Algebraic Data Types

Exercise 1. Due to Yaron Minsky.

Consider a datatype to store internet connection information. The time `when_initiated` marks the start of connecting and is not needed after the connection is established (it is only used to decide whether to give up trying to connect). The ping information is available for established connection but not straight away.

```
type connection_state =
  | Connecting
  | Connected
  | Disconnected

type connection_info = {
  state : connection_state;
  server : Inet_addr.t;
  last_ping_time : Time.t option;
  last_ping_id : int option;
  session_id : string option;
  when_initiated : Time.t option;
  when_disconnected : Time.t option;
}
```

(The types `Time.t` and `Inet_addr.t` come from the library *Core* used where Yaron Minsky works. You can replace them with `float` and `Unix.inet_addr`. Load the Unix library in the interactive toplevel by `#load "unix.cma";;`.) Rewrite the type definitions so that the datatype will contain only reasonable combinations of information.

Exercise 2. In OCaml, functions can have named arguments, and also default arguments (parameters, possibly with default values, which can be omitted when providing arguments). The names of arguments are called labels. The labels can be different from the names of the argument values:

```
let f ~meaningful_name:n = n+1
let _ = f ~meaningful_name:5                    We do not need the result so we ignore it.
```

When the label and value names are the same, the syntax is shorter:

```
let g ~pos ~len =
  StringLabels.sub "0123456789abcdefghijklmnopqrstuvwxyz" ~pos ~len
let () =                          A nicer way to mark computations that do not produce a result (return unit).
  let pos = Random.int 26 in
  let len = Random.int 10 in
  print_string (g ~pos ~len)
```

When some function arguments are optional, the function has to take non-optional arguments after the last optional argument. When the optional parameters have default values:

```
let h ?(len=1) pos = g ~pos ~len
let () = print_string (h 10)
```

Optional arguments are implemented as parameters of an option type. This allows us to check whether the argument was actually provided:

```
let foo ?bar n =
  match bar with
    | None -> "Argument = " ^ string_of_int n
    | Some m -> "Sum = " ^ string_of_int (m + n)
;;
foo 5;;
foo ~bar:5 7;;
```

We can also provide the option value directly:

```
let bar = if Random.int 10 < 5 then None else Some 7 in
foo ?bar 7;;
```

1. Observe the types that functions with labelled and optional arguments have. Come up with coding style guidelines, e.g. when to use labeled arguments.
2. Write a rectangle-drawing procedure that takes three optional arguments: left-upper corner, right-lower corner, and a width-height pair. It should draw a correct rectangle whenever two arguments are given, and raise exception otherwise. Load the graphics library in the interactive toplevel by `#load "graphics.cma";;`. Use "functions" `invalid_arg`, `Graphics.open_graph` and `Graphics.draw_rect`.
3. Write a function that takes an optional argument of arbitrary type and a function argument, and passes the optional argument to the function without inspecting it.

Exercise 3. From last year's exam.

1. Give the (most general) types of the following expressions, either by guessing or inferring by hand:
   a. `let double f y = f (f y) in fun g x -> double (g x)`
   b. `let rec tails l = match l with [] -> [] | x::xs -> xs::tails xs in fun l -> List.combine l (tails l)`
2. Give example expressions that have the following types (without using type constraints):
   a. `(int -> int) -> bool`
   b. `'a option -> 'a list`

Exercise 4. We have seen in the class, that algebraic data types can be related to analytic functions (the subset that can be defined out of polynomials via recursion) – by literally interpreting sum types (i.e. variant types) as sums and product types (i.e. tuple and record types) as products. We can extend this interpretation to all OCaml types that we introduced, by interpreting a function type $a \to b$ as b^a, b to the power of a. Note that the b^a notation is actually used to denote functions in set theory.

1. Translate a^{b+cd} and $a^b\,(a^c)^d$ into OCaml types, using any distinct types for a, b, c, d, and using the `('a,'b) choice = Left of 'a | Right of 'b` datatype for $+$. Write the bijection function in both directions.
2. Come up with a type `'t exp`, that shares with the exponential function the following property: $\frac{\partial \exp(t)}{\partial t} = \exp(t)$, where we translate a derivative of a type as a context, i.e. the type with a "hole", as in the lecture. Explain why your answer is correct. Hint: in computer science, our logarithms are mostly base 2.

Further reading:
http://bababadalgharaghtakamminarronnkonnbro.blogspot.com/2012/10/algebraic-type-systems-combinatorial.html