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

LAB 09 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: Memfunc1 Module

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

(A)

~~~

Examine `memfunc1.ml' and answer the following questions.

1. What is the name of the functor defined in the module?

2. Like all functors, this one takes a parameter module. What bindings

must the parameter module have according to its signature?

3. The module that is created by the functor has several internal

bindings but only one of them is publicly visible. What is it and

how is this visibility control enforced? (ASK ABOUT THIS)

**The name of the module is Memoize, with a parameter module FuncMod that must have the following parameters: a type arg\_t and reg\_t, a function that takes in arg\_t and returns ret\_t, and a comparison function that compares 2 arg\_t variable and returns an int. The call function is visible since the module created a signature for it.**

(B)

~~~

As in any other place a module is being defined, functors can

establish nested modules. The functor in `Memfunc1' creates two

nested modules within its body. Describe these two modules providing

their names and what their purpose appears to be. Describe how

bindings of the parameter module are used in these nested modules.

**The module MapInterface creates bindings for a type and a comparison function which are necessary in order to use the Set/Map functors later on. The parameter module is used here by binding FuncMod.arg\_t to type t, and FuncMod.compare\_arg to the comparison function.**

**The module MemoMap is a mapping of the MapInterface module by using Map.make. MemoMap is used later on for various methods.**

(C)

~~~

Analyze the bindings for `arg\_ret\_map' and the function `call'.

Describe how these are used in conjunction the a function carried by

the parameter module.

**A mutable empty mapping is bound to arg\_ret\_map. The call function uses arg\_ret\_map in order to find/add an argument. If it wasn’t found, the parameter function in the module FuncMod is called (using FuncMod.func).**

PROBLEM 2: fib Program

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

(A)

~~~

Examine the first section of the provided `fib.ml' file. Describe how

the functor in `memfunc1.ml' is used at the top of `fib.ml'. Include a

description of how the parameter module to the functor is created.

**The functor Memoize is used to create a memorized version of a Fibonacci function. The parameter module MemoFib is created by binding Memfunc1.Memoize(FibFunc) to it, where FibFunc is a module that implements the Fibonacci function.**

(B)

~~~

The main intent of the program in `fib.ml' is to compare the amount of

time it takes to make repeated calls to the `fib' function versus a

memoized version of it in `MemoFib.call'. Analyze the main code for

the program.

1. Identify where calls to `fib' and `MemoFib.func' are made.

2. Identify where the results of these calls are stored.

3. Describe what function is used to gather timing information on how

long the repeated function calls take.

**Calls to fib and MemoFib.func are called in each instance where values are being added to arrays farr for fib, and marr for MemoFub.func. This occurs twice for each function. The values are stored in these arrays, since they are mutable. Sys.time() is used to gather timing information, and the functions are being called n times where n is the nth Fibonacci number provided as an argument by the user.**

(C)

~~~

Use `ocamlc' to Compile and run the `fib.ml' program together with its

dependency `memfunc1.ml'. Run the resulting program with and paste in

the output in the space below.

Note that the program requires command line argument, an integer. Use

30 as the parameter.

