CSCI 334:
Principles of Programming Languages

Lecture 5: Higher Order Functions

Instructor: Dan Barowy

Williams

Topics

More pattern matching

Option vs exceptions

Higher order functions

Your to-dos

1. Lab 2, **due Sunday 9/25 by 10pm** (partner lab).

More Pattern Matching

Activity: Pattern matching on tuples

Write a **function** that computes the **Cartesian product** of two sets, represented by lists:

$$A \times B = \{ (a,b) \mid a \in A \text{ and } b \in B \}$$

Hint: I find it helpful to think about **base cases** first.

```
let rec cartesianProduct xs ys =
  match xs,ys with
  | []    ,_   -> []
  | _     ,[] -> []
  | x::xs',_   ->
    let zs = ys |> List.map (fun y -> (x,y))
    zs @ cartesianProduct xs' ys
```

Avoiding errors with patterns

- Another example: handling errors.
- F# has exceptions (like Java)
- But an alternative, easy way to handle many errors is to use the option type:

```
type option<'a> =
| None
| Some of 'a
```

Avoiding errors with patterns

```
let divide quot div =
  match div with
  | 0 -> None
  | _ -> Some (float quot/float div)
```

Avoiding errors with patterns

```
> divide 6 7;;
val it : float option = Some 0.8571428571

> divide 6 0;;
val it : float option = None

>
```

option type

- Why `option`?
- `option` is a data type;

  not handling errors is a static type error!

Exceptions

This code is problematic

```
let divide quot div = quot/div
```

(but only because of integer division)

Exception handling

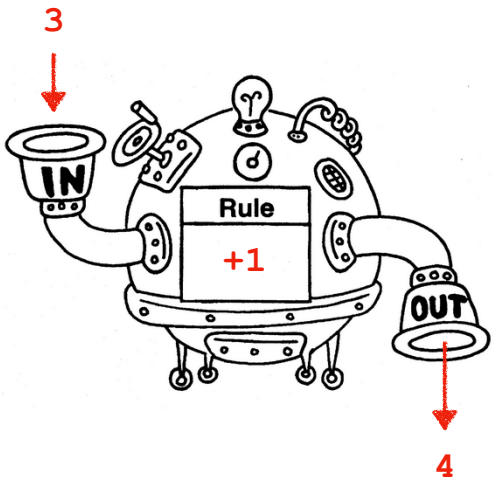
```
let divide quot div = quot/div

[<EntryPoint>]
let main args =
    let quot = int args[0]
    let divisor = int args[1]
    try
        let dividend = divide quot divisor
        printfn "%d" dividend
        0
    with
    | :? System.DivideByZeroException ->
        printfn "No way, dude!"
        1
```

