1

CSCI 334:
Principles of Programming Languages

Lecture 5: Errors / Higher Order Functions

Instructor: Dan Barowy
Williams

2

Announcements

- CS Colloquium this **Friday, Sept 22 @ 2:35pm in Wege Auditorium (TCL 123)**

Your classmates

What I Did Last Summer, Industry Edition

Short presentations by your fellow CS students about internship experiences in industry. CS Colloquium credit awarded for attendance.

3

Topics

Avoiding errors

Higher order functions

Your to-dos

1. Lab 2, **due Sunday 9/24 by 10pm** (partner lab).
2. Read *Introduction to the Lambda Calculus, Part 1* and *Grammars and Parse Trees* by **next Thursday, 9/28**.
3. Sign up for *What I Did Last Summer*

4

Activity

5

One approach is to create a variable that you can test in a pattern match.

```
let rec get_nth xs n =
  let gt = n > List.length xs
  match xs, n, gt with
  | _,_,true -> failwith "error"
  | [],_,_ -> failwith "error"
  | y::ys,1,_ -> y
  | y::ys,_,_ -> get_nth ys (n - 1)
```

Avoiding errors

Exploding programs is **no fun**.
Get in the habit of **validating input**.

6

I'm going to discuss two approaches.

Avoiding errors using patterns

7

8

Avoiding errors with patterns

- 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
```

9

A function that throws an exception

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

Here's a toy example of a function that can fail (with an exception). Although the failure mode of this function may seem obvious for this example, in general, it is often hard to see which inputs may cause a function to fail, especially if you did not write the function.

A function that throws an exception

```
> divide 6 7;;
val it : int = 0

> divide 6 0;;
System.DivideByZeroException: Attempted to
divide by zero.
…
Stopped due to error
```

10

In case it wasn't clear, here's the function failing. Note that 6/7 = 0 because we're talking about integer division.

Avoiding errors with patterns

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

11

Instead, we can rewrite our function to communicate to a potential user that it might fail. In other words, it is not defined across its entire range. To do that, we use the parametric (i.e., generic) option type. When it fails, it returns None of something. When it succeeds, it returns Some of int.

Avoiding errors with patterns

```
> divide 6 7;;
val it : int option = Some 0

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

>
```

12

Here is the same scenario as before. Observe that there is no runtime exception.

Option type

- Why `option`?
- `option` is a **data type**;
  not handling errors is a **static type error**!
- In other words, the user of our divide function **must handle the error**.

13

So why is this a good idea? In short, it forces the user of the function to acknowledge the failure mode of the program, and to write program logic to handle it. Failing to handle the error is a type error, which means that their program will not compile. Now, we cannot guarantee that the user does the “right thing” with the failure, but at least we can guarantee that they must do something. Moreover, we make the user’s life easier because they do not need to understand the domain of the function deeply— they just need to think of a corrective action.

Option type

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

[<EntryPoint>]
let main args =
  let quot = int args[0]
  let divisor = int args[1]
  let result = divide quot divisor
  match result with
  | Some z -> printfn "Oh good: %d" z
  | None   -> printfn "Bad numbers!"
  0
```

14

Here is a complete example, with a main method. Try it yourself!

Exceptions

15

Of course, F# also has exceptions.

We could have used exception, right?

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

16

This function naturally uses exceptions, since integer division by zero is undefined and throws a floating point exception.

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
```

17

Here's a complete example. Observe that the burden is shifted entirely to the user of the divide function. Also, F# does not force users to handle exceptions, so if they do not actively anticipate errors, they are likely to miss the fact that they need to do this. Still, it is a simple mechanism and can work reliably when the domain is communicated clearly to the user of the function (e.g., through comments).

Option vs Exceptions

- When do I use each one?
  - `option` prevents errors at **compile time**.
  - Exceptions prevent errors at **runtime**.

18

Why might you want to use option vs exceptions? The question comes down to when you want errors in program logic handled. Option ensures that logic errors are handled at compile time. Exception handlers ensure that logic errors are handled at runtime. I have a personal preference for the former because I think coding is hard and that we need all the help that we can get.

Activity solution using option and when

