Records and Objects

We will start by discussing records, a convenient way of arranging information together. Later on we combine records with closures and mutation to discuss concepts related to objects and object-oriented programming.

Records

A **record** is a bit like a tuple, except that each component is identified by a "name" instead, typically called a *field*. It must start with a lowercase letter. Here is an example:

```
{ a=4; b="hi!"; c=[1; 2] }
```

This record would have a *record type*:

```
{ a: int; b: string; c: int list }
```

One peculiar feature in OCAML is that a record type needs to be given a type alias first before a value of its type can be formed. So for example we can do:

```
type t = { a: int; b: string; c: int list }
let rcrd = { a=4; b="hi!"; c=[1; 2] }
```

Accessing the values in a record can be done in two ways. The one is a pattern match:

```
let { a=a; b=b; c=c } = rcrd in c
(* Convenient form when using var names same as fields *)
let { a; b; c } = rcrd in c
```

The other is with a "record access" notation:

```
rcrd.c
```

In general when records are concerned, it is best that their types are specified beforehand rather than inferred. For instance if all the information we have about a record is the expression `rcrd.c`, then all we know is that it has a field "c" and not much else. Type reconstruction becomes a bit harder. We shall revisit this issue when we discuss subtyping.

Objects

Even though OCAML natively supports objects, we will instead approach the subject from an exploratory direction: What are the key language constructs needed in order to have "objects"? What are objects anyway?

- Objects bring together data (**member variables**) and code (**methods**) that acts on this data.

- In most cases objects are organized in "classes". These classes often house "class variables" that all objects can access, as well as the object methods (occasionally also class methods).
- The method to be called depends on the object and is determined at runtime via a process known as **dynamic dispatch**. This is effectively a form of ad-hoc polymorphism.
- Objects typically provide **encapsulation**: From "outside" the object can only access the member variables via the object's methods. This encourages decoupling.
- Most (but not all) object-oriented languages allow **inheritance**, where the objects of one class can "inherit" from another class, having the same method signatures but perhaps overriding some of the methods and also adding new methods. In general an object from a subclass can be used anywhere that an object from its superclass is expected. On a static-typed system this would require subtyping, which OCAML does not really provide. We will look at some minimal examples of inheritance now, and will revisit the topic again when we discuss Racket.

We can already do some of the above with the tools we have so far:

- We can expose the object's methods by providing a record, since functions are first class values that can be stored anywhere any other kind of value can.
- We can make sure that the closure of the functions contains the member variables. Since the closure is not directly accessible to the rest of the program, we achieve encapsulation. We can do this via a "constructor" function, that takes as input whatever information is needed to create the object, then creates any other variables that are needed before returning a record of the object's methods.
- Class variables can be established by having another function that creates the constructor function. Then the closure of the constructor function can contain any "class variables".
- Using references for those variables allows the methods to mutate the object's variables.

Here is perhaps the simplest object we can imagine, a "counter":

```
type counterObj = { incr: unit -> unit; value: unit -> int; reset: unit -> unit }
let makeCounter init =
    let r = ref init
    in {
        incr = (fun () -> r := !r + 1);
        value = (fun () -> !r);
        reset = (fun () -> r := init)
    }
```

Here is an interaction with these objects:

```
let c1 = makeCounter 5
c1.value()      (* ---> 5 *)
c1.incr()
c1.value()      (* ---> 6 *)
c1.reset()
c1.value()      (* ---> 5 *)
```

Another key characteristic is that methods can call other methods from the object, possibly some defined in a superclass. We will see other ways of doing this in the future, but for now we can use recursion, and have the record we return be part of the closure of the functions in it:

```
type counterObj = { incr: unit -> unit; value: unit -> int;
                    add: int -> unit; reset: unit -> unit }
let makeCounter init =
    let r = ref init in
    let rec o = {
            incr = (fun () -> o.add 1);
            value = (fun () -> !r);
            add = (fun i -> r := !r + i);
            reset = (fun () -> r := init)
        }
    in o
```

We will revisit objects later when we discuss Racket.

Practice

1. Create a constructor for "account" objects. An account object should keep a "balance" as, a floating point number, that is initialized to 0, and should allow one to "deposit" a positive amount or "withdraw" a positive amount as long as the balance would not go below 0. It should also have a "checkBalance" method to report the current balance.
2. Create a constructor for "bank" objects. A bank object contains a list of accounts, starting with an empty list, and supports the following operations:
    - Creating (and returning) a new account (and also updating the list of accounts)
    - Computing the number of accounts
    - Computing the total balance in all accounts
    - Increasing all accounts by a `r` interest rate (e.g. `r=0.05` for a 5% increase in all accounts)