Foundations of Computer Science

The basics of lists

Dr. Robert Harle & Dr. Jeremy Yallop

2020–2021

Warm-Up

Question 1: What does this return?

```
In[1]: 3 + -0.2

  Out: Error: This expression has type float
       but an expression was expected of type int
       Line 1, characters 2-3:
       Hint: Did you mean to use '+.'?
```

Question 2: What is the **complexity** of matrix addition for a square matrix of size n?

Question 3: What do we call a function whose computation does not nest?

Warm-Up

Question 1: What does this return?

```
In[1]: 3 + -0.2

  Out: Error: This expression has type float
       but an expression was expected of type int
       Line 1, characters 2-3:
       Hint: Did you mean to use ‘+.’?
```

Question 2: What is the **complexity** of matrix addition for a square matrix of size n?

Question 3: What do we call a function whose computation does not nest?

Warm-Up

Question 1: What does this return?

```
In[1]: 3 + -0.2

 Out: Error: This expression has type float
      but an expression was expected of type int
      Line 1, characters 2-3:
      Hint: Did you mean to use ‘+.’?
```

Question 2: What is the **complexity** of matrix addition for a square matrix of size n?

Question 3: What do we call a function whose computation does not nest?

Warm-Up

Question 1: What does this return?

```
In[1]:  3 + -0.2

  Out:  Error: This expression has type float
        but an expression was expected of type int
        Line 1, characters 2-3:
        Hint: Did you mean to use ‘+.’?
```

Question 2: What is the **complexity** of matrix addition for a square matrix of size n?

$$O(n^2)$$

Question 3: What do we call a function whose computation does not nest?

Warm-Up

Question 1: What does this return?

```
In[1]: 3 + -0.2

  Out: Error: This expression has type float
       but an expression was expected of type int
       Line 1, characters 2-3:
       Hint: Did you mean to use ‘+.’?
```

Question 2: What is the **complexity** of matrix addition for a square matrix of size n?

$$O(n^2)$$

Question 3: What do we call a function whose computation does not nest?

Iterative or tail-recursive

Lists

A list is a **finite sequence** of elements

The elements may have **any type**

All elements must have **same** type

```
 In[2]: [3; 5; 9]
Out[2]: - : int list = [3; 5; 9]
 In[3]: [[3]; []; [5; 6]]
Out[3]: - : int list list = [[3]; []; [5; 6]]
 In[4]: [3; [5]; 9]
   Out: Line 1, characters 4-7:
        1 | [3; [5]; 9]
                ^^^
        Error: This expression has type 'a list
        but an expression was expected of type int
```

Lists

A list is a **finite sequence** of elements

The elements may have **any type**

All elements must have **same type**

```
 In[2]: [3; 5; 9]
Out[2]: - : int list = [3; 5; 9]
 In[3]: [[3]; []; [5; 6]]
Out[3]: - : int list list = [[3]; []; [5; 6]]
 In[4]: [3; [5]; 9]
   Out: Line 1, characters 4-7:
        1 | [3; [5]; 9]
                ^^^
        Error: This expression has type 'a list
        but an expression was expected of type int
```

Lists

A list is a **finite sequence** of elements

The elements may have **any type**

All elements must have **same** type

```
  In[2]: [3; 5; 9]
 Out[2]: - : int list = [3; 5; 9]
  In[3]: [[3]; []; [5; 6]]
 Out[3]: - : int list list = [[3]; []; [5; 6]]
  In[4]: [3; [5]; 9]
    Out: Line 1, characters 4-7:
         1 | [3; [5]; 9]
                 ^^^
         Error: This expression has type 'a list
         but an expression was expected of type int
```

Lists

A list is a **finite sequence** of elements

The elements may have **any type**

All elements must have **same** type

```
 In[2]: [3; 5; 9]
Out[2]: - : int list = [3; 5; 9]
 In[3]: [[3]; []; [5; 6]]
Out[3]: - : int list list = [[3]; []; [5; 6]]
 In[4]: [3; [5]; 9]
   Out: Line 1, characters 4-7:
        1 | [3; [5]; 9]
                ^^^
        Error: This expression has type 'a list
        but an expression was expected of type int
```

