Final Examination
Computer Science 312, Cornell University 15 May 2003

Solutions

1. True/False [12 pts] (parts a–f; **4** points off for each wrong answer, 2 points off for each blank answer, minimum problem score 0.)

   (a) With copying garbage collection, half of the heap storage is unused while the program executes.

   True

   (b) SML infers that the term `(fn(x,y)=>x, fn(x,y)=>y)` has a type equivalent to `('a*'b->'a)*('c*'d->'d)`.

   True

   (c) The shortest-path algorithm does not work correctly in general if any edges of the graph have zero weight.

   False

   (d) A string searching algorithm that looks for a pattern string in a larger text string (for example, the Boyer-Moore algorithm) must examine each character of the text string to avoid missing an occurrence of the pattern string.

   False

   (e) The treap data structure simultaneously satisfies both the heap ordering invariant and the binary search tree invariant.

   True

   (f) In the environment model an expression of the form `(fn x =>` e`)` evaluates to an unboxed value.

   False

2. Zardoz [15 pts] (parts a–b)

   For each of the following expressions, give an *expression* to replace the box $\boxed{}$, so that the entire containing expression evaluates to 42.

   (a) [7 pts]

```
let val zardoz: ('a->'b->'a) * (string * int) = [    ]
    fun g(z: ('a->'b->'a) * ('b*'a)): 'a =
      let val (f, (m, n)) = z in f n m end
in
    g(zardoz) + String.size (#1 (#2 zardoz))
end
```

Answer: `(fn(x:'a) => fn(y:'b) => x, ("312", 39)) (fn(x:'a) => fn(y:'b) => x, ("", 42))` etc.

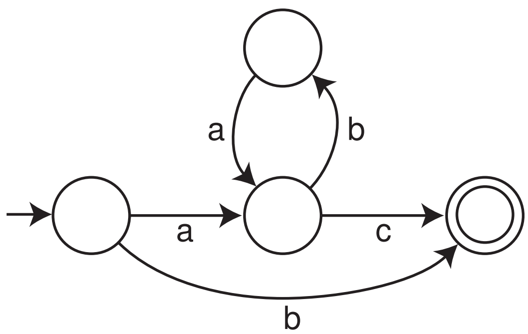
(b) [8 pts]

```
let
  fun zardoz (m: int -> int) (k: int) =
    if k > 13 then [   ]
    else m k - zardoz m (k + 1)
in
  zardoz (fn i => i + 3) 12
end
```

Answer: 43

3. Strings [8 pts] (parts a–b)

Consider the following finite automaton:



Recall that a finite automaton accepts any strings consisting of the symbols traversed along the edges from the start state to an accept state. The start state is indicated by an incoming arrow and an accept state is indicated by a double circle.

(a) [3 pts] Give three distinct strings that are accepted by this automaton.

Answer:

`b`, `ac`, `abac`

(b) [5 pts] Give a regular expression that matches exactly the strings that the automaton accepts.

Answer:

`b|a(ba)*c`

4. Graphs [16 pts] (parts a–b)

(a) [8 pts]

Suppose that we have a simple priority queue implementation in which the queue is represented as a sorted list. Extracting the minimum element takes $O(1)$ time, but inserting a new element or changing the priority of an existing element both take $O(n)$ time. What would be the asymptotic run time of Dijkstra's shortest-path algorithm if implemented using this priority queue? Explain briefly.

Answer:

The shortest-path algorithm does V insertions, V extract-mins, and $O(E)$ priority increases. The size of the queue is $O(V)$, so the insertions take $O(V^2)$ time. However, the $O(E)$ priority increases take $O(VE)$ time, which is asymptotically larger if the graph is connected. So the total time is $O(VE)$.

(b) [8 pts]

Dijkstra's shortest-path algorithm finds the length of the shortest path from a single vertex of the graph to every vertex to the graph. Suppose that instead you have a *set* of source vertices V_0 and want to know the shortest distance to every vertex in the graph, starting from any vertex $v \in V_0$. Explain briefly how to use or modify the shortest-path algorithm obtain this distance instead. For full credit your solution should be asymptotically as efficient as Dijkstra's algorithm.

