UMass Boston Computer Science

CS450 High Level Languages

Accumulators

Thursday, March 6, 2025

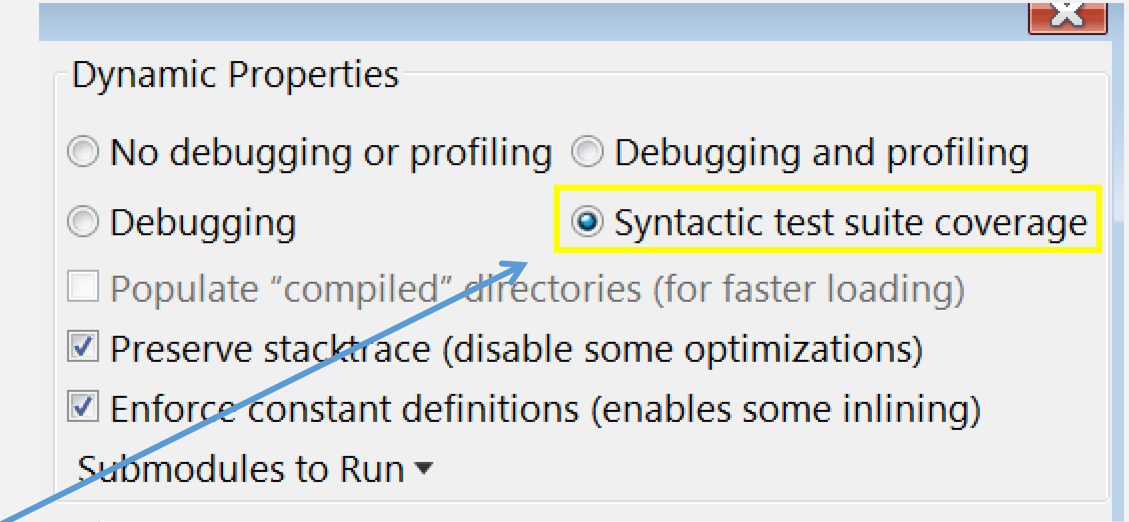
Logistics

- HW 5 out
  - Due: Tues 3/11 11am EST

  - Yes: `cons`, `first`, `rest`, `empty`, `list`, `empty?`
  - No: map, fold, filter, etc
  - 1 `if` allowed, if following multi-arg recipe

HW Minimum Submission Requirements

- "`main`" runs without errors
- Tests run without errors
- 100% (Test / Example) "Coverage"
  - In "Choose Language" Menu
  - NOTE: only works with single files



Dynamic Properties

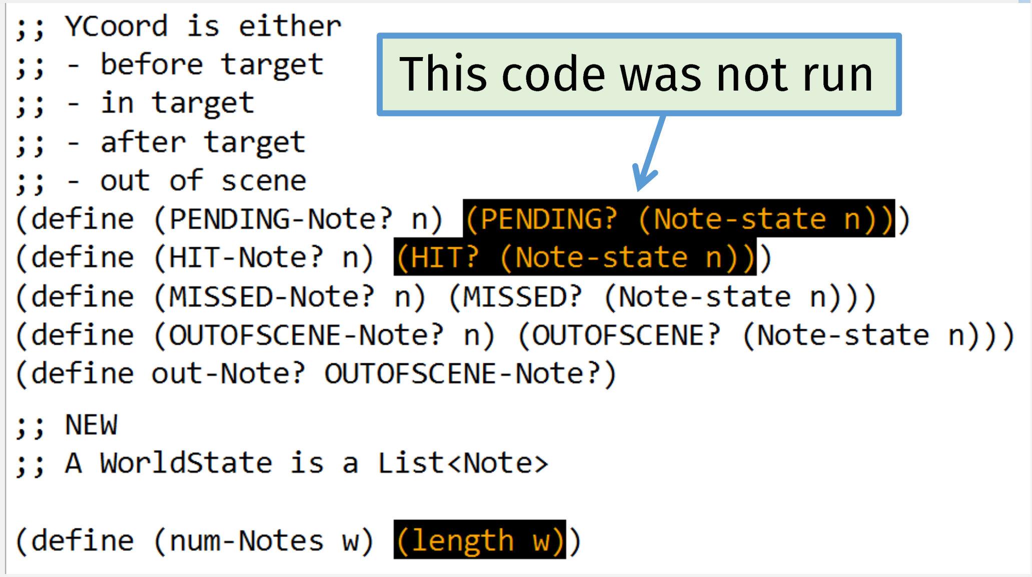
○ No debugging or profiling ○ Debugging and profiling

○ Debugging ● Syntactic test suite coverage

□ Populate "compiled" directories (for faster loading)

☑ Preserve stacktrace (disable some optimizations)

☑ Enforce constant definitions (enables some inlining)



This code was not run

```
;; YCoord is either
;; - before target
;; - in target
;; - after target
;; - out of scene
(define (PENDING-Note? n) (PENDING? (Note-state n)))
(define (HIT-Note? n) (HIT? (Note-state n)))
(define (MISSED-Note? n) (MISSED? (Note-state n)))
(define (OUTOFSCENE-Note? n) (OUTOFSCENE? (Note-state n)))
(define out-Note? OUTOFSCENE-Note?)

;; NEW
;; A WorldState is a List<Note>

(define (num-Notes w) (length w))
```

