Racket HW Guide

How to do your Racket homework problems

- this directive describes how to answer both expository problems and programming problems. For detailed instructions on how to format your homework as a single (**.rkt**) file and submit it, see the Racket HW Checklist.

Expository problems

- Some homework problems will be conventional expository questions that require a short written answer.
- All of the assigned expository problems should be answered in the same Racket `.rkt` file as your Racket programming exercises. Of course, you "comment out" each answer either by:

    a. Beginning each line in the answer to an expository question with the *comment escape character* **;** (semicolon) ; or
    b. Enclosing the entire expository answer in a Racket *comment box* created using Racket block comment brackets **#|** and **|#** (sometimes called *bookends*).

- Note that you can create your entire homework file including solutions to expository problems using the DrRacket editor.

Hand evaluation problems

- Most expository problems will be hand-evaluation problems where you are asked to evaluate a particular Racket program invocation. You must format your hand evaluation exactly like our examples (except you will need to insert a comment escape character if the evaluation is embedded in a Racket file with executable Racket code.
    - Example 1: Hand evaluation

```
Given
          (define poly (lambda (x y) (+ (expt 2 x) y)))
      (poly 3 5)
   => ((lambda (x y) (+ (expt 2 x) y)) 3 5)
   => (+ (expt 2 3) 5))
   => (+ 8 5)
   => 13
```

    - Example 2: Hand evaluation

```
Given
          (define fact (lambda (n) (if (zero? n) 1 (* n (fact (- n 1))))))
      (fact 4)
   => (if (zero? 4) 1 (* 4 (fact (- 4 1))))
   => (if false 1 (* 4 (fact (- 4 1))))
   => (* 4 (fact (- 4 1)))
   => (* 4 (fact 3))
   => (* 4 (if (zero? 3) 1 (* 3 (fact (- 3 1)))))
   => (* 4 (if false 1 (* 3 (fact (- 3 1)))))
   => (* 4 (* 3 (fact (- 3 1))))
   => (* 4 (* 3 (fact 2)))
   => (* 4 (* 3 (if (zero? 2) 1 (* 2 (fact (- 2 1))))))
   => (* 4 (* 3 (if false 1 (* 2 (fact (- 2 1))))))
   => (* 4 (* 3 (* 2 (fact (- 2 1)))))
   => (* 4 (* 3 (* 2 (fact 1))))
   => (* 4 (* 3 (* 2 (if (zero? 1) 1 (* 1 (fact (- 1 1)))))))
   => (* 4 (* 3 (* 2 (if false 1 (* 1 (fact (- 1 1)))))))
   => (* 4 (* 3 (* 2 (* 1 (fact (- 1 1))))))
   => (* 4 (* 3 (* 2 (* 1 (fact 0)))))
   => (* 4 (* 3 (* 2 (* 1 (if (zero? 0) 1 (* 0 (fact (- 0 1))))))))
   => (* 4 (* 3 (* 2 (* 1 (if true 1 (* 0 (fact (- 0 1))))))))
   => (* 4 (* 3 (* 2 (* 1 1))))
   => (* 4 (* 3 (* 2 1)))
   => (* 4 (* 3 2))
   => (* 4 6)
   => 24
```

Programming problems

- Most of your homework problems will be programming problems.

- Half (50%) of your grade on programming problems depends on good style and following the recipe (program grading is described in detail on the homework grading page). In particular, these points will be based on
  - 30% for following the design recipe carefully and documenting your program *as you are taught in this course* (see below), and
  - 20% for good programming style *as taught in the course*.
- The other half of your grade will be based on demonstrated correctness:
  - 30% for passing all of our tests; and
  - 20% for constructing a comprehensive set of unit tests for each function in your program.

- All assigned programming problems should be done in the same `.rkt` file.
- At the top of your programming solution file, please put a header with the assignment number, your name, and your e-mail address like this:

```
;; COMP 311 HW #01
;; Christopher Warrington <chrisw@rice.edu>
```

- Strictly follow the formatting and documentation directives given below under the heading **Requirements**. The easiest way to follow these requirements is to imitate the Sample Program solution below.

Requirements

You must include a data definition and corresponding template for each form of data processed used in your homework submissions unless instructed otherwise. You only need to include a data definition and corresponding template once for an assignment, not once for every problem that uses that data defintion.

