Name : ________________________________________ (*please print clearly!*)

# CSE 341 Winter 2018 Midterm **SOLUTION**

**Please do not turn the page until 12:30.**

Rules:

- The exam is closed-book, closed-note, etc. except *one side* of a 8.5x11in page.
- **Please stop promptly at 1:20.**
- There are **100 points**, distributed **evenly** among **5** multi-part questions.
- ***QUESTIONS VARY GREATLY IN DIFFICULTY. GET EASY POINTS FIRST!!!***
- The exam is printed double-sided, with pages numbered up to 17.

Advice:

- Read the questions carefully. Understand before you answer.
- Write down thoughts and intermediate steps so we can give partial credit.
- **Clearly indicate your final answer.**
- Questions are not in order of difficulty. **Answer everything.**
- If you have questions, ask.
- Relax. You are here to learn.

Name : ________________________________________ (*please print clearly!*)

**QUESTION 1 (20 points)**

(a) Consider the type `pos` and conversions from `int` to `pos`:

```
(* how "option" is defined in SML, just here for reference *)
datatype 'a option =
    NONE
  | SOME of 'a

datatype pos =
    One
  | S of pos

fun pos_of_int i =
  if i <= 0 then NONE
  else if i = 1 then SOME One
  else case pos_of_int (i - 1) of
              NONE => NONE
            | SOME p => SOME (S p)
```

What is the type of `pos_of_int` ? **`int -> pos option`**

What does `(pos_of_int ~1)` evaluate to ? **`NONE`**

What does `(pos_of_int 3)` evaluate to ? **`SOME (S (S One))`**

`pos_of_int` is tail recursive : T / **F**

Name : ___________________________________________ (*please print clearly!*)

(b) Consider this candidate for an "inverse" of `pos_of_int`, `int_of_pos` :

```
fun int_of_pos p =
  case p of
      One => 1
    | S p' => 1 + int_of_pos p'
```

What is the type of `int_of_pos` ? **`pos -> int`**

`int_of_pos` is tail recursive : T / **F**

Name : ______________________________________________ (*please print clearly!*)

(c) Consider this alternative version of `pos_of_int` :

```
fun pos_of_int' i =
  let fun loop acc i =
    if i = 1
    then acc
    else loop (S acc) (i - 1)
  in
    if i <= 0 then NONE
    else SOME (loop One i)
  end
```

What is the type of `pos_of_int'` ? **`int -> pos option`**

`pos_of_int'` is tail recursive : **T** / F

Is it true that, for all integer arguments `x`, `pos_of_int x = pos_of_int' x` ?
If so, simply write "Yes" in the blank. If not, please provide an input that causes the two functions to produce different results.

**Yes**

Name : ________________________________________ (*please print clearly!*)

(d) Consider one more version of `pos_of_int` :

```
exception NonPos

fun pos_of_int'' i =
  if i <= 0 then raise NonPos
  else if i = 1 then One
  else S (pos_of_int'' (i - 1))
```

What is the type of `pos_of_int''` ? **`int -> pos`**

`pos_of_int''` is tail recursive : T / **F**

Is it true that, for all integer arguments `x`, `pos_of_int x = pos_of_int'' x` ? If so, simply write "Yes" in the blank. If not, please provide an input that causes the two functions to produce different results.

**1 (any input would work, return types are different)**

Name : ______________________________________________ (*please print clearly!*)

**QUESTION 2 (20 points)**

(a) Consider the `return` function:

```
fun return x =
  SOME x
```

What is the type of `return` ? **'a -> 'a option**

**Caveat**: For the next two blanks, ignore the value restriction (that was the weird rule about not generalizing types if an expression is not a "syntactic value" -- just assume we can safely generalize types in SML for purposes of answering these).

