```
(* Some general points for Part B
- For a 3 marks question, every mistake will lead to 2 marks
deduction (so two mistakes will lead to 0 marks).
- For a 4 marks question, every mistake will lead to 2 marks
deduction (so two mistakes will lead to 0 marks).
- For a 6 marks question, every mistake will lead to 2 marks
deduction (so three mistakes will lead to 0 marks).
- If you have used the standard List notation (e.g. `[]` for
empty, `x::xs` for separating head and tail, etc.), 2 marks
will be deducated for each such mistake.
- If you have defined a local helper function without `let`
and `in`, 2 marks will be deducated for each such mistake. (I
am not sure if there is any other way of definining a local
helper function in OCaml. Just using `let` without `in` is
wrong. If you think otherwise, you may try compiling such code
and let me know if it works for you.)
- If you have defined any additional helper function in global
scope, then no marks will be given for that question.
- If you have modified any function signature (arguments or
return types), then no marks will be given for that question.
- If you have used any built-in list functions (e.g.
`List.map`, `List.filter`, etc.), then no marks will be given
for that question.
- If you have used for or while loops, then no marks will be
given for that question.
*)

(*
PART B: Train Reservation System
   You are tasked with implementing a train reservation system
using a custom recursive list
   data structure (my_list) instead of OCaml's built-in lists.
The system should support:
   - Basic list operations (map, filter, fold, sort)
   - Train searching and sorting
   - Seat availability checking and booking
   - Tatkal pricing
   - Passenger management and searching

   The types and data structures are provided. Your task is to
implement the UNIMPLEMENTED functions
   according to their specifications.
   - DO NOT MODIFY THE ALREADY IMPLEMENTED FUNCTIONS.
   - DO NOT CHANGE THE FUNCTION SIGNATURES (ARGUMENTS AND
RETURN TYPES).
   - DO NOT USE ANY BUILT-IN LIST FUNCTIONS (like List.map,
List.filter, etc.)
   - DO NOT ADD ANY OTHER FUNCTIONS IN THE GLOBAL SCOPE.
   - YOU ARE FREE TO ADD ANY HELPER FUNCTIONS WITHIN THE BODY
OF THE PROVIDED FUNCTION DEFINITIONS.
   - DO NOT USE FOR AND WHILE LOOPS.
   - YOU DON'T NEED TO HANDLE ERRORS.
   - FOR EACH FUNCTION, WE HAVE PROVIDED A HINT WHICH CONTAINS
THE LIST OF FUNCTIONS YOU CAN/SHOULD USE TO IMPLEMENT THE
FUNCTION.
   - YOU NEED NOT STRICTLY FOLLOW THE HINT. BUT WE ADVICE YOU
USE THE HINT AND NOT OVERTHINK.
   - PLEASE TAKE 10-15 MINUTES TO READ AND UNDERSTAND THE FULL
QUESTION CAREFULLY BEFORE YOU START.
   - MOST IMPORTANTLY, DO NOT OVERTHINK! THERE ARE NO TRICKY
OR MISLEADING QUESTIONS. MOST OF THE QUESTIONS ARE
STRAIGHTFORWARD.
*)

(* Type definitions - DO NOT MODIFY *)
type 'a option = None | Some of 'a
type 'a my_list = Empty | Node of 'a * 'a my_list
(* [] is not a valid empty list. *)
type station = {code: string; name: string; arrival_time:
string;
                departure_time: string; distance_from_source:
int}
type seat_class = Sleeper | AC3 | AC2 | AC1
type passenger = {name: string; age: int; gender: string;}
type seat_availability = {class_type: seat_class; price:
float; available_seats: int}
type train = {train_number: string; train_name: string;
classes: seat_availability my_list;
            schedule: station my_list;
            departure_time: string; arrival_time: string}
type booking = {user_name: string; train_number: string;
class_booked: seat_class;
              passengers: passenger my_list; is_tatkal: bool}

(** Important Assumptions: For Simplicity,
1. our train reservation system does not have any notion of
date. So, whenever we talk about booking a ticket or checking
seat availability in a specific train, we do not care about
date. In other words, there is only a single instance of any
train for which we are going to book tickets or check seat
availability
2. Passengers are not alloted any seat numbers. We will just
use the seat class and check seat availability to book
tickets. While cancelling tickets, we will accordingly update
the seat availability.
