UMass Boston Computer Science

CS450 **High Level Languages** (section 2)

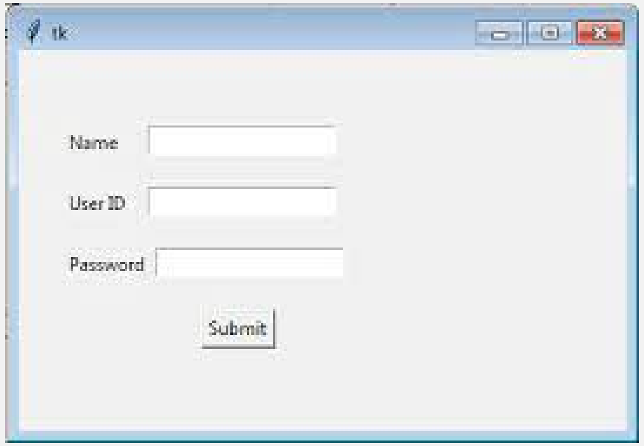
Accumulators

Monday, November 6, 2023

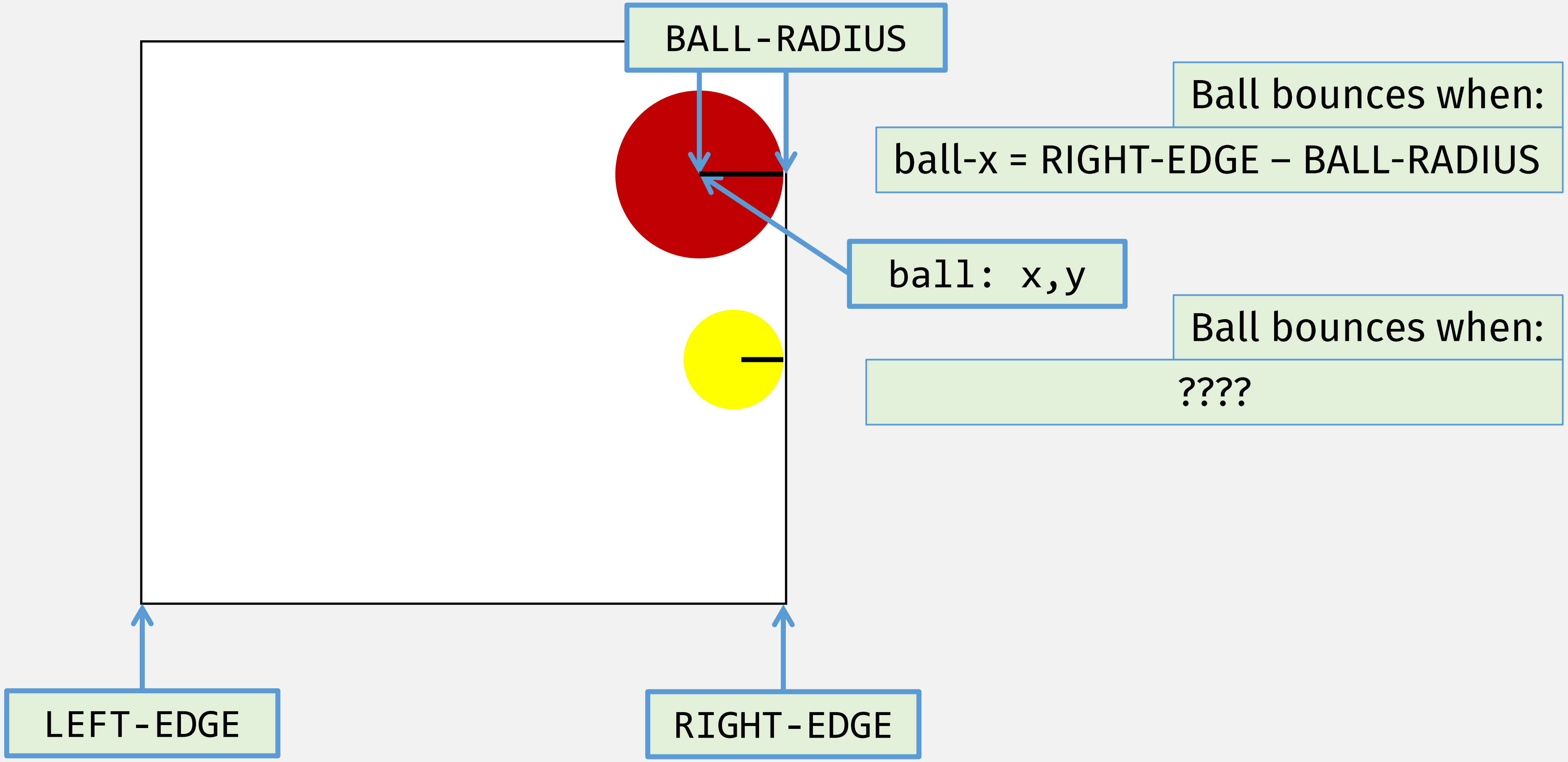
Logistics

