CS 131 Week 1 Worksheet Answers

(5 mins) OCaml Setup & Workflow

You can use OCaml on the SEASnet servers:

```
$ ssh [username]@lnxsrv11.seas.ucla.edu
$ ocaml
```

If you prefer running locally, here are some resources:

Install OCaml: https://ocaml.org/install

(Optionally) install `utop`, an alternative *toplevel* that's much better than the default: https://github.com/ocaml-community/utop

For VS Code users:

- the Ocaml Platform extension (read the installation notes carefully - there's a few dependencies

REPL tips:

- Run `utop` or `ocaml` from the command line
- Now you can write and evaluate expressions on the fly!
- Evaluate an expression by putting `;;` at the end
    - OCaml is not newline-sensitive, so without this the REPL will wait for you to write more code
- To load and execute a file (as in hw1 & hw2): `#use "path/to/file.ml";;`
- For files, use
  `let () = <main expression here>`
  as the entry point. Don't directly call functions on the top level like in Python.

For discussion problems I suggest typing in Notepad or similar and copy-pasting into a REPL. You don't get autocomplete or LSP on a written test.

(10 mins) OCaml Overview

Introduce yourself!
Q: What is something you are looking forward to this quarter?

Let's compare OCaml to languages you've seen before. Don't worry about "right" answers, this is meant to contextualize OCaml for you.

OCaml is statically typed, like _____ and _____ but unlike _____ and _____.
Static typing is where variable types in a program are known *at compile time*. Examples of statically typed languages would be **Java, C, and C++**. In a dynamically typed language, a single variable may take on different types *at runtime*. Examples would be **JavaScript and Python**. Can you think of any others?

OCaml is garbage collected, like _____ and _____ but unlike _____ and _____.
Garbage collection is a system that finds blocks of memory that will no longer be used by the program and reclaims their space for later use. **Java and Python are garbage collected. C++, C and Rust do not garbage collect!** You need to free memory manually when you are done with it (or in the case of Rust, the borrow checker takes care of that for you without doing any expensive garbage collection!).

Values in OCaml are immutable, like _____ variables in C++.
const variables

OCaml uses type inference, like C++'s _____ keyword but everywhere and more powerful. (C++ doesn't infer templates while OCaml infers polymorphic functions, and OCaml can type more intricate programs).
auto keyword

You can write optional type annotations in OCaml like in _____. But unlike _____, types are always checked and enforced at compile time.
Fun fact: type annotations actually are sometimes necessary, but only for advanced OCaml features outside the scope of CS 131.
Python

(15 mins) Talking Types 🗣🗣

`option` is a standard OCaml type like `list`. It is polymorphic like `list`, meaning you could have a `string option` or `int option` or even `int option option`.

`option` has two variants `Some` and `None` that represent the presence or absence of a value. Its type declaration is this:

```
type 'a option =
  | None
  | Some of 'a
```

You create `option` values with the constructors `None` and `Some x` where `x` is the value you want to "wrap" with the `option`. You can pattern match against them like this:

```
let is_some x =
  match x with
```

```
  | Some(value) -> ...
  | None -> ...
```

`option` is used for a similar purpose as `null` in Java or `nullptr` in C++: to represent "no value". But `option` is a safer option because it eliminates the uncertainty of "is this reference/pointer *actually* a reference/pointer, or is it that special `null` value that will break all my code if I try to dereference it?".
In OCaml when you have a value you *really do* have that value, no exceptions.

OCaml's type notation is very important to know. Fill in the blanks.

There may be multiple answers. To check your answers, run your statement in the OCaml interpreter.

