Recursive Functions

```plaintext
# let rec factorial n = 
   if n = 0 then 1 else n * factorial (n - 1);;
val factorial : int -> int = <fun>

# factorial 5;;
- : int = 120

# (* rec is needed for recursive function declarations *)
```
Recursion Example

Compute $n^2$ recursively using:

$$n^2 = (2 \times n - 1) + (n - 1)^2$$

```ocaml
# let rec nthsq n =         (* rec for recursion *)
  match n              (* pattern matching for cases *)
  with 0 -> 0                  (* base case *)
  | n -> (2 * n -1)           (* recursive case *)
     + nthsq (n -1);;   (* recursive call *)
val nthsq : int -> int = <fun>
# nthsq 3;;
- : int = 9
```

Structure of recursion similar to inductive proof
Recursion and Induction

```ocaml
# let rec nthsq n = match n with 0 -> 0
  | n -> (2 * n - 1) + nthsq (n - 1) ;;
```

- Base case is the last case; it stops the computation
- Recursive call must be to arguments that are somehow smaller - must progress to base case
- `if` or `match` must contain base case
- Failure of these may cause failure of termination
Evaluating expressions in OCaml

- Evaluation uses an environment $\rho$
  - $\text{Eval} (e, \rho)$

- A constant evaluates to itself, including primitive operators like $+$ and $=$
  - $\text{Eval} (c, \rho) \Rightarrow \text{Val} c$

- To evaluate a variable $v$, look it up in $\rho$:
  - $\text{Eval} (v, \rho) \Rightarrow \text{Val} (\rho(v))$
Evaluating expressions in OCaml

- To evaluate a tuple \((e_1, \ldots, e_n)\),
  - Evaluate each \(e_i\) to \(v_i\), right to left for OCaml
  - Then make value \((v_1, \ldots, v_n)\)
  - \(\text{Eval}((e_1, \ldots, e_n), \rho) \Rightarrow \text{Eval}((e_1, \ldots, \text{Eval} (e_n, \rho)), \rho)\)
  - \(\text{Eval}((e_1, \ldots, e_i, \text{Val} v_{i+1}, \ldots, \text{Val} v_n), \rho) \Rightarrow \text{Eval}((e_1, \ldots, \text{Eval}(e_i, \rho), \text{Val} v_{i+1}, \ldots, \text{Val} v_n), \rho)\)
  - \(\text{Eval}((\text{Val} v_1, \ldots, \text{Val} v_n), \rho) \Rightarrow \text{Val} (v_1, \ldots, v_n)\)
Evaluating expressions in OCaml

- To evaluate uses of +, -, *, +., etc, eval args, then do operation ⊗ (+, -, *, +., ....)
  - Eval(e₁ ⊗ e₂, ρ) => Eval(e₁ ⊗ Eval(e₂, ρ), ρ))
  - Eval(e₁ ⊗ Val e₂, ρ) => Eval(Eval(e₁, ρ) ⊗ Val v₂, ρ))
  - Eval(Val v₁ ⊗ Val v₂) => Val (v₁ ⊗ v₂)

- Function expression evaluates to its closure
  - Eval (fun x -> e, ρ) => Val < x -> e, ρ>
Evaluating expressions in OCaml

To evaluate a local dec: `let x = e1 in e2`
- Eval `e1` to `v`, then eval `e2` using `{x → v} + ρ`

- Eval(let `x = e1` in `e2`, ρ) =>
  Eval(let `x = Eval(e1, ρ)` in `e2`, ρ)

- Eval(let `x = Val v` in `e2`, ρ) =>
  Eval(e2, `{x → v} + ρ`)
Evaluating expressions in OCaml

To evaluate a conditional expression:
`if b then e₁ else e₂`
- Evaluate `b` to a value `v`
- If `v` is `True`, evaluate `e₁`
- If `v` is `False`, evaluate `e₂`

- `Eval(if b then e₁ else e₂, ρ) => Eval(if Eval(b, ρ) then e₁ else e₂, ρ)`
- `Eval(if Val true then e₁ else e₂, ρ) => Eval(e₁, ρ)`
- `Eval(if Val false then e₁ else e₂, ρ) => Eval(e₂, ρ)`
Evaluation of Application with Closures

