15–150: Functional Programming

Midterm Examination

March 8, 2012

- There are 23 pages in this examination, comprising 5 questions worth a total of 80 points.
- You have 80 minutes to complete this examination.
- Please answer all questions in the space provided with the question.
- You may refer to one double-sided 8.5" x 11" page of notes, but to no other person or source besides the course staff, during the examination.
- Your answers for this exam must be written in blue or black ink.
- Unless otherwise indicated, you do not need to give purpose/examples/tests for functions.
- In multi-part coding questions, we will assume the helper functions are correct while grading the later tasks. So it is in your interest to attempt the later tasks, even if you don't solve the earlier ones.

Full Name: ____

Andrew ID: ____

Question:	Short Answer	Recursion	Proof	Analysis	HOFs	Total
Points:	11	16	20	19	14	80
Score:						

Question 1 [11]: Short Answer

(a) (2 points) State the formal definitions of the following terms.

i. An expression e is *valuable* if and only if

____

Solution: there exists some value v such that $e \cong v$

ii. A function $f : \alpha \to \beta$ is *total* if and only if

____

Solution: for all expressions $e : \alpha$, if e valuable then $f\ e$ valuable.

(b) (5 points) For each of the following expressions, circle all descriptors that apply. If the expression has a type, also give the type of the expression in the blank provided.

i. `9 + "hello"`

ill-typed   well-typed with type ____________________

value   valuable   total

Solution: ill-typed

ii. `(9 + 5)*43`

ill-typed   well-typed with type ____________________

value   valuable   total

Solution: well-typed with type `int`; valuable

iii. `(fn x :  int => 1 div 0)`

ill-typed   well-typed with type ____________________

value   valuable   total

Solution: well-typed with type `int` $\to$ `int`; value; valuable

iv. `(fn x :  int => 500)`

ill-typed   well-typed with type ____________________

value   valuable   total

Solution: well-typed with type `int` → `int`; value; valuable; total

v. `(fn x :  int => 9)(1 div 0)`

ill-typed well-typed with type ____________________

value valuable total

Solution: well-typed with type `int`

(c) (4 points) Give the most general, possibly polymorphic, type of each of the following expressions.

i. `fn x => x ^ "dolly"`

> **Solution:** `string` $\rightarrow$ `string`

ii. `fn x => x`

> **Solution:** $\alpha \rightarrow \alpha$

iii. `fn x => "this is luis, dolly"`

> **Solution:** $\alpha \rightarrow$ `string`

iv.
```
let
  fun f x = f x
in
  f
end
```

> **Solution:** This expression has type $\alpha \rightarrow \beta$.
> The `let-in-end` lets us declare a recursive function; since we just evaluate to that function it's enough to determine the type of that function.
> To figure out the type, let's add some type variables to the declaration and then look at the body of the function to see how they get used. We know that any function has a type for its arguments and a return type, so we get
> ```
>   fun f (x : 'a) : 'b = f x
> ```
> Inside the body of `f`, we see that `f` is applied to `x`. This type checks because we assumed that `x` had type `'a` and that `f` took arguments of type `'a`. We assumed that `f` produces results of type `'b`, so the expression `f x` has type `'b`. This matches all of our assumptions, so we don't learn anything new. Our final view is the same as our initial view: that `f` is a function of type `'a -> 'b`.
> Intuitively, types are a prediction of what will happen with an expression. Since `f` obviously runs for ever on all input, we can't make any specific predictions about what will be returned and therefore have to assume the most general thing.

Question 2 [16]: Recursion

(a) (5 points) For two numbers *lower* and *upper* such that $lower \leq upper$, the *interval* $[lower, upper]$ contains the numbers from *lower* to *upper*, inclusive of the end-points:

x is in $[lower, upper]$ iff $lower \leq x \leq upper$.

We define a datatype representing the three possible relationships between a number and an interval:

```
datatype interval_order =
    NumIsWithin   (* number is in the interval *)
  | NumIsLess     (* number is less than every number in the interval *)
  | NumIsGreater  (* number is greater than every number in the interval *)
```

Your task is to write a function

```
compare_interval : int * (int * int) -> interval_order
```

such that

for all `x`, `lower`, and `upper` such that `lower` ≤ `upper`,
`compare_interval(x,(lower,upper))` determines the relationship between `x` and the interval [`lower`, `upper`].

