FOURTH QUIZ

You have 15 minutes from the start of class to complete this quiz. Give partial answers if you can’t give complete ones. Read the questions with care; work with deliberate speed. Don’t give us more than we ask for. The usual instructions apply. Good luck!

Problem 1 (4 points)

Take a look at this Java method:

```java
void riskyCode() {
    System.out.println("Tortilla");
    try {
        System.out.println("Crepe");
        danger();
        System.out.println("Blini");
    } catch (Exception myEx) {
        System.out.println("Pancake");
    }
    finally {
        System.out.println("Flapjack");
    }
}
```

(a) What does the code above print out if the call to `danger()` raises an exception?

(b) What does the code above print out if the call to `danger()` does not raise any exceptions?

Problem 2 (4 points)

Three techniques proposed in the 1970s to solve the “software crisis” of expensive, unreliable programs were structured programming, formal verification, and n-version programming. Pick one of these and (a) give a brief one-sentence description of the technique and (b) say whether the technique was successful or not and, in one brief sentence, one main reason why it succeeded or failed.

2 points for (a), 2 points for (b); they don’t have to label the parts (a)/(b). Anything reasonably true and applicable gets credit.

Structured programming (single-entry, single-exit, gotoless code) succeeded; makes more modular code, manages complexity of large programs.

Formal verification uses mathematical logic to show that code matches (formal) specs. It didn’t really succeed in a widespread way (we don’t all do it, although some critical apps might) because it’s hard to write formal specs, the specs themselves can have errors, it’s an expensive process that requires trained people, programs are huge so nobody wants to read them or their proofs, there’s no community out there to verify the proof (unlike in math), which would require disclosure of source code (which software producer doesn’t want to do).

N-version programming says to take N teams writing independently in different languages, so you have N independent programs to “vote” when they don’t all reach the same answer. But the assumption of independence is wrong; people tend to make the same mistakes. Hardware fault-tolerance (redundant components) doesn’t protect against design errors, and that’s what we’re trying to do here. Also it’s expensive.
Problem 3 (17 points)

(a) (9 points) Draw a state transition diagram for an FSA that accepts standard telephone numbers. The number may have a parenthesized area code or not; if it does, it may optionally have a "1" before the area code. Area codes may not start with a zero; seven-digit phone numbers may not start with a zero or a one. You may omit drawing the error state. Below are some examples:

Valid: 824–5072      (949) 824–5072      1 (949) 824–5072      1 (800) 800–0000

(b) (8 points) Draw the state transition table for your FSA above. You may leave blank any unspecified transitions; you may omit the error state. We have supplied horizontal lines; you will supply the vertical lines and everything else. You might not need all the rows.