Isabelle/HOL Exercises

1 Counting occurences

Define a function *occurs*, such that *occurs x xs* is the number of occurrences of the element *x* in the list *xs*.

consts *occurs :: "'a ⇒ 'a list ⇒ nat"*

Prove or disprove (by counter example) the lemmas that follow. You may have to prove additional lemmas first. Use the *[simp]*-attribute only if the equation is truly a simplification and is necessary for some later proof.

lemma *"occurs a xs = occurs a (rev xs)"*
lemma *"occurs a xs <= length xs"*

Function *map* applies a function to all elements of a list: *map f* $[x_1, \ldots, x_n]$ = $[f\ x_1, \ldots, f\ x_n]$.

lemma *"occurs a (map f xs) = occurs (f a) xs"*

Function *filter :: ('a ⇒ bool) ⇒ 'a list ⇒ 'a list* is defined by

```
filter P [] = []
filter P (x # xs) = (if P x then x # filter P xs else filter P xs)
```

Find an expression *e* not containing *filter* such that the following becomes a true lemma, and prove it:

lemma *"occurs a (filter P xs) = e"*

With the help of *occurs*, define a function *remDups* that removes all duplicates from a list.

consts *remDups :: "'a list ⇒ 'a list"*

Find an expression *e* not containing *remDups* such that the following becomes a true lemma, and prove it:

lemma `"occurs x (remDups xs) = e"`

With the help of *occurs* define a function *unique*, such that *unique xs* is true iff every element in *xs* occurs only once.

consts `unique :: "'a list ⇒ bool"`

Show that the result of *remDups* is *unique*.

2 Tree traversal

Define a datatype *'a tree* for binary trees. Both leaf and internal nodes store information.

datatype `'a tree =`

Define the functions *preOrder*, *postOrder*, and *inOrder* that traverse an *'a tree* in the respective order.

consts

```
  preOrder :: "'a tree ⇒ 'a list"
  postOrder :: "'a tree ⇒ 'a list"
  inOrder :: "'a tree ⇒ 'a list"
```

Define a function *mirror* that returns the mirror image of an *'a tree*.

consts

```
  mirror :: "'a tree ⇒ 'a tree"
```

Suppose that *xOrder* and *yOrder* are tree traversal functions chosen from *preOrder*, *postOrder*, and *inOrder*. Formulate and prove all valid properties of the form `xOrder (mirror xt) = rev (yOrder xt)`.

Define the functions *root*, *leftmost* and *rightmost*, that return the root, leftmost, and rightmost element respectively.

consts

```
  root :: "'a tree ⇒ 'a"
  leftmost :: "'a tree ⇒ 'a"
  rightmost :: "'a tree ⇒ 'a"
```

Prove or disprove (by counter example) the lemmas that follow. You may have to prove some lemmas first.

lemma `"last(inOrder xt) = rightmost xt"`
lemma `"hd (inOrder xt) = leftmost xt"`

```
lemma "hd(preOrder xt) = last(postOrder xt)"
lemma "hd(preOrder xt) = root xt"
lemma "hd(inOrder xt) = root xt"
lemma "last(postOrder xt) = root xt"
```

3 Natural deduction

3.1 Propositional logic

The focus of this exercise are single step natural deduction proofs. The following restrictions apply:

