\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

LAB 04 QUESTIONS

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

- Name: (FILL THIS in)

- NetID: (THE kauf0095 IN kauf0095@umn.edu)

Answer the questions below according to the lab specification. Write

your answers directly in this text file and submit it to complete the

lab.

PROBLEM 1: Pairs and Association Lists

======================================

Tuples allow multiple data of different types to be stored

together. OCaml allows arbitrarily large tuples but in practice one

works mostly with pairs (2-tuples) and triples (3-tuples).

This problem explores \*Association List\* to map keys to values; these

make extensive use of pairs (2-tuples). They are simple means of

implementing the idea of a "map" from keys to values. In many

functional programming contexts, the are implemented simply as an

unordered list of pairs. This gives them linear complexity for

operations like lookup. For small collections of key/values this is

reasonable but as the collection goes large, one typically switches to

using a sorted tree (logarithmic operations) or hash tables (amortized

constant operations).

(A)

~~~

Examine the first function in `assoc\_lists.ml' which is called

`assoc'. It operates on a key and an association list like the two

provided as `alist1' and `alist2'.

,----

| let rec assoc query\_key alist =

| match alist with

| | [] -> raise Not\_found

| | (key,value)::tail when query\_key=key -> value

| | \_::tail -> assoc query\_key tail

| ;;

|

| let alist1 = [(9,"nine"); (5,"five"); (2,"two");]

| let alist2 = [("nine",3.0); ("five",2.24); ("two",1.41); ("six",2.45)];;

`----

In a REPL, load this code and run the function on the following

inputs.

- 9 and alist1

- 2 and alist1

- 7 and alist1

- "six" and alist2

- "one" and alist2

- 1 and alist2

Show your results below and explain what the function is doing. Also

explain any errors for the above inputs and why the are happening.

**Solution :solution:**

**--------**

**`assoc' returns the value associated with the specified key. A**

**`Not\_found' exception is raised if `key' is not associated with**

**anything.**

**,----**

**| # #use "assoc\_lists.ml";;**

**| val assoc : 'a -> ('a \* 'b) list -> 'b = <fun>**

**| val alist1 : (int \* string) list = [(9, "nine"); (5, "five"); (2, "two")]**

**| val alist2 : (string \* float) list = [("nine", 3.); ("five", 2.24); ("two", 1.41); ("six", 2.45)]**

**|**

**| ...**

**|**

**| # assoc 9 alist1;; (\* looks up value associated with key 9 \*)**

**| - : string = "nine"**

**| # assoc 2 alist1;; (\* .. with key 2 \*)**

**| - : string = "two"**

**| # assoc 7 alist1;; (\* .. with key 7 : not found so exception \*)**

**| Exception: Not\_found.**

**| # assoc "six" alist2;; (\* with key "six" \*)**

**| - : float = 2.45**

**| # assoc "one" alist2;; (\* with key "one" : not found so exception \*)**

**| Exception: Not\_found.**

**| # assoc 1 alist2;; (\* type mismatch: alist2 has string \* float elements \*)**

**| Characters 8-14: (\* key must be type string, not int \*)**

**| assoc 1 alist2;;**

**| ^^^^^^**

**| Error: This expression has type (string \* float) list**

**| but an expression was expected of type (int \* 'a) list**

**| Type string is not compatible with type int**

**`----**

(B)

~~~

Examine the provided function `add\_assoc' which adds an key/val pair

to an association list if the key is not present or modifies the value

associated with an existing key.

,----

| 1 (\* return a list with the given key/value added; if the key already

| 2 exists, changes association to the new value \*)

| 3 let rec add\_assoc key value alist =

| 4 match alist with

| 5 | [] -> (key,value)::[]

| 6 | (k,v) :: tail when key=k -> (key,value)::tail

| 7 | (k,v) :: tail -> (k,v) :: (add\_assoc key value tail)

| 8 ;;

`----

You may wish to run this function on some inputs like

,----

| # add\_assoc 7 "seven" alist1;;

`----

to get acquainted with its operation.

Describe how this function works. Include the following in your

answer.

- Identify each case in the match/with statement as either a base case

or a recursive case.

- What situations does each case of the pattern matching expression

handle?

- How is pattern matching used to handle pairs in the list?

