15-150 Fall 2013
Midterm Practice Questions

Midterm on 17 October 2013

Disclaimer : This is not a past midterm, but is rather a collection of questions meant to give a sampler of possible midterm questions. There will be no solutions posted for these questions, but feel free to ask the TAs and other students questions about them.

1 GCD

Below is a `GCD` function for computing the greatest common divisor of two non-negative numbers. The g.c.d. of `m` and `n` is the largest integer that divides both `m` and `n`, with zero remainder.

```
(* GCD: int * int -> int *)
(* REQUIRES m, n >= 0    *)
(* ENSURES GCD (m, n) evaluates to the g.c.d. of m and n *)

fun GCD (m: int, 0): int = m
  | GCD (0, n: int): int = n
  | GCD (m: int, n: int): int =
        if m > n
        then GCD(m mod n, n)
        else GCD(m, n mod m)
```

Task 1.1 Let's prove that the code meets the specification, using complete induction on the product `mn`.
You may use the following lemmas in your proof as facts, but be sure to cite them where you do.

Lemma 1: `m mod n = m - (m div n) * n` for all natural numbers m and n.

Lemma 2: The g.c.d of m and n is the same as the g.c.d of m mod n and n, where $m > n$

Theorem:

Proof: By ___________________ (method) on _____________________ (expression)

Base Case:

NTS (Need to Show):

Inductive Step:

IH (Inductive Hypothesis):

NTS (Need to Show):

2 More Polynomials

Recall from lab 07 that we can represent a polynomial

$$\sum_{i=0}^{n} = c_0 + c_1 x + c_2 x^2 + \ldots + c_n x^n$$

as a function that maps a natural number, i, to the coefficient c_i of x^i, up to the degree n of the polynomial (i.e., its largest non-zero coefficient).

Thus, we define the type `poly` to represent polynomials as follows:

```
type poly = (int -> real)
```

The function part is a mapping from exponent (integers) to coefficient (reals). Table 1 shows two examples.

polynomial	representation
$5x^2 + 9x^1 + 0x^0$	`val p1 = (fn 2 => 5.0 | 1 => 9.0 | 0 => 0.0 | _ => 0.0)`
$9x^0$	`val p2 = (fn 0 => 9.0 | _ => 0.0)`

Table 1: Polynomials as functions

As an example of how to define operations on polynomials represented in this manner, here is the definition of addition of polynomials:

```
fun add (p1 : poly, p2 : poly) : poly =
     (fn x => p1 x + p2 x)
```

We now look at some interesting uses of polynomials, made possible by our old friend, Calculus.

Recall from calculus that the derivative of a polynomial of degree n is defined as follows:

$$\frac{d}{dx}\sum_{i=0}^{n} c_i x^i \quad = \quad \sum_{i=1}^{n} i c_i x^{i-1}$$

Task 2.1 Define the function

```
differentiate : poly -> poly
```

that computes the derivative of a polynomial. Note that `differentiate` should *not* be recursive.

Recall from calculus that integration of polynomials is defined as follows:

$$\int \sum_{i=0}^{n} c_i x^i \, dx = C + \sum_{i=0}^{n} \frac{c_i}{i+1} x^{i+1}$$

where C is an arbitrary constant known as the constant of integration. As C can be any number, the result of integration is a family of polynomials, one for each choice of C. Therefore, we will represent the result of integration as a function of type `real -> poly`.

Task 2.2 Define the function

```
integrate : poly -> (real -> poly)
```

that computes the family of polynomials corresponding to the integral of the argument polynomial. Note that `integrate` should *not* be recursive.

3 Halloween Functions

Adam, the master of all things spooky, decided one day to build a haunted house. But naturally, before he could begin construction, he had to plan how to maximize the spookiness of his haunted house. To do this, he fiddled with different methods until finally he found a function that accurately modeled spookiness as a function of the number of surprises in his haunted house! By using a patented value `spookyNum`, he wrote the following:

```
val spookyNum = (* Trade secret *)

fun spookiness 0 = spookyNum
  | spookiness 1 = spookyNum
  | spookiness (surprises: int): int =
      let
        val half = surprises div 2
      in
        (spookyNum * (spookiness (surprises - half) + spookiness half)) div 2
      end
```

Adam used two recursive calls to increase the precision of his function. However, he is concerned that by doing this he is making his SML function run too slow, so he called in a functional programming expert — that's you — to analyze his code.

Task 3.1 Determine the work and the span of `spookiness` (n) as a function of n. Write the recurrence relations for both and give tight big-O bounds for them.

