Preliminary Examination II
Computer Science 312, Cornell University 15 April 2003

Before starting the exam, write your name on this page and your netid on both this page and the next page.

There are 5 problems on this exam. It is 11 pages long; make sure you have the whole exam. You will have 90 minutes in which to work on the problems. You will likely find some problems easier than others; read all problems before beginning to work, and use your time wisely. The prelim is worth 100 points total. The point breakdown for the parts of each problem is printed with the problem. Some of the problems have several parts, so make sure you do all of them!

This is an closed-book examination; you **may not** use outside materials, calculators, computers, etc.

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. It is to your advantage to show your work — we will award partial credit for incorrect solutions that are headed in the right direction. If you feel rushed, try to write a brief statement that captures key ideas relevant to the solution of the problem.

If you finish in the last ten minutes of the exam, please remain in your seat until the end of the exam as a courtesy to your fellow students.

Name and NetID _____

Problem	Points	Score
1	10	
2	20	
3	25	
4	20	
5	25	
Total	100	

1. True/False [10 pts]

   (parts a–e; **4** points off for each wrong answer, 2 points off for each blank answer, minimum problem score 0.)

a. ____ Imperative data abstractions have mutators.

b. ____ The red-black tree invariant ensures that every path from the root to the leaf has the same height.

c. ____ The worst-case performance of hash-table lookup is $O(n)$ where n is the number of elements, even if the number of buckets is n.

d. ____ The update operation (`:=`) is $O(1)$ in a language implementation that manages memory through reference counting.

e. ____ Breadth-first search of a graph requires time proportional to $n \lg n$ where n is the number of nodes and edges in the graph.

2. Zardoz Refs [20 pts]

For each of the following expressions, give a *value* that causes the expression to evaluate to 42 if the box ☐ is replaced by that value.

(a) [10 pts]

```
let
   val zardoz: 'a * 'a ref * 'a ref * 'a ref ref -> unit =
      ☐
   val x: int ref ref = ref(ref(1))
   val y: int ref ref = ref(ref(1))
   val z: int ref = ref(8)
in
   zardoz(6,z,!x,y);
   (!(!x))  * (!(!y) - 1)
end
```

Your answer:

(b) [10 pts]

```
let val zardoz: unit->unit->int = ☐
    val f = zardoz()
in
  f() + f() + 1
end
```

Your answer:

3. Correctness [25 pts]

Consider the following data abstraction for Booleans and its implementation:

```
signature BOOL = sig
  (* A "boolean" is a Boolean with the usual operations. *)
  type boolean
  val true: boolean
  val false: boolean
  val and_: boolean * boolean -> boolean
  val or: boolean * boolean -> boolean
end
structure Boolean :> BOOL = struct
  type boolean = int
  val true = 1
  val false = 0
  fun and_(x,y) = x*y handle overflow => true
  fun or(x,y) = x+y handle overflow => true
end
```

(a) [3 pts] What is the abstraction function for this implementation?

(b) [3 pts] What is the representation invariant maintained by this implementation?

(c) [4 pts] Now let's prove that the function `or` is implemented correctly. Start by stating a proposition that, if true, means `or` is implemented correctly. This proposition should be expressed using the abstraction function *AF* and the representation invariant *RI*.

(d) [15 pts] Prove the proposition you stated in part 3(c). *Hint*: consider possible cases on x and y.

4. Complexity [20 pts]

Let $T(n)$ be the time to perform a merge sort of n elements. The recurrence is:

$$T(1) = 1$$

$$T(n) = 2T(n/2) + n + 1$$

Let's `prove` that $T(n) = O(n)$ for all n by induction on n.

Base case: $T(1) = 1 = O(1)$
Assume that for any m, $1 \leq m < n$, $T(m) = O(m)$. Then

$$\begin{aligned} T(n) &= 2T(n/2) + n + 1 \\ &= 2O(n/2) + n + 1 \qquad \text{by IH} \\ &= O(2(n/2) + n + 1) \\ &= O(2n + 1) \\ &= O(n). \end{aligned}$$

"QED".

(a) [1 pt] What is the correct asymptotic complexity of merge sort?

(b) [10 pts] What's wrong with this proof? Explain briefly how and where the reasoning is incorrect.

Consider the following recurrence:

$$T(1) = 1$$

$$T(n) = 2T(n/2) + n^2$$

Use the substitution method to determine whether the following statements are true or false. Show your work.

