# 15-150

# Principles of Functional Programming

## Lecture 4

January 23, 2025

Michael Erdmann

# Tail Recursion

**More about**

**Lists**

**Structural Induction**

```
(* length : int list -> int
   REQUIRES: true
   ENSURES: length(L) returns the
            number of elements in L.
*)

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

```
(* length : int list → int
   REQUIRES: true
   ENSURES: length(L) returns the
            number of elements in L.
*)

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

```
   length [4,7,9,2]
⟹ 1 + length [7, 9, 2]
```

Why?

```
(* length : int list → int
   REQUIRES: true
   ENSURES: length(L) returns the
            number of elements in L.
*)

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

length [4,7,9,2]
⟹ 1 + length [7,9,2]

# Why?

Because [4,7,9,2] means 4::[7,9,2]
and so length [4,7,9,2]
⟹ [..., 4/x, [7,9,2]/xs] 1 + length(xs)
⟹ 1 + length [7,9,2]

( ... means the environment when length was defined ) length [4,7,9,2]

$\Longrightarrow$ 1 + length [7,9,2]

$\Longrightarrow$ 1 + (1 + length [9,2])

$\Longrightarrow$ 1 + (1 + (1 + length [2]))

$\Longrightarrow$ 1 + (1 + (1 + (1 + length [])))

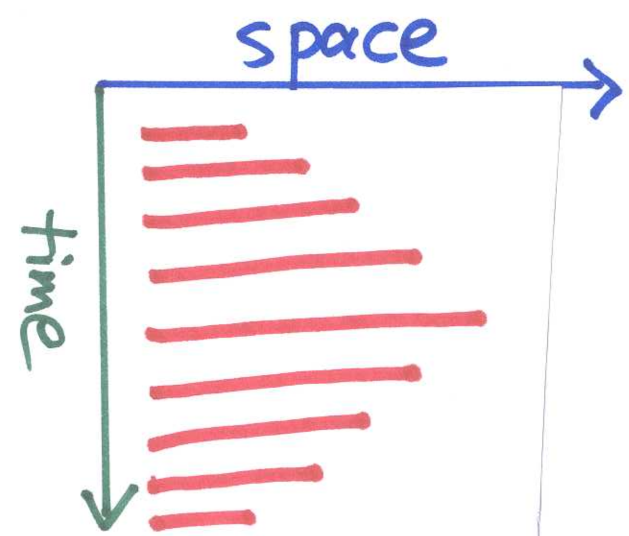
$\Longrightarrow$ 1 + (1 + (1 + (1 + 0)))

$\Longrightarrow$ 1 + (1 + (1 + 1))

$\Longrightarrow$ 1 + (1 + 2)

$\Longrightarrow$ 1 + 3

$\Longrightarrow$ 4



accumulator

(* tlength: int list * int → int

REQUIRES: true

ENSURES: $\text{tlength}(L, \text{acc}) \cong (\text{length } L) + \text{acc}$

*)

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

fun tlength ([] : int list, acc : int) : int = ?
```

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

fun tlength ([]: int list, acc : int) : int = acc
```

```
(* 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 : 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)
```

```
(* 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)
```

tail call

---

tlength is tail recursive

---

# Definition

A function is tail recursive if it is recursive and if it performs no computations after calling itself recursively.

Such recursive calls are said to be tail calls

(as in: "tail" meaning "at the end").

If the body of a function contains multiple locations at which a recursive call occurs, then every recursive call must be a tail call for the function to be tail recursive.

Now implement a length function based on tlength:

```
(* leng : int list → int
     REQUIRES & ENSURES as for length
*)

fun leng (L: int list) : int =

          ???
```

Now implement a length function based on tlength:

```
(* leng : int list → int
     REQUIRES & ENSURES as for length
*)

fun leng (L: int list) : int =

        tlength (L, 0)
```

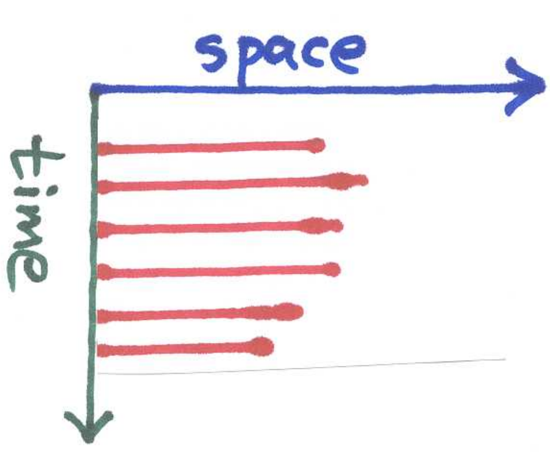
tlength (L, acc) $\cong$ (length L) + acc

Now implement a length function based on tlength:

