Slide 1

Administrivia

- Reminder: Homework 7 due Thursday.
- One more homework, to be on the Web by tomorrow and due during finals week (Wednesday?).

Slide 2

"Informal Formal Methods" Revisited

- The textbook discusses the very basics of one approach to proving program correctness. Can be helpful, but probably too much work to apply in detail to non-trivial programs.
- However, my claim is that the basic ideas can be applied in a less formal way, to good effect. Examples? try a couple of sorting algorithms.

Slide 3

Example — Specification for Sorting Algorithm

- Precondition: A is an array (of some type for which the $\leq$ operator makes sense).
- Postcondition: After the sort, the elements of A are a permutation of the original values, and for all neighbor pairs $(A_i, A_{i+1}$

$$A_i \leq A_{i+1}$$

Slide 4

Bubble Sort

- Scala code for bubble sort (of `Int`s):

```
def bubbleSort(a : Array[Int])
  for (stopIndex <- a.size to 2 by -1) {
    // make exchange pass over a(0 .. stopIndex-1)
    for (j <- 1 until stopIndex)
      if (a(j-1) > a(j)) {
        val temp = a(j-1) ; a(j-1) = a(j) ; a(j) = temp
      }
}
```

- Very informally, this works because the first pass through the inner loop puts the largest value at the end, and the next pass puts the next-to-largest value next to the end, and so forth. Can we make a semi-formal argument for that?

Slide 5

Bubble Sort, Continued

- What we want as a postcondition for the inner loop is that `a(stopIndex-1)` (the last element involved in this iteration of the outer loop) is greater than or equal to all the elements to its left.
- We can get that from a loop invariant involving the variable `j` we use to index through the array.
- We could take a similar approach to arguing for correctness of the outer loop, though there's maybe less need for that.
- Details in annotated code linked from "lecture topics and assignments" page here.
- (Strictly speaking we probably should also include as part of our invariants something about the values of `a` being a permutation of the original values!)

Slide 6

Example — Quicksort

- Scala code for quicksort (call as `quickSort(a, 0, a.size-1)`):

```
def quickSort(a : Array[Int], start : Int, end : Int) {
  if (start < end) {
    val splitIndex = partition(a, start, end)
    quickSort(a, start, splitIndex-1)
    quickSort(a, splitIndex+1, end)
  }
}
```

- Very informally, this works if `partition` "does the right thing" in rearranging `a` to put elements smaller than the pivot to the left, elements larger than the pivot to the right, and returning the position for the pivot. `partition` can be tricky to write, but maybe a loop invariant can help.

Slide 7

Quicksort, Continued

- The idea of `partition` is this:

  First we pull out the first element and call it the pivot; this leaves a "hole" in the array (index `splitIndex`), which will move as needed.

  We then scan through the rest of the array from the starting index through the ending index, maintaining the invariant that everything to the left of the split index is less than or equal to the pivot, and everything between the split index and the current scan point is greater.

- Details in annotated code linked from "lecture topics and assignments" page here. (If this seems complicated — yes, it is, and thinking it through can be non-trivial, but if you're careful you spend less time debugging.)

Slide 8

Minute Essay

- None — quiz.