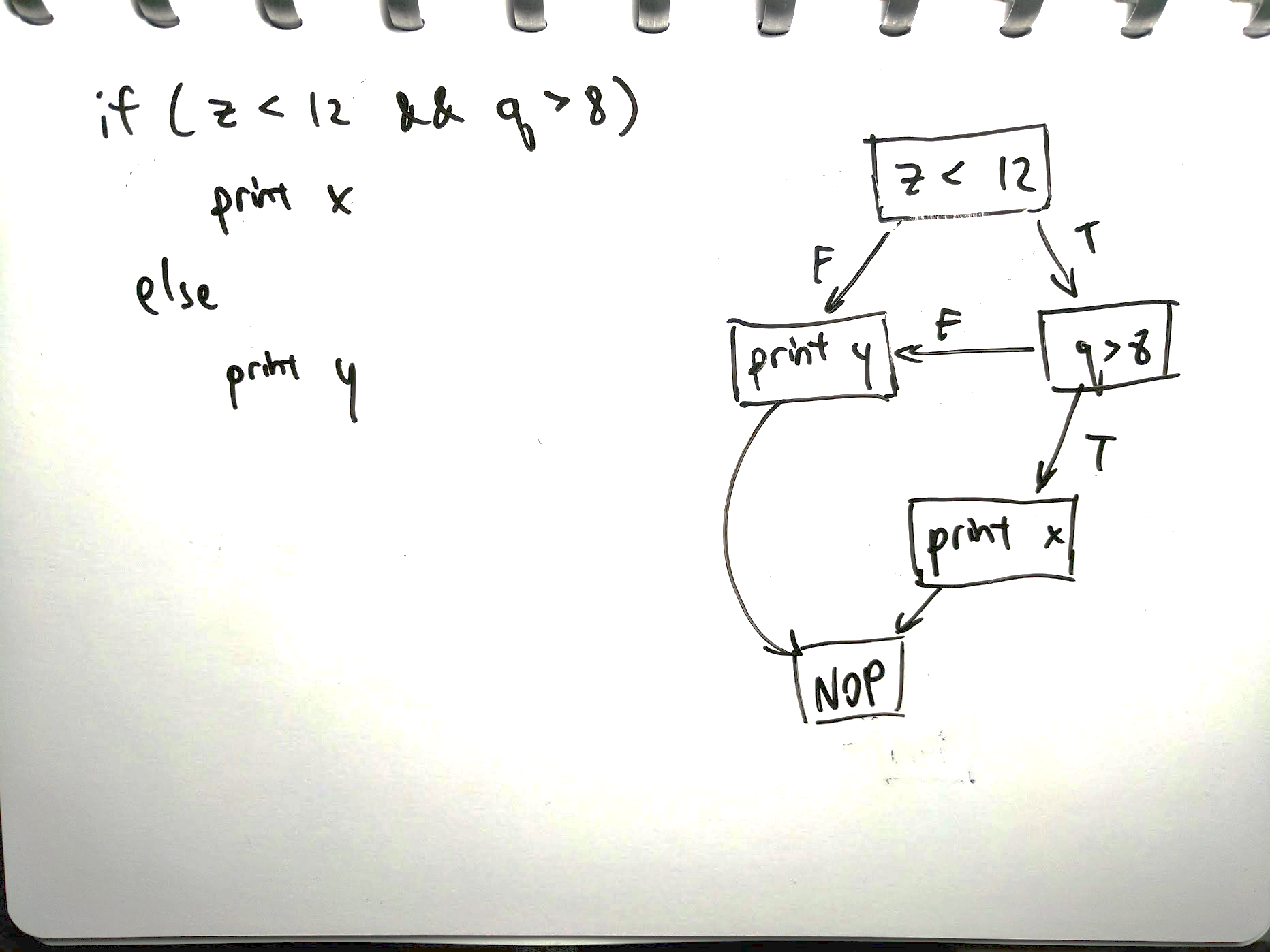
**Test Cases & Coverage**

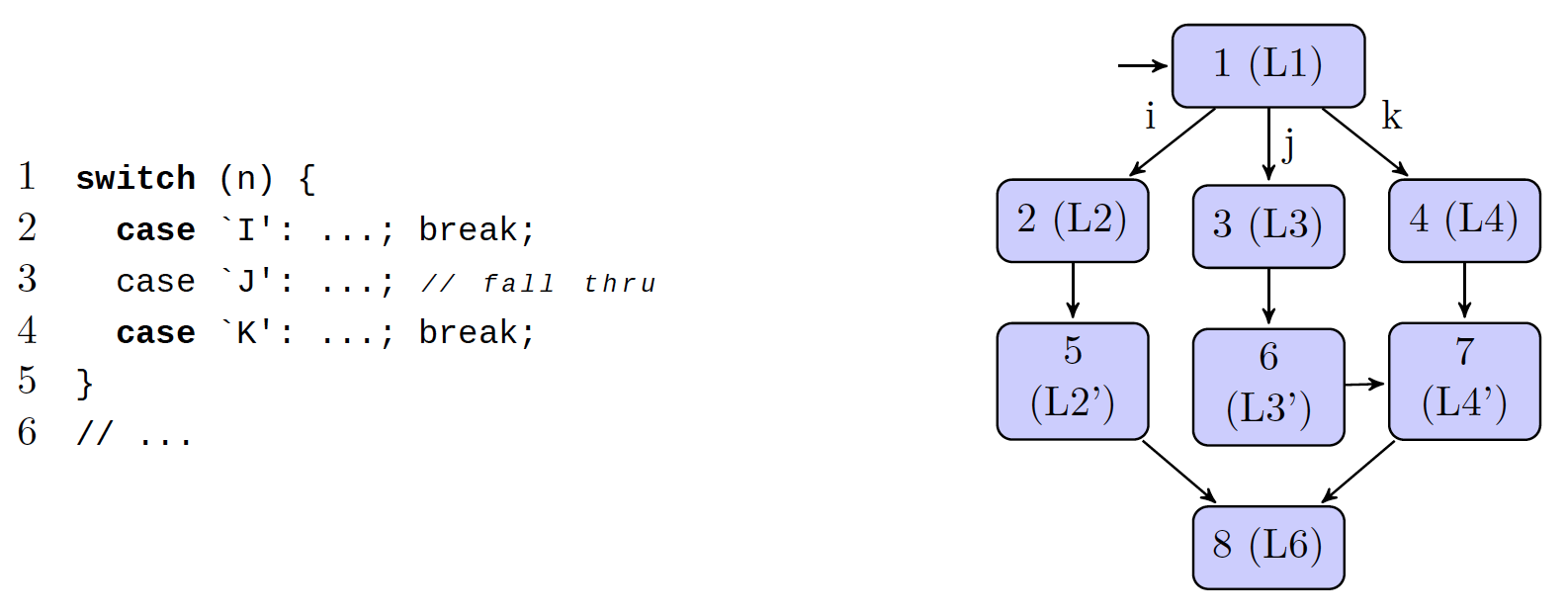
* A **test case** consists of:
  + Test case values: input values necessary to complete some execution of the software
  + Expected results: result to be produced if & only if program satisfies intended behaviour
  + Prefix values: inputs to prepare software for test case values
  + Postfix values: inputs for software after test case values
    - Verification values: inputs to show results of test case values
    - Exit commands: inputs to terminate program or to return it to initial state
* **Test requirement (TR)** – a specific element of a (software) artifact that a test case must satisfy or cover
  + E.g. to achieve branch coverage, each branch gives 2 TRs (branch is true; branch is false)
  + Infeasible test requirements – e.g. dead code
* **Coverage level** = # of TRs satisfied by a test set/total # of TRs

