# Computer Science 15–212–X: Midterm Examination

February 20, 1997

## Name

Name: ______________________________

Andrew ID: ______________________________

## Instructions

- This is a closed-book, closed-notes examination.
- There are 10 pages in this examination, including two worksheets.
- This examination consists of three questions worth a total of 100 points. The point value of each question is given with the question.
- Read each question completely before attempting to solve any part.
- Write your answers legibly *in the space provided* on the examination sheet. If you use the back of a sheet, indicate clearly that you have done so on the front.
- Write your name and Andrew id in the space provided at the top of this page, and write your name at the top of each page in the space provided.
- Two worksheets are attached to the end of this examination for your personal use. *Under no circumstances will anything written on the worksheets count for or against your grade on this exam.*

## Grading

| Question | 1 | 2 | 3 | Total |
|---|---|---|---|---|
| Score | | | | |
| Maximum | 25 | 35 | 40 | 100 |

# 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, _, _) =

  | common (_, nil, _) =

  | common (_, _, nil) =

  | common (l as x::xs, m as y::ys, n as z::zs) =
```

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`):

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

## 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!

```
structure IntSet :> INT_SET = struct
  type set = int->bool
```

```
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?

# 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
```

```
structure CursorList :> CURSOR_LIST = struct
```

```
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.

## Worksheet

## Worksheet