Carleton University
Department of Systems and Computer Engineering
SYSC 3101 - Programming Languages - Winter 2019

Lab 2 - Processing Lists in Scheme/Racket

References

Introduction to Computing, http://computingbook.org/

- Section 5.1.
- Section 5.2. For now, you can ignore 5.2.1, *Making Pairs*, and 5.2.2, *Triples to Octuples.*
- Section 5.3.
- Section 5.4: Introductory paragraphs; Section 5.4.1 (*Procedures that Examine Paragraphs*); Example 5.3 from Section 5.4.2 (*Generic Accumulators*) - for now, you can ignore the paragraphs in 5.4.2 on Page 85; Examples 5.6, 5.7 and 5.8 from Section 5.4.3 (*Procedures that Construct Lists*) - for now, you can ignore Examples 5.4 and 5.5.

Two documents at the Racket website provide plenty of information about the Racket dialect of Scheme:

The Racket Guide, https://docs.racket-lang.org/guide/index.html

- See Section 3.8, *Pairs and Lists*

The Racket Reference, https://docs.racket-lang.org/reference/index.html

- Section 4.9, *Pairs and Lists*, summarizes the Racket procedures that operate on immutable lists and pairs.

A guide to the DrRacket IDE can be found here:

http://docs.racket-lang.org/drracket/index.html

Racket Coding Conventions

Please adhere to the conventions described in the Lab 1 handout.

Getting Started

Launch the DrRacket IDE.

If necessary, configure DrRacket so that the programming language is Racket. To do this, select Language > Choose Language from the menu bar, then select The Racket Language in the Choose Language dialog box.

`#lang racket` should appear at the top of the definitions area. Don't delete this line.

"The Rules"

Do not use special forms that have not been presented in lectures. Specifically,

- Do not use `set!` to perform assignment; i.e., rebind a name to a new value.
- Do not use any of the Racket procedures that support *mutable* pairs and lists (`mpair`, `mcons`, `mcar`, `mcdr`, `set-mcar!`, `set-mcdr!`), as described in Section 4.10 of *The*

Racket Reference.
- Do not use begin expressions to group expressions that are to be evaluated in sequence.

You can use lambda expressions to create procedures and let expressions to create local variables, but they aren't required.

You are allowed to use the procedures that are described in these sections of *The Racket Reference*:

- Section 4.9.1, *Pair Constructors and Selector*
- Section 4.9.2, *List Operations*
- Section 4.9.6, *Pair Accessor Shorthands*
- Section 4.9.7, *Additional List Functions and Synonyms*: empty, cons?, empty?, first, rest, second through tenth, last, last-pair.

Unless otherwise noted, you are **not** allowed to use the procedures that are described in these sections of *The Racket Reference*:

- Section 4.9.3, *List Iteration*
- Section 4.9.4, *List Filtering*
- Section 4.9.5, *List Searching*
- Section 4.9.7, *Additional List Functions and Synonyms*: with the exception of the permitted procedures listed earlier.
- Section 4.9.8, *Immutable Cyclic Data*

You can save your solutions to the exercises in a single file; for example, lab2.rkt, or you can create a different file for each exercise.

Exercise 0

Here is the contains procedure that was presented in class. It returns true if target is found in list items; otherwise it returns false.

```
(define (contains? items target)
  (cond
    [(null? items) false]
    [(= (car items) target) true]
    [else (contains? (cdr items) target)]))
```

This procedure has two base cases:

- if items is an empty list, (null? items) is true and target can't be in the list, so return false.

- if the first element in the list, (car items), equals target, return true.

The procedure has one recursive case:

- parameter items refers to the first pair in a list, so (cdr items) returns a reference to the sublist that starts with the second pair. Check if target is in the sublist; i.e., (contains? (cdr items) target)

Type the definition of contains? into the online 61A Scheme interpreter (https://scheme.cs61a.org/), after the scm> prompt. (Note: this interpreter doesn't have the empty? predicate for determining if a list is empty, so the null? predicate is used instead.) The interpreter screen should look like this:

```
scm> (define (contains? items target)
       (cond
         [(null? items) false]
         [(= (car items) target) true]
         [else (contains? (cdr items) target)]))
contains?

scm>
```

After the procedure definition has been entered, the interpreter displays contains?, which indicates that a variable named contains? (bound to a procedure object) has been created.