*)
```

```
(** QUESTION 1: Implement my_map [3 marks]
    This function applies a function to each element of a list
to create a new list
    Input:
    - f: function of type ('a -> 'b) to apply to each element
    - lst: input list of type 'a my_list
    Returns:
    - a new list of type 'b my_list containing f applied to
each element
    Examples:
    my_map (fun x -> x * 2) (Node(1, Node(2, Node(3, Empty))))
    (* returns Node(2, Node(4, Node(6, Empty))) because each
number is multiplied by 2 *)
    my_map String.uppercase_ascii (Node("a", Node("b",
Node("c", Empty))))
    (* returns Node("A", Node("B", Node("C", Empty))) because
each letter is capitalized *)
    Hint: Use pattern matching and recursion *)
let rec my_map (f: 'a -> 'b) (lst: 'a my_list) : 'b my_list =
  match lst with
  | Empty -> Empty
  | Node(x, rest) -> Node(f x, my_map f rest)
  (* -2 marks for either [] or x :: rest or both *)
```

```
(** QUESTION 2: Implement my_filter [4 marks]
    This function returns only those elements in a my_list
that satisfy a given condition
    Input:
    - f: function of type ('a -> bool) that tests each element
    - lst: input list of type 'a my_list
    Returns:
    - a new list containing only elements for which f returns
true
    Examples:
    my_filter (fun x -> x > 2) (Node(1, Node(3, Node(2,
Empty))))
    (* returns Node(3, Empty) because only 3 is greater than 2
*)
    my_filter (fun s -> String.length s = 3) (Node("hi",
Node("cat", Node("dog", Empty))))
    (* returns Node("cat", Node("dog", Empty)) because only
"cat" and "dog" have length 3 *)
    Hint: Use pattern matching and recursion *)
let rec my_filter (f: 'a -> bool) (lst: 'a my_list) : 'a
my_list =
  match lst with
  | Empty -> Empty
  | Node(x, rest) -> if f x then Node(x, my_filter f rest)
else my_filter f rest
  (* -2 marks for either [] or x :: rest or both *)
  (* 0 if pattern matching is not used *)

(** QUESTION 3: Implement my_length [3 marks]
    This function counts the number of elements in a list
    Input:
    - lst: input list of type 'a my_list
    Returns:
    - an integer representing the number of elements in the
list
    Examples:
    my_length (Node(1, Node(2, Node(3, Empty))))
    (* returns 3 because there are three elements: 1, 2, and 3
*)
    my_length Empty
    (* returns 0 because the list is empty *)
    Hint: Use recursion *)
let rec my_length (lst: 'a my_list) : int =
  match lst with
  | Empty -> 0
  | Node(_, rest) -> 1 + my_length rest
  (* -2 marks for either [] or x :: rest or both *)
  (* 0 if pattern matching is not used *)
```

```
(** QUESTION 4: Implement my_fold_left [4 marks]
    This function combines all elements of a list using an
accumulator function
    Input:
    - f: function of type ('a -> 'b -> 'a) that combines
accumulator with each element
    - acc: initial accumulator value of type 'a
    - lst: input list of type 'b my_list
    Returns:
    - final accumulator value of type 'a after processing all
elements
    Examples:
    my_fold_left (+) 0 (Node(1, Node(2, Node(3, Empty))))
    (* returns 6 because: 0 + 1 = 1, then 1 + 2 = 3, then 3 +
3 = 6 *)
    my_fold_left (fun acc x -> acc ^ x) "" (Node("a",
Node("b", Node("c", Empty))))
    (* returns "abc" because: "" ^ "a" = "a", then "a" ^ "b" =
"ab", then "ab" ^ "c" = "abc" *)
    Hint: Use pattern matching and recursion *)
let rec my_fold_left (f: 'a -> 'b -> 'a) (acc: 'a) (lst: 'b
my_list) : 'a =
  match lst with
  | Empty -> acc
  | Node(x, rest) -> my_fold_left f (f acc x) rest
```

```
(** QUESTION 5: Implement insert_sorted [4 marks]
    This function inserts an element into a sorted list while
maintaining ascending order
    Input:
    - cmp: comparison function of type ('a -> 'a -> int) that
returns:
           negative if first arg < second arg
           zero if args are equal
           positive if first arg > second arg
    - x: element to insert
    - lst: sorted input list
    Returns:
    - a new sorted list with x inserted in the correct
position
    Examples:
    insert_sorted compare 2 (Node(1, Node(3, Empty)))
    (* returns Node(1, Node(2, Node(3, Empty))) because 2
belongs between 1 and 3 *)
