UMass Boston Computer Science

CS450 High Level Languages (section 2)

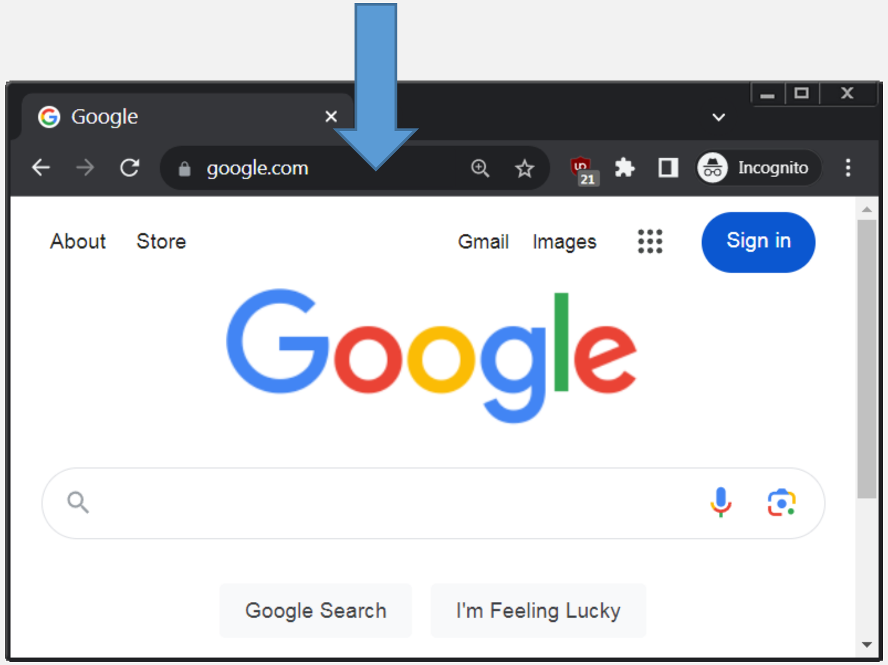
Accumulators

Monday, October 21, 2024

Logistics

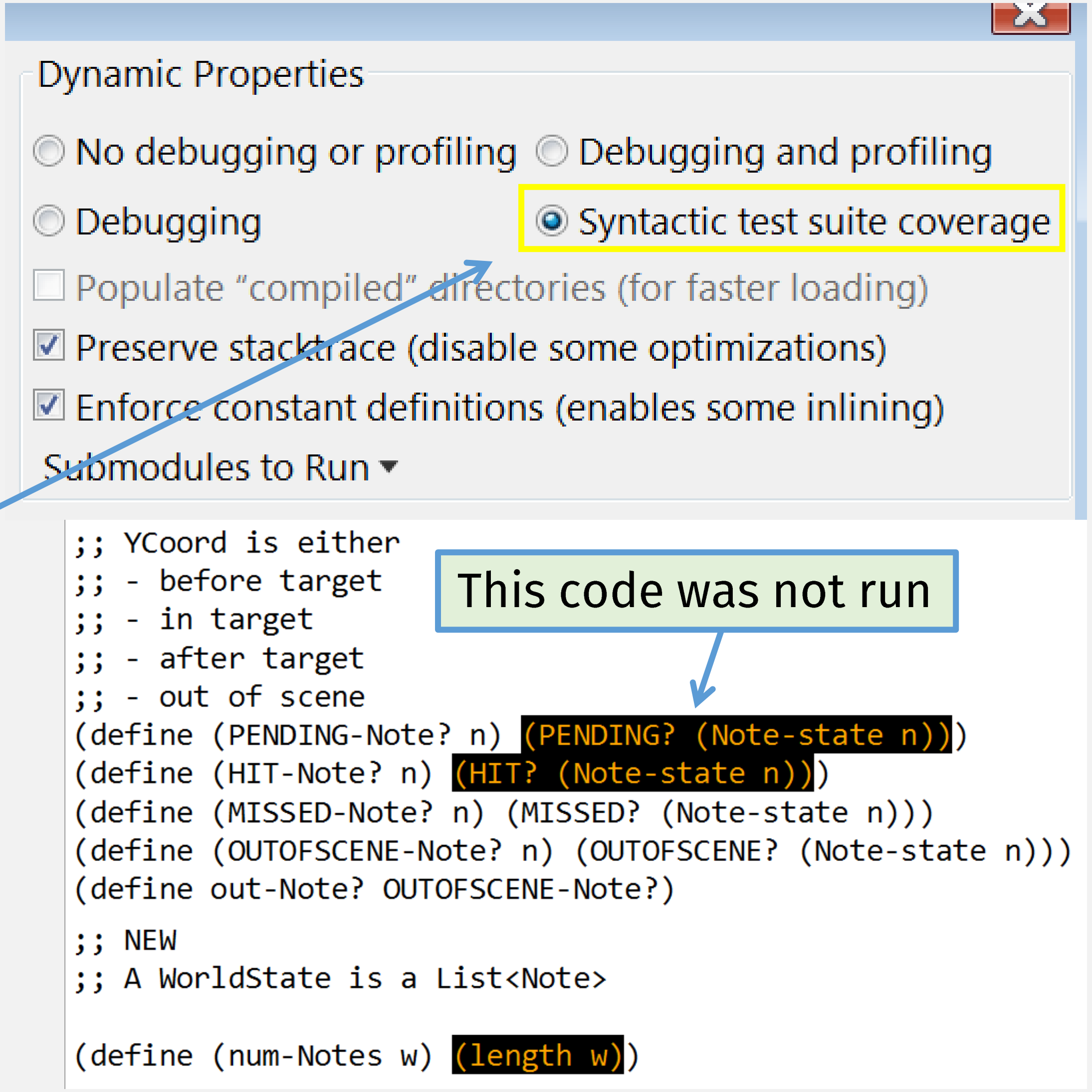
- HW 6 in
  - ~~Due: Mon 10/21 12pm (noon) EDT~~