Lists

A list is a **finite sequence** of elements

The elements may have **any type**

All elements must have **same** type

```
  In[2]: [3; 5; 9]
 Out[2]: - : int list = [3; 5; 9]
  In[3]: [[3]; []; [5; 6]]
 Out[3]: - : int list list = [[3]; []; [5; 6]]
  In[4]: [3; [5]; 9]
    Out: Line 1, characters 4-7:
         1 | [3; [5]; 9]
                 ^^^
         Error: This expression has type 'a list
         but an expression was expected of type int
```

Lists

A list is a **finite sequence** of elements

The elements may have **any type**

All elements must have **same** type

```
 In[2]: [3; 5; 9]
Out[2]: - : int list = [3; 5; 9]
 In[3]: [[3]; []; [5; 6]]
Out[3]: - : int list list = [[3]; []; [5; 6]]
 In[4]: [3; [5]; 9]
   Out: Line 1, characters 4-7:
        1 | [3; [5]; 9]
                ^^^
        Error: This expression has type 'a list
        but an expression was expected of type int
```

Lists

A list is a **finite sequence** of elements

The elements may have **any type**

All elements must have **same** type

```
 In[2]: [3; 5; 9]
Out[2]: - : int list = [3; 5; 9]
 In[3]: [[3]; []; [5; 6]]
Out[3]: - : int list list = [[3]; []; [5; 6]]
 In[4]: [3; [5]; 9]
   Out: Line 1, characters 4-7:
        1 | [3; [5]; 9]
                ^^^
        Error: This expression has type 'a list
        but an expression was expected of type int
```

Lists

```
 In[5]: let it = [3; 5; 9]
Out[5]: val it : int list = [3; 5; 9]
```

append

```
 In[6]: it @ [2; 10]
Out[6]: - : int list = [3; 5; 9; 2; 10]
```

reverse

```
 In[7]: List.rev [(1, "one"); (2, "two")]
Out[7]: - : (int * string) list
          = [(2, "two"); (1, "one")]
```

Lists

```
 In[5]: let it = [3; 5; 9]
Out[5]: val it : int list = [3; 5; 9]
   append
 In[6]: it @ [2; 10]
Out[6]: - : int list = [3; 5; 9; 2; 10]
      reverse
 In[7]: List.rev [(1, "one"); (2, "two")]
Out[7]: - : (int * string) list
          = [(2, "two"); (1, "one")]
```

Lists

```
 In[5]:  let it = [3; 5; 9]
Out[5]:  val it : int list = [3; 5; 9]
```

append

```
 In[6]:  it @ [2; 10]
Out[6]:  - : int list = [3; 5; 9; 2; 10]
```

reverse

```
 In[7]:  List.rev [(1, "one"); (2, "two")]
Out[7]:  - : (int * string) list
            = [(2, "two"); (1, "one")]
```

Lists

```
 In[5]:  let it = [3; 5; 9]
Out[5]:  val it : int list = [3; 5; 9]
```

append

```
 In[6]:  it @ [2; 10]
Out[6]:  - : int list = [3; 5; 9; 2; 10]
```

reverse

```
 In[7]:  List.rev [(1, "one"); (2, "two")]
Out[7]:  - : (int * string) list
           = [(2, "two"); (1, "one")]
```

Lists

```
 In[5]: let it = [3; 5; 9]
Out[5]: val it : int list = [3; 5; 9]
```

append

```
 In[6]: it @ [2; 10]
Out[6]: - : int list = [3; 5; 9; 2; 10]
```

reverse

```
 In[7]: List.rev [(1, "one"); (2, "two")]
Out[7]: - : (int * string) list
          = [(2, "two"); (1, "one")]
```

