Name: Andrew Hayes
E-mail: a.hayes18@universityofgalway.ie
ID: 21321503

CT331

2023–11–10

Assignment 2: Functional Programming with Scheme

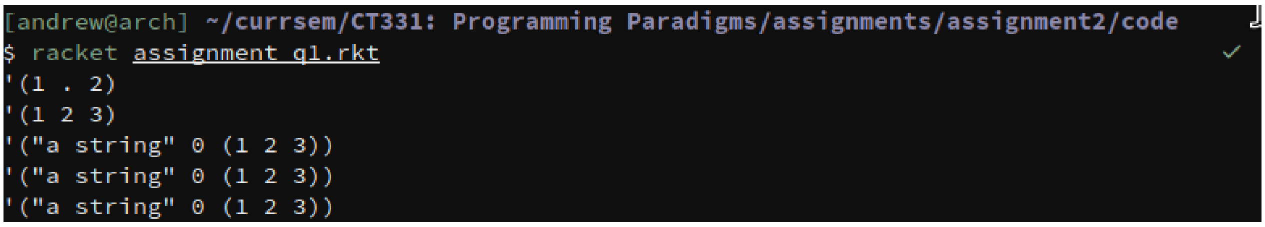
1 Question 1

1.1 Part (A): Code

```
#lang racket

;; a cons pair of two numbers
(cons 1 2)

;; a list of 3 numbers using only the cons function
;; this could be more easily done using the single quote `'` (i.e., `'(1 2 3)`) but i don't use it as it
↪  seemed against the spirit of the question
(cons 1 (cons 2 (cons 3 empty)))

;; a list containing a string, a number, and a nested list of three numbers using only the cons function
(cons "a string"
    (cons 0
        (cons (cons 1 (cons 2 (cons 3 empty))) empty)
    )
)

;; a list containing a string, a number, and a nested list of three numbers, using only the list function
(list "a string" 0 (list 1 2 3))

;; a list containing a string, a number, and a nested list of three numbers, using only the append
↪  function
;; using `'` as the arguments of the `append` function must be themselves lists
(append '("a string") '(0) '((1 2 3)))
```

Listing 1: `assignment_q1.rkt`

1.2 Part (B): Comments



```
[andrew@arch] ~/currsem/CT331: Programming Paradigms/assignments/assignment2/code
$ racket assignment_q1.rkt
'(1 . 2)
'(1 2 3)
'("a string" 0 (1 2 3))
'("a string" 0 (1 2 3))
'("a string" 0 (1 2 3))
```

Figure 1: Output of `assignment_q1.rkt`

Comments on each line of output:

1. The ***cons*** function creates a ***cons*** pair, which is not always a “list”. A list is a ***cons*** pair in which the second element is another itself another list or is ***empty***. When a ***cons*** pair that is not a list is printed, its elements are delimited by a “`.`”, as can be seen from the first line of output.

2. The second section of code produces a list of three numbers using only the ***cons*** function: first we create a one-element list with `(cons 3 empty)`, then we create a two-element list by making a ***cons*** pair of `2` and the already-created one-element list, and finally we create a three-element list by making a ***cons*** pair of `1` and the two-element list. This could of course be achieved far more simply by just using `(cons 1 '(2 3))` or even justs `'(1 2 3)` but I felt that this would be against the spirit of the exercise.

3. To create a nested list using only **cons** in the third section of code, we make the `'(1 2 3)` as previously, **cons** it with **empty** to make a nested list, and then **cons** it with `0`, and **cons** that with a string literal.

4. Like **cons**, **list** can take either atomics or lists as arguments. To create the list using only the **list** function, we can simply make a list of `(list 1 2 3)`, and then create a list consisting of `"a string"`, `0`, & the aforementioned list. This is much simpler than using **cons** because **list** can take as many arguments as we want, while **cons** can only take two arguments.

5. Although I opted not to make use of the "`'`" operator to create lists for the previous exercises, I make use of it here as there is no other way to create a list using only **append** and nothing else, as **append** only accepts lists as arguments. We make a list consisting of only one element (`"a string"`), another list consisting of only one element (`0`), and finally a list consisting of three elements `'(1 2 3)` and append them into one to create the desired list.

