1 Defining New Types and Exceptions

```ocaml
type button = On | Off;;

let press_button b =
  match b with
  On -> Off
| Off -> On;;
```

Functions defined in terms of pattern matching can also use the `function` construct. Ocaml correctly infers the function type.

```ocaml
let press_button2 = function
  On -> Off
| Off -> On;;
```

Let’s look at a linguistic example.

```ocaml
type category = Noun | Verb | Preposition | Adjective | Adverb | Determiner | Complementizer;;

let able (s,c) =
  match (s,c) with
  (x,Verb) -> (x" able", Adjective)
| _ -> (s,c) ;;
```

Really, we’d like to prevent able from applying if its given a string that is not of type Verb. We can use exceptions to achieve this.

```ocaml
exception WrongKindOfCategory;;

let able (s,c) =
  match (s,c) with
  (x,Verb) -> (x" able", Adjective)
| _ -> raise WrongKindOfCategory;;
```

Exercise 1 Write the `able` function using the `function` construct above.

```ocaml
let able = function
  (x,Verb) -> (x" able", Adjective)
| _ -> raise WrongKindOfCategory;;
```

Exceptions can also take a string as an argument using the `of` construct.
exception WrongKindOfCategory of string ;;

let able = function
  (x, Verb) => (x""able", Adjective)
| _ => raise (WrongKindOfCategory("able requires Verbs"));

A pre-defined exception you can always use is the Failure exception which is defined as Failure of string.

let able = function
  (x, Verb) => (x""able", Adjective)
| _ => raise (Failure("able requires Verbs"));

Exercise 2 Write 5 more functions so that well-formed morphological combinations have their category correctly paired with their surface form, as in

# (ism (anti (arian (dis (ment("establish", Verb)))))));
- : string * category = ("antidisestablishmentarianism", Noun)

2 Records

Another useful type are records. These are like tuples, except we identify the parts of the data object by names instead of coordinate position.

type car = {model: string; year: int};;

let mycar = {model="sienna"; year=2006};;

We can access aspects of the record as follows.

mycar.model ;;
mycar.year ;;

Let’s examine the useful filter function for lists.

let carlist = [{model="sienna"; year=2006}; {model="T"; year=1904}; {model="sienna"; year=2009}];;

let is_siena a = a.model = "sienna";;
Exercise 3 Define a function filter which behaves exactly as List.filter does.

3 Recursive Types

Here we define a new kind of object – a “list structure.” It can either have the value Empty, or Cons(x, y), where x is an object of type 'a, and y is a list structure of objects of type 'a.

```plaintext
type 'a list_structure = Empty | Cons of 'a * 'a list_structure;;
```

This is how to use list structures. Note that the cases in the match with construct line up exactly with the cases in the definition of a list structure.

```plaintext
let rec length ls = match ls with
  | Empty -> 0
  | Cons (l, ll) -> 1 + length ll;;
```

This is a tail-recursive (i.e. better) implementation of the above function. It carries information about how many items it has seen so far with it in an extra argument, which means that we can perform the addition step after each call to the function, instead of having to wait until we finish traversing the list. This makes a big difference if our lists are loooong. Try it and see.

```plaintext
let alt_length ls =
  let rec length_help n ls = match ls with
    | Empty -> n
    | Cons (l, ll) -> length_help (n+1) ll
  in
  length_help 0 ls;;
```

Exercise 4 Write the List.hd function in terms of the 'a list_structure data type. Don’t forget to define an exception for the case of the empty list!

Exercise 5 Write the List.tl function in terms of the 'a list_structure data type.

Exercise 6 Write the List.map function in terms of the 'a list_structure data type.

Exercise 7 Write the List.fold_left function in terms of the 'a list_structure data type.

Exercise 8 Define a new data type for binary tree structures.