Data Definitions and Templates:

You need to devise (and document) your data design before you start writing functions that process this data. Data definitions should be documented as follows:

- Example 3. Data definition of *shape*:

```
;; A shape is either
;; * a triangle (make-triangle b h),
;;   where b and h are positive-reals
;; * a square (make-square s),
;;   where s is a positive-real.
(define-struct triangle (base height))
(define-struct square (side))
;;
;; Examples:
;; (make-triangle 1 2)
;; (make-triangle 2 5)
;; (make-square 4)
;; (make-square 2.5)
;;
;; Template for shape
#|
;; shape-function : shape -> ...
(define (shape-function ... shape ...)
  (cond [(triangle? shape) ... (triangle-base shape)   ...
                           ... (triangle-height shape) ...]
        [(square? shape)   ... (square-side shape)     ...]))
|#
```

- Example 4. Data definition of *list-of-numbers*:

```
;; A list-of-numbers is either
;;    empty, or
;;    (cons n lon)
;; where n is a number, and lon is a list-of-numbers
;;
;; Examples:
;; empty
;; (cons 1 empty)
;; (cons 1 (cons 4 empty))
;;
;; Template for list-of-numbers
#|
;; lon-f : list-of-numbers -> ...
(define (lon-f ... a-lon ...)
  (cond
    [(empty? a-lon) ...]
    [(cons? a-lon) ... (first a-lon) ...
                   ... (lon-f ... (rest a-lon) ...) ... ]))
|#
```

- Once your have written a data definition, you can use it in the rest of the assignment.
- The template for writing a function that processes a particular kind of data (data type) is based **only** on the corresponding data definition, **not** on the particular functions that you happen define on that type. Hence, there is only one template per data type. This same template is used as the starting point for writing all functions that process that data type.
- If the type is a structure, the template should include the field extraction operations. If the type is a union, the template needs to have an appropriate `cond` statement (see example 3). If the type is recursive, the template includes the expected recursive calls on the recursive data components (see example 4).

Basic form of a function definition

- The basic form for each function that you write is shown below:
    - Example 5. Function definition for *computing product of list-of-number*:

```
;; product-of-lon : list-of-numbers -> number
;; to compute the product of numbers in a list
;; assuming product of empty list is 1

;; Examples:
;; (product-of-lon empty) => 1
;; (product-of-lon (cons 2 empty)) => 2
;; (product-of-lon (cons 3 (cons 2 empty))) => 6

;; Template instantiation
#|
 (define (product-of-lon a-lon)
   (cond
     [(empty? a-lon) ...]
     [(cons? a-lon) ... (first a-lon) ...
                    ... (product-of-lon (rest a-lon)) ... ]))
|#

;; Code
(define (product-of-lon a-lon)
  (cond
     [(empty? a-lon) 1]
     [(cons? a-lon) (* (first a-lon)
                       (product-of-lon (rest a-lon)))]))

;; Test Examples:
(check-expect (product-of-lon empty) 1)
(check-expect (product-of-lon (cons 2 empty)) 2)
(check-expect (product-of-lon (cons 3 (cons 2 empty))) 6)
;; Provide enough examples and tests to show you tested thoroughly
```

- Remember to follow the design recipe.
- It is important that things are presented in this order, so that is clear that you know the correct order for doing things.

- Be careful with regard to using the **`equal?`** operation in the program code that you write because the worst case running time is the size of the smaller of its two inputs (on inputs that are equal or nearly equal). Of course, the **`equal?`** operation and operations that call **`equal?`** (like **`check-expect`**) are extensively used in test code.
- If your examples get too big, then simply `define` a name for that big argument somewhere before you use it. You can use this name both in your comments in the example section and in the test cases in the Tests section.
- Be sure to test your code thoroughly. Corner cases and edge cases should be tested. When dealing with numerical functions, `0` and `1` are often important test inputs.
- When testing lists, make sure you test the following cases:
    - empty list: **`empty`**
    - list with one element: ex: **`(cons 3 empty)`**
    - list with more than one element: ex: **`(cons 1 (cons 3 (cons 3 (cons 7 empty))))`**
- Local functions cannot be tested individually, so specific tests are not required for them. However, you main function's tests need to be comprehensive enough to test the local functions.