Lists

```
 In[5]: let it = [3; 5; 9]
Out[5]: val it : int list = [3; 5; 9]

  append
 In[6]: it @ [2; 10]
Out[6]: - : int list = [3; 5; 9; 2; 10]

     reverse
 In[7]: List.rev [(1, "one"); (2, "two")]
Out[7]: - : (int * string) list
          = [(2, "two"); (1, "one")]
```

Lists

```
 In[5]:  let it = [3; 5; 9]
Out[5]:  val it : int list = [3; 5; 9]

  append

 In[6]:  it @ [2; 10]
Out[6]:  - : int list = [3; 5; 9; 2; 10]

      reverse

 In[7]:  List.rev [(1, "one"); (2, "two")]
Out[7]:  - : (int * string) list
           = [(2, "two"); (1, "one")]
```

The List Primitives

We build a list using **two primitives**:

$$[\,]$$
$$::$$

Example: the list $[3;\ 5;\ 9]$ is constructed as follows:

$$\begin{aligned} 9 \ :: \ [] &= [9] \\ 5 \ :: \ [9] &= [5;\ 9] \\ 3 \ :: \ [5;\ 9] &= [3;\ 5;\ 9] \end{aligned}$$

The List Primitives

Two kinds of list

[]	is the empty list
x :: l	is the list with head x and tail l

List notation

$$[x_1;\ x_2;\ \ldots;\ x_n] \quad \equiv \quad \underbrace{x_1}_{\text{head}} :: \underbrace{(x_2 :: \ \cdots (x_n :: []))}_{\text{tail}}$$

The List Primitives

Internally: linked structure

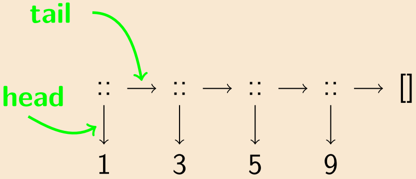
$$\begin{tikzcd}
:: \arrow[r] \arrow[d] & :: \arrow[r] \arrow[d] & :: \arrow[r] \arrow[d] & :: \arrow[r] \arrow[d] & {[]} \\
1 & 3 & 5 & 9 &
\end{tikzcd}$$

Note that :: is an $O(1)$ operation

Taking a list's head or tail takes **constant time**

The List Primitives

Internally: linked structure



Note that :: is an $O(1)$ operation

Taking a list's head or tail takes **constant time**

The List Primitives

```
 In[8]: let rec up_to m n =
          if m > n then []
          else m :: up_to (m + 1) n

Out[8]: val up_to : int -> int -> int list = <fun>

 In[9]: up_to 2 5

Out[9]: - : int list = [2; 3; 4; 5]
```

The List Primitives

```
 In[8]:  let rec up_to m n =
           if m > n then []
           else m :: up_to (m + 1) n

Out[8]:  val up_to : int -> int -> int list = <fun>

 In[9]:  up_to 2 5

Out[9]:  - : int list = [2; 3; 4; 5]
```

The List Primitives

```
 In[8]:  let rec up_to m n =
           if m > n then []
           else m :: up_to (m + 1) n

Out[8]:  val up_to : int -> int -> int list = <fun>

 In[9]:  up_to 2 5

Out[9]:  - : int list = [2; 3; 4; 5]
```

The List Primitives

```
 In[8]:  let rec up_to m n =
           if m > n then []
           else m :: up_to (m + 1) n

Out[8]:  val up_to : int -> int -> int list = <fun>

 In[9]:  up_to 2 5

Out[9]:  - : int list = [2; 3; 4; 5]
```

The List Primitives

```
 In[8]:  let rec up_to m n =
           if m > n then []
           else m :: up_to (m + 1) n

Out[8]:  val up_to : int -> int -> int list = <fun>

 In[9]:  up_to 2 5

Out[9]:  - : int list = [2; 3; 4; 5]
```

