Map, filter and reduce

Ken Q Pu, Faculty of Science, Ontario Tech University

Comprehensive Guide

https://clojure.org/api/cheatsheet

Ken Q Pu, Faculty of Science, Ontario Tech University

Principles of functional programming

- Use functional whenever possible
- Abstract the problem with functions and higher order functions
- Build programs by compositions of functions using higher order functions

Ken Q Pu, Faculty of Science, Ontario Tech University

Motivation for functional approach to iteration

Good software engineering practices:

- Minimize modifications
- Maximize function reusability
- No special syntax

Motivation for functional approach to iteration

```
(def xs '(1 2 3 4 5 6 7))
```

```
(for [x xs]
  (pow 2 x))
```

2^x

```
⇒ (2 4 8 16 32 64 128)
```

Ken Q Pu, Faculty of Science, Ontario Tech University

Motivation for functional approach to iteration

```
(def xs '(1 2 3 4 5 6 7))
```

```
(for [x xs]
  (pow 2 x)
  (/ (inc (pow 2 x)) 2))
```

$(2^x + 1) / 2$

```
⇒ (1.5 2.5 4.5 8.5 16.5 32.5 64.5)
```

Ken Q Pu, Faculty of Science, Ontario Tech University

Motivation for functional approach to iteration

```
(def xs '(1 2 3 4 5 6 7))
```

```
(for [x xs]
  (pow 2 x)
  (/ (inc (pow 2 x)) 2)
  (let [y (/ (inc (pow 2 x)) 2)]
    (if (> y 5) 0 1)))

⇒ (1 1 1 0 0 0 0)
```

```
if (2 ^ x + 1) / 2 > 5 then
    0
else
    1
```

Ken Q Pu, Faculty of Science, Ontario Tech University

Motivation for functional approach to iteration

```
(def xs '(1 2 3 4 5 6 7))


(for [x xs]
  (pow 2 x)
  (/ (inc (pow 2 x)) 2)
  (let [y (/ (inc (pow 2 x)) 2)]
    (if (> y 5) 0 1)))

⇒ (1 1 1 0 0 0 0)
```

Modification of inner loop

Code such as (/ (inc (pow 2 x)) 2) are not reusable elsewhere.

Ken Q Pu, Faculty of Science, Ontario Tech University

for-forms are not composable.

```
(def xs '(1 2 3 4 5 6 7))
```

```
(for [x xs]
 (pow 2 x))
```

```
(for [x (for [x xs] (pow 2 x))]
 (inc x))
```

```
(for [x (for [x (for [x xs] (pow 2 x))]
          (inc x))]
 (/ x 2))
```

Growing code complexity

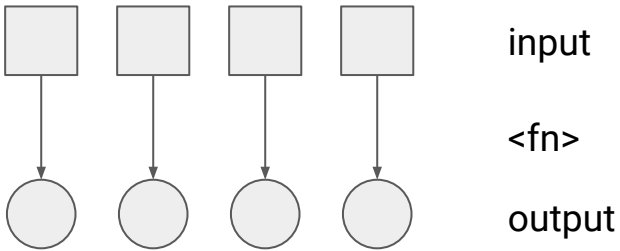
Ken Q Pu, Faculty of Science, Ontario Tech University

Introducing map function

Map is a higher-order function. There is no special syntax for it.

```
(map <fn> <sequence>)
```

This is functionally equivalent to the basic for-form.



Implementation of map

```
(defn map [f xs] (for [x xs] (f x)))
```

*So, we do we care to use **map** instead of the for-form?*

Map based iteration

```
(def xs '(1 2 3 4 5 6 7))
```

```
(map (partial pow 2) xs)

⇒ (2 4 8 16 32 64 128)
```

```
(for [x xs]
  (pow 2 x))

⇒ (2 4 8 16 32 64 128)
```

Map based iteration

```
(def xs '(1 2 3 4 5 6 7))
```

```
(map (comp
       inc
       (partial pow 2) xs)
```

```
(for [x xs]
  (pow 2 x)
  (/ (inc (pow 2 x)) 2))
```

