

# 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 Data types

- > typing is dynamic
- > types:
  - > boolean: #t, #f
  - > integer: 9125
  - > binary: #b10001110100101
  - > octal: #o21645
  - > hexadecimal: #x23a5
  - > real: 91.25
  - > rational: 91/25
  - > complex: 91+25i
  - > character: #\A, #\\$ \lambda\$, #\u30BB
  - > null element: '(), null
  - > string: "Hello, world!"

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

## 1.3 Variables

- > variables are immutable

- > parallel binding: let
- > serial binding: let\*
- > recursive binding: letrec

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

### 1.3.1 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)
```

### 1.3.2 Equivalence

- > numbers equivalence: =
- > objects or numbers equivalence: eq?
- > objects equivalence: eqv?
- > objects equivalence: equal?

```
(= 1 1) ; => #t
(eq? 1 0) ; => #f
(eqv? 'yes 'yes) ; => #t
(equal? 'yes 'no) ; => #f
```

### 1.3.3 Basic operations

- > all operations are in prefix notation <operator> <operand> ...

## 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 2) ; => e ** 2 = 7.38905609893065
(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
```

### 1.3.4 Types conversion

- > inexact and exact: `inexact->exact`, `exact->inexact`
- > integer and float: `integer->float`, `float->integer`
- > integer and rational: `integer->rational`, `rational->integer`
- > list and vector: `list->vector`, `vector->list`
- > vector and string: `vector->string`, `string->vector`

## 1.4 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 datum is false: `false?`
- > check if a number is positive: `positive?`
- > check if a number is negative: `negative?`
- > check if a number is zero: `zero?`
- > check if an object is immutable: `immutable?`

```
(even? 2) ; => #t
(odd? 2) ; => #f
(true? #t) ; => #t
(false? #t) ; => #f
(positive? 1) ; => #t
(negative? 1) ; => #f
(zero? 1) ; => #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` <function> <list>
- > apply a filter: `filter` <predicate> <list>
- > apply a function to each element of a list and flatten the result: `apply` <function> <list>
- > fold a list: `foldl` <function> <accumulator> <list>
- > fold a list: `foldr` <function> <accumulator> <list>
- > `foldl` has space complexity  $O(1)$
- > `foldr` has space complexity  $O(n)$

```
(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 Mutation

- > all mutators end with !
- > `set!` is used to mutate variables
- > `vector-set!` is used to mutate vectors

```
(define x 5) ; => x = 5  
(set! x 6) ; => x = 6  
(define v (vector 2 2 3 4)) ; => v = '#(2 2 3 4)  
(vector-set! v 0 1) ; => v = '#(1 2 3 4)
```

## 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: `caddr`, `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  
(caddr '((1 . 2) . 3)) ; => 3
```

### 1.7.3 Lists

- > lists are composed of pairs
- > manually defined via `quote`: `'(1 2 3)`
- > empty list: `'()`
- > list of length `n`: `build-list <n> <procedure>`
- > list of length `n` with initial value `<init>`: `make-list <n> <init>`
- > 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`
- > get the elements after the first: `rest <list>`
- > get the first element: `first`
- > get the last element: `last`
- > get the `n`-th element: `list-ref <list> <n>`
- > get the elements after the `n`-th: `list-tail <list> <pos>`
- > get the first `n` elements: `take <list> <n>`
- > get the last `n` elements: `drop <list> <n>`
- > count the occurrences of an element: `count <predicate> <list>`
- > apply a filter: `filter <predicate> <list>`
- > apply a function to each element: `map <function> <list>`
- > get the reverse of a list: `reverse <list>`

```
(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`

```
(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?`
- > 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?`

```
(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 '(1 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

- > definition: `define-syntax`((< literals>)[(< syntax-rule> ...), ...])
- > syntax rules are defined via `syntax-rules`(< pattern> < expansion>)
- > macros are expanded at compile time
- > the ... operator indicates repetitions of patterns
- > the `_` operator is used to match any syntax object

```
(define-syntax while
  (syntax-rules () ; no reserved keywords
    ((_ condition body ...) ; pattern P
     (let loop () ; expansion of P
       (when condition
         ((begin body ...)
          (loop)))))))
```

## 1.10 Continuations

- > two ways to call a continuation:
  - > `call-with-current-continuation` <procedure>
  - > `call/cc` <procedure>
- > saving the continuation: `save!` <continuation>