- Given application expression $f \, e$
- In Ocaml, evaluate $e$ to value $v$
- In environment $\rho$, evaluate left term to closure, $c = \langle (x_1, \ldots, x_n) \to b, \rho' \rangle$
  - $(x_1, \ldots, x_n)$ variables in (first) argument
  - $v$ must have form $(v_1, \ldots, v_n)$
- Update the environment $\rho'$ to $\rho'' = \{ x_1 \to v_1, \ldots, x_n \to v_n \} + \rho'$
- Evaluate body $b$ in environment $\rho''$
Evaluation of Application with Closures

- Eval(f e, ρ) => Eval(f (Eval(e, ρ)), ρ)

- Eval(f (Val v), ρ) => Eval((Eval(f, ρ)) (Val v), ρ)

- Eval((Val <(x₁,...,xₙ) → b, ρ’)>(Val (v₁,...,vₙ)), ρ) => Eval(b, {x₁ → v₁,..., xₙ → vₙ})+ρ’
Evaluation of Application of plus_x;

- Have environment:
  \[ \rho = \{ \text{plus}_x \rightarrow <y \rightarrow y + x, \rho_{\text{plus}_x} >, \ldots, y \rightarrow 19, x \rightarrow 17, z \rightarrow 3, \ldots \} \]
  where \( \rho_{\text{plus}_x} = \{ x \rightarrow 12, \ldots, y \rightarrow 24, \ldots \} \)

- Eval (plus_x z, \rho) =>

- Eval(plus_x (Eval(z, \rho))) => ...
Evaluation of Application of plus_x;;

- Have environment:
  \[ \rho = \{ \text{plus}_x \rightarrow \langle y \rightarrow y + x, \; \rho_{\text{plus}_x} \rangle, \ldots, \; y \rightarrow 19, \; x \rightarrow 17, \; z \rightarrow 3, \; \ldots \} \]

  where \( \rho_{\text{plus}_x} = \{x \rightarrow 12, \; \ldots, \; y \rightarrow 24, \; \ldots \} \)

- \( \text{Eval} (\text{plus}_x \; z, \; \rho) \) =>
- \( \text{Eval}(\text{plus}_x \; (\text{Eval}(z, \; \rho)), \; \rho) \) =>
- \( \text{Eval}(\text{plus}_x \; (\text{Val} \; 3), \; \rho) \) =>  ...
Evaluation of Application of plus_x;;

- Have environment:

\[ \rho = \{\text{plus}_x \rightarrow <y \rightarrow y + x, \rho_{\text{plus}_x}>, \ldots, y \rightarrow 19, x \rightarrow 17, z \rightarrow 3, \ldots\} \]

where \( \rho_{\text{plus}_x} = \{x \rightarrow 12, \ldots, y \rightarrow 24, \ldots\} \)

- Eval (\text{plus}_x z, \rho) =>
- Eval (\text{plus}_x (\text{Eval}(z, \rho)), \rho) =>
- Eval (\text{plus}_x (\text{Val} 3), \rho) =>
- Eval ((\text{Eval}(\text{plus}_x, \rho))(\text{Val} 3), \rho) => ...
Evaluation of Application of plus_x;;

- Have environment:

\[ \rho = \{ \text{plus}_x \rightarrow <y \rightarrow y + x, \rho_{\text{plus}_x} >, \ldots, y \rightarrow 19, x \rightarrow 17, z \rightarrow 3, \ldots \} \]

where \( \rho_{\text{plus}_x} = \{ x \rightarrow 12, \ldots, y \rightarrow 24, \ldots \} \)

- Eval (plus_x z, \( \rho \)) =>
- Eval (plus_x (Eval(z, \( \rho \))), \( \rho \)) =>
- Eval (plus_x (Val 3), \( \rho \)) =>
- Eval (Eval(plus_x, \( \rho \))(Val 3), \( \rho \)) =>
- Eval (Val<y \rightarrow y + x, \( \rho_{\text{plus}_x} >))(Val 3 ), \( \rho \)) => ...
Evaluation of Application of plus_x;;

- Have environment:

\[ \rho = \{ \text{plus}_x \rightarrow <y \rightarrow y + x, \ \rho_{\text{plus}_x} >, \ldots, \ y \rightarrow 19, \ x \rightarrow 17, \ z \rightarrow 3, \ldots \} \]

where \[ \rho_{\text{plus}_x} = \{ x \rightarrow 12, \ldots, \ y \rightarrow 24, \ldots \} \]

