SOLUTIONS

1. Program Design (16 points)

Use the four-step design methodology to implement a function called \texttt{trim} that, when given an integer \( n \) and a list \( x \), returns the list with all occurrences of \( n \) removed \textit{from the beginning of the list}.

For example, \texttt{trim 1 [1;1;2;1]} should yield the list \([2;1]\).

\begin{enumerate}
  \item \textbf{Step 1} is \textit{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.
  \item \textbf{Step 2} is \textit{formalizing the interface}. Write down the \textit{type} of the \texttt{trim} function as you might find it in a \texttt{.mli} file or module interface.

\begin{verbatim}
val trim : int -> int list -> int list
\end{verbatim}

\textit{Grading Scheme: 2 points. Accepting both \texttt{int} and \texttt{'}a type}

\item \textbf{Step 3} is \textit{writing test cases}. Complete the following tests with the expected behavior. We have done the first one for you, based on the problem description.

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” input numbers and lists. Fill in the description string of the \texttt{run_test} function with a short explanation of \textit{why} the test case is interesting.

\begin{verbatim}
i. let test () : bool =
    trim 1 [1;1;2;1] = [2;1];
    \;
    \texttt{run_test "given from the problem description" test}
\end{verbatim}

\textit{Grading Scheme: 3 points each. Interesting: empty, single element \( n \), list only contains \( n \), \( n \) not at beginning, one \( n \) at beginning, many \( n \)’s at beginning, \( n \) not in list at all. -1 poor or no description (i.e. description just states what the test case is, not why it was interesting.)}
d. Step 4 is implementing the program.

```ml
let rec trim (x:int) (ys:int list) : int list =
    begin match ys with
    | [] -> []
    | y :: tl -> if x = y then trim x tl else ys
    end
```

**Grading Scheme:** 8 points total, rough guidelines

- 2 for base case returning []
- 1 for checking the head
- 1 for correct type annotations (consistent w/ first part)
- 2 for recursing when the head matches
- 2 for not recursing when the head doesn’t match
- -1 for minor type/syntax errors/forgetting arg to recursive call to trim
- -2 for non-exhaustive pattern matching
- other deductions at discretion
2. Types (20 points)

For each OCaml value below, fill in the blank where the type annotation could go or write “ill typed” if there is a type error. Recall that the @ operator appends two lists together in OCaml. We have done the first one for you.

```ocaml
let x : ___________ int list ______________ = [ 120 ]

let a : ______ int list _______________ = 1 :: [2]
let b : ______ ill typed _______________ = 1 :: [[]]
let c : ___( int * bool ) list ________ = [(1,true)]
let d : _____ int * bool ______________ = (1, true)
let e : _____ ill typed _______________ = [1; true]
let f : _____ int list list ____________ = [[]] @ [[]]
let g : _____ int list _________________ = [1] @ []
let h : _____ int -> int ______________ = (fun x -> fun y -> x + y) 3
let i : ____ (int -> int) list ________ = [fun x -> x + 1; fun x -> x - 1]
let j : ____ int option ________________ = Some 3
```

Grading scheme: 2 points per answer. Half credit for omitting necessary parentheses (such as int * bool list vs. (int * bool) list or int -> int list vs. (int -> int) list). Half credit for types that are too generic.
3. Datatypes and Trees (16 points)

Consider the following definition of trees with integers stored only at the leaves:

```haskell
type leafy_tree =
  | Leaf of int
  | Branch of leafy_tree * leafy_tree
```

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

a. let rec \( f \) (t : leafy_tree) : int =
    begin match t with
    | Leaf x -> x
    | Branch (l, r) -> max (f l) (f r)
    end

    let \( r \) : int = f (Branch (Leaf 1, Branch (Leaf 3, Leaf 2)))

Answer: \( r = 3 \)

b. let rec \( g \) (y : int) (t : leafy_tree) : leafy_tree =
    begin match t with
    | Leaf x -> Leaf y
    | Branch (l, r) -> Branch (g y l, g y r)
    end

    let \( r \) : leafy_tree = g 4 (Branch (Leaf 1, Leaf 2))

