# CSI 3350
# Fall 2019 Homework 03

Suggested reading:
STRUCTURE AND INTERPRETATION OF COMPUTER PROGRAMS, SECTIONS 1.1.6, 1.1.7, 1.3

IMPORTANT:
Always check hw03-test.rkt to see the full usage of the functions. Not all use cases are listed in this file.
--
The previous homework introduced you to recursion and a higher order procedure called map. In this homework we will see how certain patterns of recursion can be expressed using a small set of higher order procedures.
--

The next 3 problems ask you to implement functions that can be found in the standard library, http://docs.racket-lang.org/reference/index.html , with the exception that yours do not have to be able to support multiple lists as arguments.

You are NOT allowed to use these functions in the solutions for the next 3 problems. However, you are encouraged to play with them to better understand their behavior.

The names of the standard library functions do not have the trailing "-335".

What to submit:
    write your solution into hw03-answer-sheet.rkt, then
    rename it to hw03-yourlastname.rkt, and
    submit hw03-yourlastname.rkt ONLY.
=========================================================================
1. [20p] fold
1.a [10p]

Implement a function foldl-335 ("fold left") with the following signature:

```
> (define (foldl-335 op default-element lst) 'todo)
```

Input:
op: is a two argument function
default-element: is expected to be the default element of the operator
                 function (e.g. 0 for +, 1 for *) but technically it
                 can be any value.
lst: a list of elements

```
Result:
      a value computed by successively applying the "op" function
      to each element of the list and the result of previous "op"
      function calls (where no such result exists the default-element
      is used)

> (foldl-335 + 0 '(1 2 3 4))
10

;;string-append is a library function that takes an arbitrary number
;;of strings and concatenates them.
> (string-append "one-string " "---" "two-string")
"one-string ---two-string"

> (foldl-335 string-append "" (list "!!!" "42"  "is "
                           "answer " "the " ))
"the answer is 42!!!"

;;here we see that if we use 0 as the zero element of the
;;multiplication function the expression will yield the
;;value 0.
> (foldl-335 * 0 '(1 2 3 4))
0

;;this expression computes the sum of squares of each element.
> (foldl-335 (lambda (x y) (+ (expt x 2) y)) 0 '(1 2 3 4 5))
55
```

As you can see from this example the operator is always applied in the same order:

```
> (foldl-335 - 0 '(1 2 3 4))
2

> (foldl-335 - 0 '(4 3 2 1))
-2
```

Since the minus operation is not commutative it gives different results depending on the ordering of the elements.

NOTE: you can NOT use the already existing Scheme function foldl or folder anywhere in your solution code.

--
1.b [10p]

Implement a function foldr-335 with the same signature as the one from 1a, with the difference that the operator is applied in the opposite order as foldl.

```
> (foldr-335 - 0 '(1 2 3 4))
```

```
-2
> (foldr-335 - 0 '(4 3 2 1))
2

> (foldr-335 string-append "" (list "the " "answer " "is "  “35”
                               "!!!"))
"the answer is 35!!!”
```

NOTE: you can NOT use the standard library function foldl or foldr anywhere in your code to solve this problem.

==============================================================================

2. [10p] andmap

Implement the function "andmap-335" in terms of foldl or foldr (you may use either your version, i.e., foldl-335, or the ones from the standard library), with the following signature:

```
> (define (andmap-335 test-op lst) 'todo)
```

```
Input:
  test-op: a one argument function that returns a boolean
  lst: a list of elements

Output:
  #t if for all the elements in lst, the function test-op returns #t
  #f if at least for one element the function test-op returns #f
```

```
;; odd? is a library function that tests whether or not a number is
;; odd
> (odd? 5)
#t

> (andmap-335 odd? '(1 3 5))
#t

> (andmap-335 odd? '(1 3 42))
#f

> (andmap-335 odd? '(1 6 42))
#f
```

Besides andmap, the standard library includes the function ormap which behaves like andmap except that it returns #t if at least one of the element in the list holds the property described by the function.

NOTE: you can NOT use the standard library function andmap in your definition of andmap-335.

