Midterm Exam

CAS CS 320: Principles of Programming Languages

February 27, 2024

Name:

BUID:

Location:

- You will have approximately 75 minutes to complete this exam.
- Make sure to read every question, some are easier than others.
- You may not use any functions from the OCaml standard library except for constructors and operators, unless otherwise specified.
- You must write your answers as neatly as possible in the specified boxes. You will be evaluated only on what appears in the boxes. We reserve the right to dock points for illegible or unclear answers.
- There are extra pages at the end of the exam for scratch work. Please do not remove them.
- Please write your name and BUID **on every page**.

(Extra page)

1 Suffixes

Implement the function `suffixes` which, given a list `l`, returns the list of all suffixes of `l` in decreasing order of length (`suf` is a suffix of a list `l` if there is a list `pre` such that `pre @ suf = l`). Write your final answer in the box below.

```
let rec suffixes (l : 'a list) : 'a list list =
      match l with
      | [] -> [[]]
      | x :: xs -> (x :: xs) :: suffixes xs
```

```
let _ =
assert(suffixes [1;2;3] = [[1;2;3]; [2;3]; [3]; []])
```

(Extra page)

2 Evaluation

Consider the following OCaml function.

```
let f =
  let rec g lst =
    match lst with
    | 1 :: 1 :: xs -> h xs
    | [] -> true
    | _ -> false
  and h lst =
    match lst with
    | 0 :: xs -> k xs
    | _ -> false
  and k lst = g lst || h lst
  in k
```

A. What is the type of `f`? Write your final answer in the box below.

int list → bool

B. Evaluate the following expressions. Write your final answers in the boxes below.

`f []` = true

`f [1]` = false

`f [1;1;0]` = true

`f [1;1;1;0;0]` = false

`f [1;1;1;0;0;1;1;0]` = false

(Extra page)

3 Fixpoints

Implement the function `fix` which, given

```
f : 'a -> 'a
```

returns a function of type `'a -> 'a` given by

$$f^{\infty}(x) = \begin{cases} f^n(x) & \text{for any } n \text{ s.t. } f^n(x) = f^{n+1}(x) \\ \text{undefined} & \text{otherwise} \end{cases}$$

In other words, `fix f x` is the result of applying `f` to `x` until the value doesn't change, and it runs forever if this never occurs. **Your solution must be tail-recursive.** Write your final answer in the box below.

```
let rec fix (f : 'a -> 'a) : 'a -> 'a =
  let rec go x =
       if x = f x
       then x
       else go (f x)
     in go
```

```
let f x = if x * x > 100 then x else x * x
let _ = assert (fix f 3 = 81)
```

(Extra page)

4 Transpose

Fill in the following implementation of the function `transpose` which, given

```
m :  'a list list
```

returns the transpose of `m`, assuming `m` represents the rows of a valid $m \times n$ matrix, where m and n are positive. Recall that `hd` (used in the first line of `transpose`) returns the first element of a list and `tl` returns everything except the first element. You may use these functions in your solution. Write your final answers in the boxes below.

```
open List

let f1 = ...
let f2 = ...

let rec transpose m =
  if is_empty (hd m)
  then []
  else map f1 m :: transpose (map f2 m)
```

```
let f1 = hd
```

```
let f2 = tl
```

(Extra page)

5 Filter Any

Implement the function `filter_any` which, given

```
f :  'a -> 'b -> bool

l :  'a list

r :  'b list
```

returns the members `x` of `r`—in order—for which there is *some* member `a` of `l` such that `f a x` is `true`. You may use the functions `List.fold_left`, `List.fold_right`, `List.filter`, and `List.map`. Write your final answer in the box below.

```
let filter_any f l r =
  let rec p l x =
     match l with
     | [] -> false
     | a :: as -> f a x || p as x
  in
  List.filter (p l) r
```

```
let l = [1;2;3;4;5;6;7]
let _ = assert (filter_any (>) [4;2] l = [1;2;3])
let _ = assert (filter_any (<) [4;2] l = [3;4;5;6;7])
let _ = assert (filter_any (=) [4;2] [1;2;3] = [2])
```

(Extra page)

6 Game Player

Consider the following small interface for a player in a video game.

```
let greet (p : player) = "Hello " ^ p.name ^ "!"

let level_up (p : player) =
  { p with level = p.level + 1 }

let take_potion (p : player) =
  match p.curr_item with
  | Potion info ->
    { p with
      health = p.health + info.boost ;
      curr_item = List.hd p.inventory ;
      inventory = List.tl p.inventory ;
    }
  | _ -> p

let value (i : item) =
  match i with
  | Potion { boost = n } -> n * 2
  | Sword -> 100
  | Map -> 0
```

Define the types `item` and `player` so that the following code type-checks. Write your final answers in the boxes below.

```
type item =
   Potion of { boost : int }
  | Sword
  | Map
```

```
type player = {
   level : int ;        inventory : item list ;
   name : string ;      curr_item : item ;
   health : int ;  }
```

(Extra page)

7 Double Lists

A `doublelist` is a data type which represents a list, but elements are '`Cons`'ed two at a time. In order to represent lists with an odd number of elements, there is also a constructor for a singleton list.

```
type 'a doublelist
  = Nil
  | Single of 'a
  | Cons of 'a * 'a * 'a doublelist
```

Implement the function `rev`, which reverses a `doublelist`. Write your final answer in the box below. **Your solution must be linear-time.** (**Extra Credit.** Give a tail-recursive implementation.)

```
let rec rev (l : 'a doublelist) : 'a doublelist =
    let go acc l
      match l with
      | Nil -> acc
      | Single x -> x :: acc
      | Cons (x, y, ys) -> go (y :: x :: acc) ys
    in
    let rec to_dl l =
      match l with
      | [ ] -> Nil
      | [x] -> Single x
      | x :: y :: ys -> Cons (x, y, to_dl ys)
    in to_dl (go [ ] l)
```

```
let _ = assert
 (rev (Cons (1, 2, Nil)) = Cons (2, 1, Nil))
let _ = assert
 (rev (Cons (1, 2, Single 3)) = Cons (3, 2, Single 1))
```

(Extra page)

Extra Credit:

```
let rev l =
   let rec go acc1 a acc2 l =
      match l with
      | Nil -> acc1
      | Single x -> Cons (x, a, acc2)
      | Cons (x, y, ys) ->
            go (Cons (y, x, acc1)) y
               (Cons (x, a, acc2)) ys
   in match l with
      | Nil -> Nil
      | Single x -> Single x
      | Cons (x, y, ys) ->
            go (Cons (y, x, Nil) y
               (Single x)   ys
```

(Extra page) DO NOT REMOVE

(Extra page) DO NOT REMOVE