Introduction to Racket

Advanced Functional Programming

Kostis Sagonas

November 2017

(original set of slides by Jean-Noël Monette)

Racket

- 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 from http://www.racket-lang.org/download
- run the self-extracting archive
- type **`racket`** (REPL) or **`drracket`** (IDE)

Getting Started

Installation

- download from http://www.racket-lang.org/download
- run the self-extracting archive
- type **`racket`** (REPL) or **`drracket`** (IDE)

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

- tutorials
- comprehensive guide
- reference manual
- ... and a lot more ...

A First Example

```
#lang racket

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

(fact 42)
```

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 parentheses
- Square brackets `[ ]` and braces `{}` can be used instead of parentheses (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

Strict evaluation:

- arguments to a procedure are evaluated before the procedure

Dynamically typed:

- `(fact "some string")` is a runtime error, not a compilation error

Racket Basics

Strict evaluation:

- arguments to a procedure are evaluated before the procedure

Dynamically typed:

- `(fact "some string")` is a runtime error, not a compilation error

But

Racket Basics

Strict evaluation:

- arguments to a procedure are evaluated before the procedure

Dynamically typed:

- `(fact "some string")` is a runtime error, not a compilation error

But

Macros can emulate laziness

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

Appear between parentheses

- the first expression must evaluate to the procedure
- the remaining ones are the arguments

```
(+ 1 2 3 4)
(string? "Hello")
(equal? 42 "bar")
```

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 with a lower case letter

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

Numbers

- Arbitrary large integers and (exact) rationals: `(/ 1 3)`
- Floating point numbers: `(+ 3.14 -inf.0 +nan.0)`
- Complex numbers: `42+1/2i`
- Test procedures: `number? real? rational? integer?`
  `inexact? exact?`

Booleans

Two boolean 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 42 #f)` returns `42`

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 numbers numerically:

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

Comparing Objects

There are (at least) four comparison procedures in Racket

- `=` compares numbers numerically:

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

- `eq?` compares objects by reference:

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

  This is fast but not reliable in all cases

Comparing Objects (2)

- `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)

- **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
```

- **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
```

Comparing Objects (2)

- **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
```

- **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
```

Suggestion: prefer the use of **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"])
```

If no condition evaluates to true and there is no `else` clause, the result is `(void)`

Printing

There are (at least) three ways to output data to the console:

- `display` removes all quotation marks and string delimiters
- `print` does not remove any quotation marks or string delimiters
- `write` removes the outermost quotation mark if any

In addition, `(newline)` prints a newline.

```
(displayln '(a "azer" 3))
(print '(a "azer" 3))
(newline)
(write '(a "azer" 3))
```

Anonymous Procedures

`(lambda (x) (+ x 1))` defines an anonymous procedure

```
(define inc (lambda (x) (+ x 1)))
(inc 41)
((lambda (x) (+ x 1)) 41)
```

Procedure Body

- A procedure body is composed of any number of (local) definitions followed by any number of expressions
- The return value of the procedure is the value of the last expression
- Internal defines can be mutually recursive:

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

Here `sum` is defined at the top-level, while `suma` is a local definition.

Local Declarations

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

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

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))
```

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 procedures).

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

Local Declaration of Procedures

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

This creates a procedure called `loop` and executes it.

This particular example is probably not very interesting...

Local Declaration of Procedures

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

This creates a procedure called `loop` and executes it.

This particular example is probably not very interesting...

Below, `sum-help` is a procedure 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.

Dots and Infix Notation

A fake list is displayed like that:

`'(1 2 . 3)`

One can also use it when entering a list:

`'(1 2 . (3 4))` is equivalent to the list `'(1 2 3 4)`

Dots and Infix Notation

A fake list is displayed like that:

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

One can also use it when entering a list:

`'(1 2 . (3 4))` is equivalent to the list `'(1 2 3 4)`

One can also use two dots around an element of the s-expr to make it the first one.

```
(code (4 . + . 5)) " is transformed into " (code (+ 4 5))
```

This can be useful if you are not comfortable with the prefix notation.

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.