- HW 7 out
  - Due: Mon 10/28 12pm (noon) EDT
  - A TextBox



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



Last Time

List (Recursive) Data Definition 1

```
;; A ListofInt is one of:
;; - empty
;; - (cons Int ListofInt)
```

Last Time

List (Recursive) Data Definition 1: Fn Template

Recursive call matches recursion in data definition

```
;; A ListofInt is one of:
;; - empty
;; - (cons Int ListofInt)
```

```
;; TEMPLATE for list-fn
;; list-fn : ListofInt -> ???
(define (list-fn lst)
  (cond
    [(empty? lst) ....]
    [(cons? lst) .... (first lst) ....
          .... (list-fn (rest lst)) ....]))
```

cond clause for each itemization item

Extract pieces of compound data

Last Time

Recursive List Fn Example 1: **`inc-list`**

Function design recipe:
1. Name
2. Signature
3. Description
4. Examples
5. Template

...

```
;; inc-list : ListofInt -> ListofInt
;; increments each list element by 1
(define (inc-lst lst)
  (cond
    [(empty? lst) ....]
    [(cons? lst) .... (first lst)  ....
          ....  (inc-lst (rest lst)) ....]))
```

```
(check-equal?
  (inc-list (list 1 2 3))
            (list 2 3 4))
```

Last Time

Recursive List Fn Example 1: **inc-list**

```
;; inc-list : ListofInt -> ListofInt
;; increments each list element by 1
(define (inc-lst lst)
  (cond
    [(empty? lst) empty]
    [(cons? lst) .... (first lst)  ....
          ....  (inc-lst (rest lst)) ....]))
```

Empty input produces empty output
(look at signature for help if needed)

Last Time

Recursive List Fn Example 1: **inc-list**

```
;; inc-list : ListofInt -> ListofInt
;; increments each list element by 1
(define (inc-lst lst)
  (cond
    [(empty? lst) empty]
    [else ....  (add1 (first lst)) ....
           ....  (inc-lst (rest lst)) ....]))
```

Call another function to process (first) (Int) list element

Last Time

Recursive List Fn Example 1: **inc-list**

```
;; inc-list : ListofInt -> ListofInt
;; increments each list element by 1
(define (inc-lst lst)
  (cond
    [(empty? lst) empty]
    [else (cons (add1 (first lst))
                (inc-lst (rest lst)))]))
```

Figure out how to “combine” with recursive call result
(look at signature for help if needed)

Last Time

List (Recursive) Data Definition 2

```
;; A ListofBall is one of:
;; - empty
;; - (cons Ball ListofBall)
```

Last Time

List (Recursive) Data Definition 2: Fn Template

Recursive call matches recursion in data definition?

```
;; A ListofBall is one of:
;; - empty
;; - (cons Ball ListofBall)
```

```
;; TEMPLATE for list-fn
;; list-fn : ListofBall -> ???
(define (list-fn lst)
  (cond
    [(empty? lst) ....]
    [(cons? lst) .... (first lst)  ....
          ....  (list-fn (rest lst)) ....]))
```

cond clause for each itemization item?

Extract pieces of compound data?

