Computer Science 15–212: Midterm Examination (Sample Solutions)

February 17, 2011

Name

Name: ______________________________

Andrew ID: ______________________________

Recitation Section: ________

Instructions

- Write your name, Andrew id, and recitation section in the space provided at the top of this page, and write your name at the top of each page in the space provided.
- This is a closed-book, closed-notes, closed-computer examination.
- There are 18 pages in this sample solution set. Solutions for each question follow immediately after the question.
- The 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.
- Look at all questions and all parts before starting the exam. Different questions have different levels of difficulty. Early questions need not be easier than later questions.
- Write your answers legibly *in the space provided* on the examination sheets.

Grading

Question	1	2	3	Total
Score				
Maximum	35	35	30	100

Question 1: Recursion with Lists and Trees [35 Points]

1.1 Integer Statistics [20 Points]

Consider the following specification for the function `stats`:

```
(* val stats : int list -> int * int * int
   stats(nil) ==>  (0, 0, 0)
   stats([x1, ..., xn]) ==> (max, altsum, n)
       where max is the largest integer in the list [x1, ..., xn],
             altsum is the alternating sum
                   x1 - x2 + ··· + (−1)^(n+1) xn,
             and n is the total number of integers in the list.
   Invariants:   xi ≥ 0, for each xi in the list [x1, ..., xn].
   Effects: none
*)
```

For example: `stats([3,8,1,5]) ==> (8, ~9, 4)`. This is because: (i) 8 is the largest element in the list, (ii) the alternating sum is $3 - 8 + 1 - 5 =$ ~9, and (iii) there are 4 elements in the list.

Your task will be to implement `stats` in four different ways, as described next. Throughout, you may find the SML function `Int.max : int * int -> int` helpful.

(a) First, implement the function `stats` using standard recursion, by filling in the blanks below:

```
fun stats(nil)  = ____________________

  | stats(__________) =
    let

      val (mx, asum, n) = ______________________________
    in

      ( _________________ , _________________ , _________________ )
    end
```

Hint: Observe that $x_1 - x_2 + \cdots + (-1)^{n+1} x_n \quad = \quad x_1 - (x_2 - x_3 + \cdots + (-1)^n x_n)$.

[please go to the next page]

(b) Now, reimplement `stats` using a curried tail-recursive helper function
`st : int list -> (int * int * int) -> (int * int * int)` as follows:

```
fun stats l =
    let

      fun st [] acc = ____________________________

        | st __________  (mx, asum, n) =

            st _______  ( _____________________________________________________ ,

                         _____________________________________________________ ,

                         _____________________________________________________ )
    in

      st _________________________  _________________________
    end
```

Be sure to compute the alternating sum properly. In case you find it useful, you may assume the existence of the function `even : int -> bool`, such that `even(n)` ↪ `true` whenever `n` is an even integer and `even(n)` ↪ `false` whenever `n` is an odd integer.

(c) Reimplement `stats` using the higher-order function `foldr`.
In case you have forgotten, `foldr` has type `('a * 'b -> 'b) -> 'b -> 'a list -> 'b`.

```
val stats =
     foldr (fn (x, (mx, asum, n)) =>

                ____________________________________________________________________ )

            (0, 0, 0)
```

(d) Finally, reimplement `stats` using the higher-order function `foldl`.
In case you have forgotten, `foldl` has type `('a * 'b -> 'b) -> 'b -> 'a list -> 'b`.

```
val stats =
     foldl (fn (x, (mx, asum, n)) =>

                ____________________________________________________________________ )

            ____________________________
```

1.2 Operations on n-ary Trees [15 Points]

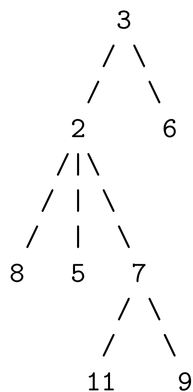
The following datatype models a tree with arbitrary branching factor at each node:

```
datatype 'a tree = Tree of 'a * 'a tree list
```

Thus a tree consists of an element and a list of subtrees. For example, the value

```
val tr : int tree =  Tree(3, [Tree(2, [Tree(8, nil),
                                        Tree(5, nil),
                                        Tree(7, [Tree(11, nil),
                                                 Tree( 9, nil)])]),
                               Tree(6, nil)])
```

represents the tree



(a) Implement a recursive function `appendall : 'a list list -> 'a list` that expects a list of lists and returns a single list formed by appending all the input lists together, in the order given. (The function should use straightforward recursion; don't worry about tail recursion.)

