Foundations of Computer Science

Appending & reversing lists

Dr. Robert Harle & Dr. Jeremy Yallop

2020–2021

Append: List Concatenation

```
In[1]:  let rec append xs ys =
          match xs with
          | [] -> ys
          | x::xs -> x :: append xs ys

Out[1]: val append : 'a list -> 'a list -> 'a list = <fun>
```

Append: List Concatenation

```
In[1]:  let rec append xs ys =
          match xs with
          | [] -> ys
          | x::xs -> x :: append xs ys

Out[1]: val append : 'a list -> 'a list -> 'a list = <fun>
```

```
In[1]:  let rec append xs ys =
          match xs with
          | [] -> ys
          | x::xs -> x :: append xs ys

Out[1]: val append : 'a list -> 'a list -> 'a list = <fun>
```

```
In[1]: let rec append xs ys =
         match xs with
         | [] -> ys
         | x::xs -> x :: append xs ys

Out[1]: val append : 'a list -> 'a list -> 'a list = <fun>
```

append [1; 2; 3] [4]

Append: List Concatenation

```
In[1]:  let rec append xs ys =
          match xs with
          | [] -> ys
          | x::xs -> x :: append xs ys

Out[1]: val append : 'a list -> 'a list -> 'a list = <fun>
```

append [1; 2; 3] [4] $\Rightarrow$ 1 :: append [2;3] [4]

Append: List Concatenation

```
In[1]:  let rec append xs ys =
          match xs with
          | [] -> ys
          | x::xs -> x :: append xs ys

Out[1]: val append : 'a list -> 'a list -> 'a list = <fun>
```

$$\text{append } [1;\ 2;\ 3]\ \ [4] \quad \Rightarrow \quad 1\ ::\ \text{append } [2;3]\ \ [4]$$
$$\Rightarrow \quad 1\ ::\ (2\ ::\ \text{append } [3]\ \ [4])$$

In[1]:

```
let rec append xs ys =
  match xs with
  | [] -> ys
  | x::xs -> x :: append xs ys
```

Out[1]:

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

$$
\begin{aligned}
\text{append } [1;\ 2;\ 3]\ \ [4] &\Rightarrow 1\ ::\ \text{append } [2;3]\ \ [4] \\
&\Rightarrow 1\ ::\ (2\ ::\ \text{append } [3]\ \ [4]) \\
&\Rightarrow 1\ ::\ (2\ ::\ (3\ ::\ \text{append } []\ \ [4]))
\end{aligned}
$$

Append: List Concatenation

In[1]:
```
let rec append xs ys =
  match xs with
  | [] -> ys
  | x::xs -> x :: append xs ys
```

Out[1]:
```
val append : 'a list -> 'a list -> 'a list = <fun>
```

$$
\begin{aligned}
\text{append } [1;\ 2;\ 3]\ \ [4] \quad &\Rightarrow \quad 1\ ::\ \text{append } [2;3]\ \ [4] \\
&\Rightarrow \quad 1\ ::\ (2\ ::\ \text{append } [3]\ \ [4]) \\
&\Rightarrow \quad 1\ ::\ (2\ ::\ (3\ ::\ \text{append } []\ \ [4])) \\
&\Rightarrow \quad 1\ ::\ (2\ ::\ (3\ ::\ \ [4]))
\end{aligned}
$$

```
In[1]:  let rec append xs ys =
          match xs with
          | [] -> ys
          | x::xs -> x :: append xs ys

Out[1]: val append : 'a list -> 'a list -> 'a list = <fun>
```

append [1; 2; 3] [4] $\Rightarrow$ 1 :: append [2;3] [4]
$\Rightarrow$ 1 :: (2 :: append [3] [4])
$\Rightarrow$ 1 :: (2 :: (3 :: append [] [4]))
$\Rightarrow$ 1 :: (2 :: (3 :: [4]))
$\Rightarrow$ [1; 2; 3; 4]

```
In[1]: let rec append xs ys =
         match xs with
         | [] -> ys
         | x::xs -> x :: append xs ys

Out[1]: val append : 'a list -> 'a list -> 'a list = <fun>
```