*P.S. If the caveat above makes you feel uncomfortable, don't worry! You are doing great and the value restriction is just a weird thing that we're ignoring here. In fact, you should just imagine I didn't say anything at all about it if you can't quite remember what it is right now. I promise you don't need to understand it AT ALL to get these right :)*

What does `(return NONE)` evaluate to ? **`SOME NONE`**

What is the type of `(return NONE)` ? **`'a option option`**

Name : ____________________________________________ (*please print clearly!*)

(b) This part refers to definitions from Question 1. Consider `bind` and `lift` :

```
fun bind x f =
  case x of
      NONE => NONE
    | SOME y => f y

fun lift f =
  fn x => return (f x)
```

What is the type of `bind` ? **`'a option -> ('a -> 'b option) -> 'b option`**

What is the type of `lift` ? **`('a -> 'b) -> 'a -> 'b option`**

What does `(bind (pos_of_int ~1) (lift int_of_pos))` evaluate to ?

**`NONE`**

What does `(bind (pos_of_int 3) (lift int_of_pos))` evaluate to ?

**`SOME 3`**

What is the type of `(fn x => bind (pos_of_int x) (lift int_of_pos))` ?

**`int -> int option`**

Name : ______________________________________________ (*please print clearly!*)

(c) Fill in the blanks with the type for each of the following functions.

```
fun flip f x y =
  f y x

fun get k s =
  s k
```

`flip:` **`('a -> 'b -> 'c) -> 'b -> 'a -> 'c`**

`get:` **`'a -> ('a -> 'b) -> 'b`**

(Note: The final page builds on this question for (OPTIONAL) extra credit!)

Name : ______________________________________________ (*please print clearly!*)

**QUESTION 3 (20 points)**

Consider these types:

```
datatype a = M

datatype b = P | Q

datatype c = CA of a
           | CB of b

datatype d = DA of d * a
           | DB of d * b

datatype e = EA of e * a
           | EB of b
```

How many distinct *values* are there of each type (e.g., “zero”, “one”, “two”, …, “infinity”)?

a : **one**

b : **two**

c : **three**

d : **zero**

e : **infinity**

Name : ________________________________________ (*please print clearly!*)

**QUESTION 4 (20 points)**

(a) Consider this function:

```
fun snoc (x, xs) =
  case xs of
      [] => [x]
    | x' :: xs' => x' :: snoc (x, xs')
```

Circle all the alternate definitions below which are equivalent to the one above:

```
fun snoc (x, xs) =
  List.rev (x :: xs)
```

```
fun snoc (x, xs) =
  x :: (List.rev xs)
```

```
fun snoc (x, xs) =
  List.rev (x :: (List.rev xs))
```

```
fun snoc (x, xs) =
  [x] @ List.rev xs
```

```
fun snoc (x, xs) =
  [xs] @ x
```

```
fun snoc (x, xs) =
  xs @ [x]
```

```
fun snoc (x, xs) =
  xs :: x
```

(b) For reference, here are some curried versions of "hall of fame" list functions we saw in lecture:

```
fun append xs ys =
  case xs of [] => ys
           | x :: xs' => x :: append xs' ys

fun map f xs =
  case xs of [] => []
           | x :: xs' => f x :: map f xs'

fun filter f xs =
  case xs of [] => []
           | x :: xs' => if f x
                         then x :: filter f xs'
                         else filter f xs'

fun fold f acc xs =
  case xs of [] => acc
           | x :: xs' => fold f (f acc x) xs'
```

**Which of the pairs of expressions on the *next page* are equivalent?**

In the left column for each row, please write "***Always***" if the expressions are always equivalent, "***Pure***" if the expressions are equivalent when `f` and `g` are pure (always terminate, never throw exceptions, never read or write references, etc.), or "***No***" if the expressions are not equivalent. Remember that `div` is used for integer division in SML.

The first three rows are filled out as examples. Please write answers clearly!

