CSE1729 - Introduction to Programming April 21, 2017

Laboratory Assignment 12

Objectives

- Work with strings
- Work with heaps

Activities

1. (1 point) Define a Scheme function `num-occurs`, that takes two parameters: a character c and list of characters L. The function `num-occurs` should return as its output value the number of occurrences of c in L. For example

```
> (num-occurs #\e '( #\S #\c #\h #\e #\m #\e))
2
```

Solution:

```
(define (num-occurs i lst)
  (define (num-occurs-aux num lst)
    (cond ((null? lst) num)
          ((equal? (car lst) i)
                   (num-occurs-aux (+ num 1) (cdr lst)))
          (else (num-occurs-aux num (cdr lst)))))
  (num-occurs-aux 0 lst))
```

2. Define a Scheme function, named `freq-list`, which takes a list of characters (a text) as a parameter and returns a list of character-frequency pairs. That is, the function will produce a list of pairs where the first value in the pair is a character and the second value in the pair is the frequency that character appears in the input text. There will be only one pair for each distinct character in the input text. Note, the comparison functions `char<?` and `char>?` are the analogue functions for characters to the `string<?` and `string>?` functions for strings and may be of some use.

   Hint: first use the given code below about binary search tree to define a helper function called `extract-alphabet` that takes a list and returns a new list that contains all the unique elements. Then define another helper function `freq-aux` that takes two lists, the first representing a string of characters and the second containing all the unique characters from the first list, and returns a list of character-frequency pairs.

```
(define (value T) (car T))
```

```
(define (right T) (caddr T))
(define (left T)  (cadr T))
(define (make-tree value left right)
  (list value left right))
 (define (insert x T)
  (cond ((null? T) (make-tree x '() '()))
        ((equal? x (value T)) T)
        ((char<? x (value T)) (make-tree (value T)
                                         (insert x (left T))
                                         (right T)))
        ((char>? x (value T)) (make-tree (value T)
                                         (left T)
                                         (insert x (right T))))))
 (define (insert-all lst T)
    (if (null? lst)
         T
        (insert-all  (cdr lst) (insert (car lst) T))))

(define (extract-all T)
      (define (extract-to T lst)
           (if (null? T)
                lst
               (cons (value T)
                     (extract-to (left T)
                                 (extract-to (right T)
                                             lst)))))
      (extract-to T (list)))
```

Solution:

```
(define (freq-list lst)
  (define (extract-alphabet lst)
    (extract-all (insert-all lst (list))))

  (define (freq-aux a)
    (if (null? a)
        (list)
        (cons
             (cons (car a)
                   (/ (num-occurs (car a) lst) (length lst)))
              (freq-aux (cdr a)))))
  (let ((alph (extract-alphabet lst)))
    (freq-aux alph)))
```

3. Build the logic for a heap structure that stores character-frequency-pairs, making it easy to find the most frequent letters stored in the heap. Specifically, you should produce scheme procedures for heap such as `create-heap`, `h-max`, `left`, `right`, `insert-pair` ,`insert-list`, `extract-most-frequent`. `insert-pair` takes a character-frequency pair and a heap and returns a new heap with the pair inserted. `insert-list` takes a list of character-frequency pairs and a heap and returns a new heap with the list inserted. `extract-most-frequent` takes a heap and returns a pair, where the first value in the pair is the most frequent letter and the second value is the new heap with that letter removed.

Solution:

```
(define (create-heap v H1 H2)
  (list v H1 H2))
(define (h-max H) (car H))
(define (left H)  (cadr H))
(define (right H) (caddr H))
(define (insert-pair x H)
  (if (null? H)
      (create-heap x '() '())
      (let ((child-value
             (if (> (cdr x) (cdr (h-max H)))  (h-max H) x))
             (root-value
             (if (> (cdr x) (cdr (h-max H))) x (h-max H) )))
        (create-heap root-value
                     (right H)
                     (insert-pair child-value (left H))))))

 (define (insert-list lst H)
    (if (null? lst)
          H
        (insert-list (cdr lst) (insert-pair (car lst) H))))

(define (combine-heaps H1 H2)
  (cond ((null? H1) H2)
        ((null? H2) H1)
        ((> (cdr (h-max H1)) (cdr (h-max H2)))
         (create-heap (h-max H1)
                      H2
                      (combine-heaps (left H1) (right H1))))
        (else
         (create-heap (h-max H2)
                      H1
                     (combine-heaps (left H2) (right H2))))))
(define (extract-most-frequent H)
  (cons (car (h-max H)) (combine-heaps (left H) (right H) )))
```

4. Make up an English sentence of your choosing that contains at least five words. Using your procedures from question 1-3, write test code that finds the frequencies of all letters in the sequence and adds them all to a heap. Finally, have your test code extract the most frequent three letters from the heap and verify that they were, in fact the most common three letters in your sentence.

Solution:

```
(define lst (string->list "hello␣scheme␣world"))
(define first-step
    (extract-most-frequent (insert-list (freq-list lst) (list))))
(car first-step)
(define second-step (extract-most-frequent (cdr first-step))
(car second-step)
(define third-step (extract-most-frequent (cdr second-step))
(car third-step)
```