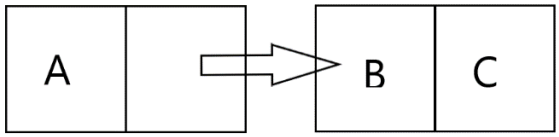
3.1: Cons Cell Representations (check your answers with Racket)

a)



Racket:

```
> (cons ’A (cons ’B ’C))
‘(A B . C)
```

b)

```
(cons((cons 'A(cons 'B 'C))(cons 'B 'C)))
```

ابتدا یک Cons Cell ساخته‌شده که دارای دو Cons Cell دیگر است که خانه اولِ Cons Cell اول Cons Cell دوم، خانه‌ی دومِ Cons Cell اول Cons Cell (B . C) اول، خانه‌ی اولِ Cons Cell دوم، خانه‌ی دومِ Cons Cell دوم Cons Cell (B . C) دوم هستند.

c)

```
(lambda (x) cons (cons('A x))(cons 'B 'C))
```

باید از lambda استفاده‌کنیم تا Cons Cell (B . C) به هر دو bind Cons Cell شود.

3.2: Conditional Expressions in Lisp

a)

از آن‌جایی که نمی‌دانیم آیا این برنامه halt می‌کند یا خیر، پیاده‌سازی آن معادل حل مسئله‌ی halting است(غیرممکن).

b)

می‌توانیم شرط‌ها را به‌صورت موازی بررسی کرده و مقدار اولین true را بازگردانیم(یا undefined در صورتی که همه‌ی آن‌ها false بودند).

c)

```
(defun odd (x) (cond (atom x)
                     (cond ((eq x 0) nil)
                     ((eq x 1)t))
                     ((> x 0)(odd (- x 2)))
                     (t(odd(+ x 2)))))
```

d) S_{COR} P_{OR}

برای S_{COR} قسمت (a) مناسب‌تر است و برای P_{OR} نامناسب؛ چرا که شرط‌ها به ترتیب بررسی می‌شوند.
برای P_{OR} قسمت (b) مناسب‌تر است و برای S_{COR} نامناسب؛ چرا که شرط‌ها به‌صورت موازی بررسی می‌شوند.

3.4: Lisp and Higher-Order Functions (write example codes and test in Racket)

طبق صورت مسئله، ابتدا قسمت‌های b و c و سپس a پاسخ داده‌شده‌اند.

b)

i)maplist
ii)car

c)

compose f1 f2 = lambda(x) (f1(f2 x)) = lambda(xs)(f)
define f1 (lambda(x) f1)
define f2 (lambda(x) cond(#t f))

a)

```
(define compose2
        (lambda (g h)
                (lambda (f xs)
                        (g (lambda (xs) (f(h(xs))) xs)
)))
```

3.5: Definition of Garbage

a)

بله، از آن‌جایی که برنامه‌های Lisp تنها به base registerها دسترسی دارند، اگر برنامه نتواند به یک آدرس حافظه دسترسی داشته‌باشد، پس، از طریق base registerها هم نمی‌توان به آن دسترسی داشت؛ بنابراین، بر اساس تعریف McCarthy هم garbage محسوب می‌شود.

b)

بله، مانند قسمت قبل، اگر برنامه به یک base register دسترسی نداشته‌باشد، دسترسی به دیگر خانه‌های حافظه از طریق آن base register نیز غیرممکن است؛ پس، بر اساس تعریف "ما" هم garbage است.

c)

نه لزوماً؛ به عنوان مثال در برنامه‌هایی که به زبان c نوشته شده‌اند، از آن‌جایی که نمی‌توانیم مطمئن باشیم که دیگر به بخشی از خانه‌های حافظه نمی‌توان دسترسی داشت و می‌توانیم حتی به خانه‌هایی از حافظه که متعلق به برنامه نیستند نیز دسترسی داشته‌باشیم، نمی‌توان یک garbage collector براساس تعریف خودمان بسازیم.

3.6: Reference Counting

a)

ابتدا c و d در یک cons cell قرار می‌گیرند؛ سپس a و b هم در یک cons cell دیگر قرار می‌گیرند و بعد از آن این دو cons cell در یک cons cell.
ابتدا cdr ارزیابی می‌شود که نتیجه آن (c . d) است؛ پس (a . b) و cons cell بزرگ‌تر حذف می‌شوند.
سپس car ارزیابی می‌شود که نتیجه آن c است؛ پس (c . d) حذف می‌شود.
بنابراین، هیچ‌کدام از cons cell ها باقی نمی‌مانند.

b)

در صورت استفاده از دستورات rplaca و rplacd، از آن‌جا که یک cons cell جدید تولید نمی‌شود و فقط مقدار پوینتر آن(head یا tail) تغییر می‌کند، نمی‌توان میزان ارجاع‌ها به یک خانه از حافظه را کنترل کرد.

به عنوان مثال:

(lambda(x) rplaca (rplacd (x x)nil))(cons(cons('a 'b)'c))

برای اتم c، ابتدا پوینتر آن در تابع استفاده می‌شود و سپس دچار تغییر می‌شود که با پوینتر اول یکسان نیست؛ پس نمی‌توان الگوریتم reference-counting را در impure lisp داشته‌باشیم.

Exercise. Implement a fixed-point combinator (as you have seen in the lambda-calculus) in Lisp (or in Racket). Can you define recursive functions using your combinator?

Y = λf. (λx. f (x x)) (λx. f (x x))
Z = λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))

```
#lang racket
(define Y (lambda (b) ((lambda (f) (b (lambda (x) ((f f) x))))
                       (lambda (f) (b (lambda (x) ((f f) x)))))))

(define (Z f)
  ((lambda (x) (x x))
   (lambda (x) (f (lambda (y) ((x x) y))))))

(define Fib-Y
  (Y (lambda (fib) (lambda (n) (if (<= n 1) n (+ (fib (- n 1)) (fib (- n 2))))))))

(define Fib-Z
  (Y (lambda (fib) (lambda (n) (if (<= n 1) n (+ (fib (- n 1)) (fib (- n 2))))))))

(Fib-Y 16)
(Fib-Z 16)
```

```
>987
>987
```