```
; [X] N [N -> X] -> [List-of X]
; constructs a list by applying f to 0, 1, ..., (sub1 n)
; (build-list n f) == (list (f 0) ... (f (- n 1)))
(define (build-list n f) ...)

; [X] [X -> Boolean] [List-of X] -> [List-of X]
; produces a list from those items on lx for which p holds
(define (filter p lx) ...)

; [X] [List-of X] [X X -> Boolean] -> [List-of X]
; produces a version of lx that is sorted according to cmp
(define (sort lx cmp) ...)

; [X Y] [X -> Y] [List-of X] -> [List-of Y]
; constructs a list by applying f to each item on lx
; (map f (list x-1 ... x-n)) == (list (f x-1) ... (f x-n))
(define (map f lx) ...)

; [X] [X -> Boolean] [List-of X] -> Boolean
; determines whether p holds for every item on lx
; (andmap p (list x-1 ... x-n)) == (and (p x-1) ... (p x-n))
(define (andmap p lx) ...)

; [X] [X -> Boolean] [List-of X] -> Boolean
; determines whether p holds for at least one item on lx
; (ormap p (list x-1 ... x-n)) == (or (p x-1) ... (p x-n))
(define (ormap p lx) ...)
```

Figure 95: ISL's abstract functions for list processing (1)

```
; [X Y] [X Y -> Y] Y [List-of X] -> Y
; applies f from right to left to each item in lx and b
; (foldr f b (list x-1 ... x-n)) == (f x-1 ... (f x-n b))
(define (foldr f b lx) ...)

(foldr + 0 '(1 2 3 4 5))
== (+ 1 (+ 2 (+ 3 (+ 4 (+ 5 0)))))
== (+ 1 (+ 2 (+ 3 (+ 4 5))))
== (+ 1 (+ 2 (+ 3 9)))
== (+ 1 (+ 2 12))
== (+ 1 14)

; [X Y] [X Y -> Y] Y [List-of X] -> Y
; applies f from left to right to each item in lx and b
; (foldl f b (list x-1 ... x-n)) == (f x-n ... (f x-1 b))
(define (foldl f b lx) ...)

(foldl + 0 '(1 2 3 4 5))
== (+ 5 (+ 4 (+ 3 (+ 2 (+ 1 0)))))
== (+ 5 (+ 4 (+ 3 (+ 2 1))))
== (+ 5 (+ 4 (+ 3 3)))
== (+ 5 (+ 4 6))
== (+ 5 10)
```

Figure 96: ISL's abstract functions for list processing (2)