CSci 2041: Advanced Programming Principles

Spring 2018

Homework 3, due at 5:00pm on Wednesday March 7, 2018, via GitHub.
(Submission instructions are given at the end of this document.)

Question 1: Power function, over natural numbers
Recall the OCaml `power` function over natural numbers, shown below:

```
let rec power n x =
  match n with
  | 0 -> 1.0
  | _ -> x *. power (n-1) x
```

Using induction over natural numbers show that

$$\texttt{power n x} = x^n.$$

Your proof must explicitly and clearly indicate the base case you prove, the inductive case you prove and what the inductive hypothesis provides in the proof.
Each step in your proof must be accompanied by a justification describing why that step could be taken.

Question 2: Power over structured numbers
Recall from lecture the OCaml type `nat`, the function `toInt`, and the `power` function working over this representation of natural numbers:

```
type nat = Zero | Succ of nat

let toInt = function
  | Zero -> 0
  | Succ n -> toInt n + 1

let rec power n x =  match n with
  | Zero -> 1.0
  | Succ n'-> x *. power n' x
```

What is the principle of induction for the type `nat`?

Using induction over `nat` values show that

$$\texttt{power n x} = x^{\texttt{toInt}(n)}$$

Your proof must explicitly and clearly indicate the base case you prove, the inductive case you prove and what the inductive hypothesis provides in the proof.
Each step in your proof must be accompanied by a justification describing why that step could be taken.

Question 3: List reverse and append
Consider the following definition of `append`:

```
let rec reverse l = match l with
  | [] -> []
  | (h::t) -> reverse t @ [h]

let rec append l1 l2 = match l1 with
  | [] -> l2
  | (h::t) -> h :: (append t l2)
```

Using the definition of `reverse` and the definition of `append` above, show, using induction, that

```
reverse (append l1 l2) = append (reverse l2) (reverse l1)
```

Your proof must explicitly and clearly indicate the base case you prove, the inductive case you prove and what the inductive hypothesis provides in the proof.
Each step in your proof must be accompanied by a justification describing why that step could be taken.

Question 4: List processing
Consider the following OCaml function definitions.

```
let isupper c = Char.code c >= Char.code 'A' &&
                Char.code c <= Char.code 'Z'
let rec someupper lst = match lst with
  | [] -> false
  | x::xs -> isupper x || someupper xs
```

Using the definition above, show using induction that

```
someupper (l1 @ l2) = someupper l1 || someupper l2
```

Your proof must explicitly and clearly indicate the base case you prove, the inductive case you prove and what the inductive hypothesis provides in the proof.
Each step in your proof must be accompanied by a justification describing why that step could be taken.

Question 5: List processing and folds Below we show again the functions defined in the previous problem and a new alternative implementation of the recursive function defined in Question 4. This new version uses `foldr`. The definition of `foldr` is the same as you've seen before.

```
let isupper c = Char.code c >= Char.code 'A' &&
                Char.code c <= Char.code 'Z'
let rec someupper lst = match lst with
  | [] -> false
  | x::xs -> isupper x || someupper xs

let rec foldr (f:'a -> 'b -> 'b) (l:'a list) (v:'b) : 'b =
  match l with
  | [] -> v
  | x::xs -> f x (foldr f xs v)

let upperor c b = isupper c || b
let foldupper lst = foldr upperor lst false
```

By induction, show that

`someupper chs = foldupper chs`

Your proof must explicitly and clearly indicate the base case you prove, the inductive case you prove and what the inductive hypothesis provides in the proof.
Each step in your proof must be accompanied by a justification describing why that step could be taken.

Question 6: Tree processing

Consider the following OCaml definitions of a tree type and a few functions over values of this type.

```
type 'a tree = Leaf of 'a
             | Branch of 'a tree * 'a tree
let min x y = if x < y then x else y

let rec mintree t = match t with
  | Leaf v -> v
  | Branch (t1, t2) -> min (mintree t1) (mintree t2)

let rec tfold (l:'a -> 'b) (f: 'b -> 'b -> 'b) (t: 'a tree) : 'b = match t with
  | Leaf v -> l v
  | Branch (t1, t2) -> f (tfold l f t1) (tfold l f t2)

let fold_mintree t = tfold (fun x -> x) min t
```

Prove using induction that for any tree `t` of type `int tree`

`mintree t = fold_mintree t`

Your proof must explicitly and clearly indicate the base case you prove, the inductive case you prove and what the inductive hypothesis provides in the proof.
Each step in your proof must be accompanied by a justification describing why that step could be taken.

Submission instructions: Writing proofs such as these requires a bit of clear thinking and it is important to check your work.

Checking your work means you need to be able to read it. And to assess it, we need to be able to read it as well.

Thus, we are requiring your solutions be electronically generated. You may turn your work in using any of the following forms.

1. A PDF file - named `hwk_03.pdf`.

   You may use LaTeX, enscript, or even MS Word to generate a PDF file that contains your solutions.

2. A Markdown file - named `hwk_03.md`.

   This is used for the lab and other homework specifications and makes it easy to see your solution in GitHub.

3. A text file - named `hwk_03.txt`.

   We've written proofs in text files in class and examples can be found in the `Notes` directory of the public class repository.

Scanned or photographed versions of hand-written solutions will not be accepted.
This work is to be submitted via GitHub in a folder named `Hwk_03`.
The work is due by 5:00pm on Wednesday, March 7.