CS51 LAB 4: MODULES & FUNCTORS
DISCUSSION AND CONSIDERATIONS

0. Introduction

In this document, we'll review the thought process and provide some additional context or information involved with a subset of the problems presented in Lab 4. The intention of this document is to serve as review for the midterm (or final), and is not meant to replace your attempt of the problems. In fact, if you have not completed Lab 4, please do so before reading the remainder of this document: completing the lab is the best way to truly internalize the material presented. Even after you've completed the lab, we hope you find the following useful and clarifying for your studies.

Throughout this document, we assume that you have completed Lab 4 and have your solutions open adjacent to this discussion as you follow along.

Part 2. Files as modules

1. **Exercise 2C: Write a TF signature.** In real-world programming, the task of writing a signature to match a pre-existing module implementation is arguably an uncommon approach: you normally would pre-define the abstraction and write code that accomplishes the goals of the specification.

This is not to say that exercising this uncommon task is without value. In fact, as you'll see when we discuss functors in more detail, it can be advantageous to identify common methods across multiple modules and figure out ways to leverage those similarities to extend the capabilities of pre-existing modules. Ultimately, the intention of this exercise is to practice your ability to think more deeply about types, and successful completion requires the ability to determine *compatible* types.

The exercise asks you to determine the set of functions and labels that are compatible between the `Sam` and `Gabbi` modules, and then record the most generic type in the signature that maintains that compatibility. We'll look at some examples.

1.1. *Incompatible types.* Here is the `least_favorite` function defined in `Sam.ml`:

```
let least_favorite_function = (land)
```

And here is the same from `Gabbi.ml`:

```
let least_favorite_function = ( ** )
```

From the Pervasives module documentation, we see that `land` is a function for "bitwise logical and" and is of type `int -> int -> int`, whereas the `(**)` operator is exponentiation and is of type `float -> float -> float`. These functions are incompatible types and we cannot write a signature that is compatible with both. As a result, the signature for this exercise *cannot contain a* `least_favorite_function`.

You may wonder why we could not write the following signature to apply to both:

```
(* This is bad news! *)
val least_favorite_function =  'a -> 'a -> 'a
```

But this does not work, because the signature guarantees the types of the implementation. Neither of the `least_favorite_function` implementations defined in the modules can accept a generic type. As an example, the above signature would allow `least_favorite_function` to accept a string type, but this is an incompatible type with both implementations. Although there are times you may want a generic type in a signature, it would be wrong for this exercise because the modules would not be a valid implementation of it.

1.2. *Compatible types.* On the other hand, the `favorite_function` implementations *are* compatible. `Sam.ml` has the following implementation[1]:

```
let favorite_function x y =
  log x +. log y +. float_of_string "0x74686563616b656973616c6965"
```

Which has type `float -> float -> float`. `Gabbi.ml` has the following implementation:

```
let favorite_function x y = x +. float_of_int (succ (int_of_float y))
```

We can see here that the two function types are compatible, so when writing the signature we would then include the following:

```
val favorite_function =  float -> float -> float
```

2. **Exercise 2D: TFGabbi and TFSam.** This exercise mainly requires the lookup of the correct syntax. We complete the exercise with the following lines of code:

```
module TFGabbi : TF = Gabbi ;;
module TFSam : TF = Sam ;;
```

Which is syntactic sugar for the following:

```
module TFGabbi = (Gabbi : TF) ;;
module TFSam = (Sam : TF) ;;
```

[1]By the way, have you tried converting the hexadecimal number from Sam's `favorite_function` into a string?

In either case, we might interpret the lines as creating a new module, like `TFEmily`, that is the result of the binding of the `TF` signature to the `Gabbi` module. The binding ensures that the `TFGabbi` module only contains the methods and properties specified by the `TF` signature.

The latter syntax should not be confused with OCaml's notion of first-class modules; although we did not address this topic in lab, you may have come across examples of this elsewhere. First-class modules use syntax like `(module Name : SIG)` to encapsulate a module as a value. Using this syntax, modules can be passed as parameters to functions or stored in labels using `let`. We do not do this for this exercise, however, because in order to access values or functions in the encapsulated modules we would need to unpack them. Purely for your curiosity, we show a concrete example of this below.

```
# (* What follows is NOT the correct answer for this exercise *)
  (* First, we encapsulate the Emily module and store it in a label *)
  let tfg = (module Gabbi : TF) ;;
val tfg : (module TF) = <module>

# (* Now we'll attempt to access a value, but it's encapsulated *)
  tfg.hometown ;;
Error: Unbound record field hometown

# (* We must unpack the module to access it *)
  module TFG = (val tfg : TF) ;;
module TFG : TF

# TFG.hometown ;;
- : bytes = "Raleigh, NC"
```

Again, do not be confused by the similarity in the syntax. First-class modules are defined by the `(module Name : SIG)` syntax, whereas we define a module using `(Name : SIG)`. The latter evaluates the module defined by `Name`, and binds the signature `SIG` to it; either this syntax or its syntactic sugar, provided early in this discussion, are the intended answers for this exercise.

