Solutions

1. True/False [20 pts] (parts a–j; **2** points off for each wrong or blank answer)

   (a) Mark-and-sweep garbage collection can collect cyclic garbage.

   True

   (b) Boxed values in the environment model correspond to heap-allocated storage.

   True

   (c) A binary heap supports efficient lookup of elements it contains.

   False

   (d) Functions are typically curried by sauteeing them with tumeric, cumin, and some vegetables.

   False

   (e) A variable is bound if it can take on only a finite set of values.

   False

   (f) Specification A refines specification B if and only if any valid implementation of A is also a valid implementation of B.

   True

   (g) It is important to document the representation invariant as part of a module's interface.

   False

   (h) The worst-case performance of the splay tree insertion operation is linear in the number of nodes in the splay tree.

   True

   (i) If a function is $\Theta(f(n))$, it is therefore also $O(f(n))$.

   True

   (j) Well-typed SML expressions never get stuck during evaluation.

   True

2. Short answers [24 pts] (parts a–e)

   (a) [7 pts] Suppose that we implement Dijkstra's shortest-path algorithm using a priority queue implemented by a linked list kept in sorted order. Assuming that there are at most some constant k outgoing edges per graph vertex, what is the asymptotic performance of the algorithm as a function of the number of vertices V? Explain briefly.

   The algorithm removes elements from the queue V times, but each removal takes only $O(1)$ time because the minimum element is the head of the list. There are also V insertions into the queue, taking $O(V)$ time each, and up to kV priority updates taking $O(V)$ time each. Therefore, the run time is $O(kV^2) = O(V^2)$.

   (b) [2 pts] What would be a better data structure to use in Part 2(a)?

   Answer:

   A binary heap, or any balanced binary search tree, such as a splay tree, a red-black tree, or an AVL tree.

(c) [4 pts] Give an example of an SML expression that is *not* well-typed, yet evaluates successfully in the substitution model to the value 0.

Answer:

Here is one of many ways to accomplish it: `if true then 0 else "hi"`

(d) [5 pts]

Suppose we want to implement a hash table data structure in SML to map keys of type `key` to values of type `value`. Give a possible type named `htable` for representing this data structure. You may also define additional types to use for defining `htable`. **Note:** The hash table does not need to be resizable.

Surprisingly, a lot of people were confused about what a hash table is and how it works. The common feature of all hash tables is an array into which keys are hashed. For this hash table, both keys and values must be stored in the array. There must also be some way to deal with collisions; the simplest is to have chained buckets, as in the following possible representation:

```
type htable = { nelems: int,
                buckets: (key*value) list array }
```

(e) [6 pts]

Continuing from Part 2(d), suppose that the hash table is based on a hash function `hash: key->int`. What is the representation invariant of the hash table data structure you have defined?

For each element of the buckets array, all of the keys in the stored list hash to that index of the buckets array. No key appears in a bucket more than one. And `nelems` is equal to the total number of key–value pairs stored in all of the buckets.

3. ML programming [20 pts] (parts a–c)

When implementing the operations of binary search trees, one finds that the same tree traversal code occurs in different operations. It's natural to try to factor out this common code. Consider the following immutable set interface. It's fairly standard except that the client provides a special value (`alt`) that is used to signal the absence of an element.

```
type set
type element
(* compare(e1,e2) is a total ordering on elements.
   If e1 and e2 are considered equal, they cannot both
   be members of the same set). *)
val compare: element * element -> order
(* lookup(s,elem,alt) is an element of s that is equal
   to elem or alt if there is no such element. *)
val lookup: set * element * element -> element
(* insert(s,elem,alt) is a pair (s',elem'). If s already contains
   an element e equal to elem (according to compare), s' is the same
   as s except that e is replaced by elem, and elem' is e. Otherwise,
   s' has just the elements of s, plus elem, and elem' is alt.
*)
val insert: set * element * element -> set * element
```

Now, suppose we write the following generic tree traversal code in order to implement both `lookup` and `insert`, assuming that `element` and `compare` are suitably defined:

```
datatype tree =
    Empty
  | Node of {left: tree, elem: element, right: tree }

type set = tree

fun traverse(t: tree, elem: element, f1, f2): tree * element =
case t of
  Empty => f1()
| Node {left, elem = e, right} =>
    case compare(elem, e) of
      LESS  => let val (t',elem') = traverse(left, elem, f1, f2) in
                (Node{left=t', elem=e, right=right}, elem')
               end
  | GREATER => let val (t',elem') = traverse(right, elem, f1, f2) in
                (Node{left=left, elem=e, right=t'}, elem')
               end
  |   EQUAL => let val (elem', elem'') = f2(e) in
                (Node{left=left, elem=elem'', right=right}, elem')
               end
```

(a) [3 pts] What are the types of `f1` and `f2`?

