CSC172 LAB 14

DATA IN SCHEME

1 Introduction

The labs in CSC172 will follow a pair programming paradigm. Every student is encouraged (but not strictly required) to have a lab partner. Labs will typically have an even number of components. The two partners in a pair programming environment take turns at the keyboard. This paradigm facilitates code improvement through collaborative efforts, and exercises the programmers cognitive ability to understand and discuss concepts fundamental to computer programming. The use of pair programming is optional in CSC172. It is not a requirement. You can learn more about the pair programming paradigm, its history, methods, practical benefits, philosophical underpinnings, and scientific validation at http://en.wikipedia.org/wiki/Pair_programming .

Every student must hand in their own work, but every student must list the name of their lab partner if any on all labs.

This lab has six parts. You and your partner(s) should switch off typing each part, as explained by your lab TA. As one person types the lab, the other should be watching over the code and offering suggestions. Each part should be in addition to the previous parts, so do not erase any previous work when you switch.

The textbook should present examples of the code necessary to complete this lab. However, collaboration is allowed. You and your lab partner may discuss the lab with other pairs in the lab. It is acceptable to write code on the white board for the benefit of other lab pairs, but you are not allowed to electronically copy and/or transfer files between groups.

2 More Than Just Parentheses

1. Since this is a course in data structures, let's look at structuring some data in Lisp/Scheme. The fundamental concept of data aggregation in this language is the *list*. (You never would have guessed that, right?) We can construct lists in lisp with the `list` operator. Let's try this, just for fun. Type in the following and examine the results (make sure you're using the **R5RS** language for this lab):

```
(list 1 2 3)
(list 1 2 3 5 6 7 8)
(define list1 (list 0 1 2 3 4 5 6 7 8 9))
(display list1)
(newline)
(list "abe" "bea" "cal" "deb")
(define bintree (list (list 1 2) (list 3 4)))
(display bintree)
(newline)
(define arry3x3 (list (list 1 2 3) (list 4 5 6) (list 7 8 9)))
(display arry3x3)
(newline)
'()
(list)
(null? '())
(null? (list))
(null? (list 1 2 3))
```

So, it should be clear from the above that we can make lists. We can even make empty (a.k.a "null") lists if we like. We can define variables to refer to lists. We can also make lists of lists. Perhaps you might have noticed that lists of lists could look like arrays and that more complex nested list structures look like binary trees? (Don't worry if you don't – it should become apparent later.)

2. Ok, we can construct a list, but how do we get stuff out of it? In Lisp, there are two commands of fundamental importance. The first is the `car` operator[1]. The second is the `cdr` operator (pronounced "cud-er")[2]. The `car` operator gives us the head of a list. The `cdr` operator gives us everything but the head. Type the following:

```
(car (list 1 2 3))
(cdr (list 1 2 3))
(define list2 (list 0 1 2 3 4 5 6 7 8 9))
(display list2)
(newline)
(car list2)
(cdr list2)
```

1 Once upon a time, this stood for "**C**ontents of **A**ddress part of **R**egister", on an IBM 704 computer.
2 Similarly, "**C**ontents of **D**ecrement part of **R**egister".

```
(cdr (cdr list2))
(cdr (cdr (cdr list2)))
(car (cdr (cdr (cdr list2))))
```

It turns out that there is a nice abbreviation syntax to make the code a bit more compact. Instead of typing `(cdr (cdr someList))` we can use `(cddr someList)`. It works with the "a" in car as well. Type in and observe the following:

```
(display list2)
(newline)
(cdr list2)
(cddr list2)
(cdddr list2)
(car (cdddr list2))
(cadddr list2)

(define list3 (list
    (list (list 1 2) (list 3 4) (list 5 6) )
    (list (list 7 8) (list 9 10) (list 11 12) )
    (list (list 13 14) (list 15 16) (list 17 18 ) )
  )
)
(display list3)
(newline)
(car list3)
(caar list3)
(caaar list3)
(cdr list3)
(cadr list3)
(cdar list3)
(cdadr list3)
```