2 Question 2

```racket
#lang racket

(provide ins_beg)
(provide ins_end)
(provide count_top_level)
(provide count_instances)
(provide count_instances_tr)
(provide count_instances_deep)

;; function to insert an element at the beginning of a list
(define (ins_beg el lst)
    ;; assuming that the second element is always a list
    (cons el lst)
)

;; function to insert an element at the end of a list
(define (ins_end el lst)
    ;; making el into a list if it isn't already
    (append lst (cons el empty))
)

;; function to count the number of top-level items in a list
(define (count_top_level lst)
    (if (null? lst)
        0                                    ;; return 0 if we've reached the end of the list
        (+ 1 (count_top_level (cdr lst)))   ;; return 1 plus the count_top_level of the second element of
        ↪  the cons pair (the rest of the list)
    )
)

;; non-tail recursive function to count the number of times a given item occurs in a list (assuming items
↪  are atomic)
(define (count_instances item lst)
    (if (null? lst)
        0   ;; return 0 if at the end of the list
        (+
            (if (equal? item (car lst))
                1   ;; if the item is equal to the first element of the list, add 1
                0   ;; if the item is not equal to the first element of the list, add 0
            )
            (count_instances item (cdr lst))    ;; recurse with the remainder of the list
        )
    )
)

;; helper function for count_instances_tr
(define (count_instances_tr_helper item lst cnt)
    (cond
        ;; return the count if the end of the list is reached (0 for empty list)
        ((null? lst)
            cnt
        )
        ;; if the first element of the list is equal to the item
        ((eq? (car lst) item)
            ;; recurse with the remainder of the list and an incremented count
            (count_instances_tr_helper item (cdr lst) (+ cnt 1))
        )
        ;; if the first element of the list is not equal to the item
        (else
            ;; recurse with the remainder of the list and an unchanged count
            (count_instances_tr_helper item (cdr lst) cnt)
        )
    )
)

;; tail recursive function to count the number of times a given item occurs in a list (assuming items are
↪  atomic)
(define (count_instances_tr item lst)
    ;; calling helper function with the list and the count so far (0)
    (count_instances_tr_helper item lst 0)
)

;; function to count the number of times an item occurs in a list and its sub-lists
(define (count_instances_deep item lst)
    (cond
        ;; return nothing if we've reached the end of the list
        ((null? lst)
            0
        )

        ;; if the first item is a list, recurse through the first element and then the rest and return
        ↪  the sum of the two results
        ((pair? (car lst))
            (+ (count_instances_deep item (car lst)) (count_instances_deep item (cdr lst)))
        )

        ;; if the first element is equal to the item, add 1 to the count and recurse with the rest of the
        ↪  list
        ((eq? item (car lst)) ; If the first element is equal to the item, increment count
            (+ 1 (count_instances_deep item (cdr lst)))
        )

        ;; else if the first element is not equal to the item, recurse with the rest of the list
        (else
            (count_instances_deep item (cdr lst))
        )
    )
)
```