$$
\begin{aligned}
\text{append } [1;\ 2;\ 3]\ [4] \quad &\Rightarrow \quad 1 :: \text{append } [2;3]\ [4] \\
&\Rightarrow \quad 1 :: (2 :: \text{append } [3]\ [4]) \\
&\Rightarrow \quad 1 :: (2 :: (3 :: \text{append } []\ [4])) \\
&\Rightarrow \quad 1 :: (2 :: (3 :: [4])) \\
&\Rightarrow \quad [1;\ 2;\ 3;\ 4]
\end{aligned}
$$

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

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

```
In[2]:  let rec nrev = function
          | [] -> []
          | x::xs -> (nrev xs) @ [x]

Out[2]: val nrev : 'a list-> 'a list = <fun>
```

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

```
In[2]:  let rec nrev = function
          | [] -> []
          | x::xs -> (nrev xs) @ [x]

Out[2]: val nrev : 'a list-> 'a list = <fun>
```

In[2]:

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

Out[2]:

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

```
In[2]: let rec nrev = function
         | [] -> []
         | x::xs -> (nrev xs) @ [x]

Out[2]: val nrev : 'a list-> 'a list = <fun>
```

nrev [a; b; c]

Reversing a List in O(n^2)

In[2]:
```
let rec nrev = function
  | [] -> []
  | x::xs -> (nrev xs) @ [x]
```

Out[2]:
```
val nrev : 'a list-> 'a list = <fun>
```

nrev [a; b; c] $\Rightarrow$ nrev [b; c] @ [a]

In[2]:
```
let rec nrev = function
  | [] -> []
  | x::xs -> (nrev xs) @ [x]
```

Out[2]:
```
val nrev : 'a list-> 'a list = <fun>
```

$$
\begin{aligned}
\text{nrev [a; b; c]} &\Rightarrow \text{nrev [b; c] @ [a]} \\
&\Rightarrow \text{(nrev [c] @ [b]) @ [a]}
\end{aligned}
$$

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

```
In[2]:  let rec nrev = function
          | [] -> []
          | x::xs -> (nrev xs) @ [x]

Out[2]: val nrev : 'a list-> 'a list = <fun>
```

$$
\begin{aligned}
\text{nrev } [a;\ b;\ c] &\Rightarrow \text{nrev } [b;\ c] \ @\ [a] \\
&\Rightarrow (\text{nrev } [c] \ @\ [b]) \ @\ [a] \\
&\Rightarrow ((\text{nrev } [] \ @\ [c]) \ @\ [b]) \ @\ [a]
\end{aligned}
$$

```
In[2]:  let rec nrev = function
          | [] -> []
          | x::xs -> (nrev xs) @ [x]

Out[2]: val nrev : 'a list-> 'a list = <fun>
```

$$
\begin{aligned}
\text{nrev } [a;\ b;\ c] &\Rightarrow \text{nrev } [b;\ c]\ @\ [a] \\
&\Rightarrow (\text{nrev } [c]\ @\ [b])\ @\ [a] \\
&\Rightarrow ((\text{nrev } []\ @\ [c])\ @\ [b])\ @\ [a] \\
&\Rightarrow (([]\ @\ [c])\ @\ [b])\ @\ [a]
\end{aligned}
$$

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

In[2]:
```
let rec nrev = function
    | [] -> []
    | x::xs -> (nrev xs) @ [x]
```