- Eval ((Val<\ y \rightarrow \ y + x, \ \rho_{\text{plus}_x} >)(Val 3 ), \ \rho) => ...
Evaluation of Application of plus_x;;

- Have environment:
  \[\rho = \{\text{plus}_x \rightarrow <y \rightarrow y + x, \rho_{\text{plus}_x}>, \ldots, y \rightarrow 19, x \rightarrow 17, z \rightarrow 3, \ldots\}\]

  where \(\rho_{\text{plus}_x} = \{x \rightarrow 12, \ldots, y \rightarrow 24, \ldots\}\)

- Eval \(((\text{Val}<y \rightarrow y + x, \rho_{\text{plus}_x}>)(\text{Val} 3), \rho)\) =>

- Eval \((y + x, \{y \rightarrow 3\} + \rho_{\text{plus}_x})\) => ...
Evaluation of Application of plus_x;;

- Have environment:
  \[ \rho = \{ \text{plus}_x \rightarrow <y \rightarrow y + x, \rho_{\text{plus}_x} >, \ldots, y \rightarrow 19, x \rightarrow 17, z \rightarrow 3, \ldots \} \]
  where \( \rho_{\text{plus}_x} = \{ x \rightarrow 12, \ldots, y \rightarrow 24, \ldots \} \)

- \[ \text{Eval} \left( \left( \text{Val} <y \rightarrow y + x, \rho_{\text{plus}_x} > \right) \left( \text{Val} 3 \right), \rho \right) \]
  =>

- \[ \text{Eval} \left( y + x, \{ y \rightarrow 3 \} + \rho_{\text{plus}_x} \right) \] =>

- \[ \text{Eval} \left( y + \text{Eval}(x, \{ y \rightarrow 3 \} + \rho_{\text{plus}_x}), \{ y \rightarrow 3 \} + \rho_{\text{plus}_x} \right) \] => ...
Evaluation of Application of $\text{plus}_x$;

- Have environment:
  \[
  \rho = \{ \text{plus}_x \rightarrow <y \rightarrow y + x, \rho_{\text{plus}_x} >, \ldots, \\
  y \rightarrow 19, x \rightarrow 17, z \rightarrow 3, \ldots \}\n  \]

  where $\rho_{\text{plus}_x} = \{ x \rightarrow 12, \ldots, y \rightarrow 24, \ldots \}$

- Eval \((\text{Val} <y \rightarrow y + x, \rho_{\text{plus}_x} >)(\text{Val} 3 ), \rho)\) =>

- Eval \((y + x, \{y \rightarrow 3\} + \rho_{\text{plus}_x} )\) =>

- Eval\((y + \text{Eval}(x, \{y \rightarrow 3\} + \rho_{\text{plus}_x} ), \{y \rightarrow 3\} + \rho_{\text{plus}_x} )\) =>

- Eval\((y + \text{Val} 12, \{y \rightarrow 3\} + \rho_{\text{plus}_x} )\) => ...
Evaluation of Application of plus_x;;

Have environment:

\[ \rho = \{ \text{plus}_x \rightarrow <y \rightarrow y + x, \rho_{\text{plus}_x}>, \ldots, y \rightarrow 19, x \rightarrow 17, z \rightarrow 3, \ldots \} \]

where \( \rho_{\text{plus}_x} = \{ x \rightarrow 12, \ldots, y \rightarrow 24, \ldots \} \)

- \( \text{Eval}(y + \text{Eval}(x, \{ y \rightarrow 3 \} + \rho_{\text{plus}_x}), \{ y \rightarrow 3 \} + \rho_{\text{plus}_x}) \Rightarrow \)
- \( \text{Eval}(y + \text{Val} 12, \{ y \rightarrow 3 \} + \rho_{\text{plus}_x}) \Rightarrow \)
- \( \text{Eval}(\text{Eval}(y, \{ y \rightarrow 3 \} + \rho_{\text{plus}_x}) + \text{Val} 12, \{ y \rightarrow 3 \} + \rho_{\text{plus}_x}) \Rightarrow \ldots \)
Evaluation of Application of plus_x;;