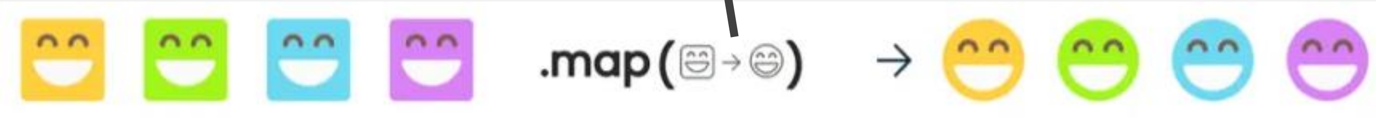
Last Time

Common List Function #1: **map**

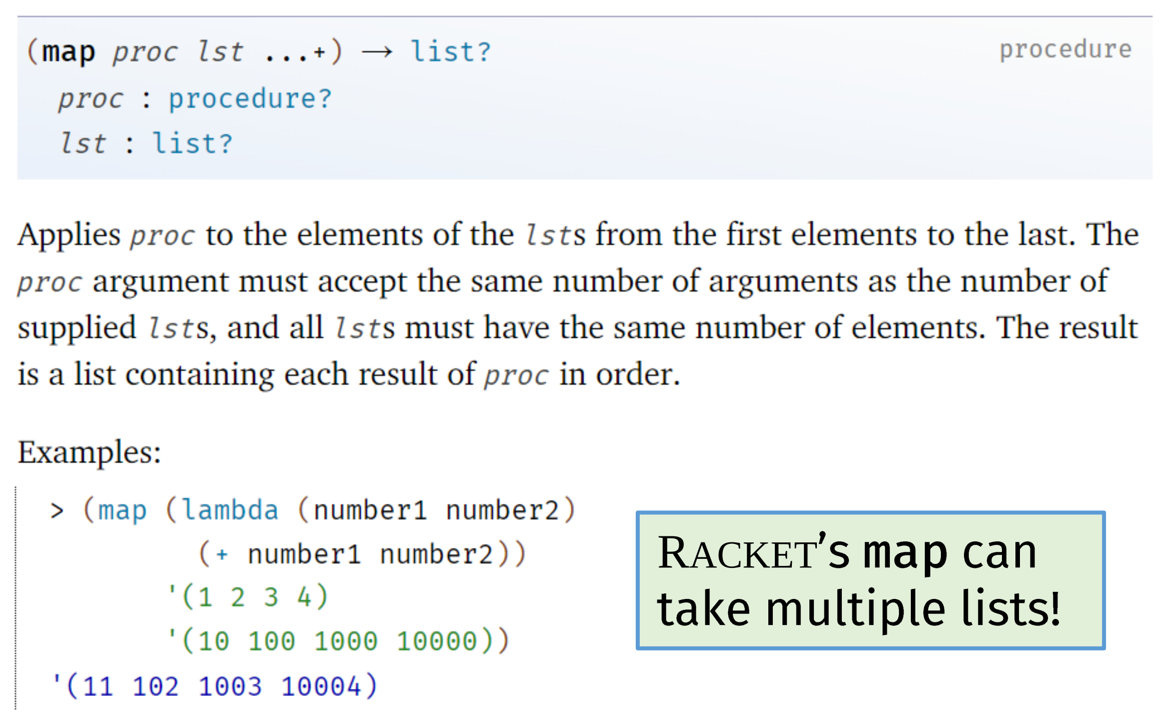
```
;; map: (X -> Y) Listof<X> -> Listof<Y>
;; Produces a list resulting from applying
;; a given fn to each element of a given lst
```

```
(define (map fn lst)
  (cond
    [(empty? lst) empty]
    [else (cons (fn (first lst))
                (map (rest lst)))]))
```

function "**application**"
(in high-level langugues)
= function "call"
(in imperative languages)



https://javascript.plainenglish.io/map-filter-find-reduce-functions-in-js-fbcdf6c670d7



```
(map proc lst ...+) → list?                procedure
  proc : procedure?
  lst : list?
```

