Isabelle/HOL Exercises
Trees, Inductive Data Types

Folding Lists and Trees

Some more list functions

Recall the summation function

```
  sum :: "nat list ⇒ nat"
primrec
  "sum [] = 0"
  "sum (x # xs) = x + sum xs"
```

In the Isabelle library, you will find (in the theory `List.thy`) the functions *foldr* and *foldl*, which allow you to define some list functions, among them *sum* and *length*. Show the following:

```
lemma sum_foldr: "sum xs = foldr (op +) xs 0"
  apply (induct xs)
  apply auto
done

lemma length_foldr: "length xs = foldr (λ x res. 1 + res) xs 0"
  apply (induct xs)
  apply auto
done
```

Repeated application of *foldr* and *map* has the disadvantage that a list is traversed several times. A single traversal is sufficient, as illustrated by the following example:

```
lemma "sum (map (λ x. x + 3) xs) = foldr h xs b"
```

Find terms *h* and *b* which solve this equation.

```
lemma "sum (map (λ x. x + 3) xs) = foldr (λ x y. x + y + 3) xs 0"
  apply (induct xs)
  apply auto
done
```

Generalize this result, i.e. show for appropriate *h* and *b*:

```
lemma "foldr g (map f xs) a = foldr h xs b"
```

Hint: Isabelle can help you find the solution if you use the equalities arising during a proof attempt.

```
lemma "foldr g (map f xs) a = foldr (λx acc. g (f x) acc) xs a"
  apply (induct xs)
  apply auto
done
```

The following function `rev_acc` reverses a list in linear time:

```
consts
  rev_acc :: "['a list, 'a list] ⇒ 'a list"
primrec
  "rev_acc [] ys = ys"
  "rev_acc (x#xs) ys = (rev_acc xs (x#ys))"
```

Show that `rev_acc` can be defined by means of `foldl`.

```
lemma rev_acc_foldl_aux [rule_format]:
  "∀a. rev_acc xs a = foldl (λ ys x. x # ys) a xs"
  apply (induct xs)
  apply auto
done

lemma rev_acc_foldl: "rev_acc xs a = foldl (λ ys x. x # ys) a xs"
  by (rule rev_acc_foldl_aux)
```

Prove the following distributivity property for `sum`:

```
lemma sum_append [simp]: "sum (xs @ ys) = sum xs + sum ys"
  apply (induct xs)
  apply auto
done
```

Prove a similar property for `foldr`, i.e. something like `foldr f (xs @ ys) a = f (foldr f xs a) (foldr f ys a)`. However, you will have to strengthen the premises by taking into account algebraic properties of `f` and `a`.

```
constdefs
  left_neutral :: "['a ⇒ 'b ⇒ 'b, 'a] ⇒ bool"
  "left_neutral f a == (∀ x. (f a x = x))"
  assoc :: "['a ⇒ 'a ⇒ 'a] ⇒ bool"
  "assoc f == (∀ x y z. f (f x y) z = f x (f y z))"

lemma foldr_append:
  "⟦ left_neutral f a; assoc f ⟧ ⟹ foldr f (xs @ ys) a = f (foldr f xs a)
(foldr f ys a)"
```

```
  apply (induct xs)
    apply (simp add: left_neutral_def)
  apply (simp add: assoc_def)
done
```

Now, define the function *prod*, which computes the product of all list elements

```
  prod :: "nat list ⇒ nat"

defs
  prod_def: "prod xs == foldr (op *) xs 1"
```

directly with the aid of a fold and prove the following:

```
lemma "prod (xs @ ys) = prod xs * prod ys"
  apply (simp only: prod_def)
  apply (rule foldr_append)
    apply (simp add: left_neutral_def)
  apply (simp add: assoc_def)
done
```

Functions on Trees

Consider the following type of binary trees:

```
datatype 'a tree = Tip | Node "'a tree" 'a "'a tree"
```

Define functions which convert a tree into a list by traversing it in pre-, resp. postorder:

```
  preorder :: "'a tree ⇒ 'a list"
  postorder :: "'a tree ⇒ 'a list"

primrec
  "preorder Tip          = []"
  "preorder (Node l x r) = x # ((preorder l) @ (preorder r))"

primrec
  "postorder Tip          = []"
  "postorder (Node l x r) =  (postorder l) @ (postorder r) @ [x]"
```

You have certainly realized that computation of postorder traversal can be efficiently realized with an accumulator, in analogy to *rev_acc*:

```
consts
  postorder_acc :: "['a tree, 'a list] ⇒ 'a list"

primrec
```

```
  "postorder_acc Tip          xs = xs"
  "postorder_acc (Node l x r) xs = (postorder_acc l (postorder_acc r (x#xs)))"
```

Define this function and show:

```
lemma postorder_acc_aux [rule_format]:
  "∀xs. postorder_acc t xs = (postorder t) @ xs"
  apply (induct t)
  apply auto
done

lemma "postorder_acc t xs = (postorder t) @ xs"
  by (rule postorder_acc_aux)
```

postorder_acc is the instance of a function *foldl_tree*, which is similar to *foldl*.

```
consts
  foldl_tree :: "('b => 'a => 'b) ⇒ 'b ⇒ 'a tree ⇒ 'b"

primrec
  "foldl_tree f a Tip          = a"
  "foldl_tree f a (Node l x r) = (foldl_tree f (foldl_tree f (f a x) r) l)"
```

Show the following:

```
lemma "∀ a. postorder_acc t a = foldl_tree (λ xs x. Cons x xs) a t"
  apply (induct t)
  apply auto
done
```

Define a function *tree_sum* that computes the sum of the elements of a tree of natural numbers:

```
consts
  tree_sum :: "nat tree ⇒ nat"

primrec
  "tree_sum Tip          = 0"
  "tree_sum (Node l x r) = (tree_sum l) + x + (tree_sum r)"
```

and show that this function satisfies

```
lemma "tree_sum t = sum (preorder t)"
  apply (induct t)
  apply auto
done
```