    insert_sorted String.compare "b" (Node("a", Node("c",
Empty)))
    (* returns Node("a", Node("b", Node("c", Empty))) because
"b" belongs between "a" and "c" *)
    Hint: Use pattern matching, recursion, and comparison
function *)
let rec insert_sorted (cmp: 'a -> 'a -> int) (x: 'a) (lst: 'a
my_list) : 'a my_list =
  match lst with
  | Empty -> Node(x, Empty)
  | Node(y, rest) -> if cmp x y < 0 then Node(x, Node(y,
rest)) else Node(y, insert_sorted cmp x rest)
```

```
(** QUESTION 6: Implement my_sort [4 marks]
    This function sorts a list in ascending order using a
comparison function
    Input:
    - cmp: comparison function of type ('a -> 'a -> int) that
returns:
           negative if first arg < second arg
           zero if args are equal
           positive if first arg > second arg
    - lst: input list to sort
    Returns:
    - a new list with all elements sorted according to the
comparison function
    Examples:
    my_sort compare (Node(3, Node(1, Node(2, Empty))))
    (* returns Node(1, Node(2, Node(3, Empty))) because 1 < 2
< 3 *)
    my_sort (fun x y -> compare (String.length x)
(String.length y))
           (Node("abc", Node("a", Node("ab", Empty))))
    (* returns Node("a", Node("ab", Node("abc", Empty)))
because lengths: 1 < 2 < 3 *)
    Hint: Use pattern matching, insert_sorted, and recursion
*)
let rec my_sort (cmp: 'a -> 'a -> int) (lst: 'a my_list) : 'a
my_list =
  match lst with
  | Empty -> Empty
  (* | Node(x, Empty) -> Node(x, Empty) *)
  | Node(x, rest) -> insert_sorted cmp x (my_sort cmp rest)
```

```
(** QUESTION 7: Implement my_mem [4 marks]
    This function checks if an element exists in a list
    Input:
    - x: element to search for
    - lst: list to search in
    Returns:
    - true if element exists in list, false otherwise
    Examples:
    my_mem 2 (Node(1, Node(2, Node(3, Empty))))
    (* returns true because 2 is in the list *)
    my_mem "d" (Node("a", Node("b", Node("c", Empty))))
    (* returns false because "d" is not in the list *)
    Hint: Use pattern matching and recursion *)
let rec my_mem (x: 'a) (lst: 'a my_list) : bool =
  match lst with
  | Empty -> false
  | Node(y, rest) -> x=y || my_mem x rest
  (* -2 marks for either [] or x :: rest or both *)
  (* 0 if pattern matching is not used *)
  (* -2 if == is used *)
  (* Alternate solutiuon: if x = y then true else my_mem x
rest *)
```

```
(** QUESTION 8: Implement sort_trains_by_class [6 marks]
    This function sorts trains based on price or seat
availability
    Input:
    - trains: list of trains to sort
    - class_type: seat class to compare (Sleeper, AC3, etc.)
    - sort_by: string indicating sort criterion ("price" or
"available_seats")
    Returns:
    - list of trains sorted in ascending order by the
specified criterion
    Examples:
    sort_trains_by_class trains Sleeper "price"
    (* returns trains sorted by Sleeper class price:
       Node({price=300.0; ...}, Node({price=500.0; ...},
Node({price=700.0; ...}, Empty))) *)
    sort_trains_by_class trains AC3 "available_seats"
    (* returns trains sorted by AC3 available seats:
       Node({seats=5; ...}, Node({seats=10; ...},
Node({seats=15; ...}, Empty))) *)
    Hint: Use pattern matching, my_sort, my_filter *)
let sort_trains_by_class (trains: train my_list) (class_type:
seat_class)
    (sort_by: string) : train my_list =
  let extract_compare_value (t: train) : seat_availability =
    let only_class_info = my_filter (fun x -> x.class_type =
class_type) t.classes in
    match only_class_info with
    | Node(y, _) -> y
    | Empty -> failwith "No class details found" in
  match sort_by with
  | "price" -> my_sort (fun t1 t2 -> compare
(extract_compare_value t1).price (extract_compare_value
t2).price) trains
  | "available_seats" -> my_sort (fun t1 t2 -> compare
(extract_compare_value t1).available_seats
(extract_compare_value t2).available_seats) trains
  | _ -> failwith "Invalid sort criterion"
```

```
(** QUESTION 9: Implement check_seat_availability [6 marks]
    This function verifies if requested number of seats are
available
    Input:
    - train: train to check
    - class_type: seat class to check (Sleeper, AC3, etc.)
