CS450

Structure of Higher Level Languages

Lecture 3: Data structures

Tiago Cogumbreiro

Happy new year!

🐷

Go Pats!

Homework Assignment 1

February 12 at 5:30pm

(Covers Lectures 1, 2, and 3)

Sorry, but no late submissions will be accepted!

HW1 Errata

Typo in the example listed in Exercise 1.b

The example should be:

```
(define ex2
  (list
    (* 3.14159 (* 10 10))
    (* 3.14159 100)
    314.159))
```

HW1 Errata

Typo in the example listed in Exercise 2

The example should be:

```
boolean ex3(double x, float y) {
    return ...;
}
```

- Since Racket is a dynamically typed language, you are **not expected to use types in your solution.**
- Use a **function definition** and not a basic definition

On homework assignment 1

- Exercises 1 and 2 must be **syntactically** equivalent, not just *semantically*. $2 + 3$ is syntactically different than $3 + 2$!
- You are responsible for submitting a solution that runs with Racket 7 and for writing tests that exercise the correctness of your solution.
- A Racket program with **syntax error gets 0 points**.
- A Racket program that **does *not* follow the homework template likely gets 0 points**.
- If you see the error message below, please contact me.

```
The autograder failed to execute correctly. Contact your course staff
for help in debugging this issue. Make sure to include a link to this
page so that they can help you most effectively.
```

Today we will learn about...

- data structures as constructors and accessors
- pairs
- lists
- user-data structures
- serializing code with `quote`

> Cover up until Section 2.2.1 of the SICP book. Try out the interactive version of section 2.1 of the SICP book.

Data structures

Data structures

When presenting each data structure we will introduce two sets of functions:

- **Constructors:** functions needed to build the data structure
- **Accessors:** functions needed to retrieve each component of the data structure. Also known as **selectors.**

> Each example we discuss is prefaced by some unit tests. We are following a Test Driven Development methodology.

Pairs

The pair datatype

Constructor: cons

```
expression = ··· | pair
pair = (cons expression expression )
```

Function `cons` constructs a pair with the evaluation of the arguments, which Racket prints as: `'(v1 . v2)`

Example

```
#lang racket
(cons (+ 1 2) (* 2 3))
```

Output

```
$ racket pair.rkt
'(3 . 6)
```

The pair datatype

Accessors: car and cdr

- Function `car` returns the left-hand-side element (the first element) of the pair.
- Function `cdr` returns the right-hand-side element (the second element) of the pair.

Example

```
#lang racket
(define pair (cons (+ 1 2) (* 2 3)))
(car pair)
(cdr pair)
```

```
$ racket pair.rkt
3
6
```

Pairs: example 1

Swap the elements of a pair: `(pair-swap p)`

Spec

```
; Paste this at the end of "pairs.rkt"
(require rackunit)
(check-equal?
  (cons 2 1)
  (pair-swap (cons 1 2)))
```

Pairs: example 1

Swap the elements of a pair: (pair-swap p)

Spec

```racket
; Paste this at the end of "pairs.rkt"
(require rackunit)
(check-equal?
  (cons 2 1)
  (pair-swap (cons 1 2)))
```

Solution

```racket
#lang racket
(define (pair-swap p)
  (cons
    (cdr p)
    (car p)))
```

Pairs: example 2

Point-wise addition of two pairs: `(pair+ l r)`

Unit test

```
(require rackunit)
(check-equal?
  (cons 4 6)
  (pair+ (cons 1 2) (cons 3 4)))
```

Pairs: example 2

Point-wise addition of two pairs: `(pair+ l r)`

Unit test

```
(require rackunit)
(check-equal?
  (cons 4 6)
  (pair+ (cons 1 2) (cons 3 4)))
```

Solution

```
#lang racket
(define (pair+ l r)
  (cons (+ (car l) (car r))
        (+ (cdr l) (cdr r))))
```

Pairs: example 3

Lexicographical ordering of a pair

```
(require rackunit)
(check-true (pair< (cons 1 3) (cons 2 3)))
(check-true (pair< (cons 1 2) (cons 1 3)))
(check-false (pair< (cons 1 3) (cons 1 3)))
(check-false (pair< (cons 1 3) (cons 1 0)))
```

Pairs: example 3

Lexicographical ordering of a pair

```
(require rackunit)
(check-true (pair< (cons 1 3) (cons 2 3)))
(check-true (pair< (cons 1 2) (cons 1 3)))
(check-false (pair< (cons 1 3) (cons 1 3)))
(check-false (pair< (cons 1 3) (cons 1 0)))
```

```
#lang racket
(define (pair< l r)
  (or (< (car l) (car r))
      (and (= (car l) (car r))
           (< (cdr l) (cdr r)))))
```