Getting at the Head and Tail

```
 In[10]: let hd (x::_) = x
         Warning 8: this pattern-matching is not
         exhaustive.
         Here is an example of a case that is not
         matched:
         []

Out[10]: val hd : 'a list -> 'a = <fun>
```

```
 In[11]: List.tl [7; 6; 5]

Out[11]: - : int list = [6; 5]
```

Pattern-matching

```
 In[12]: let null = function
 1st case ⟶| [] -> true
 2nd case ⟶| _::_ -> false

Out[12]: val null : 'a list -> bool = <fun>
```

Getting at the Head and Tail

```
 In[10]:  let hd (x::_) = x
          Warning 8: this pattern-matching is not
          exhaustive.
          Here is an example of a case that is not
          matched:
          []

Out[10]:  val hd : 'a list -> 'a = <fun>

 In[11]:  List.tl [7; 6; 5]

Out[11]:  - : int list = [6; 5]
```

Pattern-matching

```
 In[12]:  let null = function
 1st case ⟶| [] -> true
2nd case ⟶| _::_ -> false

Out[12]:  val null : 'a list -> bool = <fun>
```

Getting at the Head and Tail

```
In[10]:  let hd (x::_) = x
         Warning 8: this pattern-matching is not
         exhaustive.
         Here is an example of a case that is not
         matched:
         []

Out[10]: val hd : 'a list -> 'a = <fun>

In[11]:  List.tl [7; 6; 5]

Out[11]: - : int list = [6; 5]
```

Pattern-matching

```
In[12]:  let null = function
1st case ⟶| [] -> true
2nd case ⟶| _::_ -> false

Out[12]: val null : 'a list -> bool = <fun>
```

Getting at the Head and Tail

```
 In[10]: let hd (x::_) = x
         Warning 8: this pattern-matching is not
         exhaustive.
         Here is an example of a case that is not
         matched:
         []
Out[10]: val hd : 'a list -> 'a = <fun>

 In[11]: List.tl [7; 6; 5]
Out[11]: - : int list = [6; 5]
```

Pattern-matching

```
 In[12]: let null = function
1st case ⟶| [] -> true
2nd case ⟶| _::_ -> false
Out[12]: val null : 'a list -> bool = <fun>
```

Getting at the Head and Tail

```
 In[10]: let hd (x::_) = x
         Warning 8: this pattern-matching is not
         exhaustive.
         Here is an example of a case that is not
         matched:
         []
Out[10]: val hd : 'a list -> 'a = <fun>

 In[11]: List.tl [7; 6; 5]
Out[11]: - : int list = [6; 5]
```

Pattern-matching

```
 In[12]: let null = function
 1st case ⟶| [] -> true
 2nd case ⟶| _::_ -> false
Out[12]: val null : 'a list -> bool = <fun>
```

Getting at the Head and Tail

```
 In[10]: let hd (x::_) = x
         Warning 8: this pattern-matching is not
         exhaustive.
         Here is an example of a case that is not
         matched:
         []
Out[10]: val hd : 'a list -> 'a = <fun>

 In[11]: List.tl [7; 6; 5]
Out[11]: - : int list = [6; 5]
```

Pattern-matching

```
 In[12]: let null = function
 1st case ⟶| [] -> true
 2nd case ⟶| _::_ -> false
Out[12]: val null : 'a list -> bool = <fun>
```

Getting at the Head and Tail

```
 In[10]:  let hd (x::_) = x
          Warning 8: this pattern-matching is not
          exhaustive.
          Here is an example of a case that is not
          matched:
          []

Out[10]:  val hd : 'a list -> 'a = <fun>

 In[11]:  List.tl [7; 6; 5]

Out[11]:  - : int list = [6; 5]
```

Pattern-matching

```
 In[12]:  let null = function
 1st case ⟶| [] -> true
 2nd case ⟶| _::_ -> false

Out[12]:  val null : 'a list -> bool = <fun>
```

Getting at the Head and Tail

```
 In[10]:  let hd (x::_) = x
          Warning 8: this pattern-matching is not
          exhaustive.
          Here is an example of a case that is not
          matched:
          []
Out[10]:  val hd : 'a list -> 'a = <fun>

 In[11]:  List.tl [7; 6; 5]
Out[11]:  - : int list = [6; 5]
```

