Instructions

Write your name and netid on this page. There are 5 problems on this exam; check now that you have the whole exam. The exam is worth 100 points total. The point breakdown for the parts of each problem is printed with the problem. Do all written work on the exam itself. If you are running low on space, write on the back of the exam sheets and be sure to write (OVER) on the front side. This is an closed-book examination; you *may not* use outside materials, calculators, computers, etc. You have 2 1/2 hours. **Good luck!**

Name and NetID ______________________________

Problem	Points	Score
1	20	
2	20	
3	20	
4	20	
5	20	
Total	100	

1. True/False [20 pts] (parts a–j)

Each correct answer is 2 pts; each wrong answer is -2 pts; and each blank answer is 0 pts.

a. ____ Software testing proves the presence of bugs, but cannot prove their absence.

b. ____ The type that SML infers for the expression `fn (x,y) => fn x => (y,x)` is: `'a * 'b -> 'c -> 'b * 'c`.

c. ____ The function $f(n) = \lg(n \ \lg n)$ is $O(\lg n)$.

d. ____ At each collection, a copying collector must traverse all of the data in the program (including data unreachable from the roots).

e. ____ The implementation of Dijkstra's shortest-paths algorithm requires a stack data structure.

f. ____ The function `foldl` is tail-recursive.

g. ____ It is possible that a hash table with n elements and a load factor of 2 has a bucket that contains all of the n elements.

h. ____ When a program exhibits temporal locality, it will access the same memory location in the near future.

i. ____ Any lookup operation in a splay tree with n nodes is $O(\lg n)$.

j. ____ Data races can occur during the execution of message-passing concurrent programs.

2. Sets [20 pts] (parts a–c)

The following is a standard set interface:

```
signature SET = sig
 (* A 'a set is a set of items of type 'a. *)
 type 'a set

 (* empty is the empty set *)
 val empty : 'a set
 (* add(s,e) is s union {e} *)
 val add: 'a set * 'a -> 'a set
 (* fold over the elements of the set *)
 val fold: ('a*'b->'b) -> 'b -> 'a set -> 'b
end
```

(a) [5 pts] Extend the interface with a function `remove` that removes an item from a set. Provide a signature *and* a specification for `remove`. Define an appropriate exception if necessary.

(b) [7 pts] Write an implementation for function `remove` using the other functions in the signature. Assume that an equality funtion for items `equal':'a*'a->bool` is also available (`'a` being the type of the items in the set). Your function `remove` should not visit any of the items more than once.

(c) [8 pts] Consider now a function `cartprod` that takes two sets of items and yields a set of pairs representing the Cartesian product:

```
(* cartprod(s1,s2) is the cartesian product of s1 and s2 *)
val cartprod: 'a set * 'a set -> ('a * 'a) set
```

Remember that the Cartesian product $A \times B$ of two sets A and B is the set of all pairs (a, b) where $a \in A$ and $b \in B$. That is, $A \times B = \{(a, b) \mid a \in A,\ b \in B\}$.

For simplicity, assume that sets are implemented using lists (`type 'a set = 'a list`). Below are some examples of using `cartprod`:

```
cartprod ([1,2], [3,4]) = [(1,3), (1,4), (2,3), (2,4)]
cartprod (["a"], ["b", "c"]) = [("a","b"), ("a","c")]
cartprod ([1,2], []) = []
```

Write the function `cartprod`, assuming a list implementation of sets. You *may not* use the list concatenation operator "`@`" in your solution.

Note: It is possible to write `cartprod` such that is works for any implementation of sets, not only lists. Feel free to write such a function.

3. Trees [20 pts] (parts a–c)

The following is the standard datatype for binary search trees containing integer values:

```
datatype tree = Leaf | Node of tree * int * tree
```

(a) [5 pts] Consider two functions `min` and `max` that compute the smallest and the largest numbers in a tree:

```
(* min(t) is the smallest element of t,
 *        or the largest integer if t is a leaf. *)
val min : tree -> int
(* max(t) is the largest element of t,
 *        or the smallest integer if t is a leaf. *)
val max : tree -> int
```

Using these functions, write a function `repOK : tree -> bool` that returns true if and only if the tree satisfies the binary search tree invariant. (For an informal description of the invariant you'll receive partial credit).

(b) [5 pts] Several kinds of binary trees (including AVL, red-black, and splay trees) use rotations for rebalancing. The following is the basic right rotation:

```
fun rotate (t:tree) :tree =
    case t of
      Node(Node(A,x,B), y, C) => Node(A, x, Node(B,y,C))
    | _ => t
```

Show that the above function `rotate` maintains the binary search tree invariant.

(c) [10 pts] Consider now an imperative implementation of binary search trees:

```
datatype itree = Leaf | Node of (itree ref) * int * (itree ref)

(* irotate(t) performs a right rotation at the root of t
 * Effects: destructively updates t
 * Returns: the new root after the rotation. *)
val irotate: itree -> itree
```

Write an implementation for `irotate`.

4. Correctness and Complexity [20 pts] (parts a–g)

Consider the following program:

```
fun f (x, y) =
  if y = 0 then 1 else
    let
      val p = f (x, y div 2)
      val sp = p * p
    in
      if y mod 2 = 0 then sp else x * sp
    end
```

(a) [3 pts] What does `f(x,y)` compute?

The following questions ask you to prove the correctness of function `f`, with respect to your answer above.

(b) [2 pts] Write the property $P(n)$ that you need to prove and specify the initial value n_0 of n.

(c) [2 pts] State whether you'll use strong or weak induction.

(d) [2 pts] Prove the base case.

(e) [4 pts] State the induction hypothesis and prove the induction step.

Next, analyze the run-time complexity of `f`.

(f) [4 pts] Write the recurrence relations for the running time of `f(2,n)`. Use constants c_1, c_2, etc. for operations that take constant time.

(g) [3 pts] What is the run-time complexity of `f(2,n)`? You don't have to prove your result.

5. Environment Model [20 pts] (parts a–d)

The program below is written in a ML-like language that doesn't allow recursive functions, but has references and higher-order functions:

```
let val rf = ref (fn x => x)
    val f = fn y => let val (n,p,q) = y in
                    if n = 0 then y else
                    (!rf)(n - 1, q, p + q)
                    end
    val () = rf := f
in
    f(5,1,1)
    (* GC *)
end
```

(a) [3 pts] What are the types of `f` and `rf`?

(b) [3 pts] What is the result of the evaluation?

(c) [10 pts] Draw the environment diagram that arises during the evaluation of the second call to `f` (i.e., when the program starts evaluating the function body for that call).

(d) [4 pts] What heap cells (not environment entries!) can a garbage collector reclaim at the program point labeled `GC` in the code?