Introduction to Racket

Advanced Functional Programming

Jean-Noël Monette

November 2013

Racket is

- a programming language — a dialect of Lisp and a descendant of Scheme;
- a family of programming languages — variants of Racket, and more;
- a set of tools — for using a family of programming languages.

Getting Started

Installation

- Download at http://www.racket-lang.org/download
- Run the self-extracting archive.
- type in `racket` (REPL) or `drracket` (IDE).

Documentation at http://docs.racket-lang.org

- Tutorials
- Comprehensive guide
- Reference
- And a lot more...

A First Example

```
(define (fact x)
  (cond [(> x 0) (* x (fact (sub1 x)))]
        [else 1]))
(fact 10)
```

Syntax

The syntax is uniform and made of s-expressions.

An s-expression is an atom or a sequence of atoms separated by spaces and enclosed in parenthesis.

Square brackets `[]` and braces `{}` can be used instead of parenthesis (as long as they match per type).

There are a few syntactic shortcuts such as e.g. `'` `,` `#` .

Choosing your language

Always start your files with `#lang racket` to define the language.

We will mainly use `racket` as a language but others do exist.

Examples: typed/racket, slideshow, scribble, ...

Racket Basics

Greedy Evaluation: Arguments to a procedure are evaluated before the procedure.

Dynamically typed: `(fact "coucou")` is a runtime error, not a compilation error.

but

Macros can emulate lazyness.

Contracts can help catch "type errors"

Comments

`;` starts a comment to the end of the line.

`;;` is usually used to mark more important comments.

`#;` comments the following s-expression.

Procedure calls

In between parenthesis, the first expression must evaluate to the function, the remaining ones are the arguments.

```
(+ 1 2 3 4)
(string? "Hello")
(equal? 3 "row")
```

Definitions

```
(define x 5)
(define (inc x) (+ x 1)) ; predefined as add1
(define 3*2 (* 3 2))
```

Identifiers can be composed of any characters but `()[]{}",'`;#|\`

Identifiers usually start by a lower case letter.

Compound names are usually separated by `-` , e.g. `sum-two-numbers`

Numbers

- Arbitrary large integers and (exact) rationals `(/ 1 2)`
- Floating point numbers `(+ 2.0 -inf.0 +nan.0)`
- Complex numbers `2+1/2i`
- test functions:
```
number? integer? complex?
rational? real? byte?
```

Booleans

Two booleans literals: `#t` and `#f`

Everything not `#f` is considered as true in conditions.

`(boolean? x)` tells whether x is a boolean value.

`(and)` and `(or)` take any number of arguments (including zero) and short-circuit.

For instance , `(or 3 #f)` returns `3`.

Characters and Strings

Characters are Unicode scalar values.

```
#\A
```

Converting to/from integers with `char->integer` and `integer->char`

Strings are sequences of characters (in between double quotes).

```
"Hello, World!"
```

```
(string-length (string-append "Hello" ", " "world!"))
```

Comparing Objects

There are (at least) four comparison procedures in Racket.

- `=` compares number numerically.

```
(= 1 1.0) => #t
(= 0.0 -0.0) => #t
(= 1/10 0.1) => #f
```

- `eq?` compares objects by reference. Fast but not reliable in all cases.

```
(eq? (cons 1 2) (cons 1 2)) => #f
(let ([x (cons 1 2)]) (eq? x x)) => #t
```

- `eqv?` is like `eq?` except for numbers and characters.

```
(eq? (expt 2 100) (expt 2 100)) => #f
(eqv? (expt 2 100) (expt 2 100)) => #t
```

Comparing Objects (2)

- `equal?` is like `eqv?` except for strings and decomposable structures (lists, hash-table, structures).

```
(eqv? "banana" (string-append "ban" "ana")) => #f
(equal? "banana" (string-append "ban" "ana")) => #t
(equal? (list 1 2) (cons 1 (cons 2 '()))) => #t
```

Use preferably `equal?` as it is more reliable, and `=` for (exact) numeric comparisons.

Conditionals

```
(if (> 1 0) "Good" 'nogood)

(cond [(not (number? x)) "NaN"]
      [(> x 0) "Pos"]
      [(< x 0) "Neg"]
      [else "Zero"])
```

Anonymous functions

`(lambda (x) (+ x 1))` defines an anonymous function.

```
(define inc (lambda (x) (+ x 1)))
(inc 4)
((lambda (x) (+ x 1)) 4)
```

Function Body

A function body is composed of any number of (local) definitions followed by any number of expressions. The return value of the function is the value of the last expression.

