CS450

Structure of Higher Level Languages

Lecture 3: Function declarations, pairs, lists

Tiago Cogumbreiro

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

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

Solution

```
#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

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

Solution

```racket
#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

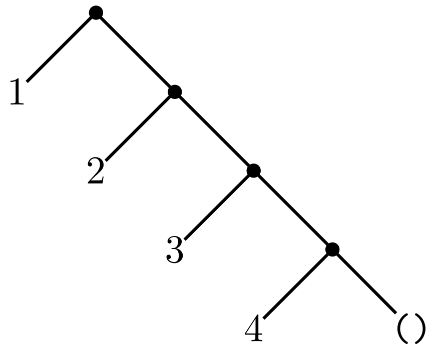
Accessors: car, cdr

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))
```

User data-structures

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) (first pt))
(define (point-y pt) (second pt))
(define (point-z pt) (third pt))
; Example function
(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 e_i.
- Serializing a (`define x e`) yields a list with: symbol `'define`, the serialization of variable `x`, and the serialization of `e`. Serializing $(\texttt{define}\ (x_1 \cdots x_n)\ e)$ yields a list with symbol `'define` followed by a nonempty list of symbols $'x_i$ followed by serialized e.
- Serializing $(\texttt{lambda}\ (x_1 \ldots x_n)\ e)$ yields a list with symbol `'lambda`, followed by a possibly-empty list of symbols x_i, and the serialized expression e.
- Serializing a $(\texttt{cond}\ (b_1\ e_1) \cdots (b_n\ e_n))$ 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)))
(check-equal? (list 'define (list 'x)) (quote (define (x))))
```

Manipulating quoted terms

Specification

function-dec = `(` `lambda` `(` *variable** `)` *term*+`)`

- How do we get the parameter list?
- How do we get the body?
- What does *variable** mean?
- What does *term*+ mean?

On HW1 Q.4

- The input format of the quoted term are **precisely** described in the slides of Lecture 3
- You do **not** need to test recursively if the terms in the body of a function declaration or definition are valid.
- A list, with one symbol `lambda` followed by zero or more symbols, and one or more terms.

Racket spec

HW1: Question 4

```
program = #lang racket term*

term = definition | expression

definition = basic-def | function-def
basic-def = ( define identifier expression )
function-def = ( define (variable+ ) term+ )

expression = value | variable | function-call | function-decl | ···
value = number | ···
function-call = ( expression+ )
function-dec = ( lambda ( variable* ) term+)
```

Tips for solving HW1

HW1: Question 4

1. Do all parts except `lambda?`, `define?`, and `define-func?`.
2. Write `lambda?`
3. Write `define-func?`
4. Write `define?`

More tips

- Function application is simpler than it seems
- All acceptance-tests from `define-func?` should pass in `define?`

Being successful in CS 450

Forum questions policy

1. Private questions (Discord) have the **lowest** priority
2. Instructor/TAs cannot comment on why a student's submission is not working
3. If a student lists which test-cases have been used, then the instructor/TAs can give more inputs or test cases
4. Private questions regarding code must always be accompanied with the URL of latest Gradescope submission
5. Students cannot share their solutions (partial/full) in public posts

The final grade is given by the instructor

(not by the autograder)

We are grading the correctness of a solution

The autograder only ***approximates*** your grade

- Students may request for manual grading
- Grading partial solutions automatically is ***hard***:
  - Solution may be using disallowed functions
  - Solution may be tricking the autograder system

Tip #1: avoid fighting the autograder

1. **It's not personal:** The autograder is not against you
2. **It's not picky:** The autograder is not against one specific solution
3. **Correlation is not causation:** Having a colleague with the same problem as you have, does **not** imply that the autograder is wrong
4. **Spend your time wisely**: don't spend it thinking the autograder is wrong

Instead, discuss

1. **Use the autograder for your benefit:** submit solution to test your hypothesis
2. **Think before resubmitting:** try explaining your solution to someone
3. **Ask before resubmitting:** write test cases and discuss those test cases with others

5% of your grade is participation, so discuss!

Tip #2: participate

5% of your grade is participation

> Software engineering and academic life is about ***communication***: you are expected to interact to solve your homework assignments.

1. Exercises are explained succinctly on purpose: **ask questions** to know more
2. Exercises have few test cases on purpose: **share test-cases** to know more

Make time in your schedule to interact

Tip #3: time management

Work on your homework assignment incrementally

- after each class you can solve a new exercise (with few exceptions)
- when you get stuck in an exercise: (1) **ask** in our forum, and while you are waiting (2) **continue working** on other exercises
- don't leave everything to the weekend before submission

Tip #4: learn to ask questions

The better your formulate a question,

The faster you will get an answer

Ask yourself

1. Which slides do you think the exercise relates to?
2. Which test-cases have you tried that counter your intuition?

Asking question

1. Describe the problem you are having (relate exercise and lessons)
2. Explain your attempts at fixing the problem (list used tests)