Computer Science 15–212–X: Midterm Examination
Sample Solutions

February 20, 1997

1 Induction and Recursion [25 Points]

1. (10 Points) Define an ML function `common` of type

   ```
   int list * int list * int list -> int option
   ```

   that, given three lists of integers, each sorted into increasing order, returns *either* `SOME` x, where x is the least element occurring in all three lists, *or* `NONE` in the case that the three lists have no element in common.

   Hint: proceed by simultaneous recursion the three lists. Fill in the following template:

   ```
   fun common(nil, _, _) = NONE
     | common(_, nil, _) = NONE
     | common(_, _, nil) = NONE
     | common(l as x::xs, m as y::ys, n as z::zs) =
         if x<y then
           common(xs, m, n)
         else if y<z then
           common(l, ys, n)
         else if z<x then
           common(l, m, zs)
         else
           SOME x
   ```

2. (15 Points) Give a rigorous proof of correctness of your implementation of `common` by simultaneous induction on the structure of the three lists.

   (a) Basis (one of the three lists is `nil`): Clearly the three lists have no common element if any of them is empty.

   (b) Induction step: State the inductive assumption, and prove the correctness of your implementation for the inductive case:

   We assume that common behaves correctly if any one of the three lists that we recur on is is smaller than the three input lists.

   If $x < y$, then note that x must also be smaller than all of the ys, so x cannot occur in ys. Therefore, any common element must occur among the xs, m, and n. Similarly if $y < z$ then y must be smaller than all of the zs, and if $z < x$ then z must be smaller than all of the xs. So if any of these three cases are true, we recur on (l', m', n') such that either $l' = xs$, *or* $m' = ys$, *or* $n' = zs$. By induction, the recursive calls find the least common element among the lists with either x, y, or z removed, as appropriate, and since we know by the above argument that the least common element of the lists (l, m, n) must occur within the lists (l', m', n'), we are done. If on the other hand $x >= y$ and $y >= z$ and $z >= x$, then $x = y = z$, and we have discovered the least common element.

2 Higher-Order Functions (35 Points)

A set of natural numbers A is *decidable* if there is an ML function f, called the *characteristic function* for A, such that

1. $f(x)$ evaluates to `true` iff $x \in A$, and
2. $f(x)$ evaluates to `false` iff $x \notin A$.

Thus the function f "decides" whether a given natural number is in the set by yielding either `true` or `false` as appropriate.

1. (25 Points) Give an implementation of the following signature of decidable sets of integers using the representation of sets as characteristic functions. The required signature is as follows:

```
signature INT_SET = sig
  type set

  val member : int * set -> bool
  val empty : set
  val singleton : int -> set
  val add : int * set -> set
  val remove : int * set -> set
  val intersect : set * set -> set
  val complement : set -> set
end
```

   The function `member` tests membership of an integer in a set. The value `empty` is the empty set. The function `singleton` creates the one-element set containing only the given integer. The function `add` adds an element to a set; the function `remove` removes an element from the set (returning the same set if it is not present). The function `intersect` computes the intersection of two sets. The function `complement` computes the complement of a set (the set of integers not in that set).

   Fill in the template given on the following page. You are *required* to use the given implementation type!

```
signature INT_SET = sig
  type set
  val member : int * set -> bool
  val empty : set
  val singleton : int -> set
  val add : int * set -> set
  val remove : int * set -> set
  val intersect : set * set -> set
  val complement : set -> set
end
structure IntSet :> INT_SET = struct
    type set = int -> bool
    fun member (x, s) = s(x)
    fun singleton x = fn y => x=y
    fun add x s = fn y => x=y orelse s(y)
    fun remove x s = fn y => not (x=y) andalso s(y)
    fun intersect (s, t) = fn y => s(y) andalso t(y)
    fun complement s = not o s
end
```

2. (10 Points) A set of integers A is said to be *semi-decidable* iff there is an ML function f, called a characteristic *partial* function for A, such that

   $f(x)$ evaluates to `true` iff $x \in A$.

   In other words, the second of the two requirements for a characteristic function is omitted. This means that if $x \notin A$, then $f(x)$ may either return false or may never return (loop forever). (We ignore the possibility that it may raise an uncaught exception.)