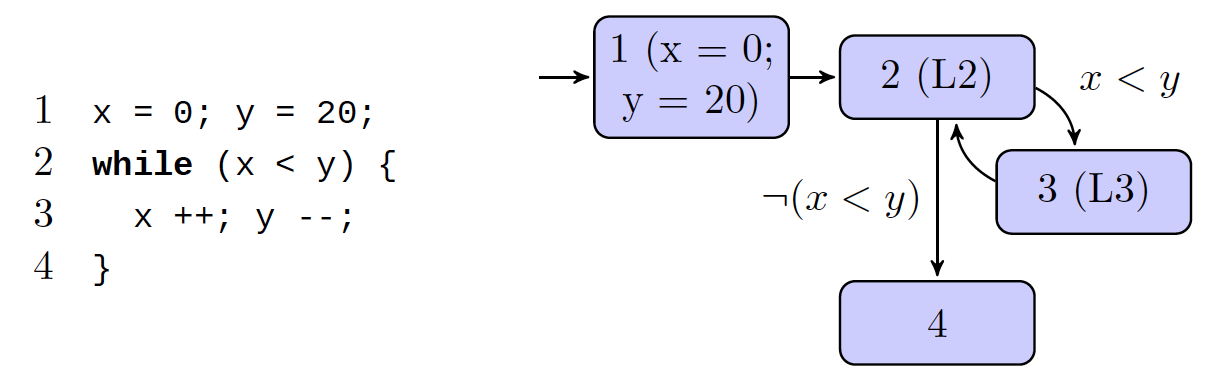
**Exploratory Testing**

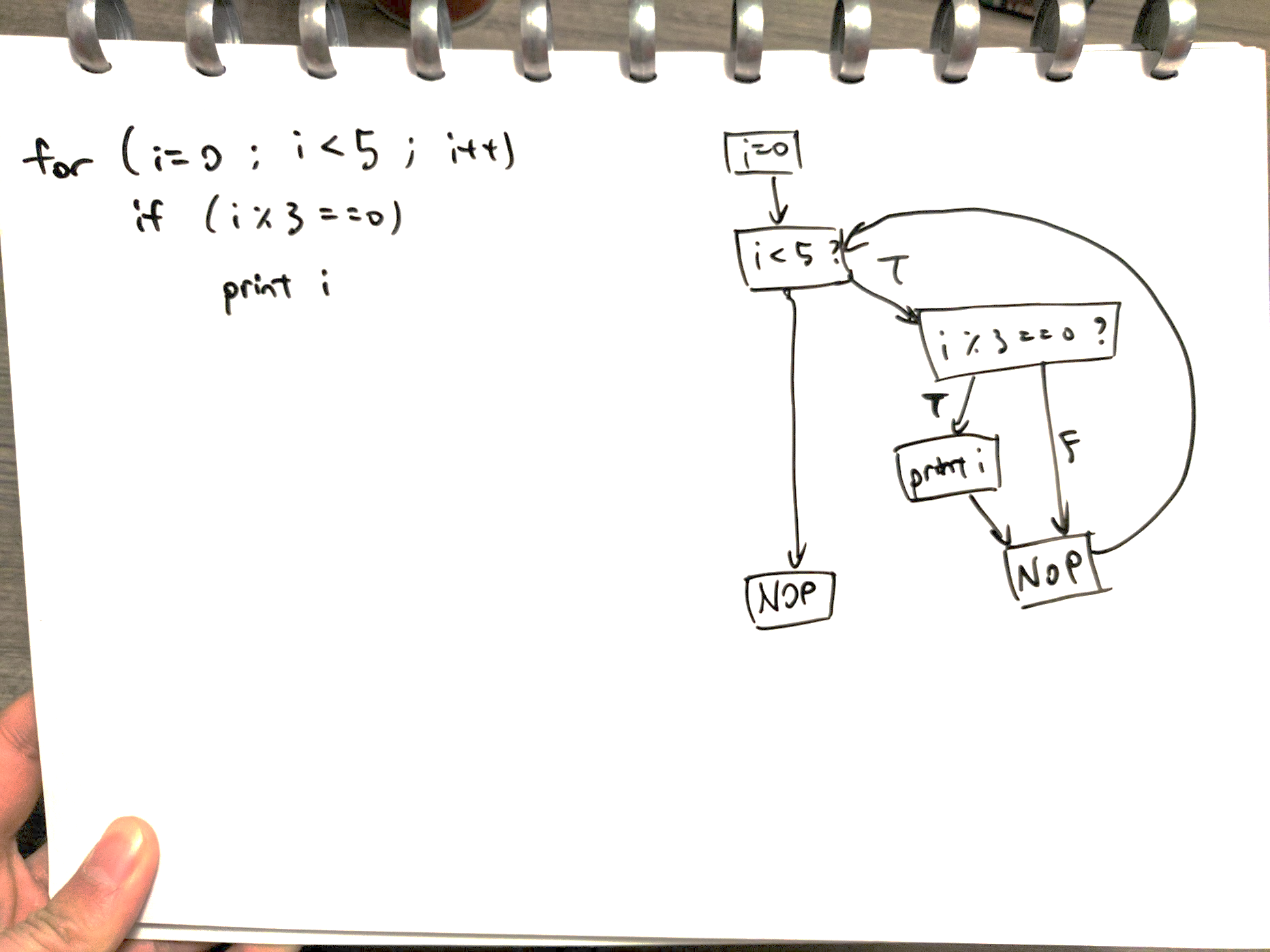
* **Exploratory testing** – “simultaneous learning, test design, and test execution”
  + In contrast: scripted testing – test design happens ahead of time, then execution occurs repeatedly
  + Scenarios where exploratory testing excels:
    - Providing rapid feedback on new product/feature;
    - Learning product quickly;
    - Diversifying testing beyond scripts;
    - Finding single most important bug in shortest time;
    - Independent investigation of another tester's work;
    - Investigating and isolating a particular defect;
    - Investigate status of a particular risk to evaluate need for scripted tests.