======================================================================

```
=========
3. [10p] filter

Implement a function called filter-335 with the signature:

> (define (filter-335 test-op lst) 'todo)

Input:
  test-op: a one argument function that returns a boolean
  lst: a list of elements

Output:
  a list containing all the elements of "lst" for which the test-op
  function returned #t

> (filter-335 odd? '(1 2 3 4 5 6 7 8 9 10))
'(1 3 5 7 9)

> (filter-335 number? '(42 'not-a-number "not-a-number"))
42
==================================================================
=============================RELEVANT EXAMPLES=====================
==================================================================

In the next part of the homework we will be solving problems by
composing the above functions in various ways. From here onwards,
you should use the standard library functions instead of yours
because they are more likely to be bug free and they have the
additional semantics described below:
---
as discussed in class, the standard library function ”map" supports
multiple lists as arguments, although one list can be thought of
as its special case:

> (map + '(1 2 3) '(4 5 6))
'(5 7 9)

If we supply "n" lists as arguments, the argument function has to be
able to take "n" arguments as well.

The behavior of the above statement can be described as follows (read
from bottom to top):

5   7   9 ;;the resulting list
^   ^   ^
|   |   |

1   2   3;;input list one
+   +   +
4   5   6;;input list two
```

Similar to "map" there is the function called "for-each" which has the same signature and similar semantics except that it does not return a value, it is so called “side effects”.

```
> (for-each (lambda (x) (print x) (newline)) '(1 2 3))
1
2
3
```

Without the print function, this expression would have no visible effect.

```
> (for-each (lambda (x) x) '(1 2 3))
;;the return values of the argument function are simply discarded
;;and no new list
;;is built.
```

---

```
> (define (subtract-then-add x y accumulator)
   (+ (- x y) accumulator))

> (foldl subtract-then-add 0 '(32 16 8) '(15 7 3))
31
```

NOTE:

```
;        the default parameter is really the the accumulator, for
;        which please see slides 22 & 23 of Oct 7 lecture notes posted
;        inside moodle. It is the very last parameter in the parameter
;        list of subtract-then-add function, which is then used by the
;        higher-order foldl to generate result. We pointed out the
;        importance of accumulator's position in the lecture slide
;        illustrating foldl, already (again slide 22 & 23 of Oct 7
;         Lecture notes).

;the expression above winds up computing:
;(32 - 15) + (16 - 7) + (8 - 3)
```

If we provide "n" lists, the function passed as an argument to foldl has to take "n + 1" parameters, where the ast parameter is always the accumulator (this is very IMPORTANT to get the result right!)

---

The names for  andmap, ormap are poorly chosen since they do not behave like the "map" function, but rather more like the foldl function.

```
> (define (sum-42? x y) (= (+ x  y) 42))

;this will verify if the sum of every pair of i-th elements is 42.
> (andmap sum-42? '(1 2) '(41 40))
```

```
#t

> (ormap sum-42? '(1 2) '(41 2))
#t

> (ormap sum-42? '(1 2) '(335 2))
#f

If we pass "n" lists as arguments then the argument function has to
take "n” arguments. (Compare it with the case for foldl.)


---
Consider the library function range (we demonstrated it in class)
> (range 5)
'(0 1 2 3 4)

Ponder the utility of this function in conjunction with map and foldl.

=====================================================================
Your solutions for the following problems, from problem 4 to problem
7, need to use the above mentioned map, foldl and range functions.
=====================================================================
4. [10p] map reduce

Implement a function "map-reduce" with the signature:
> (define (map-reduce m-op r-op default-el lst) 'todo)

Input:
   m-op   : a one argument operator
   r-op   : a two argument operator
   default-el: the default element for r-op
   lst    : a list of elements

Output:
  a value resulted from combining each element of the list using r-op
  after you have applied m-op on the said elements.

> (define add-forty-two
    (lambda (x) (+ x 42)))

> (map-reduce add-forty-two + 0 '(0 1 2))
129

=====================================================================
5. [10p] revisiting series

In the previous homework you had to compute a series of the form:
       (-1)^n
An = -----------
```

```
       (n + 1)!

;for n >= 0
Sn = 1 - 1/2 + 1/6 - 1/24 + ...

This time, use map and foldl to compute Sn.
====================================================================
6. [10p] zip

Implement a function "zip" with the signature:
> (define (zip lst1 lst2) 'todo)

Input:
  lst1, lst2, two lists of equal length.

Output:
  a list of pairs containing one element from each list.

> (zip '(1 2 3) '(one two three))
'((1 one) (2 two) (3 three))

You may assume that the lists are of equal length.
====================================================================
7. [10p] matrix-to-vector

Consider the list representation of NxM matrices. In this
representation a 3x4 matrix looks like the following:

> (define example-matrix
   '((1 2 3 4)
     (5 6 7 8)
     (9 0 1 2))
  )

Write a function "matrix-to-vector" with the following signature:

it takes a function and a matrix as arguments and outputs an M sized
vector resulted from successively applying the operation on the
elements of a column.

> (matrix-to-vector + example-matrix)
'(15 8 11 14)

> (define string-matrix
  '(("a" "c" "e")
    ("b" "d" "f"))
   )

> (matrix-to-vector string-append string-matrix)
'("ab"  "cd"  "ef")
```