Recursion

Johan Montelius

November 2, 2017

Simple arithmetic functions

Implement the following arithmetic functions and save them in a file `recursion.erl`. Start with simple recursive implementations that are not necessarily tail-recursive. In order not to create any conflicts with builtin function, you will have to call your functions something different i.e. prod instead of product etc.

When you implement them: write a small documentation, write the type definition and the definition, then try them using GHCi. For example:

```
-module(recursion).

%% product of n and m :
%%   if n is 0
%%          then ....
%%     otherwise
%%          the result ...


prod(.., ..) ->
    case ...  of
       ... ->  ...;
       ... ->  ...
    end.
```

- prod: complete the above implementation of product
- prod: change the above definition so that it also works for negative numbers
- power: implement the power function for non-negative exponents
- qpower: use the two functions *div* and *rem* to implement a faster version of the power function by dividing the exponent by two in each recursive step (what must be done if the exponent is odd?)

Binary

Implement a function `binary/1` that takes an integer and produces its binary encoding, represented ad a list of ones and zeros. Your firts solution should look something like this:

```
binary(0) ->
    [];
binary(...) ->
    binary(...) ++ [...].
```

Note that the `++` operator is just a short form of writing `append/2`. What is the deficiency of this implementation. What is the runtime complexity?

Change your implementation so that it has a *O(n)* komplexity. The solution will look something like this:

```
binary2(I) ->
    binary(I, ...).

binary2(0, Sofar) ->
    lists:reverse(Sofar);
binary2(..., ....) ->
    binary2(..., ...).
```

We're using the library function `lists:reverse/1` to reverse the acumulated value.

Fibonacci

Try some more complex functions, for example the Fibonacci function:

$$fib(n) = \begin{cases} 0 & \text{if } n \text{ is } 0 \\ 1 & \text{if } n \text{ is } 1 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

The straight forward implementation is a bit inefficient, when evaluating fib(n-1) in the recursion we of course learn fib(n-2) as a by product but this is not taken advantage of. Can you change the implementation so that it only computes each Fibonacci number once?

Ackermann

You can also give the Ackermann function a try:

$$ackerman(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ ackerman(m-1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ ackerman(m-1, ackerman(m, n-1)) & \text{otherwise} \end{cases}$$

This looks like an innocent little function but don't try too high numbers :-)

list processing

Open up new module and implement the following list processing functions (again we will have to call them something not that obvious in order not to create a conflict with the functions in Prelude):

- drp: drop the first n elements of a list
- tak: take the first n elements of a list
- append: append two lists
- rev: reverse a list
- palindrome: return $True$ if a list is a palindrome.

sorting

The final assignment will be to implement the *merge sort* algorithm. Open up a new file `Sort.hs` and start with a new module.

```
module Sort
       where
```

To implement the algorithm we need some additional tools. The algorithm is straight forward i.e. split a list in two, sort the two parts and then merge them together, but the tricky issue is how to write a function that returns two list. The simplest way to do this is to return a tuple that consist of two elements. The syntax for tuples is `(a,b)` and the split function has the following type:

```
split :: Ord a => [a] -> ([a], [a])
```

To implement `split` we might want to use a help function, a version of split that takes three arguments: the rest of the list that we should split, half of the elements that we have seen and the other half. Call this function `distribute` and then define `split` as follows:

```
split xs = distribute xs [] []
```

The next function we need is the `merge` function. It should take two sorted lists and merge them into one sorted lists. If one of the two lists is empty he other list is of course the result, if not we must compare the two first elements of the two lists.

```
merge :: Ord a => [a] -> [a] -> [a]
merge [] ys = ...
merge xs [] = ...
merge (x:xs) (y:ys) = ..... | ....
merge (x:xs) (y:ys) = ..... | ....
```

Time to wrap it up, we now need a construct that lets us first split the list and then work on the two sub lists. This is done using a `let in` expression. The expression allows us to perform one or more function calls to evaluate results that we need in our main expression.

```
msort :: Ord a => [a] -> [a]
msort [] = ...
msort [x] = ...
msort xs =
   let
     (as, bs) = split xs
   in
     merge ... ...
```

I this definition we use the let expression to get direct access to the two list that `split` returns. The `(as,bs)` on the right hand side works as a pattern that the result is matched with. The variables `as` and `bs` will as a result be bound to the two lists.

If you have time you can try to implement quick sort. You might find that an `if` expression comes handy:

```
   if x < p then (x:lo, hi) else (lo, x:hi)
```