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

LAB 07 QUESTIONS ANSWERS

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

- 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: Higher-order Practice and Currying

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

The file `nested\_lists.ml' contains two nested lists,

- strll : string list list

- intll : int list list

Several functions are described in this file that operate on such

nested lists. Fill in their definitions. All of them involve

application of appropriate higher-order functions on lists. Some of

them may also benefit from partial application of curried functions.

You may freely alter prototypes such as adding/removing arguments so

long as the resulting functions work as indicated in the examples.

SOLUTION :solution:

--------

,----

| (\* nested\_lists.ml: Define some functions on nested lists (e.g. int

| list list and string list list) using higher-order functions. \*)

|

| open Printf;;

|

| (\* sample string list list \*)

| let strll = [

| ["Korra";"Mako";"Bolin";"Asami";];

| ["Tenzin";"Pema"];

| ["Meelo";"Jinora";"Iki"];

| ["Amon";"Kuvira";"Zaheer"];

| ];;

|

| (\* sample int list list \*)

| let intll = [

| [1;2;3];

| [4;5;6];

| [7;8;9;10];

| [11];

| ];;

|

| (\* val flatten : 'a list list -> 'a list

|

| Converts a list of lists to a single "flat" list. Each list is

| appended onto the last. Since this function is polymorphic, no

| special versions are needed for different types of list. Makes use

| of List.fold\_left. This function is equivalent to the standard

| List.flatten function built into ocaml.

|

| # flatten intll;;

| - : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11]

| # flatten strll;;

| - : string list =

| ["Korra"; "Mako"; "Bolin"; "Asami"; "Tenzin"; "Pema"; "Meelo"; "Jinora";

| "Iki"; "Amon"; "Kuvira"; "Zaheer"]

| \*)

**| let flatten list\_list =**

**| List.fold\_left (@) [] list\_list**

**| ;;**

|

| (\* curried version \*)

| let flatten =

| List.fold\_left (@) []

| ;;

|

| (\* val totlen : 'a list list -> int

|

| Calculates the total length of all combined lists using fold\_left.

| Does not use flatten but does use List.length.

|

| # totlen intll;;

| - : int = 11

| # totlen strll;;

| - : int = 12

| \*)

**| let totlen list\_list =**

**| let addlen tot list =**

**| tot + (List.length list)**

**| in**

**| List.fold\_left addlen 0 list\_list**

**| ;;**

|

| (\* Terse curried version \*)

| let totlen =

| List.fold\_left (fun t l->t+(List.length l)) 0

| ;;

|

| (\* val print\_list\_list : ('a -> unit) -> 'a list list -> unit

|

| Print all lists in a list. argument print\_elem is a function that

| prints the a single element of any list. Each list is printed on

| its own line starting with an open square brace [ and ending with a

| clsoe square brace ]. See specific output for print\_str\_list\_list

| and print int\_list\_list below which both use this function. \*)

| let print\_list\_list print\_elem listlist =

| let print\_list list =

| printf "[";

| List.iter print\_elem list;

| printf "]\n";

| in

| List.iter print\_list listlist

| ;;

|

| (\* val print\_str\_list\_list : string list list -> unit

|

| Print all string lists in a list. Each string is printed preceded

| by a space. Otherwise the conventions of print\_list\_list are used.

|

| # print\_str\_list\_list strll;;

| [ Korra Mako Bolin Asami]

| [ Tenzin Pema]

| [ Meelo Jinora Iki]

| [ Amon Kuvira Zaheer]

| - : unit = ()

| \*)

| let print\_str\_list\_list =

| print\_list\_list (printf " %s")

| ;;

|

| (\* val print\_int\_list\_list : int list list -> unit

|

| Print all int lists in a list. Each integer is printed preceded by

| a space. Otherwise the conventions of print\_list\_list are used.

|

| # print\_int\_list\_list intll;;

| [ 1 2 3]

| [ 4 5 6]

| [ 7 8 9 10]

| [ 11]

| - : unit = ()

| \*)

| let print\_int\_list\_list =

| print\_list\_list (printf " %d")

| ;;

`----

PROBLEM 2: Curry Trouble

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

The provided file `curry\_trouble.ml' is intended to compile and run as

follows.

