Do not begin the exam until you are told to do so.

You have 50 minutes to complete the exam.

There are 100 total points.

There are 9 pages in this exam.

Make sure your name and Pennkey (a.k.a. username) is on the top of this page.
1. Program Design (25 points total)

An integer list \( l_1 \) is a subsequence of the integer list \( l_2 \) if all the elements of \( l_1 \) appear in the same order within the list \( l_2 \), perhaps with other elements between them. For example, \([1;2;2;3]\) is a subsequence of \([1;2;4;2;2;3]\). Use the four-step design methodology to implement a function called subsequence that determines whether one list is a subsequence of another.

(0 points) Step 1 is understanding the problem. You don’t have to write anything for this part—your answers below will demonstrate whether or not you succeeded with Step 1.

(3 points) Step 2 is formalizing the interface. Write down the type of the subsequence function as you might find it in a .mli file or module interface:

```ocaml
val subsequence : ...
```

(9 points) Step 3 is writing test cases. Complete the following tests with examples of the expected behavior. We have done the first one for you. Note that some test cases are better than others, and credit will be assigned accordingly: make sure your tests cover a sufficiently broad range of “interesting” inputs. Fill in the description string of the run_test function with a short explanation of why the test case is interesting.

i. let test () : bool =
   true = (subsequence [1;2;2;3] [1;2;4;2;2;3])
   ;; run_test "comes from the problem description" test

ii. let test () : bool =
    _______ = (subsequence ________________ ________________)
    ;; run_test "___________________________________________________" test

iii. let test () : bool =
    _______ = (subsequence ________________ ________________)
    ;; run_test "___________________________________________________" test

iv. let test () : bool =
    _______ = (subsequence ________________ ________________)
    ;; run_test "___________________________________________________" test
(13 points) Step 4 is implementing the program. Fill in the body of the `subsequence` function to complete the design. Do not use any list library functions (such as `List.map`, `fold`, or `@`) to solve this problem.

```ocaml
let rec subsequence (l1:___________) (l2:___________) : ____________ =
```

```ocaml
```
2. List Processing (20 points)

Recall the definition of \( \text{fold} \) we saw in class:

\[
\begin{align*}
\text{let rec } \text{fold} \ (\text{combine : } \text{’}a \rightarrow \text{’}b \rightarrow \text{’}b) \ (\text{base : } \text{’}b) \ (\text{l : } \text{’}a \ \text{list}) : \text{’}b &= \\
\begin{align*}
\text{begin } \\
&\text{match } \text{l} \\
&\quad\text{with} \\
&\quad\quad| \ [\] \rightarrow \text{base} \\
&\quad\quad| \ x::\text{tl} \rightarrow \text{combine } x \ (\text{fold } \text{combine } \text{base } \text{tl}) \\
\end{align*}
\end{align*}
\]

For each of the following programs, write the value computed for \( r \):

a. \textbf{let rec } h \ (l: \text{int list list}) : \text{int list} = \\
\begin{align*}
\text{begin } \text{match } l \text{ with} \\
&\quad| [] \rightarrow [] \\
&\quad| x::t \rightarrow x@\text{(h } t) \quad(\ast \text{ @ is the built-in } \text{’}append\text{’ function } \ast) \\
\text{end} \\
\text{let } r : \text{int list} = h [[1;2;3];[2];[3]]
\end{align*}

b. \textbf{let rec } g \ (f: \text{’}a \rightarrow \text{bool}) \ (l: \text{’}a \ \text{list}) : \text{’}a \ \text{list} = \\
\begin{align*}
\text{begin } \text{match } l \text{ with} \\
&\quad| [] \rightarrow [] \\
&\quad| x::t \rightarrow \text{if } (f x) \ \text{then } x::x::(g \ f \ t) \ \text{else } x::(g \ f \ t) \\
\text{end} \\
\text{let } r : \text{int list} = g (\text{fun } (x: \text{int}) : x > 2) [1;2;3;4;5]
\end{align*}

c. \textbf{let } \text{combine } (x: \text{int}) \ (y: \text{int}) : \text{int} = x + y \\
\text{let } r : \text{int} = \text{fold } \text{combine } 0 [1;2;3;4]

d. \textbf{let } m \ (l: \text{int list}) : \text{int option} = \\
\begin{align*}
\text{begin } \text{match } l \text{ with} \\
&\quad| [] \rightarrow \text{None} \\
&\quad| x::t \rightarrow \text{Some } (\text{fold } \text{min } x \ t) \\
\text{end} \\
\text{let } r : \text{int option} = m [3;2;1;4;5]
\end{align*}
3. Types (10 points)

For each OCaml expression below, write down its type or write “ill typed” if there is a type error. If an expression can have multiple types, give the most generic one. We have done the first one for you.

```
int 3 + 7

[1]::[]

[]::[1]

print_string

print_string::print_int::[]

fun (x:'a) -> x

(fun (x:'a) -> x) (3 + 7)

print_string "hello"; 3 + 7

let x : int = 3 in

let x : string = "hello" in

x

None

42 + (Some 120)
```
4. Binary Trees and Binary Search Trees (20 points)

Recall the definition of generic binary trees:

```ocaml
type 'a tree =  
  | Empty  
  | Node of 'a tree * 'a * 'a tree
```

a. (5 points) Circle the trees that satisfy the binary search tree invariant. (Note that we have elided the Empty nodes from these pictures.)