Ken Q Pu, Faculty of Science, Ontario Tech University

Map based iteration

```
(def xs '(1 2 3 4 5 6 7))
```

```
(map (comp
       (fn [x] (if (> x 5) 0 1))
       (fn [x] (/ x 2))
       inc
       (partial pow 2) xs)
```

```
(for [x xs]
  (pow 2 x)
  (/ (inc (pow 2 x)) 2)
  (let [y (/ (inc (pow 2 x)) 2)]
    (if (> y 5) 0 1)))
```

Ken Q Pu, Faculty of Science, Ontario Tech University

Composition of map with threading

```
(def xs '(1 2 3 4 5 6 7))

(map (comp
       (fn [x] (if (> x 5) 0 1))
       (fn [x] (/ x 2))
       inc
       (partial pow 2) xs)
```

```
(defn half [x] (/ x 2))
(defn greater-than [n]
  (fn [x] (> x n)))

(->> xs
 (map (partial pow 2))
 (map inc)
 (map half)
 (map (comp complement (partial
greater-than 5)))
```

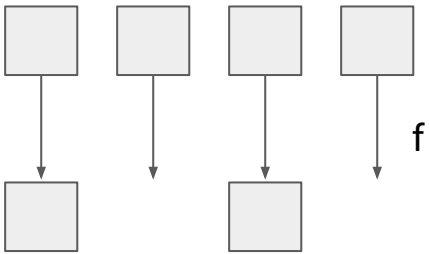
Ken Q Pu, Faculty of Science, Ontario Tech University

Filter

Filter is a higher-order function. There is no special syntax for it.

```
(filter <fn> <sequence>)
```

This is functionally equivalent to the basic for-form.



Only elements with (f x) being true is kept in the output.

Filter

```
(def xs '(1 2 3 4 5 6 7))

(filter even? xs)
⇒ (2 4 6)
```

Map and filter

```
(def xs '(1 2 3 4 5 6 7))
```

```
(for [x xs
      :when (even? x)]
  (pow x 2))
```

Compute the 2^x of all the even numbers in xs.

```
(for [sq (for [x xs
               :when (even? x)]
           (pow x 2))
      :when (< 10 sq)]
 sq)
```

2^x of even numbers that are larger than 10.

We need to nest for-forms.

Map and filter

```
(def xs '(1 2 3 4 5 6 7))

(->> xs (filter even?))
```

Even numbers in xs

Map and filter

```
(def xs '(1 2 3 4 5 6 7))

(->> xs (filter even?)
        (map (partial pow 2)))
```

2^x for even x in xs

Map and filter

```
(def xs '(1 2 3 4 5 6 7))

(->> xs (filter even?)
        (map (partial 2))
        (filter (partial < 10)))
```

Keep only these 2^x for even x in xs greater than 10.

Limitations of map and filter

Map and filter can only produce outputs that are sequences.

But some programs need to transform sequences to a value.

```
(def xs [1 2 3 4])
```

```
(loop [total 0
       xs xs]
 (if (empty? xs)
  total
  (let [x (first xs)]
   (recur (+ total x) (rest xs)))))
```

For this particular example, we can do it easily with (apply + xs).

Ken Q Pu, Faculty of Science, Ontario Tech University

Reduce: a higher-order function

```
(reduce <fn> <initial-value> <input-sequence>)
```

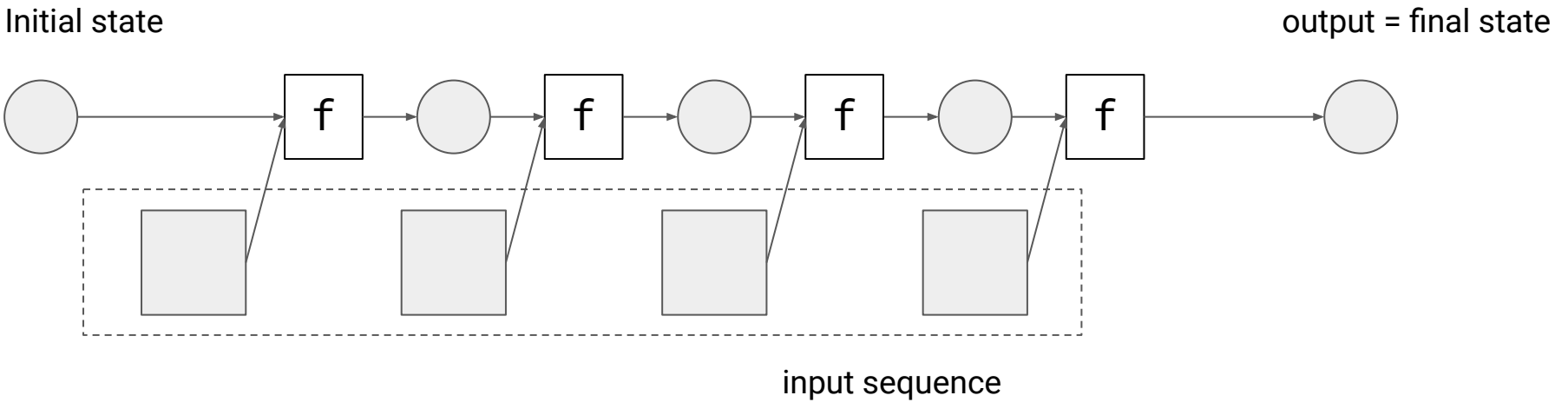
reduce maintains a state value using a reducer function while iterating over the input sequence.