## 1.11 Exceptions

- > exceptions are implemented via continuations
- > raise an exception: `raise`
- > catch an exception: `with-handlers`

```
(with-handlers ([exn:fail? (lambda (e) (printf "error:
~a\n" e))])
  (raise (exn:fail "error message"))) ; => error: error
message
```

## 2 Haskell

### 2.1 Comments

- > single line comment: `--`
- > multi-line comment: `{- ... -}`

```
-- single line comment
{- multi-line comment
   can span
   multiple lines
   end of comment -}
```

### 2.2 Data types

- > data type is inferred automatically by the compiler
- > data type can be specified explicitly via type annotations `::`
- > types:
  - > boolean: `True`, `False`
  - > integer: `1`, `2`, `3`
  - > float, double: `1.0`, `2.0`, `3.0`
  - > complex: `1 :+ 2`, `2 :+ 3`, `3 :+ 4`
  - > character: `'a'`, `'b'`, `'c'`
  - > string: `"a"`, `"b"`, `"c"` or `"abc"`
  - > lists: `[1, 2, 3]`
  - > tuples: `(1, 2, 3)`

#### 2.2.1 User defined types

- > sum types: `data <type> = <constructor1> | <constructor2> | ...`
- > product type: `data <type> = <constructor> <field1> <field2> ...`

```
data Bool = True | False -- sum type
data Point = Point Float Float -- product type
```

#### 2.2.2 Recursive types

- > syntax: `data <type> = <constructor> <field1> <field2> ... <type>`

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

#### 2.2.3 Type Synonyms

- > syntax: `type <name> = <type>`

```
type Point = [(Float, Float)]
```

### 2.3 Variables

- > variables are immutable
- > recursive binding: `let`
- > declaration with function body: `where`

```
let x = 5 in x + 1 ; => 6
let x = 5
    y = 2
in x + y ; => 7
f x = x + 1
where x = 5 ; => 6
```

#### 2.3.1 Equivalence

- > equivalence between objects, numbers, strings and characters: `==`

#### 2.3.2 Basic operations

- > prefix operators can be converted into infix notation via backticks `<operator>`
- > infix operators can be converted into prefix notation via parentheses `(<operator>)`
- > symbol `$` is used to avoid parentheses by applying the function to the right

## Operations on numbers

- > arithmetic operations: `+`, `-`, `*`, `/`
- > exponentiation: `**`
- > exponentiation by e: `exp`
- > logarithm: `log`
- > quotient: `quot`
- > remainder: `rem`
- > largest and smallest of two numbers: `max`, `min`
- > add 1: `succ`
- > subtract 1: `pred`
- > greatest common divisor: `gcd`
- > least common multiple: `lcm`

```
3 + 2 ; => 5
3 - 2 ; => 1
3 * 2 ; => 6
3 / 2 ; => 1.5
3 ** 2 ; => 9.0
exp 2 ; => 7.38905609893065
log 10 ; => 2.302585092994046
quot 5 2 ; => 2
rem 5 2 ; => 1
max 1 2 ; => 2
min 1 2 ; => 1
succ 5 ; => 6
pred 5 ; => 4
gcd 12 18 ; => 6
lcm 12 18 ; => 36
```

## Operations on strings

- > string length: `length`
- > string append: `++`
- > string to list: `words`
- > list to string: `unwords`

```
length "Hello, world!" ; => 13
"Hello, " ++ "world!" ; => "Hello, world!"
words "Hello world!" ; => ["Hello", "world!"]
unwords ["Hello", "world!"] ; => "Hello world!"
```

## Operations on bools

- > logic operations: `&&`, `||`, `not`, `xor`
- > implication: `implies`

```
True && False ; => False
True || False ; => True
not True ; => False
xor True False ; => True
implies True False ; => False
```

### 2.3.3 Types conversion

- > list to string: `show`
- > string to list: `read`

## 2.4 Functions

- > lambda functions: `\<name> <arg1> <arg2> ... -> <body>`
- > functions are defined as sequences of equations
  - > arguments are matched with the right parts of equations, top to bottom
  - > if the match succeeds, the function body is called

```
\x y -> x + y ; => \x y -> x + y

length :: [a] -> Integer
length [] = 0
length (x:xs) = 1 + length xs
1 == 1 ; => True
"abc" == "abc" ; => True
```