- Have environment:
  \[ \rho = \{ \text{plus}_x \rightarrow \langle \text{y} \rightarrow \text{y} + \text{x}, \ \rho_{\text{plus}_x} \rangle, \ldots, \ \text{y} \rightarrow 19, \ \text{x} \rightarrow 17, \ \text{z} \rightarrow 3, \ldots \} \]

  where \[ \rho_{\text{plus}_x} = \{ \text{x} \rightarrow 12, \ldots, \ \text{y} \rightarrow 24, \ldots \} \]

- \[ \text{Eval}(\text{Eval}(\text{y}, \{ \text{y} \rightarrow 3 \} + \rho_{\text{plus}_x}) + \ \text{Val 12}, \{ \text{y} \rightarrow 3 \} + \rho_{\text{plus}_x}) \Rightarrow \]

- \[ \text{Eval}(\text{Val } 3 + \ \text{Val 12}, \{ \text{y} \rightarrow 3 \} + \rho_{\text{plus}_x}) \Rightarrow \ldots \]
Evaluation of Application of plus_x;;

- Have environment:
  \[ \rho = \{ \text{plus}_x \mapsto < y \mapsto y + x, \rho_{\text{plus}_x} >, \ldots, y \mapsto 19, x \mapsto 17, z \mapsto 3, \ldots \} \]

  where \( \rho_{\text{plus}_x} = \{ x \mapsto 12, \ldots, y \mapsto 24, \ldots \} \)

- Eval(Eval(y, \{ y \mapsto 3 \} + \rho_{\text{plus}_x} ) +
  \quad \text{Val 12,}\{ y \mapsto 3 \} + \rho_{\text{plus}_x} ) =>

- Eval(Val 3 + Val 12 ,\{ y \mapsto 3 \} + \rho_{\text{plus}_x} ) =>

- Val (3 + 12) = Val 15
Evaluation of Application of \texttt{plus\_pair}

- Assume environment

\[
\rho = \{ x \mapsto 3 \ldots,
\]

\[
\text{plus\_pair} \mapsto \langle (n, m) \mapsto n + m, \rho_{\text{plus\_pair}} \rangle + \rho_{\text{plus\_pair}}
\]

- \text{Eval (plus\_pair (4, x),} \rho) = \text{>
- \text{Eval (plus\_pair (Eval ((4, x),} \rho)),} \rho) = \text{>
- \text{Eval (plus\_pair (Eval ((4, Eval (x,} \rho)),} \rho)),} \rho) = \text{>
- \text{Eval (plus\_pair (Eval ((4, Val 3),} \rho)),} \rho) = \text{>
- \text{Eval (plus\_pair (Eval ((Eval (4,} \rho), Val 3),} \rho)),} \rho) = \text{>
- \text{Eval (plus\_pair (Eval ((Val 4, Val 3),} \rho)),} \rho) = \text{>}

Evaluation of Application of plus_pair

Assume environment

$\rho = \{ x \rightarrow 3 ... ,$

$$\text{plus_pair } \rightarrow (\langle n,m \rangle \rightarrow n+m, \ \rho_{\text{plus_pair}}) \} + \rho_{\text{plus_pair}}$$

- Eval (plus_pair (Eval ((Val 4, Val 3), $\rho$)), $\rho$) =>
- Eval (plus_pair (Val (4, 3)), $\rho$) =>
- Eval (Eval (plus_pair, $\rho$), Val (4, 3)), $\rho$) => ...
- Eval ((Val $\langle n,m \rangle \rightarrow n+m, \ \rho_{\text{plus_pair}} \rangle)(\text{Val}(4,3)) , \ \rho)=> $
- Eval (n + m, $\{ n \rightarrow 4, m \rightarrow 3 \} + \rho_{\text{plus_pair}}$) =>
- Eval (4 + 3, $\{ n \rightarrow 4, m \rightarrow 3 \} + \rho_{\text{plus_pair}}$) => 7
Lists

List can take one of two forms:

- Empty list, written [ ]
- Non-empty list, written $x :: xs$
  - $x$ is head element, $xs$ is tail list, :: called “cons”

- Syntactic sugar: $[x] == x :: [ ]$
- $[ x1; x2; ...; xn ] == x1 :: x2 :: ... :: xn :: [ ]$
# let fib5 = [8;5;3;2;1;1];;
val fib5 : int list = [8; 5; 3; 2; 1; 1]
# let fib6 = 13 :: fib5;;
val fib6 : int list = [13; 8; 5; 3; 2; 1; 1]
# (8::5::3::2::1::1::[ ]) = fib5;;
- : bool = true
# fib5 @ fib6;;
- : int list = [8; 5; 3; 2; 1; 1; 13; 8; 5; 3; 2; 1; 1]
# let bad_list = [1; 3.2; 7];;

Characters 19-22:

let bad_list = [1; 3.2; 7];;

^^^^

This expression has type float but is here used with type int
Which one of these lists is invalid?

