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 1^{st} element.

In class, I told you not to worry about cases where the list was empty or where n was greater than the length of the list. As a practical matter, though, we do need to address it. Here, I use a very simple, type-safe error handling mechanism called an `option` type. The type is defined as:

```
datatype 'a option = 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.

```
fun get_nth nil n = NONE
 |  get_nth (x::xs) 1 = SOME x
 |  get_nth (x::xs) n =
      if n > 1 then get_nth xs (n-1) else NONE
```

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

```
...
val nth = get_nth n xs
case nth of
  SOME x => print ("The " ^ (Int.toString n)
            ^ "th element is " ^ x ^ "\n")
| NONE   => print ("The " ^ (Int.toString n)
            ^ "th element in a list of length "
            ^ (Int.toString (length xs))
            ^ " does not make sense.\n")
...
```

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.

```
fun number_in_month [] m = 0
 |  number_in_month ((_,m',_)::dates) m =
    (if m' = m then 1 else 0)
    + number_in_month dates m
```

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 `fold`:

```
fold <function of 'a * accumulator> <initial accumulator value> <list of 'a>
```

Let's refactor this code to use `fold`.

```
fun number_in_month dates m : int =
  foldl (fn ((_,m',_),acc) =>
    acc + (if m' = m then 1 else 0)) 0 dates
```