Last Time

Recursive List Fn Example 2: `next-world`

Function design recipe:
1. Name
2. Signature
3. Description
4. Examples
5. Template

…

```
;; next-world: ListofBall -> ListofBall
;; Updates position each ball by one tick
(define (next-world lst)
  (cond
    [(empty? lst) ....]
    [(cons? lst) .... (first lst)  ....
          ....  (next-world (rest lst)) ....]))
```

Last Time

Recursive List Fn Example 2: `next-world`

```
;; next-world: ListofBall -> ListofBall
;; Updates position each ball by one tick
(define (next-world lst)
  (cond
    [(empty? lst) empty]
    [(cons? lst) .... (first lst)  ....
         ....  (next-world (rest lst)) ....]))
```

Empty input produces empty output
(look at signature for help if needed)

Last Time

Recursive List Fn Example 2: `next-world`

```
(check-equal? (next-world (list (make-ball 0 0 1 1)))
                          (list (next-ball (make-ball 0 0 1 1))))
```

```
;; next-world: ListofBall -> ListofBall
;; Updates position each ball by one tick
(define (next-world lst)
  (cond
    [(empty? lst) empty]
    [else ....  (??? (first lst)) ....
          ....  (next-world (rest lst)) ....]))
```

Call another function to process (`first`) list element?

Ball

Recursive List Fn Example 2: next-world

Last Time

```
;; next-world: ListofBall -> ListofBall
;; Updates position each ball by one tick
(define (next-world lst)
  (cond
    [(empty? lst) empty]
    [else ....  (next-ball (first lst)) ....
          ....  (next-world (rest lst)) ....]))
```

Call another function to process (first) (Ball) list element

Last Time

Recursive List Fn Example 2: `next-world`

```
;; next-world: ListofBall -> ListofBall
;; Updates position each ball by one tick
(define (next-world lst)
  (cond
    [(empty? lst) empty]
    [else (cons (next-ball (first lst))
                (next-world (rest lst)))]))
```

Figure out how to "combine" with recursive call result
(look at signature for help if needed)

Last Time

Comparison 1

Differences?

```
;; inc-lst: ListofInt -> ListofInt
;; Returns list with each element incremented
(define (inc-lst lst)
  (cond
    [(empty? lst) empty]
    [else (cons (add1 (first lst))
                (inc-lst (rest lst)))]))
```

```
;; next-world : ListofBall -> ListofBall
;; Updates position of each ball by one tick
(define (next-world lst)
  (cond
    [(empty? lst) empty]
    [else (cons (next-ball (first lst))
                (next-world (rest lst)))]))
```

Last Time

Abstraction: Common List Function #1

Make the difference a parameter of a (function) abstraction

```
(define (lst-fn1 fn lst)
  (cond
    [(empty? lst) empty]
    [else (cons (fn (first lst))
                (lst-fn1 (rest lst)))]))
```

Abstraction: Common List Function #1

```
;; lst-fn1: (?? -> ??) Listof?? -> Listof??
;; Applies the given fn to each element of given lst
```

```
(define (lst-fn1 fn lst)
  (cond
    [(empty? lst) empty]
    [else (cons (fn (first lst))
                (lst-fn1 (rest lst)))]))
```

Abstraction of Data Definitions

```
;; A ListofInt is one of
;; - empty
;; - (cons Int ListofInt)
```

```
;; A ListofBall is one of
;; - empty
;; - (cons Ball ListofBall)
```

Abstraction of Data Definitions

```
;; A ListofInt is one of
;; - empty
;; - (cons Int ListofInt)
```

```
;; A ListofBall is one of
;; - empty
;; - (cons Ball ListofBall)
```

```
;; A Listof<X> is one of
;; - empty
;; - (cons X Listof<X>)
```

parameter

Abstraction: Common List Function #1

NOTE: **textbook writes it like this**
(both are ok, just follow data definition)

```
;; lst-fn1: [X -> Y] [Listof X] -> [Listof Y]
;; Applies the given fn to each element of given lst
```

```
;; lst-fn1: (X -> Y) Listof<X> -> Listof<Y>
;; Applies the given fn to each element of given lst
```

```
(define (lst-fn1 fn lst)
  (cond
    [(empty? lst) empty]
    [else (cons (fn (first lst))
                (lst-fn1 (rest lst)))]))
```

Abstraction: Common List Function #1

```
;; lst-fn1: (X -> Y) Listof<X> -> Listof<Y>
;; Applies the given fn to each element of given lst
```

```
(define (lst-fn1 fn lst)
  (cond
    [(empty? lst) empty]
    [else (cons (fn (first lst))
                (lst-fn1 (rest lst)))]))
```

```
(define (inc-lst lst) (lst-fn1 add1 lst))
(define (next-world lst) (lst-fn1 next-ball lst))
```

