CS2510 Lab 1

1 Inexact Numbers and Loops

We didn't see it last semester explicitly, but there is another kind of number in DrRacket. You may remember weird numbers returned by trig functions (like `cos`) and sometimes `sqrt`.

```
> (cos pi)
#i-1.0
> (tan pi)
#i-1.2246063538223773e-16
> (sqrt 2)
#i1.4142135623730951
```

Why the new `#i` notation? DrRacket differentiates between numbers that represent *exact* quantities (that might possibly require an infinite amount of memory), and those that are *inexact*, but require only a fixed amount of memory (*fixed* is the important part, since not even Bill Gates has infinite memory... yet).

The main difference between the two types is when numbers get really big, or really small. Here's an example:

```
;; A list of innocuous looking inexact numbers
(define JANUS
  (list #i31
        #i2e+34
        #i-1.2345678901235e+80
        #i2749
        #i-2939234
        #i-2e+33
        #i3.2e+270
        #i17
        #i-2.4e+270
        #i4.2344294738446e+170
        #i1
        #i-8e+269
        #i0
        #i99))
```

In order to sum them quickly, we have two easy options:

```
(define (sum-right alist)
  (foldr + 0 alist))
```

```
(define (sum-left alist)
  (foldl + 0 alist))
```

But unfortunately, these functions do not produce the same results on our list... don't believe me? Try them out:

```
(sum-left JANUS)
(sum-right JANUS)
```

Can you figure out why? Maybe reviewing the definitions of `foldl` and `foldr` will help:

```
;; foldr : [X Y  ->  Y] Y [Listof X]  ->  Y
;; (foldr f base (list x1 ... xn)) = (f x1 ... (f xn base))
(define (foldr f base alox) ...)

;; foldl : [X Y  ->  Y] Y [Listof X]  ->  Y
;; (foldl f base (list x1 ... xn)) = (f xn ... (f x1 base))
(define (foldl f base alox) ...)
```

For example, the (exact) numbers in the list `(list 3 5 8 2)` can be added (at least) two ways:

```
(+ 3 (+ 5 (+ 8 (+ 2 0))))

(+ (+ (+ (+ 0 3) 5) 8) 2)
```

Notice the difference? If you're confused, try stepping them in DrRacket. Then complete the following problems.

1. Design the function `sum-reg` using the standard *Design Recipe* (structural recursion) and explain in what order are the numbers added.

2. Convert the function into one that uses an accumulator, call it `sumacc`. What is the order of the additions now?

3. Now match the two functions `sum-reg` and `sum-acc` with `sum-left` and `sum-right` (and `foldl` and `foldr`).

2 Designing Programs with Accumulators

When do we need an accumulator? When some information/computation within the function we are designing gets lost in recursive calls. The examples of computing the *sum* or a *product* of all numbers qualify, but only if we specify the *order* in which elements should be combined.

What does the accumulator mean? First, the accumulator value is usually the same type (*class of data*) as the expected result of the function. What should the function produce when there is nothing to remember? This value becomes the initial accumulator since it is the value produced by the function when recursion completes (e.g., the empty list is reached). We call this the *base value*.

Once we understand what the accumulator is and what it does, it is important to document the invariant that the accumulator maintains (for explanation of how to specify the invariant, please read the relevant pages in the HtDP text). Usually this is a sentence like: *"The accumulator represents the Y of the Xs seen so far"*.

The template for the whole function then becomes:

```
;; produce a value of the type Y from the given list of X
;; rec-fcn: [Listof X] -> Y
(define (rec-fcn lox)
  (rec-fcn-acc lox base-acc-value))

;; recur with updated accumulator,
;; unless the end of list is reached
;; rec-fcn-acc: [Listof X] Y -> Y
(define (rec-fcn-acc lox acc)
  (cond
    ;; at the end produce the accumulated value
    [(empty? lox) acc]

    ;; otherwise invoke rec-fcn-acc with updated accumulator
    ;; and the rest of the list
    [(cons? lox)  (rec-fcn-acc (rest lox)
                               (update (first lox) acc))]))
```

Identify the parts of this template in your solution to the addition problem. What is the contract for the `update` function? Can we add the `update` function as an argument to the `rec-fcn-acc` function? Try it and compare it with the definition of `foldl` and `foldr`.

2.1 Other Associative Operations

Next we design two functions that compute the *factorial* of a given number. We recall that we can define factorial of 5 as one of the following two values:

```
5! = 1 * 2 * 3 * 4 * 5
5! = 5 * 4 * 3 * 2 * 1
```

Design the functions `fac-L->R` and `fac-R->L` that compute a factorial of the given number. For help consult HtDP, exercise 31.3.2 and the text before this exercise. Run the exercise 31.3.2 and verify the book's assertions about the times needed to compute the two results.

3 Homework Partners

We stop now to set up the homework partner teams.

4 Practice Makes Perfect

In the lectures we were computing the discount price of a list of books the a customer wants to buy. Here's our data definitions.

```
;; A Book is: (make-book String Author Number Kind)
(define-struct book (title author price kind))
```

```
;; An Author is: (make-author String Number)
(define-struct author (name yob))

;; A Kind is one of:
;;   - 'fiction
;;   - 'nonfiction
;;   - 'textbook
```

1. Make three examples of `Author`s
2. Make three examples of `Book`s
3. Write a template for functions that process `Book`s. Remember that in order to process books, you also need to be able to process `Author`s (i.e. you need two templates right?).

Now we want a function that computes the *sale price* of a `Book`. The sale price of a book depends on the daily discounts, which may differ depending on the `Kind` of the book. Assume we have the following discounts: a 30% discount on `'fiction` books, a 20% discount on `'nonfiction` books, and `'textbooks` sell at full price.

Here's the definition... we'll use it below.

```
;; compute the sale price of the given book
;; book-sale-price: Book -> Number
(define (book-sale-price abook)
  (cond
    [(symbol=? (book-kind abook) 'fiction)
     (* 0.7 (book-price abook))]
    [(symbol=? (book-kind abook) 'nonfiction)
     (* 0.8 (book-price abook))]
    [(symbol=? (book-kind abook) 'textbook)
     (book-price abook)]))
```

5 More Accumulators

Consider the following function definition:

```
;; total-sale-price: [Listof Book] -> Number
;; Compute the total sale price of all books in a list
(define (total-sale-price alob)
  (foldl + 0 (map book-sale-price alob)))
```

This function, `total-sale-price`, is very clear, but possibly not as efficient as it could be: we first produce a list of discount prices, then we loop over the list to compute the total sale price. If we were in an actual bookstore, a clerk could (and would) add each computed sale price to the *running total*.[1] That's what just what we'll do:

1. Write tests for `total-sale-price` using your books.

[1] In programming languages, removing intermediate results is a common transformation, known as *deforestation*.

2. Define the `total-sale-price-acc` function that uses an accumulator to keep track of the total sale price up till now. Do not use any of the *Scheme* loops. Run the same tests as we have defined for the original version.

3. Design the function `average-sale-price` that computes the average price per book by following the *Design Recipe* step by step. You may have to define several functions. Do not use any of the DrRacket loop functions.

4. Refactor (redesign) the function `average-sale-price` (into a new function `average-sale-price-acc`) that only walks over the list of books once. Use the same tests to make sure your new function performs correctly.

Save your work — the next lab will build on the work you have done here!