For instance, `appendall[[4, 5, 6], [1, 3], [], [2]]` $\hookrightarrow$ `[4, 5, 6, 1, 3, 2]`.

```
fun appendall(nil) = ____________

  | appendall(l::lists) = ____________________________________
```

(b) Now, using `appendall` implement a function `postorder : 'a tree -> 'a list` that expects a tree and returns a list consisting of the tree's elements as encountered in a *post-order traversal* of the tree. Recall that a post-order traversal of a tree visits the element stored at the current node *last*, after first recursively visiting all the elements in the subtrees. The order in which subtrees should be visited is given by their list order; each subtree is also visited recursively in post-order. For instance, `postorder(tr)` $\Longrightarrow$ `[8, 5, 11, 9, 7, 2, 6, 3]`.

```
fun postorder(Tree(x, tlist)) =

     appendall( ____________________________________ ) ________ ________
```

(c) Implement a function

```
treefold : ('a * 'a  -> 'a) -> ('b -> 'a) -> 'b tree -> 'a
```

that folds a function over a tree, *right-to-left* over the subtree list, as follows:

treefold f g (Tree(x, [])) $\Longrightarrow$ g(x)

treefold f g (Tree(x, [t_1, ..., t_n])) $\Longrightarrow$ f(v_1, f(v_2, ..., f(v_{n-1}, f(v_n, g(x)))$\cdots$))

where (treefold f g t_i) $\hookrightarrow$ v_i for each $i = 1, \ldots, n$.

```
fun treefold f g (Tree(x, tlist)) =

    foldr __________ __________ (map (_________________________) __________)
```

CAUTION: Note carefully the difference between the type of `treefold` and the type of `foldr`.

(d) Now reimplement `postorder` using `treefold`:

```
fun postorder t = treefold ___________________ ______________________ t
```

CAUTION: Be sure you get the syntax exactly right.

1. Solution to Question 1: Recursion with Lists and Trees

1.1 Integer Statistics

(a) Using straightforward recursion:

```
fun stats(nil)  = (0, 0, 0)
  | stats(x::l) =
     let
        val (mx, asum, n) = stats(l)
     in
        (Int.max(mx, x), x - asum , n+1)
     end
```

(b) Using a curried tail-recursive helper function:

```
fun stats l =
    let
       fun st [] acc = acc
         | st (x::l) (mx, asum, n) =
             st l (Int.max(mx, x),
                   asum + (if even(n) then x else ~x),
                   n+1 )
    in
       st l (0, 0, 0)
    end
```

(c) Using the higher-order function `foldr`:

```
val stats =
     foldr (fn (x, (mx, asum, n)) =>
                (Int.max(mx, x), x - asum, n+1) )
            (0, 0, 0)
```

(d) Using the higher-order function `foldl`:

```
val stats =
     foldl (fn (x, (mx, asum, n)) =>
                (Int.max(mx, x), asum + (if even(n) then x else ~x), n+1) )
            (0, 0, 0)
```

1.2 Operations on n-ary Trees

(a)
```
fun appendall(nil) = nil
  | appendall(l::lists) = l @ (appendall lists)
```

(b)
```
fun postorder(Tree(x, tlist)) = appendall(map postorder tlist) @ [x]
```

(c)
```
fun treefold f g (Tree(x, tlist)) = foldr f (g x) (map (treefold f g) tlist)
```

(d)
```
fun postorder t = treefold (op @) (fn a => [a]) t
```

Question 2: Hands of Cards [35 Points]

In class we defined a hand of cards recursively, in such a way that the order of cards in a hand was directly observable. In this question we will redefine a hand of cards using an abstract type, thereby dropping the ordering. This question will ask you to explore two different implementations of the abstract specification of a hand.

First, recall from class the definition of `suit`, `rank`, and `card`:

```
datatype suit = Clubs | Spades | Hearts | Diamonds

datatype rank = Two | Three | Four | Five | Six | Seven | Eight
              | Nine | Ten | Jack | Queen | King | Ace

type card = rank * suit
```

Suppose these types have already been declared in the environment.

