Computer Science 15-212
Midterm Examination
February 27, 1996

Instructions

* This is a closed-book, closed-notes exam.
* This exam consists of 5 sheets of questions and 2 worksheets.
* There are 4 questions, worth 25 points each.
* Read each question completely before attempting to solve any part.
* Write your name, Andrew ID and section letter in the space below, and write your name in the space provided at the top of every page.
* Two worksheets are attached for your personal use. Nothing you write there will count for or against your grade.
* Write your answers legibly in the spaces provided. If you use the back of a sheet, indicate clearly that you have done so.
* In writing functions, you should use clausal form and pattern matching where possible instead of explicit constructors such as *hd* and *tl*.

Name _____

Andrew ID _____

Section _____

For instructor use only

1	2	3	4	*total*

name:

1 Scope and Evaluation

What do the following expressions evaluate to?

(a) [2pts] (let val x = 5 in x end) + (let val x = 3 in x end)

evaluates to

8

(b) [2pts] let val x = 5 in (x + let val x = 6 in 2 * x end) end

evaluates to

17

(c) [2pts]
```
let
   val x = 5
   fun f(x) = 5 - x
   val x = 6
in
    f(x)
end
```

evaluates to

-1

(d) [2pts] (fn (x) => let val x = x + 1 in 2 * x end) 3

evaluates to

8

(e) [2pts]
```
let val x = 3 in
      let val x = x + 1 and f = fn(y) => x + y
      in f(x)
      end
end
```

evaluates to

7

If you type an expression into ML , it may evaluate normally; it may raise an exception; or it may fail to evaluate at all because it is ill-typed, or it has an unbound variable. (Of course, it may also be syntactically invalid, but let's ignore that).

For each of the following expressions, indicate all the "bad" outcomes that happen sometimes or always (depending on what was typed previously), by encircling options from the given list, or choose "never a bad outcome".

(f) [3pts] let val x = "abc" in let val x = 3 in x + x end end

unbound variable, ill-typed, non-termination, exception raised *never a bad outcome*

(g) [3pts] let val x: int = 3 and y: int = x in y end

unbound variable, ill-typed, non-termination, exception raised *never a bad outcome*

(h) [3pts] local val x = 5 in fun f (z: real) = z + x end

unbound variable, ill-typed, non-termination, exception raised *never a bad outcome*

(i) [3pts] fn (x) => x div 0

unbound variable, ill-typed, non-termination, exception raised *never a bad outcome*

(because the expression evaluates to a function; note there's no application of the function)

(j) [3pts] (fn (x :: xs) => x) y

unbound variable, ill-typed, non-termination, exception raised *never a bad outcome*

name:

2 Recursion and Induction

Consider a function

mostly_true: 'a list * ('a -> bool) -> bool

that, given a list and a predicate, returns true iff the predicate is true on the majority of the elements of the list. For example,

mostly_true (2, 4, 5], fn(i) => i > 3)

evaluates to true, since 2 out of 3 of the elements are greater than 3.

(a) [3 pts] Suppose we tried to implement the function recursively by writing

```
fun mostly_true (nil, f) = false
  | mostly_true (h :: t, f) = G(h, f, mostly_true(t, f))
```

where *G* is some function whose code is not shown. Why must this approach fail?

because the induction hypothesis is too weak. if mostly_true(t) is true, and f(h) is false, there's no way of knowing whether mostly_true (h::t) is true or false.

(b) [8 pts] Write the code for *mostly_true* by filling in the gaps below:

```
fun mostly_true (l, f) =

  let  (* excess (q: 'a list): int  = true_ct - false_ct
            where true_ct is the number of elements in q for which f is true
            and false_ct is the number of elements in q for which f is false
       *)

       fun excess ( nil ) = 0
         | excess (e :: q) = (if (f e) then 1 else ~1)  +  (excess q)


  in
         excess ( l ) >  0

  end
```

In the next two parts, you will prove by induction over *q* that *excess(q)* computes the right value.

(c) [5 pts] State the inductive hypothesis and prove it for the base case.

Induction hypothesis: for all lists q and predicates f, excess(q) is true_ct - false_ct, where true_ct is the number of elements in q for which f is true, and false_ct is the number of elements in q for which f is false.

The induction is over the list q. The base case is q = nil. For no elements, true_ct and false_ct are both 0, and excess(q) returns 0.

