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

February 24, 1998

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	30	40	30	100

Question 1: Operations on Integer Lists [30 Points]

1. **Max and Min** [10 **Points**]

   Write a function `extrema` with the following type:

   ```
   val extrema : int list -> int * int
   ```

   Given a list of integers, `extrema` should return a pair of integers representing the minimum and the maximum of the integers in the list. For example,

   ```
   extrema [1, 2, ~3, 4, 5, ~8]  ===>  (~8, 5)
   ```

   To simplify your code, the variable `posinf` is bound to the largest positive machine-representable integer and `neginf` is bound to the most negative machine-representable integer. You may find these two SML functions useful:

   ```
   Int.min : int * int -> int    (* returns minimum of two integers *)

   Int.max : int * int -> int    (* returns maximum of two integers *)
   ```

   Here is a code template. Please fill in the blank.

```
(*   val extrema : int list -> int * int   *)



val extrema = foldr ________________________________________ (posinf, neginf);
```

2. **Merging Two Sorted Lists** [10 **Points**]

   Write a function `merge : int list * int list -> int list` which takes two sorted lists of integers and merges them into a single sorted list of integers.

   INVARIANT: Each list is sorted in ascending order and contains no duplicates.

   Thus: (a) You may assume that the original two lists are sorted and contain no duplicate elements. (b) Your resultant list also should be sorted and contain no duplicate elements.

   Example:

   ```
   merge ([1, 4, 5, 8, 11], [~3, 2, 4, 10, 11])

       ===>   [~3, 1, 2, 4, 5, 8, 10, 11]
   ```

   On the next page is a code template. Please fill in the blanks.

```
(*   val merge : int list * int list -> int list   *)

fun merge (l1, nil) = ______________________________

  | merge (nil, l2) = ______________________________

  | merge (l1 as h1::l1', l2 as h2::l2') =


        if (h1 = h2) then _________________________________


        else if ______________________________________


        else ______________________________________;
```

3. **Merging a List of Sorted Lists** [10 **Points**]

   Now write a function `mergeall : int list list -> int list` which takes a list of lists, each of which is a sorted list of integers as in Part 2 above, and merges them all into a single sorted list of integers.

   INVARIANT: Each integer list is sorted in ascending order and contains no duplicates.

   Example:

   ```
   mergeall [[~1, 2, 3], [2, 4, 5], [~6, 3, 5, 8]]

         ===>  [~6, ~1, 2, 3, 4, 5, 8]
   ```

   Here is a code template. Please fill in the blanks.

   You may use the function `merge` from Part 2.

```
(* val mergeall : int list list -> int list *)


val mergeall = foldr ____________________  ____________;
```

Operations on Integer Lists

1.
```
val SOME posinf = Int.maxInt;
val SOME neginf = Int.minInt;

(* val extrema : int list -> int * int *)
val extrema = foldr (fn (h, (mi, mx)) => (Int.min(mi, h), Int.max(mx, h)))
                    (posinf, neginf);
```

2.
```
(* val merge : int list * int list -> int list *)
fun merge (l1, nil) = l1
  | merge (nil, l2) = l2
  | merge (l1 as h1::l1', l2 as h2::l2') =
        if (h1 = h2) then h1::merge(l1', l2')
        else if (h1 < h2) then h1::merge(l1', l2)
        else h2::merge(l1, l2');
```

3.
```
(* val mergeall : int list list -> int list *)
val mergeall = foldr merge nil;
```

Question 2: Self-Checking Code [40 Points]

A useful technique for improving the reliability of programs is to augment your program with code to certify the validity of the output of a function, reporting an error if the check fails. While self-checking code may seem like a natural idea, it is often difficult to devise a checker for a given function that is both efficient to execute and effective at catching errors.

In this question we will consider a self-checking implementation of the g.c.d algorithm. Recall that the g.c.d. of two non-negative integers m and n is the integer p such that (a) p divides m (*i.e.*, $m = k \times p$ for some integer $k \geq 0$), (b) p divides n, and (c) if q divides m and n, then q divides p. (By convention the g.c.d. of 0 and 0 is 0.)