## 2.5 Collections

### 2.5.1 Fields

- > fields can be accessed either by label or by position

### 2.5.2 Lists

- > lists are composed of pairs
- > manual definition: `[1, 2, 3]`
- > empty list: `[]`



## Operations on lists

- > list length: `length <list>`
- > get the reverse of a list: `reverse`
- > concatenate two lists: `<list1> ++ <list2>`
- > add an element: `<element> : <list>`
- > get the first element: `head <list>`
- > get the last element: `last <list>`
- > get the n-th element: `<list>! <position>!`
- > get the first n elements: `take <list> <n>`
- > delete the first n elements: `drop <n> <list>`
- > get all the elements after the first: `tail`
- > split a list in two: `splitAt <position> <list>`
- > apply a filter: `filter <predicate?> <list>`
- > apply a function to each element: `map <function> <list>`
- > sum a list: `sum <list>`
- > product of a list: `product <list>`
- > check if a list is empty: `null <list>`
- > check if an element is in a list: `elem <element> <list>`
- > check if all elements of a list satisfy a predicate: `all <predicate> <list>`
- > check if at least one element of a list satisfies a predicate: `any <predicate> <list>`
- > zip two lists: `zip <list1> <list2>`

```
length [1, 2, 3] ; => 3
reverse [1, 2, 3] ; => [3, 2, 1]
[1, 2, 3] ++ [4, 5, 6] ; => [1, 2, 3, 4, 5, 6]
1 : [2, 3] ; => [1, 2, 3]
head [1, 2, 3] ; => 1
last [1, 2, 3] ; => 3
[1, 2, 3] !! 1 ; => 2
take 2 [1, 2, 3] ; => [1, 2]
drop 2 [1, 2, 3] ; => [3]
tail [1, 2, 3] ; => [2, 3]
splitAt 1 [1, 2, 3] ; => ([1], [2, 3])
filter even [1, 2, 3, 4] ; => [2, 4]
map (+1) [1, 2, 3] ; => [2, 3, 4]
sum [1, 2, 3] ; => 6
product [1, 2, 3] ; => 6
null [] ; => True
elem 1 [1, 2, 3] ; => True
all even [2, 4, 6] ; => True
any even [1, 2, 3] ; => True
zip [1, 2, 3] [4, 5, 6] ; => [(1, 4), (2, 5), (3, 6)]
```

## Range notation

- > finite list: `[<start>..<end>]`
- > finite list with step: `[<start>,<step>..<end>]`
- > infinite list: `[<start>..]`
- > infinite list with step: `[<start>,<step>..]`
- > infinite list with one element repeated: `[<element>,<element>..]`

To explicitly evaluate a finite list use the `init` function.

```
-- all the following instructions are lazily evaluated
[1..10] ; => [1,2,3,4,5,6,7,8,9,10]
[1,3..10] ; => [1,3,5,7,9]
[1..] ; => [1,2,3,4,5,6,7,8,9,10,...]
[1,3..] ; => [1,3,5,7,9,...]
[1,1..] ; => [1,1,1,1,1,1,1,1,1,1,...]
```

## List Comprehension

- > list comprehension returns a list of elements created by evaluation of the generators
- > syntax: `[<expression> | <generator>, <generator>, ...]`

```
[x | x <- [1..10], even x] ; => [2,4,6,8,10]
[x * y | x <- [2,5,10], y <- [8,10,11]] ; =>
[16,20,22,40,50,55,80,100,110]
```

## 2.6 Control flow

### 2.6.1 Pattern matching

- > the matching process is done top to bottom, left to right
- > patterns may have boolean guards
- > character `_` matches everything (*don't care*)

```
sign x | x > 0 = 1
      | x < 0 = -1
      | otherwise = 0

take 0 _ = []
take _ [] = []
take n (x:xs) = x : take (n - 1) xs
```

### 2.6.2 Case

- > syntax: `case <value> of <pattern> -> <then> ...`
- > the `_` pattern matches everything

```
case x of
  0 -> "zero"
  1 -> "one"
  _ -> "other"
```