Answer:

One way is to augment the graph with a new vertex that is connected to every one of the multiple start vertices with a zero-length path. Then run the standard shortest-path algorithm. This only adds one vertex and $|V_0|$ edges, so the asymptotic run time is unchanged.

An equivalent approach is to modify the algorithm so that it takes multiple start vertices. These vertices are all added to the set of visited vertices with distance zero, and pushed onto the priority queue. The algorithm then proceeds as usual.

5. Trees [14 pts] (parts a–c)

Consider the following datatype for a binary search tree of elements of type `elem` ordered by an ordering function `compare: elem*elem->order`. No element occurs twice in the tree.

```
datatype tree = Nil | Node of {left: tree, value: elem, right: tree}
```

(a) [3 pts] What representation invariant does this data structure satisfy?

Answer:

Every element in the left subtree is less than `elem` *(according to* `compare`*), every element in the right subtree is greater than* `elem`*, and the left and right subtrees themselves satisfy the representation invariant.*

Consider a binary search tree whose root node contains the element x. Suppose that the tree contains at least one element greater than x. Then one of these elements greater than x must be the smallest such element. Deletion from a binary search tree works by replacing the element x with this smallest greater element (if it exists).

Let's consider binary search trees in which the elements are integers. For example, the following is a tree in which the smallest element greater than the root element (2) is at a leaf node (3) that is a right child of its parent:

```
  2
 / \
1   3
```

(b) [3 pts] Draw three more trees (and label them correspondingly):

i. Draw a tree in which the smallest element greater than the root element is at a leaf node that is a *left* child of its parent.

ii. Draw a tree in which the smallest greater element is not at a leaf node.

iii. Draw a tree containing at least two elements, in which there is no smallest greater element.

```
  2      (i)
 / \
1   4
   / \
  3   5

  2      (ii)
 / \
1   3
     \
      4

    3    (iii)
   /
  2
 /
1
```

(c) [8 pts] Write the code *and* specification for a function `next: tree -> elem` that gives the smallest such element if it exists. You may assume that `compare` is in scope. Full credit will be given only to solutions that are $O(h)$ where h is the height of the tree.

```
(* next(t) is the smallest element greater than the element at the
 * root of t.
 * Checks: t is non-empty and contains an element greater than the
 * value at its root. *)
fun next(t: tree): elem = let
  fun leftmost(t: tree): tree =
      case t of
          Node {left = t' as Node{left=left', value=value', right=right'},
                value, right} => leftmost(t')
      | _ => t
  in
    case t of
      Node {left, value, right} =>
        (case leftmost(right) of
          Node {left, value, right} => value
        | Nil => raise Fail "no greater element")
    | Nil => raise Fail "no greater element"
  end
```

6. Correctness [35 pts] (parts a–j)

Consider the following data abstraction for whole (positive) integers and its implementation:

```
signature WHOLE = sig
  (* A whole is a whole number: for example, 1, 2, ... *)
  type whole
  (* one is 1 *)
  val one: whole
  (* succ(n) is n+1 *)
  val succ: whole -> whole
  (* plus(m,n) is m+n *)
  val plus: whole*whole -> whole
  val toInteger: whole -> int
end
structure Whole : WHOLE = struct
    (* AF: nil represents 1. false::n represents 2*AF(n).
     *     true::n represents 2*AF(n)+1.
     *)
    datatype whole = bool list
    val one = nil
    fun succ(n) = case n of
        false::n' => true::n'
    |   true::n' => false::succ(n')
    |   nil => [false]
    fun toInteger(n) = ...
    fun plus(m,n) = ...
end
```

(a) [1 pt] Explain in one sentence why no representation invariant is needed.

Answer:

Every possible representation represents a whole number.

(b) [1 pt] Does every whole number have a unique representation? Answer yes or no.