,----

| > ocamlc curry\_trouble.ml

| > ./a.out

| usage: ./a.out base start stop

|

| > ./a.out 2 1 10

| 2^1 is 2

| 2^2 is 4

| 2^3 is 8

| 2^4 is 16

| 2^5 is 32

| 2^6 is 64

| 2^7 is 128

| 2^8 is 256

| 2^9 is 512

| 2^10 is 1024

|

| > ./a.out 3 4 8

| 3^4 is 81

| 3^5 is 243

| 3^6 is 729

| 3^7 is 2187

| 3^8 is 6561

`----

Unfortunately, `curry\_trouble.ml' currently has an error in it which

prevents it from being compiled. OCaml's automatic function currying

makes this error somewhat more obscure than it might otherwise

be. This problem explores this issue to practice debugging type

errors.

,----

| (\* curry\_trouble.ml: Debug the following code which has a compile time

| error due to a partial application. \*)

|

| open Printf;;

|

| (\* raise base to given exp \*)

| let pow base exp =

| let ans = ref 1 in

| for i=1 to exp do

| ans := !ans \* base;

| done;

| !ans

| ;;

|

| (\* print successive powers \*)

| let print\_powers base start stop =

| for i=start to stop do

| let x = pow base in

| printf "%d^%d is %d\n" base i x;

| done;

| ;;

|

| (\* main function \*)

| let \_ =

| if Array.length Sys.argv < 4 then

| begin

| printf "usage: %s base start stop\n" Sys.argv.(0);

| exit 1;

| end;

| let base = int\_of\_string Sys.argv.(1) in

| let start = int\_of\_string Sys.argv.(2) in

| let stop = int\_of\_string Sys.argv.(3) in

| print\_powers base start stop;

| ;;

`----

(A)

~~~

Compile the `curry\_trouble.ml' as shown above and paste the compile

error that results below.

SOLUTION :solution:

--------

,----

| > ocamlc curry\_trouble.ml

| File "curry\_trouble.ml", line 19, characters 34-35:

| Error: This expression has type int -> int

| but an expression was expected of type int

`----

(B)

~~~

The error message proclaims that this is a type problem. This is not

wrong, just misleading.

Spend some time examining the code and correct the error. Describe

what the true problem is and show below the line(s) which need to be

changed to produce a working version of `curry\_trouble.ml'.

SOLUTION :solution:

--------

While reported as a type problem, the nature of the error is that in

line 18 in the `print\_powers' function, the `pow' function is called

with only one argument rather than two. This can be fixed by modifying

line 18 to pass another argument to `pow':

,----

| let print\_powers base start stop =

| for i=start to stop do

| let x = pow base i in (\* added argument i \*)

| printf "%d^%d is %d\n" base i x;

| done;

| ;;

`----

(C)

~~~

Describe why OCaml reports a type error for the original code on a

different line from where the error actually occurs. Relate your

discussion to curried functions and partial applications.

SOLUTION :solution:

--------

`pow' is a curried function of type `int -> int -> int' so can be

partially applied. Since an argument was left off of the call to

`pow', the result is another function of `int -> int'. This is bound

to `x' but cannot be printed in the context of the following line

where just an `int' is expected.

(D)

~~~

Once you have identified the problem with `curry\_trouble.ml', consider

the equivalent Java program that is provided in `Uncurried.java'. The

mistake that is made in this file is identical. Compile it as follows

and show the error message given. Describe whether you feel this error

message is more or less indicative of the underlying problem and why.

,----

| > javac Uncurried.java

`----

SOLUTION :solution:

--------

,----

| > javac Uncurried.java

| Uncurried.java:12: error: method pow in class Uncurried cannot be applied to given types;

| int x = pow(base);

| ^

| required: int,int

| found: int

| reason: actual and formal argument lists differ in length

| 1 error

`----

The error message is directly related the problem which is that too

few arguments are passed to `pow'. Since Java does not curry its

functions, there is no ambiguity as to whether the programmers intent

on line 12 is partial or full application: partial application is not

possible in Java so the line is a mistake.