(a) [Diagram]
(b) [Diagram]
(c) [Diagram]
(d) [Diagram]
(e) [Diagram]

3 1 3 3 2
/ \ 
2 4 2 2 0 5
/ \ 
1 0 4 1 1
/ 
3

b. (8 points) Complete this definition of a function that returns the list of nodes of the given tree using an in order (left-to-right) tree traversal. For example, the in order traversal of tree (a) pictured above is [1;2;0;3;4]. You may use the @ (list append) operator.

```ocaml
let rec inorder (t:'a tree) : ______________ =  
  begin match t with  
    | Empty ->  
    | Node(lt, x, rt) -> 
  end
```

6
c. (8 points) Complete this definition of \texttt{tree\_transform}, which converts an \texttt{'a\ tree} to a \texttt{'b\ tree} by applying a given function \texttt{f} to each node of the tree while retaining the tree’s shape. (Note that this is analogous to the list \texttt{transform} function we saw in class.)

\begin{verbatim}
let rec tree_transform (f:'a -> 'b) (t: 'a tree) : 'b tree =
    begin match t with
        | Empty ->
        | Node(lt, x, rt) ->

    \end
\end{verbatim}

\d. (4 points) Suppose you know that \texttt{t} is an \texttt{int\ tree} that satisfies the binary search tree invariant. Which of the following properties must the function \texttt{f : int -> int} have so that \texttt{tree\_transform f t} also satisfies the BST invariant?

\begin{itemize}
    \item For every \texttt{x} and \texttt{y}, if \texttt{x <= y} then \texttt{(f x) <= (f y)}.
    \item For every \texttt{x} and \texttt{y}, if \texttt{x < y} then \texttt{(f x) < (f y)}.
    \item For every \texttt{x} and \texttt{y}, if \texttt{x <= y} then \texttt{(f y) <= (f x)}.
    \item For every \texttt{x} and \texttt{y}, if \texttt{x < y} then \texttt{(f y) < (f x)}.
\end{itemize}
5. Abstract Types (20 Points total)
Suppose that $S$ is a module that implements this interface for the abstract type of sets (taken directly from homework 3):

```ocaml
module type Set = sig
type 'a set
  val empty : 'a set
  val is_empty : 'a set -> bool
  val member : 'a -> 'a set -> bool
  val add : 'a -> 'a set -> 'a set
  val remove : 'a -> 'a set -> 'a set
  val equal : 'a set -> 'a set -> bool
  val elements : 'a set -> 'a list
  val fromList : 'a list -> 'a set
  val setSize : 'a set -> int
end
```

(8 points) Indicate whether each of the following is true or false:

a. T F A client of the set module could use the following program to determine whether $l$ contains 3:
```ocaml
let l : int list = ...
let answer : bool = S.member 3 l
```

b. T F It is possible that the module $S$ implements the type `'a set` internally by doing:
```ocaml
type 'a set = 'a option
```

c. T F Given the interface above, implementing a client function
```ocaml
union : 'a S.set -> 'a S.set -> 'a S.set
```
that produces a set containing all of the elements of both of its inputs requires the use of the $S$.elements function.

d. T F The $add$ function may be defined recursively.
(12 points) Suppose that you have an application that uses the setSize operation extremely often—much more often than the add and remove operations, for example. You therefore need to implement a version of the Set module that makes calling setSize very efficient.

One way to do that is to use an internal representation of sets that stores the current size of the set along with the elements themselves. We can therefore use the following implementation of the Set interface, which represents sets as pairs with an invariant:

INVARIANT: if the pair \((n, \mathbf{l})\) represents a set \(\{a_1, \ldots, a_n\}\), then \(\mathbf{l}\) is a list with no duplicates containing the elements \(a_1, \ldots, a_n\) in some order, and \(n = \text{length } \mathbf{l}\).

This invariant lets us implement the setSize function very efficiently by using the \(\text{fst}\) operation, which projects the first element of a pair (\(\text{snd}\) projects the second). However, we have to adapt the rest of the operations to maintain the new invariants. We have provided the empty, is_empty, and add functions. Complete the implementation of the remove function (which is used by add).

```ocaml
module SizeSet : Set = struct
    type 'a set = (int * 'a list)

    let empty : 'a set = (0, [])

    let is_empty (s:'a set) = (fst s = 0)

    let rec remove (x:'a) (s:'a set) : 'a set =
        ...

    let rec add (x:'a) (s:'a set) : 'a set =
        let r = remove x s in
        (1+(fst r), x::(snd r))

    let setSize (s:'a set) : int = fst s
end
```