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)
```

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))
(displayln sum)
(displayln (eval sum))
(displayln (eval (eval sum)))
```

Apply

`apply` applies a procedure 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)))
```

Procedure Arguments

Procedures can have a variable number of arguments:

```
(define (proc1 . all) (apply + (length all) all))
(proc1 12 13 14)
(proc1)
(proc1 41)
(define (proc2 x . rest) (* x (length rest)))
(proc2 7 1 2 3 4 5 6)
(proc2 42 0)
(proc2)
```

Procedure Arguments

Procedures can have a variable number of arguments:

```
(define (proc1 . all) (apply + (length all) all))
(proc1 12 13 14)
(proc1)
(proc1 41)
(define (proc2 x . rest) (* x (length rest)))
(proc2 7 1 2 3 4 5 6)
(proc2 42 0)
(proc2)
```

There can also be optional and keywords arguments:

```
(define (proc3 x [y 2]) (+ x y)) (proc3 40)
(define (proc4 x #:y y) (- x y)) (proc4 #:y 2 44)
(define (proc5 x #:y [y 7]) (* x y)) (proc5 6)
(define (proc6 x #:y y . rest) ...)
```

Curried and Higher-Order Procedures

Short way to define curried procedures:

```
(define ((add x) y) (+ x y))
(define add38 (add 38))
(add38 4)
((add 11) 31)
```

Curried and Higher-Order Procedures

Short way to define curried procedures:

```
(define ((add x) y) (+ x y))
(define add38 (add 38))
(add38 4)
((add 11) 31)
```

A simple composition of procedures:

```
(define ((comp f g) . x)
  (f (apply g x)))
(define add2 (comp add1 add1))
(add2 40)
```

Multiple Values

A procedure 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-many])
```

If no branch 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 _ '()) "A list that contains one element"]
  [(cons a a) "A pair of identical elements"]
  [(or (list y ...) (hash-table y ...))
   "A list or a hash table"]
  [(? string?) "A string"]
  [else "Something else"])
```

If no branch matches, an `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.

Guarded Operations

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

Mainly useful to enclose side-effect only code.

Quiz: What is the return value of the following code?

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

Guarded Operations

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

Mainly useful to enclose side-effect only code.

Quiz: What is the return value of the following code?

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

Also produced by the procedure `(void)`.

Parameters

Parameters are variables that can be dynamically bound:

```
(define color (make-parameter "green"))
(define (best-color) (display (color)))
(best-color)
(parameterize ([color "red"])
  (best-color))
(best-color)
```

This is preferable to **`set!`** for several reasons (tail calls, threads, exceptions).

There exist parameters for instance to define the output stream, the level of details when reporting an error, etc.

Vectors

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

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

- `(vector-ref a-vect num)` accesses the `num`-th element of `a-vect` (indices start from zero)
- Vector elements can be modified with `vector-set!`

```
(vector-set! a-vect 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")
```

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" "Apple")
                        ("B" "Banana"))))
(hash-set! ht "A" "Ananas")
(hash-ref ht "A")
```

New Datatypes

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

```
(point 1 2)
(point? (point 1 2))
(point-x (point 1 2))
```

New Datatypes

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

```
(point 1 2)
(point? (point 1 2))
(point-x (point 1 2))
```

We can also create data structures whose internal structure can be accessed (e.g. recursively by `equals?`), and its fields can be modified:

```
(struct point (x y) #:transparent #:mutable)
```

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 its second is what to do with it.

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

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

`(raise 42)` 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 142)
(kill-thread t)
```

Threads are lightweight and run inside the same physical process.

Threads and Channels

Threads can collaborate (among others) through 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 41)
(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))
```

Parallel and Nested Comprehensions

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

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

Parallel and Nested Comprehensions

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

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

`for*` and variations act as nested `for`'s:

```
(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)))
```

There is 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 — although a bit of an acquired taste
- Code = Data

Wrap-up

- Everything you can expect from a modern functional language
- Minimal syntax — although a bit of an acquired taste
- Code = Data

Next Lectures

- Macros
- Modules and Contracts
- Making your own Language

Voluntary Warm-up Exercises

- Redefine `map` and `length` as recursive procedures.
- 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`.