Foundations of Computer Science
Lecture #3: Lists

Dr. Amanda Prorok & Dr. Anil Madhavapeddy

2019-2020

Warm-Up

Question 1: What does this return?

```
In:  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, given 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

```
[3; 5; 9] : int list

[[3.1]; []; [5.7; -0.6]] : (float list) list
```

Lists

```
In[1]:   let it = [3; 5; 9];;
Out[1]:  val it : int list = [3; 5; 9]
```

append →

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

reverse →

```
In[3]:   List.rev [(1, “one"); (2, "two")];;
Out[3]:  - : (int * string) list = [(2, "two"); (1, "one")]
```

The List Primitives

- We build a list using two primitives

`[ ]`

`: :`

The list `[3; 5; 9]` is constructed as:

```
9::[] = [9]
5::[9] = [5; 9]
3::[5; 9] = [3; 5; 9]
```

The List Primitives

The two kinds of list

[ ] is the empty list

$x :: l$ is the list with head x and tail l

List notation

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

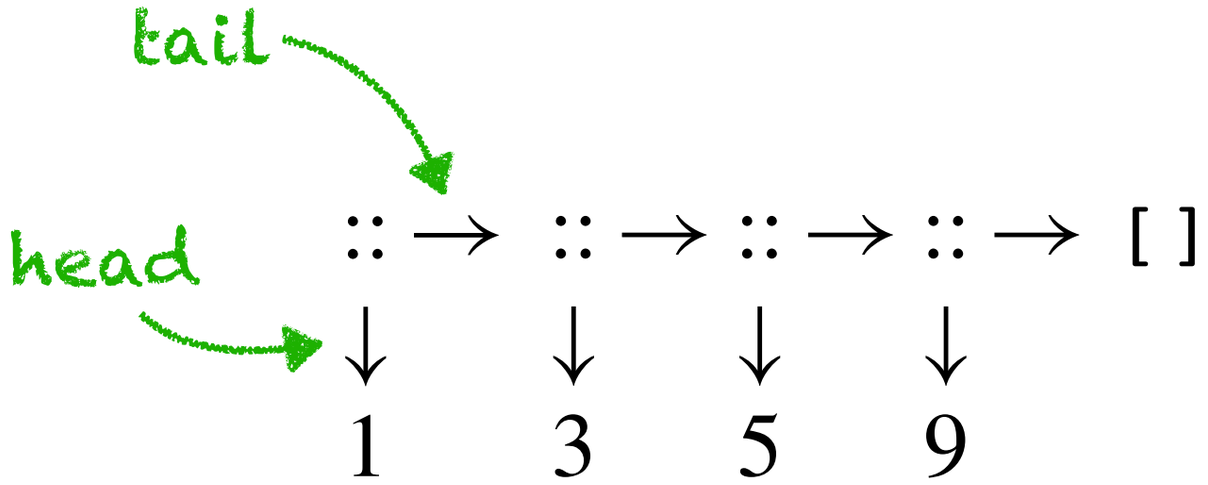
head

a′ :: a′ list

head :: tail

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

Out: val up_to : int -> int -> int list = <fun>


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

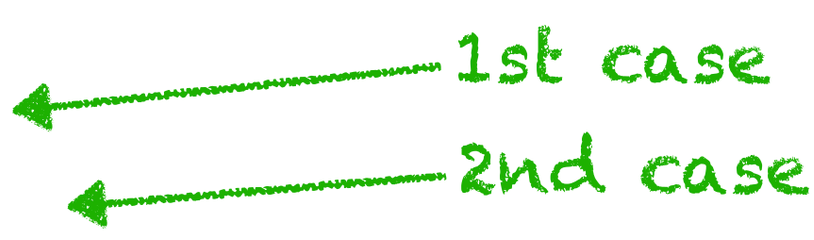
Getting at the Head and Tail

```
In:   let hd (x::_) = x;;

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

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

```
 In:  let null = function
      | []    -> true
      | _::_  -> false;;

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

pattern-matching:



Getting at the Head and Tail

Note that these three functions are polymorphic