Out[2]:
```
val nrev : 'a list-> 'a list = <fun>
```

$$
\begin{aligned}
\text{nrev } [a;\ b;\ c] &\Rightarrow \text{nrev } [b;\ c] \ @\ [a] \\
&\Rightarrow (\text{nrev } [c] \ @\ [b]) \ @\ [a] \\
&\Rightarrow ((\text{nrev } [] \ @\ [c]) \ @\ [b]) \ @\ [a] \\
&\Rightarrow (([] \ @\ [c]) \ @\ [b]) \ @\ [a] \\
&\Rightarrow [c;\ b;\ a]
\end{aligned}
$$

In[2]:

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

Out[2]:

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

$$
\begin{aligned}
\text{nrev [a; b; c]} &\Rightarrow \text{nrev [b; c] @ [a]} \\
&\Rightarrow \text{(nrev [c] @ [b]) @ [a]} \\
&\Rightarrow \text{((nrev [] @ [c]) @ [b]) @ [a]} \\
&\Rightarrow \text{(([] @ [c]) @ [b]) @ [a]} \\
&\Rightarrow \text{[c; b; a]}
\end{aligned}
$$

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

```
In[2]: let rec nrev = function
         | [] -> []
         | x::xs -> (nrev xs) @ [x]

Out[2]: val nrev : 'a list-> 'a list = <fun>
```

$$
\begin{array}{lll}
\text{nrev [a; b; c]} & \Rightarrow & \text{nrev [b; c] @ [a]} \\
 & \Rightarrow & \text{(nrev [c] @ [b]) @ [a]} \\
 & \Rightarrow & \text{((nrev [] @ [c]) @ [b]) @ [a]} \\
 & \Rightarrow & \text{(([] @ [c]) @ [b]) @ [a]} \\
 & \Rightarrow & \text{[c; b; a]}
\end{array}
$$

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

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[3]:  let rec rev_app xs ys =
          match xs with
         | [] -> ys   ⟵ accumulator
         | x::xs -> rev_app xs (x::ys)

Out[3]: val rev_app : 'a list -> 'a list -> 'a list = <fun>
```

Reversing a List in O(n)

```
In[3]:  let rec rev_app xs ys =
          match xs with
         | [] -> ys   ⟵ accumulator
         | x::xs -> rev_app xs (x::ys)

Out[3]: val rev_app : 'a list -> 'a list -> 'a list = <fun>
```

```
In[3]:  let rec rev_app xs ys =
          match xs with
         | [] -> ys   ⟵ accumulator
         | x::xs -> rev_app xs (x::ys)

Out[3]: val rev_app : 'a list -> 'a list -> 'a list = <fun>
```

```
In[3]:  let rec rev_app xs ys =
          match xs with
         | [] -> ys   ⟵ accumulator
         | x::xs -> rev_app xs (x::ys)

Out[3]: val rev_app : 'a list -> 'a list -> 'a list = <fun>
```

rev_app [a; b; c] []

Reversing a List in O(n)

```
In[3]:  let rec rev_app xs ys =
          match xs with
         | [] -> ys        accumulator
         | x::xs -> rev_app xs (x::ys)

Out[3]: val rev_app : 'a list -> 'a list -> 'a list = <fun>
```

rev_app [a; b; c] [] $\Rightarrow$ rev_app [b; c] [a]

Reversing a List in O(n)

```
In[3]:  let rec rev_app xs ys =
          match xs with
          | [] -> ys            accumulator
          | x::xs -> rev_app xs (x::ys)

Out[3]: val rev_app : 'a list -> 'a list -> 'a list = <fun>
```

$$
\begin{aligned}
\text{rev\_app } [a;\ b;\ c]\ \ [] \quad &\Rightarrow \quad \text{rev\_app } [b;\ c]\ \ [a] \\
&\Rightarrow \quad \text{rev\_app } [c]\ \ [b;\ a]
\end{aligned}
$$

```
In[3]:  let rec rev_app xs ys =
          match xs with
         | [] -> ys   ⟵ accumulator
         | x::xs -> rev_app xs (x::ys)

