Pattern matching functions

Pattern matching is a technique for elegantly handling cases in code. It is an alternative to conditionals, which are hard to read when there are more than two cases. Pattern matching also simultaneously and concisely binds values to variables.

Problem statement: Write a function get_nth that takes a list of strings and an integer n and returns the n^{th} element of the list where the head of the list is defined as the 1st element.

What happens when the list is empty or when n is greater than the length of the list? As a practical matter we need to address these cases. Here, I use a very simple, type-safe error handling mechanism called an option type. The type is defined as:

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

The option type is an example of an algebraic data type. option lets us concisely signal when a function is undefined without having to invoke an expensive error-handling mechanism like exceptions.

```
let rec get_nth xs n =
  match xs, n with
  | [], _ -> None
  | y::ys, 1 -> Some y
  | y::ys, n' -> if n' > 1 then get_nth ys (n' - 1) else None
```

To handle defined/undefined outputs, the user of the get_nth function uses pattern matching on its output.

```
let n = 2
let xs = ["hello"; "dear"; "cs334"]
let nth = get_nth xs n
match nth with
| Some x -> printfn "The %dth element is %s." n x
| None   -> printfn ("The %dth element in a list of length %d" +
                    " does not make sense.") n (List.length xs)
```

Folding

Folding is a technique for aggregating values drawn from complex data structures in ML (where "complex" simply means that the data structure is made from multiple simple pieces) like lists and trees. Folding can be used to obtain a value for any operation that can be performed by structural recursion on a data structure.

Problem statement: A date is a value of type int*int*int, where the 1st number is a year, the 2nd number is a month, and the 3rd number is a day. Write a function number_in_month that takes a list of dates and a month (an `int`) and returns how many dates are in the list for the given month. Use `fold` to solve this problem.

Since `fold` is likely new to you, let's start by thinking about what the task looks like without `fold`. Here is a recursive solution.

```
let rec number_in_month dates month =
  match dates with
  | [] -> 0
  | (_,m,_)::dates' ->
    (if m = month then 1 else 0) + (number_in_month dates' month)
```

To see how this fits into the `fold` pattern, note the presence of: (1) a base case, and (2) an accumulated value.

In the base case, we have an empty list, so clearly, there are no matching months and we return 0.

In the inductive case, we either have a matching month or we do not. If the month matches, add one, otherwise, add 0. In the above code, we do not explicitly maintain an accumulator variable, but we could have. Instead the sum is maintained implicitly by building a stack of additions using recursion.

Note the use of the pattern `((_,m',_)::dates)` to extract the month `m'` from the list of dates. The use of `_` means that, structurally, an element must appear (i.e., there must be a three-tuple `(_,m',_)`), but that the only value we want to bind to a variable is the middle element, `m'`.

Recall the definition for `List.fold`:

```
f: ('a -> 'b -> 'a) -> initial_value: 'a -> xs: 'b list -> 'a
```

In plain language, `List.fold` is a function that takes a function `f`, an initial value, and a list of values. The function `f` should be a function that takes an accumulator value (of the same type as the initial value) and an element of the list.

Let's refactor our `number_in_month` function to use `List.fold`.

```
let number_in_month dates month : int =
  List.fold (fun acc (_,m,_) ->
    acc + (if m = month then 1 else 0)
  ) 0 dates
```

or, using the *forward pipe* operator, the slightly more elegant

```
let number_in_month dates month : int =
  dates |>
    List.fold (fun acc (_,m,_) ->
      acc + (if m = month then 1 else 0)
    ) 0
```