1 Racket

1.1 Comments

> single line comment: `;`
> multi-line comment: `#| ... |#`
> multi-line comments can be nested

```
; single line comment
#|
  multi-line-comment
        can span
        multiple lines
  end of comment
|#
```

1.2 Datum evaluation

> `(quote <datum>)` or `'<datum>` leaves the datum as-is
> `(unquote <datum>)` or `,<datum>` is the opposite of `quote`
> `(quasiquote <datum>)` or `,@<datum>` allows to apply the unquote where needed

```
'(1 2 3); => (1 2 3)
(1 ,(+ 1 1) 3) ; => '(1 2 3)
```

1.3 Predicates

> all predicates end with `?`
> checks if a number is even: `even?`
> checks if a number is odd: `odd?`
> check if a datum is true: `true?`
> check if a number is positive: `positive?`
> check if a number is negative: `negative?`
> check if a number is zero: `zero?`

```
(even? 2) ; => #t
(odd? 2) ; => #f
(true? #t) ; => #t
```

1.3.1 Equivalence

> check if two numbers are equal: `=`
> checks if two objects or numbers are the same: `eq?`
> checks if two objects are the same: `eqv?`
> checks if two objects are the same: `equal?`

```
(= 1 1) ; => #t
(eq? 1 1) ; => #t
(eqv? 1 1) ; => #t
(equal? 1 1) ; => #t
```

1.4 Data types

> integer: `9125`
> binary: `#b10001110100101`
> octal: `#o21645`
> hexadecimal: `#x23a5`
> real: `91.25`
> rational: `91/25`
> complex: `91+25i`
> boolean: `#t, #f`
> character: `#\A, #\λ, #\u30BB`
> null element: `'(), null`
> lists: `'(1 2 3)`

```
(define x 5) ; => x = 5
(define y "Hello, world!") ; => y = "Hello, world!"
(define z #t) ; => z = #t
(define w #\A) ; => w = #\A
null ; => '()
```

1.4.1 Basic operations

All the operators are in the form `(<operator> <operand> ...)` *(prefix notation)*.

Operations on numbers

> arithmetic operations: `+`, `-`, `*`, `/`
> exponentiation: `expt`

> exponentiation by e: `exp`
> logarithm: `log`
> quotient: `quotient`
> remainder: `remainder`
> largest and smallest of two numbers: `max`, `min`
> add 1: `add1`
> subtract 1: `sub1`
> greatest common divisor: `gcd`
> least common multiple: `lcm`

```
(+ 1 2 3) ; => 6
(- 1 2 3) ; => -4
(expt 2 3) ; => 8
(exp 1) ; => 2.718281828459045
(log 10) ; => 2.302585092994046
(quotient 5 2) ; => 2
(remainder 5 2) ; => 1
(max 1 2) ; => 2
(min 1 2) ; => 1
(add1 5) ; => 6
(sub1 5) ; => 4
(gcd 12 18) ; => 6
(lcm 12 18) ; => 36
```

Operations on strings

> string length: `string-length`
> string append: `string-append`
> string to list: `string->list`
> list to string: `list->string`
> get n-th character: `string-ref`

```
(string-length "Hello, world!") ; => 13
(string-append "Hello, " "world!") ; => "Hello, world!"
(string->list "Hello") ; => '(#\H #\e #\l #\l #\o)
(list->string '(#\H #\e #\l #\l #\o)) ; => "Hello"
(string-ref "Hello" 0) ; => #\H
```

Operations on bools

> logic operations: `and`, `or`, `not`, `xor`
> implication: `implies`

```
(and #t #f) ; => #f
(or #t #f) ; => #t
(not #t) ; => #f
(xor #t #f) ; => #t
(implies #t #f) ; => #f
```

1.5 Functions

> anonymous functions: `(lambda (<arg1> <arg2> ...)<body>)`
> named functions: `(define (<name> <arg1> <arg2> ...)<body>)`
> old way: `(define <name> (lambda (<arg1> <arg2> ...)<body>))`

```
(lambda (x) (+ x 1)) ;
(define (add1 x) (+ x 1)) ;
```

1.5.1 Higher order functions

> apply a function to each element of a list: `map`
> apply a filter: `filter`
> apply a function to each element of a list and flatten the result: `apply`
> fold a list: `foldl`, `foldr`

```
(map add1 '(1 2 3)) ; => '(2 3 4)
(filter even? '(1 2 3 4)) ; => '(2 4)
(apply append '((1 2) (3 4))) ; => '(1 2 3 4)
(foldl + 0 '(1 2 3)) ; => 6
(foldr + 0 '(1 2 3)) ; => 6
```