### 2.6.3 Conditionals

- > if: `if <predicate> then <then> else <else>`
- > when: `when <predicate> <then>`
- > unless: `unless <predicate> <else>`

```
if True then 1 else 2 ; => 1
when True 1 ; => 1
unless False 1 ; => 1

-- equivalent to
if True then 1 else 2 ; => 1
if True then 1 ; => 1
if False then 1 ; => ()

-- equivalent to
if True then 1 else 2 ; => 1
if False then 2 else 1 ; => 1
```

### 2.6.4 Loops

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

```
for i <- [1..10] do
  print i ; => 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
for i <- [1..10], i 'mod' 2 == 0 do
  print i ; => 2, 4, 6, 8, 10
```

## 2.7 Monads

- > Monads are used to encapsulate side effects
- > the `do` notation is used to chain monadic actions

```
comb :: Maybe a -> (a -> Maybe b) -> Maybe b
comb Nothing _ = Nothing
comb (Just x) f = f x
```

### 2.7.1 Foldable

- > used for folding (either `foldl` or `foldr`)
- > given a container and a binary operation `f`, applies `f` to each element of the container
- > syntax: `foldr <function> <accumulator> <container>`

```
foldr (+) 0 [1, 2, 3] ; => 6
foldr (*) 1 [1, 2, 3] ; => 6
```

```
data Tree a = Empty | Leaf a | Node (Tree a) a (Tree a)
```

```
instance Foldable Tree where
  foldr f z Empty = z
  foldr f z (Leaf x) = f x z
  foldr f z (Node l k r) = foldr f (f k (foldr f z r)) l
```

### 2.7.2 Functor

- > used for mapping (via `fmap`)
- > signature of `fmap`: `fmap :: (a -> b) -> f a -> f b`

```
instance Functor Tree where
  fmap f Empty = Empty
  fmap f (Leaf x) = Leaf (f x)
  fmap f (Node l r) = Node (fmap f l) (fmap f r)
```

### 2.7.3 Applicative

- > used for applying a function in a context (via `<*>`)
- > is applied to type constructors that are parametric with one parameter

```
instance Applicative Maybe where
  pure = Just
  (Just f) <*> something = fmap f something
  Nothing <*> _ = Nothing
```

- > `pure` takes a value and returns an applicative (`f`) with that value
- > `<*>` takes an applicative (`f`) with a function and another applicative (`g`) and returns an applicative (`h`) with the result of applying the function to the value of `g`

### 2.7.4 Monad

- > used for sequencing (via `>>=`)
- > signature of:

```
> >>=: (>>=) :: m a -> (a -> m b) -> m b
> >>: (>>) :: m a -> m b -> m b
> return: return :: a -> m a
```

- > meaning of:

- > `>>=: m a` is a monadic action that returns a value of type `a`
- > `f` is a function that takes a value of type `a` and returns a monadic action that returns a value of type `b`
- > `>>=` returns a monadic action that returns a value of type `b`

- > `do` notation is used to chain monadic actions, it is translated into `>>=` and `>>`
- > `do` notation is available for all monads

```
instance Monad Maybe where
  return = Just
  Nothing >>= _ = Nothing
  (Just x) >>= f = f x
```

## 2.8 Type classes

- > type classes are defined via `class`
- > type classes are instantiated via `instance`

### 2.8.1 Polymorphism

- > type class constraints are resolved at compile time
- > parametric polymorphism: `<name> :: <type> -> <type>`
- > ad-hoc polymorphism: `<name> :: <type class> => <type> -> <type>`

```
-- parametric polymorphism
id :: a -> a
id x = x

-- ad-hoc polymorphism
(+) :: Num a => a -> a -> a
x + y = x + y
```

## 3 Erlang

### 3.1 Syntax

- > period . terminates expressions"
- > semicolon ; separates expressions branches and clauses
- > comma , separates alternate clauses

### 3.2 Comments

- > single line comment: %
- > multi line comments are not supported

```
% single line comment
%% sometimes the double percent is used
```

### 3.3 Data types

- > term: any Erlang data
- > integer: 1, 2, 3
- > float: 1.0, 2.0, 3.0
- > binary, hexadecimal and octal numbers: 2#101, 16#FF, 8#377
- > atoms: `atom` (see 3.3.1)
- > bit strings: <<"hello">>
- > strings: "hello"

