Semantics of Programming Languages  
Exercise Sheet 2

**Homework 2.1**  
Tree traversal

*Submission until Tuesday, November 8, 10:00am.*

Recall the tree definition from the lecture and the function `mirror` to mirror trees:

```plaintext
datatype 'a tree = Tip | Node "'a tree" 'a "'a tree"

fun mirror :: "'a tree ⇒ 'a tree" where
"mirror Tip = Tip" |
"mirror (Node l x r) = Node (mirror r) x (mirror l)"
```

Define a function `in_order`, which traverses a tree in in-order. Prove that your definition of `in_order` fulfills the specification

```plaintext
theorem "rev (in_order t) = in_order (mirror t)"
```

where `rev` is the predefined function for reversing lists.

**Homework 2.2**  
Tail-Recursive Form of Addition

*Submission until Tuesday, November 8, 10:00am.*

The list-reversing function `itrev` is an example of a tail-recursive function: Note that the right-hand side of the second equation for `itrev` is simply an application of `itrev` to different arguments.

```plaintext
fun itrev :: "'a list ⇒ 'a list ⇒ 'a list" where
"itrev [] ys = ys" |
"itrev (x#xs) ys = itrev xs (x#ys)"
```

In this homework problem you will define a tail-recursive version of addition for natural numbers, and prove that it is associative and commutative.

First, define a function `add :: nat ⇒ nat ⇒ nat` in Isabelle that calculates the sum of its arguments. Like `itrev`, your definition should be tail-recursive: That is, in the recursive case the right-hand side should only be an application of `add` to different arguments.
fun add :: “nat ⇒ nat ⇒ nat”

Next, you must prove that add is associative. Hint: The proof will require at least one additional lemma. Also remember that some proofs by induction may require generalization with arbitrary.

theorem “add (add x y) z = add x (add y z)”

Finally, you must prove that add is commutative. This may require more lemmas in addition to those used for the associativity proof.

theorem “add x y = add y x”