Out[3]: val rev_app : 'a list -> 'a list -> 'a list = <fun>
```

$$
\begin{aligned}
\text{rev\_app [a; b; c] []} &\Rightarrow \text{rev\_app [b; c] [a]} \\
&\Rightarrow \text{rev\_app [c] [b; a]} \\
&\Rightarrow \text{rev\_app [] [c; b; a]}
\end{aligned}
$$

```
In[3]:  let rec rev_app xs ys =
           match xs with
          | [] -> ys  ⟵ accumulator
          | x::xs -> rev_app xs (x::ys)

Out[3]: val rev_app : 'a list -> 'a list -> 'a list = <fun>
```

$$
\begin{aligned}
\text{rev\_app [a; b; c] []} &\Rightarrow \text{rev\_app [b; c] [a]}\\
&\Rightarrow \text{rev\_app [c] [b; a]}\\
&\Rightarrow \text{rev\_app [] [c; b; a]}\\
&\Rightarrow \text{[c; b; a]}
\end{aligned}
$$

```
In[3]: let rec rev_app xs ys =
         match xs with
        | [] -> ys        accumulator
        | x::xs -> rev_app xs (x::ys)

Out[3]: val rev_app : 'a list -> 'a list -> 'a list = <fun>
```

$$
\begin{aligned}
\text{rev\_app } [a;\ b;\ c]\ \ [] \quad &\Rightarrow \quad \text{rev\_app } [b;\ c]\ \ [a] \\
&\Rightarrow \quad \text{rev\_app } [c]\ \ [b;\ a] \\
&\Rightarrow \quad \text{rev\_app } []\ \ [c;\ b;\ a] \\
&\Rightarrow \quad [c;\ b;\ a]
\end{aligned}
$$

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

An **interface** to rev_app:

```
 In[4]: let rev xs = rev_app xs []
Out[4]: val rev : 'a list -> 'a list = <fun>
 In[5]: rev [1;2;3]
Out[5]: - : int list = [3; 2; 1]
```

Reversing a List in $O(n)$

An **interface** to rev_app:

```
 In[4]:  let rev xs = rev_app xs []
Out[4]:  val rev : 'a list -> 'a list = <fun>
 In[5]:  rev [1;2;3]
Out[5]:  - : int list = [3; 2; 1]
```

An **interface** to rev_app:

```
 In[4]: let rev xs = rev_app xs []
Out[4]: val rev : 'a list -> 'a list = <fun>
 In[5]: rev [1;2;3]
Out[5]: - : int list = [3; 2; 1]
```

Reversing a List in O(n)

An **interface** to rev_app:

```
 In[4]: let rev xs = rev_app xs []
Out[4]: val rev : 'a list -> 'a list = <fun>
 In[5]: rev [1;2;3]
Out[5]: - : int list = [3; 2; 1]
```

An **interface** to rev_app:

```
 In[4]: let rev xs = rev_app xs []
Out[4]: val rev : 'a list -> 'a list = <fun>
 In[5]: rev [1;2;3]
Out[5]: - : int list = [3; 2; 1]
```

Cons vs append

Question 3a: What does this return?

```
 In[6]: let a = [2]
Out[6]: val a : int list = [2]
 In[7]: let b = [3; 4; 5]
Out[7]: val b : int list = [3; 4; 5]
 In[8]: a::b  (* Q: what does this return? *)
   Out: Line 1, characters 5-6:
        1 | a :: b
                 ^
        Error: This expression has type int list
               but an expression was expected of type int list list
               Type int is not compatible with type int list
```

Question 3a: What does this return?

```
In[6]:  let a = [2]
Out[6]: val a : int list = [2]
In[7]:  let b = [3; 4; 5]
Out[7]: val b : int list = [3; 4; 5]
In[8]:  a::b  (* Q: what does this return? *)
   Out: Line 1, characters 5-6:
        1 | a :: b
                 ^
        Error: This expression has type int list
               but an expression was expected of type int list list
               Type int is not compatible with type int list
```

Question 3a: What does this return?