**Solution :solution:**

**--------**

**Below cases are commented with base/recursive and the situation they**

**handle. Pattern matching is used to decompose list pairs into their**

**key/value parts as in `(k,v) :: tail' where `k' is bound to the key**

**and `v' to value in the pairs.**

**,----**

**| 1 (\* return a list with the given key/value added; if the key already**

**| 2 exists, changes association to the new value \*)**

**| 3 let rec add\_assoc key value alist =**

**| 4 match alist with**

**| 5 | [] -> (key,value)::[] (\* base: not present, cons on \*)**

**| 6 | (k,v) :: tail when key=k -> (key,value)::tail (\* base: replace existing association \*)**

**| 7 | (k,v) :: tail -> (k,v) :: (add\_assoc key value tail) (\* recurse deeper \*)**

**| 8 ;;**

**`----**

(C)

~~~

At the bottom of `assoc\_lists.ml' is a commented declaration for the

`remove\_assoc key alist' function. As the comment indicates, this

should remove any existing association from a given list.

,----

| 1 (\* return a list with the given key and associated value removed; if

| 2 the given key is not present, no change is made to the list. Does

| 3 not raise exceptions. \*)

| 4 (\* let rec remove\_assoc key alist = \*)

`----

Complete this function. Use a similar code structure that that which

is used in `add\_assoc'. Demonstrate that the function works correctly

in a REPL by removing some associations from `alist1' and `alist2'

**Solution :solution:**

**--------**

**,----**

**| 1 (\* return a list with the given key and associated value removed; if**

**| 2 the given key is not present, no change is made to the list. Does**

**| 3 not raise exceptions. \*)**

**| 4 let rec remove\_assoc key alist =**

**| 5 match alist with**

**| 6 | [] -> []**

**| 7 | (k,v) :: tail when key=k -> tail**

**| 8 | (k,v) :: tail -> (k,v) :: (remove\_assoc key tail)**

**| 9 ;;**

**`----**

REPL Demo

,----

| # #use "assoc\_lists.ml";;

| ...

| val alist1 : (int \* string) list = [(9, "nine"); (5, "five"); (2, "two")]

| val alist2 : (string \* float) list = [("nine", 3.); ("five", 2.24); ("two", 1.41); ("six", 2.45)]

| val add\_assoc : 'a -> 'b -> ('a \* 'b) list -> ('a \* 'b) list = <fun>

| val remove\_assoc : 'a -> ('a \* 'b) list -> ('a \* 'b) list = <fun>

|

| # remove\_assoc "nine" alist2;;

| - : (string \* float) list = [("five", 2.24); ("two", 1.41); ("six", 2.45)]

| # remove\_assoc "two" alist2;;

| - : (string \* float) list = [("nine", 3.); ("five", 2.24); ("six", 2.45)]

| # remove\_assoc "seven" alist2;;

| - : (string \* float) list = [("nine", 3.); ("five", 2.24); ("two", 1.41); ("six", 2.45)]

| # remove\_assoc 2 alist1;;

| - : (int \* string) list = [(9, "nine"); (5, "five")]

`----

Note: Built-in Association Lists

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

OCaml has functions operating on Association Lists built in. They are

available in the `List' module and can be called as follows.

,----

| # let alist = [(9, "nine"); (5, "five"); (2, "two")];;

| val alist : (int \* string) list = [(9, "nine"); (5, "five"); (2, "two")]

|

| # List.assoc 2 alist;;

| - : string = "two"

|

| # List.assoc 6 alist;;

| Exception: Not\_found.

|

| # List.remove\_assoc 5 alist;;

| - : (int \* string) list = [(9, "nine"); (2, "two")]

`----

More details are in the module documentation for list here:

[https://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html#1\_Associationlists]

Note that the semantics for standard association lists are somewhat

different than those we implemented here: keys may be duplicated and

"left-most" bindings are returned. This makes adding on more efficient

at the expense of memory and removal complexity.

PROBLEM 2: Records

==================

Records are types defined to have multiple named \*fields\* of different

kinds. These are similar to C's structs and Java's objects. This

problem explores basic record definition and use.

(A)

~~~

Examine the provided source file `record\_use.ml'. The first few lines

of this file contain a type declaration for a new record type called

`force\_user'.

,----

| 1 (\* record type declaration \*)

| 2 type force\_user = {

| 3 name : string; (\* field 1 \*)

| 4 darkside : bool; (\* field 2 \*)

| 5 episodes : int list; (\* field 3 \*)

| 6 };;

`----

Below it are several examples of creating `force\_user' records. Make

use of this pattern to declare some additional `force\_user' records

and paste your new code below. Demonstrate that your code works

properly by loading it in a REPL.

