DATA ABSTRACTION AND SEQUENCES 4

COMPUTER SCIENCE 61AS

Basics of Pairs and Sequences

1. What is the value of `(car '(1 . 2))`?

Solution: 1

2. What is the value of `(cdr '(1))`?

Solution: `()`

Remember that `(list a b c)` is the same thing as `(cons a (cons b (cons c ())))`, and that other invocations of list behave in the same way. `'()` is the empty list — its a special value that is a list but not a pair. You can check if something is the empty list using the predicate null?.

3. Suppose I enter `(cons '(1 . 2) '(3 . 4))` into the interpreter. What will Racket print?

Solution: `((1 . 2) 3 . 4)`

How you can think about Racket printing a pair:
It prints a `(`, recursively prints the car, prints `.`, recursively prints the cdr, and prints `)`. Once this is done, it goes back, and if it ever sees `. (`, it throws away the `.`, the `(`, and the matching `)`.

In this example, our first idea is to say `((1 . 2) . (3 . 4))`. Then according to the rule above, we take out the `. (` and its matching parenthesis to end up with
`((1 . 2) 3 . 4)`.

Practice with Pairs and Sequences

1. What will Racket print if each of these are entered into the interpreter? If it is an error, write 'error'. Draw the box and pointer diagram for each one if possible.

   a. `(cons (cons (cons 1 4) 5) (cons 3 (list 4)))`

   Solution: `(((1 . 4) . 5) 3 4)`

   b. `(append (cons 1 (list 3)) (list (cons 1 2)))`

   Solution: `(1 3 (1 . 2))`

   c. ``(list (append (list 2 4) (list (list (list 6))) `(3)) `(1))``

   Solution: `((2 4 ((6)) 3) (1))`

Basics of Data Abstraction

1. Why do we define constructors and selectors rather than just telling people use `car` and `cdr`?

Solution: Using constructors and selectors allows us to abstract away the internal representation of objects and be more expressive when we create them.

The idea of data abstraction is to conceal the representation of some data and to instead reveal a standard "interface" that is more aligned with what the data represents as opposed to how the data is represented.

To give a concrete example when you use numbers in Racket, you dont care how they are represented you just care about whether you know what they mean. Would you prefer to code `(* 22 7)` to get `154` or `(* 10110 111)` to get `10011010` (the internal representation)?

2. I want to create an ADT representing a point in 2D space that keeps track of its x and y coordinates. Define a set of possible constructor/selectors called `make-point`, `point-x`, and `point-y`.

Solution: One possible solution:

```
(define make-point cons)
(define point-x car)
(define point-y cdr)
```

Another possible solution (using a list this time):

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

Practice with Data Abstraction

1. Suppose we want to write a procedure which, given a list of grades (numbers between 0 and 100), finds both the minimum and the maximum grade. Since in Racket we can only return one thing, well invent a new ADT a num-pair. Assuming you have already defined the constructor, `make-num-pair`, and the selectors, `first-num` and `second-num`, write a procedure `minmax` which takes in a list of numbers and returns a `num-pair` whose `first-num` is the minimum element and whose `second-num` is the maximum element. You may assume that the input list has at least one number.

Solution:

```
(define (minmax grades)
  (make-num-pair (min-grades grades) (max-grades grades)))

(define (min-grades grades)
  (if (null? (cdr grades))
      (car grades)
      (min (car grades) (min-grades (cdr grades)))))

(define (max-grades grades)
  (if (null? (cdr grades))
      (car grades)
      (max (car grades) (max-grades (cdr grades)))))
```

2. Now lets actually write the constructors and selectors. But lets test 3 different types!

   a. Write constructors and selectors which represent a num-pair as a pair/list.

```
> (pair? (make-num-pair 50 60))
#t
> (second-num (minmax (89 94 83 95 91 50)))
95
```

Solution:

```
(define make-num-pair cons)
(define first-num car)
(define second-num cdr)
```

b. Write constructors and selectors which represent a num-pair as a number. (Hint: Remember that since the numbers represent grades, they must be in the range 0-100, inclusive.)

```
> (number? (make-num-pair 50 60))
#t
> (first-num (minmax (89 94 83 95 91 50)))
50
```

Solution:

```
(define (make-num-pair a b) (+ (* 101 a) b))
(define (first-num pair) (quotient pair 101))
(define (second-num pair) (remainder pair 101))
```

c. Write constructors and selectors which represent a num-pair as a procedure.

```
> (procedure? (make-num-pair 50 60))
#t
> (second-num (minmax (89 94 83 95 91 50)))
95
```

Solution:

```
(define (make-num-pair a b)
  (lambda (m)
    (if (= m 0) a b)))
(define (first-num pair) (pair 0))
(define (second-num pair) (pair 1))
```