Midterm 2, CSCI 2041, Mock-2

There are two questions to this midterm, for a total of 100 points. You have 50 minutes to complete it. You are allowed one single-sided page of notes in your own handwriting with this midterm.

Recall that you are not to discuss midterms, even after you completed it.

Name: Instructor's solutions

1. Mirror mirror (20+20 points)

Consider the following ‘arithmetic expression’ data-type and corresponding function mirror that mirrors it:

```
type expr
  = Int of int
  | Plus of t * t
let rec mirror t = match t with
  | Int n -> Int n
  | Plus (l, r) -> Plus (mirror r, mirror l)
```

You will be asked to prove that mirror is its own inverse, using induction.
You may ***only*** use the following rules (and case distinction, induction, and helper-lemmas you prove using only these rules):

`mir-i`	`mirror (Int n) = (Int n)`
`mir-p`	`mirror (Plus (l,r))` `= Plus (mirror l, mirror r)`

Prove that: mirror (mirror x) = x
Use induction on x.

Write the non-recursive case (base case) here:

case x = Int i

mirror (mirror x)
= {case}
mirror (mirror (Int i))
= {mir-i}
mirror (Int i)
= {mir-i}
(Int i)
= {case}
x

Write the inductive case here (specify the IH(s) and variables used clearly):

case x = Plus (l, r)
IH1: mirror (mirror l) = l
IH2: mirror (mirror r) = r

mirror (mirror x)
= {case}
mirror (mirror (Plus (l,r)))
= {mir-p}
mirror (Plus (mirror r,mirror l))
= {mir-p}
Plus (mirror (mirror l),mirror (mirror r))
= {IH1}
Plus (l,mirror (mirror r))
= {IH2}
Plus (l,r)
= {case}
x

2. Functors (20 + 20 + 20)

Consider the following module type with comments,
a functor and two helper functions. Assume Inverse is a module that satisfies the properties as specified in the signature *Involution*:

```
module type Involution = sig
  type t
  (** Requires that inv is its own inverse:
  [inv (inv x) = x] **)
  val inv : t -> t
end

module MapInverse (Inverse : Involution) = struct
  type t = Inverse.t list
  let rec map lst = match lst with
    | [] -> []
    | x::xs -> (Inverse.inv x)::(map xs)
  (* let inv = map *)
end

let rec append xs ys = match xs with
  | [] -> ys
  | x::xs -> x::(append xs ys)
let rec rev xs = match xs with
  | [] -> []
  | x::xs -> append (rev xs) [x]
```

You may assume the following helper lemma:
map (append a b) = append (map a) (map b)
Use the following rules for your proofs:

`helper`	`map (append a b) = append (map a) (map b)`
`map-n`	`map [] = []`
`map-t`	`map (h::tl) = Inverse.inv h :: map tl`
`ap-n`	`append [] lst = lst`
`ap-t`	`append (h::tl) lst = h::append tl lst`
`rev-n`	`rev [] = []`
`rev-t`	`rev (h::tl) = append (rev tl) (h::[])`

2a. Prove that:
map (rev lst) = rev (map lst)

use induction on lst. Write the base case here.

case lst = []

map (rev lst)
= {case}
map (rev [])
= {rev-n}
map []
= {map-n}
[]
= {rev-n}
rev []
= {map-n}
rev (map [])
= {case}
rev (map lst)

2b. proceed with the inductive case here:

case lst = h::tl
IH: map (rev tl) = rev (map tl)

map (rev lst)
= {case}
map (rev (h::tl))
= {rev-c}
map (append (rev tl) (h::[]))
= {helper}
append (map (rev tl)) (map (h::[]))
= {IH}
append (rev (map tl)) (map (h::[]))
= {map-c}
append (rev (map tl)) (Inverse.inv h::map [])
= {map-n}
append (rev (map tl)) (Inverse.inv h::[])
= {rev-c}
rev (Inverse.inv h::map tl)
= {map-c}
rev (map (h::tl))
= {case}
rev (map lst)

2c. Prove that the identity function on integers given by:

```
let id (x : int) : int = x
```

constitutes a valid instance for Involution. Write the instance and prove any required properties.

```
(* the name Identity is made up, you can choose a different name *)
module Identity = struct
  type t = int
  let inv = id
end
```

a proof that inv(inv x) = x

```
Identity.inv (Identity.inv x)
= {inv-def}
id (Identity.inv x)
= {inv-def}
id (id x)
= {id-def}
id x
= {id-def}
x
```

If you're looking to do more proofs:

- the helper lemma from this exercise can be proven with a simple induction proof
- defining 'let inv = map' would turn Map into an involution (provided that its argument is one). Proving it requires simple induction
- we've seen a proof that: rev (rev x) = x
  (during lectures). Given the previous, you might be tempted to think that 'revmap x = rev (map x)' is an involution too. You'd be right, and there's an inductive proof for it. There's also a direct proof (by which I mean: one without induction) that uses only properties mentioned in this file: it requires a property that is specific to 'rev' and 'map'
- if you define 'map' on trees (say: treemap) then mirror behaves like 'rev': 'treemap (mirror x) = mirror (treemap x)'