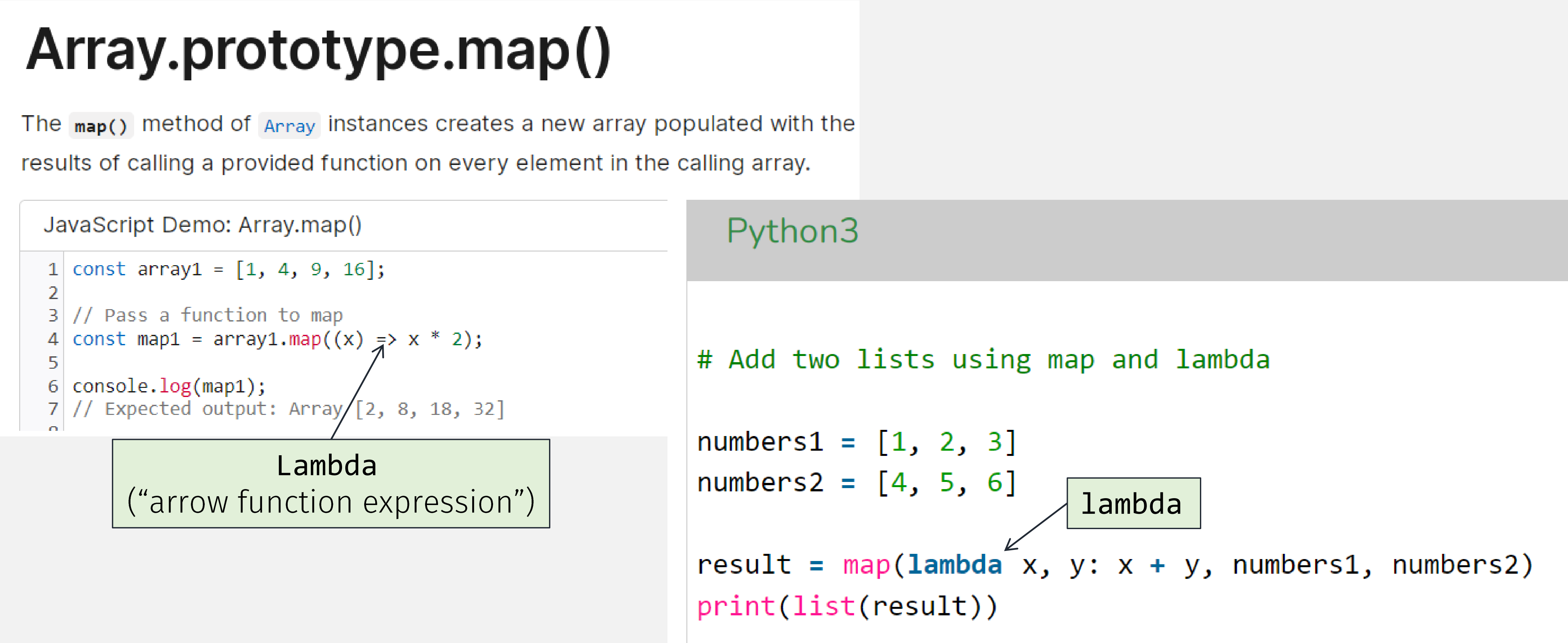
Applies *proc* to the elements of the *lsts* from the first elements to the last. The *proc* argument must accept the same number of arguments as the number of supplied *lsts*, and all *lsts* must have the same number of elements. The result is a list containing each result of *proc* in order.

Examples:

```
> (map (lambda (number1 number2)
         (+ number1 number2))
       '(1 2 3 4)
       '(10 100 1000 10000))
'(11 102 1003 10004)
```

RACKET's **map** can take multiple lists!

`map` in other high-level languages



Array.prototype.map()

The `map()` method of `Array` instances creates a new array populated with the results of calling a provided function on every element in the calling array.

JavaScript Demo: Array.map()

```
const array1 = [1, 4, 9, 16];

// Pass a function to map
const map1 = array1.map((x) => x * 2);

console.log(map1);
// Expected output: Array [2, 8, 18, 32]
```

Lambda
(“arrow function expression”)

Python3

```
# Add two lists using map and lambda

numbers1 = [1, 2, 3]
numbers2 = [4, 5, 6]

result = map(lambda x, y: x + y, numbers1, numbers2)
print(list(result))
```

lambda

Last Time

Common List Function #2: `foldl` / `foldr`

```
;; foldr: (X Y -> Y) Y Listof<X> -> Y
;; Computes a single value from given list, determined by given fn and initial val.
;; fn is applied to each list element, last-element-first
(define (foldr fn initial lst)
  (cond
   [(empty? lst) initial]
   [else (fn (first lst) (foldr fn initial (rest lst)))]))
```

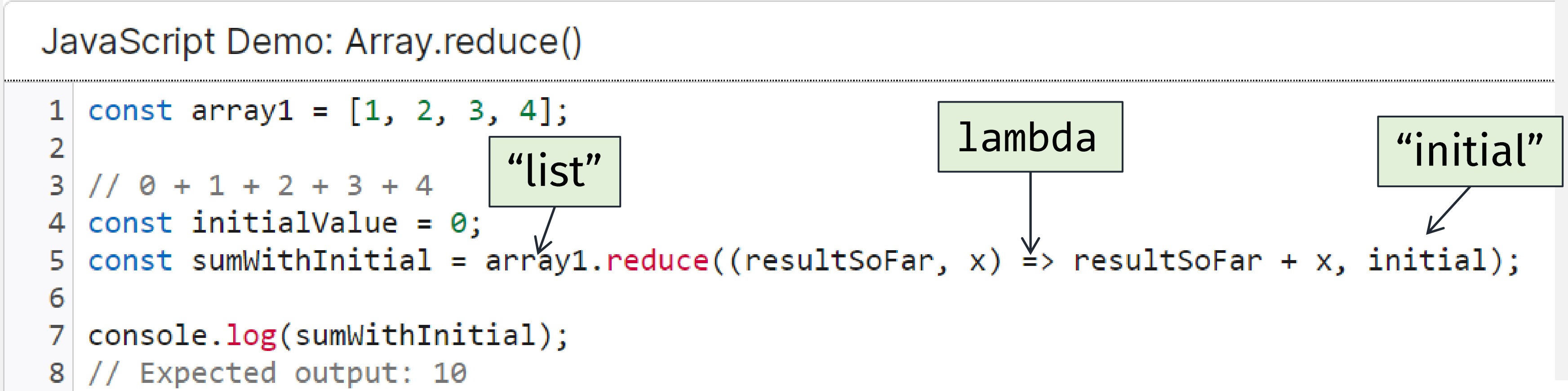
```
(1 + (2 + (3 + 0)))
(1 - (2 - (3 - 0)))
```

```
;; foldl: (X Y -> Y) Y Listof<X> -> Y
;; Computes a single value from given list, determined by given fn and initial val.
;; fn is applied to each list element, first-element-first
(define (foldl fn result-so-far lst)
  (cond
   [(empty? lst) result-so-far]
   [else (foldl fn (fn (first lst) result-so-far) (rest lst))]))
```

```
(((1 + 0) + 2) + 3)
(((1 - 0) - 2) - 3)
```