Define the function:

```
fun compare_interval (x : int, (lower, upper) : int * int) : interval_order =
```

Solution:

```
fun compare_interval (x : int, (lower, upper) : int * int) : interval_order =
    case x < lower of
        true => NumIsLess
      | false => (case x > upper of
                     true => NumIsGreater
                   | false => NumIsWithin)
```

(b) (11 points) The following datatype `bst` represents *binary search trees*:

```
datatype bst = Empty
             | Node of bst * int * bst
```

A binary search tree is a tree that has data at each node and that is *sorted*:

- `Empty` is sorted
- `Node(l,x,r)`is sorted iff `l` is sorted, `r` is sorted,
  `x` is $\geq$ every element of `l`, and `x` is $\leq$ every element of `r`.

In this problem, you will write a function

```
subtree : bst * (int * int) -> bst
```

according to the following spec:

> For all `t`, `lower`, `upper`, if `t` is sorted and `lower` $\leq$ `upper` then `subtree(t,(lower,upper))` computes a sorted tree containing all and only the elements of `t` that are in the interval [`lower`, `upper`].

Using `compare_interval`, define the function

```
fun subtree (t : bst, _____________________ : int * int) : bst =
```

Solution:

```
fun subtree (t : bst, i : int * int) : bst =
    case t of
        Empty => Empty
      | Node(l,x,r) =>
            case compare_interval (x , i) of
                NumIsLess    => subtree (r,i)
              | NumIsGreater => subtree (l,i)
              | NumIsWithin  => Node(subtree (l,i),x,subtree(r,i))
```

THIS PAGE IS INTENTIONALLY LEFT BLANK

THIS PAGE IS INTENTIONALLY LEFT BLANK

Question 3 [20]: Proof

Recall map, reduce, and mapreduce on trees:[1]

```
datatype 'a tree =
    Leaf of 'a
  | Node of 'a tree * 'a tree

fun map (f : 'a -> 'b) (t : 'a tree) : 'b tree =
    case t of
        Leaf x => Leaf (f x)
      | Node (t1,t2) => Node (map f t1, map f t2)

fun reduce (n : 'a * 'a -> 'a) (t : 'a tree) : 'a =
    case t of
        Leaf x => x
      | Node (t1,t2) => n (reduce n t1, reduce n t2)

fun mapreduce (f : 'a -> 'b) (n : 'b * 'b -> 'b) (t : 'a tree) : 'b =
    case t of
        Leaf x => f x
      | Node (t1,t2) => n (mapreduce f n t1, mapreduce f n t2)
```

Suppose you have code like

```
reduce (fn (x,y) => x + y) (map String.size <some big tree>)
```

that adds up the sizes of all of the strings in the tree. This code can be optimized to

```
mapreduce String.size (fn (x,y) => x + y) <some big tree>
```

While the asymptotic running time is the same, the `mapreduce` version is more efficient, in terms of both time and space. In the first version, `map` creates an intermediate tree, which is immediately consumed by the `reduce`; this takes time and space. The `mapreduce` version avoids this intermediate tree, and makes one pass over the data structure, rather than two.

Deforestation is a program optimization based on this idea of eliminating intermediate trees, transforming a `map` followed by a `reduce` into a single `mapreduce`.

Question contintues on the next page

[1] Note that we've removed the `Empty` constructor to simplify this problem, so all trees have at least one element.

The justification for deforestation is the following theorem:

Theorem 1. *Fix values* `f:'a -> 'b` *and* `n:'b * 'b -> 'b`*, and assume that* `f` *is total. Then:*

For all values `t:'a tree`*,* `reduce n (map f t)` $\cong$ `mapreduce f n t`

Your task is to prove this theorem by induction on `t`. **Be sure to give a justification for each equivalence.** You may use the fact that `map f` is total (because `f` is total).

(a) (8 points) **Case for `Leaf x`:**
To show:

Solution: `reduce n (map f (Leaf x))` $\cong$ `mapreduce f n (Leaf x)`

Proof:

Solution:

	`reduce n (map f (Leaf x))`	
$\cong$	`reduce n (Leaf (f x))`	step
$\cong$	`(f x)`	step, `Leaf (f x)` valuable because `f x` valuable since `f` total
$\cong$	`mapreduce f n (Leaf x)`	rstep

(b) (12 points) **Case for** `Node(l,r)`:

IH 1:

Solution: `reduce n (map f l)` $\cong$ `mapreduce f n l`

IH 2:

Solution: `reduce n (map f r)` $\cong$ `mapreduce f n r`

To show:

Solution: `reduce n (map f (Node(l,r)))` $\cong$ `mapreduce f n (Node(l,r))`

Proof:

