1

CSCI 334:
Principles of Programming Languages

Lecture 9: Higher Order Functions

Instructor: Dan Barowy

Williams

2

Topics

Higher order functions

3

Announcements

- CS Colloquium this **Friday, Mar 1 @ 2:35pm in Wege Auditorium (TCL 123)**

Lightweight, Modular Verification for Systems Compilers
Prof. Alexa VanHattum (Wellesley)

Language-level system guarantees, like runtime isolation for WebAssembly modules, are only as strong as the compiler that produces a native-machine-specific executable. Subtle wrong-code bugs in the compiler can introduce serious security flaws. In this talk, I'll describe Crocus, our system for lightweight, modular verification of instruction-lowering rules in Cranelift, an industry WebAssembly compiler. Crocus reproduces known bugs (including a 9.9/10 severity security bug) and identifies previously-unknown bugs and underspecified compiler invariants. More broadly, I'll discuss how integrating lightweight formal methods can free systems engineers from having to choose between prioritizing efficiency and correctness.

Your to-dos

1. Lab 4, **due Monday 3/4** (solo lab)
2. Reminder: office hours today, 2-3pm, TPL 306

4

Quiz

5

Quiz: debriefing and tip

6

Pro tip: use your code editor to help match parens when you are doing problem sets.

Higher order functions

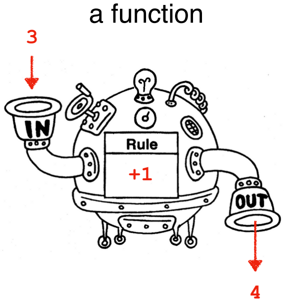
7 HOFs are one of the most important features of functional languages, and its something that makes them stand apart from conventional languages. HOFs give you great flexibility in how you design programs.

Three amazing functional concepts

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

8 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.

a function



9 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 without returning anything is not a true function; it is more properly called a “procedure.”

“first class” function

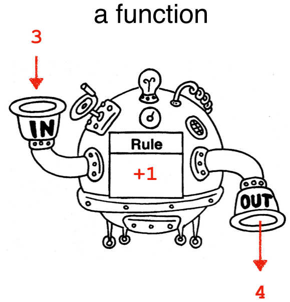
Function definitions are **values** in a functional programming language

10

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. Few programming languages allow you to do this, but the most popular modern ones are adding this capability. E.g., you can do this in Python.

