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.
QUESTION 1 (20 points)

(a) Consider the type \texttt{pos} and conversions from \texttt{int} to \texttt{pos}:

(* how “option” is defined in SML, just here for reference *)

datatype \texttt{`a option} =
  \texttt{NONE} \\
  | \texttt{SOME of `a}

datatype \texttt{pos} =
  \texttt{One} \\
  | \texttt{S of pos}

fun \texttt{pos_of_int} \texttt{i} =
  \texttt{if i} <= 0 \texttt{then} \texttt{NONE} \\
  \texttt{else if i} = 1 \texttt{then} \texttt{SOME One} \\
  \texttt{else case pos_of_int (i - 1) of} \\
  \texttt{NONE} \Rightarrow \texttt{NONE} \\
  | \texttt{SOME p} \Rightarrow \texttt{SOME (S p)}

What is the type of \texttt{pos_of_int} ? ________________________________

What does \texttt{(pos_of_int ~1)} evaluate to ? ___________________________

What does \texttt{(pos_of_int 3)} evaluate to ? ___________________________

\texttt{pos_of_int} is tail recursive : \texttt{T} / \texttt{F}
(b) Consider this candidate for an “inverse” of \( \text{pos} \_\text{of} \_\text{int} \), \( \text{int} \_\text{of} \_\text{pos} \):

\[
\text{fun} \ \text{int} \_\text{of} \_\text{pos} \ p =
\begin{align*}
\text{case} \ p \ \text{of} \\
\text{One} & \Rightarrow 1 \\
\mid S \ p' & \Rightarrow 1 + \text{int} \_\text{of} \_\text{pos} \ p'
\end{align*}
\]

What is the type of \( \text{int} \_\text{of} \_\text{pos} \)?

\( \text{int} \_\text{of} \_\text{pos} \) is tail recursive: \( T \) / \( F \)
(c) Consider this alternative version of pos_of_int:

```plaintext
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'?  ____________________________________

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

```haskell
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''`?  

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.

__________________________________________________________________
QUESTION 2 (20 points)

(a) Consider the \texttt{return} function:

\begin{verbatim}
fun return x = 
    SOME x
\end{verbatim}

What is the type of \texttt{return} ? \_______________________________

\textbf{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).

\textit{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 \texttt{(return NONE)} evaluate to? \_____________________________

What is the type of \texttt{(return NONE)}? \______________________________
(b) This part refers to definitions from Question 1. Consider `bind` and `lift`:

```haskell
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`? ___________________________________________

What is the type of `lift`? ___________________________________________

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

_____________________________________________________________________

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

_____________________________________________________________________

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

_____________________________________________________________________
(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: _______________________________________________________________

going: ______________________________________________________________

(Note: The final page builds on this question for (OPTIONAL) extra credit!)
QUESTION 3 (20 points)

Consider these types:

```plaintext
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 : __________________

b : __________________

c : __________________

d : __________________

e : __________________
QUESTION 4 (20 points)

(a) Consider this function:

```haskell
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:

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

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

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

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

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

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

```haskell
fun snoc (x, xs) =
    xs :: x
```
(b) For reference, here are some curried versions of “hall of fame” list functions we saw in lecture:

\[
\text{fun append } xs \ ys = \\
\quad \text{case } xs \text{ of } [] \Rightarrow ys \\
\quad \quad | \ x :: xs' \Rightarrow x :: \text{append } xs' \ ys
\]

\[
\text{fun map } f \ xs = \\
\quad \text{case } xs \text{ of } [] \Rightarrow [] \\
\quad \quad | \ x :: xs' \Rightarrow f x :: \text{map } f \ xs'
\]

\[
\text{fun filter } f \ xs = \\
\quad \text{case } xs \text{ of } [] \Rightarrow [] \\
\quad \quad | \ x :: xs' \Rightarrow \text{if } f x \\
\quad \quad \quad \quad \text{then } x :: \text{filter } f \ xs' \\
\quad \quad \quad \quad \text{else } \text{filter } f \ xs'
\]

\[
\text{fun fold } f \ acc \ xs = \\
\quad \text{case } xs \text{ of } [] \Rightarrow acc \\
\quad \quad | \ x :: xs' \Rightarrow \text{fold } f \ (f \ acc \ x) \ xs'
\]

**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 \( \text{div} \) is used for integer division in SML.

The first three rows are filled out as examples. Please write answers clearly!
| Equiv? | Always | Pure | No |
|--------|--------|------|----|---|
| Always | $x + y$ | $y + x$ | | | |
| Pure   | $f x + g y$ | $g y + f x$ | | | |
| No     | $\frac{x}{y}$ | $\frac{y}{x}$ | | | |
| (fn $x \Rightarrow f x$) $x$ | $f x$ | | | |
| (fn $x y \Rightarrow f x y$) $x y$ | $f x$ | | | |
| filter $f$ (append $xs y$s) | append (filter $f$ $xs$) (filter $f$ $ys$) | | | |
| map $f$ | fold (fn $acc \Rightarrow f x :: acc$) $[]$ | | | |
| map $f$ | fold (fn $acc \Rightarrow acc @ [f x]$) $[]$ | | | |
| map $f$ (append $xs y$s) | append (map $f$ $xs$) (map $f$ $ys$) | | | |
| map $f$ (map $g$ $xs$) | map (fn $x \Rightarrow f (g x)$) $xs$ | | | |
| filter $f$ (map $g$ $xs$) | map $g$ (filter $f$ $xs$) | | | |
| filter $f$ (filter $g$ $xs$) | filter (fn $x \Rightarrow f x$ andalso $g x$) $xs$ | | | |
QUESTION 5 (20 points)

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

```plaintext
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
```

13
(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 *)

  fun pop q = (* TODO *)

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

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

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

wrap: ________________________________________________________________
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.

```sml
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
      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
    |> get v
    |> bind (lift print)

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