Isabelle/HOL Exercises
Trees, Inductive Data Types

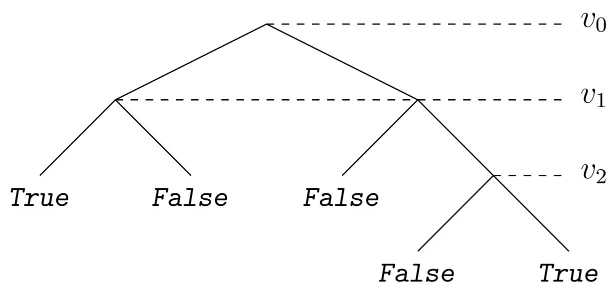
Binary Decision Diagrams

Boolean functions (in finitely many variables) can be represented by so-called *binary decision diagrams* (BDDs), which are given by the following data type:

```
datatype bdd = Leaf bool | Branch bdd bdd
```

A constructor `Branch b1 b2` that is i steps away from the root of the tree corresponds to a case distinction based on the value of the variable v_i. If the value of v_i is `False`, the left subtree `b1` is evaluated, otherwise the right subtree `b2` is evaluated. The following figure shows a Boolean function and the corresponding BDD.

v_0	v_1	v_2	$f(v_0, v_1, v_2)$
False	False	*	True
False	True	*	False
True	False	*	False
True	True	False	False
True	True	True	True



Exercise 1: Define a function

`eval :: (nat ⇒ bool) ⇒ nat ⇒ bdd ⇒ bool`

that evaluates a BDD under a given variable assignment, beginning at a variable with a given index.

```
primrec eval :: "(nat ⇒ bool) ⇒ nat ⇒ bdd ⇒ bool" where
  "eval e i (Leaf x) = x"
| "eval e i (Branch b1 b2) =
     (if e i then eval e (Suc i) b2 else eval e (Suc i) b1)"
```

Exercise 2: Define two functions

```
bdd_unop :: (bool ⇒ bool) ⇒ bdd ⇒ bdd
bdd_binop :: (bool ⇒ bool ⇒ bool) ⇒ bdd ⇒ bdd ⇒ bdd
```

for the application of unary and binary operators to BDDs, and prove their correctness.

```
primrec bdd_unop :: "(bool ⇒ bool) ⇒ bdd ⇒ bdd" where
```

```
  "bdd_unop f (Leaf x) = Leaf (f x)"
| "bdd_unop f (Branch b1 b2) = Branch (bdd_unop f b1) (bdd_unop f b2)"

primrec bdd_binop :: "(bool ⇒ bool ⇒ bool) ⇒ bdd ⇒ bdd ⇒ bdd" where
  "bdd_binop f (Leaf x) b = bdd_unop (f x) b"
| "bdd_binop f (Branch b1 b2) b = (case b of
       Leaf x ⇒ Branch (bdd_binop f b1 (Leaf x)) (bdd_binop f b2 (Leaf x))
     | Branch b1' b2' ⇒ Branch (bdd_binop f b1 b1') (bdd_binop f b2 b2'))"

theorem bdd_unop_correct:
  "∀i. eval e i (bdd_unop f b) = f (eval e i b)"
  apply (induct b)
  apply auto
done

theorem bdd_binop_correct:
  "∀i b2. eval e i (bdd_binop f b1 b2) = f (eval e i b1) (eval e i b2)"
  apply (induct b1)
  apply (auto simp add: bdd_unop_correct)
  apply (case_tac b2)
  apply auto
  apply (case_tac b2)
  apply auto
  apply (case_tac b2)
  apply auto
  apply (case_tac b2)
  apply auto
done
```

Now use *bdd_unop* and *bdd_binop* to define

```
consts
  bdd_and :: "bdd ⇒ bdd ⇒ bdd"
  bdd_or :: "bdd ⇒ bdd ⇒ bdd"
  bdd_not :: "bdd ⇒ bdd"
  bdd_xor :: "bdd ⇒ bdd ⇒ bdd"
```

and show correctness.

```
defs
  bdd_and_def: "bdd_and ≡ bdd_binop op ∧"
  bdd_or_def: "bdd_or ≡ bdd_binop op ∨"
  bdd_not_def: "bdd_not ≡ bdd_unop Not"
  bdd_xor_def: "bdd_xor b1 b2 == (bdd_or
     (bdd_and (bdd_not b1) b2) (bdd_and b1 (bdd_not b2)))"
```

```
theorem bdd_and_correct:
  "eval e i (bdd_and b1 b2) = (eval e i b1 ∧ eval e i b2)"
  apply (auto simp add: bdd_and_def bdd_binop_correct)
done

theorem bdd_or_correct:
  "eval e i (bdd_or b1 b2) = (eval e i b1 ∨ eval e i b2)"
  apply (auto simp add: bdd_or_def bdd_binop_correct)
done

theorem bdd_not_correct: "eval e i (bdd_not b) = (¬ eval e i b)"
  apply (auto simp add: bdd_not_def bdd_unop_correct)
done
```

Finally, define a function

`bdd_var :: nat ⇒ bdd`

to create a BDD that evaluates to `True` if and only if the variable with the given index evaluates to `True`. Again prove a suitable correctness theorem.

Hint: If a lemma cannot be proven by induction because in the inductive step a different value is used for a (non-induction) variable than in the induction hypothesis, it may be necessary to strengthen the lemma by universal quantification over that variable (cf. Section 3.2 in the Tutorial on Isabelle/HOL).

Example: instead of	Strengthening:
lemma `"P (b::bdd) x"`	**lemma** `"∀x. P (b::bdd) x"`
apply `(induct b)`	**apply** `(induct b)`

```
primrec bdd_var :: "nat ⇒ bdd" where
  "bdd_var 0 = Branch (Leaf False) (Leaf True)"
| "bdd_var (Suc i) = Branch (bdd_var i) (bdd_var i)"

theorem bdd_var_correct: "∀j. eval e j (bdd_var i) = e (i+j)"
  apply (induct i)
  apply auto
done
```

Exercise 3: Recall the following data type of propositional formulae (cf. the exercise on “Representation of Propositional Formulae by Polynomials”)

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

together with the evaluation function `evalf`:

```
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)"
```

Define a function

`mk_bdd :: form ⇒ bdd`

that transforms a propositional formula of type `form` into a BDD. Prove the correctness theorem

```
theorem mk_bdd_correct: "eval e 0 (mk_bdd f) = evalf e f"
primrec mk_bdd :: "form ⇒ bdd" where
  "mk_bdd T = Leaf True"
| "mk_bdd (Var i) = bdd_var i"
| "mk_bdd (And f1 f2) = bdd_and (mk_bdd f1) (mk_bdd f2)"
| "mk_bdd (Xor f1 f2) = bdd_xor (mk_bdd f1) (mk_bdd f2)"

theorem mk_bdd_correct: "eval e 0 (mk_bdd f) = evalf e f"
  apply (induct f)
  apply (auto simp add: bdd_var_correct
    bdd_and_correct bdd_or_correct bdd_not_correct bdd_xor_def xor_def)
done
```