1. [2; 3; 4; 6]
2. [2,3; 4,5; 6,7]
3. [(2.3,4); (3.2,5); (6,7.2)]
4. [[“hi”; “there”]; [“wahcha”]; [ ]; [“doin”]]
Answer

Which one of these lists is invalid?

1. [2; 3; 4; 6]
2. [2,3; 4,5; 6,7]
3. [(2.3,4); (3.2,5); (6,7,2)]
4. [[“hi”; “there”]; [“wahcha”]; [ ]; [“doin”]]

- 3 is invalid because of last pair
Functions Over Lists

```ocaml
# let rec double_up list =
  match list with
  | [] -> [] (* pattern before ->, expression after *)
  | (x :: xs) -> (x :: x :: double_up xs);

val double_up : 'a list -> 'a list = <fun>

# let fib5_2 = double_up fib5;;
val fib5_2 : int list = [8; 8; 5; 5; 3; 3; 2; 2; 1; 1; 1]
```
Functions Over Lists

```ocaml
# let silly = double_up ["hi"; "there"];;
val silly : string list = ["hi"; "hi"; "there"; "there"]

# let rec poor_rev list =
  match list
  with [] -> []
     | (x::xs) -> poor_rev xs @ [x];;
val poor_rev : 'a list -> 'a list = <fun>

# poor_rev silly;;
- : string list = ["there"; "there"; "hi"; "hi"]
```
Structural Recursion

- Functions on recursive datatypes (e.g., lists) tend to be recursive.
- Recursion over recursive datatypes generally by structural recursion.
  - Recursive calls made to components of structure of the same recursive type.
  - Base cases of recursive types stop the recursion of the function.
Question: Length of list

- Problem: write code for the length of the list
  - How to start?

```ml
let rec length list =
```
Question: Length of list

- Problem: write code for the length of the list
  - How to start?

```latex
let rec length list = match list with
```

Question: Length of list

- Problem: write code for the length of the list
  - What patterns should we match against?

```ml
let rec length list =
  match list with
```
Question: Length of list

- Problem: write code for the length of the list
  - What patterns should we match against?

let rec length list =
  match list with [] ->
  | (a :: bs) ->
Question: Length of list

Problem: write code for the length of the list

What result do we give when list is empty?

let rec length list =
  match list with [] -> 0
  | (a :: bs) ->
Problem: write code for the length of the list

What result do we give when list is not empty?

```ml
let rec length list =
  match list with [] -> 0
  | (a :: bs) ->
```
Problem: write code for the length of the list 

What result do we give when list is not empty?

```ocaml
let rec length list =
  match list with [] -> 0
  | (a :: bs) -> 1 + length bs
```
Structural Recursion : List Example

```ml
# let rec length list = match list
  with [ ] -> 0 (* Nil case *)
  | a :: bs -> 1 + length bs;; (* Cons case *)
val length : 'a list -> int = <fun>

# length [5; 4; 3; 2];;
- : int = 4
```

- Nil case `[ ]` is base case
- Cons case case recurses on component list `bs`
How can we efficiently answer if two lists have the same length?
How can we efficiently answer if two lists have the same length?

```ocaml
let rec same_length list1 list2 =
  match list1 with [] ->
    (match list2 with [] -> true
     | (y::ys) -> false)
  | (x::xs) ->
    (match list2 with [] -> false
     | (y::ys) -> same_length xs ys)
```
Your turn: doubleList : int list -> int list

- Write a function that takes a list of int and returns a list of the same length, where each element has been multiplied by 2

let rec doubleList list =
Your turn: doubleList : int list -> int list

- Write a function that takes a list of int and returns a list of the same length, where each element has been multiplied by 2

```ocaml
let rec doubleList list =
  match list
  with [] -> []
  | x :: xs -> (2 * x) :: doubleList xs
```
Your turn: doubleList : int list -> int list