Value	Type
`3`	int
`[None; Some "hi"]`	string option list option and list are "polymorphic variants", with the same purpose as C++ templates. You have a _something_ list (written `... list`) like how C++ has a vector _of something_ (written `vector<...>`). We know this value is a list because of the square brackets and semicolon. So it's a 'a list. To determine 'a, we look at the values inside the list, which are 'b options. So it's a 'b option list To determine 'b, we look at the value inside the Some, which is the string "hi". So 'b = string and it's a string option list. In grand French tradition, OCaml writes things backwards from English (C++). The C++ analogue would be list<option<string>>.

let g = fun x -> x You can think of 'a as a placeholder for a type. In fact they are called **type variables** in analogy to regular variables. Nobody can actually *have* a value of type 'a; rather 'a **corresponds to the spot where a type would go.** Because 'a -> 'a doesn't specify what 'a is, the function with type 'a -> 'a can't really *do* anything with its parameter (or else OCaml's type system may be able to identify the provided argument as a more specific type rather than a type variable). The identity function (which just returns its parameter) is one example, and we write it as fun x -> x.	`'a -> 'a`
`let always_true _ = true`	‘a -> bool _ commonly means "don't care". Here we don't care what the parameter is—but note there still is a parameter. Because always_true doesn't do anything with its parameter, it is just 'a which is as general as you can get. This is an important idea to understand: OCaml's type system will essentially infer the most general type it can when it comes to function parameters.

`let flatten opt =`
`  match opt with`
`  | Some Some x -> Some x`
`  | _ -> None`
 	
Or equivalently:
 	
`let flatten opt =`
`  match opt with`
`  | Some x -> x`
`  | _ -> None`
	`'a option option -> 'a option`
	(you could name this function "flatten")

let swap (a, b) = (b, a) swap takes one parameter. It then immediately **destructures** that parameter which is how OCaml determines it must be a 2-tuple.	`('a * 'b) -> ('b * 'a)` (you could name this function "swap")

We never do anything *between* a and b, so OCaml keeps their types separate. That's why the parameter's type is ('a * 'b) which is more general than ('a * 'a). Finally OCaml tracks the variables a and b, and sees that swap returns a 2-tuple (b, a). So it determines that the function evaluates to a value of type ('b * 'a).	

`let is_some x =`
`  match x with`
`  | Some(_) -> true`
`  | _ -> false`
	'a option -> bool
	
	This demonstrates how OCaml infers types based on their context. It sees that we match x against the pattern Some(_), so x must be an 'a option.
	
	Note that the function never does anything with the value inside the Some, so OCaml does not restrict it any further than 'a.
	
	We are left with 'a option -> bool.

let opt_map f opt =
  match opt with
  | None -> None
  | Some x -> Some (f x)
 	
Here one of the parameters to opt_map is itself a function. The parentheses around ('a -> 'b) are significant because without them there would be two parameters, one 'a and one 'b, instead of a function parameter as we want.
	`('a -> 'b) -> 'a option -> 'b option`
	(you could name this function "opt_map")

`let compose f g x =` `  f (g x)`	The type of `compose`: `('a -> 'b) -> ('c ->'a) -> 'c -> 'b` The variables f, g, x are related by function application. Because we call (g x) OCaml knows that g must take a parameter that is the same type—'c—as x. Because we call (f (g x)) OCaml knows that f must take a parameter that is the same type—'a—as (g x). And because we return (f (g x)) OCaml knows that compose returns the same type—'b—as (f (g x)).

	The type of `compose f g` where `f` is `int -> int` and `g` is `string -> int`: string -> int We "line up" f with the first parameter to compose, int -> int = 'a -> 'b and "line up" g with the second parameter to compose. string -> int = 'c -> 'a This tells us that 'a is int, 'b is int and 'c is string. Now after compose has been given two parameters, it has received the ('a -> 'b) part and the ('c -> 'a) part that are expected by (‘a -> ’b) -> (‘c ->’a) -> ‘c -> ‘b, so what remains is 'c -> 'b which is string -> int. What is the type of `compose (+)`? Why is passing `(+)` allowed given the type signature of `compose`? ('c -> int) -> 'c -> int -> int Note first that (+) has type int -> int -> int. Like before, "line up" (+) with the first parameter to compose, int -> int -> int = 'a -> 'b This is interesting because the left side has two arrows and the right side has one. But it's OK, since a function type is still a type. So we determine that 'a is int and 'b is int -> int. Therefore compose (+) has type ('c -> int) -> 'c -> int -> int. (technically it's a weak type, not 'c, but that's out of 131 scope)
let and_then f opt = match opt with | None -> None | Some x -> f x	`('a -> 'b option) -> 'a option -> 'b option` (Rust calls this function "and_then", the description of which may be helpful!)
`(+.)`	float -> float -> float