Answer: \( r = \text{Branch (Leaf 4, Leaf 4)} \)

c. let rec \( h \) (y : int) (t : leafy_tree) : leafy_tree =
    begin match t with
    | Leaf x -> Leaf (y + x)
    | Branch (l, r) -> h y l
    end

    let \( r \) : leafy_tree = h 4 (Branch (Leaf 1, Leaf 2))

Answer: \( r = \text{Leaf 5} \)

d. let rec \( j \) (t : leafy_tree) : int list =
    begin match t with
    | Leaf x -> [x]
    | Branch (l, r) -> (j l) @ (j r)
    end

    let \( r \) : int list = j (Branch (Leaf 1, Branch (Leaf 3, Leaf 2)))

Answer: \( r = [1;3;2] \)

Grading Scheme: 4 points per answer. No deduction for minor syntax errors, such as \([1,3,2]\) instead of \([1;3;2]\). No partial credit.
4. Higher-order function patterns (12 points)

Recall the functions `transform` and `fold` discussed in lecture and used in HW04:

```ocaml
let rec transform (f: 'a -> 'b) (x: 'a list): 'b list = 
begin match x with 
| [] -> [] 
| h :: t -> (f h) :: (transform f t) 
end

let rec fold (combine: 'a -> 'b -> 'b) (base:'b) (x : 'a list) : 'b = 
begin match x with 
| [] -> base 
| h :: t -> combine h (fold combine base t) 
end
```

The following recursive functions have been given for you. Rewrite each of them using either `transform` or `fold`.

a. `let rec member (elt : int) (x : int list) : bool = 
   begin match x with 
   | [] -> false 
   | h :: t -> h = elt || member elt t 
   end

   let member (elt : int) (x : int list) : bool = 
   fold (fun (h:int) (a:bool) -> h = elt || a) false x

b. `let rec add_ancestor_labels_list (rs: tree list): labeled_tree list = 
   begin match rs with 
   | [] -> [] 
   | h :: t -> add_ancestor_labels h :: add_ancestor_labels_list t 
   end

   let add_ancestor_labels_list (rs : tree list) : labeled_tree list = 
   transform add_ancestor_labels rs 
   or 
   fold (fun (h:tree) (a:labeled_tree list) -> add_ancestor_labels h :: a) [] rs`

Grading Scheme: 6 points per answer. Partial credit given as appropriate. (i.e for part (a) 2 point for identifying that must be fold. 1 point for giving the base case for fold. 1 point for correct type annotations combine argument. 1 point for providing list argument.) Ok to omit typing annotations from parameters of higher-order functions, but 1 point deduction if incorrect. 1 point deduction getting the arguments to fold out of order. Minor syntax errors with no deductions.
5. Modules and Abstract types (12 points)

Consider the following module definition

```ocaml
module M = struct
  type t = int
  let zero : t = 0
  let incr (x : t) : t = x + 1
  let to_int (x: t) : int = x
  let from_int (x : int) : t = x
end
```

and the following invariant that the module designer would like to maintain

*A value of type* `M.t` *is never negative.*

Evaluate whether each of the following signatures for `M` could be used to maintain this invariant.

a. type `t`
   - val zero : `t`
   - val incr : `t` -> `t`
   - val to_int : `t` -> int
   - val from_int : bool -> `t`

Circle one:

i. This interface prevents all clients from breaking the invariant
ii. A client could break the invariant if `M` used this interface
iii. *This interface doesn’t match `M` (it would cause a compilation error)*

b. type `t`
   - val zero : `t`
   - val incr : `t` -> `t`
   - val to_int : `t` -> int

Circle one:

i. *This interface prevents all clients from breaking the invariant*
ii. A client could break the invariant if `M` used this interface
iii. This interface doesn’t match `M` (it would cause a compilation error)

c. type `t`
   - val zero : `t`
   - val incr : `t` -> `t`
   - val to_int : `t` -> int
   - val from_int : int -> `t`

Circle one:

i. This interface prevents all clients from breaking the invariant
ii. *A client could break the invariant if `M` used this interface*
iii. This interface doesn’t match `M` (it would cause a compilation error)

*Grading Scheme: 4 points per answer*
6. Binary Search Trees (12 points)