- Write a function that takes a list of int and returns a list of the same length, where each element has been multiplied by 2

```ocaml
let rec doubleList list =
  match list
  with [] -> []
  | x :: xs -> (2 * x) :: doubleList xs
```

9/12/22
Higher-Order Functions Over Lists

```ocaml
# let rec map f list = 
  match list 
  with [] -> [] 
  | (h::t) -> (f h) :: (map f t);;

val map : ('a -> 'b) -> 'a list -> 'b list = <fun>

# map plus_two fib5;;
- : int list = [10; 7; 5; 4; 3; 3]

# map (fun x -> x - 1) fib6;;
: int list = [12; 7; 4; 2; 1; 0; 0]
```
# let rec map f list =

match list
with [ ] -> []
| (h::t) -> (f h) :: (map f t);;

val map : ('a -> 'b) -> 'a list -> 'b list = <fun>

# map plus_two fib5;;
- : int list = [10; 7; 5; 4; 3; 3]
# map (fun x -> x - 1) fib6;;
: int list = [12; 7; 4; 2; 1; 0; 0]
Can use the higher-order recursive map function instead of direct recursion

```ocaml
# let doubleList list = List.map (fun x -> 2 * x) list;;
val doubleList : int list -> int list = <fun>
# doubleList [2;3;4];;
- : int list = [4; 6; 8]
```
Mapping Recursion

- Can use the higher-order recursive map function instead of direct recursion

```ocaml
# let doubleList list = List.map (fun x -> 2 * x) list;;
val doubleList : int list -> int list = <fun>
# doubleList [2;3;4];;
- : int list = [4; 6; 8]

- Same function, but no explicit recursion
Another common form “folds” an operation over the elements of the structure

```ocaml
# let rec multList list = match list
  with [ ] -> 1
  | x::xs -> x * multList xs;;
val multList : int list -> int = <fun>
# multList [2;4;6];;
- : int = 48
```

Computes \((2 \times (4 \times (6 \times 1)))\)
Folding Recursion : Length Example

# let rec length list = match list
    with [ ] -> 0 (* Nil case *)
    | a :: bs -> 1 + length bs;; (* Cons case *)
val length : 'a list -> int = <fun>

# length [5; 4; 3; 2];;
- : int = 4

- Nil case [ ] is base case, 0 is the base value
- Cons case recurses on component list bs
- What do multList and length have in common?
Forward Recursion

- In Structural Recursion, split input into components and (eventually) recurse

Forward Recursion form of Structural Recursion

- In forward recursion, first call the function recursively on all recursive components, and then build final result from partial results

- Wait until whole structure has been traversed to start building answer
Forward Recursion: Examples

```ocaml
# let rec double_up list =
    match list
    with [ ] -> [ ]
    | (x :: xs) -> (x :: x :: double_up xs);;
val double_up : 'a list -> 'a list = <fun>

# let rec poor_rev list =
    match list
    with [] -> []
    | (x::xs) -> let r = poor_rev xs in r @ [x];;
val poor_rev : 'a list -> 'a list = <fun>
```
Forward Recursion: Examples

```ocaml
# let rec double_up list = 
  match list 
  with [ ] -> [ ] 
  | (x :: xs) -> (x :: x :: double_up xs);
val double_up : 'a list -> 'a list = <fun>

Base Case  Operator  Recursive Call
```

```ocaml
# let rec poor_rev list = 
  match list 
  with [ ] -> [ ] 
  | (x :: xs) -> let r = poor_rev xs in r @ [x];
val poor_rev : 'a list -> 'a list = <fun>
```

Base Case  Operator  Recursive Call
Recursing over lists

```ocaml
# let rec fold_right f list b =
  match list
  with [] -> b
  | (x :: xs) -> f x (fold_right f xs b);
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
# fold_right
  (fun s -> fun () -> print_string s)
  ["hi"; "there"]
  ();
therehi- : unit = ()
```

The Primitive Recursion Fairy
# let rec length list = match list
   with [ ] -> 0 (* Nil case *)
   | a :: bs -> 1 + length bs;; (* Cons case *)
val length : 'a list -> int = <fun>

# let length list =
fold_right (fun a -> fun r -> 1 + r) list 0;;
val length : 'a list -> int = <fun>

# length [5; 4; 3; 2];;
- : int = 4
- multList folds to the right
- Same as:

