```racket
#lang racket

; lambda definition
(lambda (x y) (+ x y))

; code blocks
(begin
  (displayln 42)
  (displayln 24))

; function definition
(define (minimum L)
  (let ((x (car L)) (xs (cdr L)))
    (if (null? xs)
        x
        (min x (minimum xs)))))
(minimum '(4 2 1 5)) ; => 1
; variable number of arguments
(define (minimum2 x . rest)
  (if (null? rest)
      x
      (min x (minimum rest))))
(minimum2 4 2 1 5) ; => 1

; Note on functions: parameters are passed by
; reference like in Java

; let binding
(let ((x 10) (y 15))
  (+ x y))

; static scoping
(let ((a 1))
  (let ((f (lambda () (displayln a))))
    (let ((a 2))
      (f)))) ; => 1

; global variables
(define x 12)
(set! x 42)
(define double (lambda (x) (* 2 x)))

; lists
(define l1 '(1 2 3))
(define l2 (cons 1 (cons 2 (cons 3 '())))) ;
    => '(1 2 3)
(member 2 '(1 2 3)) ; => '(2 3)
(car l1) ; => 1
(cdr l1) ; => '(2 3)
(apply + '(1 2 3 4)) ; => 10
(null? l1) ; => #f
(null? '()) ; => #t
(list 1 2 3 4) ; => '(1 2 3 4)
(list* 1 2 3 (list* 4 5 6)) ; => '(1 2 3 4 5
    6)
(length l1) ; => 3
(list-ref l1 2) ; => 3
(append l1 l1 l1) ; => '(1 2 3 1 2 3 1 2 3)
(reverse l1) ; => '(3 2 1)
(take '(5 8 4 1) 2) ; => '(5 8)
(range 5) ; => '(0 1 2 3 4)
(range 2 5) ; => '(2 3 4)
(range 2 10 2) ; => '(2 4 6 8)

; mutable lists with mcons, set-mcar!,
    set-mcdr!

; if-else
(if (= (+ 1 2) 3) 42 24) ; => 42
(when (< 10 12)
  (displayln "ciao")
  (displayln "mondo")
  42) ; => ciao mondo 42

; equivalence
(eq? 'ciao 'ciao) ; => #t
(eqv? 42 42) ; => #t
(equal? '(1 2 3) '(1 2 3)) ; => #t

(case (car '(c d)) ; case uses eqv?
  ((a e i o u) 'vowel)
  (else 'consonant)) ; => 'consonant
(cond ((> 3 3) 'greater)
      ((< 3 3) 'less)
      (else 'equal)) ; => 'equal

; loops
(let label ((x 0))
  (when (< x 10)
    (displayln x)
    (label (+ x 1)))) ; => 1 2 ... 9
(for-each (lambda (x) (displayln x))
          '(1 2 3 4))

; vectors
(define vec (vector 1 2 3))
(vector-ref vec 1) ; => 2
(vector-set! vec 1 42) ; => #(1 42 3)
(vector-length vec) ; => 3
; vector-for-each
(define (vector-for-each body vect)
    (let ((len (vector-length vect)))
        (let loop ((i 0))
            (when (< i len)
                (body (vector-ref vect i))
                (loop (+ i 1))))))
(vector-for-each (lambda (x) (display x))
    vec) ; => 123

; structs
(struct being
  (name
   (age #:mutable)))
(define (say-hello x)
  (if (being? x)
      (printf "Hi ~a (~a)~n"
              (being-name x)
              (being-age x))
      (displayln "Not a being")))
(define edo (being "Edo" 22))
(set-being-age! edo 23)
(say-hello edo) ; => Hi Edo (23)
(struct may-being being
  ((alive? #:mutable))) ; inheritance

; closures
(define (make-adder n)
  (lambda (x) (+ x n)))
(define add5 (make-adder 5))
(add5 10) ; => 15

; useful functions
(map (lambda (x) (+ x 5)) '(1 2 3)) ; => '(6
    7 8)
(filter (lambda (x) (>= x 10)) '(1 10 100))
    ; => '(10 100)
; (foldl f i L) = f(L_n, f(L_2, f(L_1, i)))
(foldl cons '() '(1 2 3)) ; => '(3 2 1)
; (foldr f i L) = f(L_1, f(L_2, F(L_n, i)))
(foldr cons '() '(1 2 3)) ; => '(1 2 3)
(foldl * 1 '(1 2 3 4)) ; => 24
; apply f first to the first, going from the
    left
(define (fold-left f i L)
  (if (null? L)
      i
      (fold-left f (f (car L) i) (cdr L))))
; apply f first to the last, going from the
    right
(define (fold-right f i L)
  (if (null? L)
      i
      (f (car L) (fold-right f i (cdr L)))))

; macro
(define-syntax while
  (syntax-rules ()
    ((_ condition body ...) ; (while cond
        (a) (b)...)
     (let loop ()
       (when condition
         (begin
           body ...
           (loop)))))))
(define-syntax my-let*
    (syntax-rules ()
    ;; base (= only one variable )
    ((_ (( var val )) istr ...)