Listing 2: assignment_q2.rkt

3 Question 3

```racket
#lang racket

;; function to display the contents of a binary search tree in sorted order
(define (display_contents bst)
    (cond
        ;; if the binary search tree is null, print an empty string (nothing)
        [(null? bst) (display "")]

        ;; if the binary search tree has nodes
        [else
            ;; display the contents of the left sub-tree of the current node
            (display_contents (cadr bst))

            ;; display the current node
            (display (car bst))
            (newline)

            ;; display the contents of the right sub-tree of the current node
            (display_contents (caddr bst))
        ]
    )
)

;; function to search a tree and tell whether a given item is presesnt in a given tree
(define (search_tree item bst)
    (cond
        ;; return false if we've reached the end of the tree without finding a match
        ((null? bst) #f)

        ;; return true if the current node is equal to the item
        ((equal? item (car bst)) #t)

        ;; else return whether the item was found in the left sub-tree or the right sub-tree
        (else
            (or
                (search_tree item (cadr bst))   ;; search left sub-tree
                (search_tree item (caddr bst))  ;; search right sub-tree
            )
        )
    )
)

;; function to insert an item into a binary search tree
(define (insert_item item bst)
    (cond
        ;; if there are no nodes in the tree, create a new tree with the item as the root
        ((null? bst)
            (list item '() '())
        )

        ;; if the item is less than the current node, insert it into the left-hand side of the tree
        ((< item (car bst))
            ;; create new bst with same root node, same right-hand side, but a left-hand side that has
            ↪  had the item inserted
            (list (car bst) (insert_item item (cadr bst)) (caddr bst))
        )

        ;; if the item is greater than the current node, insert it into the right-hand side of the tree
        ((> item (car bst))
            ;; create new bst with same root node, same left-hand side, but a right-hand side that has
            ↪  had the item inserted
            (list (car bst) (cadr bst) (insert_item item (caddr bst)))
        )

        ;; else the item is equal to the current node, so do nothing
        (else bst)
    )
)

;; function to insert a list of items into a binary search tree
(define (insert_list lst bst)
    (if (null? lst)
        ;; if the list is null, just return the bst with no changes
        bst

        ;; otherwise, recurse with the remainder of the list and the binary tree produced by inserting
        ↪  the first item of the list into bst
        (insert_list (cdr lst) (insert_item (car lst) bst))
    )
)

;; tree-sort function
(define (tree_sort lst)
    ;; inserting the list into a tree structure to sort it and then displaying the contents of that tree
    (display_contents (insert_list lst '()))
)

;; function to insert an item into a binary search tree based off a sorting function
;; the sorting function should return accept two items and arguments, and return true if they were passed
↪  in order, and false otherwise or if they are equal
(define (insert_item_custom item bst sorter)
    (cond
        ;; if there are no nodes in the tree, create a new tree with the item as the root
        ((null? bst)
            (list item '() '())
        )

        ;; if the item is goes before the current node, insert it into the left-hand side of the tree
        ((sorter item (car bst))
            ;; create new bst with same root node, same right-hand side, but a left-hand side that has
            ↪  had the item inserted
            (list (car bst) (insert_item_custom item (cadr bst) sorter) (caddr bst))
        )

        ;; if the item goes after the current node, insert it into the right-hand side of the tree
        ((sorter (car bst) item)
            ;; create new bst with same root node, same left-hand side, but a right-hand side that has
            ↪  had the item inserted
            (list (car bst) (cadr bst) (insert_item_custom item (caddr bst) sorter))
        )

        ;; else the item is equal to the current node, so do nothing
        (else bst)
    )
)

;; sorter function which states whether the two arguments were supplied in strictly ascending order (i.e.,
↪  if item == item2, return false)
(define (sort_ascending item1 item2)
    (if (< item1 item2)
        #t
        #f
    )
)
```

```
;; sorter function which states whether the two arguments were supplied in strictly descending order
↪  (i.e., if item == item2, return false)
(define (sort_descending item1 item2)
    (if (> item1 item2)
        #t
        #f
    )
)

;; sorter function which states whether the two arguments were supplied in strictly ascending order based
↪  on the final digit (i.e., if item == item2, return false)
(define (sort_ascending_last item1 item2)
    (if (< (modulo item1 10) (modulo item2 10))
        #t
        #f
    )
)

;; function to insert a list of items into a binary search tree in the order determined by a sorting
↪  function
(define (insert_list_custom lst bst sorter)
    (if (null? lst)
        ;; if the list is null, just return the bst with no changes
        bst

        ;; otherwise, recurse with the remainder of the list and the binary tree produced by inserting
        ↪  the first item of the list into bst
        (insert_list_custom (cdr lst) (insert_item_custom (car lst) bst sorter) sorter)
    )
)
```

Listing 3: `assignment_q3.rkt`

It is worth noting here that the function `sort_ascending_last` operates in a manner that may be undesirable. The function determines whether the two numbers passed to it as arguments were passed in strictly ascending order based on the final digit, i.e. it returns `#t` if the final digit of the first argument is less than the final digit of the second argument, and `#f` otherwise. Because this function considers only the final digit of the numbers, it considers two numbers to be equal if they share a final digit, e.g. it considers the numbers `99` & `9` to be the same. Therefore, if one attempts to insert those two values into the binary search tree using this function as the “sorter”, only the former value will get inserted, as binary search trees do not allow duplicate values, and the `sort_ascending_last` function considers those two values to be equal. However, I thought it would be incorrect for the function to consider the $n - 1^{\text{th}}$ digits in the case of the n^{th} digits being identical, as a) the assignment did not ask for that and b) that would really just be no different to sorting the numbers in ascending order.