Pattern-matching

```
 In[12]:  let null = function
 1st case ⟶| [] -> true
 2nd case ⟶| _::_ -> false
Out[12]:  val null : 'a list -> bool = <fun>
```

Getting at the Head and Tail

```
 In[10]:  let hd (x::_) = x
          Warning 8: this pattern-matching is not
          exhaustive.
          Here is an example of a case that is not
          matched:
          []
Out[10]:  val hd : 'a list -> 'a = <fun>

 In[11]:  List.tl [7; 6; 5]
Out[11]:  - : int list = [6; 5]
```

Pattern-matching

```
 In[12]:  let null = function
1st case ⟶| [] -> true
2nd case ⟶| _::_ -> false
Out[12]:  val null : 'a list -> bool = <fun>
```

Getting at the Head and Tail

Note: all three functions are **polymorphic**:

`val null : 'a list -> bool`	is a list empty?
`val hd : 'a list -> 'a`	head of a non-empty list
`val tl : 'a list -> 'a list`	tail of a non-empty list

Computing the Length of a List

```
 In[13]: let rec nlength = function
         | [] -> 0
         | _ :: xs -> 1 + nlength xs

Out[13]: val nlength : 'a list -> int = <fun>
```

Computing the Length of a List

```
In[13]:  let rec nlength = function
         | [] -> 0
         | _ :: xs -> 1 + nlength xs

Out[13]: val nlength : 'a list -> int = <fun>
```

Computing the Length of a List

```
 In[13]: let rec nlength = function
         | [] -> 0
         | _ :: xs -> 1 + nlength xs

Out[13]: val nlength : 'a list -> int = <fun>
```

Computing the Length of a List

```
In[13]:  let rec nlength = function
         | [] -> 0
         | _ :: xs -> 1 + nlength xs

Out[13]: val nlength : 'a list -> int = <fun>
```

nlength [3; 5; 9] is evaluated as follows:

nlength [3; 5; 9]

Computing the Length of a List

```
 In[13]:  let rec nlength = function
          | [] -> 0
          | _ :: xs -> 1 + nlength xs

Out[13]:  val nlength : 'a list -> int = <fun>
```

nlength [3; 5; 9] is evaluated as follows:

$$\text{nlength } [3;\ 5;\ 9] \quad \Rightarrow \quad 1 + \text{nlength } [5;\ 9]$$

Computing the Length of a List

```
 In[13]: let rec nlength = function
         | [] -> 0
         | _ :: xs -> 1 + nlength xs

Out[13]: val nlength : 'a list -> int = <fun>
```

nlength [3; 5; 9] is evaluated as follows:

$$\begin{aligned}\text{nlength } [3;\ 5;\ 9] &\Rightarrow 1 + \text{nlength } [5;\ 9]\\ &\Rightarrow 1 + (1 + \text{nlength } [9])\end{aligned}$$

Computing the Length of a List

```
 In[13]: let rec nlength = function
         | [] -> 0
         | _ :: xs -> 1 + nlength xs

Out[13]: val nlength : 'a list -> int = <fun>
```

nlength [3; 5; 9] is evaluated as follows:

$$\begin{aligned}
\text{nlength}\ [3;\ 5;\ 9] &\Rightarrow 1 + \text{nlength}\ [5;\ 9] \\
&\Rightarrow 1 + (1 + \text{nlength}\ [9]) \\
&\Rightarrow 1 + (1 + (1 + \text{nlength}\ []))
\end{aligned}$$