---

a function

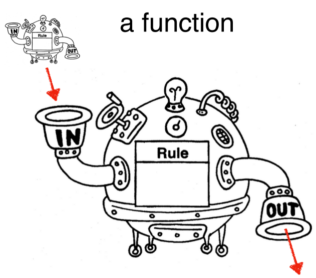


11

Returning to our simple notion of a function…

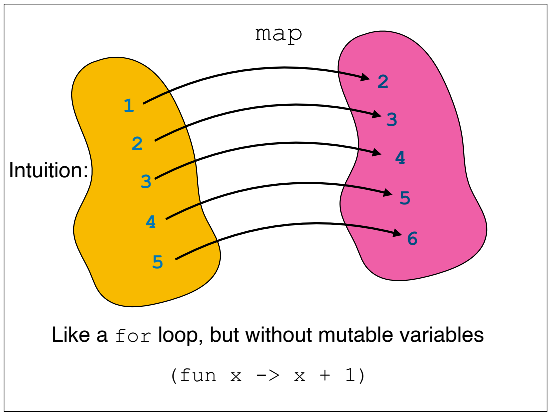
---

a function

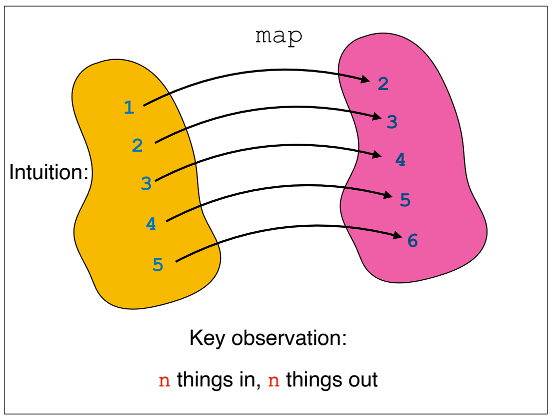


12

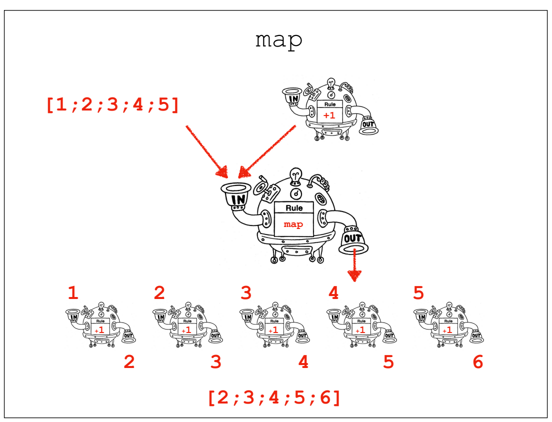
… this is an example of a higher-order function. Observe that it depends on the existence of first class functions. A higher order function takes a function definition as an argument.



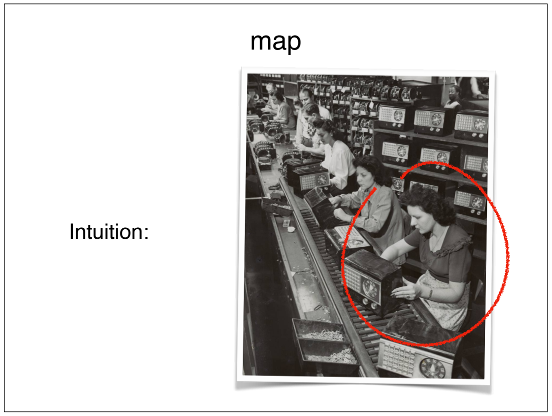
13 Our first example of a higher order function is map. Map takes a function as an argument, and it applies it to every element given to it. You can accomplish the same thing with a loop, but observe that this is actually simpler. The “body” of this “loop” only says what to do when given a single element. 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.



14 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. It’s pretty easy to write a while loop that is supposed to return n things but actually returns n - 1 things instead, by accident. Such mistakes are impossible when using map.

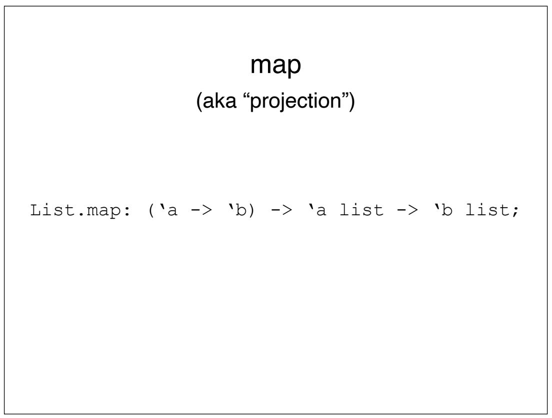


15 How does it work? List.map will apply the given machine (here a +1 machine) to every element of the input list, yielding a new output list.



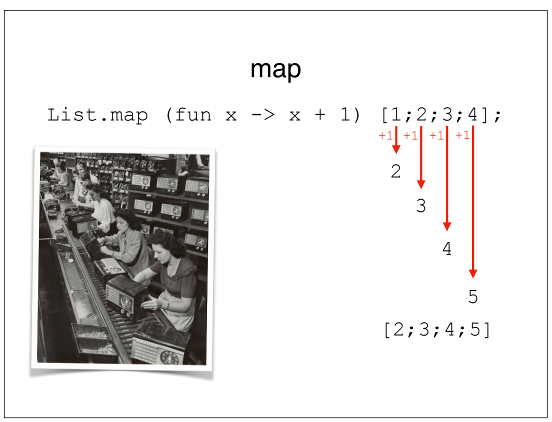
16

The intuition behind map is that is behaves like a 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."



