ALF Review

Abstract list functions overview

Abstraction is the process of finding similarities or common aspects, and forgetting unimportant differences.

In lecture you covered a number of the abstract list functions that Racket has available. Below are the abstract list functions we will be talking about today.

```
;;build-list: Nat (Nat ⟶ Y) ⟶ (listof Y)
;;map: (X ⟶ Y) (listof X) ⟶ (listof Y)
;;filter: (X ⟶ Bool) (listof X) ⟶ (listof X)
;;foldr: (X Y ⟶ Y) Y (listof X) ⟶ Y
;;foldl: (X Y ⟶ Y) Y (listof X) ⟶ Y
;;sort: (listof X) (X X ⟶ Bool) ⟶ (listof X)
```

ALF: build-list

;;build-list Nat (Nat ⟶ Y) ⟶ (listof Y)

Constructs a list by applying f to the numbers between 0 and (- n 1).

(build-list n f) = (list (f 0) ... (f (- n 1)))

```
(build-list 3 identity) ;;⇒ (list 0 1 2)

(build-list 5 sqr) ;;⇒ (list 0 1 4 9 16)

(build-list 4 (lambda (x) (build-list x add1))) ;;⇒ (list empty (list 1) (list 1 2) (list 1 2 3))
```

ALF: map

```
;;map (X ⟶ Y) (listof X) ⟶ (listof Y)
```

Constructs a new list by applying a function to each item on one or more existing lists

```
(map f (list x-1 ... x-n)) = (list (f x-1) ... (f x-n))

(map add1 (list 1 2 3))

;;⇒ (list 2 3 4)

(map sqr (build-list 4 identity))

;;⇒ (list 0 1 4 9)

(map string-length (list "shake" "it" "off"))

;;⇒ (list 5 2 3)
```

ALF: filter

;;filter (X → Bool) (listof X) → (listof X)

Constructs a list from all those items on a list for which the predicate holds.

Examples:

```
(filter positive? (list −2 1 0 −3 4 −7 −4 150))
;;⇒ (list 1 4 150)
(filter not (list true false false true))
;;⇒ (list false false)

(define taylor-lyrics '("Blank Space"  "Shake it Off"  "Red"  "Fearless"  "Love Story"))
(define taylor? (lambda (x) (member? x taylor-lyrics))
(filter taylor? (list "Call me Maybe"  "Red"  "Party in the USA"  "Love Story")))
;;⇒ (list ”Red” ”Love Story”)
```

ALF: foldr

;;foldr (X Y ⟶ Y) Y (listof X) ⟶ Y

foldr consumes a function, a base case, and a list. foldr applies the given function to every element and the recursive rest of the list. At the final element it applies the function to that element and the base case.

```
(foldr f base (list x-1 ... x-n)) = (f x-1 ... (f x-n base))

(foldr f base (list x-1 ... x-n) (list y-1 ... y-n))= (f x-1 y-1 ... (f x-n y-n base))

(define lst (list item1 item2 item3))  (foldl function basecase lst)

(function item1 (function item2 (function item3 basecase)))
```

Examples:

```
(foldr + 0 (list 1 2 3 4 5))
;;⇒ 15
(foldr string-append ""  '("It's "  "like "  "I "  "got "  "this "  "music "  "in "  "my "  "mind"))
;;⇒ "It's like I got this music in my mind"
```

ALF: foldl

;;foldl (X Y → Y) Y (listof X) → Y

foldl is an abstract list function which consumes a function, an accumulator, and a list. foldl applies the given function to every element and the accumulator. At the first element it applies the function to that element and the initial accumulator.

(foldl f base (list x-1 ... x-n)) = (f x-n ... (f x-1 base))

(foldl f base (list x-1 ... x-n) (list x-1 ... x-n))= (f x-n y-n ... (f x-1 y-1 base))

```
(define lst (list item1 item2 item3))  (foldl function basecase lst)
```

```
(function item3 (function item2 (function item1 basecase)))
```

Examples:

```
(foldl + 0 (list 1 2 3 4 5))
;;⇒ 15
(foldl cons empty (list 1 2 3))
;;⇒ (list 3 2 1)
(foldl − −3 (list 1 −2 3))
;;⇒ 9
```

ALF: sort

;;sort (listof X) (X X → Bool) → (listof X)

sort is an abstract list function which consumes a function and a list. Sort uses the given function to compare elements in the list. If consumed function produces true if the first of two elements is less than the second. The produced list is re-ordered accordingly.

Examples:

```
(sort (list 5 1 4 2 3) <)
⇒ (list 1 2 3 4 5)
(sort (list "Josh" "Adam" "Andy" "Andrew" "Pearl" "Nick") string<?)
⇒ (list "Adam" "Andrew" "Andy" "Josh" "Nick" "Pearl")
(sort (list #\c #\a #\b) char<?)
⇒ (list #\a #\b #\c)
```