Higher order functions

Three amazing functional concepts

- First-class functions
- Higher-order functions
  - map
  - fold

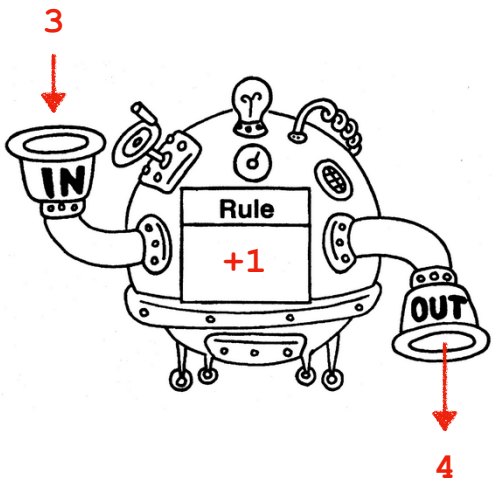
a function



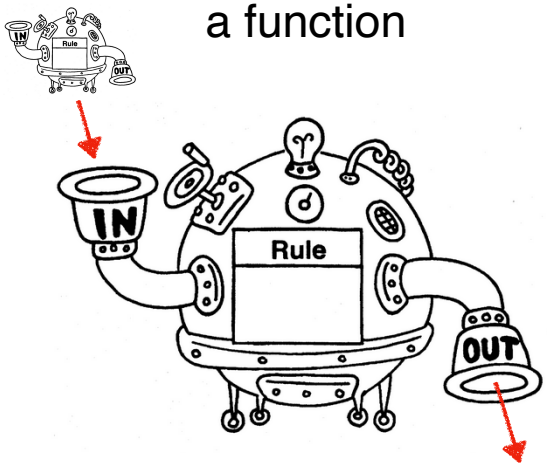
“first class” function

Function definitions are **values** in a functional programming language

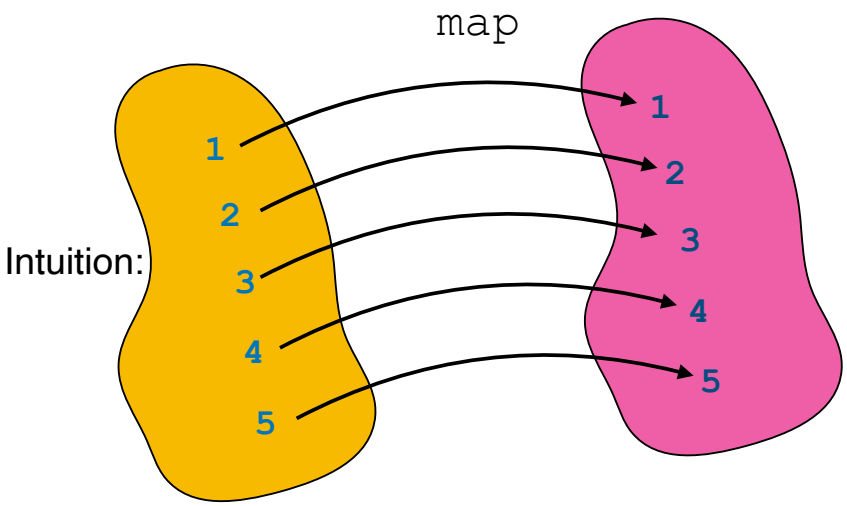
a function



a function



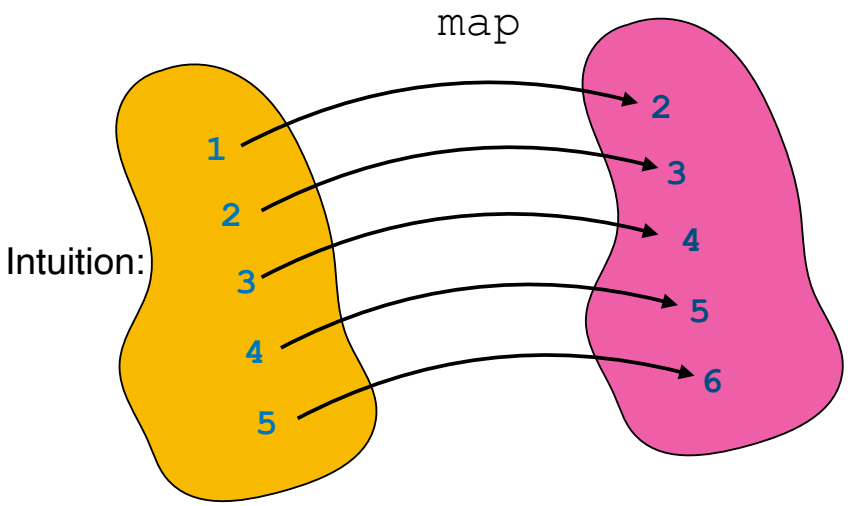
map



Like a `for` loop, but without mutable variables

```
List.map (fun x -> x + 1) [1;2;3;4;5]
```

