Isabelle/HOL Exercises
Logic and Sets

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 parentheses.

Two grammars

The most natural definition of valid sequences of parentheses 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 $S = T$.

The alphabet:

```
datatype alpha = A | B
```

Standard grammar:

```
inductive_set S :: "alpha list set" where
S1: "[] : S" |
S2: "w : S ⟹ A#w@[B] : S" |
S3: "v : S ⟹ w : S ⟹ v @ w : S"
```

```
declare S1 [iff] S2[intro!,simp]
```

Nonstandard grammar:

```
inductive_set T :: "alpha list set" where
T1: "[] : T" |
T23: "v : T ⟹ w : T ⟹ v @ A # w @ [B]: T"
```

```
declare T1 [iff]
```

T is a subset of *S*:

```
lemma T2S: "w : T ⟹ w : S"
  apply(erule T.induct)
    apply simp
  apply(blast intro: S3)
done
```

S is a subset of *T*:

```
lemma T2: "w : T ⟹ A#w@[B] : T"
  using T23[where v = "[]"] by simp

lemma T3: "v : T ⟹ u : T ⟹ u@v : T"
  apply(erule T.induct)
    apply fastsimp
  apply(simp add: append_assoc[symmetric] del:append_assoc)
  apply(blast intro: T23)
done

lemma S2T: "w : S ⟹ w : T"
  apply(erule S.induct)
      apply simp
    apply(blast intro: T2)
  apply(blast intro: T3)
done
```

S = T:

```
lemma "S = T"
  by(blast intro: S2T T2S)
```

A recursive function

Instead of a grammar, we can also define valid sequences of parentheses via a test function: traverse the word from left to right while counting how many closing parentheses 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.

```
consts balanced :: "alpha list * nat ⇒ bool"

recdef balanced "measure(%(xs,n). length xs)"
```

```
  "balanced ([], 0)       = True"
  "balanced (A#w, n)      = balanced(w,Suc n)"
  "balanced (B#w, Suc n) = balanced(w,n)"
  "balanced (w, n)        = False"
```

Correctness of the recognizer w.r.t. *S*:

```
lemma [simp]: "balanced(w,n) ⟹ balanced(w@[B],Suc n)"
  apply(induct w n rule:balanced.induct)
  apply simp_all
done

lemma [simp]: "⟦balanced (v, n); balanced (w, 0)⟧ ⟹ balanced (v @ w, n)"
  apply(induct v n rule:balanced.induct)
  apply simp_all
done

lemma "w : S ⟹ balanced(w,0)"
  apply(erule S.induct)
  apply simp_all
done
```

Completeness of the recognizer w.r.t. *S*:

```
lemma [iff]: "[A,B] : S"
  using S2[where w = "[]"] by simp

lemma AB: assumes u: "u ∈ S" shows "⋀v w. u = v@w ⟹ v @ A # B # w ∈ S"
using u
proof(induct)
  case S1 thus ?case by simp
next
  case (S2 u)
  have uS: "u ∈ S" and
       IH: "⋀v w. u = v @ w ⟹ v @ A # B # w ∈ S" and
       asm: "A # u @ [B] = v @ w" .
  show "v @ A # B # w ∈ S"
  proof (cases v)
    case Nil
    hence "w = A # u @ [B]" using asm by simp
    hence "w ∈ S" using uS by simp
    hence "[A,B] @ w ∈ S" by(blast intro:S3)
    thus ?thesis using Nil by simp
  next
    case (Cons x v')
```

```
    show ?thesis
    proof (cases w rule:rev_cases)
      case Nil
      from uS have "(A # u @ [B]) @ [A,B] ∈ S" by(blast intro:S3)
      thus ?thesis using Nil Cons asm by auto
    next
      case (snoc w' y)
      hence u: "u = v' @ w'" and [simp]: "x = A & y = B"
        using Cons asm by auto
      from u have "v' @ A # B # w' ∈ S" by(rule IH)
      hence "A # (v' @ A # B # w') @ [B] ∈ S" by(rule S.S2)
      thus ?thesis using Cons snoc by auto
    qed
  qed
next
  case (S3 v' w')
  have v'S: "v' ∈ S" and w'S: "w' ∈ S"
   and IHv: "⋀v w. v' = v @ w ⟹ v @ A # B # w ∈ S"
   and IHw: "⋀v w. w' = v @ w ⟹ v @ A # B # w ∈ S"
   and asm: "v' @ w' = v @ w" .
  then obtain r where "v' = v @ r ∧ r @ w' = w ∨ v' @ r = v ∧ w' = r @ w"
    (is "?A ∨ ?B")
    by (auto simp:append_eq_append_conv2)
  thus "v @ A # B # w ∈ S"
  proof
    assume A: ?A
    hence "v @ A # B # r ∈ S" using IHv by blast
    hence "(v @ A # B # r) @ w' ∈ S" using w'S by(rule S.S3)
    thus ?thesis using A by auto
  next
    assume B: ?B
    hence "r @ A # B # w ∈ S" using IHw by blast
    with v'S have "v' @ (r @ A # B # w) ∈ S" by(rule S.S3)
    thus ?thesis using B by auto
  qed
qed
```

The same lemma for friends of the apply style:

```
lemma "u ∈ S ⟹ ALL v w. u = v@w ⟶ v @ A # B # w ∈ S"
apply(erule S.induct)
  apply simp
 apply(rename_tac u)
 apply (clarsimp simp:Cons_eq_append_conv)
```

```
 apply(rule conjI)
  apply (clarsimp)
  apply(subgoal_tac "[A,B] @ (A # u @ [B]) : S")
   apply(simp)
  apply(blast intro:S3)
 apply(clarsimp simp:append_eq_append_conv2 Cons_eq_append_conv)
 apply(rename_tac w w1 w2)
 apply(erule disjE)
  apply clarsimp
  apply(subgoal_tac "A # (w1 @ A # B # w2) @ [B] : S")
   apply simp
  apply(blast intro:S3)
 apply clarsimp
 apply(erule disjE)
  apply clarsimp
  apply(subgoal_tac "A # (u @ [A,B]) @ [B] : S")
   apply(simp)
  apply(blast intro:S3)
 apply clarsimp
 apply(subgoal_tac "(A # u @ [B]) @ [A,B] : S")
  apply(simp)
 apply(blast intro:S3)
apply(clarsimp simp:append_eq_append_conv2)
apply(rename_tac u v w x y)
apply(erule disjE)
 apply clarsimp
 apply(subgoal_tac "(w @ A # B # y) @ v : S")
  apply(simp)
 apply(blast intro:S3)
apply clarsimp
apply(blast intro:S3)
done

lemma "balanced(w,n) ⟹ replicate n A @ w : S"
  apply (induct w n rule:balanced.induct)
  apply (simp_all add:replicate_app_Cons_same)
  apply (simp add:replicate_app_Cons_same[symmetric])
  apply (simp add: AB)
done
```