Programming Languages and Compilers (CS 421)

Talia Ringer (they/them)

4218 SC, UIUC

https://courses.grainger.illinois.edu/cs421/fa2023/

Based heavily on slides by Elsa Gunter, which were based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Objectives for Today

- On Thursday, we saw an introduction to **recursion** and **pattern matching** in OCaml.
- We also saw how to **evaluate expressions**.
- Today, we will take a much more in depth look at **pattern matching** and **recursion**, defining functions over the **list** datatype.
- We will also preview some common **higher-order functions** over lists.

Objectives for Today

- On Thursday, we saw an introduction to **recursion** and **pattern matching** in OCaml.
- We also saw how to **evaluate expressions**.
- Today, we will take a much more in depth look at **pattern matching** and **recursion**, defining functions over the **list** datatype.
- We will also preview some common **higher-order functions** over lists.

Questions from last time?

Lists in OCaml

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 :: [ ]

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 :: [ ]

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 :: [ ]

Lists

```
# 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]
```

Lists

```
# 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]
```

Lists

```
# 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]
```

Lists

```
# 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]
```

Lists are Homogeneous

```
# 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
```

Question

- 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"]]

Question

- 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

```
# 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; 1]
# let silly = double_up ["hi"; "there"];;
val silly : string list = ["hi"; "hi"; "there"; "there"]
```

Functions Over Lists

```
# 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; 1]
# let silly = double_up ["hi"; "there"];;
val silly : string list = ["hi"; "hi"; "there"; "there"]
```

Functions Over Lists

```
# 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; 1]
# let silly = double_up ["hi"; "there"];;
val silly : string list = ["hi"; "hi"; "there"; "there"]
```

Functions Over Lists

```
# 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

- Lists are an example of a **recursive datatype**
- **Functions** on recursive datatypes 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?

```
let rec length list =
```

Question: Length of list

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

```
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
```

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
     | [] ->
     | (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) ->
```

Question: Length of list

- Problem: write code for the length of the list
  - What result do we give when list is **not empty**?

```
let rec length list =
   match list with
    | [] -> 0
    | (a :: bs) ->
```

Question: Length of list

- Problem: write code for the length of the list
  - What result do we give when list is **not empty**?

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

Question: Length of list

- Problem: write code for the length of the list

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

Question: Length of list

- Problem: write code for the length of the list

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

- Nil case [] is **base case**
- Cons case **recurses** on component list bs

Same Length

- How can we efficiently answer if two lists have the same length?

Same Length

- How can we efficiently answer if two lists have the same length?

```
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)
```

Same Length

- How can we efficiently answer if two lists have the same length?

```
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)
```

Same Length

- How can we efficiently answer if two lists have the same length?

```
let rec same_length list1 list2 =
    match (list1, list2) with
    | [], [] -> true
    | x::xs, y::ys -> same_length xs ys
    | _ -> false
```

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**

```
let rec doubleList list =
  match list with
  | [] ->
  | x :: 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**

```
let rec doubleList list =
  match list with
  | [] -> []
  | x :: 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**

```
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**

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

Mapping over Lists

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

Map

Higher-Order Functions: Map

- Write a function that takes a **list** of 'a and returns a **list of the same length**, where **each element** has been **transformed by f**

```
let rec map f list =
  match list with
  | [] -> []
  | x :: xs -> (f x) :: map f xs
```

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

Map

Higher-Order Functions: Map

- Write a function that takes a **list** of 'a and returns a **list of the same length**, where **each element** has been **transformed by f**

```
let rec map f list =
  match list with
  | [] -> []
  | x :: xs -> (f x) :: map f xs
```

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

Higher-Order Functions: Map

- Write a function that takes a **list** of 'a and returns a **list of the same length**, where **each element** has been **transformed by f**

```
let rec map f list =
  match list with
  | [] -> []
  | x :: xs -> (f x) :: map f xs
```

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

Map

Higher-Order Functions: Map

- Write a function that takes a **list** of 'a and returns a **list of the same length**, where **each element** has been **transformed by f**

```
let rec map f list =
  match list with
  | [] -> []
  | x :: xs -> (f x) :: map f xs
```

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

Higher-Order Functions: Map

- 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 doubleList list =
  List.map ? list
```

Map

Higher-Order Functions: Map

- 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 doubleList list =
  List.map (fun x -> 2 * x) list
```

Map

Higher-Order Functions: Map

- Write a function that takes a **list** of pairs and returns a **list of the first element of every pair**

```
let fstAll list =
  List.map (fun (a, b) -> a) list
```

Higher-Order Functions: Map

- Write a function that takes a **list** of pairs and returns a **list of the first element of every pair**

```
let fstAll list =
  List.map fst list
```

Map

Higher-Order Functions: Map

- Write a function that takes a **list** of 'a and returns a **list of the same length**, where **each element** has been **transformed by f**