    - num_passengers: number of seats needed
    Returns:
    - true if enough seats are available, false otherwise
    Examples:
    check_seat_availability train Sleeper 2
    (* returns true if Sleeper class has at least 2 seats
available *)
    check_seat_availability train AC1 5
    (* returns false if AC1 class has fewer than 5 seats
available *)
    Hint: Use pattern matching and my_filter *)
let check_seat_availability (train: train) (class_type:
seat_class)
    (num_passengers: int) : bool =
  let only_class_info = my_filter (fun x -> x.class_type =
class_type) train.classes in
  match only_class_info with
  | Node(y, _) -> y.available_seats >= num_passengers
  | Empty -> false
```

```
(** QUESTION 10: Implement tatkal_pricing [3 marks]
    This function implements dynamic pricing for tatkal (last-
minute) tickets
    Input:
    - surcharge: float (e.g., 1.5 for 50% extra charge)
    Returns:
    - a function of type (float -> float) that calculates
tatkal price from base price
    Examples:
    let apply_tatkal = tatkal_pricing 1.5 in
    apply_tatkal 1000.0;;  (* returns 1500.0 because
base_price * surcharge = 1000.0 * 1.5 *)
    Hint: No hints for this question *)
let tatkal_pricing (surcharge: float) : (float -> float) =
  fun base_price -> base_price *. surcharge
  (* Alternate solution: base_price can have any other name
but surcharge must be the same *)
  (* -3 marks for base_price *. surcharge => value return
should be given zero marks *)
```

```
(** QUESTION 11: Implement combine_passenger_lists [4 marks]
    This function merges two passenger lists into one
    Input:
    - acc: first passenger list (accumulator)
    - p: second passenger list to add
    Returns:
    - a new list containing all passengers from both lists
    Examples:
    let p1 = {name="Alice"; age=25; gender="F"} in
    let p2 = {name="Bob"; age=30; gender="M"} in
    combine_passenger_lists (Node(p1, Empty)) (Node(p2,
Empty))
    (* returns Node(p2, Node(p1, Empty)) because it combines
both lists *)
    combine_passenger_lists Empty (Node(p1, Empty))
    (* returns Node(p1, Empty) because first list is empty *)
    Hint: Use pattern matching and recursion *)
let rec combine_passenger_lists (acc: passenger my_list) (p:
passenger my_list)
    : passenger my_list =
  match acc with
  | Empty -> p
  | Node(x, rest) -> Node(x, combine_passenger_lists rest p)
```

```
(** QUESTION 12: Implement get_passengers_for_class [6 marks]
    This function finds all matching passengers in a specific
class
    Input:
    - class_type: seat class to search in
    - bookings: list of all bookings to search through
    - filter_fn: function of type (passenger -> bool) that
defines matching criteria
    Returns:
    - matching_passenger_list
    Examples:
    get_passengers_for_class Sleeper bookings (fun p -> p.age
> 60)
    (* returns Node({name="John"; age=65; ...}, Empty)
       if John is the only senior citizen in Sleeper class *)
    get_passengers_for_class AC3 bookings (fun p -> p.gender =
"F")
    (* returns Node({name="Alice"; ...},
Node({name="Mary"; ...}, Empty))
       if Alice and Mary are the female passengers in AC3 *)
    Hint: Use my_fold_left, my_filter and
combine_passenger_lists *)
let get_passengers_for_class (class_type: seat_class)
(bookings: booking my_list)
    (filter_fn: passenger -> bool) : passenger my_list =
  my_fold_left (fun acc booking ->
    if booking.class_booked = class_type then
      combine_passenger_lists acc (my_filter filter_fn
booking.passengers)
    else acc
  ) Empty bookings
```

```
(** QUESTION 13: Implement search_passengers [6 marks]
    This function finds matching passengers across all seat
classes
    Input:
    - classes: list of seat classes to search through
    - bookings: list of all bookings to search through
    - filter_fn: function of type (passenger -> bool) that
defines matching criteria
    Returns:
    - list of pairs, each containing (seat_class *
matching_passenger_list)
    Examples:
    search_passengers classes bookings (fun p -> p.age > 60)
    (* returns Node((Sleeper, Node({name="John"; age=65; ...},
Empty)),
