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

LAB 13 QUESTIONS

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

- Name: Moti Begna

- NetID: begna002

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: `lazy\_calls.ml'

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

Compile the file `lazy\_calls.ml' and run it with several command line

parameters such as

,----

| > ocamlc lazy\_calls.ml

| > ./a.out 20

| ...

| > ./a.out 25

| ...

| > ./a.out 30

| ...

| > ./a.out 32

| ...

`----

The input parameter is passed to a `fib n' function so will repeatedly

compute the `nth' Fibonacci number. The questions below pertain to

timing repeated calls to these functions so pick an `n' that is large

enough to see timing info printed but not so large as to make repeated

runs tediously long.

(A)

~~~

Examine the timing information output for the section marked

,----

| =====(1) either\_args=====

`----

and find the corresponding block of code.

Answer the following questions.

1. Describe what the function `either\_args' does, especially its

dependence on the `(flag1,flag2)' parameters.

2. Explain why the times printed for each invocation of `either\_args'

is identical regardless of the flags passed in. Incorporate the

notion of \*eager evaluation\* in your answer.

**The function either\_args prints out the first or second argument value which depends on whether or not Flag1 or Flag2 are respectively set to True. Arg1 and Arg2 are calculated Fibonacci values which are computed during the call to either\_args**

**Because eager arguments first evaluate the arguments that are passed into a function, and then calls the function with the results, the computed times are the same since the Fibonacci value given is always being computed twice.**

(B)

~~~

Examine the output with section marked

,----

| =====(2) either\_susp, new suspension each iter=====

`----

and locate the corresponding code in `lazy\_calls.ml'.

Contrast this with the first section by answering the following

questions.

1. How are `either\_susp' and `either\_arg' from the previous section

different?

2. Why are the times for some of the calls of `either\_susp' different

from each other?

3. Explain why some of the times in section (1) and (2) differ and

some are the same.

1. **Rather than the computed fibonncai values being passed into the function, a “suspension” is passed in as an argument.**
2. **Because of this suspension, only when the arguments are called within the body of either\_susp are the Fibonacci values calculated. Because of this, if the values have to be calculated (when either or both flags are true), then the calculation will cost some time.**
3. **There is one instance when sections (1) and (2) cost roughly the same amount of time: when the Fibonacci value of n has to be evaluated both times. This happens all four times in section (2), but only once in section (2) when both flags are set to true.**

(C)

~~~

Examine the output with section marked

,----

| =====(3) either\_lazy, lazy expr each iteration======

`----

and locate the corresponding code in `lazy\_calls.ml'.

1. How does the code associated with the `either\_lazy' differ from

`either\_susp'? What module and special syntax are used?

2. Compare the timings from output section (2) and (3) to each other

and explain any similarities or differences.

1. **In section (3), a Lazy evaluation strategy is used rather than suspension as in section (2). This function uses the Ocaml integrated Lazy module, using calls to lazy() and Lazy.force.**
2. **Both section (3) and (2) have relatively similar timings since they are both doing essentially the same thing. Only when the flags are set to true will a call to fib actually be executed, which occurs in the body of both respective functions.**

(D)

~~~

Examine the output with section marked `====(4) either\_susp, single

suspension before loop====' and locate the corresponding code in

`lazy\_calls.ml'.

Contrast the code associated with (4) and the previous use of

suspensions in (2): how do the codes differ? Explain why their timings

are the same or why there are differences.

**Rather than passing the suspension itself to the function either\_susp, an instance of the suspension is being called through the assignment of (fib\_susp) to this suspension. While this slightly changes the syntax of section (4) from section (2), the timings are relatively the same because they utilize the same method of only calling the fib function when the flags are set to true.**