**Running loops on fib/mem functions**

**fib time: 0.0781 secs**

**mem time: 0.0781 secs**

**fib( 1) = 1 | MemoFib.call( 1) = 1**

**fib( 2) = 1 | MemoFib.call( 2) = 1**

**fib( 3) = 2 | MemoFib.call( 3) = 2**

**fib( 4) = 3 | MemoFib.call( 4) = 3**

**fib( 5) = 5 | MemoFib.call( 5) = 5**

**fib( 6) = 8 | MemoFib.call( 6) = 8**

**fib( 7) = 13 | MemoFib.call( 7) = 13**

**fib( 8) = 21 | MemoFib.call( 8) = 21**

**fib( 9) = 34 | MemoFib.call( 9) = 34**

**fib(10) = 55 | MemoFib.call(10) = 55**

**fib(11) = 89 | MemoFib.call(11) = 89**

**fib(12) = 144 | MemoFib.call(12) = 144**

**fib(13) = 233 | MemoFib.call(13) = 233**

**fib(14) = 377 | MemoFib.call(14) = 377**

**fib(15) = 610 | MemoFib.call(15) = 610**

**fib(16) = 987 | MemoFib.call(16) = 987**

**fib(17) = 1597 | MemoFib.call(17) = 1597**

**fib(18) = 2584 | MemoFib.call(18) = 2584**

**fib(19) = 4181 | MemoFib.call(19) = 4181**

**fib(20) = 6765 | MemoFib.call(20) = 6765**

**fib(21) = 10946 | MemoFib.call(21) = 10946**

**fib(22) = 17711 | MemoFib.call(22) = 17711**

**fib(23) = 28657 | MemoFib.call(23) = 28657**

**fib(24) = 46368 | MemoFib.call(24) = 46368**

**fib(25) = 75025 | MemoFib.call(25) = 75025**

**fib(26) = 121393 | MemoFib.call(26) = 121393**

**fib(27) = 196418 | MemoFib.call(27) = 196418**

**fib(28) = 317811 | MemoFib.call(28) = 317811**

**fib(29) = 514229 | MemoFib.call(29) = 514229**

**fib(30) = 832040 | MemoFib.call(30) = 832040**

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

**Re-running loops on fib/mem again**

**fib time: 0.0781 secs**

**mem time: 0.0000 secs**

**fib( 1) = 1 | MemoFib.call( 1) = 1**

**fib( 2) = 1 | MemoFib.call( 2) = 1**

**fib( 3) = 2 | MemoFib.call( 3) = 2**

**fib( 4) = 3 | MemoFib.call( 4) = 3**

**fib( 5) = 5 | MemoFib.call( 5) = 5**

**fib( 6) = 8 | MemoFib.call( 6) = 8**

**fib( 7) = 13 | MemoFib.call( 7) = 13**

**fib( 8) = 21 | MemoFib.call( 8) = 21**

**fib( 9) = 34 | MemoFib.call( 9) = 34**

**fib(10) = 55 | MemoFib.call(10) = 55**

**fib(11) = 89 | MemoFib.call(11) = 89**

**fib(12) = 144 | MemoFib.call(12) = 144**

**fib(13) = 233 | MemoFib.call(13) = 233**

**fib(14) = 377 | MemoFib.call(14) = 377**

**fib(15) = 610 | MemoFib.call(15) = 610**

**fib(16) = 987 | MemoFib.call(16) = 987**

**fib(17) = 1597 | MemoFib.call(17) = 1597**

**fib(18) = 2584 | MemoFib.call(18) = 2584**

**fib(19) = 4181 | MemoFib.call(19) = 4181**

**fib(20) = 6765 | MemoFib.call(20) = 6765**

**fib(21) = 10946 | MemoFib.call(21) = 10946**

**fib(22) = 17711 | MemoFib.call(22) = 17711**

**fib(23) = 28657 | MemoFib.call(23) = 28657**

**fib(24) = 46368 | MemoFib.call(24) = 46368**

**fib(25) = 75025 | MemoFib.call(25) = 75025**

**fib(26) = 121393 | MemoFib.call(26) = 121393**

**fib(27) = 196418 | MemoFib.call(27) = 196418**

**fib(28) = 317811 | MemoFib.call(28) = 317811**

**fib(29) = 514229 | MemoFib.call(29) = 514229**

**fib(30) = 832040 | MemoFib.call(30) = 832040**

(D)

~~~

Describe why the time to run `fib' and `MemoFib.call' are very similar

in the first timing loop but drastically different in the second

timing loop. Relate your answer to the internal workings of the

`Memofunc1.Memoize' functor.

**The first time that fib and MemoFib.call are used, MemoFib does not already have stored Fibonacci values, thus it must use the fib function to calculate the values from 1 to n. The second time that the functions are called however, MemoFib already has the values stored in the mutable mapping arg\_ret\_map. Thus, the argument n is used as a key to find the value associated with it in the Fibonacci sequence, eliminating the need to constantly call the fib function.**

Optional Enrichment Problems

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

1. We have seen that `MemoFib.call 30' will take some time on the

first call but a subsequent call runs in almost no time. Does this

mean that the speed of `MemoFib.call 31' will be shorter as well?

Why or why not?

2. The functor established in `Memofunc1' works for functions of 1

parameter. How would one go about handling functions of two,

three, or four arguments? If one is writing functions to be

compatible with memoization, how could one write the functions of

multiple arguments so as to avoid the need write more functors?