#### 3.3.1 Atoms

- > any sequence of letters, digits, underscore, at sign, dollar sign and full stop
- > atoms are used to represent constants
- > syntax: <atom> or '<atom>'
- > if unquoted, atoms can contain only lowercase letters, digits and underscore

```
atom ; => atom
'atom' ; => atom
'ATOM' ; => 'ATOM'
```

### 3.4 Functions

- > functions are defined as sequences of equations
  - > arguments are matched with the right parts of equations, top to bottom
  - > if the match succeeds, the function body is called
- > functions have boolean guards `when <predicate> -><then>`
  - > guards are evaluated in constant time

```
factorial(0) -> 1;
factorial(N) when N > 0 -> N * factorial(N - 1).
```

#### 3.4.1 Apply

- > syntax: `apply(<module>, <function>, <arguments>)`
- > any expression can be used as a function
- > the function is applied to the arguments
- > `$MODULE` is a macro that expands to the name of the current module

```
% assumption: factorial is defined in the current module
apply($MODULE, factorial, [5]) ; => 120
```

#### 3.4.2 Function Guards

- > `X` is a number: `number(X)`
- > `X` is an integer: `integer(X)`
- > `X` is a float: `float(X)`
- > `X` is an atom: `atom(X)`
- > `X` is a list: `is_list(X)`
- > `X` is a tuple: `is_tuple(X)`
- > `X` is a map: `is_map(X)`
- > `X` is greater than `Y`: `X > Y`
- > `X` is less than `Y`: `X < Y`
- > `X` is exactly equal to `Y`: `X == Y`
- > `X` is equal to `Y` when converted to the `int`: `X == Y`
- > `X` is not equal to `Y`: `X /= Y`
- > `X` is a list of length `N`: `length(X) == N`
- > `X` is a tuple of length `N`: `size(X) == N`

### 3.4.3 Function Calls

- > function and modules names must be atoms
- > function call `<name>(<arg1>, <arg2>, ...)`
- > alternative `<module>:<name>(<arg1>, <arg2>, ...)`
- > use `-import` to avoid specifying the module name

```
my_module:my_function(1, 2, 3) ; => 6
-import(my_module, [my_function/3]).
my_function(1, 2, 3) ; => 6
```

### 3.5 Variables

- > variables are immutable and can be bound only once
- > variables start with an uppercase letter
- > there is no keyword for variable declaration

```
X = 5 ; => X = 5
A_very_long_variable_name = 5 ; =>
  A_very_long_variable_name = 5
```

### 3.6 Collections

#### 3.6.1 Lists

- > lists are composed of pairs
- > lists are immutable
- > manual definition: `[1, 2, 3]`
- > empty list: `[]`

```
[1, 2, 3] ; => [1, 2, 3]
[1, [2, 3]] ; => [1, [2, 3]]
```

### Operations on lists

- > list length: `length(<list>)`
- > add an element at the beginning: `[<element> | <list>]`
- > add an element at the end: `<list> ++ [<element>]`
- > get the first element: `lists:first(<list>)`
- > get the last element: `lists:last(<list>)`
- > get the n-th element: `lists:nth(<position>, <list>)`
- > get the first n elements: `lists:sublist(<list>, 0, <n>)`
- > append two lists: `lists:append(<list1>, <list2>)`
- > concatenate two lists: `<list1> ++ <list2>`
- > delete the first occurrence of an element: `lists:delete(<element>, <list>)`
- > delete the last element of a list: `lists:droplast(<list>)`
- > delete the first element of a list: `lists:drop(<list>)`
- > apply a filter: `lists:filter(<predicate>, <list>)`
- > flatten a list: `lists:flatten(<list>)`
- > call a function on each element: `lists:foreach(<function>, <list>)`
- > map a function on each element: `lists:map(<function>, <list>)`
- > find the maximum element: `lists:max(<list>)`
- > find the minimum element: `lists:min(<list>)`

#### Lists folding

- > fold a list from the left: `lists:foldl(<function>, <accumulator>, <list>)`
- > fold a list from the right: `lists:foldr(<function>, <accumulator>, <list>)`

```
lists:foldl(fun(X, Acc) -> X + Acc end, 0, [1, 2, 3]) ; => 6
lists:foldr(fun(X, Acc) -> X + Acc end, 0, [1, 2, 3]) ; => 6
```