The reducer function computes the next state value based on the previous state and current element in the sequence.

```
(<fn> <state> <elem>)
```

The final state is the output value.

Reduce: a higher-order function



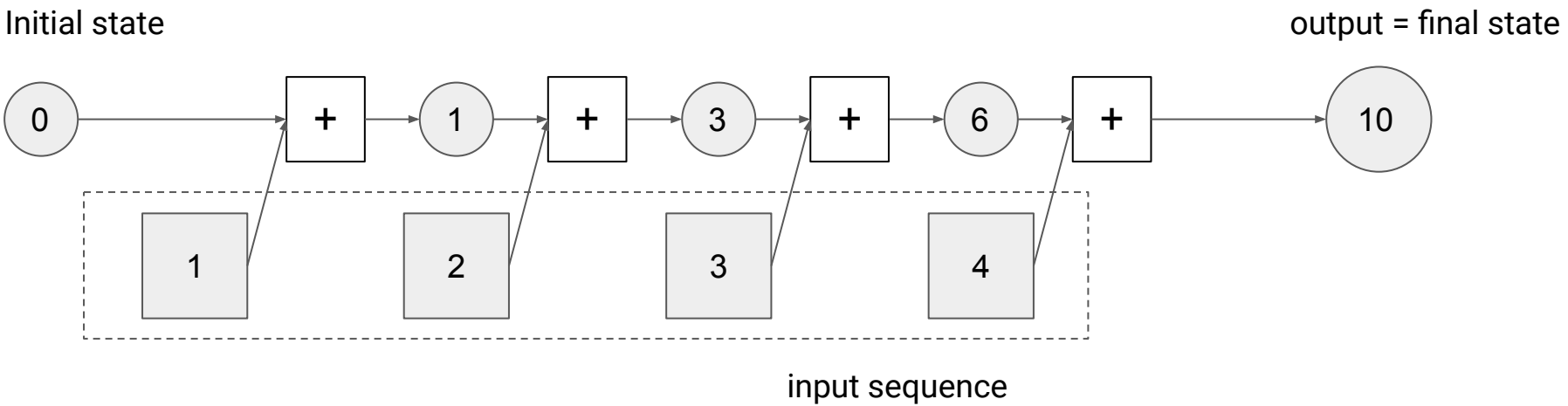
Reduce

```
(def xs [1 2 3 4])
```

```
(reduce + 0 xs)
```

```
(loop [total 0
       xs xs]
 (if (empty? xs)
  total
  (let [x (first xs)]
   (recur (+ total x) (rest xs)))))
```

Reduce: a higher-order function



Reduce is flexible

```
(defn my-map [f xs]
 (reduce
  (fn [s x] (conj s (f x)))
  []
  xs))
```

```
(defn my-filter[f xs]
 (reduce
  (fn [s x]
   (if (f x) (conj s x) s))
  []
  xs))
```

Map and filter and reduce

```
(def xs '(1 2 3 4 5 6 7))
```

```
(->> xs (filter even?)
        (map (partial 2))
        (filter (partial < 10))
        (reduce + 0))
```

Keep only these 2^x for even x in xs greater than 10.

Then find the total.

Functions are composed (preferred)
vs
Forms are nested (less preferred).