Part 5. Functors

Functors are a way to perform operations on modules; we might think of them as special functions that accept a module as a parameter, and use the implementation provided in the passed module to implement a new module. Part 5 creates a functor called `MakeStack` that, in a certain sense, is a special function that accepts a module `Element` as a "parameter" and uses it to implement a stack. Since `Element`

must adhere to the `SERIALIZE` signature, we are guaranteed that no matter which module is applied to `MakeStack`, we know that it has a type `t` and a `serialize` function that transforms that type `t` into a string. The OCaml type checker enforces this requirement at compile time.

We implement `MakeStack` as a functor so that we can create an abstract `stack` data type while still providing additional functionality that depends on the type of data of each element. In other words, we don't have to somehow create, in advance, a serialize function that works on every possible type!

1. **Exercise 5A: Stack implementation.** Based on the above information, writing the serialization method in the stack implementation depends on the `serialize` function provided by the `Element` module, accessing it using dot notation: `Element.serialize`.

   One additional hint: although it is possible to implement recursively (we even defined the function in the library code using `let rec`) it can also be done in one or two lines of code by using other functions you implemented in `MakeStack`!

2. **Exercises 5B: `IntStack` and 5C: `IntStringStack`.** These exercises necessitate the use of the `MakeStack` functor to implement two different types of stacks: one using integers and another with `(int, string)` tuples. Although you've already created the functor `MakeStack` that creates a stack, your task for these exercises is to define the module implementation that adheres to the `SERIALIZE` signature for each. In other words, you are implementing modules passed as the `Element` "parameter" to the `MakeStack` functor. Once done, the final application of the new modules to the functor create the final `IntStack` and `IntStringStack` modules.

   You may do this explicitly in two steps, like the below skeleton code:

```
module IntSerialize : SERIALIZE =
  struct
    (* complete implementation here *)
  end ;;

module IntStack = MakeStack(IntSerialize) ;;
```

   Or collapse both steps into one, applying an anonymous module to the functor:

```
module IntStack =
  MakeStack(struct
    (* complete implementation here *)
  end) ;;
```

   Choosing one over the other is a stylistic and design choice.

3. **Additional practice.** Given the length of the lab, we weren't able to add some additional practice for functors. You may wish to take the time to implement the following additional exercise:

Write a *new* module that implements a `length` function for the abstract `stack` type defined by `MakeStack`. The `length` function should accept a `stack` as a parameter and return the count of elements held within it, using only the module's `fold_left` function.

By using a functor, you will not need to modify the implementation of any of the code from part 5 to complete this additional exercise. In essence, you would be providing `length` functionality to `MakeStack` (albeit in a new module) after you have completed its implementation.

This may feel like a needless exercise, since you could easily add a `length` function to the `MakeStack` signature and implementation. The practice is not without merit, though; had we defined the stack type in a different way — naming it `t` instead of `stack` — this extension functor that you write would be applicable to *any* module that exposes a type `t` and a `fold_left` function. It is common among `Core.Std` modules to expose their abstract types in this way, though we don't use those modules in this course.

You could even further make the exercise useful by adding other, more complex functions based on `fold_left`, like `for_all`, `exists`, or `mem`. The behavior of these functions would be analogous to those with the same names as found in the List module.

Back to the exercise; try this on your own! Below is a hint if you get stuck.

3.1. *Hint.* First, create a signature that ensures that a module adheres to both `fold_left` and also a type `stack`. The type is necessary so that you can accept the data structure as parameter to `length`. You might call this signature `FOLD`, for instance, to make its purpose clear, but the name is up to you.

Next, write a functor that accepts a module that matches the signature you wrote and implements the `length` function. The function implementation should use the applied module's `fold_left` function to accomplish its task.

Finally, try to create a new module to test your extended functionality!