- Contract for take procedure:

; Signature: (take lst pos) -> lst

; Type: Procedure

; Purpose: Returns a new list whose elements are the first pos elements of lst or lst itself if lst is shorter than pos

; Pre-conditions: lst is a list and pos is a non-negative integer

; Tests:

(check-equal? (take (list 1 2 3) 2) '(1 2))

(check-equal? (take '() 2) '())

(check-equal? (take-map (list 1 2 3) (lambda (x) (* x x)) 2) '(1 4))

(check-equal? (take-map (list 1 2 3) (lambda (x) (* x x)) 4) '(1 4 9))

(check-equal? (take-filter (list 1 2 3 4) (lambda (x) (> x 1)) 2) '(2 3))

(check-equal? (take-filter (list 1 2 3) (lambda (x) (> x 3)) 2) '())

- Contract for take-map procedure:

; Signature: (take-map lst func pos) -> lst

; Type: Procedure

; Purpose: Returns a new list whose elements are the first pos elements of lst mapped by func or lst itself if lst is shorter than pos

; Pre-conditions: lst is a list, pos is a non-negative integer, and func is a function that takes one argument

; Tests:

(check-equal? (take-map (list 1 2 3) (lambda (x) (* x x)) 2) '(1 4))

(check-equal? (take-map (list 1 2 3) (lambda (x) (* x x)) 4) '(1 4 9))

- Contract for take-filter procedure:

; Signature: (take-filter lst pred pos) -> lst

; Type: Procedure

; Purpose: Returns a new list whose elements are the first pos elements of lst that satisfy pred or lst itself if the number of elements that satisfy pred is less than pos

; Pre-conditions: lst is a list, pos is a non-negative integer, and pred is a function that takes one argument and returns a boolean value

; Tests:

(check-equal? (take-filter (list 1 2 3 4) (lambda (x) (> x 1)) 2) '(2 3))

(check-equal? (take-filter (list 1 2 3) (lambda (x) (> x 3)) 2) '())

- Contract for sub-size procedure:

; Signature: (sub-size l size)

; Type: Procedure

; Purpose: Returns a new list of all the sublists of 'l' of length 'size'.

; Pre-conditions: 'l' is a list and 'size' is a non-negative integer less than or equal to the length of 'l'.

; Tests:

(check-equal? (sub-size '() 0) '(()))

(check-equal? (sub-size (list 1 2 3) 3) '((1 2 3)))

(check-equal? (sub-size (list 1 2 3) 2) '((1 2) (2 3)))

(check-equal? (sub-size (list 1 2 3) 1) '((1) (2) (3)))

- Contract for sub-size-map procedure:

; Signature: (sub-size-map l f size)

; Type: Procedure

; Purpose: Returns a new list of all the sublists of 'l' of length 'size' that all their elements are mapped by 'f'.

; Pre-conditions: 'l' is a list, 'f' is a function that takes an element of 'l' and returns a new value, and 'size' is a non-negative integer less than or equal to the length of 'l'.

; Tests:

(check-equal? (sub-size-map '() (lambda (x) (+ x 1)) 0) '(()))

(check-equal? (sub-size-map (list 1 2 3) (lambda (x) (+ x 1)) 3) '((2 3 4)))

(check-equal? (sub-size-map (list 1 2 3) (lambda (x) (+ x 1)) 2) '((2 3) (3 4)))

(check-equal? (sub-size-map (list 1 2 3) (lambda (x) (+ x 1)) 1) '((2) (3) (4)))

- Contract for root procedure:

; Signature: (root tree)

; Type: Procedure

; Purpose: Returns the value of the root of the binary tree represented by the given list.

; Pre-conditions: tree must be a valid list representation of a binary tree in L3, following the conventions described in the prompt.

; Tests: (root '(1 (#t 3 4) 2)) => 1 (root '(#t)) => error: empty tree

- Contract for left procedure:

; Signature: (left tree)

; Type: Procedure

; Purpose: Returns the left subtree of the binary tree represented by the given list, or an empty list if there is no left son.

; Pre-conditions: tree must be a valid list representation of a binary tree in L3, following the conventions described in the prompt.