```
null : 'a list -> bool
hd   : 'a list -> 'a
tl   : 'a list -> 'a list
```

is a list empty?

head of a non-empty list

tail of a non-empty list

alpha type: type variable

Computing the Length of a List

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

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

nlength `[3; 5; 9]` is constructed as:

```
nlength [a; b; c] ⇒ 1 + nlength [b; c]
                  ⇒ 1 + (1 + nlength [c])
                  ⇒ 1 + (1 + (1 + nlength []))
                  ⇒ 1 + (1 + (1 + 0)
                  ⇒ … ⇒ 3
```

base case!

What is the time and space complexity of this function?

Efficiently **Computing the Length of a List**

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

Out:  val addlen : int * 'a list -> int = <fun>
```

accumulator

```
addlen(0,[a; b; c]) ⇒ addlen (1,[b; c])
                    ⇒ addlen (2,[c])
                    ⇒ addlen (3,[])
                    ⇒ 3
```

base case!

What is the time and space complexity of this function?

Efficiently Computing the Length of a List

```
In:  let length xs = addlen (0, xs);;
Out: val length : 'a list -> int = <fun>
```

Append: List Concatenation

@

```
In:   let rec append = function
        | ([], ys)     -> ys
        | (x::xs, ys) -> x :: append (xs, ys)

Out:  val append : 'a list * 'a list -> 'a list = <fun>
```

```
append([1; 2; 3],[4]) ⇒ 1 :: append ([2; 3],[4])
                      ⇒ 1 :: (2 :: append ([3],[4]))
                      ⇒ 1 :: (2 :: (3 :: append ([],[4])))
                      ⇒ 1 :: (2 :: (3 :: [4]))
                      ⇒ [1; 2; 3; 4]
```

base case!

What is the time and space complexity of this function?

Reversing a List in $O(n^2)$

```
In:   let rec nrev = function
      | []    -> []
      | x::xs -> (nrev xs) @ [x];;

Out:  val nrev : 'a list-> 'a list = <fun>
```

```
nrev [a; b; c] ⇒ nrev [b; c] @ [a]
               ⇒ (nrev [c] @ [b]) @ [a]
               ⇒ ((nrev [] @ [c]) @ [b]) @ [a]
               ⇒ ([] @ [c]) @ [b]) @ [a] ⇒ … ⇒ [c; b; a]
```

base case: [] is tail!

What is the time and space complexity of this function?

Reversing a List in $O(n^2)$

```
In:   let rec nrev = function
      | []    -> []
      | x::xs -> (nrev xs) @ [x];;

Out:  val nrev : 'a list-> 'a list = <fun>
```

```
nrev [a; b; c] ⇒ nrev [b; c] @ [a]
               ⇒ (nrev [c] @ [b]) @ [a]
               ⇒ ((nrev [] @ [c]) @ [b]) @ [a]
               ⇒ ([] @ [c]) @ [b]) @ [a] ⇒ … ⇒ [c; b; a]
```

Recall: append is O(n), and we have n(n+1) / 2 conses, which is $O(n^2)$

Reversing a List in $O(n)$

```
In:  let rec rev_app = function
       | ([], ys)     -> ys
       | (x::xs, ys) -> rev_app (xs, x::ys);;
```

accumulator

```
Out: val rev_app : 'a list * 'a list -> 'a list = <fun>
```

```
rev_app ([a; b; c], []) ⇒ rev_app ([b; c], [a])
                        ⇒ rev_app ([c], [b; a])
                        ⇒ rev_app ([], [c; b; a])
                        ⇒ [c; b; a]
```

What is the time complexity of this function?

Reversing a List in $O(n)$

```
In:     let rev xs = rev_app (xs, [])

Out:    val rev : 'a list -> 'a list = <fun>
```

Lists, Strings, and Characters

character constants	`'A'  '"'  ...`
string constants	`""  "B"  "Oh, no!"  ...`
`String.length` s	*number of chars in string* s
s_1`^`s_2	*concatenation* of strings s_1 and s_2

Also:
The operators `<  <=  >  >=` work for strings and yield lexicographic order

```
In:   'a' < 'b';;
Out:  - : bool = true
```