- HW 5 in
  - ~~Part 1 due: **Sun 10/29 11:59 pm EST**~~
  - ~~Part 2 due: **Sun 11/5 11:59 pm EST**~~

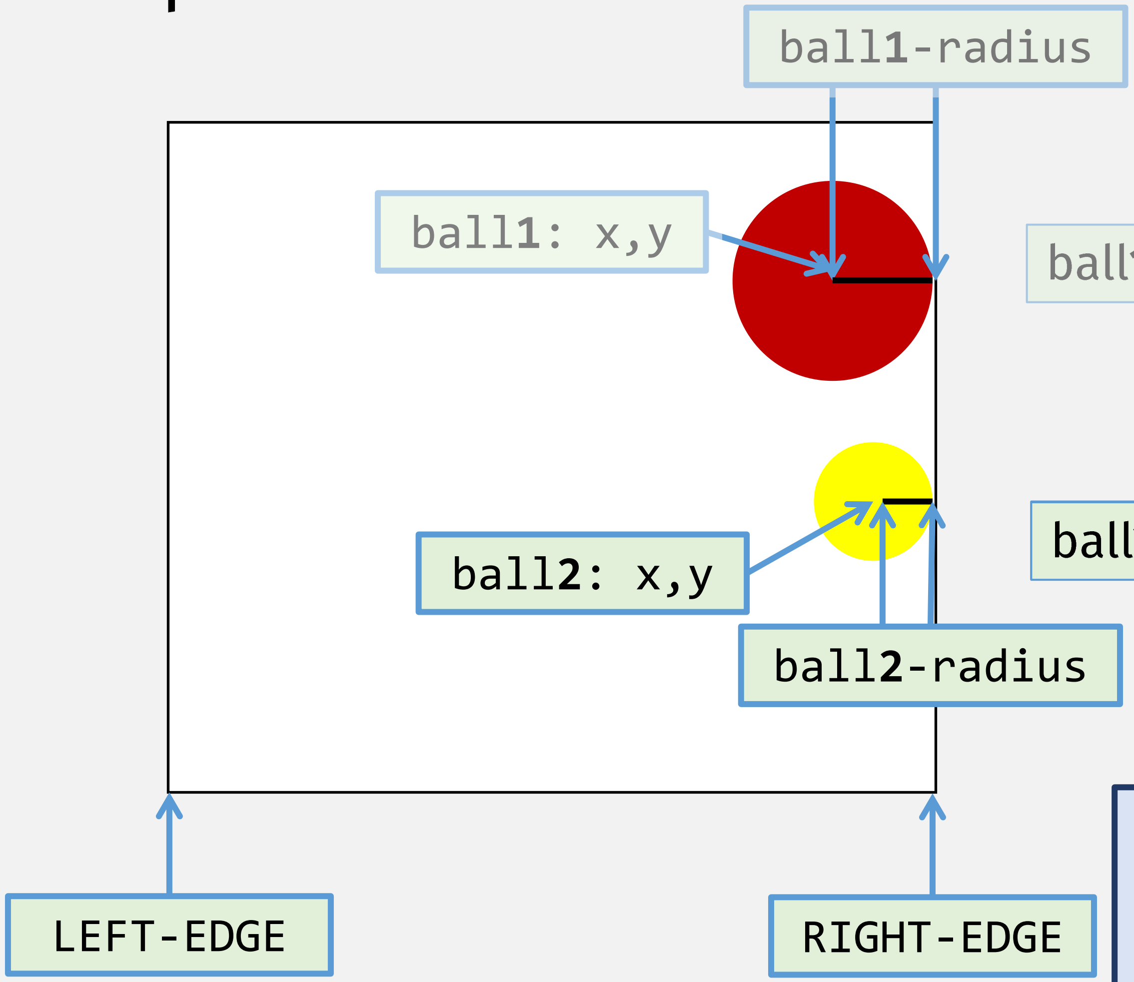
- HW 6 out
  - due: Sun 11/13 11:59 pm EST
  - `Editor`, again!