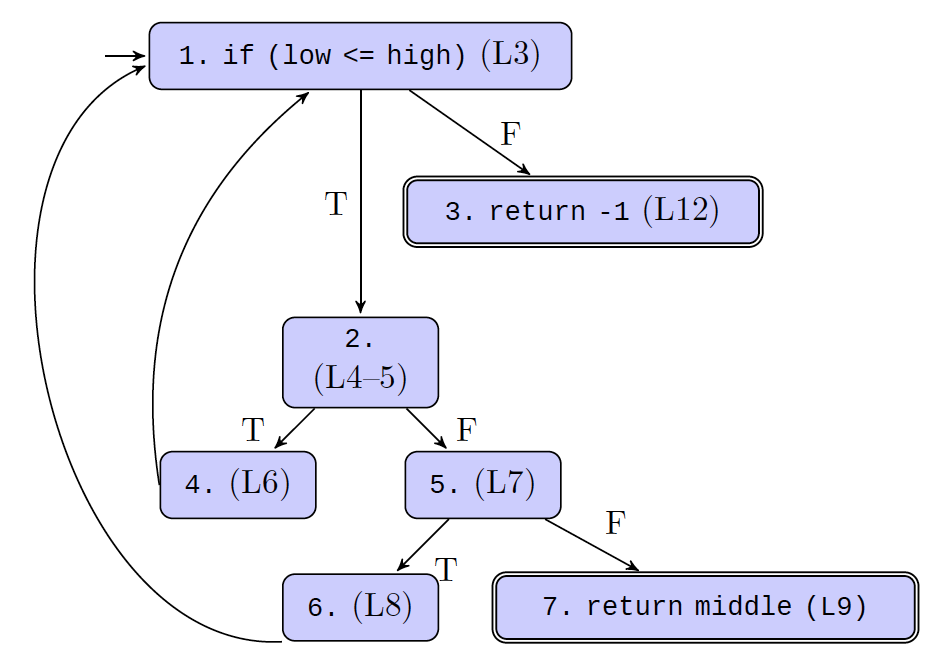
**Coverage Criteria & CFGs**

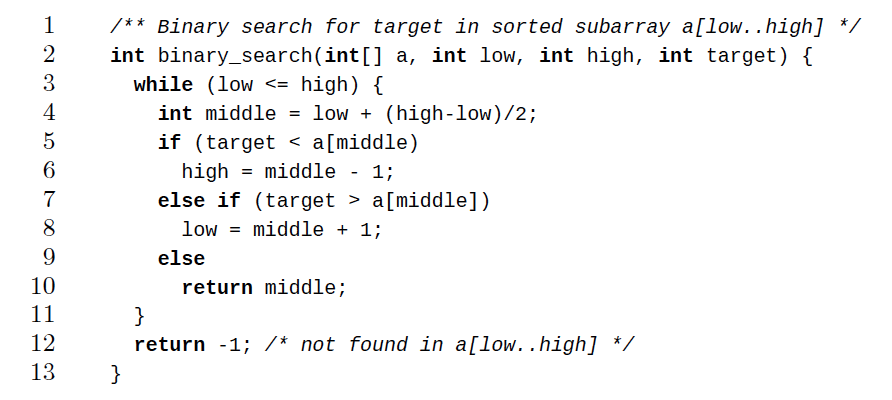
* **Control-flow graphs**
  + Node = zero or more statements (of code)
  + Edge = (s1, s2) → s1 may be followed by s2 in execution
* **Examples:
  + Short-circuit conditional:



