Reduce and friends

Johan Montelius

Spring Term 2024

Introduction

In this exercise you're going to explore to power of higher order functions and how you can use them. Your first task is to implement a set of functions using regular recursive programming and then, as a second task, implement them using your higher order functions.

Recursive functions

Let's keep it simple in the beginning and work with lists of integers. Your first task is to implement these functions using regular recursive programming:

- `length([integer()]) :: integer()` : return the length of the list.
- `even([integer()]) :: [integer()]` : return a list of all even number.
- `inc([integer()], integer()) :: [integer()]` : return a list where each element of the given list has been incremented by a value.
- `sum([integer()]) :: integer()` : return the sum of all values of the given list.
- `dec([integer()], integer()) :: [integer()]` : return a list where each element of the given list has been decremented by a value.
- `mul([integer()], integer()) [integer()]` : return a list where each element of the given list has been multiplied by a value.
- `odd([integer()]) :: [integer()]` : return a list of all odd number.

- `rem([integer()], integer()) :: [integer()]` : return a list with the result of taking the reminder of dividing the original by some integer.
- `prod([integer()]) :: integer()` : return the product of all values of the given list (what is the product of all values of a an empty list?)
- `div([integer()], integer()): [integer()]` : return a list of all numbers that are evenly divisible by some number.

These functions should not be too hard to implement even though you might feel like you're doing the same thing over and over again. If you group them into three categories you will find three different patterns; identify the patterns before you proceed.

map, reduce and filter

The first pattern is where we traverse through the list and returns a new list where we some how calculated a new value given the original value. This pattern is called *map* and will be you first function to implement.

- `map([a()], ( a() -> b())) :: [b()]` : take a list of values and a function that when applied to a value returns another value. Create a new list with the resulting values when you apply the function on each value in the original list.

The second pattern is slightly more complex in that it will reduce the original list of integers to one single value. Your implementation of `sum/1` added all the numbers together while the function `mul/1` multiplied them. If you look at your functions you probably came up with a solution where the sum of all integers in an empty list i zero and (which of course could be discussed) the product of all values in an empty list is one. We thus need not only provide the function but also an initial value to know what to return if we reduce an empty list.

- `reduce([a()], b(), (a(), b() -> b())) :: b()` : apply the function on the value of the list and the accumulated value. Return the final value.

You may now realize that we have two different ways of implementing this function and the way you choose probably depends on how you chose to implement the functions `sum/1` and `prod/1`. Is your implementation *tail recursive*? If we should provide a function that could be used with very large input we could go through the trouble to write the function as a tail

recursive function, or rather why not implement two versions of the function one tail recursive (`reducel`) and one that is not (`reducer`).

The third function is one that will select a few values from the given list and we call this `filter/2`. The function is given a list of values and a function that will return either `true` or `false` and determine if we should add the value to the list returned.

- `filter([a()], (a() -> boolean())) :: [a()]` : apply the function to all values of the given list and return a list with those values for which the function returned true.

Also this function can be implemented i two different ways; which? There is of course a third way if we do not have to maintain the order of the values - would that be more efficient?

Now that you have your higher order functions, re-implement the functions of the first task. Also implement a function that takes a list of integers and returns the sum of the square of all values less than n.

The final twist is to use the *pipe operator* (`|>`). The operator can be used to pass the returned value of one function call to the first argument of the next function call. Take for example the following definition:

```
def test(lst, x, y) do
   lst_one = one(lst, x)
   lst_two = two(lst_one, y)
   three(lst_two)
end
```

This could of core be written as follows:

```
def test(lst, x, y) do
   three(two(one(lst, x), y))
end
```

This is more compact but the question if often if it's easy to quickly see what is going on. Using the pipe operator we can write the function as follows:

```
def test(lst, x, y) do
   lst |>
     one(x) |>
     two(y) |>
     three()
end
```

This syntax saves us from inventing variables names, its rather compact while not as complicated to understand what is happening. The pipe operator should be used with care and only when it makes a difference.