```racket
#lang racket

(define numbers (list 2 20 3 5 10))

(define words (list "cat" "penguin" "fish"))

(define (sum-numbers lst)
  (if (= (length lst) 1)
      (first lst)
      (+ (first lst)
         (sum-numbers (rest lst)))))

(sum-numbers numbers)

(define (append-strings lst)
  (if (= (length lst) 1)
      (first lst)
      (string-append (first lst)
                     (append-strings (rest lst)))))

(append-strings words)

(define (append-strings-tail lst res)
  (if (= (length lst) 1)
      (string-append (first lst) res)
      (append-strings-tail (rest lst) (string-append (first lst) res))))

(append-strings-tail words "")

(define (sum-numbers-fold lst)
  (foldl (lambda (x y)(+ x y))
         0
         lst))

(sum-numbers-fold numbers)

(define (append-strings-fold lst)
  (foldl (lambda (x y)(string-append y x))
         ""
         lst))

(append-strings-fold words)

(define (count-cats lst)
  (foldl (lambda (str total)(if (equal? str "cat")
                                (+ 1 total)
                                total))
         0
         lst))

(count-cats words)
```

```

(define (append-strings-foldr lst)
  (foldr (lambda (x y)(string-append x y))
         ""
         lst))

(append-strings-foldr words)

(define (prod-numbers  lst)
  (foldl (lambda (x y) (* x y)) 1 lst))

(prod-numbers numbers)

(define (my-and lst)
  (foldl (lambda (x y)(if x
                          (if y #t #f)
                          #f))
         #f
         lst))

(my-and (list #t (= 0 0)))

(define (my-xor lst)
  (foldl (lambda (x res)(if x
                            (if res #f #t)
                            #t))
         #f
         lst))

(define notxor-bools (list #t #t #t #f #f))

(define xor-bools (list #f #f #t #f #f))

(my-xor xor-bools)

(my-xor notxor-bools)

(define (my-xor-2 lst)
  (if (= (length (filter (lambda(x)x) lst)) 1)
      #t
      #f))

(my-xor-2 xor-bools)

(my-xor-2 notxor-bools)

 (define (xor-fold lst)
   (foldl (lambda (x y)(if x
                           (if (first y)
                               (list #f 1)
                               (list #t (first (rest y))))
```

```racket
                             y))
         (list #f 0)
         lst))

  (xor-fold xor-bools)
(xor-fold notxor-bools)

(define (my-xor-3 lst)
  (let ((res (xor-fold lst)))
    (if (and (first res) (= (first (rest res)) 0))
        #t
        #f)))

(my-xor-3 notxor-bools)

(my-xor-3 xor-bools)

```