Euclid's Algorithm for computing the g.c.d. is written in ML as follows:

```
(** Input: m, n >= 0.  Output: g.c.d. of m and n. **)
fun gcd (0, n) = n
  | gcd (m, 0) = m
  | gcd (m, n) = if m > n then gcd (m mod n, n) else gcd (m, n mod m)
```

Note that this is not the same algorithm as was presented in lecture! The recursive calls are on $m \bmod n$ and $n \bmod m$, rather than $m - n$ and $n - m$, respectively.

1. [12 Points] Prove that if $m, n \geq 0$, then `gcd(`m`,`n`)` evaluates to the greatest common divisor of m and n. Proceed by induction on the *product* of m and n. *Hint:* The following equation will be useful in your proof:

   $$m \bmod n = m - ((m \operatorname{div} n) \times n)$$

   In words: $m \bmod n$ is obtained by casting out as many n's as possible from m.

   Base case: $m \times n = 0$

   Inductive step: $m \times n > 0$. Assume that the program is correct for all m' and n' such that $m' \times n' < m \times n$. Show that it is correct for m and n:

We can build a checker for `gcd` based on the following fact: d is the g.c.d. of m and n iff d divides m and n and

$$d = a \times m \ + \ b \times n$$

for some integers a and b. For example, if $m = 12$ and $n = 18$, then their g.c.d., 6, can be written as

$$6 = (-1) \times 12 \ + \ 1 \times 18$$

Given m and n, if we compute both the g.c.d d and the coefficients a and b, then we can check the result by checking that d divides both m and n (ensuring that d is a common divisor) and that $d = a \times m + b \times n$ (ensuring that it is the g.c.d.).

2. [16 Points] You are to define a function

```
scgcd : int * int -> int * int * int
```

satisfying the following specification:

> if $m, n \geq 0$, then `scgcd(`m`, `n`)` evaluates to `(`d`, `a`, `b`)`, where d is the g.c.d. of m and n and $d = a \times m \ + \ b \times n$.

The algorithm is a variation on Euclid's algorithm obtained by performing a small arithmetic calculation to determine the coefficients a and b. We'll give you the first two cases; please fill in the remaining two cases where indicated.

```
fun scgcd (0, n) = (n, 0, 1)                    (* n = 0m + 1n *)
  | scgcd (m, 0) = (m, 1, 0)                    (* m = 1m + 0n *)
  | scgcd (m, n) =
    if m > n then
       let
           val (d, a, b) = scgcd (m mod n, n)
       in



           ________________________________________
       end
    else
       let
           val (d, a, b) = scgcd (m, n mod m)
       in



           ________________________________________
       end
```

3. [12 Points] Prove the correctness of the "then" clause of your implementation by giving a *short* (three or four line) algebraic calculation that justifies your answer. You will need to make use of the inductive assumption for the recursive calls.

Self-Checking Code

1. If the product $m \times n$ is zero, then one or the other (or both) are zero, from which the result follows immediately. Otherwise, proceed by cases according to whether $m > n$ or $m \leq n$. In the former case observe that $(m \bmod n) \times n < m \times n$, so inductively we obtain the gcd d of $m \bmod n$ and n. Applying the hint, $m \bmod n = m - (m \operatorname{div} n) \times n$, so if d is a divisor of $m \bmod n$ and n, then $m \bmod n = k \times d = (m - (m \operatorname{div} n) \times n)$ and $n = l \times d$ for some integers k and l. So $k \times d = (m - (m \operatorname{div} n) \times l \times d)$, so $m = (k + (m \operatorname{div} n) \times l) \times d$ — that is, d is a divisor of m. Notice that this argument also shows that a divisor of m and n is also a divisor of $m \bmod n$ and n, so their greatest common divisors coincide. The other case $(m \leq n)$ is exactly analogous.

2. `(d, a, b - a*(m div n))` and `(d, a - b*(n div m), b)`.

3. The proof in the third case is as follows. Inductively we have that $d = a \times (m \bmod n) + b \times n$. Applying the hint, we have $d = a \times m - a \times (m \operatorname{div} n) \times n + b \times n = a \times m + (b - a \times (m \operatorname{div} n)) \times n$. The other case is similar.

