# Programming Paradigms Fall 2022 — Problem Sets

by Nikolai Kudasov

September 8, 2022

## 1 Problem set №2

1. Implement the following functions over a list of binary digits in Racket using explicit recursion. Each function should be implemented independently. Use tail recursion whenever it helps produce a more efficient implementation:

   (a) Convert into a decimal number:

   ```
   (binary-to-decimal '(1 0 1 1 0)) ; ==> 22
   ```

   (b) Count zeros in a binary string (not counting leading zeros):

   ```
   (count-zeros '(0 0 0 1 0 1 1 0)) ; ==> 2
   ```

   (c) Encode a binary string by removing leading zeros and replacing each consecutive substring of digits with its length. For example, `'(0 0 0 1 1 0 1 1 1 0 0)` has some leading zeros, then 2 ones, then 1 zero, then 3 ones, then 2 zeros, so it should be encoded as `'(2 1 3 2)`:

   ```
   (encode-with-lengths '(0 0 0 1 1 0 1 1 1 0 0)) ; ==> '(2 1 3 2)
   ```

   (d) Check whether a given binary string represents an odd number:

   ```
   (binary-odd? '(1 0 1 1 0)) ; ==> #f
   (binary-odd? '(1 0 1 1 1)) ; ==> #t
   ```

   (e) Decrement a binary number. Decrementing zero should produce zero:

   ```
   (decrement '(1 0 1 1 0)) ; ==> '(1 0 1 0 1)
   (decrement '(1 0 0 0 0)) ; ==> '(1 1 1 1)
   (decrement '(0))         ; ==> '(0)
   ```

2. Implement in Racket a function `alternating-sum` that computes a sum of a list of numbers, multiplying every second number by `-1`.
   For example, `(alternating-sum (list 6 2 4 1 3 9))` should compute $6-2+4-1+3-9 = 1$.

   (a) Implement `alternating-sum` using explicit recursion.

   (b) Use the Substitution Model to analyze evaluation of `(alternating-sum (list 1 2 3 4 5))`.

   (c) Argue whether tail recursion can be used to optimize your implementation.

3. Consider the following definitions in Racket:

   ```
   (define (dec n) (- n 1))

   (define (f n)
     (cond
       [(<= n 2) (- 10 n)]
       [else (* (f (dec (dec n))) (f (dec n)))]))
   ```

   Using the Substitution Model explain step-by-step how the following expression is computed (you can evaluate `cond`-expressions immediately, but evaluation of function calls to `f` and `dec` have to be explicit):

   ```
   (f 3)
   ```