#### 3.6.2 Tuples

- > tuples are immutable
- > tuples can be nested
- > syntax: `{<element1>, <element2>, ...}`

```
{1, 2, 3} ; => {1, 2, 3}
{1, {2, 3}} ; => {1, {2, 3}}
```

### 3.6.3 Records

- > records are tuples with named fields
- > records are defined via `-record(<name>, <field1>, <field2>, ...)`
- > records are accessed via `#<name>.<field>`

```
-record(point, {x, y}).  
#point.x ; => x
```

### 3.6.4 Maps

- > maps are defined via `key => <value>, <key> => <value>, ...#<`
- > keys are accessed via `<map>.<key>`
- > maps are updated:
  - > to add or overwrite a key-value pair: `key => <value><map>#<`
  - > to only update an existing key-value pair: `key := <value><map>#<`

```
Map = #{a => 1, b => 2} % => #{a => 1, b => 2}  
Map#{c => 3} % => #{a => 1, b => 2, c => 3}  
Map#{a := 2} % => #{a => 2, b => 2}  
Map#{d := 4} % => error
```

## 3.7 Control flow

### 3.7.1 Conditionals

#### Pattern matching

- > the matching process is done top to bottom, left to right
- > patterns may have boolean guards
- > character `_` matches everything (*don't care*)

```
sign(X) when X > 0 -> 1;  
sign(X) when X < 0 -> -1;  
sign(_) -> 0.
```

### if

- > syntax: `if <predicate> -><then>; <predicate> -><then>; ... end`
- > the `true` pattern matches everything
- > function guards are necessary

```
if  
  integer(X) -> integer_to_list(X);  
  float(X) -> float_to_list(X);  
  true -> "error" % this is the default case  
end.
```

### case

- > syntax: `case <value> of <pattern> -><then>; <pattern> -><then>; ... end`
- > the `true` pattern matches everything
- > function guards are not required

```
case X of  
  0 -> "zero";  
  1 -> "one";  
  true -> "other"  
end.
```

### 3.7.2 Loops

- > loops must be implemented via recursion

### while

- > syntax: `while(<predicate>)-><then>; ...`

```
while(X) ->  
  if  
    X > 0 -> io:format("~p~n", [X]),  
    while(X - 1);  
    true -> ok.  
  end.
```

for

> syntax: for(<variable>)-><then>; ...

```
for(X) ->
    for(X, 0).

for(X, N) when N < X ->
    io:format("~p~n", [N]),
    for(X, N + 1);
for(_, _) ->
    ok.
```

### 3.8 Concurrent programming

- > processes are created via `spawn(<module>, <function>, <arguments>)`
- > processes are identified via `Pid`
- > messages are sent via `Pid ! <message>`
- > messages are received via `receive <pattern> -><then>; ... end`

### 3.9 Examples

This section is not present for other languages, but I feel like there's no other way to explain important concepts (for the exam) in Erlang.

#### 3.9.1 Echo process

```
-module(echo).
-export([start/0, loop/0]).

start() ->
    Pid2 = spawn(echo, loop, []),
    Pid2 ! {self(), hello},

    receive
        {Pid2, Msg} -> io:format("received ~p~n", [Msg])
    end.

Pid2 ! stop.

loop() ->
    receive
        {From, Msg} ->
            From ! {self(), Msg},
            loop();
        stop ->
```

```
        true
    end.
```

#### 3.9.2 Client-server

- > requests have syntax `{request, <data>}`
- > responses have syntax `{response, <data>}`

```
%% this snippet contains the server code
-module(my_server).
-export([start/0, server/1]).
server(Data) ->
    receive
        {From, {request, X}} ->
            {R, Data1} = handle(X, Data),
            % note: handle function is defined elsewhere
            From ! {my_server, {response, R}},
            server(Data1);
        {From, stop} ->
            From ! {my_server, stopped}
    end.
```

```
%% this snippet contains the client code
-module(my_client).
-export([start/0, client/1]).
client(Server) ->
    Server ! {self(), {request, 1}},
    receive
        {my_server, {response, R}} ->
            io:format("received ~p~n", [R])
    end.

client(Server) ->
    Server ! {self(), stop},
    receive
        {my_server, stopped} ->
            ok
    end.

start() ->
    Server = spawn(my_server, server, []),
    spawn(my_client, client, [Server]).
```