map

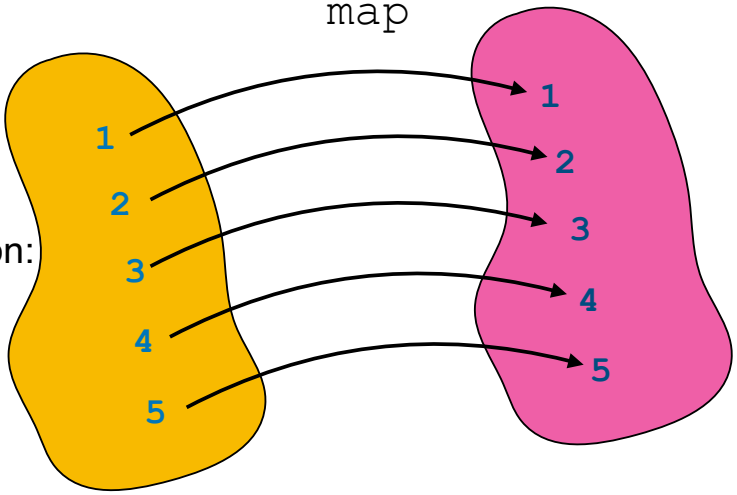


Like a `for` loop, but without mutable variables

```
[1;2;3;4;5] |> List.map (fun x -> x + 1)
```

map

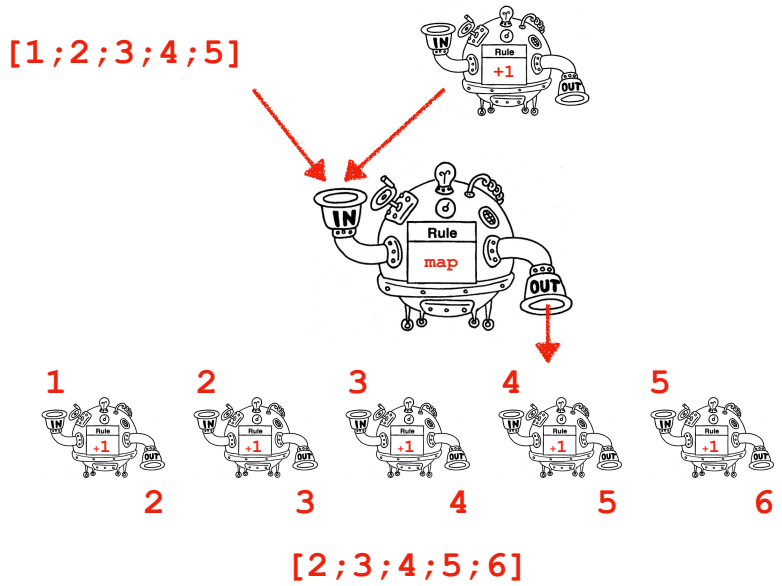
Intuition:



Key observation:

`n` things in, `n` things out

map



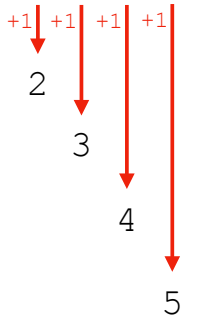
[2;3;4;5;6]

map

Intuition:

map

```
List.map (fun x -> x + 1) [1;2;3;4];
```



[2;3;4;5]

map

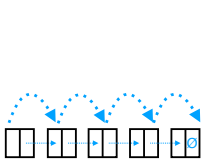
```
[2;8;22;4]
|> List.map (fun x -> x + 1)
|> List.map float
|> List.map (fun x -> x / 3.3)
|> List.sort
```

```
[0.9090909091; 1.515151515; 2.727272727;
6.96969697]
```

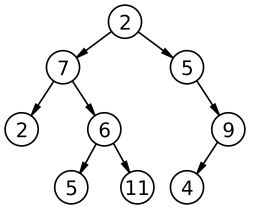
fold

structural recursion → fold it!

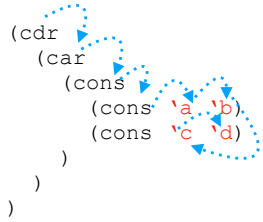
(in a nutshell: any problem that recurses on a subset of input)