```
let rec map f list =
  match list with
  | [] -> []
  | x :: xs -> (f x) :: map f xs
```

Captures common recursive pattern, so fstAll, doubleList, etc. need not be explicitly recursive.

Map

Questions so far?

Folding over Lists

Higher-Order Functions: Fold

- Write a function that **"folds"** an **operation** over the **elements** of the **structure**.

Higher-Order Functions: Fold

- Write a function that **“folds”** an **operation** over the **elements** of the **list**.

Fold

Higher-Order Functions: Fold

- Write a function that **“folds” multiplication** over the **elements** of the **list of ints**.

Folding Recursion

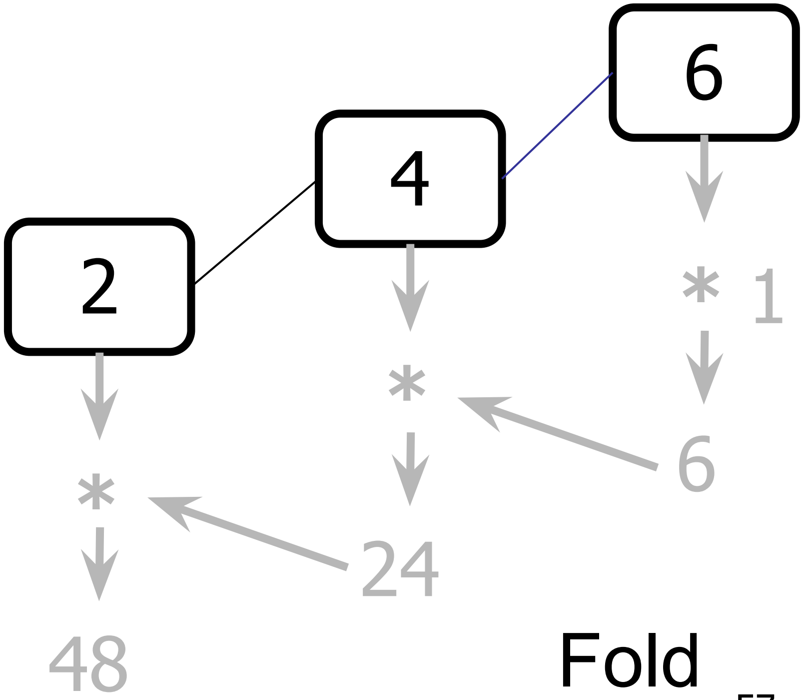
- Write a function that computes the **product** of all of the **elements** of the input **list**.