Solution:

	`reduce n (map f (Node(l,r)))`	
$\cong$	`reduce n (Node(map f l, map f r))`	step
$\cong$	`n(reduce n (map f l), reduce n (map f r))`	step, `map f l` and `map f r` valuable because `map f` total
$\cong$	`n(mapreduce f n l, mapreduce f n r)`	IH1 and IH2
$\cong$	`mapreduce f n (Node(l,r))`	rstep

Copied for your reference:

```
fun map (f : 'a -> 'b) (t : 'a tree) : 'b tree =
    case t of
        Leaf x => Leaf (f x)
      | Node (t1,t2) => Node (map f t1, map f t2)

fun reduce (n : 'a * 'a -> 'a) (t : 'a tree) : 'a =
    case t of
        Leaf x => x
      | Node (t1,t2) => n (reduce n t1, reduce n t2)

fun mapreduce (f : 'a -> 'b) (n : 'b * 'b -> 'b) (t : 'a tree) : 'b =
    case t of
        Leaf x => f x
      | Node (t1,t2) => n (mapreduce f n t1, mapreduce f n t2)
```

Question 4 [19]: Analysis

Recall the `'a tree` datatype from the previous problem. The following function converts a tree of characters to a string:

```
fun strapp (s1 : string, s2 : string) : string = s1 ^ s2

fun tts (t : char tree) : string =
    case t of
        Leaf c => charToString c
      | Node(l,r) => strapp(tts l , tts r)


val "young lad" = tts (Node
                       (Node (Node (Leaf #"y",Leaf #"o"),
                              Node (Leaf #"u",Leaf #"n")),
                        Node
                            (Node (Leaf #"g",Leaf #" "),
                             Node (Leaf #"l",Node (Leaf #"a",
                                                   Leaf #"d"))))))
```

In this problem, you will analyze `tts t`. Some helpful facts/guidelines:

- The work and span of `strapp(s1,s2)` are **linear** in the sum of the sizes of its arguments: On strings `s1` of length m_1 and `s2` of length m_2
  - $W_{\texttt{strapp}}(m_1, m_2)$ is $O(m_1 + m_2)$ (and this is a tight bound)
  - $S_{\texttt{strapp}}(m_1, m_2)$ is $O(m_1 + m_2)$ (and this is a tight bound)
- `charToString` takes constant time.
- The size of t is the number of `Leaf`'s:

$$size(\texttt{Leaf \_}) = 1$$
$$size(\texttt{Node}(l, r)) = size(l) + size(r)$$

- You may assume the size of `t` is a power of 2 and that `t` is balanced.
- Your recurrence should be exact, except you may use constants k_0, k_1, ...to stand for constant numbers of steps of evaluation.

Question contintues on the next page

(a) (1 point) If `t` has size n, then the length of `tts t` is ______________________

Solution:

One character is writen out to the result for each leaf of the input tree, of which we know there to be n, by the definition of the size of a tree.

n

(b) (4 points) Give a recurrence for the work of `tts t`, $W_{\texttt{tts}}(n)$, in terms of of the size n of `t`. Express your recurrence using $W_{\texttt{strapp}}$.

Solution:

$$W_{\texttt{tts}}(1) = k_0$$
$$W_{\texttt{tts}}(n) = k_1 + W_{\texttt{strapp}}(n/2, n/2) + 2W_{\texttt{tts}}(n/2)$$

(c) (4 points) Use the tree method to give a closed form for $W_{\texttt{tts}}(n)$:

Solution:

`Pulling the constants out from all terms (as they are common to all terms), we are left with a tree of some multiple of the actual work.`

```
       n          n
     /   \
  n/2     n/2     n
  / \     / \
n/4 n/4 n/4 n/4   n
       .
       .
       .
```

There are as many levels as one can divide n by 2 before reaching 1 ($\log n$), each of which contains 2^i nodes of $\frac{n}{2^i}$ work, or order n work. The total work is the product of the number of levels and the work done at each: $k * n * \log(n)$

(d) (1 point) Use this closed form to give a tight big-O bound for $W_{\texttt{tts}}(n)$.

Solution: $W_{\texttt{tts}}(n) = knlogn \in O(nlogn)$

(e) (1 point) Should the span of `tts` be different from the work? Briefly explain why.

> **Solution:** The span should be different from the work, for at each level, both recursive calls can be made in parallel (the halves of the tree are independent).