Answer:

```
f1:  unit -> tree * elementf2:  element -> element * element
```

(b) [12 pts] Show how to implement `lookup` and `insert` using the function `traverse`. The implementation of each should consist of a single application of `traverse`. **Hint:** Write types explicitly, and make sure that the types of the arguments to `traverse` match your answer to 3(a).

```
fun lookup(t, elem, alt) =
  traverse(t, elem,
    fn () => (Empty, alt),
    fn (elem') => (elem', elem'))
fun insert(t, elem) =
  traverse(t, elem,
    fn () => (Node{left=Empty,
                   elem=elem,
                   right=Empty},
              alt),
    fn (elem') => (elem',elem))
```

(c) [5 pts] Suppose we also want our set abstraction to support a element removal operation `remove`. Write a type *and* specification for such an operation. Design the operation and its specification to be stylistically consistent with the interface above. **Note:** Do not implement this operation.

```
(* remove(s, elem, alt) is a pair (s', e'), where e' is
   an element of s equal to elem (if there is one), and
   otherwise e' is alt.  In the former case, s' has the same
   elements as s except that e' is removed; in the latter,
   s' is identical to s. *)
val remove: set * element * element -> set * element
```

4. Recurrences and asymptotic complexity [20 pts] (parts a–b)

Consider the following function named `beauty` which operates on binary search trees:

```
datatype tree =
    Empty
  | Node of {left: tree, elem: int, right: tree }

fun beauty(t:tree, x: int): int =
  case t of
    Empty => x
  | Node {left: tree, elem: int, right: tree} =>
      if contains(t, x) then beauty(left, x-1)
      else beauty(right, x+1)
```

Let us assume that we only try to find the beauty of *perfectly balanced* binary search trees, and that `contains(t,x)` determines whether `x` is an element of `t` using a standard binary search tree look-up.

(a) [8 pts] Write a recurrence that captures the worst-case run time of the function `beauty` as a function of the number of elements in the tree.

Because the tree is balanced, the call to `lookup` takes $O(\lg n)$ time, so simplifying constants we arrive at the following recurrence:

$$\begin{aligned} T(0) &= 1 \\ T(n) &= \lg n + T(n/2) \end{aligned}$$

(b) [12 pts] Using the substitution method, show that the worst-case complexity of `beauty` is $O(\lg^2 n)$—i.e., $O((\lg n)^2)$.

Substituting $k \lg^2 n$ into the right-hand side for $T(n)$, we get

$$\begin{aligned} & \lg n + k \lg^2 (n/2) \\ = \; & \lg n + k(\lg n - 1)^2 \\ = \; & \lg n + k \lg^2 n - 2k \lg n + k \\ = \; & k \lg^2 n + (1 - 2k) \lg n + k \end{aligned}$$

As long as $k > 1/2$, the $\lg n$ term will be negative and will asymptotically dominate the $+k$ term. Therefore, $k \lg^2 n$ will be larger than the substituted RHS, and this shows that $T(n)$ is $O(\lg^2 n)$.

We also allowed solutions based on a proof by induction if they contained the essential elements of the argument above.

5. Evaluation models [25 pts] (parts a–d)

Consider the following code:

```
let
  fun counter(n) =
    let val x: int ref = ref n in
      fn () => (x := !x + 1; !x)
    end
  val c = counter(5)
in
  c() * c()
end
```

(a) [3 pts] What is the type of `counter`?

Answer:

```
int -> unit -> int
```

(b) [3 pts] What value is the result of this program?

Answer:

42

(c) [14 pts] Evaluate this program using the environment model and show the diagram that results, including both environment entries and heap-allocated storage.

Answer:

(d) [5 pts] Other than environment entries, what garbage is generated by this execution?

Answer:

The closure bound to `c`*, the closure bound to* `counter`*, and the* `ref` *bound to* `x`*.*