Fold

Folding Recursion

- Write a function that computes the **product** of all of the **elements** of the input **list**.

multList [2; 4; 6] =
2 * multList [4; 6] =
2 * (4 * multList 6) =
2 * (4 * (6 * 1)) =
48

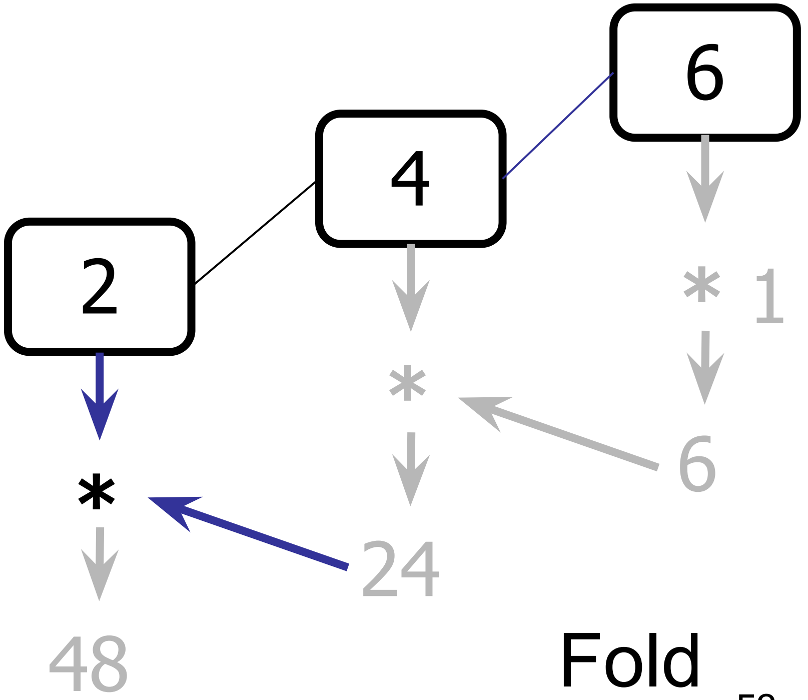


Fold

Folding Recursion

- Write a function that computes the **product** of all of the **elements** of the input **list**.

multList [2; 4; 6] =
2 * multList [4; 6] =
2 * (4 * multList 6) =
2 * (4 * (6 * 1)) =
48

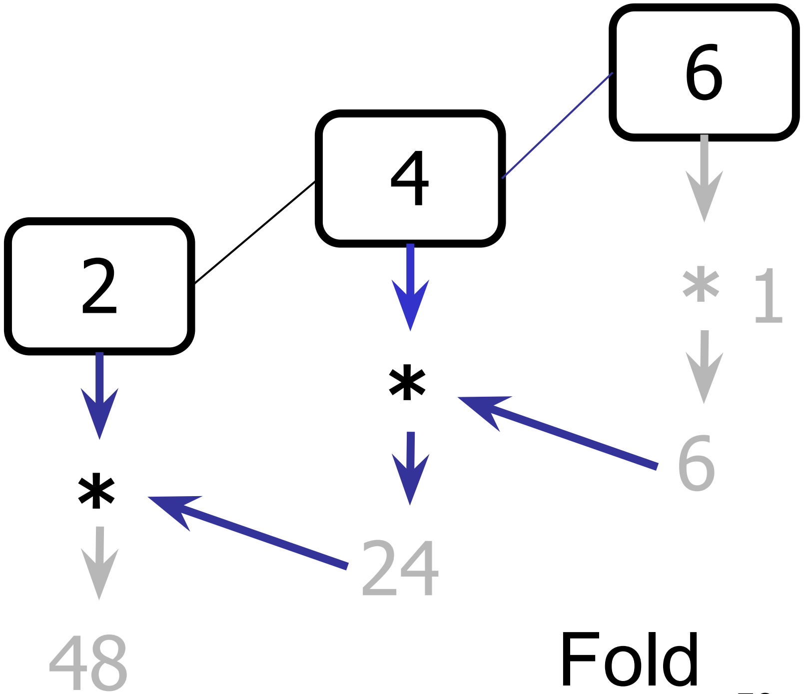


Fold

Folding Recursion

- Write a function that computes the **product** of all of the **elements** of the input **list**.

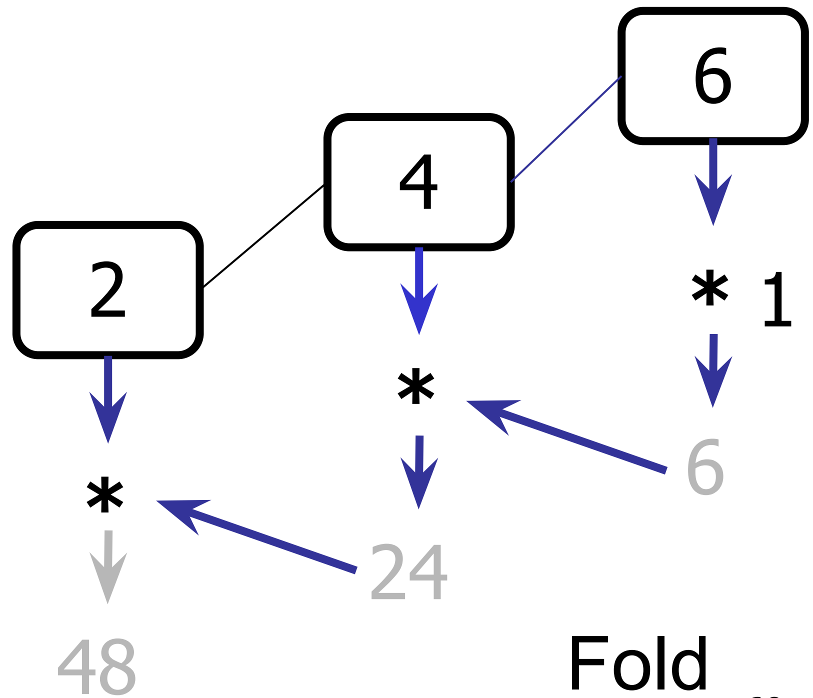
multList [2; 4; 6] =
2 * multList [4; 6] =
2 * (4 * multList 6) =
2 * (4 * (6 * 1)) =
48