```ocaml
# let multList list =
    List.fold_right
    (fun x -> fun p -> x * p)
    list 1;;
val multList : int list -> int = <fun>
# multList [2;4;6];;
- : int = 48
```
Forward Recursion: Examples

```ocaml
# let rec double_up list = 
  match list 
  with [] -> [] 
  | (x :: xs) -> (x :: x :: double_up xs);

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

Base Case  Operator  Recursive Call

```ocaml
# let double_up = 
  fold_right (fun x -> fun r -> x :: x :: r) list []

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

Operator  Recursive result  Base Case

```ocaml
# double_up ["a";"b"];;
- : string list = ["a"; "a"; "b"; "b"]
```
Encoding Forward Recursion with Fold

```ocaml
# let rec append list1 list2 =

val append : 'a list -> 'a list -> 'a list = <fun>
```
Encoding Forward Recursion with Fold

```ocaml
# let rec append list1 list2 = match list1 with

val append : 'a list -> 'a list -> 'a list = <fun>
```
Encoding Forward Recursion with Fold

```ocaml
# let rec append list1 list2 = match list1 with
[ ] -> list2
val append : 'a list -> 'a list -> 'a list = <fun>
```

```ocaml
# append [1;2;3] [4;5;6];;
- : int list = [1; 2; 3; 4; 5; 6]
```
Encoding Forward Recursion with Fold

```ocaml
# let rec append list1 list2 = match list1 with
[ ] -> list2
| x::xs -> x :: append xs list2;;
val append : 'a list -> 'a list -> 'a list = <fun>
```

Base Case
Encoding Forward Recursion with Fold

```ocaml
# let rec append list1 list2 = match list1 with
    [ ] -> list2 | x::xs ->
val append : 'a list -> 'a list -> 'a list = <fun>
```

Base Case
Encoding Forward Recursion with Fold

```ocaml
# let rec append list1 list2 = match list1 with
  [ ] -> list2 | x::xs -> x :: append xs list2;;
val append : 'a list -> 'a list -> 'a list = <fun>
```

Base Case
Encoding Forward Recursion with Fold

# let rec append list1 list2 = match list1 with
[ ] -> list2 | x::xs -> x :: append xs list2;;
val append : 'a list -> 'a list -> 'a list = <fun>

Base Case  Operation  Recursive Call
Encoding Forward Recursion with Fold

# let rec append list1 list2 = match list1 with
[ ] -> list2 | x::xs -> x :: append xs list2;;
val append : 'a list -> 'a list -> 'a list = <fun>

Base Case  Operation  Recursive Call

# let append list1 list2 =
fold_right (fun x -> fun y -> x :: y) list1 list2;;
val append : 'a list -> 'a list -> 'a list = <fun>
Encoding Forward Recursion with Fold

# let rec append list1 list2 = match list1 with
[ ] -> list2 | x::xs -> x :: append xs list2;;
val append : 'a list -> 'a list -> 'a list = <fun>

Base Case  Operation  Recursive Call

# let append list1 list2 =
  fold_right (fun x -> fun y -> x :: y) list1 list2;;
val append : 'a list -> 'a list -> 'a list = <fun>

# append [1;2;3] [4;5;6];;
- : int list = [1; 2; 3; 4; 5; 6]
Tail Recursion

- A recursive program is tail recursive if all recursive calls are tail calls
- Tail recursive programs may be optimized to be implemented as loops, thus removing the function call overhead for the recursive calls
- Tail recursion generally requires extra “accumulator” arguments to pass partial results
  - May require an auxiliary function
How can we write length with tail recursion?

