CS51 **Spring 2024**

Code Review 1: Intro to CS51 (Solutions)

1 Practice Problems

Complete the value and type questions without use of the `REPL`. Do not use the `OCaml` List Module for questions that ask you to define a function.

Problem 1.1 (Values and Types)**.** What is the type and value of each of the following OCaml expressions? If the expression is ill-typed, explain why it is ill-typed.

1. `( ** ) 3.  2.`

   Solution
   Type: `float`

   Value : `9.0`

   `( ** )` is a function of type `float -> float -> float`. It's applied on `3.` and `2.`, so the expression returns a float.

   The operator raises a float to a power; $3.^{2.} = 9$.

2. `let f x = if x then "true" else 42 in f (3 = 4)`

   Solution
   This expression is ill-typed. As OCaml is strongly typed, the function `f` can't return more than one type. Here, `f` may return a value of type `string` or a value of type `int`, so `f` is not well typed.

3. `fun x -> let term = 7.3 in x +.  term`

   Solution

   Type : float -> float

   Value: `fun x -> let term = 7.3 in x +.  term`

   We know that `term` is a float. We can tell that `x` is a float because it's added to `term`. The function body returns `x +.  term` which is of type `float`.

   The function just evaluates to itself because functions are values in `OCaml`.

4. `let f x y = (x + y, x * y) in f 3 7`

   Solution
   Type: `int * int`

Value : `(10, 21)`.

We know `x` and `y` are of type `int` due to the use of the `+` and `*` operator. `f` is thus a function of type `int -> int -> (int * int)`. Since `f` is applied on two arguments, the expression is of type `int * int`.

Problem 1.2 (Infix Operators)**.** Define an infix operator `+/.` that computes the average of two-floating point numbers. For example,

```
1.0 +/. 2.0 = 1.5
3.5 +/. 0.0 +/. 5.5 = 3.625
```

Solution

```
let (+/.) (x : float) (y : float) : float =
  (x +. y) /. 2. ;;
```

We can define an infix operator `op` by using the syntax `(op)` in the function definition.

Problem 1.3 (Fibonacci)**.** Define a function `fib` such that `fib n` is the `n`th number in the Fibonacci sequence, which is 1, 1, 2, 3, 5, 8, 13...

We can write the Fibonacci as a **recurrence**, a mathematical equation that states how the `n`th term of a sequence is equal to some combination of the previous terms.

`fib 1 = 1`
`fib 2 = 1`
`fib n = fib (n-1) + fib (n-2)` for `n > 2`.

Solution

```
let rec fib (n : int) : int =
 if n = 1 || n = 2 then 1
 else fib (n - 1) + fib (n - 2)
```

The Fibonacci sequence is defined such that the first two numbers in the sequence are 1 and subsequent numbers are the sum of the two previous numbers in the sequence. The solution follows directly from the recurrence. We can also use the framework proposed in **Section 2.2** (How to Think about Recursion).
First considering the base cases, the first two numbers in the sequence are equal to 1 so we check if the input is for the 1st or 2nd Fibonacci number. Then, in the recursive step, **assume** that we can use `fib` as a helper function and use it to compute the previous two Fibonacci numbers and them together.

Problem 1.4 (Subset Sum). Define a function `subset_sum` that checks if there exists a subset of a list with a given sum (called the *target*).

```
subset_sum 6 [1; 4; 3; 3; 2] = true
subset_sum 20 [4;5;2;7] = false
```

Solution

```
let rec subset_sum (target : int) (lst : int list) : bool =
  match lst with
  | [] -> target == 0
  | h :: t -> subset_sum target t || subset_sum (target - h) t  ;;
```

If a list is empty, `subset_sum` should clearly return false.

A technique to come up with the recursion is to **consider exhaustive cases**. For example, here consider a subset that adds up to the target value. Either the first element of the list is in the subset or not in the subset.

If we exclude the head from the subset, then there must be a subset of the tail of the list adding up to target. Following the tips in section 2.2, in our recursive step, we assume that `subset_sum` correctly solves smaller instances. Hence, we can call `subset_sum target t`.

If we include the head in the subset, then the rest of the subset contains elements in the tail of the list. Those elements must add up to `target - h`. Thus, we can call `subset (target - h) t`.

Problem 1.5 (Partition). Define a function `partition` that divides a list into two lists, such that the first list contains elements that satisfy a predicate function `f` and the second list contains elements that don't satisfy the predicate. You may assume the input list is an integer list.

```
# partition ((>) 3) [4;8;2;-3;5;9]
- : int list * int list =
([2;-3], [[4;8;5;9])
```

Solution

```
let rec partition (pred : int -> bool) (lst : int list) : int list *
   int list =
  match lst with
  | [] -> [], []
  | h :: t ->
    let y, n = partition pred t in
    if pred h then h :: y, n
    else y, h :: n ;;
```

If the list is empty, then `partition` should just return a tuple of empty lists.