(f) (4 points) Give a recurrence for the span of `tts t`, $S_{\texttt{tts}}(n)$, in terms of of the size n of `t`. Express your recurrence using $S_{\texttt{strapp}}$.

> **Solution:**
>
> $$S_{\texttt{tts}}(1) = k_0$$
> $$S_{\texttt{tts}}(n) = k_1 + S_{\texttt{strapp}}(n/2, n/2) + S_{\texttt{tts}}(n/2)$$

(g) (3 points) Use the tree method to give a closed form for $S_{\texttt{tts}}(n)$.

> **Solution:**
>
> `Pulling the constants out from all terms (as they are common to all terms), we are left with a tree of some multiple of the actual work.`
>
> ```
>     n          n
>     |
>    n/2        n/2
>     |
>    n/4        n/4
>     .
>     .
>     .
> ```
>
> There are as many levels as one can divide n by 2 before reaching 1 (log n), each of which contains 1 node of $\frac{n}{2^i}$ work. The total work is given by summing the work done at each level over all levels: $k * \sum_{i=0}^{\log(n)} \frac{n}{2^i} = k * n * \sum_{i=0}^{\log(n)} \frac{1}{2^i} <= k * n$ (By Zeno's Paradox)

(h) (1 point) Use this closed form to give a tight big-O bound for $S_{\texttt{tts}}(n)$.

Solution: $S_{\texttt{tts}}(n) = kn \in O(n)$

THIS PAGE IS INTENTIONALLY LEFT BLANK

Question 5 [14]: HOFs

Recall the following functions on the natural numbers:

```
(* Purpose: for all nats n, exp n == 2^n *)
fun exp (n : int) : int =
    case n of
        0 => 1
      | n => 2 * exp (n-1)

(* Purpose: for all nats n, fastfib n == (fib (n - 1) , fib n) *)
fun fastfib (n : int) : int * int =
    case n of
        0 => (0 , 1)
      | _ =>
        let
            val (x : int , y : int) = fastfib (n - 1)
        in
            (y , x + y)
        end
```

Both of these functions follow the *template for structural recursion on the natural numbers*. They consist of:

- A *base* case for `0`
- A *step* to compute the answer for `n`, in terms of the result of the recursive call on `n - 1`.

In this problem, you will abstract the template for structural recursion on the natural numbers into a higher-order function `iter`.

Question contintues on the next page

(a) (7 points) Fill in the type annotations and define the function `iter`:

```
fun iter (step : _____________________________)

         (base : ____________)

         (n : int) : ____________ =
```

Solution:

```
fun iter (step : 'a -> 'a) (base : 'a) (n : int) : 'a =
    case n of
        0 => base
      | _ => step (iter step base (n - 1))
```

(b) (4 points) Define `exp` and `fastfib` using `iter`:

```
val exp : int -> int =
```

Solution:

```
iter (fn x => 2 * x) 1
```

```
val fastfib : int -> int * int =
```

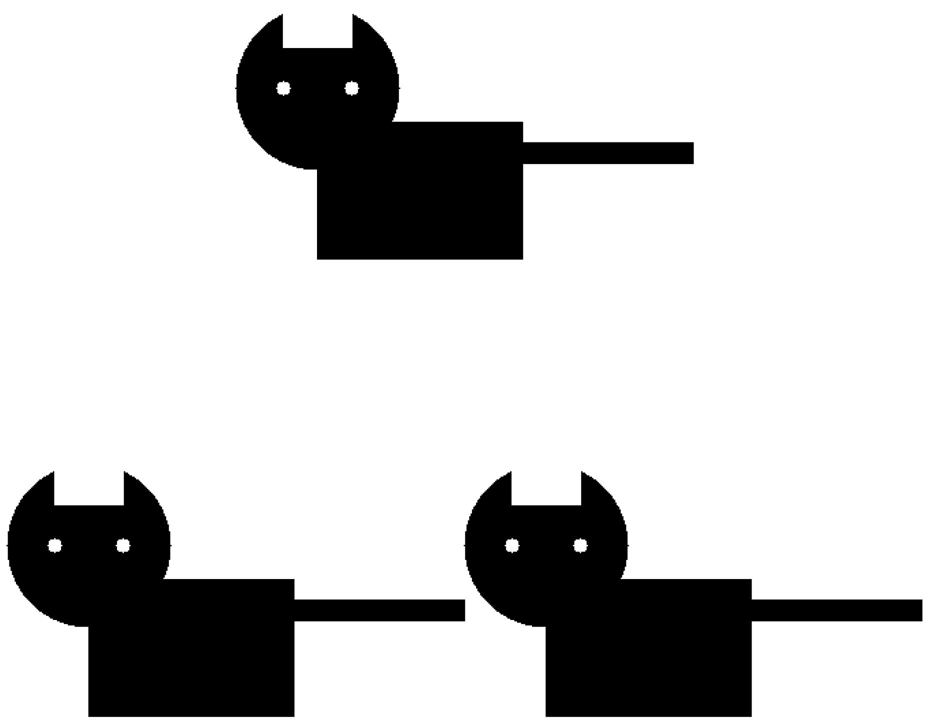
Solution:

```
iter (fn (x,y) => (y, x + y)) (0,1)
```

