Lists and Structural Induction
So far

• Simple, strong, structural induction
• Examples of their use
• Simple function using lists

Today

• More examples with lists
• Tail-recursion
• More structural induction
Lists

A list of integers is either

- `[]` (also written as `nil`), or
- `x :: xs` where `x: int` and `xs: int list`

pronounced “cons"
(* length : int list -> int
   REQUIRES: true
   ENSURES: length(L) ==> length of list L
   *)

fun length([] : int list) : int = 0
   | length(x :: xs) = 1 + length(xs)
**Evaluation**

```plaintext
fun length([] : int list) : int = 0
| length(x :: xs) = 1 + length(xs)
```

length [3,4,5]  

```plaintext
==> 1 + length [4,5]
==> 1 + ( 1 + length [5])
==> 1 + ( 1 + ( 1 + length []))
==> 1 + ( 1 + ( 1 + 0))
==> 1 + ( 1 + 1 )
==> 1 + 2
==> 3
```
Towards Tail Recursion

A function is tail recursive if the last operation it performs is a recursive call to itself.
Using an accumulator argument

(* tlength : int list * int -> int
  REQUIRES: true
  ENSURES: tlength(L, acc) == length(L) + acc
  *)
Using an accumulator argument

(* tlength : int list * int -> int
  REQUIRES: true
  ENSURES: tlength(L, acc) == length(L) + acc
  *)

fun tlength([] : int list, acc : int) : int = acc
  | tlength(x::xs , acc) = tlength(xs, 1 + acc)

fun Length (L : int list) = tlength(L, 0)
Tail Recursion

A function is tail recursive if the last operation it performs is a recursive call to itself.

```haskell
fun tlength([] : int list, acc : int) : int = acc
| tlength(x::xs , acc) = tlength(xs, 1 + acc)
```

```
tlength ([3,4,5],0) ==> tlength ([4,5],1 + 0)
  ==> tlength ([4,5],1)
  ==> tlength ([5],1 + 1)
  ==> tlength ([5],2)
  ==> tlength ([],2 + 1)
  ==> tlength ([],3)
  ==> 3
```
Using an accumulator argument

(* tlength : int list * int -> int
   REQUIRES: true
   ENSURES: tlength(L, acc) == length(L) + acc
*)

fun tlength([], acc : int) : int = acc
| tlength(_::L, acc) : int = tlength(L, 1 + acc)

fun Length(L : int list) = tlength(L, 0)

**Theorem**: For all values \(L:\text{int list}\) and \(\text{acc : int}\),
\[\text{tlength}(L, \text{acc}) \equiv \text{length}(L) + \text{acc}.\]
fun tlength([], acc) : int = acc
| tlength(_::L, acc) : int = tlength(L, 1 + acc)

fun length([], : int list) : int = 0
| length(x :: xs) = 1 + length(xs)

**Theorem:** For all values L: int list and acc : int, tlength(L, acc) ≡ length(L)+acc.
Appending lists

(* append : int list * int list -> int list
  REQUIRES:  true
  ENSURES:  append(l,r) returns the list consisting of l followed by r
  NOTE:  this is also predefined in SML as the right-associative
         infix operator @.  *)

fun append ([]: int list, r: int list) : int list = r
  | append (x::l, r) = x::append(l,r)

val [] : int list = append([],[])
val [1,2] = append([], [1,2])
val [1,2,5,6] = append([1,2], [5,6])
Evaluating `append`

\[
\text{append } ([1,2],[5,\sim6,7]) \implies 1 :: \text{append } ([2],[5,\sim6,7])
\]
\[
\implies 1 :: (2 :: \text{append } ([],[5,\sim6,7]))
\]
\[
\implies 1 :: (2 :: [5,\sim6,7])
\]
\[
\implies 1 :: [2,5,\sim6,7]
\]
\[
\implies [1,2,5,\sim6,7]
\]
(* rev : int list -> int list
  REQUIRES: true
  ENSURES: rev(L) returns the list L in reverse order
*)

fun rev ([] : int list) : int list = []
  | rev (x::xs) = (rev xs) @ [x]

val [] : int list = rev []
val [4,3,2,1] : int list = rev [1,2,3,4]
(* trev : int list * int list -> int list
  REQUIRES: true
  ENSURES: trev(L, acc) == ??? *)
(* trev : int list * int list -> int list
  REQUIRES: true
  ENSURES:  trev(L, acc) == (rev L) @ acc
*)

fun trev([], acc: int list) : int list = acc
| trev(x::xs, acc) = trev(xs, x::acc)

val [9,10,11] : int list = trev([], [9,10,11])
val [4,3,2,1,0,0,7] : int list = trev([1,2,3,4], [0,0,7])
fun trev([], acc : int list) : int list = acc

| trev(x::xs, acc) = trev(xs, x::acc)

val [9,10,11] : int list = trev([], [9,10,11])
val [4,3,2,1,0,0,7] : int list = trev([1,2,3,4], [0,0,7])

**Theorem:** For all integer lists $L$ and $acc$, 
$$\text{trev}(L, acc) \equiv \text{rev}(L) \@ acc.$$
fun trev([], : int list, acc: : int list) : int list = acc
| trev(x::xs, acc) = trev(xs, x::acc)

fun rev([] : int list) : int list = []
| rev(x::xs) = (rev xs) @ [x]

**Theorem:** For all integer lists \( L \) and \( \text{acc} \),
\[
\text{trev}(L, \text{acc}) \equiv \text{rev}(L) + \text{acc}.
\]