Sample Solution to a Programming Problem

The following text is a good solution to the problem of sorting a list of numbers into ascending order; it pulls together all of the specific pieces of design documentation, code documentation, and testing mentioned above. It would be better if it included a few more appropriately chosen tests.

```
;; COMP 311 HW #Sample
;; Corky Cartwright <cork@rice.edu>

;; A list-of-numbers is either:
;;   empty, or
;;   (cons n alon) where n is a number and alon is a list-of-numbers
;;
;; Examples:
;;   empty
;;   (cons 10 (cons -1 (cons 5 empty))) = '(10 -1 5)
;;   (cons 1 (cons 2 (cons 3 empty)))   = '(1 2 3)
;;   (cons 3 (cons 2 (cons 1 empty)))   = '(3 2 1)

#| Template: (enclosed in block comment brackets)
   (define (lon-f ... a-lon ...)
     (cond
       [(empty? a-lon) ...]
       [(cons? a-lon) ... (first a-lon) ...
                      ... (lon-f ... (rest a-lon) ...) ... ]))
|#

;; Main function: sort

;; Contract and purpose:
;; sort: list-of-numbers -> list-of-numbers
;; Purpose: (sort alon) returns the a list with same elements (including duplicates) as alon but in ascending
order.

;; Examples:
;; (sort empty) = empty
;; (sort '(0)) = '(0)
;; (sort '(1 2 3)) = '(1 2 3)
;; (sort '(3 2 1)) = '(1 2 3)
;; (sort '(10 -1 10 -20 5)) = (-20 -1 5 10 10)

#| Template Instantiation:
   (define (sort a-lon)
     (cond
       [(empty? a-lon) ...]
       [(cons? a-lon) ... (first a-lon) ...
                      ... (sort (rest a-lon)) ... ]))
|#
;; Code:

   (define (sort a-lon)
     (cond
       [(empty? a-lon) empty]
       [(cons? a-lon) (insert (first a-lon) (sort (rest a-lon)))]))

;; Tests
(check-expect (sort empty) empty)
(check-expect (sort '(0)) '(0))
```

```
(check-expect (sort '(1 2 3)) '(1 2 3))
(check-expect (sort '(3 2 1)) '(1 2 3))
(check-expect (sort '(10 -1 10 -20 5)) '(-20 -1 5 10 10))

;; Auxiliary function

;; Contract and purpose
;; insert: number list-of-numbers -> list-of-numbers
;; Purpose: (insert n alon), where alon is in increasing order, returns a list containing n and the elts of
alon in ascending order

;; Examples:

;;  (insert 17 empty) = '(17)
;;  (insert 17 '(17)) = '(17 17)
;;  (insert 4 '(1 2 3)) = '(1 2 3 4)
;;  (insert 0 '(1 2 3)) = '(0 1 2 3)
;;  (insert 2 '(1 1 3 4)) = '(1 1 2 3 4)

#| Template instantiation
   (define (insert n a-lon)
     (cond
       [(empty? a-lon) ...]
       [(cons? a-lon) ... (first a-lon) ...
                      ... (insert n (rest a-lon)) ... ]))
|#

;; Code
   (define (insert n a-lon)
     (cond
       [(empty? a-lon) (cons n empty)]
       [(cons? a-lon)
        (if (<= n (first a-lon)) (cons n a-lon)
            (cons (first a-lon) (insert n (rest a-lon))))]))
;; Tests

(check-expect (insert 17 empty) '(17))
(check-expect (insert 17 '(17)) '(17 17))
(check-expect (insert 4 '(1 2 3)) '(1 2 3 4))
(check-expect (insert 0 '(1 2 3)) '(0 1 2 3))
(check-expect (insert 2 '(1 1 3 4)) '(1 1 2 3 4))
```

Note: the Examples and Tests for each function above can be collapsed into a single entry by cutting out each Tests block and pasting it over the corresponding Examples block. Forward references in `check-expect` invocations work because the execution of `check-expect` code is deferred to the end of the contents of the definitions pane. For example, the `Auxiliary function` part can be rewritten as follows:

```
;; Auxiliary function

;; Contract and purpose
;; insert: number list-of-numbers -> list-of-numbers
;; Purpose: (insert n alon), where alon is in increasing order, returns a list containing n and the elts of
alon in ascending order

;; Examples and Tests:

(check-expect (insert 17 empty) '(17))
(check-expect (insert 17 '(17)) '(17 17))
(check-expect (insert 4 '(1 2 3)) '(1 2 3 4))
(check-expect (insert 0 '(1 2 3)) '(0 1 2 3))
(check-expect (insert 2 '(1 1 3 4)) '(1 1 2 3 4))

#| Template instantiation
   (define (insert n a-lon)
     (cond
       [(empty? a-lon) ...]
       [(cons? a-lon) ... (first a-lon) ...
                      ... (insert n (rest a-lon)) ... ]))
|#

;; Code
   (define (insert n a-lon)
     (cond
       [(empty? a-lon) (cons n empty)]
       [(cons? a-lon)
        (if (<= n (first a-lon)) (cons n a-lon)
            (cons (first a-lon) (insert n (rest a-lon))))]))
```