HW 4 Recap


BALL-RADIUS
Ball bounces when:
ball-x = RIGHT-EDGE – BALL-RADIUS
ball: x,y
Ball bounces when:
????
LEFT-EDGE
RIGHT-EDGE

HW 4 Recap



Ball **#1** bounces when:

ball**1**-x = RIGHT-EDGE – ball**1**-radius

Ball **#2** bounces when:

ball**2**-x = RIGHT-EDGE – ball**2**-radius

HWs are designed such that, **in order to do well,** you must **<u>think about the data definitions</u>** <u>first</u>

Last Time

Data Definitions, With Invariants

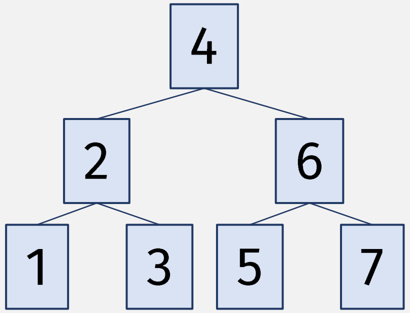
```
;; A BinarySearchTree<X> (BST) is a Tree<X>
;; where, if tree is a (node left data right):
;; Invariant 1: ∀ x ∈ left tree, x < node-data
;; Invariant 2: ∀ y ∈ right tree, y ≥ node-data
;; Invariant 3: left subtree must be a BST
;; Invariant 4: right subtree must be a BST
```

Last Time

Valid BSTs

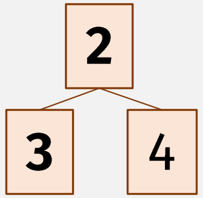
```
;; valid-bst? : Tree<X> -> Bool
;; Returns true if the given tree is a BST
;; (i.e., satisfies the BST invariants)
```

Valid

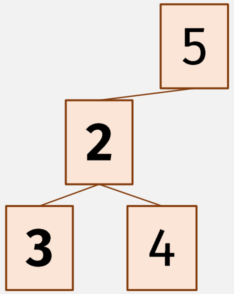


For every node,
☑ left subtree vals < node-data
☑ right subtree vals ≥ node-data
☑ left subtree is BST
☑ right subtree is BST

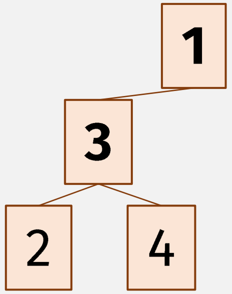
Not Valid



left value > root ☒



left values less than root ☑,
but left subtree not BST ☒



Left subtree is valid BST ☑,
but left values not less than root ☒

Last Time

Valid BSTs

```
;; valid-bst? : Tree<X> -> Bool
;; Returns true if the tree is a BST
```

```
;; A BinarySearchTree<X> (BST) is a Tree<X>
;; where, if tree is a node:
;; Invariant 1: ∀ x ∈ left tree, x < node-data
;; Invariant 2: ∀ y ∈ right tree, y ≥ node-data
;; Invariant 3: left subtree must be a BST
;; Invariant 4: right subtree must be a BST
```

```
(define (valid-bst? t)
  (cond
    [(empty? t) true]
    [else
     (and (tree-all? (curry > (node-data t)) (node-left t))
          (tree-all? (curry <= (node-data t)) (node-right t))
          (valid-bst? (node-left t))
          (valid-bst? (node-right t)))]))
```

cond that evaluates to boolean is boolean arithmetic!

```
(define (valid-bst? t)
  (or (empty? t)
      (and (tree-all? (curry > (node-data t)) (node-left t))
           (tree-all? (curry <= (node-data t)) (node-right t))
           (valid-bst? (node-left t))
           (valid-bst? (node-right t)))))
```