Folding Recursion

- Write a function that computes the **product** of all of the **elements** of the input **list**.

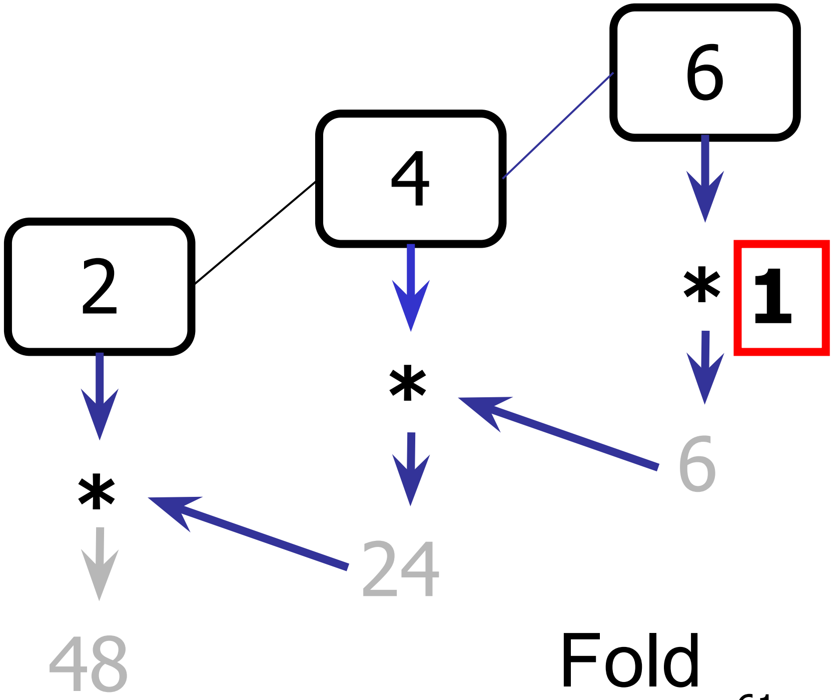
multList [2; 4; 6] =
2 * multList [4; 6] =
2 * (4 * multList 6) =
2 * (4 * (6 * 1)) =
48



Folding Recursion

- Write a function that computes the **product** of all of the **elements** of the input **list**.

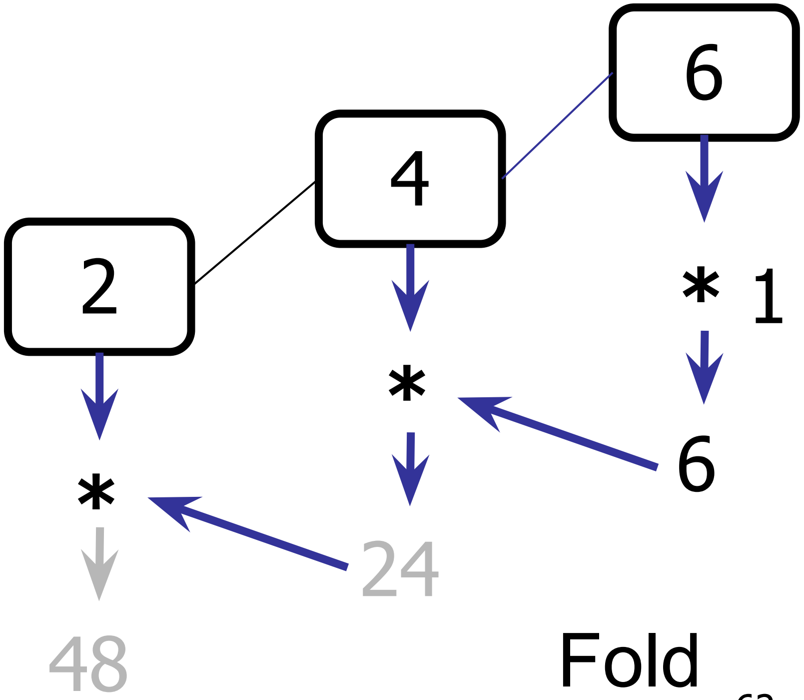
multList [2; 4; 6] =
2 * multList [4; 6] =
2 * (4 * multList 6) =
2 * (4 * (6 * 1)) =
48



Folding Recursion

- Write a function that computes the **product** of all of the **elements** of the input **list**.

multList [2; 4; 6] =
2 * multList [4; 6] =
2 * (4 * multList 6) =
2 * (4 * (6 * 1)) =
48

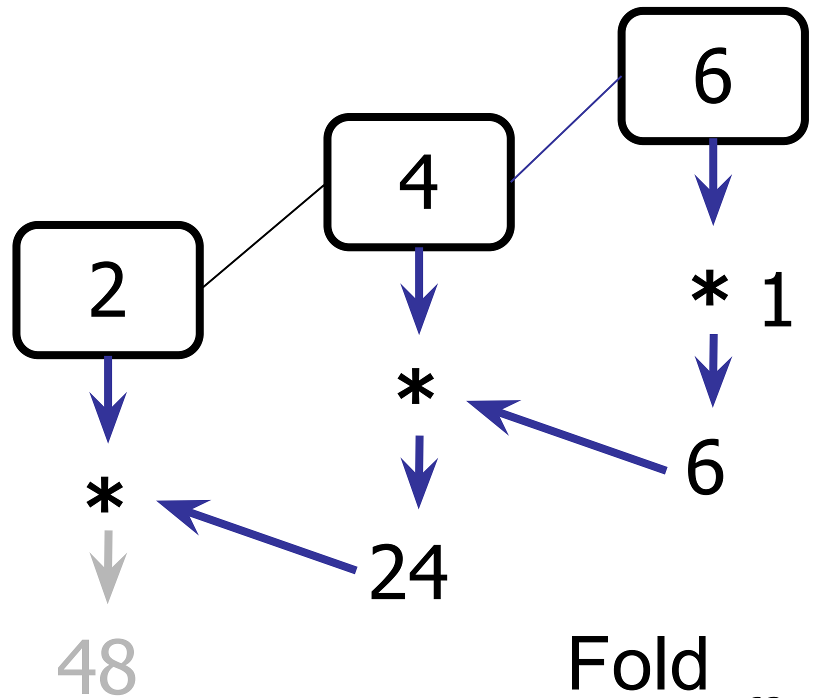


Fold

Folding Recursion