```
(* leng : int list → int
     REQUIRES & ENSURES as for length
*)

fun leng (L:int list) : int =

        tlength (L,0)


     leng [4,7,9,2]
⟹ tlength([4,7,9,2], 0)
```

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

```
   leng [4,7,9,2]
⟹ tlength ([4,7,9,2], 0)
⟹ tlength ([7,9,2], 1)
⟹ tlength ([9,2], 2)
⟹ tlength ([2], 3)
⟹ tlength ([], 4)
⟹ 4
```



## Theorem

For all values $L$ : int list and acc : int,

$$\text{tlength}(L, \text{acc}) \cong (\text{length } L) + \text{acc}.$$

During lecture:

We proved the theorem using structural induction.
See online code file for details.

```
(* append : int list * int list -> int list
   REQUIRES: true
   ENSURES:
     append (X, Y) returns a list
     consisting of the elements of
     X followed by the elements of
     Y, preserving order.
   Example:  append ([3,4], [1,3,10])
               ==> [3,4,1,3,10]
*)

fun append ([]: int list, Y : int list) : int list = Y
  | append (x::xs, Y) = x :: append (xs, Y)
```

```
(* append : int list * int list -> int list
     REQUIRES: true
     ENSURES:
       append (X, Y) returns a list
       consisting of the elements of
       X followed by the elements of
       Y, preserving order.
   Example:  append([3,4], [1,3,10])
               ==> [3,4,1,3,10]
*)

fun append ([]: int list, Y :int list): int list = Y
  | append (x::xs, Y) = x :: append(xs, Y)
```

---

append (X, Y) has time complexity $O(|X|)$.

---

```
fun append ([]: int list, Y : int list) : int list = Y
  | append (x::xs, Y) = x :: append (xs, Y)
```

---

```
append ([1,2], [5,~6,7])
⇒ 1::append ([2], [5,~6,7])
⇒ 1:: (2 :: append ([ ], [5,~6,7]))
⇒ 1:: (2 :: [5,~6,7])
⇒ 1:: [2,5,~6,7]
⇒ [1,2,5,~6,7]
```

append is predefined in SML as the right-associative infix operator @.

So [1,2] @ [3,4] @ [6,9,10]

means

[1,2] @ ( [3,4] @ [6,9,10] )

$\Longrightarrow$ [1,2] @ [3,4,6,9,10]

$\Longrightarrow$ [1,2,3,4,6,9,10]

(* rev : int list → int list

REQUIRES: true

ENSURES: rev L returns a list consisting of the elements of L in reverse order.

**Example:** rev [7,9,2] ⟹ [2,9,7].

*)

```
(* rev : int list → int list
   REQUIRES: true
   ENSURES:  rev L returns a list
       consisting of the elements of L
       in reverse order.
   Example:  rev [7,9,2] ⇒ [2,9,7].
*)

fun rev ([]:int list):int list = ?
```

```
(* rev : int list -> int list
   REQUIRES: true
   ENSURES:  rev L returns a list
        consisting of the elements of L
        in reverse order.
   Example:  rev [7,9,2] ==> [2,9,7].
*)

fun rev ([]:int list):int list = []
```

```
(* rev : int list → int list
   REQUIRES: true
   ENSURES:  rev L returns a list
       consisting of the elements of L
       in reverse order.
   Example:  rev [7,9,2] ⇒ [2,9,7].
*)

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

```
(* rev : int list -> int list
   REQUIRES: true
   ENSURES:  rev L returns a list
        consisting of the elements of L
        in reverse order.
   Example:  rev [7,9,2] ==> [2,9,7].
*)

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

```
(* rev : int list → int list
   REQUIRES: true
   ENSURES:  rev L returns a list
        consisting of the elements of L
        in reverse order.
   Example:  rev [7,9,2] ⟹ [2,9,7].
*)
```

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

What is the time complexity?