Computing the Length of a List

```
 In[13]: let rec nlength = function
         | [] -> 0
         | _ :: xs -> 1 + nlength xs

Out[13]: val nlength : 'a list -> int = <fun>
```

nlength [3; 5; 9] is evaluated as follows:

$$\begin{aligned}
\text{nlength}\ [3;\ 5;\ 9] &\Rightarrow 1 + \text{nlength}\ [5;\ 9] \\
&\Rightarrow 1 + (1 + \text{nlength}\ [9]) \\
&\Rightarrow 1 + (1 + (1 + \text{nlength}\ [])) \\
&\Rightarrow 1 + (1 + (1 + 0)
\end{aligned}$$

Computing the Length of a List

```
 In[13]: let rec nlength = function
         | [] -> 0
         | _ :: xs -> 1 + nlength xs

Out[13]: val nlength : 'a list -> int = <fun>
```

nlength [3; 5; 9] is evaluated as follows:

$$\begin{aligned}
\text{nlength } [3;\ 5;\ 9] &\Rightarrow 1 + \text{nlength } [5;\ 9] \\
&\Rightarrow 1 + (1 + \text{nlength } [9]) \\
&\Rightarrow 1 + (1 + (1 + \text{nlength } [])) \\
&\Rightarrow 1 + (1 + (1 + 0)) \\
&\Rightarrow \ldots \Rightarrow 3
\end{aligned}$$

Computing the Length of a List

```
 In[13]: let rec nlength = function
         | [] -> 0
         | _ :: xs -> 1 + nlength xs

Out[13]: val nlength : 'a list -> int = <fun>
```

nlength [3; 5; 9] is evaluated as follows:

$$\begin{aligned}
\text{nlength}\ [3;\ 5;\ 9] &\Rightarrow 1 + \text{nlength}\ [5;\ 9] \\
&\Rightarrow 1 + (1 + \text{nlength}\ [9]) \\
&\Rightarrow 1 + (1 + (1 + \text{nlength}\ [])) \\
&\Rightarrow 1 + (1 + (1 + 0)) \\
&\Rightarrow \ldots \Rightarrow 3
\end{aligned}$$

What is the **time and space complexity** of this function?

Efficiently Computing the Length of a List

```
 In[14]: let rec addlen n = function
           | [] -> n
           | _::xs -> addlen (n + 1) xs

Out[14]: val addlen : int -> 'a list -> int = <fun>

 In[15]: let length xs = addlen 0 xs

Out[15]: val length : 'a list -> int = <fun>
```

Efficiently Computing the Length of a List

```
 In[14]:  let rec addlen n = function
          | [] -> n
          | _::xs -> addlen (n + 1) xs

Out[14]:  val addlen : int -> 'a list -> int = <fun>

 In[15]:  let length xs = addlen 0 xs

Out[15]:  val length : 'a list -> int = <fun>
```

Efficiently Computing the Length of a List

```
 In[14]:  let rec addlen n = function
          | [] -> n
          | _::xs -> addlen (n + 1) xs

Out[14]:  val addlen : int -> 'a list -> int = <fun>

 In[15]:  let length xs = addlen 0 xs

Out[15]:  val length : 'a list -> int = <fun>
```

Efficiently Computing the Length of a List

```
 In[14]:  let rec addlen n = function
          | [] -> n
          | _::xs -> addlen (n + 1) xs

Out[14]:  val addlen : int -> 'a list -> int = <fun>

 In[15]:  let length xs = addlen 0 xs

Out[15]:  val length : 'a list -> int = <fun>
```

Efficiently Computing the Length of a List

```
 In[14]: let rec addlen n = function
         | [] -> n
         | _::xs -> addlen (n + 1) xs

Out[14]: val addlen : int -> 'a list -> int = <fun>

 In[15]: let length xs = addlen 0 xs

Out[15]: val length : 'a list -> int = <fun>
```

Efficiently Computing the Length of a List

```
 In[14]: let rec addlen n = function
         | [] -> n
         | _::xs -> addlen (n + 1) xs
Out[14]: val addlen : int -> 'a list -> int = <fun>
 In[15]: let length xs = addlen 0 xs
Out[15]: val length : 'a list -> int = <fun>
```

length [3; 5; 9]