Now, if we consider our recursive step; we try calling `partition` on the tail of the list. **Treating the recursive call as correct**, `partition` gives us a pair of lists `(y, n)` where `y` contains elements of the tail that satisfy the predicate and `n` contains elements that don't satisfy the predicate.

Now, we can consider the head of the list. If the head satisfies the predicate, we should add it to `y`, otherwise we should add it to `n`.

Problem 1.6 (Pack Consecutive Duplicates)**.** Define a function `pack` that packs consecutive duplicates of list elements into sublists. You may assume that the input lists are string lists.

```
# pack ["a"; "a"; "a"; "a"; "b"; "c"; "c"; "a"; "a"; "d"; "d"; "e"; "e"
   ; "e"; "e"];;
- : string list list =
[["a"; "a"; "a"; "a"]; ["b"]; ["c"; "c"]; ["a"; "a"]; ["d"; "d"];
 ["e"; "e"; "e"; "e"]]
```

Solution

```
let rec pack (lst : string list) : string list list =
  match lst with
  | [] -> []
  | h :: t ->
    match pack t with
    | [] -> [[h]]
    | hd :: tl as l ->
        match hd with
        | [] -> []
        | h1 :: _ ->
          if h1 = h then (h :: hd) :: tl
          else [h] :: l ;;
```

If the input list is an empty list, we just return the empty list.

In our recursive step, we can consider what happens when we call `pack t`. Thinking recursively, we assume `pack t` is correct. On line 6, `pack t` only returns the empty list, if the list has one element, and in that case `pack` should just return a list containing a list with one element. Here, we'll consider exhaustive cases.

Overall, we classify lists into two types:

1. Lists where the first element and the second element are the same.
2. Lists where the first element and second element are different.

A list of type 1 would be `["a"; "a"; "a"; "a"; "b"]`. Here, `pack` on the tail of the list returns `[["a", "a", "a"]; ["b"]]`.

In case 1, the head of the input list is equal to the head of the first element returned from `pack t`. The output should be `[["a", "a", "a", "a"]; ["b"]]`. Thus, all we have to do is add the first element of the list to the first element of `pack t`, and then cons that to the tail of `pack t`.

A list of type 2 would look like, `["a"; "b"; "b"; "b"]`. `pack t` returns `[["b"; "b"; "b"]`. `pack` on this list should return `[["a"]; ["b"; "b"; "b"]]`. Thus, we can simply append `[h]` to `pack t` (on line 7, we use the `as` keyword to name `hd :: tl = pack t` as `l`).

Problem 1.7 (Unit Testing)**.** Write a suite of unit tests for `fib`, `subset_sum`, `pack`, and `partition`.

Problem 1.8 (Chess)**.** In chess, we move pieces across a 8x8 board such that the columns are labeled as letters and the rows are labeled as numbers.

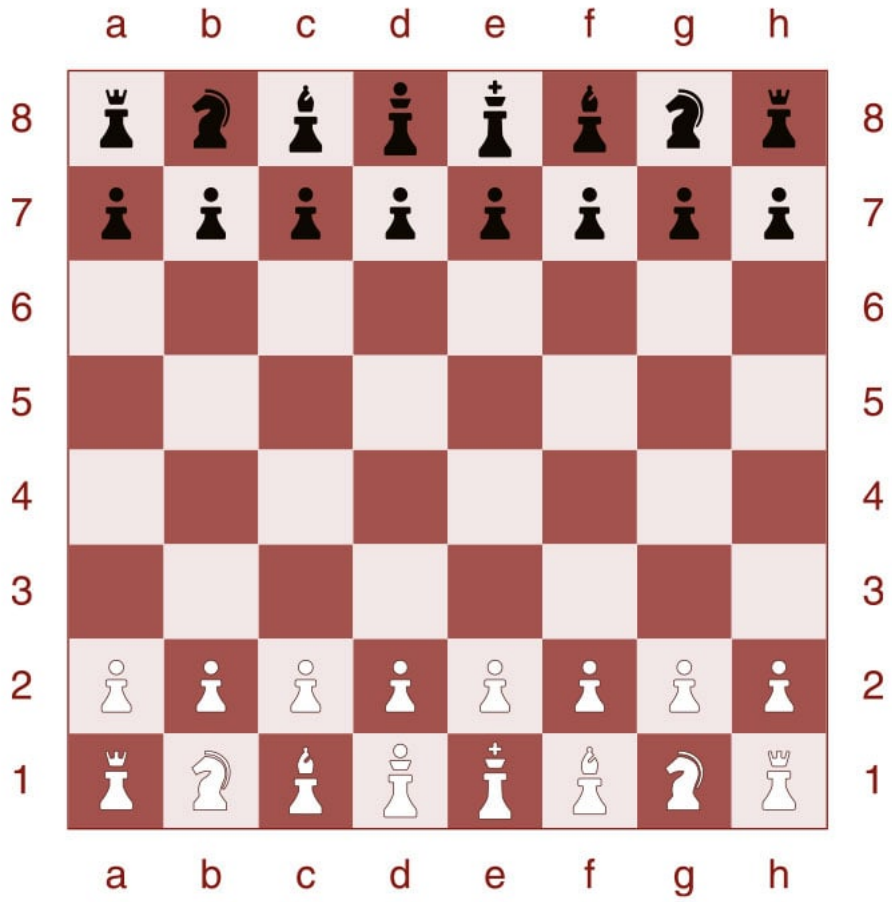

Figure 1: Diagram of chess board

Positions on a chess board are referred to as a combination of the column letter and row number. For instance, for the white pieces, we can say that the center pawn on the white square is at "e2".