Common List Function #1: **map**

```
;; map: (X -> Y) Listof<X> -> Listof<Y>
;; Applies the given fn to each element of given lst
```

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

```
(define (inc-lst lst) (map add1 lst)
(define (next-world lst) (map next-ball 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
```

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 <u>Accumulator</u> Functions

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

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

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))]))
```

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 (???? initial-lst) (???? 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
```

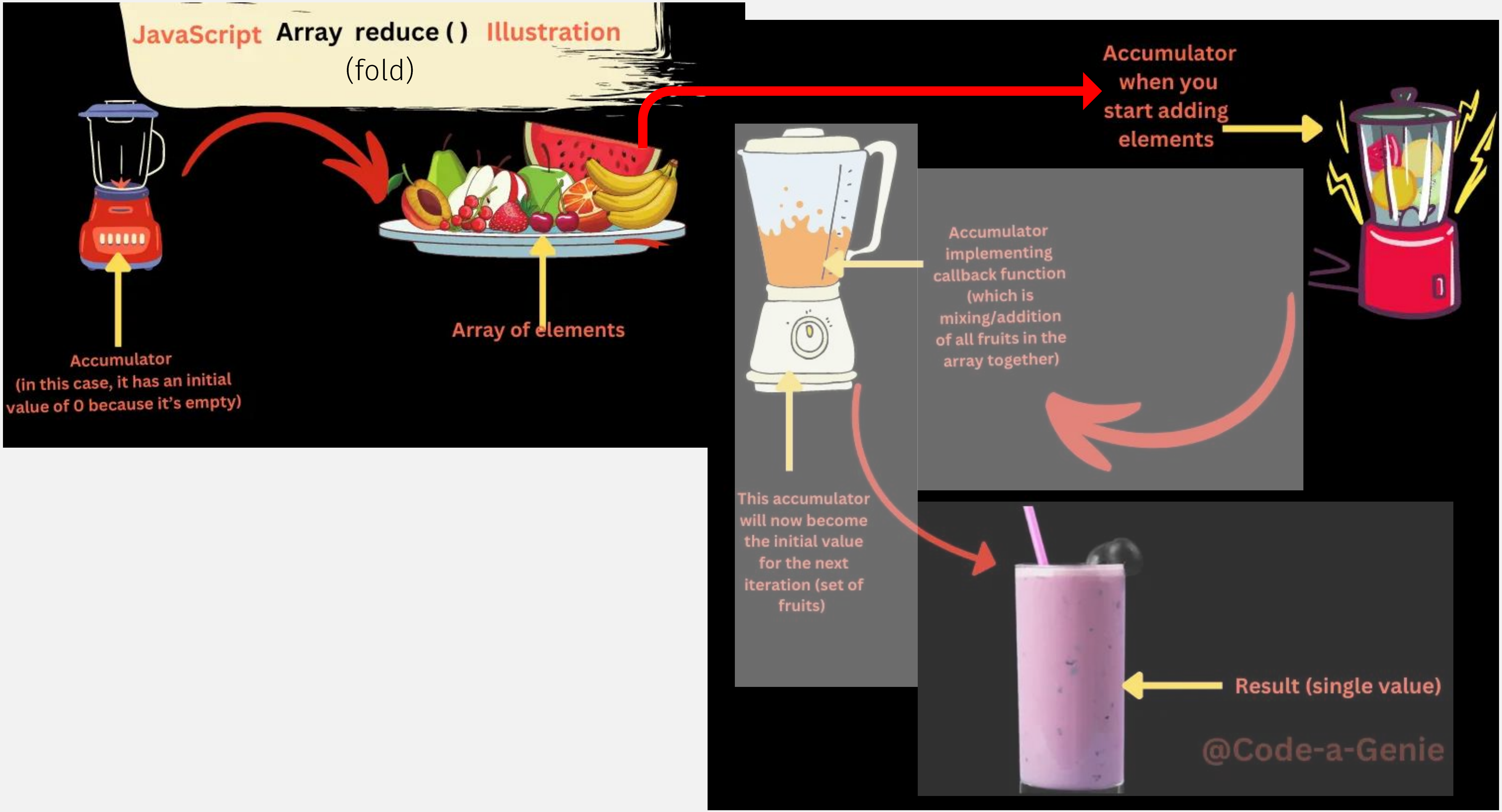
```
(define (foldl fn result-so-far lst)
  (cond
   [(empty? lst) result-so-far]
   [else (foldl fn (fn (first lst) result-so-far) (rest lst))]))
```

Accumulator!

```
;; 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: **cs450f24/in-class-10-21**
- File: **in-class-10-21**-<*Last*>-<*First*>.rkt

In-class Coding 10/21: 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**