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

LAB 05 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: Mutable vs Immutable Stack Usage

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

(A)

~~~

Examine the code in `mut\_stack.ml' which implements a mutable stack

data structure using a new record type `mut\_stack'.

In a REPL, call the `make ()' function to create a `mut\_stack' and

demonstrate some `push / pop / top / poptop' operations with integers.

What is the return value associated with each of the functions?

**Solution :solution:**

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

**- make returns a new empty stack record**

**- push/pop return unit indicating a change to the stack**

**- top returns an element without changing the stack**

**- poptop returns an element and also changes the stack**

**,----**

**| # #use "mut\_stack.ml";;**

**| ...**

**| # let istack = make ();;**

**| val istack : '\_weak2 mut\_stack = {size = 0; data = []}**

**| # push istack 5;;**

**| - : unit = ()**

**| # push istack 3;;**

**| - : unit = ()**

**| # push istack 9;;**

**| - : unit = ()**

**| # istack;;**

**| - : int mut\_stack = {size = 3; data = [9; 3; 5]}**

**| # istack.size;;**

**| - : int = 3**

**| # pop istack;;**

**| - : unit = ()**

**| # istack;;**

**| - : int mut\_stack = {size = 2; data = [3; 5]}**

**| # poptop istack;;**

**| - : int = 3**

**| # istack;;**

**| - : int mut\_stack = {size = 1; data = [5]}**

`----

(B)

~~~

In the type declaration for `mut\_stack', explain the significance of

the `'a' notation: what effect does it have on the kinds of stacks

that can be created? Demonstrate the flexible nature of `mut\_stack' in

a REPL showing several kinds of stacks.

**Solution :solution:**

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

**Since the type is defined as `'a mut\_stack', any kind of mutable stack**

**can be created such as an `int mut\_stack' from before or `string**

**mut\_stack' and `bool mut\_stack' below. This creates flexibility in**

**that any kind of data can be stored in the stack.**

**,----**

**| # let sstack = make ();;**

**| val sstack : '\_weak3 mut\_stack = {size = 0; data = []}**

**| # push sstack "hi";;**

**| - : unit = ()**

**| # push sstack "bye";;**

**| - : unit = ()**

**| # sstack;;**

**| - : string mut\_stack = {size = 2; data = ["bye"; "hi"]}**

**| # pop sstack;;**

**| - : unit = ()**

**| # sstack;;**

**| - : string mut\_stack = {size = 1; data = ["hi"]}**

**| # poptop sstack;;**

**| - : string = "hi"**

**| # let bstack = make ();;**

**| val bstack : '\_weak4 mut\_stack = {size = 0; data = []}**

**| # push bstack true;;**

**| - : unit = ()**

**| # push bstack true;;**

**| - : unit = ()**

**| # push bstack false;;**

**| - : unit = ()**

**| # bstack;;**

**| - : bool mut\_stack = {size = 3; data = [false; true; true]}**

**`----**

(C)

~~~

Examine the code in `imu\_stack.ml' which implements an immutable stack

data structure using a new record type `imu\_stack'.

In a REPL, call the `make ()' function to create a `imu\_stack' and

demonstrate some `push / pop / top / poptop' operations with integers.

What is the return value associated with each of the functions?

What is very different about making repeated `push' calls on

`imu\_stack' compared to `mut\_stack'?

**Solution :solution:**

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

**- make returns a new empty stack record**

**- push/pop return a NEW stack with the change made**

**- top returns an element without changing the stack**

**- poptop returns a pair: the top element and a new stack without the**

**top element**

**Since the stack is immutable, calls to push don't change the original**

**and instead return a new stack which must be let bound to capture it.**

**Similarly so for pop. This means repeated let bindings are required**

**to affect and appearance of change in the stack which was not the case**

**for the `mut\_stack'.**

**,----**

**| # #use "imu\_stack.ml";;**

**| ...**

**| # let istack = make ();;**

**| val istack : 'a imu\_stack = {size = 0; data = []}**

**| # push istack 5;;**

**| - : int imu\_stack = {size = 1; data = [5]}**

**| # istack;;**

**| - : 'a imu\_stack = {size = 0; data = []}**

**| # let istack2 = push istack 2;;**

**| val istack2 : int imu\_stack = {size = 1; data = [2]}**

**| # let istack52 = push istack2 5;;**

**| val istack52 : int imu\_stack = {size = 2; data = [5; 2]}**

**| # istack;;**

**| - : 'a imu\_stack = {size = 0; data = []}**

**| # pop istack52;;**

**| - : int imu\_stack = {size = 1; data = [2]}**

**| # istack52;;**

**| - : int imu\_stack = {size = 2; data = [5; 2]}**

**| # top istack52;;**

**| - : int = 5**

**| # poptop istack52;;**

**| - : int imu\_stack \* int = ({size = 1; data = [2]}, 5)**

**`----**

PROBLEM 2: Weak vs Polymorphic Types

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

(A)

