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

LAB 03 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: concat\_all and match/with

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

(A)

~~~

Examine the code provided in `concat\_all.ml'

,----

| let rec concat\_all\_crap strlist =

| if strlist=[] then

| ""

| else

| let head = List.hd strlist in

| let tail = List.tl strlist in

| let rest = concat\_all\_crap tail in

| head ^ " " ^ rest

| ;;

|

| let rec concat\_all\_good strlist =

| if strlist=[] then

| ""

| else

| let head = List.hd strlist in

| let tail = List.tl strlist in

| if tail=[] then

| head

| else

| let rest = concat\_all\_good tail in

| head ^ " " ^ rest

| ;;

`----

Two functions are present which perform a similar operation but their

output varies subtly.

- Describe what the functions do.

- Execute both functions on the following inputs in a REPL and paste

the results below.

- []

- ["Fold-em"]

- ["Muh";"muh"]

- ["P"; "p"; "p"; "poker"; "face"]

- Describe the difference in return value between the two

functions. Pay particular attention to the ends of the strings.

**Solution :solution:**

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

**The two functions both concatenate all strings in a list together to**

**form a single string with each separated by spaces. The primary**

**difference is**

**- concat\_all\_crap always ends with a space**

**- concat\_all\_good does finer case analysis so the resulting string**

**doesn't end with a space**

**,----**

**| # #use "concat\_all.ml";;**

**| val concat\_all\_crap : string list -> string = <fun>**

**| val concat\_all\_good : string list -> string = <fun>**

**| # concat\_all\_crap [];;**

**| - : string = ""**

**| # concat\_all\_good [];;**

**| - : string = ""**

**| # concat\_all\_crap ["Fold-em"];;**

**| - : string = "Fold-em "**

**| # concat\_all\_good ["Fold-em"];;**

**| - : string = "Fold-em"**

**| # concat\_all\_crap ["Muh";"muh"];;**

**| - : string = "Muh muh "**

**| # concat\_all\_good ["Muh";"muh"];;**

**| - : string = "Muh muh"**

**| # concat\_all\_crap ["P"; "p"; "p"; "poker"; "face"];;**

**| - : string = "P p p poker face "**

**| # concat\_all\_good ["P"; "p"; "p"; "poker"; "face"];;**

**| - : string = "P p p poker face"**

**`----**

(B)

~~~

Create a function `concat\_all' which operates the same way that

`concat\_all\_good' does but uses PATTERN MATCHING via the match/with

construct. Paste your code below and the results of testing it in a

REPL.

**Solution :solution:**

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

**,----**

**| (\* pattern matching version: concatenate several string \*)**

**| let rec concat\_all strlist =**

**| match strlist with**

**| | [] -> ""**

**| | head :: [] -> head**

**| | head :: tail -> head ^ " " ^ (concat\_all tail)**

**| ;;**

**|**

**| (\* # concat\_all [];;**

**| - : string = ""**

**| # concat\_all ["Fold-em"];;**

**| - : string = "Fold-em"**

**| # concat\_all ["Muh";"muh"];;**

**| - : string = "Muh muh"**

**| # concat\_all ["P"; "p"; "p"; "poker"; "face"];;**

**| - : string = "P p p poker face"**

**| \*)**

**`----**

PROBLEM 2: colsum and the stack

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

(A)

~~~

Analyze the code in `colsum.ml'. It contains a function called

`colsum\_nt' which computes a certain sequence of numbers and sums the

sequence.

This file can be compiled and run as a program which will perform

`colsum\_nt 10' and print some intermediate and final results. Paste

the lines you use to compile and run the `colsum.ml' below as well as

the output for the program.

**Solution :solution:**

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

**,----**

**| > ocamlc colsum.ml**

**| > ./a.out**

**| 10**

**| 5**

**| 16**

**| 8**

**| 4**

**| 2**

**| 1**

**| sum: 46**

**`----**

(B)

~~~

Consider the source code for `colsum\_nt'