Name : ______________________________________________ (*please print clearly!*)

| Equiv? | | |
|---|---|---|
| ***Always*** | `x + y` | `y + x` |
| ***Pure*** | `f x + g y` | `g y + f x` |
| ***No*** | `x div y` | `y div x` |
| ***Always*** | `(fn x => f x) x` | `f x` |
| ***No*** | `(fn x y => f x y) x y` | `f x` |
| ***Always*** | `filter f (append xs ys)` | `append (filter f xs) (filter f ys)` |
| ***No*** | `map f` | `fold (fn acc x => f x :: acc) []` |
| ***Always*** | `map f` | `fold (fn acc x => acc @ [f x]) []` |
| ***Always*** | `map f (append xs ys)` | `append (map f xs) (map f ys)` |
| ***Pure*** | `map f (map g xs)` | `map (fn x => f (g x)) xs` |
| ***No*** | `filter f (map g xs)` | `map g (filter f xs)` |
| ***Pure*** | `filter f (filter g xs)` | `filter (fn x => f x andalso g x) xs` |

Name : ___________________________________________ (*please print clearly!*)

**QUESTION 5 (20 points)**

Consider this signature and module for polymorphic first-in-first-out (FIFO) queues:

```
signature QUEUE = sig
  type 'a t
  val empty : 'a t
  val push : 'a -> 'a t -> 'a t
  val pop : 'a t -> (('a * 'a t) option)
end

structure FastQueue :> QUEUE = struct

  type 'a t =
    'a list * 'a list

  val empty =
    ([], [])

  fun push a (xs, ys) =
    (xs, a :: ys)

  fun canon (xs, ys) =
    case xs of [] => (List.rev ys, [])
             | _ => (xs, ys)

  fun pop q =
    case (canon q) of ([], _) => NONE
                    | (x :: xs, ys) => SOME (x, (xs, ys))

end
```

Name : ______________________________________ (*please print clearly!*)

(a) Complete this alternate implementation of `QUEUE` based on lists so that it is equivalent to `FastQueue`:

```
structure ListQueue :> QUEUE = struct

  type 'a t = 'a list

  val empty = (* TODO *)
      []

  fun push a q = (* TODO *)
      q @ [a]

  fun pop q = (* TODO *)
      case q of [] => NONE
              | x :: xs => SOME (x, xs)

end
```

Name : ____________________________________________ (*please print clearly!*)

(b) What invariant does your implementation of `ListQueue` maintain?

**The oldest element is at the front of the list.**

(c) Why is it important that the type `t` for queues is held abstract?

**It enables different implementations of the module to use different types.**

(d) For which operations is your implementation of `ListQueue` slower on average than the corresponding operation in `FastQueue`?

**push**

Name : ______________________________________________ (*please print clearly!*)

***EXTREMELY OPTIONAL EXTRA CREDIT (2 points)***

Fill in the blanks with the type for the following functions. They depend on definitions from Question 2.

```
fun set k v s =
  fn k' => if k' = k
           then SOME v
           else s k'

fun wrap f s =
  fn k => bind (s k) f
```

set: **''a -> 'b -> (''a -> 'b option) -> ''a -> 'b option**

wrap: **('a -> 'b option) -> ('c -> 'a option) -> 'c -> 'b option**

Name : ____________________________________________ (*please print clearly!*)

***MORE EXTREMELY OPTIONAL EXTRA CREDIT (2 points)***

The code below uses functions defined earlier in the exam. It has a few subtle type errors. **Clearly circle two** of them and write a **brief comment** explaining why SML will not be able to type check the program at that point.

```
infix |>
fun x |> f = f x

fun fact s =
  bind (get "x" s) (fn x =>
   bind (get "ans" s) (fn ans =>
      if x < 1 then
        s (* need return here to make this an option *)
      else (
        s |> set "x" (x - 1)
          |> set "ans" (x * ans)
          |> fact)))

(* note: “print” has type string -> unit *)
fun print_var v s =
  s |> wrap Int.toString (* need lift here to work on state *)
    |> get v
    |> bind (lift print) (* need flip here to work in pipeline *)

val _ =
  (fn x => NONE)
    |> set "x" 5
    |> set "ans" 1
    |> fact
    |> flip bind (print_var "ans")
```