Question 3: One-Time Passwords [30 Points]

One-Time Passwords are passwords that work once. After you've used such a password, you need to use a new password the next time. An acceptor function checks to make sure you are using the correct password, and then tells you what your next password will be.

For instance, maybe you will be using the nonnegative even numbers as your sequence of passwords:

```
0, 2, 4, 6, 8, 10, 12, ....
```

The first time, you use 0 as your password. If you use it correctly, then the next time you have to use 2 as your password, and so forth.

Along with this sequence of passwords is a sequence of acceptor functions:

$$\mathtt{acc}_0,\ \mathtt{acc}_1,\ \mathtt{acc}_2,\ \mathtt{acc}_3,\ \ldots$$

Here $\mathtt{acc}_0$ is a function that checks to make sure you are using the password 0. If so, then it returns *two items*, namely:

1. the new password 2, *and*
2. the new acceptor function $\mathtt{acc}_1$.

In turn, $\mathtt{acc}_1$ checks that you next use 2 as your password. If so, $\mathtt{acc}_1$ returns 4 and $\mathtt{acc}_2$. In other words, $\mathtt{acc}_1$ tells you that 4 will be your next password, and it creates a new acceptor function for that password. And so forth.

The key point: Each acceptor function, when called with the correct password, returns a new password and a new acceptor function for that password. If called with an incorrect password, it returns "`Wrong`".

In ML let's model one-time passwords using the following datatype:

```
datatype passaccept = Wrong | Good of int * (int -> passaccept)
```

We can then think of one-time passwords as a sequence of values

$$\mathtt{Good}(\mathtt{p}_0,\ \mathtt{acc}_0),\ \mathtt{Good}(\mathtt{p}_1,\ \mathtt{acc}_1),\ \ldots$$

where

$$\mathtt{Good}(\mathtt{p}_i,\ \mathtt{acc}_i)\ :\ \mathtt{passaccept}$$

$$\mathtt{p}_i\ :\ \mathtt{int},$$
$$\mathtt{acc}_i\ :\ \mathtt{int\ \text{->}\ passaccept}.$$

The idea is: $\texttt{p}_i$ is the password you use at the i^{th} step. If you use it correctly, then $\texttt{acc}_i$ returns the next password along with the next acceptor function. In other words,

$$\texttt{acc}_i(\texttt{n}) = \begin{cases} \texttt{Good}(\texttt{p}_{i+1}\texttt{,}\ \texttt{acc}_{i+1}\texttt{)}, & \text{if } \texttt{n} = \texttt{p}_i, \\ \texttt{Wrong}, & \text{otherwise.} \end{cases}$$

Our job as programmers is to write code for $\texttt{acc}_0$ to start off the whole sequence.

1. [10 **points**] For the password sequence of nonnegative even integers `0, 2, 4, 6, ...` we would write code for $\texttt{acc}_0$ as follows. Please fill in the missing blanks:
   [CAUTION: Look at the function `check` and determine its type carefully.]

```
(* val int0 : int -> passaccept *)

fun int0(n) =
  let

     fun check p i =  if (i=p) then Good(____________ , check ____________)

                      else Wrong
  in
     check 0 n
  end;
```

2. [20 **points**] Now suppose the desired password sequence is the Fibonacci numbers

   `1, 1, 2, 3, 5, 8, 13, ...`

   Write the function `fib0 : int -> passaccept` to start off the whole process. Here is some skeleton code. Fill in the blank lines.

```
(* val fib0 : int -> passaccept *)

fun fib0(n) =
  let
     fun check (p1, p0) i =


           if (i = p1) then ______________________________________________

                        else Wrong
  in


     ________________________________________________________
  end;
```

One-Time Passwords

1.
```
(* val int0 : int -> passaccept *)
fun int0(n) =
  let
     fun check p i =  if (i=p) then Good(p+2, check (p+2))
                      else Wrong
  in
     check 0 n
  end;
```

2.
```
(* val fib0 : int -> passaccept *)
fun fib0(n) =
  let
     fun check (p1, p0) i =
           if (i = p1) then Good(p1+p0, check (p1+p0, p1))
                       else Wrong
  in
     check (1, 0) n
  end;
```

Worksheet

Worksheet