Updated Sample Program

A function named **`sort`** is included in the core library for most (all?) of the Racket dialects supported by DrRacket. The following revision (with no changes to the code other than the name of the **`sort`** function) should run in all of the HTDP student languages except **Beginner** which excludes abbreviated list notation using **`'`**.

```
;; COMP 311 HW #Sample
;; Corky Cartwright <cork@rice.edu>

;; A list-of-numbers is either:
;;   empty, or
;;   (cons n alon) where n is a number and alon is a list-of-numbers
;;
;; Examples (some written as binding of variable names to values)
;;   empty
;;   (cons 10 (cons -1 (cons 5 empty))) = '(10 -1 5)
(define l0    (cons 0 empty))                    ;;  '(0)
(define l123  (cons 1 (cons 2 (cons 3 empty))))  ;;  '(1 2 3)
(define l321  (list 3 2 1))                      ;;  '(3 2 1)

#| Template for the data type list-of-numbers (enclosed in block comment brackets):
   (define (lon-f ... a-lon ...)
     (cond
       [(empty? a-lon) ...]
       [(cons? a-lon) ... (first a-lon) ...
                      ... (lon-f ... (rest a-lon) ...) ... ]))
|#

;; Main function: _sort_  (the name sort is already defined in the library)

;; Contract and purpose:
;; _sort_: list-of-numbers -> list-of-numbers
;; Purpose: (_sort_ alon) returns the a list with same elements (including duplicates) as alon but in ascending
order.

;; Examples (written as executable tests)
```

```
;; Named values which make the executable tests more concise

;; Executable tests (most using named values define as list-of-numbers examples)
(check-expect (_sort_ empty) empty)
(check-expect (_sort_ l0) l0)
(check-expect (_sort_ l123) l123)
(check-expect (_sort_ l321) l123)
(check-expect (_sort_ '(10 -1 10 -20 5)) '(-20 -1 5 10 10))

;; Tests already set up as executable examples

#| Template Instantiation for _sort_ function:
   (define (_sort_ a-lon)
     (cond
       [(empty? a-lon) ...]
       [(cons? a-lon) ... (first a-lon) ...
                      ... (_sort_ (rest a-lon)) ... ]))
|#
;; Code (using not yet defined auxiliary function insert)

   (define (_sort_ a-lon)
     (cond
       [(empty? a-lon) empty]
       [(cons? a-lon) (insert (first a-lon) (_sort_ (rest a-lon)))]))

;; Auxiliary function

;; Contract and purpose
;; insert: number list-of-numbers -> list-of-numbers
;; Purpose: (insert n alon), where alon is in increasing order, returns a list containing n and the elts of
alon in ascending order

;; Examples:

(check-expect (insert 17 empty)  '(17))
(check-expect (insert 17 '(17))  '(17 17))
(check-expect (insert 4 l123) '(1 2 3 4))
(check-expect (insert 0 l123) '(0 1 2 3))
(check-expect (insert 2 '(1 1 3 4)) '(1 1 2 3 4))

#| Template instantiation
   (define (insert n a-lon)
     (cond
       [(empty? a-lon) ...]
       [(cons? a-lon) ... (first a-lon) ...
                      ... (insert n (rest a-lon)) ... ]))
|#

;; Code
   (define (insert n a-lon)
     (cond
       [(empty? a-lon) (cons n empty)]
       [(cons? a-lon)
        (if (<= n (first a-lon)) (cons n a-lon)
            (cons (first a-lon) (insert n (rest a-lon))))]))

;; Tests already set up as executable examples
```

Termination Argument: (Only programs using generative forms of recursion given in Chapter 23 and beyond)

If the template does not guarantee that a function terminates, you are required to explain why that function will terminate for all possible inputs. The standard structural recursion templates guarantee termination.

- Example 7. Termination argument for `quick-sort` :

```
      ;; quick-sort list-of-number -> list-of-number
      ;; Purpose: (quick-sort lon) sorts lon into ascending order
      ;;
      ;; Examples:
      (check-expect (quick-sort empty) empty)
      (check-expect (quick-sort '(1)) '(1))
      (check-expect (quick-sort '(1 4 3 5)) '(1 3 4 5))
      (check-expect '(1 4 3 4 2 5)) '(1 2 3 4 4 5))
      ;;
      ;; Termination:
      ;; On each call, quick-sort partitions the list alon into three sublists using
      ;; smaller-than, larger-than, and equal-to.  The lists produced by smaller-than
      ;; and larger-than are sorted using recursive applications of quick-sort. Since
      ;; the lists produced by smaller-than and larger-than are strictly shorter than
      ;; alon (the given list), quick-sort terminates.
      (define (quick-sort alon)
        (cond
          [(empty? alon) empty]
          [else (append (quick-sort (smaller-items alon (first alon)))
                        (equal-to alon (first alon))
                        (quick-sort (larger-items alon (first alon))))]))

      ;; Remainder of this example (including the definitions of smaller-items and larger-items) is
elided
      ...
```