Review: higher-order functions

- Functions are values
- Can use them **anywhere** we use values
  - Arguments, results, parts of tuples, bound to variables...
- Functions can **take** functions as arguments
- Functions can **return** functions as results

Review: anonymous functions

(aka *function expressions*)

- **Syntax: `fun x -> e`**

- **Type checking:**
  - Conclude that `fun x -> e : t1 -> t2` if `e : t2` under assumption `x : t1`

- **Evaluation:**
  - A function is already a value

Lambda

- Anonymous functions a.k.a. *lambda expressions:* `λx . e`
- The lambda means "what follows is an anonymous function"
  - x is its argument
  - e is its body
  - Just like `fun x -> e`, but slightly different syntax
- Standard feature of any functional language (ML, Haskell, Scheme, …)

- You'll see "lambda" show up in many places in PL, e.g.:
  - PHP: http://www.php.net/manual/en/function.create-function.php
  - A popular PL blog: http://lambda-the-ultimate.org/
  - Lambda style: https://www.youtube.com/watch?v=Ci48kqp11F8

Map

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

`map : ('a -> 'b) -> 'a list -> 'b list`

Map is HUGE:

- You use it **all the time** once you know it
- Exists in standard library as **`List.map`**, but the idea can be used in any data structure (trees, stacks, queues…)

Question

What is value of lst after this code?

```
let is_even x = (x mod 2 = 0)
let lst = map is_even [1;2;3;4]
```

A. [1;2;3;4]
B. [2;4]
C. [false; true; false; true]
D. false

Question

What is value of lst after this code?

```
let is_even x = (x mod 2 = 0)
let lst = map is_even [1;2;3;4]
```

A. [1;2;3;4]
B. [2;4]
C. **[false; true; false; true]**
D. false

Question

What is value of lst after this code?

```
let is_even x = (x mod 2 = 0)
let lst = filter is_even [1;2;3;4]
```

A. [1;2;3;4]
B. [2;4]
C. [false; true; false; true]
D. false

Filter

```
let rec filter f = function
    | [] -> []
    | x::xs -> if f x
                 then x::(filter f xs)
                 else filter f xs
```

```
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
```

Question

What is value of lst after this code?

```
let is_even x = (x mod 2 = 0)
let lst = filter is_even [1;2;3;4]
```

A. [1;2;3;4]
B. [2;4]
C. [false; true; false; true]
D. false

Iterators

- Map and filter are *iterators*
  - Not built-in to the language, an idiom

- Benefit of iterators: separate recursive traversal from data processing
  - Can reuse same traversal for different data processing
  - Can reuse same data processing for different data structures
  - leads to modular, maintainable, beautiful code!

- So far: iterators that change or omit data
  - what about combining data?
  - e.g., sum all elements of list

Fold v1.0

Idea: *stick an operator between every element of list*

folding **`[1;2;3]`** with **`(+)`**

becomes

`1+2+3`

`==>`

`6`

Fold v2.0

Idea: *stick an operator between every element of list*
But list could have 1 element, so need an initial value

folding **[1]** with **0** and **(+)**

becomes

0+1

==>

1

Fold v2.0

Idea: *stick an operator between every element of list*
But list could have 1 element, so need an initial value

folding **`[1;2;3]`** with **0** and **`(+)`**

becomes

0+1+2+3

==>

6

Fold v2.0

Idea: *stick an operator between every element of list*

But list could have 1 element, so need an initial value

Or list could be empty; just return initial value

folding **[ ]** with **0** and **(+)**

becomes

0

Question #4

What should the result of folding `[1;2;3;4]` with `1` and `( * )` be?

A. 1
B. 24
C. 10
D. 0

Question #4

What should the result of folding `[1;2;3;4]` with **1** and `( * )` be?

A. 1

B. 24

C. 10

D. 0

Fold v3.0

Idea: *stick an operator between every element of list*
But list could have 1 element, so need an initial value
Or list could be empty; just return initial value

Implementation detail: iterate left-to-right or right-to-left?

folding **`[1;2;3]`** with **0** and **`(+)`**

left to right becomes: ((0+1)+2)+3

right to left becomes: 1+(2+(3+0))

Both evaluate to 6; does it matter?

Yes: not all operators are associative, e.g. subtraction, division, exponentiation, …

Fold v4.0

- (+) *accumulated* a result of the same type as list itself
- What about operators that change the type?
  - e.g., `::` has type `'a -> 'a list -> 'a list`

    folding from the right `[1;2;3]` with `[]` and `::`

    should produce

    `1::(2::(3::[])) = [1;2;3]`
- So the operator needs to accept
  - the accumulated result so far, and
  - the next element of the list

    …which may have different types!

Fold for real

Two versions in OCaml library:

```
List.fold_left
: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

List.fold_right
: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

Fold for real

Two versions in OCaml library:

```
List.fold_left
: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

List.fold_right
: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

Operator

Fold for real

Two versions in OCaml library:

```
List.fold_left
: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

```
List.fold_right
: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

Input list

Fold for real

Two versions in OCaml library:

```
List.fold_left
: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

List.fold_right
: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

Initial value of accumulator

Fold for real

Two versions in OCaml library:

```
List.fold_left
: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

```
List.fold_right
: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

Final value of accumulator

fold_left

```
let rec fold_left f acc xs =
   match xs with
      []       -> acc
    | x::xs’  -> fold_left f (f acc x) xs’
```

Accumulates an answer by

- repeatedly applying **f** to “answer so far”,
- starting with initial value **acc**,
- folding “from the left”

fold_left f acc [a;b;c]

computes

f (f (f acc a) b) c

fold_right

```
let rec fold_right f xs acc =
   match xs with
     []      -> acc
   | x::xs'  -> f x (fold_right f xs' acc)
```

Accumulates an answer by

- repeatedly applying **f** to "answer so far",
- starting with initial value **acc**,
- folding "from the right"

fold_right f [a;b;c] acc

computes

f a (f b (f c acc))

Behold the HUGE power of fold

Implement so many other functions with fold!

```
let rev xs = fold_left (fun xs x -> x::xs) [] xs
let length xs = fold_left (fun a _ -> a+1) 0 xs
let map f xs = fold_right
  (fun x a -> (f x)::a) xs []
let filter f xs = fold_right
  (fun x a -> if f x then x::a else a) xs []
```

Beware the efficiency of fold

- **`fold_left`** is tail recursive, **`fold_right`** is not
- **`fold_right`** might make it easier to express computation (e.g., **`map`**)
- Rule of thumb: for lists with > 10,000 elements, use tail recursion

MapReduce

- Fold has many synonyms/cousins in various functional languages, including **`scan`** and **`reduce`**
- Google organizes large-scale data-parallel computations with MapReduce
  - open source implementation by Apache called Hadoop

"[Google's MapReduce] abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a map operation to each logical record in our input in order to compute a set of intermediate key/value pairs, and then applying a reduce operation to all the values that shared the same key in order to combine the derived data appropriately."
[Dean and Ghemawat, 2008]