Last Time

One-pass `valid-bst?`

```
;; valid-bst/one-pass? : ??? Tree<X> -> Bool
;; Returns true if the tree is a BST
```

```
(define (valid-bst/one-pass? ??? t)
  (or (empty? t)
      (and (valid-bst/one-pass? ??? ??? (node-left t))
           (valid-bst/one-pass? ??? ??? (node-right t)))))
```

- Need extra argument(s) …
- … to keep track of allowed `node-data` values

More generally:

- Tree traversal processes each node <u>independently</u> …
- Extra argument allows "<u>remembering</u>" information from other nodes

Last Time

One-pass **valid-bst?**, Functional Style!

```
;; valid-bst/p? : Tree<X> (X -> Bool) -> Bool
;; Returns true if (p? (node-data t)) = true, and t is a BST
```

```
(define (valid-bst/p? p? t)
  (or (empty? t)
      (and (p? (node-data t))
           (valid-bst/p? (conjoin p? (curry > (node-data t)))
                         (node-left t))
           (valid-bst/p? (conjoin p? (curry <= (node-data t)))
                         (node-right t)))))
```

Extra argument, to "remember" information (valid node-data values) from other nodes

```
;; A BinarySearchTree<X> is a Tree
;; where, if tree is a node:
;; Inv1: ∀ x ∈ left, x < node-data
;; Inv2: ∀ y ∈ right, y ≥ node-data
;; Inv3: left subtree must be BST
;; Inv4: right subtree must be BST
```

"Extra argument" is called an **accumulator**

"conjunction" = AND

```
(conjoin p1? p2?)
==
(λ (x) (and (p1? x) (p2? x)))
```

```
(define (valid-bst? t)
  (valid-bst/p? (lambda (x) true) t))
```

Design Recipe For Accumulator Functions

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

1. *Specify* **accumulator:**
   - Name
   - Signature
   - Invariant
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

Design Recipe For Accumulators

Function needs "extra information" ...

```
;; valid-bst? : Tree<X> -> Bool
;; Returns true if t is a BST
(define (valid-bst? t)

  ;; accumulator p? : (X -> Bool)
  ;; invariant: if t = (node l data r), p? remembers valid vals
  ;; for node-data such that (p? (node-data t)) is always true
  (define (valid-bst/p? p? t)
    (or (empty? t)
        (and (p? (node-data t))
             (valid-bst/p? (conjoin p? (curry > (node-data t)))
                           (node-left t))
             (valid-bst/p? (conjoin p? (curry <= (node-data t)))
                           (node-right t)))))

  (valid-bst/p? (lambda (x) true) t))
```

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

2. *Define* internal "helper" fn with **accumulator** arg

3. *Call* "helper" fn, with initial **accumulator**

A List Example

Function needs "extra information" ...

```
;; 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)
                           (if (> (first lst) max-so-far)
                               (first lst)
                               max-so-far))]))

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

Helper needs signature, etc if different

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

2. *Define* internal "helper" fn with **accumulator** arg

A List 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)
                           (if (> (first lst) max-so-far)
                               (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 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)
                           (if (> (first lst) max-so-far)
                               (first lst)
                               max-so-far))]))

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

- Repo: **cs450f23/lecture17-inclass**
- File: **rev-with-acc-**<*your last name*>**.rkt**

In-class Coding 11/6 #1: 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**

A List 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/a (rest rst-lst)
                       (if (> (first rst-lst) max-so-far)
                              (first rst-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
```