Last Time

`fold` (`reduce`) in other high-level languages



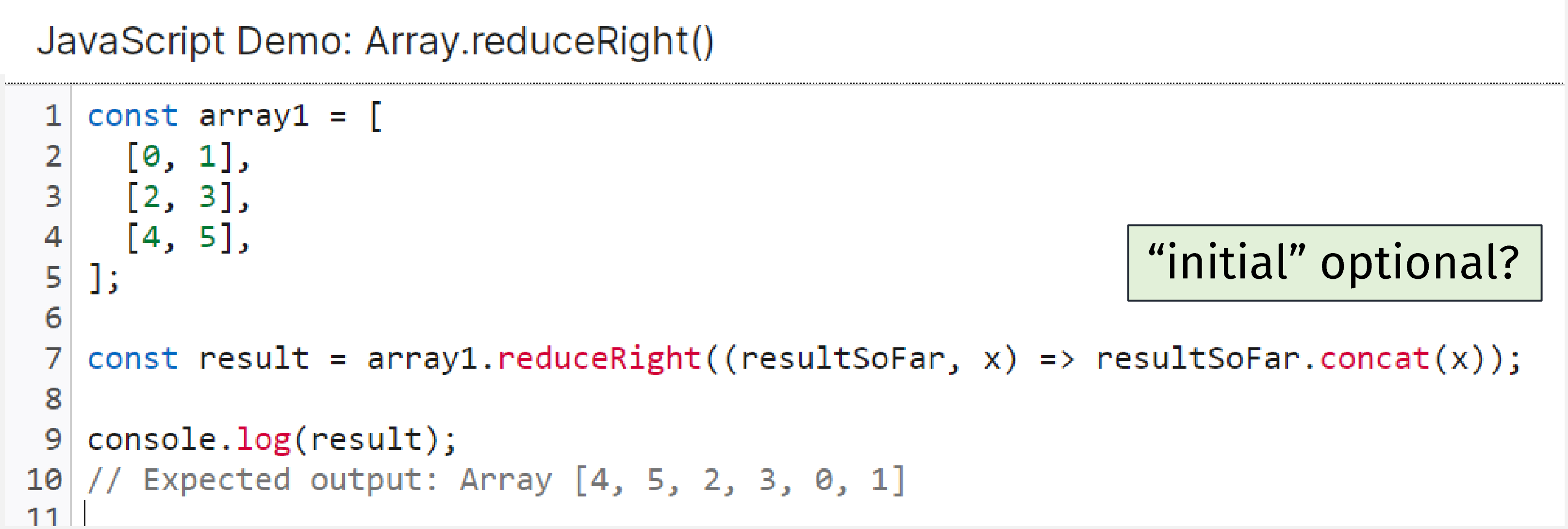
JavaScript Demo: Array.reduce()

```javascript
const array1 = [1, 2, 3, 4];

// 0 + 1 + 2 + 3 + 4
const initialValue = 0;
const sumWithInitial = array1.reduce((resultSoFar, x) => resultSoFar + x, initial);

console.log(sumWithInitial);
// Expected output: 10

```

"list"

lambda

"initial"