`(=)` (try to do this one by using the REPLI and seeing what works instead of looking it up)	‘a -> ‘a -> bool Remember we can check 1 = 1 just as well as "hi" = "bye", so (=) must be a polymorphic function. But we can't check "hi" = 1, so both parameters to (=) must be the same type. Therefore it's 'a -> 'a -> bool and not 'a -> 'b -> bool. Try this out in the REPL!
`List.cons` (`List.cons a b` is `a::b`) (interestingly `::` isn't an operator but an infix type constructor; `(::)` is not a valid expression)	‘a -> ’a list -> ’a list
let filter f x = match x with | None -> None | Some x -> if f x then Some x else None	`('a -> bool) -> 'a option -> 'a option` (if you think about it, options are like lists with length exactly 0 or exactly 1, so this is kinda like List.filter)

If `f` has type `'a -> ('b -> 'c) -> 'd`, and `f x y` typechecks,
What is the type of `x`? ‘a
What is the type of `y`? `('b -> 'c) or ‘b -> ‘c`
What is the type of `f x y`? ‘d

If `f` has type `'a -> ('a -> 'c) -> bool list`, and `f x y` typechecks,
What is the type of `x`? ‘a
What is the type of `y`? `('a -> 'c) or ‘a -> ‘c`
What is the type of `f x y`? `bool list`

(15 mins) Pattern Matching

The function `two_or_more` when applied to a `list` returns true if its argument contains 2 or more elements and false otherwise.

What is the type of `two_or_more`? 'a list -> bool
We only care about the length of the lists and not the contents of them, so this function should work for any 'a.

Implement `two_or_more` using `List.length`.

```
let two_or_more xs = List.length xs >= 2
```

Why is this implementation of `two_or_more` inefficient?

This implementation forces the length of the list to be calculated, which is an O(n) operation. We can do better than this as we only care whether there are 2 or more elements (i.e. a list with 2 elements and a list with 1000 are effectively the same to us).

Implement `two_or_more` again using pattern matching, and avoid the efficiency problem this time.

```
let two_or_more xs = match xs with
  | _ :: _ :: _ -> true
  | _ -> false
```

You know how to match on a list with one or more elements:

```
  match xs with
    | h :: t -> ...
    | _ -> ...
```

For checking two or more elements, we can phrase it as:

- xs has a head and a tail, and
- the tail of xs has one or more elements.

You could do this with a match on xs and then a nested match on t:

```
match xs with
    | h :: t ->
      match t with
      | h2 :: t2 -> true
      | _ -> false
    | _ -> false
```

But a cleaner approach is to use the pattern

```
  first :: second :: rest
```

which puts the pattern `second :: rest` *inside* the pattern `first :: ...`

And of course we can use "don't-care" _ everywhere since we're only interested in the *structure/shape* of the list, not the values contained in it. Thus the pattern `_ :: _ :: _`.

Say we extend this to a function `n_or_more` that takes two arguments; the first is an integer and the second is a list. `n_or_more 2 xs` is equivalent to `two_or_more xs`. Implement `n_or_more` using pattern matching and recursion. Hint: you can create then immediately match on a tuple: `match x, y with ...`.

```
let rec n_or_more n xs = match n, xs with
  | 0, _ -> true
```

```
  | _, [] -> false
  | n, _ :: tail -> n_or_more (n - 1) tail
  | _ -> false
```

We will do pattern matching over more complicated types next week.

Further practice

Read the problems we didn't get to and try them out (typing them into your IDE or a REPL can really help!) If you don't know how you would solve them, all the more reason to try! You're welcome to post on Piazza or ask your peers.

Start Homework 1 and attempt to implement the set functions without the `List` module especially if you haven't had much exposure to functional patterns before.

Check out the Homework 1 hints and think about `filter_blind_alleys`.

If you want to develop locally for CS 131, get a head start on software installation with the "local installation software links" pinned in Piazza.