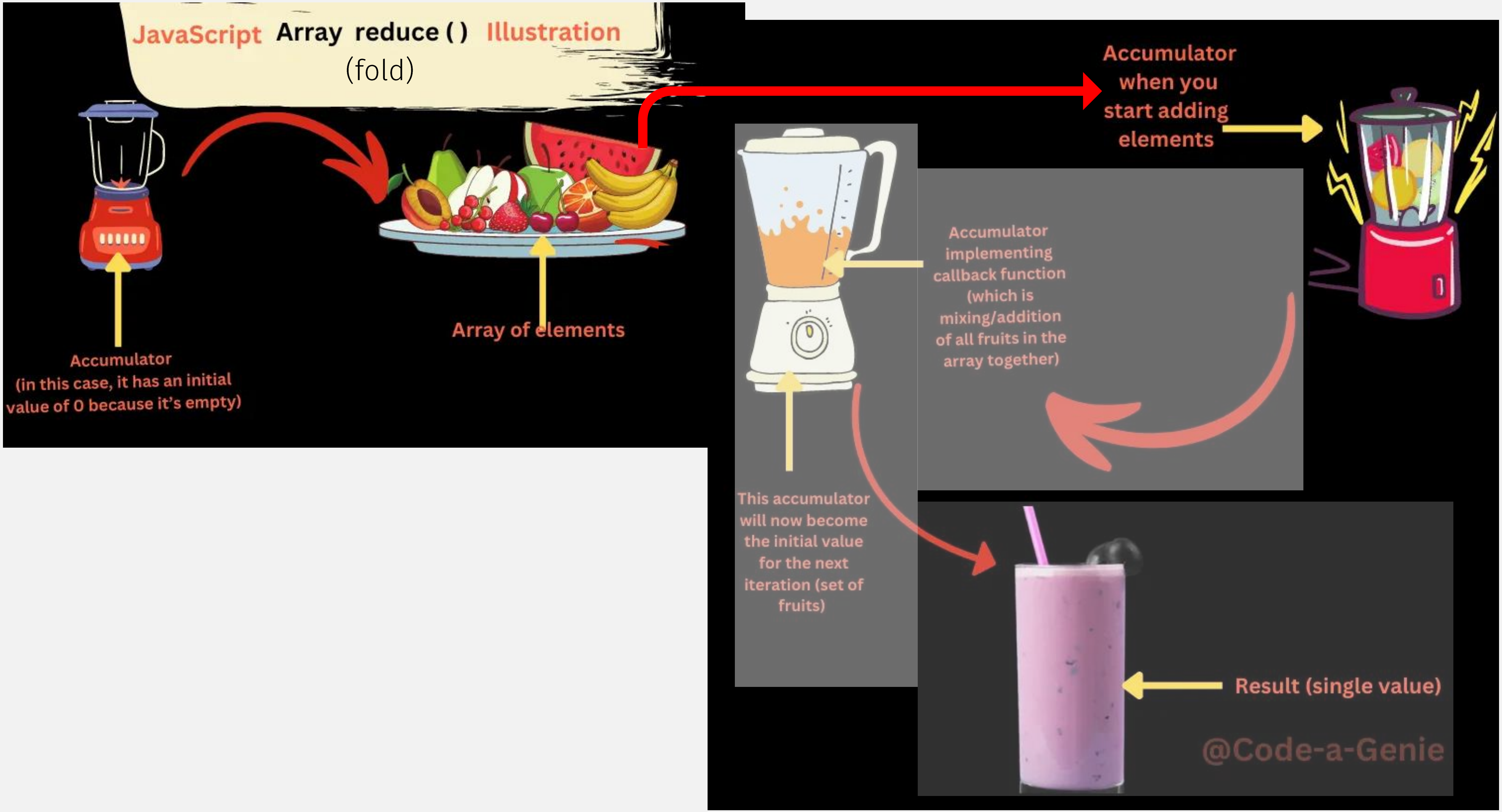
```
(define (foldl fn result-so-far lst)
  (cond
   [(empty? lst) result-so-far]
   [else (foldl fn (fn (first lst) result-so-far) (rest lst))]))
```

Accumulator!

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

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

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



JavaScript Array reduce ( ) Illustration
(fold)
Accumulator
(in this case, it has an initial
value of 0 because it's empty)
Array of elements
Accumulator
when you
start adding
elements
Accumulator
implementing
callback function
(which is
mixing/addition
of all fruits in the
array together)
This accumulator
will now become
the initial value
for the next
iteration (set of
fruits)
Result (single value)
@Code-a-Genie

- Repo: **cs450f23/lecture17-inclass**
- File: **tree-max**-<*your last name*>.rkt

In-class Coding 11/6 #2: Tree Max

```
;; tree-max : TreeNode<X> -> X
;; Returns the maximum value in a given (non-empty) tree node
(define (tree-max tree0)

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

  (define (tree-max/a tree acc ???)
    ???
  )

  (tree-max/a tree0 ???))
```

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

2. *Define* internal "helper" fn with **accumulator** arg

3. *Call* "helper" fn, with initial **accumulator**

No More Quizzes!