Problem 1 (20 points). Using pattern matching, write a function called `dayname` with type `int -> string`, such that `dayname n` evaluates to the name of the 0-indexed n-th day of the week—i.e., such that:

```
  dayname 0 ↓ "Monday"
  dayname 1 ↓ "Tuesday"
  dayname 2 ↓ "Wednesday"
  dayname 3 ↓ "Thursday"
  ...
```

As long as you account for the seven days of the week, your match need not be exhaustive on `ints`, but this should be specified in the comments (which must be provided to document your code).

Problem 2 (30 points). Let a month offset be the number of days of the week after Monday that a given month begins; for example, February 1 falls on a Saturday this year, so its month offset is 5. Using the `dayname` function defined in the previous problem and the `List.map` function discussed in class, write a function called `add_daynames` that takes a month offset and a list of dates `l` of that month, and returns another list consisting of pairs `(d, n)`, where `n` is the day of the week corresponding to `d`. We can specify this function as follows:

```
(*
  add_daynames : int -> int list -> (int * string) list
  in : a month offset o, list of calendar dates
  l = [d1, ..., dk]
  out : list [(d1, dni1), ..., (dk, dni)], where for all
  i in {1, ..., k}, dni = dayname di given o
*)
```

For example:

```
add_daynames 5 [1;2;3;10;11]
⇓
[(1, "Saturday"); (2, "Sunday"); (3, "Monday"); (10, "Monday"); (11, "Tuesday")]
```

If you use `List.map` correctly, `add_daynames` can be written on a single line of code; this level of elegance should be your goal. 

*Hint:* The OCaml standard library includes an infix integer modulo operator called `mod`, which should be useful.
Problem 3 (25 points). Recalling that datastructures can have nested structure, we can easily imagine that “flattening” is a useful operation. For example, given a list of lists, we might want to obtain a list of all the elements in the nested lists, which would flatten the structure. So, given an appropriately defined polymorphic function flatten:

\[
\text{flatten } \begin{bmatrix} [2;3];[7;9] \end{bmatrix} \Downarrow [2; 3; 7; 9] \\
\text{flatten } \begin{bmatrix} ['a'];[\ ];['c';'d';'g'] \end{bmatrix} \Downarrow ['a'; 'c'; 'd'; 'g']
\]

Thus, flatten has the following type signature:

\[
\text{flatten : 'a list list -> 'a list}
\]

The specification of flatten is as follows:

\[
(* \\
\text{flatten : 'a list list -> 'a list} \\
in : list of ('a lists) l1 through ln \\
out : an 'a list consisting of all elements in l1 through ln \\
*)
\]

In this problem, you are to define flatten according to these specifications.

Problem 4 (25 points). Another useful polymorphic list-oriented function is member, with the following type signature:

\[
\text{member : 'a -> 'a list -> bool}
\]

The specification of member is as follows:

\[
(* \\
\text{member : 'a -> 'a list -> bool} \\
in : value v, list l \\
out : true iff v is in l \\
*)
\]

In other words, member checks whether a given element is a member of a list. For example:

\[
\text{member 1 [1;5;7] \Downarrow true} \\
\text{member 'a' ['c';'d';'b'] \Downarrow false}
\]

In this problem, you are to define member according to these specifications.