Now let us abstractly define a *hand of cards* to be an unordered collection of `card`s. Duplicates are permitted. For example, the collection

$$\{\texttt{(Ace, Hearts)}, \texttt{(Queen, Diamonds)}, \texttt{(Five, Spades)}\}$$

is a hand of cards. It consists of three distinct cards. It is the same hand of cards as

$$\{\texttt{(Queen, Diamonds)}, \texttt{(Five, Spades)}, \texttt{(Ace, Hearts)}\}.$$

Observe that the collection

$$\{\texttt{(Four, Clubs)}, \texttt{(Three, Diamonds)}, \texttt{(Four, Clubs)}\}$$

is a hand of cards consisting of three cards in which the card `(Four, Clubs)` appears twice.

[please turn to the next page]

Here is the signature for the data abstraction of a hand of cards:

```
signature HAND =
sig
  type hand           (* abstract type *)

  (* constructors *)
  val empty     : hand
  val add       : card * hand -> hand

  exception NotInHand

  val remove    : hand * card -> hand
  val removeall : hand * card -> hand
  val combine   : hand * hand -> hand

  (* card observers *)
  val inhand    : hand * card -> bool
  val count     : hand * card -> int
end
```

The detailed meanings of the individual components of the signature are:

- `hand` is an abstract type used to represent a hand of cards.
- `empty` is the hand of cards consisting of no cards.
- `add(c, h)` returns a new hand formed by adding the card `c` to the hand `h`.
- `remove(h, c)` returns a hand consisting of the cards in hand `h` except that *one* occurrence of card `c` has been removed. If card `c` does not appear in hand `h` then the function should raise exception `NotInHand`.
- `removeall(h, c)` returns a hand consisting of the cards from hand `h` after *all* occurrences of card `c` have been removed. This function should never raise an exception, even if card `c` does not appear in hand `h`.
- `combine(h1, h2)` returns a hand consisting of all the cards in the two hands `h1` and `h2` (including all duplicates).
- `inhand(h, c)` returns `true` if the card `c` appears in the hand `h` and `false` otherwise.
- `count(h, c)` returns the number of times the card `c` appears in the hand `h`.

2.1 Implementation using the datatype `hand` [15 Points]

In the first implementation of `HAND` you will use the `hand` datatype originally given in class. The abstraction function for this implementation is straightforward, namely:
A value `Hand(c`$_1$`, Hand(c`$_2$`, ..., Hand(c`$_n$`, Empty)`$\cdots$`))` represents the hand of cards $\{c_1, c_2, \ldots, c_n\}$. There are no representation invariants.

The following structure `Hand` ascribes opaquely to signature `HAND`. Please fill in the missing lines of code.

```
structure Hand :> HAND =
struct
  datatype hand = Empty | Hand of card * hand

  val empty = ____________________

  val add = ____________________

  exception NotInHand

  fun remove(Empty, c) = ________________________
    | remove(Hand(c',h), c) =
         if c'=c then ________________________
         else ________________________________

  fun removeall(Empty, c) = ______________________
   |  removeall(Hand(c',h), c) =
         if c'=c then ________________________
         else ________________________________

  fun combine(Empty, h2) = ______________________________
    | combine(Hand(c, h), h2) = ______________________________

  fun inhand(Empty, c) = ______________________________
    | inhand(Hand(c',h), c) = ________________________________

  fun count(Empty, c) = ______________________________
    | count(Hand(c',h), c) =
        ___________________________________________________
end
```

2.2 Functional Implementation [15 Points]

In the second implementation of `HAND` you will use a function of type `card -> int`, counting the number of times each card appears in a hand. The abstraction function for this implementation is: A value `h : card -> int` represents the hand consisting of all cards `c` for which `h(c)` > 0. Moreover, `h(c)` specifies the number of occurrences of card `c` in the hand. The representation invariants are: (i) `h(c)` returns a value for all cards `c`, and (ii) `h(c)` ≥ 0 for all `c`.

