CS135 Tutorial 06

Accumulative Recursion

Simple Recursion -> Accumulative Recursion

```
;; (sum-sr n) produces the sum of the numbers from 0 to n, inclusive.
;; sum-sr: Nat -> Nat
(define (sum-sr n)
  (cond [(zero? n) 0]
        [else (+ n (sum-sr (sub1 n)))]))

(check-expect (sum-sr 3) (+ 0 1 2 3))
```

sum-sr Trace

```
(sum-sr 10) =>
(+ 10 (sum-sr 9)) =>
(+ 10 (+ 9 (sum-sr 8))) =>
(+ 10 (+ 9 (+ 8 (sum-sr 7)))) =>*
(+ 10 (+ 9 (+ 8 (+ 7 (+ 6 (+ 5 (+ 4 (+ 3 (+ 2 (+ 1 (sum-sr 0))))))))))) =>
(+ 10 (+ 9 (+ 8 (+ 7 (+ 6 (+ 5 (+ 4 (+ 3 (+ 2 (+ 1 0)))))))))) =>
(+ 10 (+ 9 (+ 8 (+ 7 (+ 6 (+ 5 (+ 4 (+ 3 (+ 2 1))))))))) =>* 55
```

Simple Recursion -> Accumulative Recursion

```
;; (sum-ar n) produces the sum of the numbers from 0 to n, inclusive.
;; sum-ar: Nat -> Nat
(define (sum-ar n) (sum-ar/acc ...))

(check-expect (sum-ar 3) (+ 0 1 2 3))

(define (sum-ar/acc ...) ...)
```

Simple Recursion -> Accumulative Recursion

```
;; (sum-ar n) produces the sum of the numbers from 0 to n, inclusive.
;; sum-ar: Nat -> Nat
(define (sum-ar n) (sum-ar/acc n 0))

(check-expect (sum-ar 3) (+ 0 1 2 3))

;; (sum-ar/acc n sum-so-far) produces the sum from 0 to n + sum-so-far
;; sum-ar/acc: Nat Nat -> Nat
(define (sum-ar/acc n sum-so-far)
  (cond [(zero? n) sum-so-far]
        [else (sum-ar/acc (sub1 n) (+ n sum-so-far))]))

(check-expect (sum-ar/acc 2 3) 6)
```

Simple Recursion -> Accumulative Recursion

```
(sum-ar 10) =>
(sum-ar/acc 10 0) =>
(sum-ar/acc 9 10) =>
(sum-ar/acc 8 19) =>
(sum-ar/acc 7 27) =>
(sum-ar/acc 6 34) =>
(sum-ar/acc 5 40) =>
(sum-ar/acc 4 45) =>
(sum-ar/acc 3 49) =>
(sum-ar/acc 2 52) =>
(sum-ar/acc 1 54) =>
(sum-ar/acc 0 55) =>
55
```

Simple Recursion -> Accumulative Recursion

	Advantages	Disadvantages
Simple Recursion		
Accumulative Recursion		

Simple Recursion -> Accumulative Recursion

	Advantages	Disadvantages
Simple Recursion	• No helper required. • Shorter • Easier to understand • Easier to reason about	• Deferred calculations
Accumulative Recursion	• No deferred calculations	• Needs a helper • Longer • Harder to understand • Hard to reason about

span

```
;; (span lon) produces the difference between the largest and smallest
;; values in lon.
;; span (ne-listof Num) -> Num
(define (span lst) ...)


(check-expect (span (list 1)) 0)
(check-expect (span (list 1 3 3 3)) 2)
(check-expect (span (list 3 1 5 10 0)) 10)
```

span-v1

```
(define (span-v1 lon)
  (- (max-lst lon) (min-lst lon)))

;; (max-list lon) produces the largest number in lon.
;; max-list: (ne-listof Num) -> Num
(define (max-lst lon)
  (cond [(empty? (rest lon)) (first lon)]
        [else (max (first lon) (max-lst (rest lon)))]))

(define (min-lst lon)
  (cond [(empty? (rest lon)) (first lon)]
        [else (min (first lon) (min-lst (rest lon)))]))
```

span-v2

```
;; (span lon) produces the difference between the largest and smallest
;; values in lon.
;; span (ne-listof Num) -> Num
(define (span lon)
  (span/acc ...))


(define (span/acc ...)
  ...)


(check-expect (span (list 1)) 0)
(check-expect (span (list 1 3 3 3)) 2)
(check-expect (span (list 3 1 5 10 0)) 10)
```

span-v3

```
;; (span-v3 lon) produces the difference between the largest and smallest
;; values in lon.
;; span-v3: (ne-listof Num) -> Num
(define (span-v3 lon)
  (span/acc (rest lon) (first lon) (first lon)))

(define (span/acc lon min-so-far max-so-far)
  (cond [(empty? lon) (- max-so-far min-so-far)]
        [else (span/acc (rest lon)
                        (min min-so-far (first lon))
                        (max max-so-far (first lon)))]))
```

mergesort

- We discussed merge in class. It consumes two sorted lists and merges them together to make one sorted list.

```
(define (merge lon1 lon2)
  (cond [(and (empty? lon1) (empty? lon2)) empty]
        [(and (empty? lon1) (cons? lon2)) lon2]
        [(and (cons? lon1) (empty? lon2)) lon1]
        [(and (cons? lon1) (cons? lon2))
         (cond [(< (first lon1) (first lon2))
                (cons (first lon1) (merge (rest lon1) lon2))]
               [else (cons (first lon2) (merge lon1 (rest lon2)))])]))
```

- This is really, really, close to a sorting algorithm called mergesort: Divide the list in half, sort each half, and then merge them together.

mergesort

```
;; (mergesort lst) sorts lst in increasing order.
;; mergesort: (listof Num) -> (listof Num)
(define (mergesort lon)
  (cond [(or (empty? lon) (empty? (rest lon))) lon]
        [else (merge (mergesort (one-half lon))
                     (mergesort (other-half lon)))]))

(check-expect (mergesort (list 8 10 3 1 2 4 7 5 6 9))
              (list 1 2 3 4 5 6 7 8 9 10))
```

mergesort-v2

Partition the list with one pass by using two accumulators.

```
;; (partition lst half other) produces half of the elements in lst and the
;; other half of the elements in lst.

(check-expect (partition (list 1 2)     empty empty) (list (list 1) (list 2)))
(check-expect (partition (list 1)       empty empty) (list (list 1) empty))
(check-expect (partition (list 1 2 3 4) empty empty)
              (list (list 3 1) (list 4 2)))

;; partition: (listof X) (listof X) (listof X) -> (list (listof X) (listof X))
(define (partition lst half other) ...)
```