```
(* rev : int list → int list
   REQUIRES: true
   ENSURES: rev L returns a list
       consisting of the elements of L
       in reverse order.
   Example: rev [7,9,2] ⟹ [2,9,7].
*)

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

What is the time complexity?

$O(n^2)$,

with $n$ the number of elements in the list.

```
rev [1,2,3,4]
⟹ (rev [2,3,4]) @ [1]
⟹ ((rev [3,4]) @ [2]) @ [1]
⟹ (((rev [4]) @ [3]) @ [2]) @ [1]
⟹ ((((rev [ ]) @ [4]) @ [3]) @ [2]) @ [1]
⟹ ((( [ ] @ [4]) @ [3]) @ [2]) @ [1]
⟹ (( [4] @ [3]) @ [2]) @ [1]
⟹ (( 4 :: [ ] @ [3]) @ [2]) @ [1]
⟹ (( 4 :: [3]) @ [2]) @ [1]
⟹ ( [4,3] @ [2]) @ [1]
⟹ ( 4 :: [3] @ [2]) @ [1]
⟹ ( 4 :: (3 :: [ ] @ [2])) @ [1]
⟹ (4 :: ( 3 :: [2] )) @ [1]
⟹ ( 4 :: [3,2]) @ [1]
⟹ [4,3,2] @ [1]
⟹ 4 :: [3,2] @ [1]
⟹ 4 :: (3 :: [2] @ [1])
⟹ 4 :: (3 :: (2 :: [ ] @ [1]))
⟹ 4 :: (3 :: ( 2 :: [1] ))
⟹ 4 :: (3 :: [2,1])
⟹ 4 :: [3,2,1]
⟹ [4,3,2,1]
```

Finally, [4,3,2,1]

```
(* trev : int list * int list -> int list
   REQUIRES: true
   ENSURES:  trev(L,acc)
                 ≅
             (rev L)@ acc
*)
```

(* trev : int list * int list → int list
   REQUIRES: true
   ENSURES: trev(L,acc) $\cong$ (rev L)@ acc
*)

```
fun trev ([]:int list, acc:int list):int list =
      ?
```

```
(* trev : int list * int list -> int list
   REQUIRES: true
   ENSURES:  trev(L,acc)
                ≅
             (rev L)@ acc
*)

fun trev ([]:int list, acc:int list) : int list =
          acc
```

```
(* trev : int list * int list -> int list
   REQUIRES: true
   ENSURES:  trev(L,acc)
                  ≅
             (rev L)@ acc
*)

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

```
(* trev : int list * int list -> int list
   REQUIRES: true
   ENSURES:   trev(L,acc)
                  ≅
              (rev L)@ acc
*)

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

```
(* trev : int list * int list -> int list
   REQUIRES: true
   ENSURES:   trev(L,acc)
                  ≅
              (rev L)@ acc
*)

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

What is the time complexity?

```
(* trev : int list * int list -> int list
   REQUIRES: true
   ENSURES:  trev(L,acc)
                 ≅
             (rev L)@ acc
*)
```

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

What is the time complexity?

$O(n)$,

with n the number of elements in the first list.

trev([1,2,3,4], [ ])

$\Longrightarrow$ trev([2,3,4], [1])

$\Longrightarrow$ trev([3,4], [2,1])

$\Longrightarrow$ trev([4], [3,2,1])

$\Longrightarrow$ trev([ ], [4,3,2,1])

$\Longrightarrow$ [4,3,2,1]

Can now implement list reversal more efficiently:

(* reverse : int list → int list *)

```
fun reverse (L:int list):int list =
```

? ? ?

$$\text{trev}\,(L, acc) \cong (\text{rev } L) @ acc$$

Can now implement list reversal more efficiently:

```
(* reverse : int list → int list *)

fun reverse (L:int list):int list =

        trev (L, [])
```

$\text{trev}(L, acc) \cong (\text{rev } L)@acc$

## Theorem

For all values $L$ : int list and $acc$ : int list,

$$\text{trev}\,(L, acc) \cong (\text{rev } L)\,@\,acc.$$

During lecture:

We proved the theorem using structural induction.
See online notes for details.

That is all.

Have a good weekend.

See you Tuesday.