Recursion

In this section we discuss various aspects of recursive functions. We assume that students already have a working knowledge of the basics of recursion.

Reading

- Sections 6.1-6.6
- Practice exercises (6.8): 2, 3, 4, 6, 7, 8

What is Recursion

Recursion is a fundamental building block of functions, and its idea is simple. Let us recall the basic idea behind a function:

> A function takes inputs of a certain type, and produces outputs of a certain type, and we need to specify how those outputs relate to the inputs.

The key logic that separates one function from another is how the function processes those inputs, and how it reacts to different kinds of inputs. We have already seen one key technique for writing functions in the form of pattern-matching and guarded expressions: We consider different *forms* that the input values can take, and act accordingly.

Recursion builds on this idea, but with a more specific idea in mind.

> Recursive functions express their outputs based on their input by taking into account the outputs they would produce on "simpler inputs". A recursive function gives itself a simpler version of the problem it has to answer. It builds on the resulting output to produce its own output.

All recursive functions contain three key elements:

- A description of which cases are "simpler" than others.
- A description of what the "base cases", the simplest cases are, and what the output is in those cases.
- A description on how to compute the output for the given input by using the outputs for some smaller inputs. These are the "recursive cases".

As a simple example, consider a function that adds all the numbers from a list:

```
sum :: Num t => [t] -> t
sum []     = 0
sum (x:xs) = x + sum xs
```

In this example, we have one base case, that of the empty list. And the result is 0. We also have a recursive case. If our function has the form of a non-empty list (x:xs), then we call ourselves on the "simpler" input xs (simpler because it has one less value in it). Obtaining that result up, we then combine it with the x value to get our final answer.

So in this example the "simpler cases" are cases with smaller lists. Once we know how to do those, we can also handle longer lists.

A different example of the same idea is trying to add up all the numbers from 1 to n. We could do this as follows:

```
addUpTo n :: Int -> Int
addUpTo n | n <= 0       = 0
          | n > 0        = n + addUpTo (n-1)
```

So in this example we have a base case of n<=0. And a recursive case which decreases the n by 1. The "simpler cases" here amount to smaller values of the integer input n.

To understand how the evaluation of the above function would go, here's a typical run for n=3:

```
addUpTo 3
3 + addUpTo 2
3 + (2 + addUpTo 1)
3 + (2 + (1 + addUpTo 0))
3 + (2 + (1 + 0))
3 + (2 + 1)
3 + 3
6
```

The book contains many more examples of recursive functions, I urge you to study them. We will now discuss the different patterns exhibited by these examples.

Different Forms of Recursion

Recursive functions can be broadly categorized in two different dimensions. This results in four different cases.

Input classification

The simplest distinguishing aspect is the kind of input value. This fits broadly in two types:

- **Numerical** recursion has inputs which are numbers. We typically recurse to "simpler cases" by making the number smaller, until it reaches something like 0. The function addUpTo is a good example of this kind of recursion.

- **Structural** recursion has inputs that are more complex data structures themselves built in a recursive manner from smaller structures. Working through a list like in the example of sum is a good example of structural recursion.

State classification

A recursive function can also be categorized in how it handles "state". Is there some information passed from one level of the recursion to the next, or not?

- **Stateless** recursion, the most normal kind of recursion, does not communicate any kind of state from one function call to the next. A function simply calls itself on a "smaller" input, reads back the output and perhaps changes it a bit before returning itself. Both the addUpTo function and the sum function above are examples of stateless recursion.
- **Stateful** recursion maintains some sort of accumulated "state" for the problem, and passes it on from function to function during the recursive calls. We have not really seen examples of this recursion yet, but we are about to do so. Stateful recursion often requires a helper function that receives the extra accumulated-state parameter, and often tends to be tail recursive.

As an example of stateful recursion, we will implement the earlier functions using stateful recursion instead.

For the addUpTo function, we consider a *helper* function, addHelper. This function takes two arguments: The number n and a second number, sumSoFar, which keeps track of what the sum of the numbers is so far. The addHelper function then simply adds n to this sum, then decrements n and calls itself again. When n reaches 0, the function reads the answer from sumSoFar. Finally the addUpTo function simply kick-starts the process by calling addHelper and giving it an initial sum of 0. Here's how the code would look like:

```
addHelper :: Int -> Int -> Int
addHelper n sumSoFar | n <= 0       = sumSoFar
                     | n > 0        = addHelper (n - 1) (n + sumSoFar)

addUpTo :: Int -> Int
addUpTo n = addHelper n 0
```

Here's how a typical computation might go.

```
addUpTo 3
addHelper 3         0
addHelper (3 - 1) (3 + 0)
addHelper 2         (3 + 0)
addHelper (2 - 1) (2 + (3 + 0))
addHelper 1         (2 + (3 + 0))
addHelper (1 - 1) (1 + (2 + (3 + 0)))
addHelper 0         (1 + (2 + (3 + 0)))
1 + (2 + (3 + 0))
1 + (2 + 3)
```

```
1 + 5
6
```

This looks a bit awkward because of the *lazy evaluation* nature of Haskell. In a so-called *strict evaluation* language, the same steps would look as follows:

```
addUpTo 3
addHelper 3       0
addHelper (3 - 1) (3 + 0)
addHelper 2       3
addHelper (2 - 1) (2 + 3)
addHelper 1       5
addHelper (1 - 1) (1 + 5)
addHelper 0       6
6
```

The key aspect to note here is that there are no "unfinished computations". If we compare to the stateless addUpTo example from earlier, that example had a series of additions that had to occur at the end, once the function stopped calling itself. Not so in this case, at least if lazy evaluation was not an issue. This is one of the main advantages of stateful recursion.