,----

| 1 let rec colsum\_nt n =

| 2 Printf.printf "%d\n" n;

| 3 if n = 1 then

| 4 1

| 5 else

| 6 let next =

| 7 if n mod 2 = 0

| 8 then n/2

| 9 else 3\*n+1

| 10 in

| 11 let rest = colsum\_nt next in

| 12 n + rest

| 13 ;;

| 14

| 15 let sum = colsum\_nt 10 in

| 16 Printf.printf "sum: %d\n" sum;

| 17 ;;

`----

The `colsum\_nt' function is NOT tail recursive. This means it builds a

sequence of activation records as it recurses at line 11 until it

reaches the base case at line 4. The initial call is `colsum\_nt 10'

and the first few frames of this sequence of activation records are

below. Add on to this table to show all Activation Records present

when line 4 is first reached in `colsum\_nt'.

----------------------

FRAME SYM VAL

----------------------

init ... ..

line:15 sum ??

----------------------

colsum\_nt n 10

line:11 next 5

rest ??

----------------------

colsum\_nt n 5

line:11 next 16

rest ??

----------------------

... ... ...

**Solution :solution:**

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

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

**FRAME SYM VAL**

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

**init ... ..**

**line:15 sum ??**

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

**colsum\_nt n 10**

**line:11 next 5**

**rest ??**

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

**colsum\_nt n 5**

**line:11 next 16**

**rest ??**

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

**colsum\_nt n 16**

**line:11 next 8**

**rest ??**

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

**colsum\_nt n 8**

**line:11 next 4**

**rest ??**

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

**colsum\_nt n 4**

**line:11 next 2**

**rest ??**

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

**colsum\_nt n 2**

**line:11 next 1**

**rest ??**

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

**colsum\_nt n 1**

**line:4 next ??**

**rest ??**

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

(C)

~~~

Programmers new to the idea of tail recursion may at times think

trivial changes code re-arrangements such as the one below will make a

function tail recursive. Notice in the `colsum\_alt' version below how

line 11 now has both the addition and recursive call.

,----

| 1 let rec colsum\_alt n =

| 2 Printf.printf "%d\n" n;

| 3 if n = 1 then

| 4 1

| 5 else

| 6 let next =

| 7 if n mod 2 = 0

| 8 then n/2

| 9 else 3\*n+1

| 10 in

| 11 n + (colsum\_alt next) (\* changed line \*)

| 12 ;;

`----

Explain why this version is still NOT tail recursive and in fact will

execute identically to the previous `colsum\_nt' version.

**Solution :solution:**

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

**The recursive call at line 11, `(colsum\_alt next)' must execute first**

**before a number can be established add onto `n'. Thus there will be**

**an unnamed temporary variable created by the compiler to hold the**

**results of the recursive call. The stack will grow down until a base**

**case is hit and these temporaries will get filled in as the stack**

**unwinds. Thus this version is identical to the `colsum\_nt' version**

**where the results are given a binding to name `rest'.**

(D)

~~~

Write a tail recursive version of `colsum' called `colsum\_tr'. A good

strategy for this is to use an internal helper function which takes

some additional parameters beyond single value. Paste your code below

and show that it works identically to `colsum\_nt'.

**Solution :solution:**

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

**,----**

**| > cat -n colsum\_answer.ml**

**| 1 (\* tail recursive version \*)**

**| 2 let rec colsum\_tr n =**

**| 3 let rec helper n sum =**

**| 4 Printf.printf "%d\n" n;**

**| 5 if n=1 then**

**| 6 sum+1**

**| 7 else**

**| 8 let next =**

**| 9 if n mod 2 = 0**

**| 10 then n/2**

**| 11 else 3\*n+1**

**| 12 in**

**| 13 helper next (sum+n)**

**| 14 in**

**| 15 helper n 0**

**| 16 ;;**

**| 17**

**| 18 let sum = colsum\_tr 10 in**

**| 19 Printf.printf "sum: %d\n" sum;**

**| 20 ;;**

**| > ocamlc colsum\_answer.ml**

**| > ./a.out**

**| 10**

**| 5**

**| 16**

**| 8**

**| 4**

**| 2**

**| 1**

**| sum: 46**

**`----**