CS312 Fall 2001
Final Exam
December 20, 2001

Name______________________________________

NetID__________**@cornell.edu**

Instructions

Do not begin until instructed. Write your name and netID on the lines above. There are 8 numbered problems on 10 pages. Please check now that you have a complete exam with no missing pages.

The exam is $2\frac{1}{2}$ hours, closed book. No papers, books, notes, or electronic devices of any kind may be used, except for the SML cheat sheet, which will be provided. Please put your answers in the boxes provided. Ambiguous answers will be considered incorrect. Extra paper is available if you need it.

Some problems are more difficult than others. Don't waste a lot of time on a problem that is giving you a hard time—move on to another problem and then return to it later. Read the questions very carefully; some of them are trick questions.

Good luck!

1	2	3	4	5	6	7	8	Σ
/12	/12	/9	/10	/9	/9	/9	/9	/79

1. (12 points) Write one-line definitions of each of the following functions. You may use previously defined functions in later definitions. Use pattern matching and higher-order functions to make your definitions as concise as possible. Example:

```
(* return the tail of a list, or [] if it is empty *)
fun safeTail(_::t | t as []) = t

(* delete the last element of a list if it is nonempty,
 * otherwise return [] *)
fun deleteLast x = rev(safeTail(rev x))
```

(a)
```
(* apply a given function to all elements of a list
 * except the first *)
```

(b)
```
(* apply a given function to all elements of a list
 * except the last *)
```

(c)
```
(* reverse a string *)
```

(d)
```
(* given list of pairs, delete all pairs whose
 * first element satisfies a given predicate *)
```

(e)
```
(* delete NONEs from a list, replace SOME x with x
 * Example:  [SOME 4,NONE,SOME 3] => [4,3] *)
```

(f)
```
(* factorial *)
fun fact n = foldl (
```

2. (12 points) Supply a definition for `zardoz` that causes the entire expression to evaluate to 42. If impossible, say so and explain why. Example:

```
let val zardoz = (42,43)
in #1 zardoz
end
```

(a)
```
let
  val zardoz = 
  val S = fn x => fn y => fn z => x z (y z)
  val K = fn x => fn y => x
in
  S K K zardoz
end
```

(b)
```
let
  val zardoz = 
  val zardoz = zardoz + (zardoz + 1)
  val zardoz = zardoz * (zardoz + 1)
in
  zardoz
end
```

(c)
```
let
  datatype zardoz = 
  val zardoz : zardoz = ZED (fn x => 42)
in
  (fn (zed as ZED f) => f zed) zardoz
end
```

(d)
```
let
  val zardoz = 
  val zed = ref 41
in
  zardoz();
  !zed
end
```

3. (9 points) A *two-dimensional list* is a list of lists.

   (a) (7 points) Write a general procedure `twoDFind` that takes two arguments, a Boolean-valued function `pred` and a two-dimensional list `twoDList`, and finds the first element in `twoDList` satisfying the predicate, returning `NONE` if there is no such element. Here "first" means the first element satisfying the predicate on the first list of `twoDList` containing such an element. Example:

```
- fun odd x = x mod 2 = 1;
val odd = fn : int -> bool
- twoDFind odd [[2,44,10],[],[62,39],[5,88]];
val it = SOME 39 : int option
```

```
fun twoDFind pred twoDList =
```

   You may not use refs in your solution. For full credit, your procedure should run in time proportional to the number of elements on `twoDList` in front of the element that is returned.

   (b) (2 points) What is the type of your procedure?

4. (10 points) Consider the following code.

```
let
  fun crazy x = let val y = ref x in fn () => ref y end
  val a = crazy 0
  val b = a()
  val _ = !b := 42
  val c = a()
  val d = crazy 312
  val e = d()
  val _ = e := !b
  val f = d()
in
  (!(!b), !(!c), !(!e), !(!f))
end
```

(a) (6 points) Draw the box-and-pointer diagram at the point just before the expression `(!(!b), !(!c), !(!e), !(!f))` is evaluated.

(b) (4 points) What is the value of the expression?