- Only the following rules may be used:
  notI: (A ⟹ False) ⟹ ¬ A,
  notE: ⟦¬ A; A⟧ ⟹ B,
  conjI: ⟦A; B⟧ ⟹ A ∧ B,
  conjE: ⟦A ∧ B; ⟦A; B⟧ ⟹ C⟧ ⟹ C,
  disjI1: A ⟹ A ∨ B,
  disjI2: A ⟹ B ∨ A,
  disjE: ⟦A ∨ B; A ⟹ C; B ⟹ C⟧ ⟹ C,
  impI: (A ⟹ B) ⟹ A ⟶ B,
  impE: ⟦A ⟶ B; A; B ⟹ C⟧ ⟹ C,
  mp: ⟦A ⟶ B; A⟧ ⟹ B
  iffI: ⟦A ⟹ B; B ⟹ A⟧ ⟹ A = B,
  iffE: ⟦A = B; ⟦A ⟶ B; B ⟶ A⟧ ⟹ C⟧ ⟹ C
  classical: (¬ A ⟹ A) ⟹ A
- Only the methods `rule`, `erule` und `assumption` may be used.

```
lemma I: "A ⟶ A"
lemma "(A ∨ B) = (B ∨ A)"
lemma "(A ∧ B) ⟶ (A ∨ B)"
lemma "((A ∨ B) ∨ C) ⟶ A ∨ (B ∨ C)"
lemma K: "A ⟶ B ⟶ A"
lemma "(A ∨ A) = (A ∧ A)"
lemma S: "(A ⟶ B ⟶ C) ⟶ (A ⟶ B) ⟶ A ⟶ C"
lemma "(A ⟶ B) ⟶ (B ⟶ C) ⟶ A ⟶ C"
lemma "¬ ¬ A ⟶ A"
lemma "(¬ A ⟶ B) ⟶ (¬ B ⟶ A)"
lemma "((A ⟶ B) ⟶ A) ⟶ A"
lemma "A ∨ ¬ A"
```

3.2 Predicate logic

You may now use the followinh additional rules:

exI: $P\ x \Longrightarrow \exists x.\ P\ x$
exE: $[\![\exists x.\ P\ x;\ \bigwedge x.\ P\ x \Longrightarrow Q]\!] \Longrightarrow Q$
allI: $(\bigwedge x.\ P\ x) \Longrightarrow \forall x.\ P\ x$
allE: $[\![\forall x.\ P\ x;\ P\ x \Longrightarrow R]\!] \Longrightarrow R$

For each of the following formulae, find a proof or explain why it is not true.

lemma "$(\forall x.\ P\ x \longrightarrow Q) = ((\exists x.\ P\ x) \longrightarrow Q)$"
lemma "$(\forall\ x.\ \forall\ y.\ R\ x\ y) = (\forall\ y.\ \forall\ x.\ R\ x\ y)$"
lemma "$((\exists\ x.\ P\ x) \vee (\exists\ x.\ Q\ x)) = (\exists\ x.\ (P\ x \vee Q\ x))$"
lemma "$((\forall\ x.\ P\ x) \vee (\forall\ x.\ Q\ x)) = (\forall\ x.\ (P\ x \vee Q\ x))$"
lemma "$(\forall x.\ \exists y.\ P\ x\ y) \longrightarrow (\exists y.\ \forall x.\ P\ x\ y)$"
lemma "$(\exists x.\ \forall y.\ P\ x\ y) \longrightarrow (\forall y.\ \exists x.\ P\ x\ y)$"
lemma "$(\neg\ (\forall\ x.\ P\ x)) = (\exists\ x.\ \neg\ P\ x)$"

3.3 A puzzle

Prove the following proposition with pen and paper, possibly using case distinctions:

If every poor person has a rich father,
then there is a rich person with a rich grandfather.

theorem
"$\forall x.\ \neg\ rich\ x \longrightarrow rich\ (father\ x) \Longrightarrow$
$\exists x.\ rich\ (father\ (father\ x)) \wedge rich\ x$"

Translate your proof into a sequence of Isabelle rule applications. Case distinctions via `case_tac` are allowed.

4 Context-free grammars

This exercise is concerned with context-free grammars (CFGs). Please read section 7.4 in the tutorial which explains how to model CFGs as inductive definitions. Our particular example is about defining valid sequences of parantheses.

4.1 Two grammars

The most natural definition of valid sequences of parantheses is this:

$$S \quad \to \quad \varepsilon \quad | \quad '(' \; S \; ')' \quad | \quad S \; S$$

where ε is the empty word.

A second, somewhat unusual grammar is the following one:

$$T \quad \to \quad \varepsilon \quad | \quad T \; '(' \; T \; ')'$$

Model both grammars as inductive sets S and T and prove $S = T$.

4.2 A recursive function

Instead of a grammar, we can also define valid sequences of paratheses via a test function: traverse the word from left to right while counting how many closing paretheses are still needed. If the counter is 0 at the end, the sequence is valid.

Define this recursive function and prove that a word is in S iff it is accepted by your function. The $\Longrightarrow$ direction is easy, the other direction more complicated.