Lists

Lists

Constructor: list

expression = ··· | *list*
list = `(list` *expression** `)`

Function call `list` constructs a list with the evaluation of a possibly-empty sequence of expressions `e1` up to `en` as values `v1` up to `vn` which Racket prints as: `'(v1 ... v2)`

```
#lang racket
(list (+ 0 1) (+ 0 1 2) (+ 0 1 2 3))
(list)
```

```
$ racket list-ex1.rkt
'(1 3 6)
'()
```

Accessing lists

Accessor: `empty?`

You can test if a list is empty with function **`empty?`**. An empty list is printed as `'()`.

```
#lang racket
(require rackunit)
(check-false (empty? (list (+ 0 1) (+ 0 1 2) (+ 0 1 2 3))))
(check-true (empty? (list)))
```

Lists are linked-lists of pairs

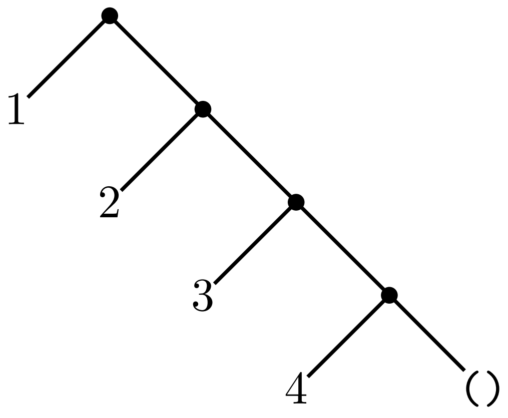
Accessors: car, cdr

Lists in Racket are implemented as a linked-list using pairs terminated by the empty list `'()`.

- Function `car` returns the head of the list, given a nonempty list.
  `car` originally meant Contents of Address Register.
- Function `cdr` returns the tail of the list, given a nonempty list.
  `cdr` originally meant Contents of Decrement Register.

```
(list 1 2 3 4)
```

Graphical representation



Textual representation

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

Lists are built from pairs example

Constructor `empty`

```
#lang racket
(require rackunit)
(check-equal?
  (cons 1
    (cons 2
      (cons 3
        (cons 4 empty)))) (list 1 2 3 4))
```

Lists: example 1

Summation of all elements of a list

Spec

```
(require rackunit)
(check-equal? 10 (sum-list (list 1 2 3 4)))
(check-equal? 0 (sum-list (list)))
```

Lists: example 1

Summation of all elements of a list

Spec

```
(require rackunit)
(check-equal? 10 (sum-list (list 1 2 3 4)))
(check-equal? 0 (sum-list (list)))
```

Solution

```
#lang racket
; Summation of all elements of a list
(define (sum-list l)
  (cond [(empty? l) 0]
        [else (+ (car l) (sum-list (cdr l)))]))
```

Lists: example 2

Returns a list from n down to 1

Spec

```
(require rackunit)
(check-equal? (list) (count-down 0))
(check-equal? (list 3 2 1) (count-down 3))
```

Lists: example 2

Returns a list from n down to 1

Spec

```
(require rackunit)
(check-equal? (list) (count-down 0))
(check-equal? (list 3 2 1) (count-down 3))
```

Solution

```
#lang racket
(define (count-down n)
  (cond [(≤ n 0) (list)]
        [else (cons n (count-down (- n 1)))]))
```

Lists: example 3

Point-wise pairing of two lists

Spec

```
(require rackunit)
(check-equal? (list (cons 3 30) (cons 2 20) (cons 1 10))
              (zip (list 3 2 1) (list 30 20 10)))
(check-equal? (list (cons 3 30) (cons 2 20) (cons 1 10))
              (zip (list 3 2 1) (list 30 20 10 5 4 3 2 1)))
(check-equal? (list (cons 3 30) (cons 2 20) (cons 1 10))
              (zip (list 3 2 1 90 180 270) (list 30 20 10)))
```

Lists: example 3

Point-wise pairing of two lists

Solution

```
#lang racket
(define list-add cons) (define pair cons)
(define (zip l1 l2)
  (cond [(empty? l1) (list)]
        [(empty? l2) (list)]
        [else
          (list-add
            (pair (car l1) (car l2))
            (zip (cdr l1) (cdr l2)))]))
```

User data-structures

We can represent data-structures using pairs/lists.