Suddenly, Adam jumped out of his seat and started drawing formulas and spooky ghosts on a nearby whiteboard. He finally writes a new function in SML and circles it, saying that it gives a much more precise answer than his previous ones, because it factors in the mysteriousness of his haunted house by calling a helper function `mystery`.

```
fun mystery (s: int) (n: int): int = (* Trade secret *)

val spookyNum: int = (* Trade secret *)

fun spookiness2 0 = spookyNum
  | spookiness2 1 = spookyNum
  | spookiness2 (surprises: int): int =
      let
        val half = surprises div 2
      in
        mystery (spookiness2 half + spookiness2 (surprises - half)) surprises
      end
```

Task 3.2 Give the recurrences for the work and span of `spookiness2` n in terms of n. You may assume `mystery` s n is $O(n)$ for both work and span.

Task 3.3 Give tight big-O bounds for `spookiness2 n`.

4 Trees

Imagine a tree with one kind of data at its internal nodes and lists of another kind of data at its leaves. This idea is formalized by the datatype

```
datatype ('data,'dir) tree =
         Empty
       | Leaf of 'data list
       | Node of ('data,'dir) tree * 'dir * ('data,'dir) tree
```

where `'data` and `'dir` are both type variables, so this is a tree parameterized over two things.

One thing you can do with values of this type is have the `'dir`s in the internal nodes form a sorted structure that directs you to data stored at the leaves. For example, the tree in Figure 1 has integers in the internal nodes, guiding you to lists of key-value pairs at the leaves

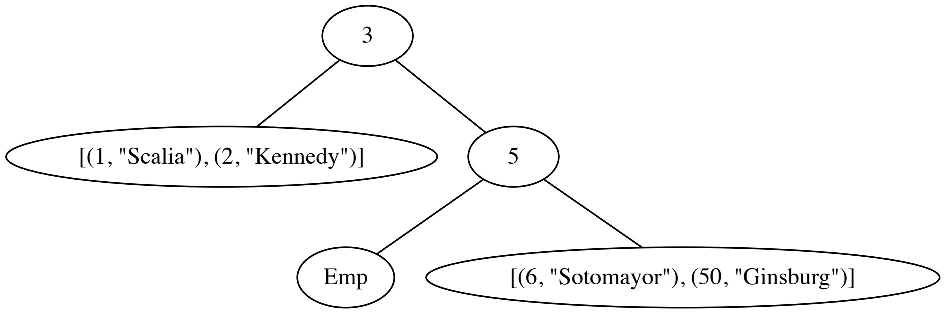
3
[(1, "Scalia"), (2, "Kennedy")]
5
Emp
[(6, "Sotomayor"), (50, "Ginsburg")]

Figure 1: A simple database

It is represented by the value

```
val db : (int * string, int) tree =
  Node(Leaf [(1, "Scalia"), (2, "Kennedy")], 3,
     Node(Empty, 5,
              Leaf [(6, "Sotomayor"), (50, "Ginsburg")]))
```

This amounts to picking `'data` to be the key-value pairs with type `int * string` and `'dir` to be `int`. We need two definitions to make this idea of internally-sorted trees of lists of key-value pairs formal enough to use:

- A key is said to be in a tree under the following conditions:
  - No key is in the tree `Empty`

  - A key `k` is in the tree `Leaf l` if the pair `(k,x)` appears in `l` for any `x`.
  - A key is in the tree `Node(l,tag,r)` if it's in `l` or if it's in `r`.

- A tree is said to be sorted with respect to a function `cmp : 'key * 'dir -> order` under the following conditions:
  - `Empty` is sorted
  - `Leaf l` is sorted for any `l`
  - `Node(l,tag,r)` is sorted if and only if:
    - `l` is sorted with respect to `cmp`
    - `r` is sorted with respect to `cmp`
    - If a `k:'key` is in `t` and `cmp(k,tag)` = `LESS`, then `k` is in `l`. Otherwise, `k` is in `r`.

Task 4.1 Write a function `lookup` such that if `t` is sorted with respect to `cmp`, and `eq` is an equality function on keys, then for any `target:'key`,

```
(lookup eq cmp target t)
```

is `SOME x` if there is a leaf in `t` that contains `(target,x)` and `NONE` otherwise.

For example, in the above example tree,

```
lookup inteq Int.compare 6 db = SOME "Sotomayor"
lookup inteq Int.compare 0 db = NONE
```

```
fun lookup (eq : 'key * 'key -> bool) (cmp : 'key * 'dir -> order)
           (target : 'key) (t : ('key * 'val, 'dir) tree) : 'val option =
```