Recursion

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

Recursion is the bread and butter of many functional programming paradigms. At its most basic, recursion simply means that a function is called from within itself, or from a function that was called from within itself. This creates the possibility of an "infinite loop", so some care needs to be taken to ensure that the process terminates.

There are two dimensions, each with two different options, for how recursion is used, resulting in 4 combinations:

1. Input type: This looks at what kinds of input we are dealing with and what our stopping conditions are.
    1. **Numerical recursion** deals with a numerical input, that increases or decreasing over time, and with some stopping conditions when e.g we reach 0.
    2. **Structural recursion** deals with input that is some complicated structure, e.g. a list. The recursive call then relates to the subcomponents of the structure, and the stopping condition has to do with base forms of that type (e.g. empty lists in the example of lists).
2. Recursion type: This looks at how the input to the recursive call relates to the input of the original call.
    1. **Normal recursion**: We express the value for the current input in terms of values for "previous/smaller" inputs.
    2. **State recursion** or *accumulated recursion*: Part of the input is an accumulated state, that we update in the recursive call.

Before we look at examples, one important thing to keep in mind is that, unlike other programming languages you may be used to, function activation records/frames in OCAML are not stored in the stack, but instead in the heap. There is therefore no stack limit on recursion.

We will consider "Normal recursion" in this set of notes, and "State recursion" in the next.

Numerical Recursion

We will start by examining what we called numerical recursion above. The standard example in this case is the following: Given a number n, assumed to be positive, compute the sum of all integers from 1 to n.

The "Normal recursion" way goes as follows: To sum all the integers from 1 to n, we first add those up to $n-1$, and to that result we add n. The stopping condition is when n is 1, in which case we return 1. So the code would look something like this:

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

The keyword rec lets OCAML know that sum is a recursively defined function, and so it should expect to see a call to sum in the body and that sum should refer to the function itself (rather than a previous binding of sum).

Let us see how the computation of sum 3 might go:

```
sum 3
(sum 2) + 3
((sum 1) + 2) + 3
(1 + 2) + 3
3 + 3
6
```

Note that the addition operation indicated in the call to sum, in the "(sum 2) + 3" part, is not going to actually happen until later. In other words, the computation of sum 3 has been suspected, awaiting for the call sum 2 to be completed, before doing the final + 3 part. Similarly, at the next step, the call to sum 2 gets suspended waiting for sum 1 to be completed. These calls are then completed in reverse order. These suspended computations, are a key characteristic of the "normal recursion".

Practice problem. Modify the above code to solve the following problem: Given a pair of numbers (a, b) where we can assume $a \leq b$, find the sum of all numbers from a up to and including b.

Practice problem 2. Same question but now we might have $a > b$. We still want the sum of all numbers, in this case from b to a.

Structural Recursion

Let us imagine a similar problem: We are given a list of numbers, namely a value of type int list. We want to add them, with the result being 0 if we add no numbers at all. The idea is similar and emblematic of structural recursion: One possibility is the empty list, and then we can return 0; the other possibility is a non-empty list, which we can think of as a combination of the first element (known as the *head*) along with the list consisting of the remaining elements (known as the *tail*). We can recursively determine the sum of elements in the tail, and add to that the result in the head.

Here's a way to implement it, using the match–with form:

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

This shows us how we use the match–with mechanic to do "structural decomposition": The first clause is our base case of an empty list, while the second clause matches a nonempty list. Here is how evaluation would proceed for list_sum [1; 2; 3]:

```
list_sum [1; 2; 3]
1 + list_sum [2; 3]
1 + (2 + list_sum [3])
1 + (2 + (3 + list_sum []))
1 + (2 + (3 + 0))
1 + (2 + 3)
1 + 5
6
```

Notice the same "suspended computations": all the additions occur at the end, after the innermost recursive call completes.

This is also emblematic of how we work with lists: We match on the form of the list; we handle the base case of an empty list in one clause, while the other clause decomposes the list into the head element and the (smaller) tail.

To close, let us discuss the general elements of *recursion via structural decomposition*:

1. Your structures come in two forms, "compound" and "atomic". In the case of lists, the atomic form is the empty list, while the compound form is the prepend operation `a :: rest`.
2. In a `match–with` clause you handle the atomic cases by returning a value, and you handle the compound cases by recursively calling your function on the substructures, then combining those values appropriately.

List practice

We will now implement various standard list problems as recursive functions. Make sure to first write out the type of the function, before implementing it.

1. A function `sq` that given a list of integers returns a list with the squares of these integers, in the same order.
2. A function `hd` that given a list of integers returns the head element. If the list is empty, it raises an exception. You can do this by the line `raise (Failure "hd")`.
3. A function `tl` that given a list of integers returns the tail list. If the list is empty, it raises an exception `Failure "tl"`.
4. A function `any` that is given a list of booleans and returns `true` if any one of them is true. It returns `false` for an empty list.
5. A function `all` that is given a list of booleans and returns `false` if all of them are true. It returns `true` for an empty list.
6. A function `append` that takes a pair of int lists `(l1, l2)` and returns the list consisting of the elements of `l1` followed by the elements of `l2`. For example `append ([1;2;3], [4;5;6]) = [1;2;3;4;5;6]`.
7. A function `increasing` that takes a list of integers and returns `true` if those integers appear in increasing order. This one is a bit tricky.