```
let rec get_nth xs n =
  match xs, n with
  | _, _ when n > List.length xs -> None
  | _, _ when n < 1 -> None
  | [], _           -> None
  | y::ys, 1        -> Some y
  | y::ys, _        -> get_nth ys (n - 1)
```

19

BTW, we can use Option and “when” syntax in our patterns to make our solution really pretty.

Higher order functions

20

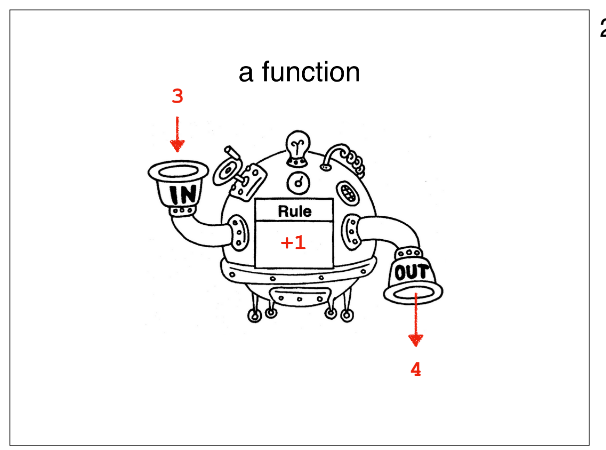
HOFs are one of the most important features of functional languages, and its something that makes them stand apart from conventional languages. HOFs give you great flexibility in how you design programs.

Three amazing functional concepts

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

21

If you learn only three ideas about functional programming this semester, I hope it is these three ideas. First-class functions, map, and fold. Nearly any program that uses loops can instead be expressed using these three ideas.



22

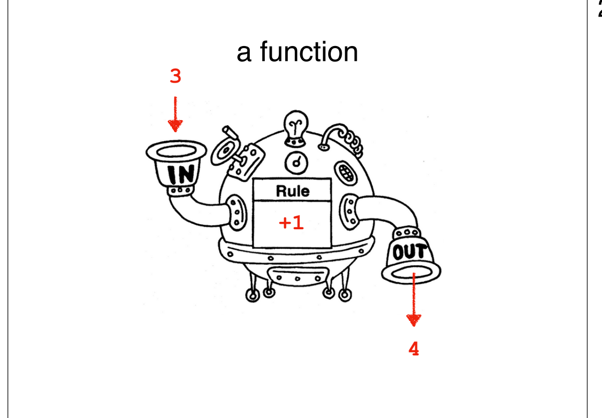
I want you to think of functions simply, in their mathematical sense. A function is a machine that takes an input and returns an output. Any other kind of “function” is not a function in a true sense. For example, a machine that does something off on the side is not a true function; it is more properly called a “procedure.”

“first class” function

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

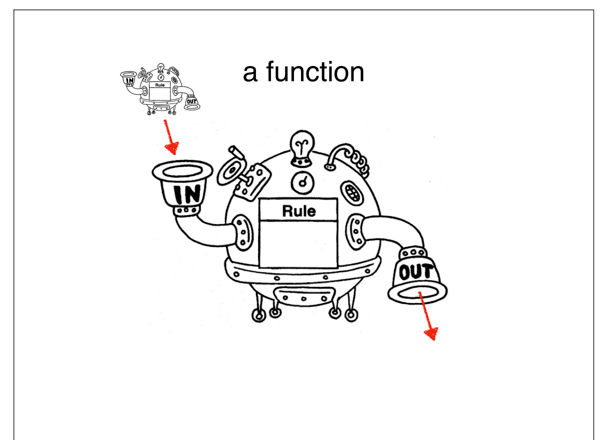
23

A first class function means that function definitions themselves are values. You can use them anywhere you can use ordinary values. You can assign them to variables. You can pass them as arguments in function calls. Very few programming languages allow you to do this.



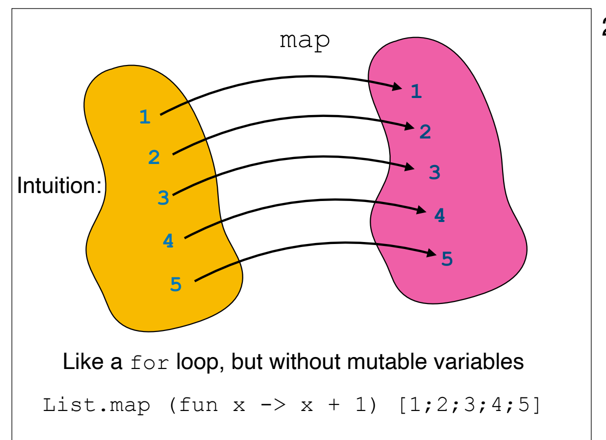
24

Returning to our simple notion of a function…



