Functional Programming
Lecture 2

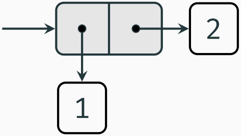
Rostislav Horčík [1] Niklas Heim Tomas Votroubek [2]

Czech Technical University in Prague
Faculty of Electrical Engineering
[1]xhorcik@fel.cvut.cz
[2]tomas.votroubek@fel.cvut.cz

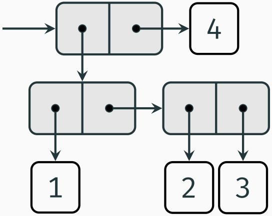
Lists

Pairs

Enable the construction hierarchical data structures.



`(cons 1 2)`



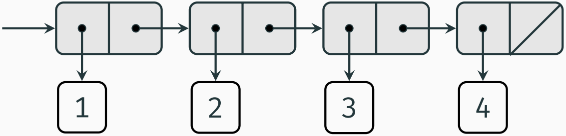
`(cons (cons 1 (cons 2 3)) 4)`

```
(cons 1 2) => '(1 . 2)
(car (cons 1 2)) => 1
(cdr (cons 1 2)) => 2
```

We can create pairs composed of *any* data types, not only basic ones.

Lists

Lists are represented as sequences of linked pairs with the empty list `'()` at the end.




```
(cons 1 (cons 2 (cons 3 (cons 4 '()))))
(list 1 2 3 4)
(car (list 1 2 3)) => 1       ; also: first
(cdr (list 1 2 3)) => (2 3)   ; also: rest
(caddr (list 1 2 3)) => 3
(list? (list 1 2 3)) => #t
(empty? (list 1 2 3)) => #f
```

Note that S-expressions are lists.

Construction

Quoting - no evaluation of arguments

```
'(1 (+ 1 2)) => '(1 (+ 1 2))
; expands to: (quote (1 (+ 1 2)))
```

Quasiquoting - evaluated parts explicitly marked (“escaped”)

```
`(* 1 ,(+ 1 1) 2) => '(* 1 2 2)
; full form: (quasiquote (* 1 (unquote (+ 1 1)) 2))
```

Listing - evaluates all arguments

```
(list 1 2 (+ 1 2)) => '(1 2 3)
```

Appending - copies first `n - 1` lists

```
(append '(1) '(2) '(3 4)) => '(1 2 3 4)
```

Equalities

Use the procedure `=` to check for numerical equivalence

```
(= 1 1.0 5/5) ; => #t
(= -0.0 0.0 0 0+0i) ; => #t
```

Use `eq?` to check if values refer to the same object

```
; typically a cheap pointer comparison
(define x '(a b c))
(define y '(a b c))
(eq? x x) ; => #t
(eq? x y) ; => #f
```

Use `equal?` to check for structural equivalence

```
; typically recursively defined
(equal? '(a 1 (2 3)) '(a 1 (2 3))) ; => #t
```

Example — filtering lists

Suppose we want to define a function filtering a given value out of a given list.

```
(my-filter val lst)
```

```
(my-filter 'a '(1 a 2 a)) => '(1 2)
```

Example — Filter

```
(define (my-filter-3 pred lst [acc '()])
  (cond
    [(null? lst) (reverse acc)]
    [(pred (car lst))
     (my-filter-3 pred (cdr lst)
                  (cons (car lst) acc))]
    [else
     (my-filter-3 pred (cdr lst) acc)]))
```

Lambda abstraction

A construction for creating anonymous functions

```
(lambda (arg1 ... argN) <exp>)
```

Define for functions is an abbreviation

```
(define (square x) (* x x))
```

is the same as

```
(define square (lambda (x) (* x x)))
```

Examples

```
(filter (lambda (x) (< x 5))
        '(1 7 3 8)) => '(7 8)

(filter (lambda (l) (not (null? l)))
        '((a b) (5) ())) => '((a b) (5))
```

Another useful function iterating through a list is `map`.
`(map f lst)` applies `f` to each element of `lst`:

```
(map (lambda (x) (* x x)) '(1 2 3)) => '(1 4 9)
(map cdr '((a b c) (1))) => '((b c) ())
```

Bad maxlist

```
(define (bad-maxlist lst)
  (if (null? lst)
      -inf.0
      (if (> (car lst) (bad-maxlist (cdr lst)))
          (car lst)
          (bad-maxlist (cdr lst)))))
```

Inefficient tree recursive function making two recursive calls!