                Node((AC2, Node({name="Mary"; age=70; ...},
Empty)), Empty))
       if John and Mary are senior citizens in their
respective classes *)
    search_passengers classes bookings (fun p -> p.gender =
"F" && p.age < 30)
    (* returns Node((AC3, Node({name="Alice"; age=25; ...},
Empty)), Empty)
       if Alice is the only female passenger under 30 *)
    Hint: Use my_fold_left and get_passengers_for_class *)
let search_passengers (classes: seat_class my_list) (bookings:
booking my_list)
    (filter_fn: passenger -> bool) : (seat_class * passenger
my_list) my_list =
  my_fold_left (fun acc class_type ->
    Node((class_type, get_passengers_for_class class_type
bookings filter_fn), acc)
  ) Empty classes
```

```
(** QUESTION 14: Implement update_seats
    This function modifies the available seats count for a
specific class
    Input:
    - train: train to update
    - class_type: seat class to modify
    - num_seats: number of seats to add (positive) or remove
(negative)
    Returns:
    - updated train record with modified seat count
    Examples:
    update_seats train Sleeper (-2)
    (* returns updated train with 2 fewer Sleeper seats
available *)
    update_seats train AC1 1
    (* returns updated train with 1 more AC1 seat available
(after cancellation) *)
    Hint: Use my_map *)
let update_seats (train: train) (class_type: seat_class)
(num_seats: int) : train =
  let updated_classes = my_map (fun x -> if x.class_type =
class_type then
    {x with available_seats = x.available_seats + num_seats}
else x) train.classes in
  {train with classes = updated_classes}
```

```
(** QUESTION 15: Implement book_ticket [6 marks]
    This function attempts to create a booking for passengers
and accordingly updates the train record
    Input:
    - user_name: name of person making the booking
    - train: train to book tickets on
    - passengers: list of passengers to book for
    - class_booked: desired seat class
    - is_tatkal: whether this is a tatkal (last-minute)
booking
    Returns:
    - Some (booking_record, updated_train_record) if
successful, None if seats not available
    Examples:
    book_ticket "Alice" train passengers Sleeper false
    (* returns Some (booking_record, updated_train_record) if
booking succeeds *)
    book_ticket "Bob" train passengers AC1 true
    (* returns None if required seats are not available *)
    Hint: Use check_seat_availability, update_seats and
my_length *)
let book_ticket (user_name: string) (train: train)
(passengers: passenger my_list)
    (class_booked: seat_class) (is_tatkal: bool)
    : (booking * train) option =
  if check_seat_availability train class_booked (my_length
passengers) then
    let updated_train = update_seats train class_booked (-
my_length passengers) in
    let updated_booking = {user_name = user_name; train_number
= updated_train.train_number;
      class_booked = class_booked; passengers = passengers;
is_tatkal = is_tatkal} in
    Some (updated_booking, updated_train)
  else None
```

```
(** QUESTION 16: Implement cancel_tickets [6 marks]
    This function removes specified passengers from a booking
    Input:
    - booking: booking to modify
    - train: train whose capacity needs updating
    - passengers: list of passengers whose tickets need to be
cancelled
    Returns:
    - tuple containing (updated booking * updated train)
    Examples:
    cancel_tickets booking train (Node({name = "Alice"; age =
25; gender = "F"}, Empty))
    (* returns (booking without Alice, train with 1 more
available seat) *)
    cancel_tickets booking train Empty
    (* returns (unchanged booking, unchanged train) when no
seats to cancel *)
    Hint: Use my_filter, my_length, my_mem and update_seats *)
let cancel_tickets (booking: booking) (train: train)
(passengers: passenger my_list)
    : booking * train =
  let updated_booking = {booking with passengers = my_filter
(fun p -> if (my_mem p passengers) then false else true)
booking.passengers} in
  let updated_train = update_seats train booking.class_booked
(-my_length passengers) in
  (updated_booking, updated_train)

(* MAIN FUNCTION - DO NOT MODIFY. YOU DON'T NEED THIS CODE.