```ocaml
let length list =
    let rec length_aux list acc_length =
        match list
        with [ ] -> acc_length
        | (x::xs) ->
            length_aux xs (1 + acc_length)
    in length_aux list 0
```
Tail Recursion - Example

```ocaml
# let rec rev_aux list revlist =
  match list with [ ] -> revlist
  | x :: xs -> rev_aux xs (x::revlist);
val rev_aux : 'a list -> 'a list -> 'a list = <fun>

# let rev list = rev_aux list [ ];;
val rev : 'a list -> 'a list = <fun>
```

- What is its running time?
Comparison

- poor_rev [1;2;3] =
- (poor_rev [2;3]) @ [1] =
- (((poor_rev [3]) @ [2]) @ [1] =
- (((poor_rev [3]) @ [2]) @ [1]) =
- ([3] @ [2]) @ [1] =
- (3 :: ([ ] @ [2])) @ [1] =
- [3;2] @ [1] =
- 3 :: ([2] @ [1]) =
- 3 :: (2 :: ([ ] @ [1])) = [3;2;1]
Comparison

- \text{rev} \ [1;2;3] =
- \text{rev\_aux} \ [1;2;3] \ [\ ] =
- \text{rev\_aux} \ [2;3] \ [1] =
- \text{rev\_aux} \ [3] \ [2;1] =
- \text{rev\_aux} \ [\ ] \ [3;2;1] = [3;2;1]
Iterating over lists

```ocaml
# let rec fold_left f a list =
  match list
  with [] -> a
  | (x :: xs) -> fold_left f (f a x) xs;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>

# fold_left
  (fun () -> print_string)
  ()
  ['hi'; 'there'];;
hithere: unit = ()
```
Folding - Tail Recursion

- # let rev list =
-    fold_left
-    (fun l -> fun x -> x :: l) //comb op
-    []       //accumulator cell
-    list
# let rec fold_left f a list = match list
  with [] -> a | (x :: xs) -> fold_left f (f a x) xs;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
fold_left f a [x₁; x₂;...;xₙ] = f(...(f (f a x₁) x₂)...xₙ)

# let rec fold_right f list b = match list
  with [ ] -> b | (x :: xs) -> f x (fold_right f xs b);;
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
fold_right f [x₁; x₂;...;xₙ] b = f x₁(f x₂(...(f xₙ b)...))
Folding

- Can replace recursion by `fold_right` in any forward primitive recursive definition
  - Primitive recursive means it only recurses on immediate subcomponents of recursive data structure
- Can replace recursion by `fold_left` in any tail primitive recursive definition
How long will it take?

- Remember the big-O notation from CS 225 and CS 374
- Question: given input of size $n$, how long to generate output?
- Express output time in terms of input size, omit constants and take biggest power
How long will it take?

Common big-O times:
- Constant time $O(1)$
  - input size doesn’t matter
- Linear time $O(n)$
  - double input $\Rightarrow$ double time
- Quadratic time $O(n^2)$
  - double input $\Rightarrow$ quadruple time
- Exponential time $O(2^n)$
  - increment input $\Rightarrow$ double time
Linear Time

- Expect most list operations to take linear time $O(n)$
- Each step of the recursion can be done in constant time
- Each step makes only one recursive call
- List example: `multList, append`
- Integer example: `factorial`
Quadratic Time

- Each step of the recursion takes time proportional to input.
- Each step of the recursion makes only one recursive call.
- List example:

```ocaml
# let rec poor_rev list = match list with [] -> [] | (x::xs) -> poor_rev xs @ [x];;
val poor_rev : 'a list -> 'a list = <fun>
```
Exponential running time

- Poor worst-case running times on input of any size
- Each step of recursion takes constant time
- Each recursion makes two recursive calls
- Easy to write naïve code that is exponential for functions that can be linear
Exponential running time

```ocaml
# let rec slow n =
    if n <= 1
    then 1
    else 1 + slow (n-1) + slow(n-2);

val slow : int -> int = <fun>

# List.map slow [1;2;3;4;5;6;7;8;9];;
- : int list = [1; 3; 5; 9; 15; 25; 41; 67; 109]
```
An Important Optimization

<table>
<thead>
<tr>
<th>Normal call</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
</tr>
<tr>
<td>g</td>
</tr>
<tr>
<td>f</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

- When a function call is made, the return address needs to be saved to the stack so we know to where to return when the call is finished.

- What if $f$ calls $g$ and $g$ calls $h$, but calling $h$ is the last thing $g$ does (a *tail call*?)
An Important Optimization

When a function call is made, the return address needs to be saved to the stack so we know to where to return when the call is finished.

What if $f$ calls $g$ and $g$ calls $h$, but calling $h$ is the last thing $g$ does (a tail call)?

Then $h$ can return directly to $f$ instead of $g$. 

Tail call

$h$

$f$

$\ldots$