Answer:

Yes.

(c) [2 pts] Give legal representations of the numbers 2, 4, 5, and 19.

Answer:

`[false]`, `[true,false]`, `[true,true,false,false]`

(d) [3 pts] Give an implementation of `toInteger`.

```
fun toInteger(n) = case n of
    nil => 1
|   false::n => 2*toInteger(n)
|   true::n => 2*toInteger(n) + 1
```

(e) [5 pts] What is the asymptotic run time of `succ` as a function of n (interpreting n as a whole number, not a list!) You do not need to justify your answer. *Hint*: Consider $n = 2^k$.

Answer:

$O(\lg n)$

(f) [5 pts] Complete the implementation of `plus`. It should use `succ`.

```
fun plus(m,n) = case (m,n) of
    (nil,n) => succ(n)
|   (false::m', false::n') => false::(plus(m', n'))
|   (true::m', false::n') => true::(plus(m', n'))
|   (true::m', true::n') => false::(succ(plus(m', n')))
|   _ => plus(n,m)
```

Now, use induction to prove that the implementation of `succ` (*not* `plus`!) is correct.

(g) [2 pts] Start by stating the proposition to be proved, in terms of the abstraction function.

Answer:

To prove: given a representation n of a whole number ($n \geq 1$),

$$AF(n) + 1 = AF(\texttt{succ}(n))$$

Now, complete a proof by induction on the whole number $AF(n)$ being represented. Remember that the representations of whole numbers are not themselves numbers!

(h) [2 pts] Show that the base case holds.

Answer:

The base case is $n =$ `nil`, where the result is obviously `[false]`*, representing 2. So*

$$AF(n) + 1 = 1 + 1 = 2 = AF([\texttt{false}]) = AF(\texttt{succ}(n))$$

(i) [2 pts]

State the induction hypothesis (in terms of n) and the induction step to be proved. Is this strong induction or ordinary induction?

Answer:

The induction hypothesis is that

$$AF(n') + 1 = AF(\texttt{succ}(n'))$$

for all $1 \leq AF(n') \leq AF(n)$. The induction step is that assuming the induction hypothesis, for any representation m such that $AF(m) = AF(n) + 1$,

$$AF(m) + 1 = AF(\texttt{succ}(m))$$

(j) [12 pts] Show that the induction step follows, by considering the two possible structures of the representation. (Why are there only two possible structures?)

Answer:

Because m represents a number larger than 1, there are two cases: it can only be `false::`m' *or* `true::`m' *where m' represents some other number $AF(m')$ such that* $1 \leq AF(m') < AF(m)$.
Suppose $m =$ `false::(`m'`)`. *Then* `succ(`m`)` *evaluates to* `true::`m'. *So*

$$\begin{aligned} & AF(m) + 1 \\ = \quad & AF(\texttt{false} :: m') + 1 \\ = \quad & 2 * AF(m') + 1 \\ = \quad & AF(\texttt{true} :: m') \\ = \quad & AF(\texttt{succ}(m)) \end{aligned}$$

Now consider a representation `true::`m'. *The evaluation of* `succ` *gives us* `false::`m'' *where m'' is the result of evaluating* `succ(`m'`)`. *Because $AF(m') < AF(m)$, the induction hypothesis gives us*

$$AF(m') + 1 = AF(\texttt{succ}(m'))$$

That is,

$$AF(m') = AF(\texttt{succ}(m')) - 1$$

Therefore, we have

$$AF(m) + 1 = AF(\texttt{true} :: m')) + 1 = 2 * AF(m') + 1 + 1 = 2 * AF(m') + 2$$

Using the induction hypothesis, this is

$$\begin{aligned} & 2 * (AF(\texttt{succ}(m')) - 1) + 2 \\ = \quad & 2 * AF(\texttt{succ}(m')) \\ = \quad & AF(\texttt{false} :: (\texttt{succ}(m'))) \\ = \quad & AF(\texttt{succ}(m)) \end{aligned}$$