THIS IS JUST FOR SHOWING HOW THE ABOVE FUNCTIONS WORK. *)
let main () =
  (* Create test data *)
  let train = {
    train_number = "12345";
    train_name = "Express A";
    classes = Node ({class_type = Sleeper; price = 500.0;
available_seats = 100},
            Node ({class_type = AC3; price = 1200.0;
available_seats = 50}, Empty));
    schedule = Node ({ code = "DEL"; name = "Delhi";
arrival_time = "--";
                      departure_time = "10:00";
distance_from_source = 0 },
                    Node ({ code = "BPL"; name = "Bhopal";
arrival_time = "15:00";
                          departure_time = "15:15";
distance_from_source = 700 }, Empty));
    departure_time = "10:00";
    arrival_time = "22:00"
  } in
  let trains = Node(train, Empty) in

  (* Test basic list functions *)
  assert (my_map (fun x -> x * 2) (Node(1, Node(2, Node(3,
Empty)))) =
         Node(2, Node(4, Node(6, Empty))));

  assert (my_filter (fun x -> x > 2) (Node(1, Node(3, Node(2,
Empty)))) =
         Node(3, Empty));

  assert (my_length (Node(1, Node(2, Node(3, Empty)))) = 3);

  assert (my_fold_left (+) 0 (Node(1, Node(2, Node(3,
Empty)))) = 6);

  assert (my_sort compare (Node(3, Node(1, Node(2, Empty)))) =
         Node(1, Node(2, Node(3, Empty))));

  assert (my_mem 2 (Node(1, Node(2, Node(3, Empty)))) = true);
  assert (my_mem 4 (Node(1, Node(2, Node(3, Empty)))) =
false);

  (* Test train sort *)
  let sorted_trains = sort_trains_by_class trains Sleeper
"price" in
  assert (my_length sorted_trains = 1);

  (* Test booking functions *)
  assert (check_seat_availability train Sleeper 2 = true);
  assert (check_seat_availability train Sleeper 101 = false);

  assert ((tatkal_pricing 1.5) 1000.0 = 1500.0);
  assert ((tatkal_pricing 2.0) 1000.0 = 2000.0);

  (* Create test bookings *)
  let booking = {
    user_name = "Alice";
    train_number = "12345";
    class_booked = Sleeper;
    passengers = Node({name = "Alice"; age = 25; gender =
"F"},
                Node({name = "Bob"; age = 30; gender = "M"},
Empty));
    is_tatkal = false
  } in
  let bookings = Node(booking, Empty) in
  (* Test passenger management *)
  let senior_passengers = get_passengers_for_class Sleeper
bookings (fun p -> p.age > 60) in
  assert (my_length senior_passengers = 0);

  let combined = combine_passenger_lists
    (Node({name = "Eve"; age = 40; gender = "F"}, Empty))
    (Node({name = "Frank"; age = 45; gender = "M"}, Empty)) in
  assert (my_length combined = 2);

  let classes = Node(Sleeper, Node(AC3, Empty)) in
  let adult_passengers = search_passengers classes bookings
(fun p -> p.age >= 30) in
  assert (my_length adult_passengers = 2);

  (* Test cancellation and booking *)
  let (updated_booking, updated_train) = cancel_tickets
booking train (Node({name = "Alice"; age = 25; gender = "F"},
Empty)) in
  assert (my_length updated_booking.passengers = 1);

  let booking_result = book_ticket "Alice" train
    (Node({name = "Alice"; age = 30; gender = "F"}, Empty))
    Sleeper false in
  assert (booking_result <> None);

  (* Return something to indicate all tests were run *)
  sorted_trains

let () =
  try
    ignore (main ());
    print_endline "All tests passed!"
  with Assert_failure (file, line, position) ->
    Printf.printf "Test failed at %s, line %d, position %d\n"
file line position
```