; Tests: (left '(1 (#t 3 4) 2)) => '(#t 3 4) (left '(1 #t 2)) => '()

- Contract for right procedure:

; Signature: (right tree)

; Type: Procedure

; Purpose: Returns the right subtree of the binary tree represented by the given list, or an empty list if there is no right son.

; Pre-conditions: tree must be a valid list representation of a binary tree in L3, following the conventions described in the prompt. ; Tests: (right '(1 (#t 3 4) 2)) => '(2) (right '(1 2 #t)) => '()

- Contract for count-node procedure:

; Signature: (count-node tree val)

; Type: Procedure

; Purpose: Returns the number of nodes in the binary tree represented by the given list that have the same value as the given atomic val.

; Pre-conditions: tree must be a valid list representation of a binary tree in L3, following the conventions described in the prompt. ; Tests: (count-node '(1 (#t 3 #t) 2) #t) => 2 (count-node '(1 (#t 3 #t) 2) 4) => 0

- Contract for mirror-tree procedure:

; Signature: (mirror-tree tree)

; Type: Procedure

; Purpose: Returns a new binary tree that is the mirror image of the binary tree represented by the given list.

; Pre-conditions:tree must be a valid list representation of a binary tree in L3, following the conventions described in the prompt.

; Tests: (mirror-tree '(1 (#t 3 #t) 2)) => '(1 2 (#t 4 3)) (mirror-tree '(#t)) => '(#t)

- Contract for make-ok procedure:

; Signature: make-ok : value -> ok

; Type: Procedure

; Purpose: Creates an ok structure for the given value

; Pre-conditions: The input value can be any Scheme value

; Tests:

(make-ok 1) ;=> #<ok>

(result? (make-ok 1)) ;=> #t

(result->val (make-ok 1)) ;=> 1

- Contract for make-error procedure:

; Signature: make-error : string -> error

; Type: Procedure

; Purpose: Creates an error structure for the given error message

; Pre-conditions: The input error message must be a string

; Tests:

(make-error "some error message") ;=> #<error>

(error? (make-error "some error message")) ;=> #t

(result? (make-error "some error message")) ;=> #t

(result->val (make-error "some error message")) ;=> "some error message"

- Contract for ok? procedure:

; Signature: ok? : any -> boolean

; Type: Predicate

; Purpose: Checks if the input is an ok structure

; Pre-conditions: The input can be any Scheme value

; Tests:

(ok? #<ok>) ;=> #t

(ok? #<error>) ;=> #f

(ok? 1) ;=> #f

- Contract for error? procedure:

; Signature: error? : any -> boolean

; Type: Predicate

; Purpose: Checks if the input is an error structure

; Pre-conditions: The input can be any Scheme value

; Tests:

(error? #<error>) ;=> #t

(error? #<ok>) ;=> #f

(error? "some error message") ;=> #f

- Contract for result? procedure:

; Signature: result? : any -> boolean

; Type: Predicate

; Purpose: Checks if the input is a result structure

; Pre-conditions: The input can be any Scheme value

; Tests:

(result? #<ok>) ;=> #t

(result? #<error>) ;=> #t

(result? "some value") ;=> #f

- Contract for result->val procedure:

; Signature: result->val : result -> value or error

; Type: Procedure

; Purpose: Returns the value represented by the result, or the error message if it's an

; error structure

; Pre-conditions: The input must be a result structure

; Tests:

(result->val #<ok>) ;=> 1

(result->val #<error>) ;=> "some error message"

(result->val "some value") ;=> "Error: not a result"

- Contract for bind procedure:

; Signature: bind : (value -> result) -> (result -> result or error)

; Type: Procedure

; Purpose: Takes a function that takes a value and returns a result, and returns a new function that takes a result and returns the activation of the input function on the value of the result, or an error structure if the input result is an error or not a result

; Pre-conditions: The input function must take a value and return a result, the input result must be a result structure

; Tests:

(define inc-result (bind (lambda (x) (make-ok (+ x 1)))))

(define ok (make-ok 1))

(result->val (inc-result ok)) ;=> 2

(define error (make-error "some error message"))

(result->val (inc-result error)) ;=> "some error message"

(result->val (inc-result "not a result")) ;=> "Error: not a result"