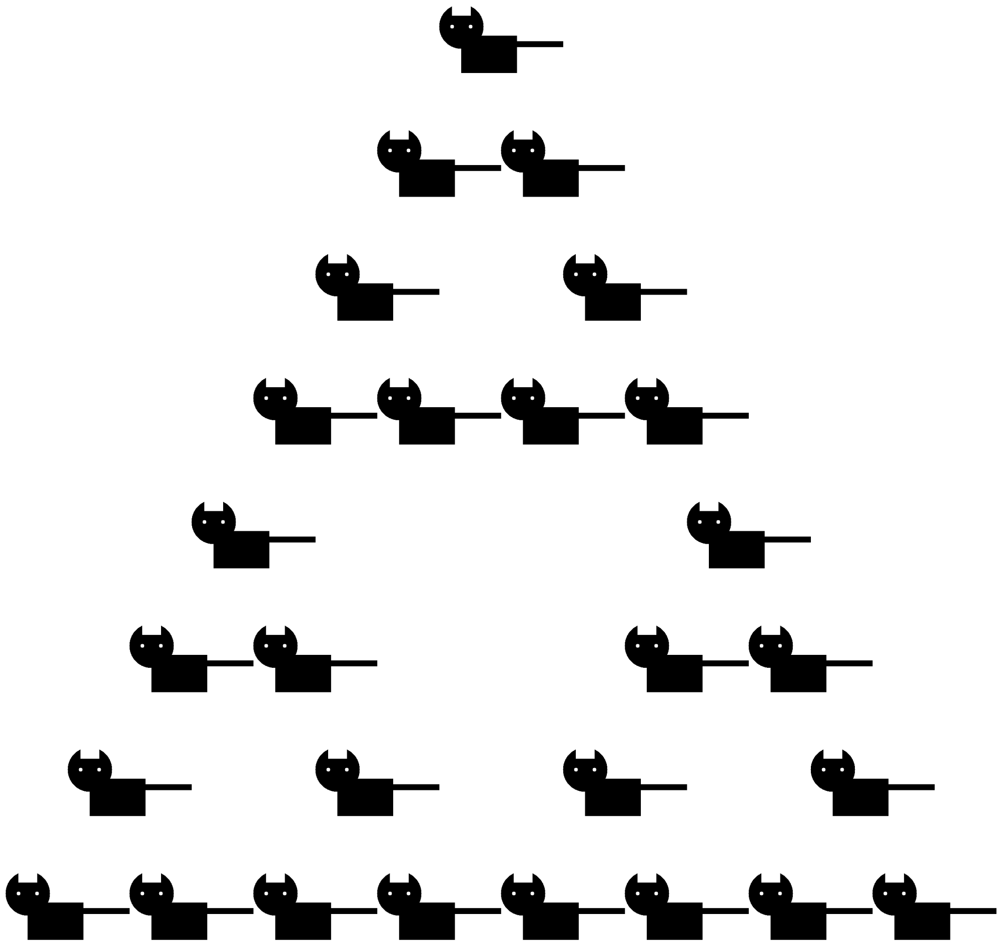
In lecture, we discussed a representation of shapes as a datatype `shape`, which can be used for generating fractals. For example, the n^{th} Sierpinski triangle is formed by adjoining three copies of the $(n-1)^{th}$ Sierpinksi triangle. Suppose you have

- `val cat : shape`, which displays as

- `val sierp_step : shape -> shape`, which adjoins three copies of a shape. For example `sierp_step cat` displays as



(c) (3 points) Write a function `catpinski : int -> shape` that computes the n^{th} Sierpinski triangle starting from `cat`. For example, `catpinski 3` displays as



Your solution must not be recursive.

```
val catpinski : int -> shape =
```

Solution:

```
iter sierp_step cat
```

THIS PAGE IS INTENTIONALLY LEFT BLANK