For instance, let us build a 3-D point data type.

```
(require rackunit)
(define p (point 1 2 3))
(check-true (point? p))
(check-equal? (list 1 2 3) p)
(check-equal? 1 (point-x p))
(check-equal? 2 (point-y p))
(check-equal? 3 (point-z p))
(check-true (origin? (list 0 0 0)))
(check-false (origin? p))
```

User data-structures

We can represent data-structures using pairs/lists.

For instance, let us build a 3-D point data type.

```
(require rackunit)
(define p (point 1 2 3))
(check-true (point? p))
(check-equal? (list 1 2 3) p)
(check-equal? 1 (point-x p))
(check-equal? 2 (point-y p))
(check-equal? 3 (point-z p))
(check-true (origin? (list 0 0 0)))
(check-false (origin? p))
```

```
; Constructor
(define (point x y z) (list x y z))
(define (point? x)
  (and (list? x)
       (= (length x) 3)))
; Accessors
(define (point-x pt) (car pt))
(define (point-y pt) (car (cdr pt)))
(define (point-z pt) (car (cdr (cdr pt))))
; Alternative solution for accessors:
; (define point-x car)
; (define point-y cadr)
; (define point-z caadr)
(define (origin? p) (equal? p (list 0 0 0)))
```

On data-structures

- We only specified **immutable** data structures
- The effect of updating a data-structure is encoded by **creating/copying** a data-structure
- This pattern is known as a persistent data structure

Serializing code

Quoting: a specification

Function `(quote e)` *serializes* expression `e`. Note that expression `e` is **not** evaluated.

- A variable `x` becomes a symbol `'x`. You can consider a *symbol* to be a special kind of string in Racket. You can test if an expression is a symbol with function `symbol?`
- A function application $(e_1 \cdots e_n)$ becomes a list of the serialization of each expression e_i.
- Serializing a `(define x e)` yields a list with symbol `'define` and the serialization of `e`. Serializing (**`define`** $(x_1 \cdots x_n)$ e) yields a list with symbol `'define` followed by a nonempty list of symbols $'x_i$ followed by serialized e.
- Serializing (**`lambda`** $(x_1 ... x_n)$ e) yields a list with symbol `'lambda`, followed by a possibly-empty list of symbols x_i, and the serialized expression e.
- Serializing a (**`cond`** $(b_1\ e_1) \cdots (b_n\ e_n)$) becomes a list with symbol `'cond` followed by a serialized branch. Each branch is a list with two components: serialized expression b_i and serialized expression e_i.

Quoting exercises:

- We can write `'term` rather than `(quote term)`
- How do we serialize term `(lambda (x) x)` with `quote`?
- How do we serialize term `(+ 1 2)` with quote?
- How do we serialize term `(cond [(> 10 x) x] [else #f])` with quote?
- *Can we serialize a syntactically invalid Racket program?*

Quoting exercises:

- We can write `'term` rather than `(quote term)`
- How do we serialize term `(lambda (x) x)` with `quote`?
- How do we serialize term `(+ 1 2)` with quote?
- How do we serialize term `(cond [(> 10 x) x] [else #f])` with quote?
- *Can we serialize a syntactically invalid Racket program?* **No!** You would not be able to serialize this expression `(`. Quote only accepts a S-expressions (parenthesis must be well-balanced, identifiers must be valid Racket identifiers, number literals must be valid).
- *Can we serialize an invalid Racket program?*

Quoting exercises:

- We can write `'term` rather than `(quote term)`
- How do we serialize term `(lambda (x) x)` with `quote`?
- How do we serialize term `(+ 1 2)` with quote?
- How do we serialize term `(cond [(> 10 x) x] [else #f])` with quote?
- *Can we serialize a syntactically invalid Racket program?* **No!** You would not be able to serialize this expression `(`. Quote only accepts a S-expressions (parenthesis must be well-balanced, identifiers must be valid Racket identifiers, number literals must be valid).
- *Can we serialize an invalid Racket program?* **Yes.** For instance, try to quote the term: `(lambda)`

Quote example

```
#lang racket
(require rackunit)
(check-equal? 3 (quote 3))  ; Serializing a number returns the number itself
(check-equal? 'x (quote x)) ; Serializing a variable named x yields symbol 'x
(check-equal? (list '+ 1 2) (quote (+ 1 2))) ; Serialization of function as a list
(check-equal? (list 'lambda (list 'x) 'x) (quote (lambda (x) x)))
```