The following structure `FunHand` ascribes opaquely to signature `HAND`. Please fill in the missing lines of code.

```
structure FunHand :> HAND =

struct

  type hand = card -> int

  val empty = ____________________

  fun add(c, h) =

        fn c' => if __________ then _________________ else _______________

  exception NotInHand

  fun remove(h, c) =

        case h(c)

          of 0 =>  ______________________________

           | k =>  fn c' => ________________________________

  fun removeall(h, c) =

        ________________________________________________

  fun combine(h1, h2) = ____________________________________

  fun inhand(h, c) = ______________________________

  fun count(h, c) = ______________________________

end
```

2.3. Scoping, Typing, and Evaluating [5 Points]

Give the type and value of each of the following expressions. If an expression is not well-typed, please briefly explain why. If an expression does not have a value, please briefly explain why.

You should assume a correct implementation of the structure `Hand :> HAND`.

1. `Hand.add((Ace, Hearts), Empty)`

2. `fn (c:card) => Hand.remove(Hand.empty, c)`

Solution to Question 2: Hands of Cards

2.1 Implementation using the datatype hand

```
structure Hand :> HAND =
struct
  datatype hand = Empty | Hand of card * hand

  val empty = Empty
  val add = Hand

  exception NotInHand

  fun remove(Empty, c) = raise NotInHand
    | remove(Hand(c',h), c) =
         if c'=c then h
         else Hand(c', remove(h,c))

  fun removeall(Empty, c) = Empty
   |  removeall(Hand(c',h), c) =
         if c'=c then removeall(h, c)
         else Hand(c', removeall(h,c))

  fun combine(Empty, h2) = h2
    | combine(Hand(c, h), h2) = Hand(c, combine(h, h2))

  fun inhand(Empty, c) = false
    | inhand(Hand(c',h), c) =   c'=c orelse inhand(h,c)

  fun count(Empty, c) = 0
    | count(Hand(c',h), c) =
        if c=c' then 1 + count(h,c) else count(h,c)
end
```

2.2 Functional Implementation

```
structure FunHand :> HAND =
struct
  type hand = card -> int

  val empty = fn _ => 0

  fun add(c, h) =
        fn c' => if c'=c then 1 + h(c') else h(c')

  exception NotInHand

  fun remove(h, c) =
        case h(c)
          of 0 => raise NotInHand
           | k => fn c' => if c'=c then k-1 else h(c')

  fun removeall(h, c) =
        fn c' => if c'=c then 0 else h(c')

  fun combine(h1, h2) = fn c => h1(c) + h2(c)

  fun inhand(h, c) = h(c) > 0

  fun count(h, c) = h(c)
end
```

2.3. Scoping, Typing, Evaluating

1. `Hand.add((Ace, Hearts), Empty)`

   This expression is not well-typed (and therefore does not have a value). Since `Hand` ascribes opaquely to `HAND`, the representation of the empty hand as `Empty` is hidden. The correct expression for adding the Ace of Hearts to the empty hand would be:
   `Hand.add((Ace, Hearts), Hand.empty)`.

2. `fn (c:card) => Hand.remove(Hand.empty, c)`

   This expression has type `card -> Hand.hand`. The value is simply a lambda expression, namely the lambda expression written.

Question 3: Higher-Order Functions and Induction [30 Points]

3.1 Self-Composition [15 Points]

The following code defines a higher-order function `cself : int -> ('a -> 'a) -> 'a -> 'a`. Given a non-negative integer `n` and a function `f`, the function `cself` computes the `n`-fold self-composition f^n of `f`. [By convention the 0-fold self-composition of a function of type `'a -> 'a` is the identity function of type `'a -> 'a`.]

```
(* val cself : int -> ('a -> 'a) -> 'a -> 'a
       cself n f ==> f^n

           where f^n is the function given by the rule

                 f^n(x) = f (f (··· f (x) ···)), for n > 0
                         [f is applied n times]
             and f^0(x) = x.

       Invariants: n ≥ 0
       Effects: none
*)

fun cself 0 f = (fn x => x)
  | cself n f = (fn x => f (cself (n-1) f x))
```

Definition: A function `f: t -> t` over some type domain `t` is said to be *total* if `f(v)` reduces to a value for all argument values `v:t`. (In other words, a function is total if it is defined for all elements in the type domain in such a way that it actually returns a value, as opposed to raising an exception or looping forever.)

The theorem on the next page asserts that the code for `cself` returns the value stated in the specification whenever the input function is total. Prove the theorem using mathematical induction, by filling in the proof template given.