* + Switch statements:
  + While loop:



* + For loop:
  + Complex example:

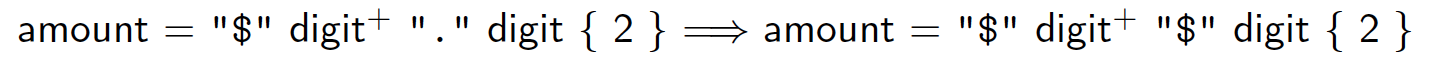
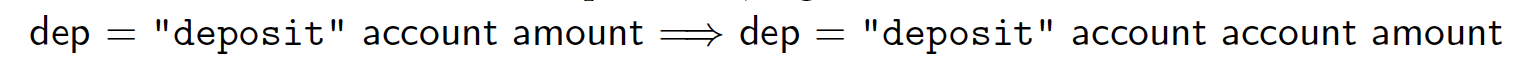


* **Test path** – a path (possibly with length 0) that starts at some initial node (N0) and ends at some final node (Nf)
  + *path(t)* is the set of test paths corresponding to test case *t*
  + *path(T)* = {*path(t)* | *t ∈ T*}
  + Each test case gives at least one test path
    - If the program is deterministic, each test case gives exactly one path; i.e. the test case determines the test path
* **Nondeterminism**
  + Caused by dependence on inputs, the thread scheduler, or memory addresses (e.g. Java’s hashCode())
  + More than one output might be a valid result of a single input
* **Statement coverage** – for each node n that can be reached by N0, TR contains a requirement to visit n
  + i.e. T satisfies statement coverage if & only if for every syntactically reachable node n, there is some path in path(T) that visits n
* **Branch coverage** – TR contains each reachable path of length ≤ 1 in G
  + i.e. tests in TR must traverse every possible edge
* In real programs, 80% coverage is usually sufficient
* Impossible to achieve with complete coverage cyclic graphs

**Finite State Machines**

* FSMs – capture the higher-level design of the software
  + Nodes = software states (e.g. sets of values for key variables)
  + Edges = transitions between software states (e.g. something changes in the environment; someone enters a command)
* **Node/state coverage** – visit every FSM state
* **Edge/transition coverage** – traverse every FSM transition
* **Edge-pair/two-trip coverage** – extend edge coverage of paths with length ≤ 2
* **Round-trip path** – path of nonzero length with no internal cycles that starts and ends at the same node
* **Simple round-trip coverage (SRTC)** – TR contains at least one round-trip path for each reachable node in G that begins and ends a round-trip path
  + i.e. if a node is part of a cycle, cover one of the cycles that the node is in
* **Complete round-trip coverage (CRTC)** - TR contains all round-trip paths for each reachable node in G
  + i.e. if a node is part of a cycle, cover all cycles that the node is in

**Syntax-Based Testing**

* Context-free grammars can be used to create inputs (both valid and invalid) or modifying programs (mutation testing)
* Using input grammars:
  + **Recognizer** – can include them in a program to validate inputs
  + **Generator** – can create program inputs for testing
    - Begin with the start production and replace non-terminals with their right-hand sides to get (eventually) strings belonging to the input languages
* Example mutation operators:
  + Non-terminal replacement
  + Terminal replacement
  + Terminal & non-terminal deletion
  + Terminal & non-terminal duplication
* **Fuzzing**
  + Generation-based fuzzing – starts with a grammar and generates inputs that match the grammar
    - Generate random strings from the grammar → feed as input and look for assertion failures
  + Mutation-based fuzzing – starts with existing test cases and randomly modifies them to explore new behaviours
    - E.g. randomly flip bytes, or parse input and change non-terminals
    - A finite given input set can only be fuzzed into a limited set of possibilities & code paths; will need to provide new input sets at that point

