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

LAB 02 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: last\_elem recursive function

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

(A)

~~~

Start the OCaml REPL in the `lab02-code' directory and load the

`rec\_funcs.ml' source file with a `#use' directive as shown.

,----

| > ocaml

| OCaml version 4.06.0

|

| # #use "rec\_funcs.ml";;

| val last\_elem : 'a list -> 'a = <fun>

| val elems\_outside : int -> int -> 'a list -> 'a list = <fun>

| #

`----

The first function loaded from the source file is called `last\_elem'.

Describe in words the type of `last\_elem' :

- What kind of argument does it take?

- What type does it return?

- What does the 'a notation mean?

Solution :solution:

--------

**- `last\_elem' has type 'a list -> 'a**

**- This means it takes any kind of list as an argument**

**- The last element of that list is returned**

**- 'a is OCaml's way of saying "arbitrary type"**

(B)

~~~

Call `last\_elem' on several input lists in the REPL and show the

return values.

- Make sure to call the function on an empty list and report what

happens.

- Make sure to use several different types of lists (int list, string

list) and a few different lengths.

Paste your REPL transcript below AND describe in a line or two what

the function does.

**Solution :solution:**

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

**Returns the last element in a list. Raises an exception if the list is**

**empty.**

**,----**

**| # last\_elem [5;2;4;9];;**

**| - : int = 9**

**| # last\_elem [5;2;4;9;1;6;8];;**

**| - : int = 8**

**| # last\_elem ["hello"];;**

**| - : string = "hello"**

**| # last\_elem [];;**

**| Exception: Failure "No last element in an empty list".**

**`----**

(C)

~~~

Examine the definition of `last\_elem' in `rec\_funcs.ml'. Study how it

operates carefully to gain insight on recursive functions.

Add comments in your own words that describe line-by-line how the

function operates. Paste your commented version of the code below.

**Solution :solution:**

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

**,----**

**| (\* Returns the last element in a list. Raises an exception if the list**

**| is empty. \*)**

**| let rec last\_elem list =**

**| if list = [] then (\* error case of empty list, raise exception \*)**

**| raise (Failure "No last element in an empty list")**

**| else (\* non-empty list \*)**

**| let elem = List.hd list in (\* peel off the first element \*)**

**| let rest = List.tl list in (\* get the rest of the list \*)**

**| if rest = [] then (\* if the rest is empty \*)**

**| elem (\* the elem is the last, return it \*)**

**| else**

**| last\_elem rest (\* otherwise recurse on remaining list \*)**

**| ;;**

**`----**

PROBLEM 2: elems\_outside with recursive helper function

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

(A)

~~~

The other function loaded by `rec\_funcs.ml' is called `elems\_outside'.

As before, describe it's type.

- How many parameters does it take and what type are they?

- What type does it return?

**Solution :solution:**

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

**- `elems\_outside' has type**

**,----**

**| int -> int -> 'a list -> 'a list**

**`----**

**- This means it takes 3 parameters**

**1. an int**

**2. another int**

**3. any type of list**

**- A list of the same type as parameter 3 is returned**

(B)

~~~

In the REPL, call `elems\_outside' on several lengths and types of

lists. Show the results generated by these calls by pasting your REPL

session below. Describe what the function appears to be doing.

**Solution :solution:**

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

**Returns elements in a list outside of the given indices**

**start/stop. Elements at start/stop are not included in the result.**

**,----**

**| # elems\_outside 3 5 [0; 1; 2; 3; 4; 5; 6; 7];;**

**| - : int list = [0; 1; 2; 6; 7]**

**| # elems\_outside 1 5 [0; 1; 2; 3; 4; 5; 6; 7];;**

**| - : int list = [0; 6; 7]**

**| # elems\_outside 2 4 [0; 1; 2; 3; 4; 5; 6; 7];;**

**| - : int list = [0; 1; 5; 6; 7]**

**| # elems\_outside 2 4 [];;**

**| - : 'a list = []**

**`----**

(C)

~~~

Examine the definition of `elems\_outside' in `rec\_funcs.ml'. Study how

it operates carefully to gain insight on recursive functions.

Add comments in your own words that describe line-by-line how the

function operates. Paste your commented version of the code below.

**Solution :solution:**

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

**,----**

**| (\* Returns elements in a list outside of the given indices**

**| start/stop. Elements at start/stop are not included in the**

**| result. Uses an internal recursive helper method. \*)**

**| let elems\_outside start stop list =**

**| let rec helper pos lst = (\* recursive helper \*)**

**| if lst=[] then (\* check for end of list/empty \*)**

**| [] (\* return empty \*)**

**| else if start<=pos && pos<=stop then (\* between start/stop \*)**

**| helper (pos+1) (List.tl lst) (\* recurse but don't cons on any parts \*)**

**| else (\* outside start/stop \*)**

**| let elem = List.hd lst in (\* peel off first element \*)**

**| let rest = List.tl lst in (\* get rest of list \*)**

**| let result = helper (pos+1) rest in (\* recurse on remainder of list \*)**

**| elem :: result (\* cons on current element \*)**

**| in (\* end helper \*)**

**| helper 0 list (\* call helper at beginning of list \*)**

**| ;;**

**`----**

Problem 3

=========

(A)

~~~

Examine the two functions provided in `sorting.ml': `sorted\_insert'

and `sort'. Describe the parameter and return types for both

functions. Based on the comments and source code, determine their

purpose.

**Solution :solution:**

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

**sorted\_insert**

**- type: 'a -> 'a list -> 'a list**

**- inserts an element into a sorted list maintaining sorted order**

**sort**

**- type: 'a list -> 'a list**

**- sorts a list via repeated insertion**

(B)

~~~

The code provided for this problem is an adaptation of code from the

Ocaml System Manual Section 1.2 which is here

[http://caml.inria.fr/pub/docs/manual-ocaml/coreexamples.html#sec9]

The original version looks like this:

,----

| let rec sort lst =

| match lst with

| [] -> []

| | head :: tail -> insert head (sort tail)

| and insert elt lst =

| match lst with

| [] -> [elt]

| | head :: tail -> if elt <= head then elt :: lst else head :: insert elt tail

| ;;

`----

The version in `sorting.ml' has been re-written so that it is somewhat

more verbose but potentially easier for a novice to understand.

- Ordering of the functions is reversed so that insertion is defined

prior to sorting

- Two separate "let" bindings are used rather than a joint "let/and"

binding

- Pattern matching via "match" is replaced with "if/else" statements

- Destructure binding is replaced with explicit calls to List.hd and

List.tl

All of the above concepts will eventually be covered and it does not

hurt one to look ahead a bit.

COMPARE the code in `sorting.ml' to the original OSM version above.

Make some observations about how the syntax associated with the

"match" statement must work.

**Solution :solution:**

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

**Pattern matching combines if/else and function calls to disassemble**

**data structures. The pattern matching shown selects code to execute**

**based on the structure of the list while also creating bindings for**

**names like "head" which would normally require a call to List.hd.**

**This can shorten code somewhat when case analysis is required.**