- Write a function that computes the **product** of all of the **elements** of the input **list**.

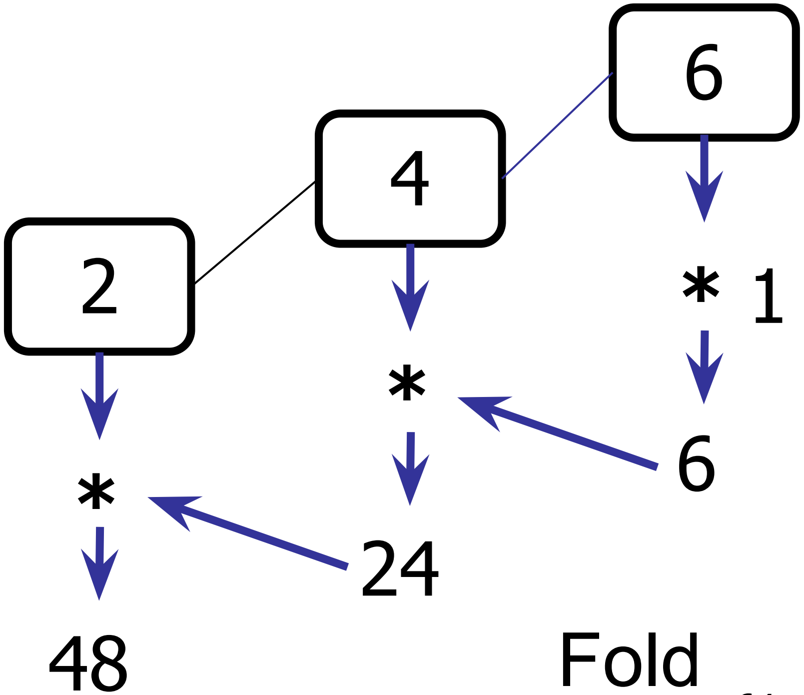
multList [2; 4; 6] =
2 * multList [4; 6] =
2 * (4 * multList 6) =
2 * (4 * (6 * 1)) =
48



Folding Recursion

- Write a function that computes the **product** of all of the **elements** of the input **list**.

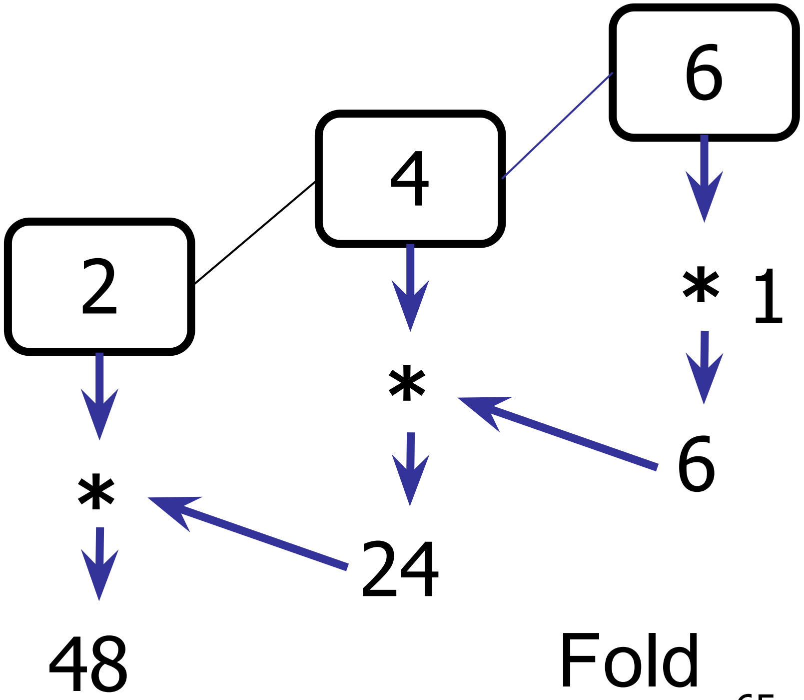
multList [2; 4; 6] =
2 * multList [4; 6] =
2 * (4 * multList 6) =
2 * (4 * (6 * 1)) =
48



Folding Recursion

- Write a function that computes the **product** of all of the **elements** of the input **list**.

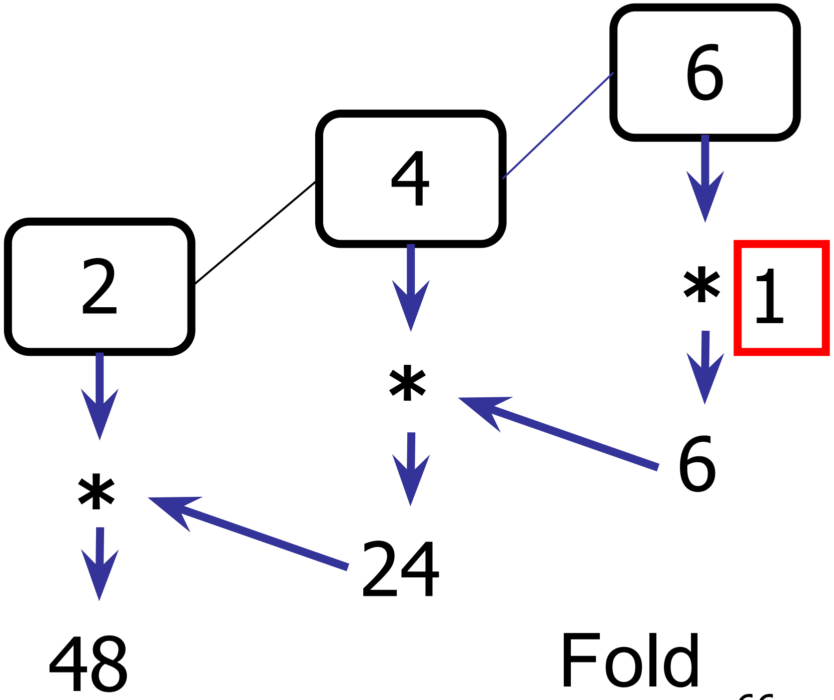
```
let rec multList list =
  match list with
  | [ ] -> 1
  | x :: xs ->
    x * multList xs;;
```



Folding Recursion