If the function addHelper is not needed on its own, then we can define it within a where clause to avoid exposing its name on the rest of the program:

```
addUpTo :: Int -> Int
addUpTo n = addHelper n 0
    where addHelper n sumSoFar | n <= 0      = sumSoFar
                               | n > 0       = addHelper (n - 1) (n + sumSoFar)
```

In general it is a good idea to define elements *as locally as possible*.

Let us follow a similar approach for the sum function. We will use a helper that adds up the elements we have encountered so far. Then the function simply adds the current next element to that total, and recursively calls itself on the rest of the list. Here's how that would look like:

```
sum :: Num t => [t] -> t
sum lst = sumHelper lst 0
    where sumHelper [] sumSoFar     = sumSoFar
          sumHelper (x:xs) sumSoFar = sumHelper xs (x + sumSoFar)
```

Here's how execution of this function would have gone if we didn't have lazy evaluation:

```
sum [1, 2, 3]
sumHelper [1, 2, 3] 0
sumHelper [2, 3]    1
sumHelper [3]       3
sumHelper []        6
6
```

These examples should demonstrate a key fact:

> Stateful recursion in functional programming is similar to iteration/accumulation in imperative programming.

Namely in a stateful recursion you effectively do the following:

- Use an *accumulator* to keep track of your state/progress.
- Initialize the accumulator to a start value like 0 or an empty list.
- Go through each element and add its contribution to the accumulator.
- When you run out of elements, return the accumulator value.

Writing recursive functions

There is typically a standard set of steps to consider when writing recursive functions. We describe them here, and you will get to practice them in what follows:

- Start by considering and writing down the type of the function.
- If you decide that you need stateful recursion:
  - Consider the type of the helper function.
  - Write the initial call to the helper function from your main function.
  - Follow the next steps for the helper function instead of the main function.
- Consider and write down the different cases, typically at least a base case and a recursive case, but some times more start cases. As you do so, be mindful of the type of the function and what types the inputs would have.
- Write down definitions in each case. Be mindful of the return/output type of your function, as that is what type the expression for each case should return.
- Trace by hand how your function would behave on a small example. Make sure to follow exactly the rules you have written for your function, and not how you *think* it should behave.

We will try to follow these steps to write a function reverse, that takes in a list and reverses it. We will first write a straightforward but quite inefficient method using stateless recursion. Then we'll write a faster method using stateful recursion. The second method works better because of the particular nature of the problem.

We start with the type of the function, and the different cases we should consider:

```
reverse :: [t] -> [t]
reverse []       = ...
reverse (x:xs)   = ...
```

We make not that the dotted parts must have type [t]. We will keep this in mind as we work out the logic for them.

The empty list case is simple: The reverse of the empty list would be the list itself.

The case of a non-empty list is a little more complicated, but basically we have to do the following: Reverse the tail of the list (xs), then append x to the end. Since there is no way to append an element to the end of a list, we would instead have to concatenate two lists together:

```
reverse :: [t] -> [t]
reverse []       = []
reverse (x:xs)   = reverse xs ++ [x]
```

We next trace out how this would work on a particular example:

```
reverse [1, 2, 3]
reverse [2, 3] ++ [1]
(reverse [3] ++ [2]) ++ [1]
((reverse [] ++ [3]) ++ [2]) ++ [1]
(([] ++ [3]) ++ [2]) ++ [1]
([3] ++ [2]) ++ [1]
[3, 2] ++ [1]
[3, 2, 1]
```

This unfortunately is quite inefficient. The reason for this has to do with how list append works: xs ++ ys requires that we create a copy of the list xs.

Let us now consider a reverse2 function that avoids list concatenations. It does so by recognizing that a helper function can help us add the elements in the final list one at a time, since the first element in the input list needs to end up being the last element of the output list. We can therefore start with something like this:

```
reverse2 :: [t] -> [t]
reverse2 lst = reverse2' lst []
  -- Type of reverse2':  [t] -> [t] -> [t]
  where reverse2' []     ys   = ...
        reverse2' (x:xs) ys   = ...
```

Now we need to fill in the dotted parts. The ys holds the already reversed part of the list, and the xs is the list we are processing. If that list is already empty, then we have reversed the entire list, so ys is our final asnwer. Otherwise, we just need to move the first element, x, over to the ys, and we continue the recursion with the tail:

```
reverse2 :: [t] -> [t]
reverse2 lst = reverse2' lst []
  -- Type of reverse2':  [t] -> [t] -> [t]
  where reverse2' []     ys   = ys
        reverse2' (x:xs) ys   = reverse2' xs (x:ys)
```

Practice problems

1. Write a function normalExpo that given a base b and an exponent e compute the power b^e by only using multiplication. You should do this by induction on the exponent e until it drops to 0, in which case the answer is 1 ($b^0 = 1$).

2. Write a function fastExpo that given a base b and an exponent e compute the power b^e just using basic multiplication but trying to do fewer multiplications. It does so as follows:

   - The case of e=0 is the base case, with result 1.
   - If the exponent e is even, e=2d, then it uses the fact that $b^{2d} = (b^2)^d$ to instead compute the result of raising the number b*b to the d power (this would be a recursive call).
   - If the exponent e is odd, e=2d+1, then it uses the formula $b^{2d+1} = b \cdot (b^2)^d$ to instead compute the result of raising the number b*b to the d power, but then multiplying the final answer by b.

3. Write a function unzip that given a list of pairs returns a pair of lists, essentially doing the opposite of zip.

4. Write a function splitWith :: (a -> Bool) -> [a] -> ([a], [a]) that takes a list and a predicate and returns a pair of lists consisting respectively of those elements that return True/False under the predicate.