MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall 2007

Recitation 5 Solutions
Data Structures and Abstractions

Scheme

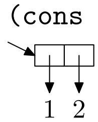
New procedures

1. `(cons a b)` - Makes a cons-cell (pair) from a and b
2. `(car c)` - extracts the value of the first part of the pair
3. `(cdr c)` - extracts the value of the second part of the pair
4. (`c`$\frac{a}{d}\frac{a}{d}\frac{a}{d}\frac{a}{d}$`r c`) - shortcuts. `(cadr x)` is the same as `(car (cdr x))`
5. `(list a b c ...)` - builds a list of the arguments to the procedure
6. `(define nil '())` - the special object `'()`, called the empty list, denotes the end of a list. We often write this as `nil` instead of `'()`.
7. `(null? a)` - returns `#t` if `a` is the empty list (`nil` or `'()`), and `#f` otherwise.

Problems

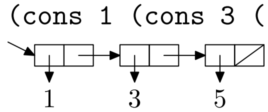
1. Draw box-and-pointer diagrams for the values of the following expressions. Also give the printed representation.

   (a) `(cons 1 2)`

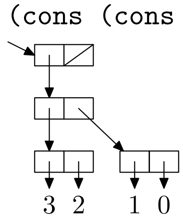
   `(1 . 2)`

   (b) `(cons 1 (cons 3 (cons 5 '())))`

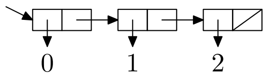
   `(1 3 5)`

   (c) `(cons (cons (cons 3 2) (cons 1 0)) '())`

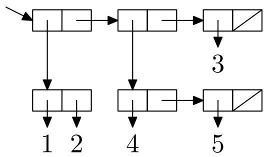
   `(((3 . 2) 1 . 0))`

(d) `(cons 0 (list 1 2))`



`(0 1 2)`

(e) `(list (cons 1 2) (list 4 5) 3)`



`((1 .  2) (4 5) 3)`

2. Write expressions whose values will print out like the following.

   (a) `(1 2 3)`

   `(list 1 2 3)` or `(cons 1 (cons 2 (cons 3 '())))`

   (b) `(1 2 .  3)`

   `(cons 1 (cons 2 3))`

   (c) `((1 2) (3 4) (5 6))`

   `(list (list 2 3) (list 3 4) (list 5 6))`

3. Create a data abstraction for points in a plane. It should have a constructor, (`make-point` *x y*), which returns a point, and two selectors (`point-x` *pt*) and (`point-y` *pt*), which return the *x* and *y* coordinates.

```
(define (make-point x y)
  (list x y))
(define (point-x pt)
  (car pt))
(define (point-y pt)
  (cadr pt))
```

4. Now, extend the point abstraction to handle line segments, with a constructor (`make-line-segment` *pt*1 *pt*2), and selectors `line-segment-start` and `line-segment-end`.

```
(define (make-line-segment pt1 pt2)
  (cons pt1 pt2))
(define (line-segment-start pt)
  (car pt))
(define (line-segment-end pt)
  (cdr pt))
```

5. Write a procedure (`intersection` *seg*1 *seg*2) that returns a point where two line segments intersect if they do, and returns #f if they do not intersect. Be sure to honor the abstractions defined.

```
(define (intersection s1 s2)
  (let ((x1 (point-x (line-segment-start s1)))
        (x2 (point-x (line-segment-end s1)))
        (x3 (point-x (line-segment-start s2)))
        (x4 (point-x (line-segment-end s2)))
        (y1 (point-y (line-segment-start s1)))
        (y2 (point-y (line-segment-end s1)))
        (y3 (point-y (line-segment-start s2)))
        (y4 (point-y (line-segment-end s2))))
    (let ((n1 (- (* (- x4 x3) (- y1 y3))
                 (* (- y4 y3) (- x1 x3))))
          (n2 (- (* (- x2 x1) (- y1 y3))
                 (* (- y2 y1) (- x1 x3))))
          (d (- (* (- y4 y3) (- x2 x1))
                (* (- x4 x3) (- y2 y1)))))
      (if (zero? d) #f
          (let ((u1 (/ n1 d))
                (u2 (/ n2 d)))
            (if (or (< u1 0) (< u2 0)
                    (> u1 1) (> u2 1))
                #f
                (make-point (+ x1 (* u1 (- x2 x1)))
                            (+ y1 (* u1 (- y2 y1))))))))))
```