(E)

~~~

Examine the output with section marked

,----

| =====(5) either\_lazy, new lazy expr each iter======

`----

and locate the corresponding code in `lazy\_calls.ml'.

Contrast the code associated with (5) and the previous use of

suspensions in (3): how do the codes differ? Explain why their timings

are the same or why there are differences.

**Like before, rather than passing the lazy implementation of the fib function itself to the function either\_lazy, an instance of the implementation is being called through the assignment of (fib\_lazy) to this implementation. This time however, the timings are different. Because the arguments passed into either\_lazy are the same instance of fib\_lazy which implements a lazy evaluation strategy, the fib function is only needed once before the information is “memoized”. Thus, after the first instance of a flag being set to true, any subsequent calls to either\_lazy will take virtually no time to compute.**

PROBLEM 2: Streams of Data

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

Compile and run the code in `stream\_demo.ml'. Examine the results

noting that you \*must press Ctrl-c to kill the program\* as it will

otherwise loop infinitely.

(A)

~~~

Analyze the first section of code associated with the heading

,----

| =====finite stream from list=====

`----

Answer the following questions about the code.

1. Describe what data source is used as a source for the stream and

what library function is used to create a stateful stream from it.

2. What library functions are used to access elements of the stream?

Examine their documentation and describe what they do.

3. Why does the output of the program differed between the list and

stream from one iteration to the next?

1. **The Stream module is used in the program, specifically the function Stream.of\_list. In this case, a list is passed to create the stateful stream.**
2. **Stream.count returns the index of the each element in the list. List.hd returns the first element of the list, while Stream.next continually moves the Stream pointer forward, printing out the next item in the list.**
3. **The Stream.next function continually alters the pointer which is positioned somewhere in the list to move one step forward. Thus, subsequent items are printed rather than the same one.**

(B)

~~~

Examine the second section of `stream\_demo.ml' with heading

,----

| =====infinite int stream from function=====

`----

Answer the following questions about the code.

1. As in the previous problem, describe what data source is used as a

source for the stream and what library function is used to create a

stateful stream from it.

2. What library functions are used to access elements of the stream?

Are they the same or different from part (A) above?

3. Will the stream in this case ever "run out" of data? Explain your

reasons why or why not?

4. Explain why this stream can produce a lot of data but requires very

little memory to represent.

1. **The Stream module is once again used, and in this case the function Stream.from is used since this time, a function is being passed in.**
2. **Stream.count and Stream.next are once again used, but this time they are used on a function rather than a list.**
3. **The stream will never run out of data since the data type used is a function that makes continues modifications to a reference variable (specifically multiplying it by two). Because of this, there is no “end”.**
4. **Memory is not an issue in this instance of a stream since the stream only holds one value that is continually being changed using a reference. Because of this mutability, past value are not saved.**

Optional Extras

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

Custom implementations of `lazy' fail

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

A first instinct for most folks observing the connection between

Problem 1's suspensions via functions and the `lazy' keyword is to

attempt their own implementation of it along the following lines. (\*

custom\_lazy.ml: optional problem exploring possible implementations of

lazy expressions. \*)

type 'a lazy\_expr = { (\* type for lazy expressions \*) expr : unit ->

'a; (\* expression to evaluate \*) mutable result : 'a option; (\*

saved results, None if uneval'd yet \*) };;

let my\_lazy expr = (\* create a lazy version of expr \*) {expr = (fun ()

-> expr); result = None} ;;

let my\_force lazy\_expr = (\* force a value out of the lazy\_expr \*)

match lazy\_expr.result with

Some a -> a (\* already evaluated \*)

None -> (\* not eval'd yet \*)

let a = lazy\_expr.expr () in (\* eval \*) lazy\_expr.result <- Some

a; (\* save result \*) a (\* return result \*) ;;

Experiment with this version by adding calls to its `my\_lazy' into the

demo in `lazy\_calls.ml'. See if it in fact creates the same effects

as the built-in `lazy' keyword.

When you determine that it does not, speculate as to why it fails and

why compiler support is required for features like `lazy'.

Stream peek vs. has\_more

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

It is natural to expect the `Stream' module to contain a function like

`Stream.has\_more stream' which returns true when the stream has more

data and false otherwise. However, there is no such `has\_more'

function present and the `peek' function which returns a None/Some is

used instead. Why would it be difficult or even impossible to write a

general purpose `has\_more' function? Consider that streams are very

general and in the sample code provided, data from two different

sources is converted to streams.