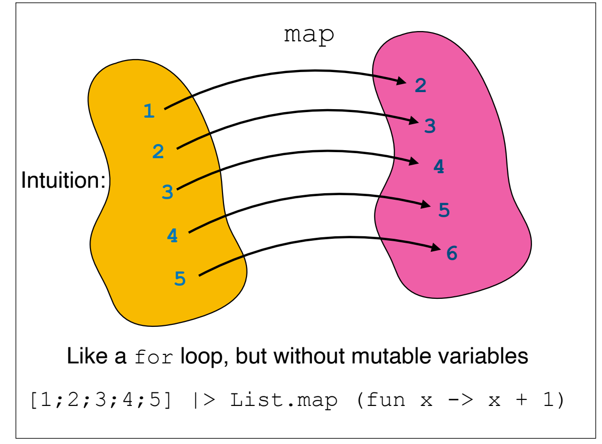
25

… this is an example of a higher-order function. Observe that it depends on the existence of first class functions. A higher order function takes a function definition as an argument.



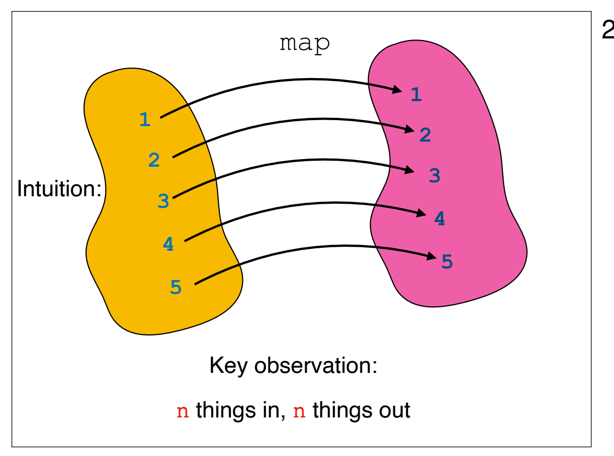
26

An example is the map function. Map takes a function as an argument, and it applies it to every element given to it. You can accomplish the same thing with a loop, but observe that this is actually simpler. The “body” of this “loop” only says what to do when given a single element. It does not worry about “how” to access the element from the list, or where to store it when it is done. List.map returns a new list, and assuming that the given function is O(1), List.map takes O(n) time, so it is efficient.



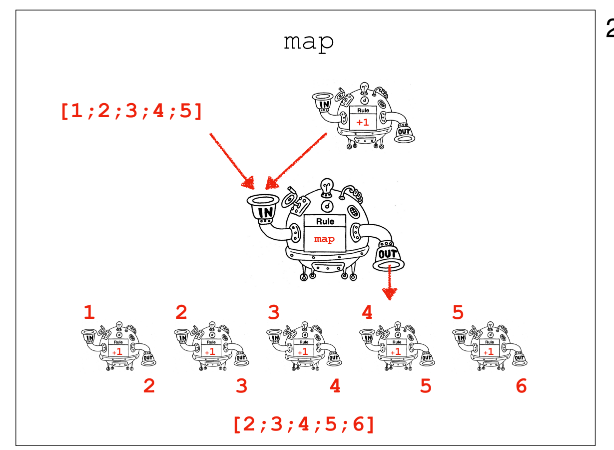
27

You can also rewrite this function using the forward pipe operator so that the data comes first. I personally prefer this style, but whichever you choose is up to you.



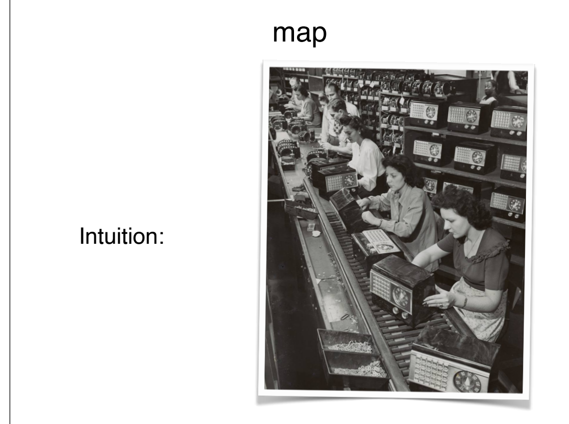
28

An important fact about map is that if you give it n things, you get n things back. Observe that for and while loops give no such guarantees, even when that's what you want them to do.



29

How does it work? List.map will apply the given machine (here a +1 machine) to every element of the input list, yielding a new output list.

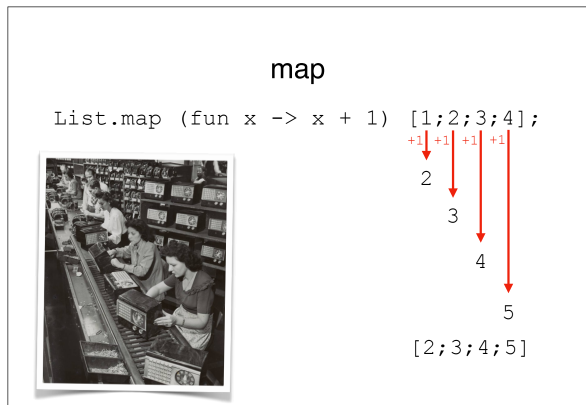


30

The intuition behind map is that is behaves like the worker in an assembly line. That one workers does the same thing over and over. For example, the first person in the line may just put the knobs on the radios. The next person may attach the power cords. And so on. Each person is a "mapper."

map

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