```
 In[6]: let a = [2]
Out[6]: val a : int list = [2]
 In[7]: let b = [3; 4; 5]
Out[7]: val b : int list = [3; 4; 5]
 In[8]: a::b  (* Q: what does this return? *)
   Out: Line 1, characters 5-6:
        1 | a :: b
                 ^
        Error: This expression has type int list
               but an expression was expected of type int list list
               Type int is not compatible with type int list
```

Question 3a: What does this return?

```
 In[6]:  let a = [2]
Out[6]:  val a : int list = [2]
 In[7]:  let b = [3; 4; 5]
Out[7]:  val b : int list = [3; 4; 5]
 In[8]:  a::b  (* Q: what does this return? *)
   Out:  Line 1, characters 5-6:
         1 | a :: b
                  ^
         Error: This expression has type int list
                but an expression was expected of type int list list
                Type int is not compatible with type int list
```

Cons vs append

Question 3a: What does this return?

```
In[6]:  let a = [2]
Out[6]: val a : int list = [2]
In[7]:  let b = [3; 4; 5]
Out[7]: val b : int list = [3; 4; 5]
In[8]:  a::b  (* Q: what does this return? *)
   Out: Line 1, characters 5-6:
        1 | a :: b
                 ^
        Error: This expression has type int list
               but an expression was expected of type int list list
               Type int is not compatible with type int list
```

Question 3a: What does this return?

```
 In[6]: let a = [2]
Out[6]: val a : int list = [2]
 In[7]: let b = [3; 4; 5]
Out[7]: val b : int list = [3; 4; 5]
 In[8]: a::b  (* Q: what does this return? *)
   Out: Line 1, characters 5-6:
        1 | a :: b
                 ^
        Error: This expression has type int list
               but an expression was expected of type int list list
               Type int is not compatible with type int list
```

Question 3a: What does this return?

```
 In[6]: let a = [2]
Out[6]: val a : int list = [2]
 In[7]: let b = [3; 4; 5]
Out[7]: val b : int list = [3; 4; 5]
 In[8]: a::b  (* Q: what does this return? *)
   Out: Line 1, characters 5-6:
        1 | a :: b
                 ^
        Error: This expression has type int list
               but an expression was expected of type int list list
               Type int is not compatible with type int list
```

Question 3b: How to concatenate a and b?

Question 3c: How can we redefine b so that a :: b works?

```
  In[9]: a @ b
 Out[9]: - : int list = [2; 3; 4; 5]
 In[10]: let b = [b]
Out[10]: val b : int list list = [[3; 4; 5]]
 In[11]: a::b
Out[11]: - : int list list = [[2]; [3, 4, 5]]
```

Question 3b: How to concatenate a and b?

Question 3c: How can we redefine b so that a :: b works?

```
  In[9]: a @ b
 Out[9]: - : int list = [2; 3; 4; 5]
 In[10]: let b = [b]
Out[10]: val b : int list list = [[3; 4; 5]]
 In[11]: a::b
Out[11]: - : int list list = [[2]; [3, 4, 5]]
```

Question 3b: How to concatenate a and b?

Question 3c: How can we redefine b so that a :: b works?

```
 In[9]:  a @ b
Out[9]:  - : int list = [2; 3; 4; 5]
In[10]:  let b = [b]
Out[10]: val b : int list list = [[3; 4; 5]]
In[11]:  a :: b
Out[11]: - : int list list = [[2]; [3; 4; 5]]
```

Cons vs append, continued

Question 3b: How to concatenate a and b?

Question 3c: How can we redefine b so that a :: b works?

```
 In[9]:  a @ b
Out[9]:  - : int list = [2; 3; 4; 5]
 In[10]: let b = [b]
Out[10]: val b : int list list = [[3; 4; 5]]
 In[11]: a :: b
Out[11]: - : int list list = [[2]; [3; 4; 5]]
```

Cons vs append, continued

Question 3b: How to concatenate a and b?

Question 3c: How can we redefine b so that a :: b works?