- Write a function that computes the **product** of all of the **elements** of the input **list**.

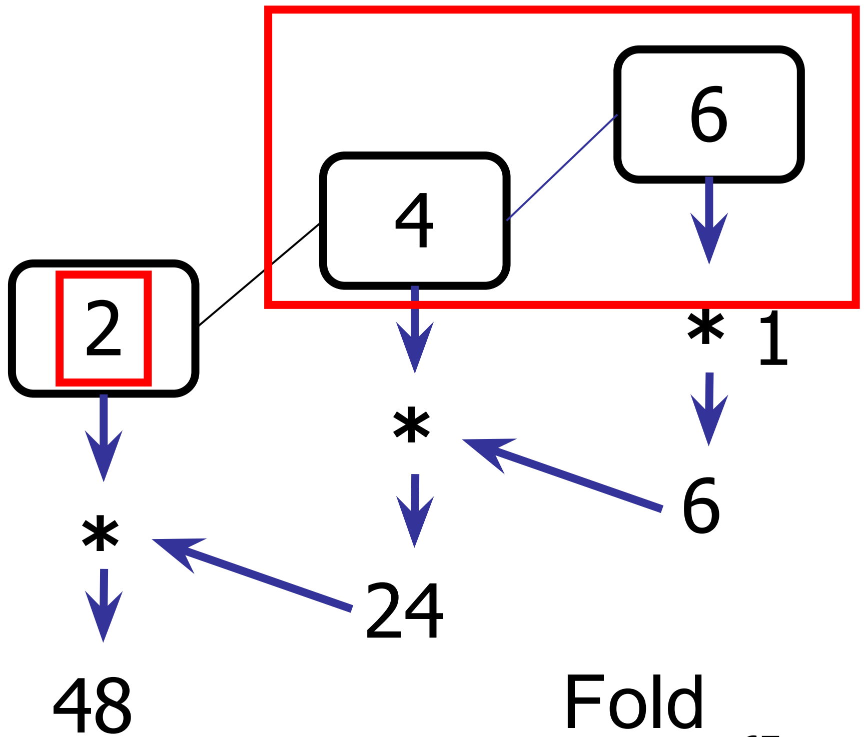
```
let rec multList list =
  match list with
  | [ ] -> 1
  | x :: xs ->
    x * multList xs;;
```



Folding Recursion

- Write a function that computes the **product** of all of the **elements** of the input **list**.

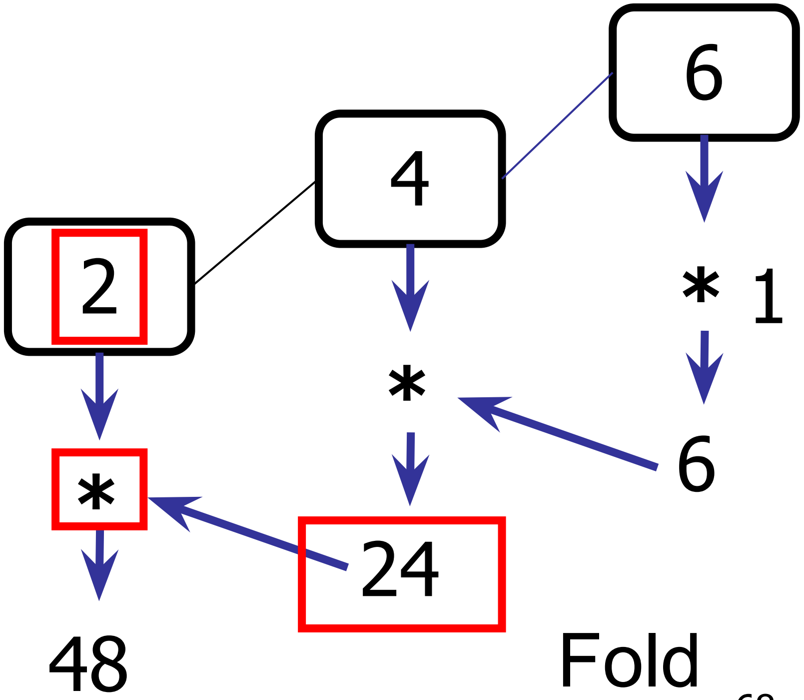
```
let rec multList list =
  match list with
  | [ ] -> 1
  | x :: xs ->
    x * multList xs;;
```



Folding Recursion

- Write a function that computes the **product** of all of the **elements** of the input **list**.

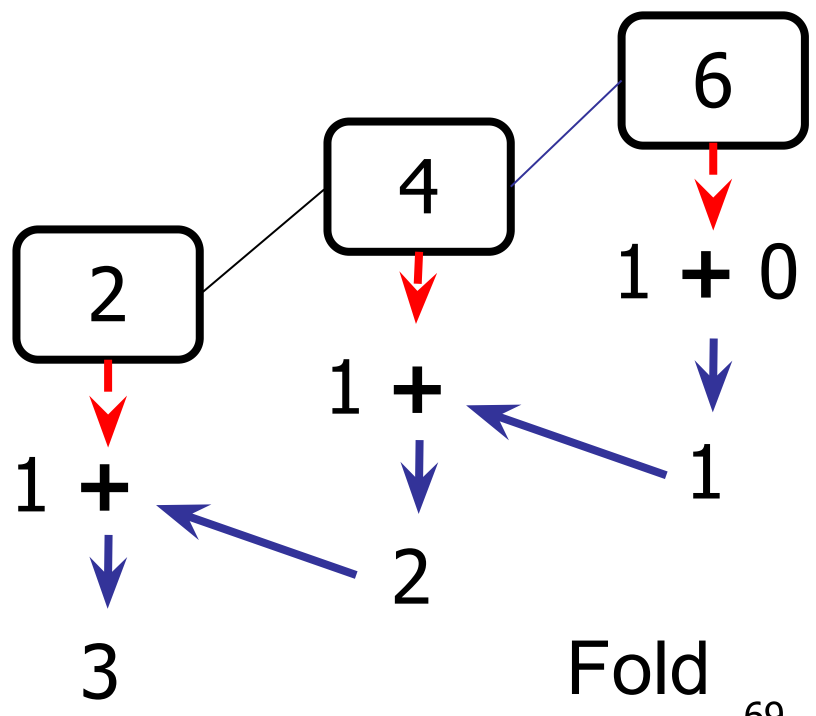
```
let rec multList list =
  match list with
  | [ ] -> 1
  | x :: xs ->
     x * multList xs;;
```