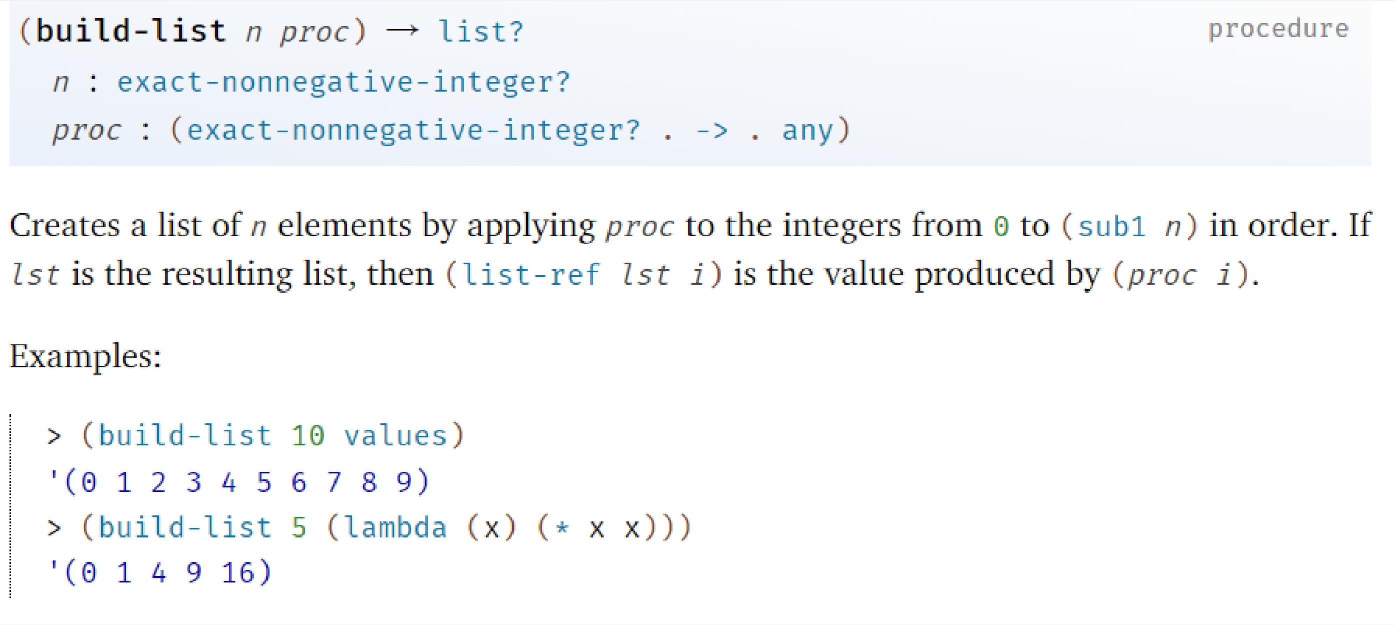


JavaScript Demo: Array.reduceRight()

```javascript
const array1 = [
  [0, 1],
  [2, 3],
  [4, 5],
];

const result = array1.reduceRight((resultSoFar, x) => resultSoFar.concat(x));

console.log(result);
// Expected output: Array [4, 5, 2, 3, 0, 1]

```

"initial" optional?

Fold “dual”: `build-list`



```
(build-list n proc) → list?                              procedure
  n : exact-nonnegative-integer?
  proc : (exact-nonnegative-integer? . -> . any)
```

Creates a list of *n* elements by applying *proc* to the integers from `0` to `(sub1 n)` in order. If *lst* is the resulting list, then `(list-ref lst i)` is the value produced by `(proc i)`.

Examples:

```
> (build-list 10 values)
'(0 1 2 3 4 5 6 7 8 9)
> (build-list 5 (lambda (x) (* x x)))
'(0 1 4 9 16)
```

```
(build-list 4 add1)
;; = (map add1 (list 0 1 2 3))
;; = (list 1 2 3 4)
```

Fold “alternative”: **apply** (with “variable-arity” fns)

```
(foldl + 0 (list 1 2 3 4)) ; = (+ (+ (+ (+ 1 0) 2) 3) 4)) = 10
```

```
(apply + (list 1 2 3 4)) ; = (+ 1 2 3 4) = 10
```

```
(apply string-append (list “a” “b” “cd”)) ; = “abcd”
```

- **`apply` applies its fn argument to** the contents of **its `list` arg**
- function arg must accept:
  # of arguments = length of `list` arg

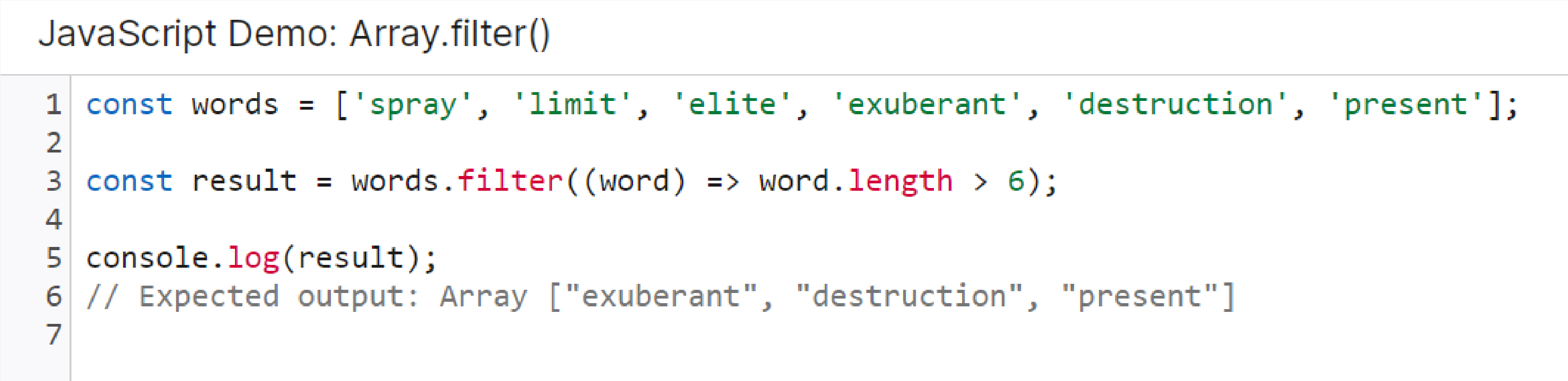
Common list function #3: filter

```
;; filter: Listof<X> (X -> Boolean) -> Listof<X>
;; Returns a list containing elements of given list
;; for which the given predicate returns true
```

```
(define (filter lst pred?)
  (cond
   [(empty? lst) empty]
   [else (if (pred? (first lst))
             (cons (first lst) (filter (rest lst)))
             (filter (rest lst)))]))
```

pred and lst must be processed together, so 1 **if** allowed here

`filter` in other high-level languages



JavaScript Demo: Array.filter()

```
const words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];

const result = words.filter((word) => word.length > 6);

console.log(result);
// Expected output: Array ["exuberant", "destruction", "present"]

```

Not allowed in HW5!