```
  In[9]: a @ b
 Out[9]: - : int list = [2; 3; 4; 5]
 In[10]: let b = [b]
Out[10]: val b : int list list = [[3; 4; 5]]
 In[11]: a :: b
Out[11]: - : int list list = [[2]; [3; 4; 5]]
```

Cons vs append, continued

Question 3b: How to concatenate a and b?

Question 3c: How can we redefine b so that a :: b works?

```
  In[9]: a @ b
 Out[9]: - : int list = [2; 3; 4; 5]
 In[10]: let b = [b]
Out[10]: val b : int list list = [[3; 4; 5]]
 In[11]: a::b
Out[11]: - : int list list = [[2]; [3; 4; 5]]
```

Question 3b: How to concatenate a and b?

Question 3c: How can we redefine b so that a :: b works?

```
 In[9]: a @ b
Out[9]: - : int list = [2; 3; 4; 5]
 In[10]: let b = [b]
Out[10]: val b : int list list = [[3; 4; 5]]
 In[11]: a::b
Out[11]: - : int list list = [[2]; [3, 4, 5]]
```

A Note on Notation: match vs function

```
In[12]:  let rec append1 = function
         | ([], ys) -> ys
         | (x::xs, ys) -> x :: append1 (xs, ys)

Out[12]: val append1 : 'a list * 'a list -> 'a list = <fun>

In[13]:  let rec append2 pair =
         match pair with
         | ([], ys) -> ys
         | (x::xs, ys) -> x :: append2 (xs, ys)

Out[13]: val append2 : 'a list * 'a list -> 'a list = <fun>
```

A Note on Notation: match vs function

```
In[12]:  let rec append1 = function
         | ([], ys) -> ys
         | (x::xs, ys) -> x :: append1 (xs, ys)

Out[12]: val append1 : 'a list * 'a list -> 'a list = <fun>

In[13]:  let rec append2 pair =
         match pair with
         | ([], ys) -> ys
         | (x::xs, ys) -> x :: append2 (xs, ys)

Out[13]: val append2 : 'a list * 'a list -> 'a list = <fun>
```

A Note on Notation: match vs function

In[12]:
```
let rec append1 = function
| ([], ys) -> ys
| (x::xs, ys) -> x :: append1 (xs, ys)
```

Out[12]:
```
val append1 : 'a list * 'a list -> 'a list = <fun>
```

In[13]:
```
let rec append2 pair =
match pair with
| ([], ys) -> ys
| (x::xs, ys) -> x :: append2 (xs, ys)
```

Out[13]:
```
val append2 : 'a list * 'a list -> 'a list = <fun>
```

A Note on Notation: match vs function

```
In[12]:  let rec append1 = function
         | ([], ys) -> ys
         | (x::xs, ys) -> x :: append1 (xs, ys)

Out[12]: val append1 : 'a list * 'a list -> 'a list = <fun>

In[13]:  let rec append2 pair =
         match pair with
         | ([], ys) -> ys
         | (x::xs, ys) -> x :: append2 (xs, ys)

Out[13]: val append2 : 'a list * 'a list -> 'a list = <fun>
```

A Note on Notation: match vs function

```
In[12]:  let rec append1 = function
         | ([], ys) -> ys
         | (x::xs, ys) -> x :: append1 (xs, ys)

Out[12]: val append1 : 'a list * 'a list -> 'a list = <fun>

In[13]:  let rec append2 pair =
         match pair with
         | ([], ys) -> ys
         | (x::xs, ys) -> x :: append2 (xs, ys)

Out[13]: val append2 : 'a list * 'a list -> 'a list = <fun>
```

A Note on Notation: Multiple vs Single match