Folding Recursion

- Write a function that computes the **length** of the input **list**.

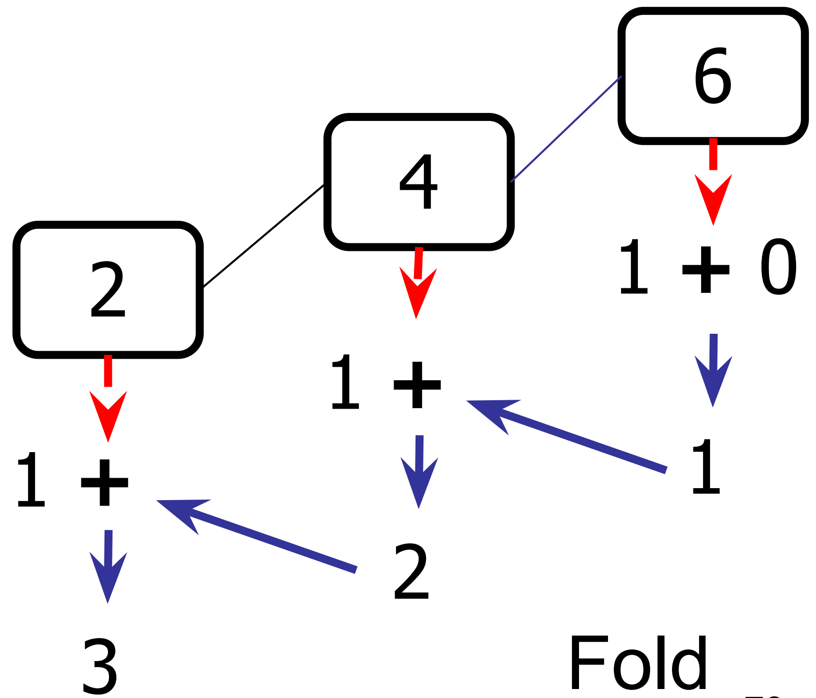
```
let rec length list =
  match list with
  | [ ] -> 0
  | a :: bs ->
     1 + length bs;;
```



Folding Recursion

- Write a function that computes the **length** of the input **list**.

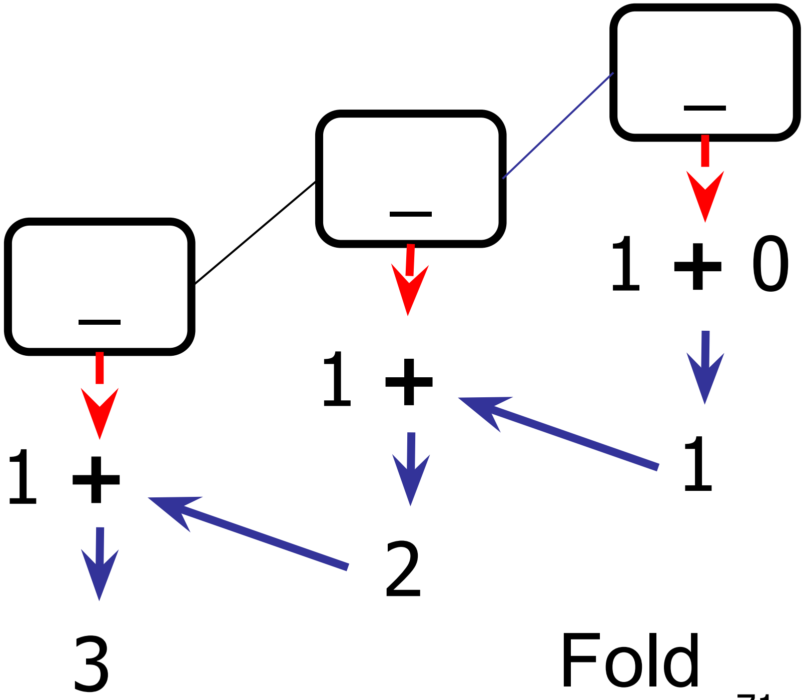
```
let rec length list =
  match list with
  | [ ] -> 0
  | a :: bs ->
    1 + length bs;;
```



Folding Recursion

- Write a function that computes the **length** of the input **list**.

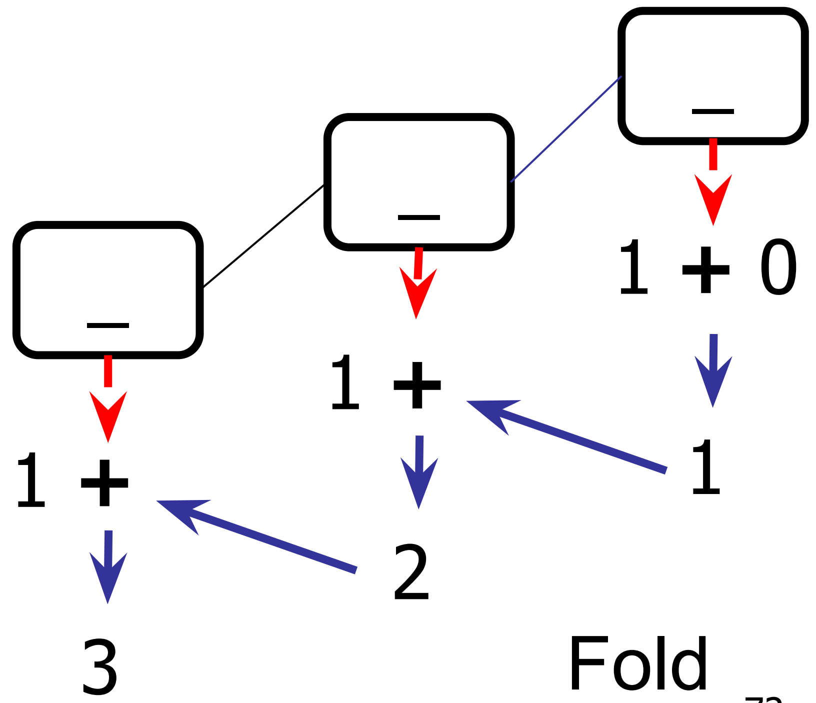
```
let rec length list =
  match list with
  | [ ] -> 0
  | _ :: bs ->
     1 + length bs;;
```



Folding Recursion

- Write a function that computes the **length** of the input **list**.

