Isabelle/HOL Exercises
Trees, Inductive Data Types

Representation of Propositional Formulae by Polynomials

Let the following data type for propositional formulae be given:

datatype *form = T | Var nat | And form form | Xor form form*

Here *T* denotes a formula that is always true, *Var n* denotes a propositional variable, its name given by a natural number, *And f1 f2* denotes the AND combination, and *Xor f1 f2* the XOR (exclusive or) combination of two formulae. A constructor *F* for a formula that is always false is not necessary, since *F* can be expressed by *Xor T T*.

Exercise 1: Define a function

evalf

that evaluates a formula under a given variable assignment.

```
definition
  xor :: "bool ⇒ bool ⇒ bool" where
  "xor x y ≡  (x ∧ ¬ y) ∨ (¬ x ∧ y)"

primrec evalf :: "(nat ⇒ bool) ⇒ form ⇒ bool" where
  "evalf e T = True"
| "evalf e (Var i) = e i"
| "evalf e (And f1 f2) = (evalf e f1 ∧ evalf e f2)"
| "evalf e (Xor f1 f2) = xor (evalf e f1) (evalf e f2)"
```

Propositional formulae can be represented by so-called *polynomials*. A polynomial is a list of lists of propositional variables, i.e. an element of type *nat list list*. The inner lists (the so-called *monomials*) are interpreted as conjunctive combination of variables, whereas the outer list is interpreted as exclusive-or combination of the inner lists.

Exercise 2: Define two functions

```
evalm :: (nat ⇒ bool) ⇒ nat list ⇒ bool
evalp :: (nat ⇒ bool) ⇒ nat list list ⇒ bool
```

for evaluation of monomials and polynomials under a given variable assignment. In particular think about how empty lists have to be evaluated.

```
primrec evalm :: "(nat ⇒ bool) ⇒ nat list ⇒ bool" where
  "evalm e [] = True"
| "evalm e (x # xs) = (e x ∧ evalm e xs)"

primrec evalp :: "(nat ⇒ bool) ⇒ nat list list ⇒ bool" where
  "evalp e [] = False"
| "evalp e (m # p) = xor (evalm e m) (evalp e p)"
```

Exercise 3: Define a function

`poly :: form ⇒ nat list list`

that turns a formula into a polynomial. You will need an auxiliary function

`mulpp :: nat list list ⇒ nat list list ⇒ nat list list`

to "multiply" two polynomials, i.e. to compute

$$((v_1^1 \odot \cdots \odot v_{m_1}^1) \oplus \cdots \oplus (v_1^k \odot \cdots \odot v_{m_k}^k)) \odot ((w_1^1 \odot \cdots \odot w_{n_1}^1) \oplus \cdots \oplus (w_1^l \odot \cdots \odot w_{n_l}^l))$$

where $\oplus$ denotes "exclusive or", and $\odot$ denotes "and". This is done using the usual calculation rules for addition and multiplication.

```
primrec mulpp :: "nat list list ⇒ nat list list ⇒ nat list list" where
  "mulpp [] q = []"
| "mulpp (m # p) q = map (op @ m) q @ (mulpp p q)"

primrec poly :: "form ⇒ nat list list" where
  "poly T = [[]]"
| "poly (Var i) = [[i]]"
| "poly (Xor f1 f2) = poly f1 @ poly f2"
| "poly (And f1 f2) = mulpp (poly f1) (poly f2)"
```

Exercise 4: Now show correctness of your function `poly`:

```
theorem poly_correct: "evalf e f = evalp e (poly f)"
```

It is useful to prove a similar correctness theorem for `mulpp` first.

```
lemma evalm_app: "evalm e (xs @ ys) = (evalm e xs ∧ evalm e ys)"
  apply (induct xs)
  apply auto
done

lemma evalp_app: "evalp e (xs @ ys) = (xor (evalp e xs) (evalp e ys))"
  apply (induct xs)
  apply (auto simp add: xor_def)
done
```

```
theorem mulmp_correct: "evalp e (map (op @ m) p) = (evalm e m ∧ evalp e p)"
  apply (induct p)
  apply (auto simp add: xor_def evalm_app)
done

theorem mulpp_correct: "evalp e (mulpp p q) = (evalp e p ∧ evalp e q)"
  apply (induct p)
  apply (auto simp add: xor_def mulmp_correct evalp_app)
done

theorem poly_correct: "evalf e f = evalp e (poly f)"
  apply (induct f)
  apply (auto simp add: xor_def mulpp_correct evalp_app)
done
```