Announcements:

- Prelim #1 tonight!
  - Conflict exam: 5:45-7:15 in 315 Upson
    - Only for people with conflicts
  - Main exam: 7:30-9:00 in Goldwin Smith Hall G64
  - Graded (late) tonight, back in section tomorrow
- PS3 due Thursday 11:59PM
  - Testing will start sometime Friday morning
  - Return is likely to be delayed due to Fall break
- Quiz #3 in class Tue Oct 18
Minimal correct induction proof

Example problem you might see on a prelim:

Recall that for any natural number \(n\), we define \(n!\) as \(n(n-1)(n-2)\ldots\), where \(0! = 1\). Write a recursive definition \(\text{fact}\ n\) that computes \(n!\), and prove your definition is correct using induction and the substitution model.

Solution:

let rec fact(n) = if n=0 then 1 else n*fact(n-1)

* Statement P\[n\]: the value of the OCaml expression \(\text{fact}(n)\) is \(n!\)
* Variable we are doing induction on: \(n\), starting at 0
* Base case: we prove P\[0\] as follows
  
  fact(0)
  b.s.m. (substitute) is
  if 0=0 then 1 else 0*fact(0-1)
  b.s.m. (primitives) is
  if true then 1 else 0*fact(0-1)
  b.s.m. (if) is
  1
  
  So the value of the expression \(\text{fact}(0)\) is 1 which is 0!
* Induction step:
  Pick an \(n \geq 0\) and assume P\[n\], then prove P\[n+1\]
  fact(n+1)
  b.s.m. (substitute) is
  if n+1=0 then 1 else n+1*fact(n+1-1)
  Since \(n \geq 0\) the value of the expression \(n+1=0\) is false
  b.s.m. (if) is
  n+1*fact(n+1-1)
  b.s.m. (primitives) is
  n+1*fact(n)
  By the induction hypothesis P\[n\] the value of \(\text{fact}(n)\) is \(n!\) so this is
  \(n+1*n!\)
  which is \(n+1!\)
• You’ve seen binary trees in CS2110
• Let’s look at a data structure called a “trie”
• A trie is a “finite map”, like a dictionary. It maps keys to values. Typically for a trie the keys are strings and the values are numbers.
• A trie is sometimes called a “prefix tree”. The basic idea is that a path through the tree represents a prefix, i.e. all strings that start with a particular substring.
  o Root is the empty string
• Example:

  ![Trie Diagram]

  • This trie is the finite map {"to"->7, "tea"->3, "ten"->12, "in"->5, "inn"->9}
    o As you saw in CS2110, tree-like data structures of this form are very efficient when they are balanced
    o Note that a trie doesn’t need to be binary, though this one is
    o In fact, 26 children or so (capitalization, punctuation)
• A trie is very efficient when there are lots of shared prefixes
  o Occurs in many situations (letters, genes, IP addresses)
• Lookup operation is obvious. Insert and delete are surprisingly similar.
  Everything takes time O(L), which is the length of the longest entry.
• This is a huge advantage of a trie. Most data structures have very asymmetric costs for lookup/insert/delete, so you need to pick the right one for your application carefully.
• Also note that if you don’t find what you are looking for you know something close to it. Useful for, e.g., spell checking.
• Important variant: radix tree (aka Patricia trie), where we ensure that every internal node has 2 or more children by merging nodes with 1 child
• Sub-variant: store at the end “black” or “white”. Then you can use this to encode strings that are present and also strings that are absent. Application is for IP routing tables.
• We will go over the trie signature in section.
• An important idea, both in the trie and point example, is what is called a REP IN Variant. This is a property of the representation that must be satisfied for the representation to be valid. For example, in our radix tree example, a node must have 2 or more children, and never 1 (could be 0 if it’s a leaf).
• You will typically want to implement this with a function repOK that returns its argument or raises an exception.
• Check this on all inputs and on output.
  o This sanity check seems wasteful, and you can turn it off in production code (for example by making repOK into the identity function).
  o But it will catch a ton of subtle bugs
• Example: lists without duplicates, or in sorted order
  o In a certain sense these are types, but they can’t be checked at compile time.
  o Another example: even numbers, or prime numbers, or even natural or whole numbers
• But let’s now return to the idea of designing a proper specification.
• Deceptively simple example: square root function, float->float
• Spec: beyond the types, what is true before we call sqrt (precondition)
  o What is true after (postcondition)
• What is the actual spec?
  o Positive input
  o Returns “closest” positive float whose square is x
    ▪ Sort of...
• What if the spec is violated?
  o Return something arbitrary? Rarely the right answer
  o Should raise an exception, in general
  o IEEE actually defines an “out of band” value, NaN