Internal defines can be mutually recursive.

```
(define (sum a b)
  (define (suma c) (+ a c))
  (suma b))
```

Local declarations

`let` declares local variables. It evaluates all the expressions before binding them.

```
(let ([x y] [y x])
  (cons x y))
```

In a `let*`, the first bindings are available to the next ones.

```
(let* ([x y] [y x])
  (cons x y))
```

In `letrec`, all bindings are available to each other (mainly for mutually recursive local functions).

```
(letrec ([x y] [y x])
  (cons x y))
```

Local declaration of functions

```
(let loop () (loop))
```

This creates a function called loop and executes it.

Below, sum-help is a function of two (optional) arguments

```
(define (sum x)
  (let sum-help ([x x] [res 0])
    (cond [(= x 0) res]
          [else (sum-help (sub1 x) (+ res x))])))
```

Lists

```
(list 1 2 3 4)
(define x (list 1 2 3 4))
(car x) (first x)
(cdr x) (rest x)
null empty
(cons 0 x)
(cons? x) (pair? x)
(null? x) (empty? x)
(length (list 9 8 7))
(map add1 (list 1 2 3 4))
(andmap string? (list "a" "b" 0))
(filter positive? (list -1 0 1 2 -5 4))
(foldl + 0 (list 1 2 3))
```

Cons revisited

`(cons 1 2)` is valid code but it does not produce a proper list.

`(list? x)` tells if it is a proper list (in constant time).

This is a difference between strongly typed code (such as SML) and Racket.

Quotation and symbols

`(list '+ 2 3 4)` produces a list `'(+ 2 3 4)` that looks like a procedure application but is not evaluated and preceded by `'`.

The s-expression is *quoted* and considered as data.

`quote` quotes its argument without evaluating it.

`(quote (map + 0 "cool"))` is simply a list of four elements.

`(quote map)` creates a *symbol* `'map` that has nothing to do with the identifier `map` (except the name).

One can directly write `'` instead of `quote`.

`quote` has no effect on literals (numbers, strings)

Symbols can be also created with `(string->symbol "aer")` or `(gensym)`

Quasiquoting and unquoting

Quasiquoting is like quoting but it can be escaped to evaluate part of the expression.

```
(quasiquote (1 2 (unquote (+ 1 2))
                 (unquote (- 5 1))))
```

Or equivalently:

```
`(1 2 ,(+ 1 2) ,(- 5 1)) => (1 2 3 4)
```

`,@` or `unquote-splicing` also decompose a list.

```
`(1 2 ,@(map add1 '(2 3))) => (1 2 3 4)
```

```
`(1 2 ,(map add1 '(2 3))) => (1 2 (3 4))
```

Eval

(Quasi-)quoted s-expressions can be evaluated using `eval`

```
(define sum ''(+ 1 2 3 4))
(display sum)
(display (eval sum))
(display (eval (eval sum)))
```

Apply

`apply` applies a function to a list of arguments.

```
(apply + '(1 2 3))
```

With more than one argument, the first ones are put in front of the list.

```
(apply + 1 '(2 3))
```

```
(apply append '(1 2) '((3 4)))
```

Function arguments

Function can have a variable number of arguments.

```
((lambda all (apply + (length all) all)) 4 3)
((lambda (x . rest) (apply + x rest)) 1 4 5)
```

There can also be optional and keywords arguments.

```
((lambda (x [y 1]) (+ x y)) 4)
((lambda (x #:y y) (+ x y)) #:y 1 4)
((lambda (x #:y [y 1]) (+ x y)) 4)
(lambda (x #:y y . rest) ...)
```

Curried and Higer-Order Functions

Short way to define curried functions.

```
(define ((add x) y) (+ x y))
(define add3 (add 3))
(add3 4)
((add 10) 20)
```

A simple composition of functions

```
(define ((comp f g) . x)
  (f (apply g x)))
(define add2 (comp add1 add1))
(add2 5)
```

Multiples values

A function can return several values at the same time with `values`

```
(values 1 2 3)
```

To bind those values to identifiers, one can use `define-values`, or `let-values`, or one of the other variants (e.g. `let-values`).

```
(define-values (x y z) (values 1 2 3))
(define-values (five) (add1 4))
```

Simple matching: case

`case` matches a given value against fixed values (with `equals?`)

```
(case v
  [(0) 'zero]
  [(1) 'one]
  [(2) 'two]
  [(3 4 5) 'many]
  [else 'too-much])
```

If nothing matches and there is no else clause, the result is `#<void>`.

More Matching: match

`match` matches a given value against patterns.

Patterns can be very complex (using e.g. and, or, not, regexp, ...).

```
(match x
  ['() "Empty"]
  [(cons _ '()) "Contains 1 element"]
  [(cons a a) "A pair of identical elements"]
  [(and (list y ...) (app length len))
   (string-append "A list of " (number->string len) " elements")]
  [(app string? #t)
   (=> fail)
   (when (= 0 (string-length x)) (fail)) "A non-empty String"]
  [else "Something else"])
```

If nothing matches, a `exn:misc:match?` exception is raised.

