Name: Khoa Tran – 5411431 – tran0707@umn.edu
Homework 3

Question 1: Power function, over natural number
Base Case:
P(0):
prove: power 0 x = x^0

  power 0 x
= 1.0
  by def. of power function
= x^0
  by arithmetic

Inductive Case:
P(n+1):
given: power n x = x^n
prove: power (n+1) x = $x^{(n+1)}$

  power (n+1) x
= x *. power (n) x
  by def. of power function
= x *. x^n
  by induction
= $x^{(n+1)}$
  by arithmetic

Question 2: Power over structured numbers
The principle of induction for the type nat is if P(Zero) and P(n) => P(Succ n)

1. we prove the base case P(Zero)
2. then prove the inductive case P(Succ n), assuming that P(n) holds.

Base Case:
P(Zero):
prove: power Zero x = $x^{toInt(Zero)}$

  power Zero x
= 1.0
  by def. power function
= x^0
  by arithmetic
= $x^{toInt(Zero)}$
  by def. toInt function

Inductive Case:

P(Succ n):
given: power n x = $x^{toInt(n)}$
prove: power (Succ n) x = $x^{toInt(Succ\ n)}$

  power (Succ n) x
= x *. power n x
  by def. of power function
= x *. $x^{toInt(n)}$
  by induction
= $x^{toInt(n) + 1}$
  by arithmetic
= $x^{toInt(succ\ n)}$
   by def. toInt function

<u>Question 3:</u> List reverse and append
Let prove that append lst [] = lst
Base Case:
P([]):
Prove: append [] [] = []
  append [] []
= []
  by def. of append function

Inductive Case:
P(x::xs):
given: append xs [] = xs
prove: append x::xs [] = x::xs

  append x::xs []
= x :: (append xs [])
  by def. of append function
= x :: xs
  by induction
It proved that append lst [] = lst

Let prove that reverse (append l1 l2) = append (reverse l2)(reverse l1)

Induction over l1
Base Case:
P([]):
prove: reverse (append [] l2) = append (reverse l2) (reverse [])
  part 1:
  reverse (append [] l2)
= reverse (l2)
  by def. of append function

```
  part 2:
  append (reverse l2) (reverse [])
= append (reverse l2) []
  by def. of reverse function
= reverse (l2)
  by the above proved for append lst [] = lst
```

From part 1 and 2, it proved that reverse (append [] l2) = append (reverse l2) (reverse [])

```
Inductive Case:
P(x::xs,l2): reverse (append x::xs l2) = append (reverse l2)
(reverse x::xs)
given: reverse (append xs l2) = append (reverse l2) (reverse
xs)
prove: reverse (append x::xs l2) = append (reverse l2) (reverse
x::xs)

  reverse (append x::xs l2)
= reverse (x :: (append xs l2))
  by def. of append function
= reverse (append xs l2) [x]
  by def. of reverse function
= append (reverse l2) (reverse xs) [x]
  by induction
= append (reverse l2) (reverse x::xs)
  by def. of reverse function
```

Question 4: List processing

```
Induction over l1
Base Case:
P([]):
someupper([]@l2) = someupper [] || someupper l2

  someupper([]@l2)
= someupper(l2)
  by understanding of @
= someupper l2 = someupper l2 || someupper []
  by equality of elements

Inductive Case:
P(x::xs,l2): someupper(x::xs @ l2) = someupper (x::xs) ||
someupper l2
given: someupper(xs@l2) = someupper xs || someupper l2
prove: someupper(x::xs@l2) = someupper (x::xs)|| someupper l2
```

```
  someupper(x::xs @ l2)
= someupper(x :: (xs @ l2))
  by understanding of @
= isupper x || someupper (xs @ l2)
  by def. of someupper function
= isupper x || someupper xs || someupper l2
  by induction
= someupper (x::xs) || someupper l2
  by def. Of someupper function
```

<u>Question 5:</u> List Processing and folds

```
Base Case:
P([]):
someupper [] = foldupper []

Part 1:
  someupper []
= false
  by def. of someupper function

Part 2:
  foldupper []
= foldr upperor [] false
  by def. of foldupper function
= false
  by def. of foldr function

From part 1 and 2, it proved that someupper [] = foldupper []

Inductive Case:
P(x::xs):
given: someupper xs = foldupper xs
prove: someupper (x::xs) = foldupper (x::xs)

Case 1: Char.code x >= Char.code 'A' && Char.code x <= Char.code
'Z'

  Part 1:
  someupper (x::xs)
= isupper x || someupper xs
  by def. of someupper function
= true || someupper xs
  by condition in case 1
= true || foldupper xs
  by induction
= true
  by understanding of ||
```

```
  Part 2:
  foldupper (x::xs)
= foldr upperor (x::xs) false
  by def. of foldupper function
= upperor x (foldr upperor xs false)
  by def. of foldr function
= isupper x || foldr upperor xs false
  by def. of upperor function
= true || foldr upperor xs false
  by the condition in the case 1
= true
  by understanding of ||
```

From part 1 and 2, the results when compare them is true, which mean they are equal. The Case 1 is proved.

Case 2: Char.code x < Char.code 'A' && Char.code x > Char.code 'Z'

```
  Part 3:
  someupper (x::xs)
= isupper x || someupper xs
  by def. of someupper function
= false || someupper xs
  by condition in case 2
= foldupper xs
  by induction

  Part 4:
  foldupper (x::xs)
= foldr upperor (x::xs) false
  by def. of foldupper function
= upperor x (foldr upperor xs false)
  by def. of foldr function
= isupper x|| foldr upperor xs false
  by def. of upperor function
= false || foldr upperor xs false
  by condition in case 2
= foldr upperor xs false
  by understand of ||
= foldupper xs
  by def. of foldupper function
```

From part 3 and 4, it proved that someupper (x::xs) = foldupper(x::xs)

```
Conclusion, someupper chs = foldupper chs
```

<u>Question 6:</u> Tree processing

```
Base Case:
P(Leaf t):
Prove: mintree (Leaf t) = fold_mintree (Leaf t)

  Part 1:
  mintree (Leaf t)
= t
  by def. of mintree function

  Part 2:
  fold_mintree (Leaf t)
= tfold (fun x-> x) min (Leaf t)
  by def. of fold_mintree function
= (fun x -> x) t
  by def. of tfold function
= t
  by applying t in (fun x -> x) which will return t

From part 1 and 2, it proved that mintree (Leaf t) = fold_mintee
(Leaf t)

Inductive Case:
P(Branch (t1,t2):
given: mintree t = fold_mintree t
prove: mintree (Branch (t1,t2)) = fold_mintree (Branch (t1,t2))

  mintree (Branch (t1,t2))
= min (mintree t1) (mintree t2)
  by def. of mintree function
= min ( fold_mintree t1) ( fold_mintree t2)
  by induction
= min ( tfold (fun x -> x) min t1) ( tfold (fun x-> x min t2)
  By def. of fold_mintree function
= tfold (fun x -> x) min Branch (t1,t2)
  By def. of tfold function
= fold_mintree (Brand (t1,t2)
  By def. of fold_mintree function
```