```
let rec length list =
  match list with
  | [ ] -> 0
  | _ :: bs ->
    1 + length bs;;
```



Generic List Fold Next Class

Preview: Kinds of Recursion

Forward Recursion

- What do multList and length have in common? Both use **forward recursion**
- Forward Recursion form of **Structural Recursion** (recurse on substructures)
- In **forward** recursion, **first call the function recursively** on all recursive components, and then **build final result**
- **Wait** until whole structure has been traversed **to start building** answer

Forward Recursion

- What do multList and length have in common? Both use **forward recursion**
- Forward Recursion form of **Structural Recursion** (recurse on substructures)
- In **forward** recursion, **first call the function recursively** on all recursive components, and then **build final result**
- **Wait** until whole structure has been traversed **to start building** answer

Forward Recursion: Examples

```
# 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

```
# 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

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

base case / id

```
# let rec poor_rev list =
  match list with
  | [] -> []
  | (x :: xs) -> let r = poor_rev xs in r @ [x];;
```

base case / id

Forward Recursion: Examples

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

base case / id

operator

```
# let rec poor_rev list =
  match list with
  | [] -> []
  | (x :: xs) -> let r = poor_rev xs in r @ [x];;
```

operator

base case / id

Forward Recursion: Examples

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

base case / id

operator

recursion (first)

```
# let rec poor_rev list =
  match list with
  | [] -> []
  | (x :: xs) -> let r = poor_rev xs in r @ [x];;
```

recursion (first)

operator

base case / id

Questions?

Takeaways

- Lists are **recursive datatypes**
- **Functions** over recursive datatypes like lists tend to be **recursive**
- We saw a particular kind of recursion called **forward recursion** in which the function is called recursively *before* building the final results.
- There are some **common paradigms** for recursion over lists (and other datatypes) that are captured by **higher-order functions**:
  - **Mapping** a function over every element of a list
  - **Folding** an operation over elements of a list

Next Class:
Forward vs. Tail Recursion,
Folding Left vs. Folding Right

Reminders

- **WA2** due **Thursday**
- **Quiz 2** on **MP3** next **Tuesday**
- All deadlines can be found on **course website**
- Use **office hours** and **class forums** for help
- **Please thank Elsa for covering <3**

Next Class

TODO takeaways, next time, assignments