list length



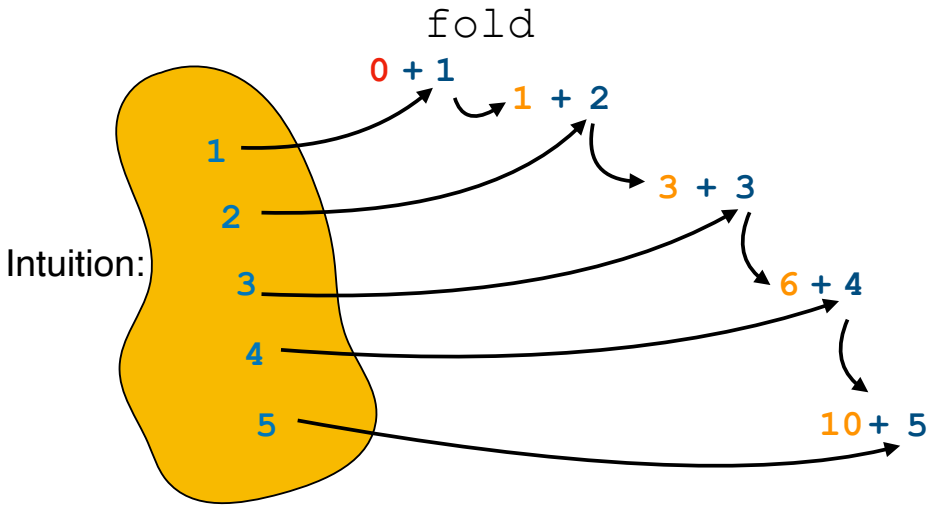
tree height



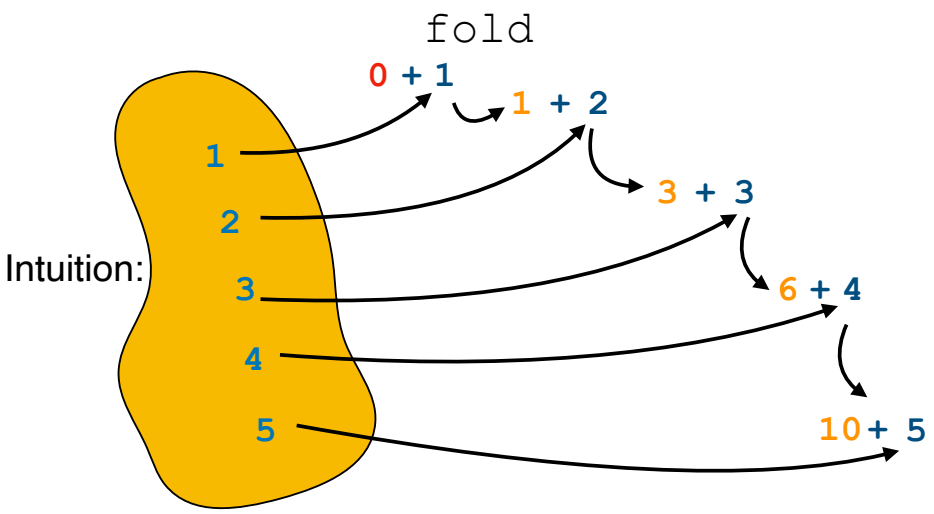
evaluation

fold

Intuition:




```
List.fold (fun acc x -> acc+x) 0 [1;2;3;4;5]
```




```
[1;2;3;4;5] |> List.fold (fun acc x -> acc+x) 0
```

fold left

```
List.fold (fun acc x -> acc+x) 0 [1;2;3;4]
```

```
acc = 0, [1;2;3;4]
acc = 0+1, [2;3;4]
  acc = 1+2, [3;4]
    acc = 3+3, [4]
      acc 6+4, []
returns acc = 10
```

what does this print?

```
List.fold (fun acc x -> acc + x)
  "williams"
  ["2";"3"]
```

fold right

```
List.foldBack
  (fun x acc -> acc+x) [1;2;3;4] 0
```

```
[1;2;3;4], acc = 0
[1;2;3], acc = 0+4
[1;2], acc = 4+3
[1] acc = 7+2
[], acc = 9+1
returns acc = 10
```

what does this print?

```
List.foldBack (fun x acc -> acc + x)
  ["2";"3"]
  "williams"
```

Activity

```
let number_in_month(ds: Date list)(month: int) : int =
```

- Write a function `number_in_month` that takes a list of dates (where a date is `int*int*int` representing year, month, and day) and an `int month` and returns how many dates are in `month`
- Use `List.fold`

fold

```
let number_in_month(ds: Date list)(month: int) : int =
  ds
  |> List.fold (fun acc (_,mm,_) ->
       if month = mm then
         acc + 1
       else
         acc
     ) 0
```

Recap & Next Class

Today:

More pattern matching
Option vs exceptions
Higher order functions

Next class:

PL foundations