3. One more simple operation. We can construct a *pair* (sometimes referred to as a *dotted-pair*) with the **cons** operation[3]. It's just a bit more complex than the list operation, but it helps us to build lists a little bit at a time. Again, type and observe.

```
(cons 1 2)
(car (cons 1 2))
(cdr (cons 1 2))
(define list4 (cons (cons 1 2) (cons 3 4)))
(display list4)
(newline)
(car list4)
(cdr list4)
(caar list4)
(cadr list4)
(cdar list4)
(cddr list4)
(cons 1 '())
(cons '() 1)
(cons 1 (cons 2 '()))
(cons 1 (cons 2 (cons 3 '())))
(cons 1 (cons 2 (cons 3 (cons 4 '()))))
```

4. So, now that we can construct data items like pairs and list. Let's make some functions that process data. Consider making a *rational number* processing system. As you know, rational numbers have an integer numerator and denominator – perfect for a pair operation. Just like in Java we can think in terms of constructor and accessor operations.

```
(define (make-rat n d ) (cons n d))
(define (numer rat) (car rat))
(define (denom rat) (cdr rat))
(define (print-rat rat)
  (display (numer rat))
  (display "/")
  (display (denom rat))
  (newline))
```

3 "cons" from "**cons**truct" - pretty clever, eh?

```
(define one-half (make-rat 1 2))
(define one-third (make-rat 1 3))
(display one-half)
(newline)
(print-rat one-half)
(display one-third)
(newline)
(print-rat one-third)
```

5. Now, let's implement the five basic arithmetic operations (addition, subtraction, multiplication, division and test for equality), on the "rational number type". What you need to do is to write function definitions for `(add-rat rat1 rat2), (sub-rat rat1 rat2), (mul-rat rat1 rat2)` and `(div-rat rat1 rat2)`. In the body of each of these you should make a call to the make-rat constructor, passing in the appropriately extracted values for the new numerator and denominator. The `(equal-rat rat1 rat2)` doesn't need to construct a new rational number. Instead, it uses a call to the "=" operator with the appropriate values. To implement these methods, use the formulas shown. The definition of add-rat is given as a template.

$$\frac{n_1}{d_1}+\frac{n_2}{d_2}=\frac{n_1 d_2+n_2 d_1}{d_1 d_2}$$

$$\frac{n_1}{d_1}-\frac{n_2}{d_2}=\frac{n_1 d_2-n_2 d_1}{d_1 d_2}$$

$$\frac{n_1}{d_1}\times\frac{n_2}{d_2}=\frac{n_1 n_2}{d_1 d_2}$$

$$\frac{n_1/d_1}{n_2/d_2}=\frac{n_1 d_2}{d_1 d_2}$$

$$\frac{n_1}{d_1}=\frac{n_2}{d_2} \text{ if and only if } n_1 d_2=n_2 d_1$$

```
(define (add-rat rat1 rat2)
          (make-rat numerator denominator))
```

Where *numerator* and *denominator* are replaced by the arithmetic operations as specified in the formula above.

6. Compete this lab constructing at least 10 different rational numbers. Demonstrate that the functions you defined in the previous section work by performing operations on the rationals you define.

3 Hand In

Hand in the source code from this lab at the appropriate location on the blackboard system at my.rochester.edu. You should hand in a single compressed/archived (i.e. “zipped” file that contains the following.)

1. A plain text file named README that includes your contact information, your partner's name, a brief explanation of the lab (A one paragraph synopsis. Include information identifying what class and lab number your files represent.). And one sentence explaining the contents of all the other files you hand in.
2. The Racket file containing the definitions you wrote in the lab as well as those copied from the lab prompt with author information commented out at the top.
3. A plain text file named OUTPUT that includes your interactions from DrRacket.

4 Grading

90% Functionality

- 30% Parts 1-4
- 50% add-rat, sub-rat, mul-rat, div-rat, equal-rat functions (10% each)
- 10% Testing the rational number functions (2% for testing each function)

10% README and OUTPUT files