1.6 Variables

> parallel binding: `let`
> serial binding: `let*`
> recursive binding: `letrec`
> recursive serial binding: `letrec*`

```
(let ((x 5) (y 2)) (list x y)) ; => '(5 2)
(let* ((x 1) (y (add1 x))) (list x y)) ; // '(1 2)
```

1.7 Collections

1.7.1 Structs

> definition: `(struct <struct-name> (<field> ...))`
> constructor: `(define <name> <struct-name> <field-value> ...)`
> getter: `<struct-name>-<field-name>`
> setter: `set-<struct-name>-<field-name>!`
> predicate: `<struct-name>?`
> structs and fields are immutable by default
> use `#:mutable` keyword on struct or field to make it mutable

```
(struct point (x y)) ; => point
(define p (point 1 2)) ; => p = (point 1 2)
(point-x p) ; => 1
(point? p) ; => #t

(struct mut-point (x y #:mutable)) ; => point
(define mp (mut-point 1 2)) ; => mp = (mut-point 1 2)
(set-mut-point-x! mp 5) ; => mp = (mut-point 5 2)
```

1.7.2 Pairs

> definition: `(cons <first> <second>)`
> getter of first element: `car`
> getter of second element: `cdr`
> car and cdr can be composed *(cdadadr, caaar)*
> pairs are immutable

```
(cons 1 2) ; => '(1 . 2)
(car '(1 . 2)) ; => 1
(cdr '(1 . 2)) ; => 2
(caar '((1 . 2) . 3)) ; => 1
(cadr '((1 . 2) . 3)) ; => 2
(cdar '((1 . 2) . 3)) ; => 2
(cddr '((1 . 2) . 3)) ; => 3
```

1.7.3 Lists

> lists are composed of pairs
> manually defined via `quote`: `'(1 2 3)`
> empty list: `'()`
> lists are made by pairs
  - the `car` contains the first value
  - the `cdr` contains the the rest of the list
  - the last pair has `cdr` equal to `'()`

```
'(1 2 3) ; => '(1 2 3)
'(1 . (2 . (3 . ()))) ; => '(1 2 3)
```

Operations on lists

> list length: `length`
> add an element at the beginning: `cons`
> add an element at the end: `append`
> take the first element: `first`
> take the last element: `last`
> take the n-th element: `list-ref <list> <n>`
> take the n-th element after pos: `list-tail <list> <pos>`
> take the first n elements: `take <list> <n>`
> take the last n elements: `drop <list> <n>`
> count the occurrences of an element: `count <predicate> <list>`
> apply a filter: `filter <predicate>`
> apply a function to each element: `map <function>`
> get the reverse of a list: `reverse`
> get the elements after the first: `rest`

```
(length '(1 2 3)) ; => 3
(cons 1 '(2 3)) ; => '(1 2 3)
(append '(1 2) '(3 4)) ; => '(1 2 3 4)
(first '(1 2 3)) ; => 1
(last '(1 2 3)) ; => 3
(list-ref '(1 2 3) 1) ; => 2
(list-tail '(1 2 3) 1) ; => '(2 3)
(take '(1 2 3) 2) ; => '(1 2)
(drop '(1 2 3) 1) ; => '(2 3)
(count even? '(1 2 3 4)) ; => 2
(filter even? '(1 2 3 4)) ; => '(2 4)
(map add1 '(1 2 3)) ; => '(2 3 4)
(reverse '(1 2 3)) ; => '(3 2 1)
(rest '(1 2 3)) ; => '(2 3)
```

Lists folding

> lists can be folded from the left with `foldl`
> lists can be folded from the right with `foldr`
> the accumulator is the first argument of the function
> the list is the second argument of the function
> the function is applied to the accumulator and the first element of the list

```
(foldl + 0 '(1 2 3 4)) ; => 10
(foldr * 1 '(1 2 3 4)) ; => 24
```

1.7.4 Vectors

> definition: `#(<element> ...)`
> getter: `vector-ref`
> vector are immutable, fixed size and zero-indexed

```
#(1 2 3) ; => '#(1 2 3)
(vector-ref '#(1 2 3) 0) ; => 1
```