(E)

~~~

Describe the cost associated with OCaml's automatic currying of

functions. Also describe if there is any way to avoid these problems

if curried functions are not needed: how would one enforce all

arguments be given together as a package in OCaml?

SOLUTION :solution:

--------

While more flexible in their use, curried functions can create

devilishly difficult type errors if they are inadvertently partially

applied.

If curried functions are not needed, OCaml functions can be written to

take tuple arguments instead. This enforces that all arguments be

given together such as in the following version of `pow':

,----

| (\* raise base to given exp, tuple argument enforces all args together \*)

| let pow (base,exp) =

| let ans = ref 1 in

| for i=1 to exp do

| ans := !ans \* base;

| done;

| !ans

| ;;

`----

PROBLEM 3: Objects... or Closures?

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

Object-oriented programming frequently features a syntax that looks

looks like the following:

,----

| My\_Object my\_object = new My\_Object(init1,init2);

| my\_object.some\_method(param1,param2);

`----

OCaml has an object system that works similarly to this which we will

discuss later. However, with the introduction of first-class

functions, we are already in a position to create an ad-hoc object

system that looks and behaves very similarly to the above

template. The problem explores the file `closure\_objects.ml' which

demonstrates this concept.

(A)

~~~

Examine the file `closure\_objects.ml' and describe

- What kind of "object" is defined

- What data is associated with instances of these objects

- What "methods" (operations) are supported for the data

SOLUTION :solution:

--------

- A `person' object is defined

- It has two pieces of mutable data, `name : string' and `age : int'

- Three operations are supported

- `birthday ()' which increases age

- `name\_change name' which alters the name

- `to\_string ()' which produces a string representation of the

person

(B)

~~~

Describe how to create an instance of the "objects" defined in

`closure\_object.ml'. What function is used, what arguments does it

take, and what type of thing does it return.

SOLUTION :solution:

--------

The `make\_person name age' function creates records of `person' type

by taking a string name and int age as the initial values for the

record's data.

(C)

~~~

Describe the syntax used to initiate the `birthday' method. Give an

example from the later "main" function. What type of thing is the

`birthday' field of each record bound to?

SOLUTION :solution:

--------

The method can be run via

,----

| person.birthday ();

`----

The field `birthday' is bound to a function which takes unit and

returns unit, incriminating the `age' field.

(D) OPTIONAL Enrichment

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

The `make\_person' uses an interesting technique that we have not

discussed. The binding starts with

,----

| let rec this = {

`----

and proceeds to use `and' bindings as in

,----

| and birthday\_func () =

| ...

| and name\_change\_func name =

`----

Note that the name `this' is NOT special in OCaml: it was chosen to

match the convention of C++/Java where `this' refers to the object

instance associated with a running method.

To this point, we have only seen `rec' associated with recursive

functions. Clearly, the `this' is neither recursive nor a function.

Neither are any of the functions associated with it recursive.

Make a copy of `closure\_objects.ml' and experiment eliminating the

`rec' and defining the record and functions separately with standard

`let/in' syntax. Describe your results.

Do some research on the purpose of `let rec/and' in OCaml and describe

its use case.

SOLUTION :solution:

--------

It is not possible to compile this code the same way using standard

`let/in' bindings. This is because `let/in' bindings must be

"sequential" in some sense:

- `let x = ... in' creates a binding for x

- `let y = ... x ... in' uses the binding for x to create a binding

for y

- etc.

The problem presented in `make\_person' is that one wants to do two

things simultaneously:

- Create a record `this' with fields bound to some functions like

`birthday\_func'

- Create some functions like `birthday\_func' which make use of the

record `this'

This cannot be accomplished via sequential `let/in' bindings.

For such cases, OCaml allows `let rec/and' bindings so that all names

being bound can refer to one another and are "simultaneously" defined

with awareness of one another. Such cases don't arise often but when

needed, they are still possible in OCaml.

The `let rec/and' construct is most often associated with mutually

recursive functions: function `foo' calls function `bar' while `bar'

in turn calls `foo'. However, the current case of data referring to

functions which refer to data is also covered by `let rec/and'.

For additional insanity, consider the following simple `let rec' which

defines a list consed onto itself.

,----

| # let rec a = "first" :: a;; (\* name a is available during definition of itself \*)

| val a : string list = ["first"; <cycle>]

| # List.hd a;;

| - : string = "first"

| # List.tl a;;

| - : string list = ["first"; <cycle>]

| # List.tl (List.tl a);;

| - : string list = ["first"; <cycle>]

| # List.tl (List.tl (List.tl a));;

| - : string list = ["first"; <cycle>]

| #

`----

Careful trying to use functions like `List.length' on such lists: they

will loop infinitely trying to find a tail that circles back on

itself.