```
In[14]: let rec append3 xs ys =
        match xs, ys with
        | [], ys -> ys
        | x::xs, ys -> x :: append3 xs ys

Out[14]: val append3 : 'a list -> 'a list -> 'a list = <fun>

In[15]: let rec append4 xs ys =
        match xs with
        | [] -> ys
        | x::xs -> x :: append4 xs ys

Out[15]: val append4 : 'a list -> 'a list -> 'a list = <fun>
```

A Note on Notation: Multiple vs Single match

In[14]:
```
let rec append3 xs ys =
match xs, ys with
| [], ys -> ys
| x::xs, ys -> x :: append3 xs ys
```

Out[14]:
```
val append3 : 'a list -> 'a list -> 'a list = <fun>
```

In[15]:
```
let rec append4 xs ys =
match xs with
| [] -> ys
| x::xs -> x :: append4 xs ys
```

Out[15]:
```
val append4 : 'a list -> 'a list -> 'a list = <fun>
```

A Note on Notation: Multiple vs Single match

```
In[14]:  let rec append3 xs ys =
         match xs, ys with
         | [], ys -> ys
         | x::xs, ys -> x :: append3 xs ys

Out[14]: val append3 : 'a list -> 'a list -> 'a list = <fun>

In[15]:  let rec append4 xs ys =
         match xs with
         | [] -> ys
         | x::xs -> x :: append4 xs ys

Out[15]: val append4 : 'a list -> 'a list -> 'a list = <fun>
```

A Note on Notation: Multiple vs Single match

```
In[14]:  let rec append3 xs ys =
         match xs, ys with
         | [], ys -> ys
         | x::xs, ys -> x :: append3 xs ys

Out[14]: val append3 : 'a list -> 'a list -> 'a list = <fun>

In[15]:  let rec append4 xs ys =
         match xs with
         | [] -> ys
         | x::xs -> x :: append4 xs ys

Out[15]: val append4 : 'a list -> 'a list -> 'a list = <fun>
```

A Note on Notation: Multiple vs Single match

```
In[14]:  let rec append3 xs ys =
         match xs, ys with
         | [], ys -> ys
         | x::xs, ys -> x :: append3 xs ys

Out[14]: val append3 : 'a list -> 'a list -> 'a list = <fun>

In[15]:  let rec append4 xs ys =
         match xs with
         | [] -> ys
         | x::xs -> x :: append4 xs ys

Out[15]: val append4 : 'a list -> 'a list -> 'a list = <fun>
```

Lists, Strings, and Characters

Character constants

'A'   '"'

String constants

"A"   "B"   "Oh, no!"

```
 In[16]: String.length "abcde"
Out[16]: - : int = 5
 In[17]: "Oh," ^ " no!"      (* concatenation *)
Out[17]: - : string = "Oh, no!"
```

Lists, Strings, and Characters

Character constants

'A' '"'

String constants

"A" "B" "Oh, no!"

```
 In[16]: String.length "abcde"
Out[16]: - : int = 5
 In[17]: "Oh," ^ " no!"       (* concatenation *)
Out[17]: - : string = "Oh, no!"
```

Character constants

'A'   '"'

String constants

"A"   "B"   "Oh, no!"

```
 In[16]: String.length "abcde"
Out[16]: - : int = 5
 In[17]: "Oh," ^ " no!"      (* concatenation *)
Out[17]: - : string = "Oh, no!"
```

Character constants

'A' '"'

String constants

"A" "B" "Oh, no!"

```
 In[16]: String.length "abcde"
Out[16]: - : int = 5
 In[17]: "Oh," ^ " no!"      (* concatenation *)
Out[17]: - : string = "Oh, no!"
```

Character constants

'A' '"'

String constants

"A" "B" "Oh, no!"

```
 In[16]: String.length "abcde"
Out[16]: - : int = 5
 In[17]: "Oh," ^ " no!"      (* concatenation *)
Out[17]: - : string = "Oh, no!"
```