~~~

An astute observer will see the following apparent change of type for

`mut\_stacks':

,----

| # let int\_stack = make ();;

| # int\_stack;;

| - : '\_weak1 mut\_stack = {size = 0; data = []}

| (\* ^^^^^^^ \*)

| (\* what is '\_weak1? \*)

|

| # push int\_stack 5;;

| - : unit = ()

|

| # int\_stack;;

| - : int mut\_stack = {size = 1; data = [5]}

| (\* ^^^ \*)

| (\* now its an int mut\_stack ?? \*)

`----

Read the first few paragraphs of the OCaml System Manual, Ch 5.1 to

learn about weak types.

[https://caml.inria.fr/pub/docs/manual-ocaml/polymorphism.html]

Explain below the peculiar `'\_weak1' type associated with an empty

`mut\_stack'. Is it the same as a polymorphic `'a' type?

**Solution :solution:**

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

**The `'\_weak1' designation is associated with a type that has not yet**

**been determined by the compiler. It most often happens in the REPL**

**associated with mutable names. In most cases, further operations will**

**resolve `'\_weak' types to a concrete type like `int' or `string' after**

**seeing typed operations are performed like `push 5' which resolves the**

**stack to an `int mut\_stack'. This usually does not happen during**

**compilation as the compiler will look at the entirety of the code and**

**see later operations which are resolve an earlier weak type.**

**It is NOT the same as `'a' type parameter which is truly polymorphic**

**and can be used in place of any concrete type.**

(B)

~~~

Consider the following sequence of operations starting with an `empty

imu\_stack'.

,----

| # #use "imu\_stack.ml";;

|

| # let empty = make ();;

| val empty : 'a imu\_stack = {size = 0; data = []}

|

| # let istack = push empty 5;;

| val istack : int imu\_stack = {size = 1; data = [5]}

|

| # let sstack = push empty "hello";;

| val sstack : string imu\_stack = {size = 1; data = ["hello"]}

|

| # empty;;

| - : 'a imu\_stack = {size = 0; data = []}

`----

Answer the following questions about the above.

- What is the type of `empty' here? Is it a weakly typed binding as

discussed in the previous section?

- Why is it possible to do both `push 5' and `push "hello"' into the

`empty'?

- Does pushing different types into `empty' change the type of

`empty'?

**Solution :solution:**

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

**,----**

**| # rey.name;;**

**| - : string = "Rey"**

**| # String.length rey.name;;**

**| - : int = 3**

**| # vader.darkside;;**

**| - : bool = true**

**| # vader.episodes @ rey.episodes;;**

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

**`----**

**The type of `empty' is `'a imu\_stack'. This means it can be used in**

**any situation where a concrete type is required. This is different**

**from the `'\_weak' types of Problem 1C which were concrete types that**

**had yet to be inferred. `empty' will always have type `'a imu\_stack'.**

**Pushing different concrete types into `empty' will produce a new stack**

**with the concrete type of the elements such as `int imu\_stack' or**

**`string imu\_stack'. This is due to the `'a' associated with `empty'**

**matching any of these. However, this does not change the type of**

**`empty'.**

(C)

~~~

Consider the following sequence of operations which are nearly

identical to the previous section except with the `mut\_stack' type

used.

,----

| # #use "mut\_stack.ml";;

|

| # let empty = make ();;

| val empty : '\_weak2 mut\_stack = {size = 0; data = []}

|

| # push empty 5;;

| - : unit = ()

|

| # empty;;

| - : int mut\_stack = {size = 1; data = [5]}

|

| # push empty "hello";;

| Characters 11-18:

| push empty "hello";;

| ^^^^^^^

| Error: This expression has type string but an expression was expected of type

| int

|

`----

Why does this sequence result in a type error?

**Solution :solution:**

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

**The initial type of `empty' in this cases is not fully known: it has a**

**`'\_weakl' type but the parameter will be replaced something concrete**

**as soon as the type system has enough information. On pushing an int**

**into the stack, `'\_weak2' becomes `int'. This precludes pushing a**

**`string' into the stack which will now only accept ints.**

PROBLEM 3: Tracking Stack History

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

(A)

~~~

Examine the file `history.ml'. It performs a series of push operations

on stacks and attempts to generate a history of the states the stack

is in. This is done first on the `imu\_stack' and then on the

`mut\_stack'.

In this file, do the operations `push' and `make' actually work on

both `mut\_stack' and `imu\_stack' or is something else going on?

Relate your answer to the `open' directives present in `history.ml'.

**Solution :solution:**

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

**Though they are named the same, there are actually two `push'**

**functions, one in `imu\_stack.ml' and one in `mut\_stack.ml'. These act**

**only on their respective data types. Likewise for the `make' function.**

**The `open' directives will bind mentions of `push/make' to the**

**functions coming from their respective modules/source files.**

(B)

~~~

Load `history.ml' into a REPL. Before doing so, you will need to

ensure that the `Mut\_stack' and `Imu\_stack' modules are available by

using the REPL's `#mod\_use' directive as follows.

