Exercise sheet 2

Max Wisniewski, Alexander Steen

Task 1

We first prove a lemma that is being used in the induction proof of the task's proposition:
Lemma: `f (foldNat' f e n) =foldNat' f (f e) n`.
Proof by induction:
Base: `n = O`

```
    f (foldNat' f e O)
= {Def.}
    f e
= {Def. inverse}
    foldNat' f (f e) O
```

Step: `n` $\rightsquigarrow$ `S n`

```
    f (foldNat' f e (S n))
= {Def.}
    f (foldNat' f (f e) n)
= {Induction Hyp.}
    foldNat' f (f (f e)) n
= {Def. inverse}
    foldNat' f (f e) (S n)
```

□

Proposition: `foldNat f e =foldNat' f e`.
Proof by induction:
Base: `n = O`

```
    foldNat f e O
= {Def.}
    e
= {Def. inverse}
    foldNat' f e O
```

Step: `n` $\rightsquigarrow$ `S n`

```
    foldNat f e (S n)
= {Def.}
    f (foldNat f e n)
= {Induction Hyp.}
    f (foldNat' f e n)
= {Lemma}
    foldNat' f (f e) n
= {Def. inverse}
    foldNat' f e (S n)
```

□

Task 2

Both solutions work essentially the same way. In `fact2` we construct the tupel explizitly. We compute the product of the current number and of the previous computed factorial.

```
fact1 :: Nat → Nat
fact1 = paraNat ((uncurry  times) ∘ (first S)) (S O)

fact2 :: Nat → Nat
fact2 = snd ∘ (foldNat' (λ(x,y) → (S x, times (S x) y)) (O, S O))
```

Both functions perform a linear amount of multiplikations. In the second we construct the number along the way, but this is only linear time.

In Haskell both are even in the place needed to store the functions. Both store up to n times functions.

In a strict language on the otherhand `fact2` can execute the `times` in each iterations never holding more than one in the background. Therefor a constant stack of functions has to be stored.

Task 3

```
lengthL :: [a] → Nat
lengthL = foldr (λ_ b → S b) O
```

We ignore the list element and just replace each `(:) a` with a `S`. The empty list is replaced by a `O`. We build up the successor for any element in the list hence we computed the length.

```
sumL :: [Nat] → Nat
sumL = foldr plus O

prodL :: [Nat] → Nat
prodL = foldr times (S O)
```

Both these functions work the same. We gave it the neutral element of the operation and then apply the operation between the elements of the list starting with the neutral element.

We apply all functions a linear number of times.

Task 4

```
headL :: [a] → Maybe a
headL =  paraList (λa _ → Just a) Nothing
```

If the list is empty `Nothing` is returned. In the other case `paraList` gets the head as first input, hence we simply return it packed in a Maybe.

```
tailL :: [a] → Maybe [a]
tailL = paraList (λ _ (as,_) → Just as) Nothing
```

Tail essentially works the same way, just that paraList has put the tail of the list as second element in the second argument.

Both functions take constant time because we will throug away the recursive step and because Haskell is performing a outmost reduction the recursion will not be executed.