17

List.map is a function that takes a function called a "mapper" and an input list, and it applies the mapper to every element of the list. This operation is sometimes called a "projection."



18

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

pipelines

```
[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]
```

19

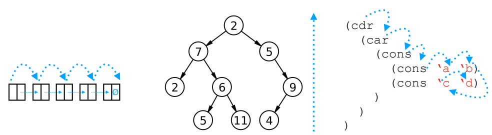
You can make an actual "assembly line" by chaining mappers together. Forward pipe makes these chains easy to read because the first operations come first. For example, x + 1 is performed first, then the conversion to float, then division by 3.3, etc. Think about how you would have to write this in Java. First, you would probably have to define a function for each "worker" in the assembly line; but then you'd have to write the assembly like "inside out." I find the above much easier to understand.

---

fold

structural recursion → fold it!

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



list length   tree height   evaluation

20

Fold is another important kind of higher order function. 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.

---

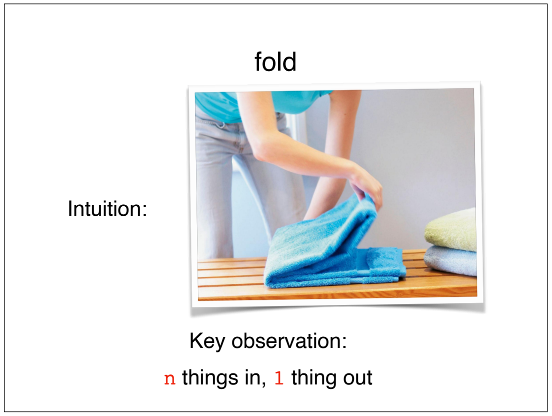
fold

(aka "reduce")

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

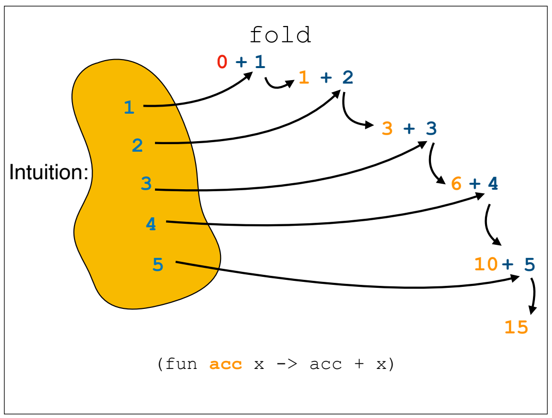
21

List.fold is a function that takes a function called a "folder," an initial value of a value called the "accumulator," and an input list. It then "folds" the accumulator and a list element together, returning a new accumulator. The process repeats until there are no more elements to fold. The value returned is the final value of the accumulator.



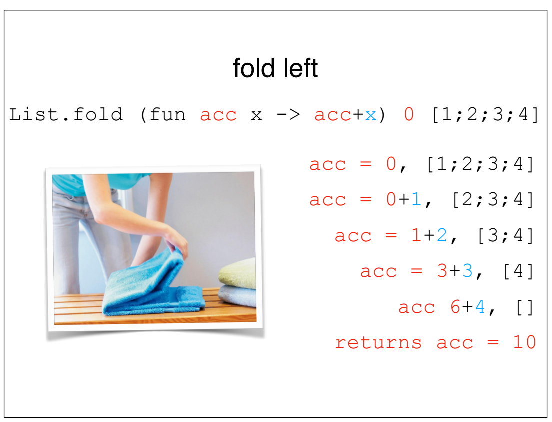
22

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.



23

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 (add two numbers), a default accumulator value (zero), and an input list (some numbers). 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



24

Another view of the same computation.

what does this return?

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

25

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

Recap & Next Class

Today:

Higher order functions

Next class:

More HOFs

The Halting Function

26