5. (9 points) Recall the following stream definitions:

```
datatype 'a stream = Nil | Cons of 'a * (unit -> 'a stream)

fun listToStream (s:'a list) : 'a stream =
  case s of [] => Nil
          | x::t => Cons(x, fn() => listToStream t)

fun nth(s:'a stream, n:int) : 'a =
  case s of Nil => raise Subscript
          | Cons(x,t) => if n = 0 then x else nth(t(),n-1)

fun map (f:'a -> 'b) (s:'a stream) : 'b stream =
  case s of Nil => Nil
          | Cons(x,t) => Cons(f x, let val tv = t() in fn() => map f tv end)

fun filter (f:'a -> bool) (s:'a stream) : 'a stream =
  case s of Nil => Nil
          | Cons(x,t) =>
              let val tv = t()
              in if f x
                then Cons(x, fn() => filter f tv)
                else filter f tv
              end

fun natsFrom (n:int) : int stream = Cons(n, fn() => natsFrom(n+1))

val naturals : int stream = natsFrom 0
val evens : int stream = filter (fn n => n mod 2 = 0) naturals
val odds : int stream = filter (fn n => n mod 2 = 1) naturals

(* print the first n elements of the integer stream s *)
fun printStream (s:int stream) (n:int) : unit =
   if n = 0 then print "\n"
   else case s of Nil => print "unexpected end of stream\n"
           | Cons(x,t) => (print (Int.toString x ^ " ");
                           printStream (t()) (n-1))
```

(continued on next page)

(continued from previous page)

Suppose s is a stream of natural numbers such that

- the numbers in s occur in strictly increasing order;
- there are infinitely many numbers that do *not* occur in s.

Complete the following definition of a function that takes such a stream s and produces its complement; that is, the stream of all numbers that do *not* occur in s. Examples:

```
- val multiplesOfThree = filter (fn x => x mod 3 = 0) naturals;
val multiplesOfThree = Cons (0,fn) : int stream
- printStream multiplesOfThree 20;
0 3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57
val it = () : unit
- printStream (complement multiplesOfThree) 20;
1 2 4 5 7 8 10 11 13 14 16 17 19 20 22 23 25 26 28 29
val it = () : unit
- printStream (complement evens) 20;
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
val it = () : unit
```

```
fun complement (s:int stream) : int stream = let
  fun complementFrom (s:int stream, n:int) : int stream =
    case s of
      Nil => natsFrom n
    | Cons(x,t) =>





in
  complementFrom (s,0)
end
```

6. (9 points) The following program computes *Ackermann's function*:

```
fun ackermann m n =
  case (m,n) of
    (0,_) => n+1
  | (_,0) => ackermann (m-1) 1
  | _ => ackermann (m-1) (ackermann m (n-1))
```

Rewrite this in tail recursive form using continuations. Use explicit continuations, not `callcc`.

```
fun ackermann' m n k =
```

Hint. For the last clause in the case expression, you want to do the inner call to `ackermann` first, i.e. the one with arguments `m` and `n-1`, passing it a continuation that does the outer call with `m-1` and the value returned by the inner call (and a continuation—which?).

7. (9 points) Prove that the function `ackermann` of Problem 6 always halts and produces a value for any pair of nonnegative integers `m` and `n`. *Hint.* Do a double induction, or an induction within an induction. The outer induction is on `m`. For each fixed `m`, do an induction on `n`.

8. (9 points) Let's define a different encoding of natural numbers in the lambda calculus than the standard Church numerals. We will use the same encoding of Booleans, namely

$$\begin{aligned} T &\stackrel{\text{def}}{=} \lambda x.\lambda y.x \quad \text{(i.e., returns the first of two arguments)} \\ F &\stackrel{\text{def}}{=} \lambda x.\lambda y.y \quad \text{(i.e., returns the second of two arguments).} \end{aligned}$$

thus the conditional test `if` B `then` P `else` Q is encoded as BPQ. For example, we can define the Boolean negation operator as $\lambda b.bFT$, since

$$\begin{aligned} (\lambda b.bFT)F &\Rightarrow FFT \Rightarrow T \\ (\lambda b.bFT)T &\Rightarrow TFT \Rightarrow F\,. \end{aligned}$$

Now define inductively

$$\begin{aligned} \overline{0} &\stackrel{\text{def}}{=} \lambda z.F \\ \overline{n+1} &\stackrel{\text{def}}{=} \lambda z.zT\overline{n} \end{aligned}$$

For example,

$$\overline{3} = \lambda z.zT(\lambda z.zT(\lambda z.zT(\lambda z.F)))\ .$$

Show how to encode a zero test Z, the successor function S, and the predecessor function P in this encoding of natural numbers. The zero test should satisfy $Z\,\overline{0} \Rightarrow T$ and $Z\,\overline{n+1} \Rightarrow F$. The successor combinator S should satisfy $S\,\overline{n} \Rightarrow \overline{n+1}$. The predecessor combinator P should satisfy $P\,\overline{n+1} \Rightarrow \overline{n}$ and $P\,\overline{0} \Rightarrow \overline{0}$.

(a) $Z \stackrel{\text{def}}{=} \lambda n.$ $\boxed{}$

(b) $S \stackrel{\text{def}}{=} \lambda n.$ $\boxed{}$

(c) $P \stackrel{\text{def}}{=} \lambda n.$ $\boxed{}$