As always, be sure to clearly state the base case, what you need to show in the base case, the induction hypothesis, what you need to show in the induction step, and where you use the induction hypothesis.

CAUTION: Be sure to evaluate and reduce the code correctly! In particular, write out intermediate steps that show when and how the lambda expressions appearing in the body of `cself` are used.

And remember that function application associates to the left. For instance, `(cself n f v)` is equivalent to `(cself n f)(v)`. Similarly, `(cself n f)` means the same thing as `(cself n)(f)`.

[please go to the next page]

Theorem: `(cself n f v)` $\hookrightarrow$ $\texttt{f}^{\texttt{n}}\texttt{(v)}$ for all integer values `n` ≥ 0
and all values `f:t->t` and `v:t` with `f` a total function over type domain `t`.

Proof by mathematical induction on __________.

Base Case: __________.

Need to show:

Show it:

Induction Step: __________.

Induction Hypothesis:

Need to show:

Show it:

3.2 Higher-Order Self-Composition [15 Points]

(a) Suppose the function `cself` is given as in Question 3.1. Please fill in the explicit type specifications in the code below, such that the return type of `compose` is polymorphic and as general as makes sense.

```
fun compose (n:int) (k:int) : ______________________________ =
    let

       val cn : _____________________________________ = cself n

       val ck : _____________________________________ = cself k
    in
       ck cn
    end
```

(b) What is the value of `(compose 0 7 (fn x => x + 1))(100)`?

(c) What is the value of `(compose 5 3 (fn x => x + 1))(100)`?

(d) Suppose `f:t->t` is a total function over some type domain `t`. Suppose `n` and `k` are nonnegative integers. Algebraically, describe the function returned by `(compose n k f)`.

[Your answer should be a very short algebraic expression (not ML code), using the symbols `n`, `k`, and `f` (parentheses are fine too).]

Solution to Question 3: Higher-Order Functions and Induction

3.1 Self-Composition

Theorem: `(cself n f v)` $\hookrightarrow$ `f`^{n}`(v)` for all integer values $n \geq 0$
and all values `f:t->t` and `v:t` with `f` a total function over type domain `t`.

```
Proof by mathematical induction on n.

Base Case:        n=0.

 Need to show:   (cself 0 f v) ↪ v for all values f:t->t and v:t
                                   with f a total function.

 Showing it:      cself 0 f v
              ==> (fn x => x) v
              ==> [v/x] x
              ==> v


Induction Step:     n ≥ 0.

 Induction Hypothesis:  Assume that for some n ≥ 0,
   and all values f:t->t and v:t with f a total function,
   (cself n f v) ↪ f^n(v).

 Need to show:  (cself (n+1) f v) ↪ f^(n+1)(v).

 Showing it:               (cself (n+1) f v)

         ==> [n+1/n']      (fn x => f (cself (n'-1) f x)) v    [since n+1 > 0]
         ==> [n+1/n'][v/x] f (cself (n'-1) f x)
         ==>               f (cself n f v)
         ==>               f (f^n(v))          [by IH]
         ==>               f^(n+1)(v)          [since f is total]


   QED
```

3.2 Higher-Order Self-Composition

(a)

```
fun compose (n:int) (k:int) : ('a -> 'a) -> ('a -> 'a) =
    let
       val cn : ('a -> 'a) -> ('a -> 'a) = cself n
       val ck : (('a -> 'a) -> ('a -> 'a)) -> ('a -> 'a) -> ('a -> 'a) = cself k
    in
       ck cn
    end
```

Reasoning: `ck` is called with argument `cn`, which is a function of type

`('a -> 'a) -> ('a -> 'a)`

(this is the most general type possible). Thus the argument and return types of `ck` must *each* be `('a -> 'a) -> ('a -> 'a)`. The return type of `compose` is the type of the expression `(ck cn)`, which is `('a -> 'a) -> ('a -> 'a)`.

(b) `(compose 0 7 (fn x => x + 1))(100)` $\hookrightarrow$ `100`

(c) `(compose 5 3 (fn x => x + 1))(100)` $\hookrightarrow$ `225`

(d) `(compose n k f)` $\cong$ $\mathtt{f}^{(\mathtt{n}^{\mathtt{k}})}$