Assignment

The value bound to an identifier can be modified using `set!`

```
(define next-number!
  (let ([n 0])
    (lambda ()
      (set! n (add1 n))
      n)))
(next-number!)
(next-number!)
```

Use with care!

Guarded Operations

```
(when with-print (print x))
(unless finished (set! x y))
```

Mainly useful to enclose side-effect only code.

What is the return value of the following code?

```
(when #f #t)
```

Also produced by the procedure `(void)`.

Vectors

- Fixed length array with constant-time access to the elements.
- Created as a list but with a # instead of the quotation mark or with the function `vector`.

```
(vector "a" 1 #f)
```

- `(vector-ref a-vec num)` accesses the numth element of `a-vec` (starting from zero).
- They can be modified with `vector-set!`.

```
(vector-set! a-vec num new-val)
```

Hash Tables

Immutable hash-tables

```
(define ht (hash "a" 3 "b" 'three))
(define ht2 (hash-set ht "c" "three"))
(hash-ref ht2 "c")
(hash-has-key? ht "c")
```

Mutable hash-tables

```
(define ht (make-hash '(("A" "Appel")
                        ("B" "Banana"))))
(hash-set! ht "A" "Ananas")
(hash-ref ht "A")
```

New datatypes

`(struct point (x y))` produces a new structure that can be used as follows:

```
(point 1 2)
(point? (point 1 2))
(point-x (point 1 2))
(struct point (x y) #:transparent #:mutable)
```

The last line creates a structure where the internal can be accessed (e.g. recursively by equals?), and modified.

Exceptions

Exceptions are raised upon runtime errors. To catch exceptions use an exception handler.

```
(with-handlers ([exn:fail:contract:divide-by-zero?
                 (lambda (exn) +inf.0)])
  (/1 0))
```

The first argument is a list of pairs, whose first element is a test to check the type of exception and second is what to do with it.

The check `exn:fail?` catches all exceptions.

`(error "string")` creates and raises a generic exception.

`(raise 5)` raises anything as an exception.

Threads

`thread` runs the given procedure in a separate thread and returns the thread identifier.

```
(define t (thread (lambda ()
                    (let loop ()
                      (display "In thread")
                      (sleep 1)
                      (loop)))))
(sleep 10)
(kill-thread t)
```

Threads are lightweight and run inside the same physical process.

Threads and Channels

Threads can collaborate (among others) throuch message passing with `thread-send` and `thread-receive`.

```
(define t0 (current-thread))
(define t1
  (thread (lambda ()
            (define v (thread-receive))
            (thread-send t0 (add1 v)))))
(thread-send t1 5)
(display (thread-receive))
```

Comprehensions

Racket provides many looping constructs.

```
(for ([i '(1 2 3 4 5)])
  (display i))
(for/list ([i '(1 2 3 4 5)])
  (modulo i 3))
(for/and ([i '(1 2 3 4 5)])
  (> 0))
(for/fold ([sum 0])
  ([i '(1 2 3 4 5)])
  (+ sum i))
```

Nested and Parallel Comprehensions

`for` and variations iterate over several sequences in parallel.

`for*` and variations act as nested for’s.

```
(for ([i '(1 2 3 4)]
      [j '(1 2 3)])
  (display (list i j)))
(for* ([i '(1 2 3 4)]
       [j '(1 2 3)])
  (display (list i j)))
```

Iterable Sequences

`for` and variations can iterate over different kinds of sequences, not only lists.

```
(for ([(k v) (hash 1 "a" 2 "b" 3 "c")]
      [i 5]) ; range 0 to 4
  (display (list i k v)))
(for ([i "abc"]
      [j (in-naturals)])
  (display (cons i j)))
```

Performance of Sequences.

To make the comprehension fast, one should "declare" the type of each sequence.

```
(for ([i (in-range 10)]
      [j (in-list '(1 2 3 4 5 6))]
      [k (in-string "qwerty")])
  (display (list i j k)))
```

Much More in Racket

- Classes and Objects
- Units and Signatures
- Input/Output
- RackUnit
- Graphical, Network, Web, DB, ... Libraries
- Other Languages (typed Racket, Scribble, ...)

Wrap-up

- Everything you can expect from a modern functional language.
- Minimal syntax
- Code = Data

Next Lectures

- Continuations
- Macros
- Modules and Contracts
- Making your own language

Voluntary Exercises

- Redefine `map` and `length` as recursive functions.
- Define a procedure `(primes n)` that returns a list of the `n` first prime numbers.
- Define a procedure `(perms xs)` that returns all permutations of the list `xs`.