   Re-consider your answer to part (1) of this question for the case of semi-decidable sets. Which operation(s) are correct in the case of semi-decidable sets? Which operation(s) are incorrect? In the case of an incorrect operation, why is it incorrect?

   All functions work properly except `complement`, which fails to return true in the case that an element x is not in the set and the characteristic function fails to return on x.

3 Data Abstraction (40 Points)

A *cursor list* is a list equipped with a "cursor" designating the "current element" of the list. The following signature describes the operations supported by cursor lists:

```
signature CURSOR_LIST = sig
  type 'a clist

  (* the empty cursor list *)
  val empty : 'a clist

  (* reset the cursor to the first element *)
  val reset : 'a clist -> 'a clist

  (* insert an element after the cursor, making it the current element *)
  val insert : 'a clist * 'a -> 'a clist

  (* move the cursor forward or backward, wrapping around at the ends *)
  val forward : 'a clist -> 'a clist
  val backward : 'a clist -> 'a clist
end
```

The `current` function returns `SOME(`x`)`, where x is the current value under the cursor, or `NONE` in the case of the empty cursor list. Insertion of an element into a cursor list makes that element the current element. The operation `forward` moves the cursor one element to the right, wrapping around to the beginning in case the cursor is positioned at the last element. The operation `backward` moves the cursor left, wrapping around to the end in the case that the cursor is positioned at the first element. Performing a move on the empty cursor list has no effect: the empty cursor list is returned.

One implementation of cursor lists is based on representing a cursor list as a pair of ordinary lists `(`*`preds`*`,`*`succs`*`)` satisfying the following *representation invariant*:

1. The elements of the cursor list are `rev(`*`preds`*`)@`*`succs`*.
2. If *succs* is `nil`, then *preds* is `nil`.
3. The current element is the first element of *succs* (if any).

On the following page, give an implementation of the signature `CURSOR_LIST` based on this representation. *Be sure to preserve the representation invariant!*

You may wish to make use of the following familiar functions on lists:

```
(* revapp l m evaluates to rev(l) @ m *)
fun revapp nil l = l
  | revapp (h::t) l = revapp t (h::l)

(* list reversal *)
fun rev l = revapp l nil
```

```
signature CURSOR_LIST = sig
    type 'a clist
    val empty : 'a clist
    val reset : 'a clist -> 'a clist
    val forward : 'a clist -> 'a clist
    val backward : 'a clist -> 'a clist
    val insert : 'a * 'a clist -> 'a clist
    val delete : 'a clist -> 'a clist
    val current : 'a clist -> 'a option
end
fun revapp (nil, r) = r
  | revapp (h::t, r) = revapp (t, h::r)
structure CursorList :> CURSOR_LIST = struct
    (* Representation Invariant: if (preds, succs) : 'a clist, then
       1. The elements of the list are rev(preds)@succs
       2. If succs is nil, then preds is nil.
       3. The cursor points at the first element of succs.
     *)
    type 'a clist = 'a list * 'a list
    val empty = (nil, nil)
    fun reset (preds, succs) = ([], revapp (preds, succs))
    fun forward (_, nil) = (nil, nil)
      | forward (preds, [cur]) = (nil, revapp (preds, [cur]))
      | forward (preds, (cur::succs)) = (cur::preds, succs)
    fun backward (_, nil) = (nil, nil)
      | backward (nil, succs) =
        let
            val (pred::preds) = rev succs
        in
            (preds, [pred])
        end
      | backward (pred::preds, succs) = (preds, pred::succs)
    fun insert (new, (_, [])) = ([], [new])
      | insert (new, (preds, cur::succs)) = (cur::preds, new::succs)
    fun delete (_,[]) = ([], [])
      | delete (preds, [cur]) = ([], rev preds)
      | delete (preds, cur::succs) = (preds, succs)
    fun current (_, []) = NONE
      | current (preds, cur::succs) = SOME cur
end
```

Extra Credit Suppose we extend the signature `CURSOR_LIST` with the operation

```
val foldr : ('a*'b->'b) -> 'b -> 'a clist -> 'b
```

which "folds" a `clist` by a given binary operation and unit element. Give a *one-line* implementation of this operation in terms of the `foldr` and `foldl` operations on ordinary lists.

```
fun foldr f i (A,B) = List.foldl f (List.foldr f i B) A
```