Common list function #3: **`filter`**

```
;; filter: Listof<X> (X -> Boolean) -> Listof<X>
;; Returns a list containing elements of g
;; for which the given predicate returns t
```

```
(define (filter lst pred?)
  (cond
   [(empty? lst) empty]
   [else (if (pred? (first lst))
             (cons (first lst) (filter (re
             (filter (rest lst)))]))
```

lambda rules:

- Can skip the **design recipe** steps, BUT
- **name**, **description**, and **signature** must be "obvious"
- **code** is arithmetic only
- otherwise, **create standalone function define**

```
;; smaller-than: Listof<Int> Int -> Listof<Int>
;; Returns a list containing elements of given list less than the given int
```

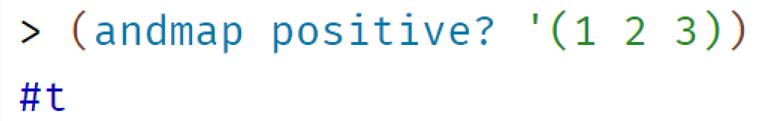
```
(define (smaller-than lst thresh)
  (filter (lambda (x) (< x thresh)) lst)
```

lambda creates an anonymous "inline" function (expression)

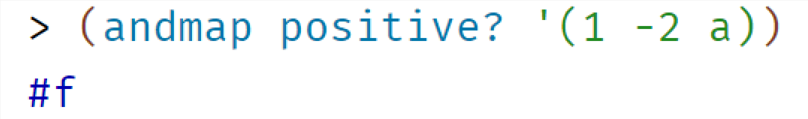
Another Useful List Function: **andmap**

“all”

“every”



```
> (andmap positive? '(1 2 3))
#t
```



```
> (andmap positive? '(1 -2 a))
#f
```

`(andmap p? lst)` similar to: `(apply and (map p? lst))` But (won’t run): **andmap** is “short circuiting”

`(foldl and #t lst)` But (won’t run): **and** is not a function

`(foldl (λ (x y) (and x y)) #t lst)`

See also: **ormap**

“any”, “some”

Another List function: `lst-max`

Function design recipe:

1. Name
2. Signature
3. Description
4. Examples
5. Template

…

```
;; lst-max : Listof<Int> -> Int
;; Returns the largest number in the given list
```

Another List function: `lst-max`

Function design recipe:

1. Name
2. Signature
3. Description
4. Examples
5. Template

…

```
;; lst-max : Listof<Int> -> Int
;; Returns the largest number in the given list
```

```
(check-equal?
  (lst-max (list 1 2 3)) 3))
```

```
(check-equal?
  (lst-max (list)) ???))
```

Another List function: `lst-max`

Function design recipe:

1. Name
2. Signature
3. Description
4. Examples
5. Template

…

```
;; lst-max : Listof<Int> -> Int
;; Returns the largest number in the given list
(define (lst-max lst)
  (cond
    [(empty? lst) ....]
    [(cons? lst) .... (first lst)  ....
        ....  (lst-max (rest lst)) ....]))
```

Another List function: `lst-max`

Function design recipe:

1. Name
2. Signature
3. Description
4. Examples
5. Template

…

```
;; lst-max : Listof<Int> -> Int
;; Returns the largest number in the given list
(define (lst-max lst)
  (cond
    [(empty? lst) ???]
    [(cons? lst) .... (first lst)  ....
        ....  (lst-max (rest lst)) ....]))
```

Another List function: `lst-max`

Function design recipe:

1. Name
2. Signature
3. Description
4. Examples
5. Template

…

```
;; lst-max : Listof<Int> -> Int
;; Returns the largest number in the given list
(define (lst-max lst init-val)
  (cond
    [(empty? lst) ???]
    [(cons? lst) .... (first lst)  ....
       ....  (lst-max (rest lst)) ....]))
```

Need extra information?

Design Recipe For Accumulator Functions

When a function needs **“extra information”:**

1. *Specify* **accumulator:**
   - Name
   - Signature
   - **Invariant**
     - A property of the accumulator that is always true

Another List function: `lst-max`

```
;; lst-max : Listof<Int> Int -> Int
;; Returns the largest number in the given list
;; accumulator max-so-far : Int
;; invariant: is the largest val in lst “so far”
(define (lst-max lst max-so-far)
  (cond
    [(empty? lst) ???]
    [(cons? lst) .... (first lst)  ....
       ....  (lst-max (rest lst)) ....]))
```

Need extra information?

Another List function: `lst-max`

```
;; lst-max : Listof<Int> Int -> Int
;; Returns the largest number in the given list
;; accumulator max-so-far : Int
;; invariant: is the largest val in lst “so far”
(define (lst-max lst max-so-far)
  (cond
    [(empty? lst) ???]
    [(cons? lst) .... (first lst)  ....
       ....  (lst-max (rest lst)) ....]))
```

Need extra information?