Local definitions

Reuse of a computation/result is often required

```
(let ([<var1> <exp1>]
      [<var2> <exp2>])
  <body-using-var1-var2>)
```

```
(let* ([<var1> <exp1>]
       [<var2> <exp2-using-var1>])
  <body-using-var1-var2>)
```

Better maxlist

```
(define (better-maxlist lst)
  (if (null? lst)
      -inf.0
      (let ([m (better-maxlist (cdr lst))])
        (if (> (car lst) m) (car lst) m))))
```

Tail recursive version is the best:

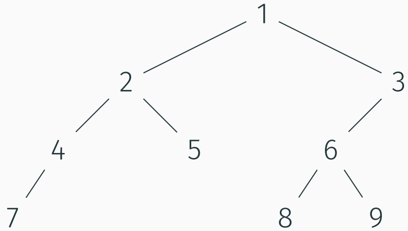
```
(define (best-maxlist lst [acc -inf.0])
  (cond
    [(null? lst) acc]
    [(> (car lst) acc)
     (best-maxlist (cdr lst) (car lst))]
    [else (best-maxlist (cdr lst) acc)]))
```

Trees

Trees

Trees can be represented by nested lists. E.g. binary trees

```
'(<data> <left> <right>)
```




```
'(1 (2 (4 (7 #f #f) #f)
       (5 #f #f))
    (3 (6 (8 #f #f)
          (9 #f #f))
        #f))
```

Tree recursion

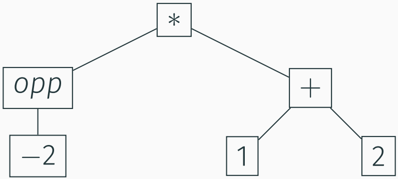
```
(define get-data car)
(define get-left cadr)
(define get-right caddr)

(define (find pred tree)
(if tree
    (let* ([data (get-data tree)]
           [left (find pred (get-left tree))]
           [right (find pred (get-right tree))]
           [both (append left right)])
      (if (pred data)
          (cons data both)
          both))
    '()))
```

Evaluate an algebraic expression

Expression - a number or a list with an operator followed by arguments, which are expressions

`13`, `'(+ 1 2 3)`, `'(* (opp -2) (+ 1 2))`




```
(eval-expr '(* (opp -2) (+ 1 2)))
=> (eval-expr '(* 2 3))
=> (eval-expr 6)
=> 6
```

Expression evaluation

```
(define (eval-expr e)
  (if (number? e)
      e
      (let ([op (car e)]
            [children (map eval-expr (cdr e))])
        (cond
          [(eq? op '+) (apply + children)]
          [(eq? op '-) (apply - children)]
          [(eq? op '*) (apply * children)]
          [(eq? op 'opp) (- (car children))]))))
```

`apply` takes a function and a list of arguments, “unwraps” the arguments - `(apply + '(1 2 3)) => (+ 1 2 3)`

Unit testing

Rackunit

Racket has a built-in unit-testing framework

```
(require rackunit)
```

Provides tools for checks, test cases and test suites

Checks check conditions, report failure if not met

Test cases are named collections of sequential checks

Test suites are named collection of test cases

Checks

Most common - checking for equality

```
(check-equal? '(a b) '(a b))
(check-equal? '(1 a) '(a) "optional message")
```

```
--------------------
; FAILURE
; lec02.rkt:6:0
name:       check-equal?
location:   lec02.rkt:6:0
message:    "optional message"
actual:     '(1 a)
expected:   '(a)
--------------------
```

Other versions - `check-eq?`, `check-within`, `check-true`, ...

Consult the documentation for the rest

Test cases

`test-case` starts a block, if any check fails, the rest are not run

```
(test-case "filter tests"
  (check-equal? (my-filter-tr 2 '(1 2 3)) '(1 2 3))
  (check-equal? (my-filter-tr 1 '(1 2 3)) '(1 2 3)))
```

```
--------------------
filter tests
; FAILURE
; lec02.rkt:33:0
name:       check-equal?
location:   lec02.rkt:33:4
actual:     '(1 3)
expected:   '(1 2 3)
--------------------
```

Shortcuts available - `test-equal?` is like `check-equal?` with an extra string argument for the name

Test submodule

Tests should not be in the same scope as other code

Top-level tests are run when loading the file

Unnecessary dependency on `rackunit`

Solution - separate test module

`module+` defines a module “by parts”, while inheriting identifiers from parent module

module+ example

```
(define (add a b) (+ a b))
(module+ test
  ; only test submodule will depend on rackunit
  (require rackunit)
  ; not running during module import
  (check-equal? (add 1 2) 3))

(define (sub a b) (- a b))
; tests are very close to code!
(module+ test
  (check-equal? (sub 1 2) -1))
```

Running tests

From DrRacket: “Run” button (`Ctrl+R`) automatically runs the submodule `test`, if defined

From terminal: `raco test <file>.rkt`

What have we learned?

- Pairs are used to construct more complex data structures.
- Lists are built from linked pairs.
- Trees can be represented by nested lists.
- There are different kinds of equalities.
- Abstract functions are defined by lambda.
- Local definitions are useful for reusing of a computation.
- Tests go into their own module, our tools run them for us.