1. Suppose, we want to define a type `position` that represents the position of a piece on the

board. What type expression is appropriate to represent a piece's position?

Solution
We can represent a position as a pair with a char and an int.

```
type position = char * int
```

2. After defining `position`, define a function `count_position` that takes in a `position` and a `position list`, and counts the number of times the `position` occurs in the `position list`.

Solution

```
let rec count_position (pos : position) (lst : position list) :
   int =
    match lst with
    | [] -> 0
    | h :: t -> (if h = pos then 1 else 0) + count_position pos t
        ;;
```

As the base case, if the list is empty, we return 0. In the recursive step, we assume `count_position` correctly counts the number of times `pos` appears in `t`. The total count should then be `count_position` or `1 + count_position`, depending on whether `h` is equal to `pos`.

3. A knight in chess can move two squares vertically and one square horizontally, or two squares horizontally and one square vertically (in an "L" shape). Define a function `legal_knight_moves` that takes in a `position` and returns a `position list` representing all the possible positions the knight can move to. *Hint*: You may find the `Char` module helpful here.

Solution

```
let legal_knight_moves ((c, r) : position) : position list =
  let possible_moves =
    [(2, 1); (2, -1); (-2, 1); (-2, -1); (1,2); (-1, 2); (-1, -2)
       ; (1, -2)] in
  let in_bounds ((c, i) : position) : bool =
    c >= 'a' && c <= 'h' && i >= 1 && i <= 8 in
  let rec aux (lst: (int * int) list) : position list =
    match lst with
    | [] -> []
    | (h, v) :: t ->
      let pros_move = (Char.chr (Char.code c + h), r + v) in
      if in_bounds pros_move then pros_move :: aux t else aux t
    in aux possible_moves ;;
```

We start by defining an `int * int` array `possible_moves` that represents the movements the knight can make in the horizontal and vertical directions. `in_bounds` is a helper function that takes in a position and returns `true` if the position is valid (within the bounds of the 8x8 board) and `false` otherwise.
The auxiliary function `aux` takes in a list movements and returns—given the position `(c, r)`—a list of position that the piece ends up in after a valid move (i.e. moves the piece to a valid square). For the base case, we naturally return the empty list if their are no moves to consider. In the recursive step, `pros_move` represents the position we end up in after making the move `(h, v)` (`h` squares up and `v` squares right) with the knight (with help from the `Char` module to convert shift letters). Then, if the position is within the bounds of the board (i.e. the move was valid) we append it to the possible position the knight can move to. If not, we move on and consider the remaining moves in the list. The `aux` is finally called on `possible_moves`, which exhaustively represents the movements the knight can make.

4. Define a function `min_knight_steps` that takes in a starting `position s` and a target `position t`, and returns the minimum number of moves it takes for a knight to go from `s` or `t`. Note, that a knight can visit any position on the board from any starting position, so there's always a solution! See the Knight's tour. You may use the `List.map` function here.

Solution

```
let rec min_knight_steps (s : position) (t : position) : int =
  let rec min_list (lst : int list) : int =
    match lst with
    | [] -> -1
    | [elt] -> elt
    | h :: t -> min h (min_list t) in
  if s = t then 0
  else 1 + min_list (List.map (fun x -> min_knight_steps x t) (
     legal_knight_moves s)) ;;
```

`min_list` is a helper function that returns the minimum element in a list. If the list is empty, we return -1 (it would be more appropriate to return an error, but this will be covered in next week's code review). If the list has a single element, then the minimum element should be the only element. Otherwise, in the recursive step, we assume `min_list` is correct. The minimum of the list can be thought of as the minimum between the head of the list and the minimum from the remaining elements.

Now, given this helper function, we can consider `min_knight_steps`. As a base case, if the starting position is already equal to the target position, then we need to make no moves, so we can return 0. Now in the recursive step, we again make the assumption that `min_knight_steps` is correct. If $s \neq t$, we need to make at least one move (hence the `1 +`. After making this move, we'll end up in some state.

We can then consider the possible states s' we end up in, and out of these states, we find the s^* that leads to the shortest path to t. From part 3, we can use our

function `legal_knight_moves` and call it on s to get a list of possible states, call `min_knight_steps` from each state, and consider the state that yields the minimum number of moves to t. The total number of moves to make is then just `1 + min_steps s t`.