State Recursion

Additional reading on recursion: https://gist.github.com/skiadas/15f2e6f82b1cb3a41f39

Last time we made a distinction between “normal recursion” and “state recursion”; as well as a distinction between “numerical recursion” and “structural recursion”. The two examples of normal recursion we saw last time were as follows:

```
let rec sum n =
  if n = 1
  then 1
  else sum (n - 1) + n
```

and

```
let rec list_sum lst =
  match lst with
  | [] -> 0
  | hd :: tl -> hd + list_sum tl
```

The key characterstic in both cases is that we build the answer out of a recursive call on a “smaller” problem together with the contribution from the current value.

There is a different approach, familiar from languages that focus on iteration and for-loops: We use an accumulator, and as we go through each value we add its contribution to the accumulator. We can do something similar for via recursion as well. It typically involves the use of a *helper (auxiliary) function*. Here is the first example in this setting:

```
let rec sum n =
  let rec aux (sofar, k) =
    if k > n
    then sofar
    else aux (sofar + k, k + 1)
  in aux (0, 1)
```

Here is how evaluation of say `sum 3` would go:

```
sum 3
aux (0, 1)
aux (0 + 1, 1 + 1)
aux (1, 2)
aux (1 + 2, 2 + 1)
aux (3, 3)
aux (3 + 3, 3 + 1)
aux (6, 4)
6
```

The key is the helper method: It takes as input the current “counter”, `k`, as well as the sum of the values so far, stored in the variable `sofar`. In the recursive call we update those values, and call the function with the updated values. When we have reached the last value, we can read off the sum from the `sofar` variable.

A similar approach would work for the `list_sum` example:

```
let list_sum lst =
  let rec aux (sofar, remaining) =
    match remaining with
    | [] -> sofar
    | x :: rest -> aux (sofar + x, rest)
  in aux (0, lst)
```

Here the auxiliary function keeps track of the sum of the already computed values, and the remainder of the list. As long as the list is nonempty, it grabs the next element, adds its contribution to the current sum, then calls itself with the updated sum and list. Computation would go as follows:

```
list_sum [1; 2; 3]
aux (0, [1; 2; 3])
aux (0 + 1, [2; 3])
aux (1, [2; 3])
aux (1 + 2, [3])
aux (3, [3])
aux (3 + 3, [])
aux (6, [])
6
```

In general you should avoid this state recursion. But some problems do call for it. This method is essentially iteration with an accumulator, in disguise: The arguments passed to the auxilliary function contain the accumulator and the "iterated variable". The resulting value is synthesized one step at a time, rather than at the end after all recursive calls have been completed. Programmers coming from procedural languages find it comforting, because of this relation to normal looping constructs. You should resist the temptation to do all recursion this way, in fact avoid using it unless you have a really good reason to do so.

Practice problems

Some of these problems favor a state recursion approach, others favor a normal recursion.

1. Write a function rev that given a list of integers returns a list of the same integers in reverse order.
2. Write a function isSquare that is given as input an integer n and tries to see if it is a square by trying the numbers 1 through n and seeing if any of them have n as their square. Return a boolean true if n is a perfect square, false otherwise. Your code should stop the moment it confirms the number is a perfect square (for example for n = 9, it should stop when it reaches 3.
3. Write a function maxMaybe that takes as input a list of integers and returns an option integer: None if the list was empty, Some v otherwise, where v is the largest integer in the list. Do not rely on any of OCAML's built-in functions.
4. Write a function someMax that takes as input a list of int-options and returns an int option as follows: It returns either Some m where m is the maximum of all the integers present in the list (in the form of Some i entries in the list) or None if there are no such integers (so if the list is empty or contains only Nones).

5. Write a function sortedInsert that takes as input a pair of a number n and a list containing integers and assuming they are in increasing order (e.g. [1; 3; 3; 3; 4]), and it returns a new list where the number n has been inserted in the appropriate spot in order to preserve the increasing order. In the case where the integer already exists in the list, you must add a new entry, and you have a choice on where to add it. For instance in the example above a new 3 would technically be insertable in four different spots, and they are all equivalent.
6. Write a function seq that takes as input a pair of integers (a, b) and returns a list of all the integers from a to b (empty list if a > b). For instance seq (2, 5) = [2; 3; 4; 5].
7. Write a function noMultiples that is given as input a pair of an integer n and a list of integers lst, and it returns a list of those elements from lst that are not perfectly divisible by n (and in the same order). You can look at the remainder of division of a by b via a mod b.
8. Write a function primes that is given as input an integer n and returns a list of all the prime numbers up to n. You should use seq to first create a list of all numbers from 2 to n, then use noMultiples along with recursion. You will probably need to also create a helper function within primes to handle this recursion (it does NOT have to be state recursion, but it does need a helper function). At each step of the process you should be able to take out the next prime number, remove its multiples, and continue recursively.