**Solution :solution:**

**--------**

**,----**

**| 1 let luke = { (\* force\_user val \*)**

**| 2 name="Luke Skywalker";**

**| 3 darkside=false;**

**| 4 episodes=[3;4;5;6;7;8];**

**| 5 };;**

**| 6**

**| 7 let sidious = { (\* force\_user val \*)**

**| 8 name="Sheev Palpatine";**

**| 9 darkside=true;**

**| 10 episodes=[1;2;3;4;5;6];**

**| 11 };;**

**| 12**

**| 13 let rey = { (\* force\_user val \*)**

**| 14 name="Rey";**

**| 15 darkside=false;**

**| 16 episodes=[7;8];**

**| 17 };;**

**| 18**

**| 19 (\* create some new force\_user records below \*)**

**| 20 let vader = {**

**| 21 name="Anakin Skywalker";**

**| 22 darkside=true;**

**| 23 episodes=[1;2;3;4;5;6];**

**| 24 };;**

**| 25**

**| 26**

**| 27 let maul = { (\* force\_user val \*)**

**| 28 name="Maul";**

**| 29 darkside=true;**

**| 30 episodes=[1]**

**| 31 };;**

**| 32**

**| 33 let obi = {**

**| 34 name="Obi-Wan Kenobi";**

**| 35 darkside=false;**

**| 36 episodes=[1;2;3;4;5;6];**

**| 37 };;**

**`----**

(B)

~~~

Record fields are accessed with dot notation. This can be done in

individual statements or during execution of functions. The following

code segments from `record\_use.ml' show some examples.

,----

| 1 (\* field access \*)

| 2 let last\_jedi1 = luke.name;;

| 3 let last\_jedi2 = rey.name;;

| 4 let sith\_reigh = List.length sidious.episodes;;

| 5

| 6 (\* functions on records \*)

| 7 let name\_of user = (\* retrieve the name \*)

| 8 user.name

| 9 ;;

| 10

| 11 let episode\_count user = (\* count episodes \*)

| 12 List.length user.episodes

| 13 ;;

`----

Demonstrate in a REPL some field accesses on the data you have defined

like

- Accessing the darkside field

- Measuring the length of a name field with String.length

- Appending two episode field lists with @ (append) operator

**Solution :solution:**

**--------**

**,----**

**| # rey.name;;**

**| - : string = "Rey"**

**| # String.length rey.name;;**

**| - : int = 3**

**| # vader.darkside;;**

**| - : bool = true**

**| # vader.episodes @ rey.episodes;;**

**| - : int list = [1; 2; 3; 4; 5; 6; 7; 8]**

**`----**

(C)

~~~

Consider the function `seduced': it takes a `force\_user' as an

argument and creates a new version of it using the `with' syntax.

,----

| 1 let seduced user = (\* new record with field changed \*)

| 2 let dark\_user = {user with darkside=true} in

| 3 dark\_user

| 4 ;;

`----

Use a REPL to demonstrate how `seduced' works. Determine whether this

function actually changes the original record or not. Explain your

answer.

**Solution :solution:**

**--------**

**As the REPL following session demonstrates, records are not changed by**

**this function. The original values persist. However the new versions**

**can be bound to different or even the same names.**

**,----**

**| # seduced rey;;**

**| - : force\_user = {name = "Rey"; darkside = true; episodes = [7; 8]}**

**|**

**| # rey;;**

**| - : force\_user = {name = "Rey"; darkside = false; episodes = [7; 8]}**

**|**

**| # let dark\_luke = seduced luke;;**

**| val dark\_luke : force\_user =**

**| {name = "Luke Skywalker"; darkside = true; episodes = [3; 4; 5; 6; 7; 8]}**

**|**

**| # luke;;**

**| - : force\_user =**

**| {name = "Luke Skywalker"; darkside = false; episodes = [3; 4; 5; 6; 7; 8]}**

**`----**

(D)

~~~

Write a function `sequel\_appearance'. It takes a `force\_user' and an

integer as parameters. It creates a new `force\_user' record from the

old one with the integer parameter appended to the end of the

`episodes' field using the @ operator. Its type and use are

demonstrated below.

,----

| # #use "record\_use.ml";;

| val sequel\_appearance : force\_user -> int -> force\_user = <fun>

|

| # rey;;

| - : force\_user = {name = "Rey"; darkside = false; episodes = [7; 8]}

| # sequel\_appearance rey 9;;

| - : force\_user = {name = "Rey"; darkside = false; episodes = [7; 8; 9]}

|

| # vader;;

| - : force\_user =

| {name = "Anakin Skywalker"; darkside = true; episodes = [1; 2; 3; 4; 5; 6]}

| # sequel\_appearance vader 10;;

| - : force\_user =

| {name = "Anakin Skywalker"; darkside = true; episodes = [1; 2; 3; 4; 5; 6; 10]}

`----

**Solution :solution:**

**--------**

**,----**

**| 1 (\* create a new force\_user with given episode number appended to end of episodes field \*)**

**| 2 let sequel\_appearance user episode\_num =**

**| 3 let new\_episodes = user.episodes @ [episode\_num] in**

**| 4 let new\_user = {user with episodes=new\_episodes} in**

**| 5 new\_user**

**| 6 ;;**

**| 7**

**| 8 (\* succinct version \*)**

**| 9 let sequel\_appearance user episode\_num =**

**| 10 {user with episodes=user.episodes @ [episode\_num]}**

**| 11 ;;**

**`----**

PROBLEM 3: Feedback Survey

==========================

For full credit on this lab, complete the Midterm Feedback Survey

which is available on Canvas

- Click "Quizzes"

- Select "Midterm Feedback"

- The survey is Anonymous and graded only on completion