Another List function: lst-max

```
;; lst-max : Listof<Int> Int -> Int
;; Returns the largest number in the given list
;; accumulator max-so-far : Int
;; invariant: is the largest val in lst “so far”
(define (lst-max lst max-so-far)
  (cond
    [(empty? lst) max-so-far]
    [(cons? lst) .... (first lst)  ....
       ....  (lst-max (rest lst)) ....]))
```

Another List function: `lst-max`

But this is not the same function as before!

```
;; lst-max : Listof<Int> Int -> Int
;; Returns the largest number in the given list
;; accumulator max-so-far : Int
;; invariant: is the largest val in lst “so far”
(define (lst-max lst max-so-far)
  (cond
    [(empty? lst) max-so-far]
    [else (lst-max (rest lst)
              (max (first lst) max-so-far))]))
```

Update the accumulator

Design Recipe For Accumulator Functions

When a function needs **"extra information"**:

1. *Specify* **accumulator:**
   - Name
   - Signature
   - Invariant
     - A property of the accumulator that is always true
2. *Define* **internal "helper" fn** with **extra accumulator** arg

(Helper fn does not need extra description, statement, or examples, if they are the same …)

3. *Call* **"helper" fn ,** with initial **accumulator value,** from **original fn**

A List Accumulator Example

Function needs “extra information” ...

```
;; lst-max : List<Int> -> Int
;; Returns the largest value in the given list
(define (lst-max initial-lst)

  ;; lst-max/accum : List<Int> Int -> Int
  ;; accumulator max-so-far : Int
  ;; invariant: is the largest val in initial-lst “so far”
  (define (lst-max/accum lst max-so-far)
    (cond
      [(empty? lst) max-so-far]
      [else (lst-max/accum (rest lst)
                           (max (first lst) max-so-far))]))

  (lst-max/accum (rest initial-lst) (first initial-lst) ))
```

1. *Specify* **accumulator:** name, signature, invariant

2. *Define* internal “helper” fn with **accumulator** arg

A List Accumulator Example

```
;; lst-max : List<Int> -> Int
;; Returns the largest value in the given list
(define (lst-max initial-lst)

  ;; lst-max/accum : List<Int> Int -> Int
  ;; accumulator max-so-far : Int
  ;; invariant: is the largest val in initial-lst   "so far"
  (define (lst-max/accum lst max-so-far)
    (cond
      [(empty? lst) max-so-far]
      [else (lst-max/accum (rest lst)
                           (max (first lst) max-so-far))]))

  (lst-max/accum (????rest initial-lst)   (first ????initial-lst)   ))
```

3.Call “helper” fn, with **initial accumulator** (and other args)

A List Accumulator Example

```
;; lst-max : List<Int> -> Int
;; Returns the largest value in the given list
(define (lst-max initial-lst)

  ;; lst-max/accum : List<Int> Int -> Int
  ;; accumulator max-so-far : Int
  ;; invariant: is the largest val in initial-lst   “so far”
  (define (lst-max/accum lst max-so-far)
    (cond
      [(empty? lst) max-so-far]
      [else (lst-max/accum (rest lst)
                           (max (first lst) max-so-far))]))

  (lst-max/accum (rest initial-lst) (first initial-lst) ))
```

3.Call “helper” fn, with **initial accumulator** (and other args)

A List Accumulator Example

```
;; lst-max : NonEmptyList<Int> -> Int
;; Returns the largest value in the given list
(define (lst-max initial-lst)

  ;; lst-max/accum : List<Int> Int -> Int
  ;; accumulator max-so-far : Int
  ;; invariant: is the largest val in initial-lst  "so far"
  (define (lst-max/accum lst max-so-far)
    (cond
      [(empty? lst) max-so-far]
      [else (lst-max/accum (rest lst)
                           (max (first lst) max-so-far))]))



  (lst-max/accum (rest initial-lst)  (first initial-lst)  ))
```

A List Accumulator Example

```
;; lst-max : NonEmptyList<Int> -> Int
;; Returns the largest value in the given list
```

```
(define (lst-max initial-lst)

  ;; lst-max/accum : List<Int> Int -> Int
  ;; accumulator max-so-far : Int
  ;; invariant: is the largest val in initial-lst “so far”
  (define (lst-max/accum lst max-so-far)
    (cond
      [(empty? lst) max-so-far]
      [else (lst-max/accum (rest lst)
                           (max (first lst) max-so-far))]))

  (lst-max/accum (rest initial-lst) (first initial-lst) ))
```

Helper needs signature, etc if different

A List Accumulator Example

```
;; lst-max : NonEmptyList<Int> -> Int
;; Returns the largest value in the given list
(define (lst-max initial-lst)

  ;; lst-max/accum : List<Int> Int -> Int
  ;; accumulator max-so-far : Int
  ;; invariant: is the largest val in initial-lst “minus” lst
  (define (lst-max/accum lst max-so-far)
    (cond
      [(empty? lst) max-so-far]
      [else (lst-max/accum (rest lst)
                           (max (first lst) max-so-far))]))



  (lst-max/accum (rest initial-lst)  (first initial-lst)  ))
```

Invariant should be specific

A List Accumulator Example

```
;; lst-max : NonEmptyList<Int> -> Int
;; Returns the largest value in the given list
(define (lst-max lst0)

   ;; lst-max/a : List<Int> Int -> Int
   ;; accumulator max-so-far : Int
   ;; invariant: is the largest val in lst0 “minus” rst-lst
   (define (lst-max/a rst-lst max-so-far)
     (cond
       [(empty? rst-lst) max-so-far]
       [else (lst-max/accum (rest lst)
                            (max (first lst) max-so-far))]))



  (lst-max/a (rest lst0) (first lst0)))
```

