Programming Paradigms Fall 2023 — Problem Sets

by Nikolai Kudasov and Khaled Ismaeel

September 7, 2023

1 Problem set №2

1. Implement the following functions over lists of symbols in Racket using **explicit recursion** (i.e. **without** using higher-order functions like `apply`, `map`, `andmap`, `ormap`, `filter`, and `foldl`). Each function should be implemented independently. Use tail recursion whenever it helps produce a more efficient implementation:

   (a) Count a given symbol in a list of symbols:

   ```
   (count-symbol 'l '(h e l l o w o r l d)) ; ==> 3
   ```

   (b) Convert a binary string represented a list of `0`s and `1`s into a (decimal) number:

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

   (c) Return the penultimate symbol in a list (you may assume it has enough symbols):

   ```
   (penultimate '(1 0 0 1 0)) ; ==> 1
   (penultimate '(h e l l o)) ; ==> 'l
   ```

   (d) Encode a 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)
   ```

   (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 `sum-and-product` that computes a sum and a product of a list of numbers.
   For example, `(sum-and-product (list 6 2 4 1))` should compute `'(13 48)`.

   (a) Implement `sum-and-product` using explicit recursion (i.e. **without** higher-order functions).

   (b) Use the Substitution Model to verify that `(first (sum-and-product (list x y z))` is equal to `(+ x y z)`.

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

3. Consider the following definitions in Racket:

   ```
   (define (dec n) (- n 2))

   (define (f n)
     (cond
       [(<= n 1) (- 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)
   ```