```
+1 +1 +1 +1
2
3
4
5
[2;3;4;5]
```

31

Again, observe we're just adding +1 to each element.

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]
```

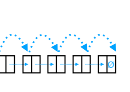
32

You can make an "assembly line" by chaining mappers together. Forward pipe makes these chains easy to read because the first operations come first. For example, x + 1 is performed first, then the conversion to float, then division by 3.3, etc.

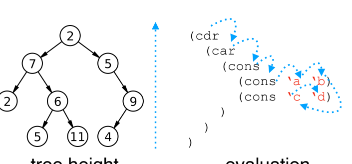
fold

structural recursion → fold it!

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



list length



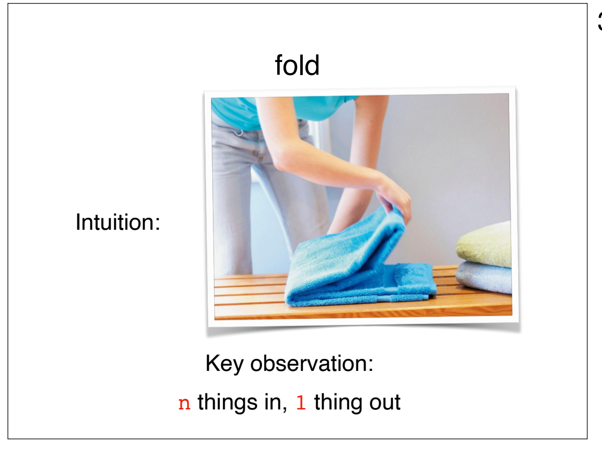
tree height

```
(cdr
  (car
    (cons
      (cons 'a 'b)
      (cons 'c 'd)
    )
  )
)
```

evaluation

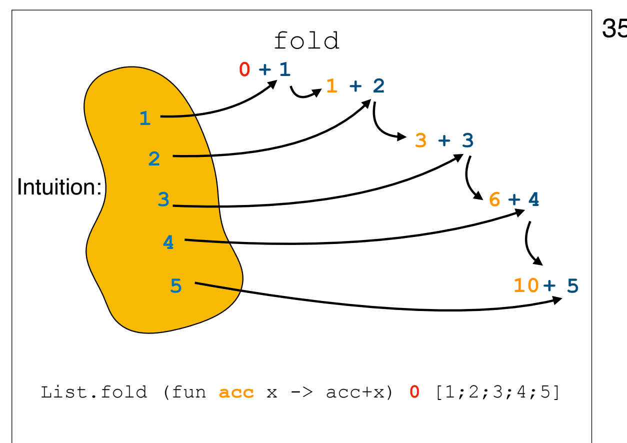
33

Fold is another important idea. It can be used for any problem that exhibits structural recursion. Structural recursion happens any time we need to solve a problem over a recursive data structure. E.g., lists and trees.



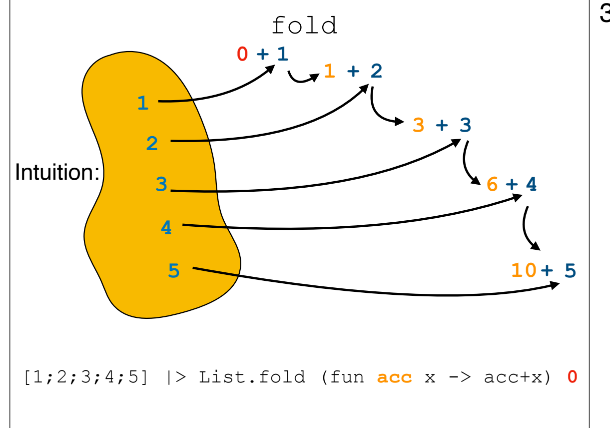
34

The intuition is like a person folding a towel. Unlike mapping, fold takes in n things and returns 1 thing. Importantly, it is *accumulating* those n things into a single thing. The idea of an accumulator is central to folding.



35

For example, suppose we want to sum some numbers. We can define this using fold. Fold takes a “folder” which is a function that says how to accumulate, a default accumulator value, and an input list. For each element, fold runs the given function on the *latest value* of the accumulator with that element. For example, in the beginning, the accumulator is zero and we add it to the first element of the list, one. The result is the new value of the accumulator. So the second element, two, is added to one. Three is new value of the accumulator, and so on.



36

Again, you can rewrite this using pipe forward to move the data to the front.

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
```

37

Another view of the same computation.

what does this return?

```
List.fold
  (fun acc x -> acc + string x)
  ""
  (Seq.toList "williams")
```

38

Try this at home. What does it return. Why?

Recap & Next Class

Today:

- More pattern matching
- Option vs exceptions
- Higher order functions

Next class:

- PL foundations

39