Programming Paradigms Fall 2022 — Problem Sets

by Nikolai Kudasov and Khaled Ismaeel

September 15, 2023

1 Problem set №3

1. Implement the following functions over the list of binary digits in Racket using higher-order functions (`apply`, `map`, `andmap`, `ormap`, `filter`, `foldl`) and **without explicit recursion**.

   (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 0s and 1s 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 *without converting to decimal*. 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. Consider this list where each entry is a tuple of the first name, gender, age, and last name:

   ```
   (define employees
     '(("John" #:male    29 . "Malkovich")
       ("Ivan" #:male    18 . "Petrov")
       ("Anna" #:female  22 . "Petrova")
       ("Ivan" #:male    43 . "Ivanov")
       ("Anna" #:female  20 . "Karenina")))
   ```

   (a) Implement a function `fullname` that takes employee and returns their full name as a pair of first and last name:

   ```
   (fullname '("John" #:male 29 . "Malkovich"))
   ; '("John" . "Malkovich")
   ```

   (b) Using higher-order functions (`map`, `ormap`, `andmap`, `filter`, `foldl`) and without explicit recursion, write down an expression that computes a list of entries from `employees` where each employee is `#:male`.

   (c) Using higher-order functions (`map`, `ormap`, `andmap`, `filter`, `foldl`) and without explicit recursion, implement a function `employees-under-21` that computes a list of full names of employees whose age is under 21 given a list of employee entries as input:

   ```
   (employees-under-21 employees)
   ; '(("Ivan" . "Petrov") ("Anna" . "Karenina"))
   ```

3. **(+1% extra credit)** Consider the following definitions:

```
(define (small? n) (< n 10))
(define (large? n) (not (small? n)))
(define (remove-small values) (filter large? values))
(define (sum-large values)
  (cond
    [(empty? values) 0]
    [(small? (first values))
     (sum-large (rest values))]
    [else
     (+ (first values) (sum-large (rest values)))]))
```

Using the Substitution Model, we can prove that for any valid list of numbers `values`, the following two expressions are equivalent:

- `(apply + (remove-small values))`
- `(sum-large values)`

Indeed, when `values` is an empty list we get

```
(apply + (remove-small '()))
= (apply + (filter small? '()))   ; by definition of remove-small
= (apply + '())                   ; by definition of filter
= 0                               ; by definition of apply
= (sum-large '())                 ; by definition of sum-large (inverted)
```

Complete the proof for the case when `values` is not empty:

```
(apply + (remove-small (cons x xs)))
= ... ; <- your proof as a sequence of equalities goes here
= (sum-large (cons x xs))
```

In addition to regular Substitution Model, you can use the following equivalences:

(a) `(apply + (remove-small xs))` ≡ `(sum-large xs)` (inductive hypothesis)

(b) for all `p`, `y`, `ys`, the following expressions are equivalent:

- `(filter p (cons y ys))`
- ```
  (cond
    [(p y) (cons y (filter p ys))]
    [else (filter p ys)])
  ```

(c) for all `y`, `ys`, `(apply + (cons y ys))` ≡ `(+ y (apply + ys))`

(d) for all `f`, `c1`, `c2`, `e1`, `e2`, the following expressions are equivalent:

- `(f (cond [c1 e1] [c2 e2]))`
- `(cond [c1 (f e1)] [c2 (f e2)])`
  ```