1.7.5 Sets

> definition: `(set <element> ...)`
> convert a list to a set: `list->set`
> add an element: `set-add`
> remove an element: `set-remove`
> test if an element is in the set: `set-member?` *(returns a boolean)*
> sets don't allow duplicates, are unordered and mutable
> methods return a new set instead of changing the original one

```
(set 1 2 3) ; => '#(1 2 3)
(list->set '(1 2 3)) ; => '#(1 2 3)
(set-add (set 1 2 3) 4) ; => '#(1 2 3 4)
(set-remove (set 1 2 3) 2) ; => '#(1 3)
(set-member? (set 1 2 3) 2) ; => #t
```

1.7.6 Hash

> definition: `(hash <key> <value> ...)`
> add a key-value pair: `hash-set`
> remove a key-value pair: `hash-remove`
> get a value from a key: `hash-ref`
> test if a key is in the hash: `hash-has-key?` *(returns a boolean)*

```
(hash 1 2 3 4) ; => '#hash((1 . 2) (3 . 4))
(hash-set (hash 1 2 3 4) 5 6) ; => '#hash((1 . 2) (3 . 4)
   (5 . 6))
(hash-remove (hash 1 2 3 4) 3) ; => '#hash((1 . 2) (4 . 4))
(hash-ref (hash 1 2 3 4) 1) ; => 2
(hash-has-key? (hash 1 2 3 4) 1) ; => #t
```

1.8 Control flow

1.8.1 Conditionals

if

> if: `(if <predicate> <then> <else>)`
> when: `(when <predicate> <then>)`
> unless: `(unless <predicate> <else>)`

```
   (if #t 1 2) ; => 1
   (when #t 1) ; => 1
   (when #f 1) ; => #<void>
   (unless #t 1) ; => #<void>
   (unless #f 1) ; => 1
```

cond - case

> cond: `(cond [<predicate> <then>] ... [<else> <else-then>])`
> case: `(case <value> [<case-clause> <then>] ... [<else> <else-then>])`
> the `else` clause is optional
> in cond, the value is evaluated against each predicate
> in case, the value is evaluated against each clause whose quote is `eqv?`

```
(case (+ 7 5)
  [(1 2 3) 'small]
  [(10 11 12) 'big]
  [else 'neither]) ; => 'big

(let ((x 0))
(cond ((positive? x) 'positive)
      ((negative? x) 'negative)
      (else 'zero))) ; => 'zero
```

pattern matching

> match: (match <value> [<pattern> <then>] ... [_ <else-then>])

```
    (define (fizzbuzz? n)
    (match (list (remainder n 3) (remainder n 5))
      [(list 0 0) 'fizzbuzz]
      [(list 0 _) 'fizz]
      [(list _ 0) 'buzz]
      [_          #f]))

  (fizzbuzz? 15) ; => 'fizzbuzz
  (fizzbuzz? 37) ; => #f
```

1.8.2 Loops

when

> when: (when <predicate> <then>)
> also available as named let

```
  ;; named let
  (let label ((x 0)) ; initialize x as 0
        (when (< x 10) ; iterate while x < 10
                (display x) ; print x
                (newline)
                (label (+ x 1)))) ; increment x, go back
                   to label

  (define (loop i)
    (when (< i 10)
      (printf "i=~a\n" i)
      (loop (add1 i))))
  (loop 5) ; => i=5, i=6, i=7, i=8, i=9,
```

for

> for in a range: (for ([<var> <start> <end>])<body>)
> for over lists: (for ([<var> <list>])<body>)
> for is available for other collections

```
(for ([i 10])
  (printf "i=~a\n" i)) ; => i=0, i=1, ...
(for ([i (in-range 5 10)])
  (printf "i=~a\n" i)) ; => i=5, i=6, ...

(for ([i (in-list '(l i s t))])
  (displayln i))

(for ([i (in-vector #(v e c t o r))])
  (displayln i))

(for ([i (in-string "string")])
  (displayln i))

(for ([i (in-set (set 'x 'y 'z))])
  (displayln i))

(for ([(k v) (in-hash (hash 'a 1 'b 2 'c 3))])
  (printf "key:~a value:~a\n" k v))
```

1.9 Macros and syntax rules

> macros are defined via define-syntax(<name> <expansion>)
> syntax rules are defined via syntax-rules(<pattern> <expansion>)
> macros are expanded at compile time
> syntax rule are pairs (<pattern> <expansion>)
> the ... operator indicates repetitions of patterns
> the _ operator is used to match any syntax object

```
  (define-syntax while
    (syntax-rules ()           ; no other keyword is needed
      ((_ condition body ...) ; pattern P
       (let loop ()           ; expansion of P
         (when condition
           ((begin body ...
                  (loop))))))))
```

1.10 Continuations

1.11 Exceptions

1.12 Object Oriented

...

2 Erlang

3 Haskell