(d) [5 pts] Prove the induction step.

Assume the hypothesis holds for some list q.
Consider the list q' = e :: q.
true_ct (q') = true_ct (h) + true_ct (q), and similarly for false_ct. So
true_ct(q') - false_ct(q') = true_ct([e]) - false_ct([e]) + (true_ct(q) - false_ct(q)).
If (f e) is true, true_ct([e]) - false_ct([e]) = 1 - 0 = 1
If (f e) is false, true_ct([e]) - false_ct([e]) = 0 - 1 = ~1
By hypothesis, excess(q) evaluates to (true_ct(q) - false_ct(q)). So (if (f e) then 1 else ~1) + (excess q)) evaluates to true_ct (q').

(e) [2 pts] Finally, using results proved so far, prove the correctness of mostly_true.

f is true on the majority of the elements of a list q if true_ct(q) > false_ct(q).
But true_ct(q) > false_ct(q) exactly when true_ct(q) - false_ct(q) > 0,
so (excess(q) > 0) produces the correct result.

(f) [2 pts] What is the precondition of mostly_true? (Hint: think about termination.)

The precondition of mostly_true (q, f) is that f must terminate when applied to any element of q.
(Puzzle: could you implement a version of mostly_true that does not require f to terminate on all elements if (f e) terminates with the value true for most elements?)

name:

3 Representation Invariants

Given a binary tree whose representation is:

```
type key = string
datatype tree = Empty | Node of (tree * key * int * tree)
```

with the representation invariant:

Empty is a valid tree
Node(left, k, left_size, right) is valid iff:

(1) for every key *kl* in *left*, *kl* < *k*

(2) for every key *kr* in *right*, *k* < *kr*

(3) *left_size* is equal to the number of non-empty nodes in *left*

write the following functions:

(a) [9 pts] A function

```
lookup : tree * int -> key
```

that, given a tree and an integer *i* returns the *i*-th smallest key in the tree. Specify any preconditions your function requires to work correctly.

```
(* lookup(t,i)
   Assumes: 1 <= i <= size of t
   Yields: i-th smallest key in the tree t
*)
fun lookup(Node(left, k, left_size, right), i) =
if (i <= left_size) then
  lookup(left,i)
else if (i = left_size + 1) then
  k
else
  lookup(right, i - left_size - 1)
```

(b) [12 pts] A function

insert : tree * key -> tree

that, given a tree t and a key k, returns the tree resulting from adding a node with key k to the tree t. Specify any preconditions your function requires to work correctly.

```
(* insert(t, k)
   Assumes: Key k does not appear in tree t
   Yields: the result of adding key k to tree t.
*)
fun insert(Empty, k') = Node(Empty, k', 0, Empty)
  | insert(Node(left, k, left_size, right), k') =
   if (k' < k) then
     Node(insert(left, k'), k', left_size + 1, right)
   else
     Node(left, k, left_size, insert(right,k'))
```

(c) [2 pts] Which of the two functions *insert* and *lookup* makes use of the invariant?

both of them

(d) [2 pts] Which of the two functions *insert* and *lookup* does work to maintain the invariant?

insert alone

name:

4 Lists

(a) [12 pts] Write a function

punctuate : (string * string) -> string list -> string

that, given a separator symbol and a terminator symbol returns a function that converts a list of strings into a single punctuated string.

For example,

(punctuate (“, ”, “! ”)) [“milk”, “eggs”, “bananas”]

evaluates to the string “milk, eggs, bananas!”. Recall that the operator that appends two strings is ^.

```
fun punctuate (delimiter, terminator) phrases =
  let
    fun d'([]) = ""
    |   d'(p::nil) = p ^ terminator
    |   d'(p::ps) = p ^ delimiter ^ d'(ps)
  in
    d'(phrases)
  end;
```

(b) [13 pts] Write a function

countMissing : int list -> int

that takes as input a sorted list of integers without repetitions, representing check numbers, and produces as a result the number of checks missing from the list. For example,

countMissing [101,102,105,107]

evaluates to 3 since the checks numbered 103, 104, and 106 are missing.

```
fun countMissing (nil) = 0
        | countMissing (x::nil) = 0
        | countMissing (x::y::ys) =
                y-x -1 + countMissing(y::ys)
```

name:

name: