Abstract list functions: Filter, Map

Using filter: (filter even? alist)

- filter consumes a **predicate** specifying which elements of the list are to be **kept**.
- The predicate must be a **one-parameter** function producing a **boolean**, where the <u>type of the parameter is same as the type of the elements of the list</u>

Using map: (map sqr alist)

- map performs the operation of **transforming** a **list** element-by-element into another list of the **same length**.
- The function consumed by map must be a **one-parameter** function where the type of the parameter is the same as the type of the elements of the list.

The abstract list function foldr

- foldr (built-in) is short for “fold right”.
- It can be viewed as “folding” a list using the provided combine function, starting from the right-hand end of the list.
- If alist is (list x_1 x_2 . . . x_n), then by our intuitive explanation of foldr, the expression

(foldr f 0 alist) equivalent to (f x_1 (f x_2 (f ... (f x_n 0))))

base

base

combine function

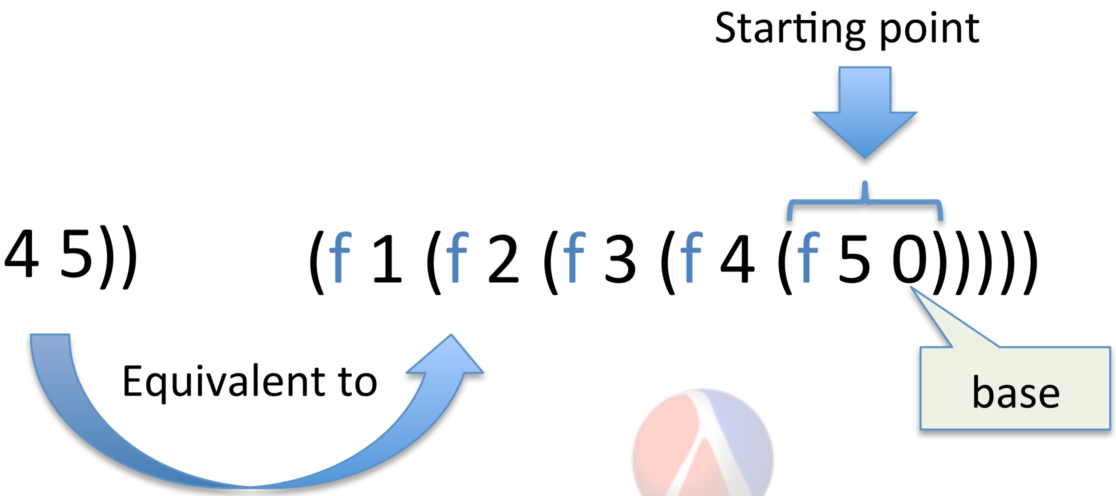
The abstract list function foldr (cont.)

The combine function provided to foldr consumes two parameters:

- **an item** in the list that foldr consumes and
- the **result** of applying foldr to the rest of the list.

Example:

```
(define (f x r)
        (+ x r))
(foldr f 0 (list 1 2 3 4 5))
```



Starting point

```
(f 1 (f 2 (f 3 (f 4 (f 5 0)))))
```

Equivalent to

base

The abstract list function foldr (cont.)

```
(define
(product-of-numbers alist)
(cond
[(empty? alist) 1]
[else
(* (first alist) (product-of-
numbers (rest alist)))]))
```

- Similarities
- Differences

```
(define
(concat-firsts alist)
(cond
[(empty? alist) ""]
[else
(cond
[(string=? "" (first alist))
(concat-firsts (rest alist))]
[else (string-append (substring
(first alist) 0 1)
(concat-firsts (rest alist)))])]))
```

The abstract list function foldr (cont.)

```
(define (foldr combine base alist)
    (cond
        [(empty? alist) base]
        [else (combine (first alist)
                       (foldr combine base (rest alist)))]))
```

Tracing foldr

(foldr f 0 (list 3 6 5))

⇒ (f 3 (foldr f 0 (list 6 5)))

⇒ (f 3 (f 6 (foldr f 0 (list 5)))

⇒ (f 3 (f 6 (f 5 (foldr f 0 empty)))

⇒ (f 3 (f 6 (f 5 0))) ⇒ . . .

Intuitively, the effect of the application (foldr f b (list x_1 x_2 . . . x_n)) is to compute the value of the expression (f x_1 (f x_2 (... (f x_n b) ...))).

Practical Exercise

Write a function *get-total* that produces the total value in a list of numbers.

```
;; write a function total that produces the sum of the
;; numbers in lon
;; total: (listof Num) -> Num
(define (get-total alist)
                  (foldr + 0 alist)
                                          )
```

Additional Practical Exercise

Write a function *m-positive* that produces the multiplication of all positive elements of a list of numbers.

```
(define lon (list 1 -2 4 -5 9))
(define (m-positive lon)
  (foldr * 1
        (filter positive? lon)
                              ))
```

Another Practical Exercise

```
(define (f item)
    (or (string? item) (boolean? item)))
(define (g n)
    (cond [(even? n) (sqr n)] [else (* n 2)]))
(define (h n s)
(string-append (substring (number->string n) 0 1) s))

a)  (filter f (list 4 "taco" #\r true "salad" 17 false #\c 8))
                                    (list "taco” true "salad” false)
b) (map g (list 4 8 7 1 3))         (list 16 64 14 2 6)
c) (foldr h "" (list 16 205 36 5))  "1235"
```

Using foldr to produce lists

- Remember:

  (foldr * 1 alist)) equivalent to (* x_1 (* x_2 (* ... (* x_n 1)...)))
- The functions we provide to foldr can also produce cons expressions, since these are also values.
- How? (cons element-from-list rest-of-list)
- Example: using foldr for negate-list.
- neg-combine takes the element, negates it, and conses it onto the result of the recursive call.

Function neg-combine

```
;; neg-combine: Num (listof Num) → (listof Num)
 (define (neg-combine item result-on-rest)
                                  (cons (− item) result-on-rest))

;; negate-list: (listof Num) → (listof Num)
(define (negate-list alist)
                                  (foldr neg-combine empty alist))
```

foldr can be used to implement map, filter, and other abstract list functions.

Boolean functions and foldr

(list 1 2 -3 4) => (list true true false true)

- (map positive? (list 1 2 -3 4))
- To check whether a predicate function produces true for every element in a list alist, we might be tempted to try:

  (foldr and true (map positive? alist))
- Problem: and is not a function, but a **special form**, and this produces an error.
- Solution: Racket provides andmap, which can be used like this: (andmap positive? alist)
- For the same reason, ormap is provided.

Foldr vs. Template

- Anything that can be done with the **list template** can be done using foldr, without explicit recursion.
- Does that mean that the list template is **obsolete**?
- No.
- Experienced Racket programmers still use the list template, for reasons of **readability** and **maintainability**.

Additional Exercise

a.
```
(define (Z? x)
        (and (> x 3) (< x 8)))
  (map sqr  (filter Z? (list 5 6 0 -4 12 9 -7)))      => (list 25 36)
```

b.
```
(define (f x)
        (cond [(> x 8) (* x 2)] [else (* x 3)]]))
  (foldr - 1 (map f (list 5 6 1 -4 12 9 -7)))       =>  -4
```

c.
```
(define (w x)
        (number->string (foldr + 0 x)))
  (map w (list (list 21 23 30) (list 40 50 60))) => (list "74" "150“)
```

Another Additional Exercise

Write a function even-length that consumes a list of strings, los, and produces a list of boolean values where the i-th member is true if the i-th string in los is of even length, and false otherwise.

For example:

```
 (even-length (list "yes" "No" "what" "maybe"))
                                      => (list false true true false)
```

For this question you must use only abstract list functions. You may not use explicit recursion.

Another Additional Exercise (cont.)

```
2.  (define (f item)
3.  (even?
          (string-length item)))
1.  (map f (list "yes" "No" "what" "maybe"))
```

More ….

We have a the following list of grades:

```
(define grades
(list (make-grade 'D 62) (make-grade 'C 79) (make-grade 'A 93) (make-grade
'B 84)   (make-grade 'F 57) (make-grade 'F 38) (make-grade 'A 90)
(make-grade 'A 95) (make-grade 'C 76) (make-grade 'A 90) (make-grade 'F
55) (make-grade 'C 74) (make-grade 'A 92) (make-grade 'B 86) (make-grade
  'F 43) (make-grade 'C 73)))
```

With the following structure defination:

```
;; A Grade is: (make-grade Symbol Number)
(define-struct grade (letter num))
```

Trying to find the biggest number in this list of grades by using abstract list functions.