Circle either T (true) or F (false) for each statement about binary search trees below. Below, *insert* and *delete* refer to the BST functions that we discussed in class. For reference, these functions appear in the appendix.

a. T F An Empty tree satisfies the BST invariant.

b. T F The tree below satisfies the BST invariant.

```
    5
   / \    
  2   7
 / \    
1   6
```

c. T F The tree below satisfies the BST invariant.

```
    3
   / \    
  2   7
 / 
2
```

d. T F Suppose we are given the following tree t that does not satisfy the BST invariant:

```
    2
   / \    
  5   3
```

Then the expression `(insert (delete 5 t) 5)` will return a tree that satisfies the BST invariant (i.e. a tree that is a BST).

e. T F If you insert a number n into a BST t, and then delete n from the result, then the resulting tree will always have exactly the same shape and same elements as t.

f. T F If you delete a number n from a BST t, and then insert n into the result, then the final tree will always have exactly the same shape and same elements as t.

*Grading Scheme: 2 points each*
7. More Binary Search Trees (12 points)

Recall the type of generic binary search trees:

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

Implement a function called `scs`, short for smallest containing subtree. This function should, when given two values that appear in a binary search tree, return the smallest subtree that contains both of those values.

For example, given the tree

```
  t1 = 4
     /\  
    2 5
   / \ 
  1 3
```

the smallest containing subtree of 1 and 3 is

```
  t2 = 2
     /\  
    1 3
```

Likewise, the smallest subtree of `t1` containing 1 and 2 is also `t2`. On the other hand, the smallest subtree of `t1` that contains both 2 and 5 is the whole tree.

You should assume that the input tree is a binary search tree, that both input values are contained within the tree, and that the first argument is smaller than the second. Your solution does not need to detect whether any of these assumptions are violated.

Your implementation must take advantage of the binary search tree invariant and must work for generic binary search trees. You may not use any auxiliary functions in your solution, such as `lookup`, `insert`, or `delete`.

(* Assume that t is a BST *)

```plaintext
let rec scs (x:'a) (y:'a) (t:'a tree) : 'a tree =
  begin match t with
    | Empty -> failwith "impossible"
    | Node (l, z, r) ->
      if y < z then scs x y l
      else if x > z then scs x y r
      else t
  end
```

Grading Scheme:

- 2 correct recursive call when both values on left
- 2 correct recursive call when both values on right
- 3 returning whole tree when \( x < z \) and \( y > z \). (can return either \( t \) or `Node(l,z,r)`)
- 2 returning whole tree when either \( x \) or \( y = z \)
- 2 correct generic type signature (1 point if not generic)
- 1 pattern match on \( t \)
- Don’t necessarily need a case for `Empty`. Can do anything there.
- No deduction for minor syntax errors, such as forgetting `rec` keyword.
Appendix - BSTs containing integers

type tree =
  | Empty
  | Node of tree * int * tree

let rec lookup (t:tree) (n:int) : bool = begin match t with
  | Empty -> false
  | Node(lt, x, rt) ->
    if x = n then true
    else if n < x then lookup lt n
    else lookup rt n
end

let rec insert (t:int tree) (n:int) : int tree = begin match t with
  | Empty -> Node (Empty, n, Empty)
  | Node (lt, x, rt) ->
    if x = n then t
    else if n < x then Node (insert lt n, x, rt)
    else Node(lt, x, insert rt n)
end

let rec tree_max (t:tree) : int = begin match t with
  | Empty -> failwith "tree_max called on empty tree"
  | Node(_,x,Empty) -> x
  | Node(_,_,rt) -> tree_max rt
end

let rec delete (n:int) (t:tree) : tree = begin match t with
  | Empty -> Empty
  | Node(lt,x,rt) ->
    if x = n then
      begin match (lt,rt) with
        | (Empty, Empty) -> Empty
        | (Node _, Empty) -> lt
        | (Empty, Node _) -> rt
        | _ -> let m = tree_max lt in
          Node(delete m lt, m, rt)
      end
    else if n < x then Node(delete n lt, x, rt)
    else Node(lt, x, delete n rt)
end