(c) [3 pts] $T(n)$ is $O(n)$

(d) [3 pts] $T(n)$ is $O(n \lg n)$

(e) [3 pts] $T(n)$ is $O(n^2)$

5. Type Checking [25 pts]

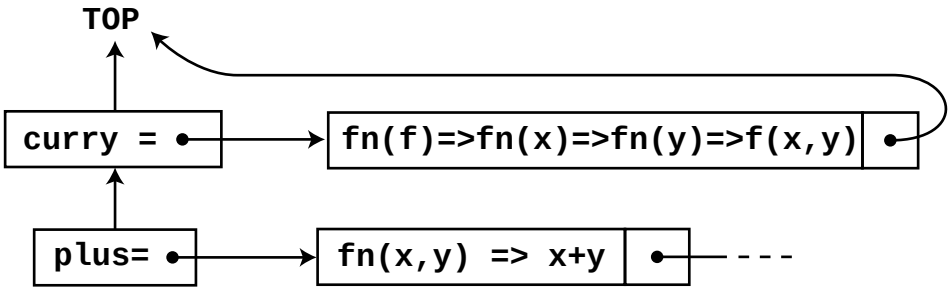
The following two functions transform a two-argument function between its curried and uncurried forms:

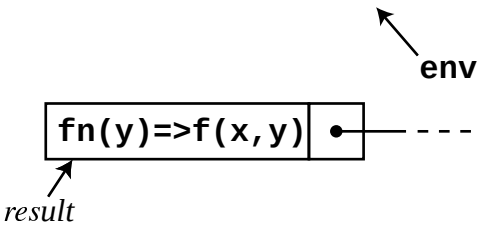
```
val curry: ('a*'b->'c) -> ('a->'b->'c) =
    fn (f: 'a*'b->'c) => fn(x:'a) => fn(y:'b) => f(x,y)
val uncurry: ('a->'b->'c) -> ('a*'b->'c) =
    fn (f: 'a->'b->'c) => fn(x:'a, y':b) => f x y
```

Consider the following code that uses `curry`:

```
let fun plus(x:int,y:int):int = x+y
    val x:string = "hi"
    val y: int->int->int = curry plus
in
    y 2
end
```

(a) [10 pts] Show the contents of the environment (including both `x`'s) and the heap at the end evaluating the `let` block (but while `x` and `y` are still in scope.) Part of the diagram is drawn below; complete it. (Note that `curry` is assumed to be already present in the environment as shown.) You may use the back side as scratch paper or redraw the whole figure there if you need more room.





A type can be interpreted as a logical proposition, where the product type operator `*` corresponds to Boolean "and" ($\wedge$), the datatype separator `|` corresponds to Boolean "or" ($\vee$), and the function type operator `->` corresponds to Boolean implication ($\Rightarrow$). For example, the type `'a->('b->'c)` corresponds to a proposition $A \Rightarrow (B \Rightarrow C)$. A type like `int` corresponds to the proposition "some integer exists".

Remarkably, a proposition holds[1] only when we can find a term whose type corresponds to that of the proposition. For example, the proposition $A \Rightarrow A$ holds for all propositions A; the term `fn(x:'a)=>x` is a proof of this claim! Conversely, if a proposition is false, then we can find no term that has the corresponding type (if we aren't allowed to use certain SML features: for example, refs or any recursion leading to nontermination). Thus, there is no term of the type `'a->'b` because the proposition $A \Rightarrow B$ is not true for all A and B.

In logic, we can show that two propositions X and Y are equivalent (written $X \equiv Y$) by showing both $X \Rightarrow Y$ and $Y \Rightarrow X$. Because propositions are types, the equivalence of two propositions has a computational significance: it means that there must exist a pair of functions that map back and forth between the types that correspond to the two propositions. For example, the `curry` and `uncurry` functions prove the logical equivalence $A \Rightarrow (B \Rightarrow C) \equiv (A \wedge B) \Rightarrow C$.

(b) [8 pts] In logic, $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$, because "and" distributes over "or". To represent "or", we use this datatype:

```
datatype ('x, 'y) sum = Left of 'x | Right of 'y
```

Prove this logical equivalence by implementing these declarations with functions that always terminate:

```
val forward: 'a*('b,'c) sum -> ('a*'b, 'a*'c) sum
val backward: ('a*'b, 'a*'c) sum -> 'a*('b,'c) sum
```

Write down all type declarations explicitly.

[1] in "constructive logic", which isn't as powerful as the classical logic covered in CS 280.

(c) [7 pts] Similarly, show $A \Rightarrow (B \wedge C) \equiv (A \Rightarrow B) \wedge (A \Rightarrow C)$ by defining two terminating functions that map between the types `'a->'b*'c` and `('a->'b)*('a->'c)`.