Can Implement with ...

map ? ☒

filter ? ☒

fold ? ☑

Prev

Common List Function: **foldl**

```
;; foldl: (X Y -> Y) Y Listof<X> -> Y
;; Computes a single value from given list,
;; determined by given fn and initial val.
;; fn is applied to each list element, first-element-first
```

```
(define (foldl fn result-so-far lst)
  (cond
   [(empty? lst) result-so-far]
   [else (foldl fn (fn (first lst) result-so-far) (rest lst))]))
```

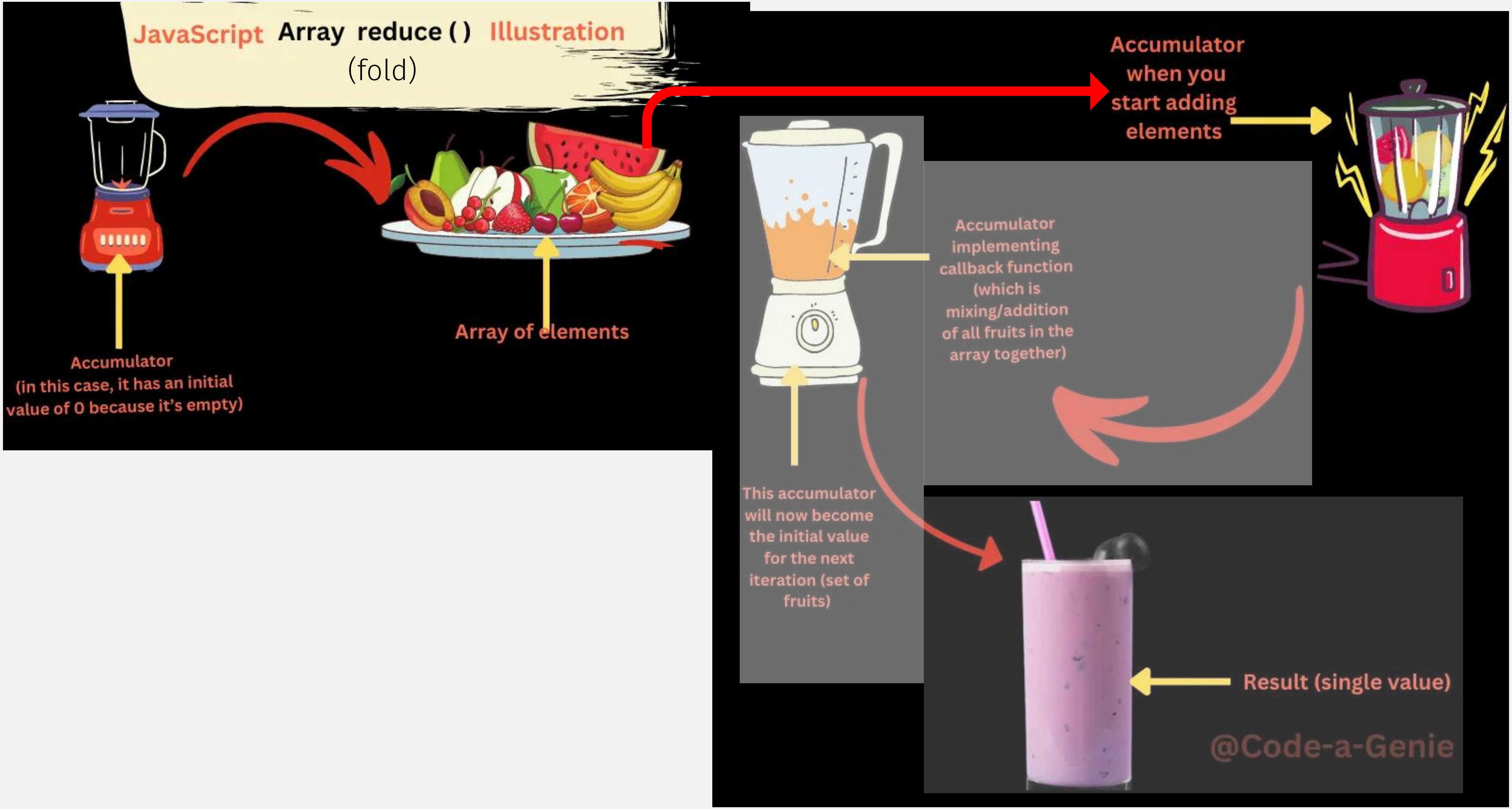
Accumulator!

Update the accumulator

```
;; sum-lst: ListofInt -> Int
(define (sum-lst lst) (foldl + 0 lst))
```

```
(((1 + 0) + 2) + 3)
```

```
(((1 - 0) - 2) - 3)
```



(fold)

https://medium.com/@code-a-genie/javascript-array-reduce-using-real-world-analogies-1b3f169bd9f3

In-class Coding 3/6: Accumulators

```
;; rev : List<X> -> List<X>
;; Returns the given list with elements in reverse order
(define (rev lst0)

  ;; accumulator ??? : ???
  ;; invariant: ???

  (define (rev/a lst acc ???)
    ???
  )




  (rev/a lst0 ???))
```

1. *Specify* **accumulator:** name, signature, invariant

2. *Define* internal “helper” fn with **accumulator** arg

3.*Call* “helper” fn, with initial **accumulator**