,----

| # #mod\_use "mut\_stack.ml";;

| ...

| # #mod\_use "imu\_stack.ml";;

| ...

| # #use "history.ml";;

| ...

`----

Show the output of running these three directives below.

We discuss the modules/signatures later but note that `#mod\_use'

prints out information about the values and types present in a source

file.

**Solution :solution:**

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

**,----**

**| # #mod\_use "mut\_stack.ml";;**

**| module Mut\_stack :**

**| sig**

**| type 'a mut\_stack = { mutable size : int; mutable data : 'a list; }**

**| val make : unit -> 'a mut\_stack**

**| val is\_empty : 'a mut\_stack -> bool**

**| val push : 'a mut\_stack -> 'a -> unit**

**| val pop : 'a mut\_stack -> unit**

**| val top : 'a mut\_stack -> 'a**

**| val poptop : 'a mut\_stack -> 'a**

**| end**

**| # #mod\_use "imu\_stack.ml";;**

**| module Imu\_stack :**

**| sig**

**| type 'a imu\_stack = { size : int; data : 'a list; }**

**| val make : unit -> 'a imu\_stack**

**| val is\_empty : 'a imu\_stack -> bool**

**| val push : 'a imu\_stack -> 'a -> 'a imu\_stack**

**| val pop : 'a imu\_stack -> 'a imu\_stack**

**| val top : 'a imu\_stack -> 'a**

**| val poptop : 'a imu\_stack -> 'a imu\_stack \* 'a**

**| end**

**| # #use "history.ml";;**

**| val imu\_history : int Imu\_stack.imu\_stack array =**

**| [|{size = 0; data = []}; {size = 1; data = [1]}; {size = 2; data = [2; 1]};**

**| {size = 3; data = [3; 2; 1]}; {size = 4; data = [4; 3; 2; 1]};**

**| {size = 5; data = [5; 4; 3; 2; 1]};**

**| {size = 6; data = [6; 5; 4; 3; 2; 1]};**

**| {size = 7; data = [7; 6; 5; 4; 3; 2; 1]};**

**| {size = 8; data = [8; 7; 6; 5; 4; 3; 2; 1]};**

**| {size = 9; data = [9; 8; 7; 6; 5; 4; 3; 2; 1]};**

**| {size = 10; data = [10; 9; 8; 7; 6; 5; 4; 3; 2; 1]}|]**

**| val mut\_history : int Mut\_stack.mut\_stack array =**

**| [|{size = 10; data = [10; 9; 8; 7; 6; 5; 4; 3; 2; 1]};**

**| {size = 10; data = [10; 9; 8; 7; 6; 5; 4; 3; 2; 1]};**

**| {size = 10; data = [10; 9; 8; 7; 6; 5; 4; 3; 2; 1]};**

**| {size = 10; data = [10; 9; 8; 7; 6; 5; 4; 3; 2; 1]};**

**| {size = 10; data = [10; 9; 8; 7; 6; 5; 4; 3; 2; 1]};**

**| {size = 10; data = [10; 9; 8; 7; 6; 5; 4; 3; 2; 1]};**

**| {size = 10; data = [10; 9; 8; 7; 6; 5; 4; 3; 2; 1]};**

**| {size = 10; data = [10; 9; 8; 7; 6; 5; 4; 3; 2; 1]};**

**| {size = 10; data = [10; 9; 8; 7; 6; 5; 4; 3; 2; 1]};**

**| {size = 10; data = [10; 9; 8; 7; 6; 5; 4; 3; 2; 1]};**

**| {size = 10; data = [10; 9; 8; 7; 6; 5; 4; 3; 2; 1]}|]**

**`----**

(C)

~~~

Examine the two values established by `history.ml'

- `imu\_history' associated with the history of an `imu\_stack'

- `mut\_history' associated with the history of a `mut\_stack'

Determine if the histories accurately reflect the different states

that the stacks of undergone or not. Describe anything strange/wrong

that you observe particularly about `mut\_history' and determine as

best as you can WHY it is happening.

**Solution :solution:**

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

**`imu\_history' is accurate as each time an element is pushed, a new**

**stack results which is saved in the array.**

**`mut\_history' is initialized with the same empty stack. While no**

**array elements are actually changed, since they all point to the same**

**mutable stack, all elements appear to change. Thus, there is no**

**history: just many pointers to the same mutating stack.**