**Mutation Testing**

* **Ground string** – a (valid) string belonging to the language of the grammar
* **Mutation operator** – a rule that specifies syntactic variations of strings generated from a grammar
* **Mutant** – the result of one application of a mutation operator to a ground string
  + Are valid programs that ought to behave differently than the ground string
* Given a mutant M generated from ground string M0
  + Test case T (strongly) kills M if running T on M gives a different output than on M0
  + Mutation score = percentage of mutants killed
* Uninteresting mutants:
  + **Stillborn** – cannot compile/immediately crash
  + **Trivial** – killed by almost any test case
  + **Equivalent** – indistinguishable from original program
* **Strong mutation** – fault must be reachable, infect state, and propagate to output (i.e. a failure)
* **Weak mutation** – fault need only be reachable and infect state (i.e. an error)
* **Integration mutation** – mutate interfaces between methods
  + Change calling method by changing actual parameter values
  + Change calling method by changing callee
  + Change callee by changing inputs and outputs
* Example mutation operators
  + < ↔ ≤
  + ≤ ↔ >
  + if (a == b) → if (true)
  + + ↔ –
  + –– ↔ ++
  + true ↔ false
  + 42 ↔ 43
  + return x ↔ return x+1
  + Remove a void method call
  + int x = doSomething() ↔ int x = 1
  + Object y = doSomething() ↔ Object y = null
* Is mutation testing any good?
  + Yes: test suites that kill more mutants are also better at finding real bugs
* Is graph coverage any good?
  + Coverage does not correlate with high quality when it comes to test suites
  + Specifically: test suites that are larger are better because they are larger, not because they have higher coverage
  + Furthermore, stronger coverage (e.g. branch vs statement, logic vs branch) doesn't result in better test suites

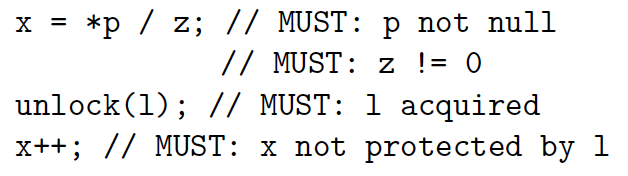
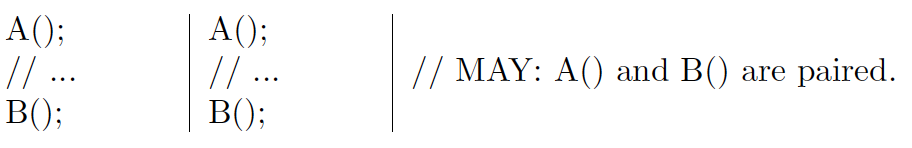
**Test Design Principles**

* Many small tests, not one big test
  + Easier to deal with failures
  + Easier to understand what’s being tested
* Make it easy to add new tests
* Unit vs. integration tests
  + Unit tests are more low-level and focus on one particular “class, module, or function"
    - They should execute quickly
  + Integration tests verify end-to-end functionality
* Write some code, write some tests, repeat
* Flaky tests are terrible
  + Timeouts can fail when something takes surprisingly long
  + Iterators can return items in random order
* Look inside the system under test
  + Avoid testing internal state, but rather only what is externally visible

**Selenium**

* Selenium automates web browsers by using a GUI-less simulator called *HtmlUnit*
* **Page object** – abstraction (interface) of the actions a user can take on a page & queries that can be made
  + Encapsulates UI elements on a page

**Coverity**

* Coverity is a static analyzer tool which finds bugs in large codebases
* Looks for contradictions & deviance in code
* **Must-beliefs** – inferred from code that has implications/requirements
  + Contradictions guarantee errors
  + E.g. redundancy: x = x, y | y, 1 \* z
  + E.g. check-then-use: if (!x) x…
  + E.g. use-then-check: x…; if (!x)
* **May-beliefs** – inferences which could be coincidental
  + Emit “check” on every belief confirm
  + Emit “error” on every deviation from belief
  + Ranking = check / error (more likely)
  + E.g. use-after-free:
    - foo(p);

p = 0 → emit check

* + - foo(p);

\*p = x → emit error

* + E.g. use-then-check:
    - p = bar();

if (!p) return;

\*p = x → emit check

* + - p = bar();

