Isabelle/HOL Exercises
Lists

Quantifying Lists

Define a universal and an existential quantifier on lists using primitive recursion. Expression *alls P xs* should be true iff *P x* holds for every element *x* of *xs*, and *exs P xs* should be true iff *P x* holds for some element *x* of *xs*.

```
primrec alls :: "('a ⇒ bool) ⇒ 'a list ⇒ bool" where
  "alls P []     = True"
| "alls P (x#xs) = (P x ∧ alls P xs)"

primrec exs  :: "('a ⇒ bool) ⇒ 'a list ⇒ bool" where
  "exs P []     = False"
| "exs P (x#xs) = (P x ∨ exs P xs)"
```

Prove or disprove (by counterexample) the following theorems. You may have to prove some lemmas first.

Use the *[simp]*-attribute only if the equation is truly a simplification and is necessary for some later proof.

```
lemma "alls (λx. P x ∧ Q x) xs = (alls P xs ∧ alls Q xs)"
  apply (induct "xs")
  apply auto
done

lemma alls_append: "alls P (xs @ ys) = (alls P xs ∧ alls P ys)"
  apply (induct "xs")
  apply auto
done

lemma "alls P (rev xs) = alls P xs"
  apply (induct "xs")
  apply (auto simp add: alls_append)
done

lemma "exs (λx. P x ∧ Q x) xs = (exs P xs ∧ exs Q xs)"
  quickcheck
⋮
```

A possible counterexample is: P = even, Q = odd, xs = [0, 1]

```
lemma "exs P (map f xs) = exs (P o f) xs"
  apply (induct "xs")
  apply auto
done

lemma exs_append: "exs P (xs @ ys) = (exs P xs ∨ exs P ys)"
  apply (induct "xs")
  apply auto
done

lemma "exs P (rev xs) = exs P xs"
  apply (induct "xs")
  apply (auto simp add: exs_append)
done
```

Find a (non-trivial) term *Z* such that the following equation holds:

```
lemma "exs (λx. P x ∨ Q x) xs = Z"
lemma "exs (λx. P x ∨ Q x) xs = (exs P xs ∨ exs Q xs)"
  apply (induct "xs")
  apply auto
done
```

Express the existential via the universal quantifier – *exs* should not occur on the right-hand side:

```
lemma "exs P xs = Z"
lemma "exs P xs = (¬ alls (λx. ¬ P x) xs)"
  apply (induct "xs")
  apply auto
done
```

Define a primitive-recursive function *is_in x xs* that checks if *x* occurs in *xs*. Now express *is_in* via *exs*:

```
primrec is_in :: "'a ⇒ 'a list ⇒ bool" where
  "is_in x []      = False"
| "is_in x (z#zs) = (x=z ∨ is_in x zs)"

lemma "is_in a xs = exs (λx. x=a) xs"
  apply (induct "xs")
  apply auto
done
```

Define a primitive-recursive function `nodups xs` that is true iff `xs` does not contain duplicates, and a function `deldups xs` that removes all duplicates. Note that `deldups [x, y, x]` (where `x` and `y` are distinct) can be either `[x, y]` or `[y, x]`.

```
primrec nodups :: "'a list ⇒ bool" where
  "nodups []     = True"
| "nodups (x#xs) = (¬ is_in x xs ∧ nodups xs)"

primrec deldups :: "'a list ⇒ 'a list" where
  "deldups []     = []"
| "deldups (x#xs) = (if is_in x xs then deldups xs else x # deldups xs)"
```

Prove or disprove (by counterexample) the following theorems.

```
lemma "length (deldups xs) <= length xs"
  apply (induct "xs")
  apply auto
done

lemma is_in_deldups: "is_in a (deldups xs) = is_in a xs"
  apply (induct "xs")
  apply auto
done

lemma "nodups (deldups xs)"
  apply (induct "xs")
  apply (auto simp add: is_in_deldups)
done

lemma "deldups (rev xs) = rev (deldups xs)"
  quickcheck
⋮
```

A possible counterexample is: xs = [0, 1, 0]