Practice Midterm

CAS CS 320: Principles of Programming Languages February 21, 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.
- Please write your name and BUID on every page.

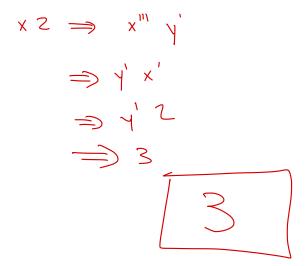
1 Shadowing

Consider the following definition.

```
let x y =
  let x' = y in
  let y' x" = x" + 1 in
  let x" y" = y" x' in
```

A. What is the the type of x?

B. What is the value of x = 2?



2 Number of Digits

Implement the function num_digits which, given an integer n, returns the number of digits in n. You may only use arithmetic operations.

let num_digits (n : int) : int = let rec go n = match n with 10 = 0<math>|n = 1 + go(n/10)if n = 0 then

1
else if n < 0 then

go (-n)

else go n

```
let _ = assert (num_digits 12345 = 5)
let _ = assert (num_digits -10 = 2)
let _ = assert (num_digits 0 = 1)
```

3 Records and Variants

Write down the type animal so that the following function type-checks.

```
let get_name_and_age (a : animal) (year : int) =
 match a with
 | Cow { name = n ; age = i } -> (Some n, Some i)
 | Chicken info ->
   (Some info.name, Some (year - info.birth_year))
  | Pig age -> (None, Some age)
 | Goose -> (None, None)
type animal =
   Cow of &
        name: string;
           age: int;
    Chicken of &
name: string;
birth year: int;
```

4 List Expressions

Circle the list expressions which type-check in OCaml.

(1 :: 2 :: []) :: (3 :: [4])

B. ((1 :: 2) :: []) :: (3 :: [])

C. 1 :: 2 :: [3] :: 4 :: [5]

D.)1 :: 2 :: [3] @ (4 :: [])

E. (1 :: 2) :: [3] @ (4 :: [])

5 Bitonic Sequences

A finite sequence of integers is **bitonic** if it is monotonically increasing, monotonically decreasing, or monotonically increasing and then monotonically decreasing. That is, given s_1, s_2, \ldots, s_n , either $s_1 < \cdots < s_n$ or $s_1 > \cdots > s_n$ or there is an index i such that

$$s_1 < \dots s_{i-1} < s_i > s_{i+1} \dots > s_n$$

Implement the function bitonic which, given a list of integers, returns true if it is bitonic, and false otherwise. Your solution should be self-contained (you may write helper functions as local definitions).

let bitonic (1: int list): bool =

let rec down l =

match l with

| x :: y :: ys - 5 x > y lk down (y::ys)

| - > + true

let rec go l =

match l with

| x :: y :: ys ->

x < y lk (go (y::ys) || down (y::ys))

x < y lk (go (y::ys) || down (y::ys))

| - > + true

in go l ll down l

```
let _ = assert (bitonic [1;2;3;2;1] = true)
let _ = assert (bitonic [1;2;3] = true)
let _ = assert (bitonic [3;2;1;2] = false)
let _ = assert (bitonic [] = true)
let _ = assert (bitonic [1;1] = false)
let _ = assert (bitonic [1;2;1;2] = false)
```

6 Evaluation

Consider the following pair of functions.

```
let rec foo l =
  match l with
  | [] -> []
  | false :: bs ->
    List.map (fun x -> x - 1) (0 :: foo bs)
  | true :: bs -> bar l
and bar l =
  match l with
  | [] -> []
  | false :: bs -> foo l
  | true :: bs -> List.map ((+) 1) (0 :: bar bs)
```

A. What is the type of foo?

foo: bool list - int list

B. What is the value of the following expression?

foo [true;true;false;false;false;false;true;true;false]

(1; 2; 3; 2; 1; 0; -1; 0; 1; 0]

Function Maximum

Implement the function func_max which, given fs, a list of functions from int to int, returns a function from int to int which is given by

$$\mathsf{funcMax}(x) = \max(\max_{f \in \mathtt{fs}} \{f(x)\}, 0)$$

You should accomplish this by a single call to fold.

```
let op accum next = ...
let base n = \dots
let func_max (fs : (int -> int) list) : int -> int =
  List.fold_left op base fs
let _ = assert
  (func_max [(+) 1; fun x -> x * x] 1 = 2)
let _ = assert
  (func_max [(+) 1; fun x -> x * x] (-2) = 4)
```

let op accum next =

fun n ->
max (accum n) (next n)

8 Tail Recursion

Without using any functions from the standard library, implement a tail-recursive version of rev_concat, the function which, given a list of lists, concatenates them in reverse order.

rev_concat (ls: 'a list list): 'a list =

let rec go ls acc =

match ls with

[[] - acc

[[] :: lss - s go lss acc

[[] (x :: xs) :: lss - > go (xs:: lss)

[[] (x :: acc)