\*p = x; → emit error

**Clones**

* **Bellon’s Taxonomy:**
  + Type 1 – token streams are identical (may differ in whitespace & comments)
  + Type 2 – literals & identifiers may be different
  + Type 3 – may have extra/missing sections
  + Type 4 – semantically identical
* Clone detection methods
  + Sequence-based approaches
    - String-based
    - Token-based
  + Graph-based approaches
    - AST-based (abstract syntax tree)
    - PDG-based (program dependence graph)
  + Metrics-based approaches
    - Compare measurements instead of structures

**Test Engineering Principles**

* **Regression tests** should be:
  + Automated (because low yield of finding bugs)
  + Appropriated sized; fast (because should be run continuously)
  + Kept up-to-date; get rid of irrelevant tests over time
* **Test design**
  + Tests can verify state (e.g. calling accessor methods) or verify behaviour (e.g. using mocks)
  + Reduce test code duplication by using:
    - Expected objects
    - Custom asserts
    - Verification methods
  + Avoid logic in tests (e.g. ifs & loops) because tests aren’t testable
* Types of **test doubles**:
  + Dummy objects – placeholder objects that don’t do anything
  + Fake objects – have actual correct behaviour, unsuitable for use in production
  + Stubs – produces canned answers in response to interactions
  + Mocks – produces canned answers, also check that classic under test makes correct calls
    - Set up/record expectations (which methods are called with what arguments), which are verified as test is executed
  + Spies – wrapper around real object which monitors interactions
* **Flaky tests** – tests sometimes fail non-deterministically
  + Can label flaky tests and re-run them to see if they ever pass; or get rid of them
* Causes of flakiness:
  + Improper waits for async responses – don’t hard-code wait durations
  + Concurrency – proper use of concurrency primitives
  + Test order dependency, some tests expect others to be executed first – remove dependencies
* **Continuous integration** – use of a single shared master branch, where changes are merged in continuously
  + Advantages:
    - Software stays in a working state
    - Developers don’t take a long time to integrate changes
  + Requires continuous builds & automated testing
  + Broken builds need to be fixed immediately – do not accept commits that don’t pass tests
  + Building should be fast; tests should be tiered (fast → slow)
  + Test in prod-like environment (e.g. VMs)
* Anatomy of a **bug report**:
  + Summary/title – one-line recap
  + Description
  + Steps to reproduce – specifically describe each action
  + Expected results
  + Actual results
  + Build date & platform

**Static vs. Dynamic Analysis**

* **Static analysis** – have partial information about all executions and states
* PMD – a static code analyzer that flags syntax & style issues in code
* XPath – a query language used to navigate through an XML document

|  |  |
| --- | --- |
| **Syntax** | **Description** |
| / | Selects from immediate children |
| // | Selects from anywhere in the tree |
| . | Current node |
| .. | Parent node |
| […] | Predicate; “such that” |
| @ | Selects attribute |
| count() | Counts # of occurrences |

* + Example:

/Function

[.//Annotation

/Name[@Image='Test']]

[count(.//Statement

[//Prefix

/Name[starts-with(@Image, 'assert')]]

//ArgumentList/Expression//Prefix

/Name[@Image='Target'])=1]

* + Matches:

→ Function

→ Annotation

→ Name ‘Test’

→ Statement

→ Prefix

→ Name ‘assertFoobar’

→ ArgumentList

→ Expression

→ Prefix

→ Name ‘Target’

* Facebook Infer – open-source static code analyzer
  + *Eradicate* – only references annotated with @Nullable can be assigned null
    - Guarantees no null-pointer exceptions
  + Flags potential resource/memory leaks
  + Flags for exposure of “tainted” values (unsafe or secret data) to outside world or sensitive functions
* **Dynamic analysis** – have complete information about some program states based on observations
  + E.g. Valgrind detects memory errors dynamically
    - Checks by emulating a CPU
    - Illegal reads/writes, reads of uninitialized variables, illegal frees, memory leaks
  + E.g. AddressSanitizer
    - Checks by translating memory calls to its own versions, and using shadow memory
    - Out-of-bounds memory accesses, use-after-free, use-after-return, use-after-scope
  + E.g. Helgrind checks for race conditions
    - Keeps track of which program holds which locks
    - Monitors shared memory