Efficiently Computing the Length of a List

```
 In[14]: let rec addlen n = function
         | [] -> n
         | _::xs -> addlen (n + 1) xs

Out[14]: val addlen : int -> 'a list -> int = <fun>

 In[15]: let length xs = addlen 0 xs

Out[15]: val length : 'a list -> int = <fun>
```

$$\text{length } [3;\ 5;\ 9] \quad \Rightarrow \quad \text{addlen } 0\ [3;\ 5;\ 9]$$

Efficiently Computing the Length of a List

```
 In[14]: let rec addlen n = function
         | [] -> n
         | _::xs -> addlen (n + 1) xs

Out[14]: val addlen : int -> 'a list -> int = <fun>

 In[15]: let length xs = addlen 0 xs

Out[15]: val length : 'a list -> int = <fun>
```

$$\begin{aligned}\text{length } [3;\ 5;\ 9] &\Rightarrow \text{addlen } 0\ [3;\ 5;\ 9]\\ &\Rightarrow \text{addlen } 1\ [5;\ 9]\end{aligned}$$

Efficiently Computing the Length of a List

```
 In[14]: let rec addlen n = function
         | [] -> n
         | _::xs -> addlen (n + 1) xs

Out[14]: val addlen : int -> 'a list -> int = <fun>

 In[15]: let length xs = addlen 0 xs

Out[15]: val length : 'a list -> int = <fun>
```

$$
\begin{aligned}
\text{length } [3;\ 5;\ 9] &\Rightarrow \text{addlen } 0\ [3;\ 5;\ 9] \\
&\Rightarrow \text{addlen } 1\ [5;\ 9] \\
&\Rightarrow \text{addlen } 2\ [9]
\end{aligned}
$$

Efficiently Computing the Length of a List

```
 In[14]: let rec addlen n = function
         | [] -> n
         | _::xs -> addlen (n + 1) xs

Out[14]: val addlen : int -> 'a list -> int = <fun>

 In[15]: let length xs = addlen 0 xs

Out[15]: val length : 'a list -> int = <fun>
```

$$
\begin{aligned}
\text{length } [3;\ 5;\ 9] &\Rightarrow \text{addlen } 0\ [3;\ 5;\ 9]\\
&\Rightarrow \text{addlen } 1\ [5;\ 9]\\
&\Rightarrow \text{addlen } 2\ [9]\\
&\Rightarrow \text{addlen } 3\ []
\end{aligned}
$$

Efficiently Computing the Length of a List

```
 In[14]:  let rec addlen n = function
          | [] -> n
          | _::xs -> addlen (n + 1) xs

Out[14]:  val addlen : int -> 'a list -> int = <fun>

 In[15]:  let length xs = addlen 0 xs

Out[15]:  val length : 'a list -> int = <fun>
```

$$
\begin{aligned}
\text{length } [3;\ 5;\ 9] &\Rightarrow \text{addlen } 0\ [3;\ 5;\ 9] \\
&\Rightarrow \text{addlen } 1\ [5;\ 9] \\
&\Rightarrow \text{addlen } 2\ [9] \\
&\Rightarrow \text{addlen } 3\ [] \\
&\Rightarrow 3
\end{aligned}
$$

Efficiently Computing the Length of a List

```
 In[14]: let rec addlen n = function
         | [] -> n
         | _::xs -> addlen (n + 1) xs

Out[14]: val addlen : int -> 'a list -> int = <fun>

 In[15]: let length xs = addlen 0 xs

Out[15]: val length : 'a list -> int = <fun>
```

$$
\begin{aligned}
\text{length } [3;\ 5;\ 9] &\Rightarrow \text{addlen } 0\ [3;\ 5;\ 9] \\
&\Rightarrow \text{addlen } 1\ [5;\ 9] \\
&\Rightarrow \text{addlen } 2\ [9] \\
&\Rightarrow \text{addlen } 3\ [] \\
&\Rightarrow 3
\end{aligned}
$$

What is the **time and space complexity** of this function?