Now test the procedure, by typing these following expressions:

```
scm> (contains? '(1 2 3 4 ) 1)
scm> (contains? '(1 2 3 4 ) 3)
scm> (contains? '(1 2 3 4 ) 4)
scm> (contains? '(1 2 3 4 ) 7)
scm> (contains? '() 1)
```

We can visualize the execution of contains? by using the interpreter's visualize procedure. Try these expressions:

```
scm> (visualize (contains? '(1 2 3 4 ) 1))
scm> (visualize (contains? '(1 2 3 4 ) 3))
scm> (visualize (contains? '(1 2 3 4 ) 4))
scm> (visualize (contains? '(1 2 3 4 ) 7))
scm> (visualize (contains? '() 1))
```

Use the < and > buttons to observe the procedure's execution, step-by-step. Notice how, as contains? is called recursively, procedure argument (cdr items) is passed to parameter items. This causes the reference stored in parameter items to "move down" the list.

Exercises 1 - 5

Use DrRacket to code and test your solutions to Exercises 1 through 5. Feel free to use the 61A Scheme interpreter to help you debug your procedures.

File Lab_2_test_cases.rkt contains some test cases that you can copy to the file(s) containing your procedure definitions.

For example, we could put the definition of the contains? procedure from Exercise 0 and the following test cases in a .rkt file:

```
(define (contains? items target)
  (cond
    [(null? items) false]
    [(= (car items) target) true]
    [else (contains? (cdr items) target)]))

(display "Testing contains?")
(newline)
(display "Expected: #t, actual: ")
(contains? '(1 2 3 4 ) 1)
(display "Expected: #t, actual: ")
(contains? '(1 2 3 4 ) 3)
(display "Expected: #t, actual: ")
(contains? '(1 2 3 4 ) 4)
(display "Expected: #f, actual: ")
(contains? '(1 2 3 4 ) 7)
(display "Expected: #f, actual: ")
(contains? '() 1)
(newline)
```

When we click DrRacket's `Run` button, the output from the test cases is displayed in the interactions pane:

```
Testing contains?
Expected: #t, actual: #t
Expected: #t, actual: #t
Expected: #t, actual: #t
Expected: #f, actual: #f
Expected: #f, actual: #f
```

Exercise 1

Part (a) Define procedure `(sum-numbers numbers)`. It takes a list of numbers as an argument and returns their sum. Your procedure must recursively sum the numbers. Do not use Racket's `apply` procedure to calculate the sum by applying `+` to the list.

Part (b) Define procedure `(average numbers)`. It takes a list of numbers and returns their average. This procedure must call the `sum-numbers` procedure you defined for part (a).

Exercise 2

Define procedure `(occurrences numbers n)`. It takes a list of numbers and a number, `n`, and calculates how many times `n` occurs in the list. For example,

```
 > (occurrences '(1 3 5 2 7 5 8 9 5) 5)  ; How many 5's in the list?
3
```

Remember, you are not allowed to use any of Racket's list searching procedures (*The Racket Reference*, Sections 4.9.5 and 4.9.7).

Exercise 3

Define procedure `convert`. It takes a list of decimal digits and produces the corresponding integer number. The first digit in the list is the least significant digit. For example:

```
> (convert (cons 1 (cons 2 (cons 3 empty))))
321

> (convert (list 4 5 6))
654

> (convert '(2))
2
```

Exercise 4

Define procedure `convertFC`. It takes a list of temperature measurements in degrees Fahrenheit and returns a list of the equivalent Celsius temperatures. Feel free to define "helper" procedure(s) that are called by `convertFC`.

Remember, you are not allowed to use `map` or any of the other list iteration procedures provided by Racket (*The Racket Reference*, Sections 4.9.3 and 4.9.7).

Exercise 5

Define procedure `eliminate-threshold`. This procedure takes a list of numbers and a threshold value. It returns a list containing all numbers that are below or equal to the threshold. For example,

```
> (eliminate-threshold (list 3 7 0 4 1 5) 4)
'(3 0 4 1)

> (eliminate-threshold (list 3 7 0 4 1 5) -1)
'()  ; returns the empty list
```

Remember, you are not allowed to use `map`, `filter`, or any of the other "forbidden" procedures listed in *"The Rules"* on Page 2.