        ((lambda (var) istr ...)
        val))
    ;; more than one
```

```
    ((_ ((var val) . rest) istr ...)
        ((lambda (var)
            (my-let* rest istr ...))
        val))))
(define-syntax my-let
    (syntax-rules ()
        ((_ ((var expr) ...) body ...)
        ((lambda (var ...) body ...) expr
            ...))))
(define-syntax For
  (syntax-rules (from to do) ; extra keyword
    ((_ var from min to max do body ...)
     (let loop ((var min))
       body ...
       (when (< var max)
         (loop (+ var 1)))))))
(For i from 1 to 5 do (displayln i)) ; => 1
    2 3 4 5

; continuations
(define saved-cont #f)
(define (test-cont)
  (let ((x 0))
    (call/cc (lambda (k) (set! saved-cont
        k)))
    (set! x (+ x 1))
    (displayln x)))
(test-cont) ; => 1
(saved-cont) ; => 2
(define other-cont saved-cont)
(test-cont) ; => 1
(other-cont) ; => 3
(saved-cont) ; => 2

; exception handling with continuations
; handlers
(define *handlers* (list))
; push new handler
(define (push-handler proc)
    (set! *handlers* (cons proc *handlers*)))
; pop handler
(define (pop-handler)
    (let ((h (car *handlers*)))
        (set! *handlers* (cdr *handlers*))
        h))
; throw
(define (throw x)
    (if (pair? *handlers*)
        ((pop-handler) x)
        (apply error x)))
; try catch (macro)
(define-syntax try
    (syntax-rules ( catch )
        ((_ exp1 ...
            (catch what hand ...))
        (call/cc (lambda (exit)
            ; install the handler
            (push-handler (lambda (x)
                (if (equal? x what)
                    (exit
                        (begin
                            hand ...))
                    (throw x))))
            (let ((res ;; evaluate the body
                (begin exp1 ...)))
                ; ok : discard the handler
                (pop-handler)
                res))))))
; example
(define (foo x)
    (display x) (newline)
    (throw "hello"))
(try
    (display "Before foo")
    (newline)
    (foo "hi !")
    (display "After foo") ; unreached code
    (catch "hello"
        ;this is the handler block
        (display "I caught a throw.")
            (newline)
        #f))
; it prints:
;
; Before foo
; hi !
; I caught a throw.
;
; and returns #f

; closure as objects
(define (make-simple-object)
  (let ((my-var 0))  ; attributes
    (define (my-add x)
      (set! my-var (+ my-var x))
      my-var)
    (define (get-my-var)
      my-var)
    (define (my-display)
      (printf "my-var=~a~n" my-var))

    (lambda (message . args)
      (apply (case message
               ((my-add) my-add)
               ((get-my-var) get-my-var)
               ((my-display) my-display)
               (else (error "Unknown
                   method")))
             args))))
(define obj (make-simple-object))
(obj 'my-add 3)
(obj 'get-my-var) ; => 3
(obj 'my-display)
; inheritance
(define (make-son)
  (let ((parent (make-simple-object))
        (name "test"))
    (define (hello)
      "hi")
    (define (my-display)
      (printf "My name is ~a and " name)
      (parent 'my-display))
    (lambda (message . args)
      (case message
        ((hello) (apply hello args))
        ((my-display) (apply my-display
            args))
        (else (apply parent (cons message
            args)))))))
(define obj2 (make-son))
(obj2 'hello) ; => hi
(obj2 'my-display) ; => My name is test and
    my-var=0

; Proto-oo
(define new-object make-hash)
(define clone hash-copy)
(define-syntax !! ; setter
  (syntax-rules ()
    ((_ object msg new-val)
     (hash-set! object 'msg new-val))))
(define-syntax ?? ; getter
  (syntax-rules ()
    ((_ object msg)
     (hash-ref object 'msg))))
(define-syntax -> ; send message
  (syntax-rules ()
    ((_ object msg arg ...)
     ((hash-ref object 'msg) object arg
         ...))))

(define Pino (new-object))
(!! Pino name "Pino")
(!! Pino hello (lambda (self) (printf "Name
    is ~v~n" (?? self name))))
(!! Pino set-name (lambda (self x) (!! self
    name x)))
(define Pina (clone Pino))
(-> Pina set-name "Pina")
(-> Pina hello) ; => Name is Pina
; inheritance
(define (son-of parent)
  (let ((o (new-object)))
    (!! o <<parent>> parent)
    o))
(define (dispatch object msg)
  (if (eq? object 'unknown)
      (error "Unknown message" msg)
      (let ((slot (hash-ref object msg
          'unknown)))
        (if (eq? slot 'unknown)
```

```
            (dispatch (hash-ref object
                '<<parent>> 'unknown) msg)
            slot))))
(define-syntax ??? ; reader ** should be ??
    **
  (syntax-rules ()
    ((_ object msg)
     (dispatch object 'msg))))
(define-syntax --> ; send message ** should
    be -> **
  (syntax-rules ()
    ((_ object msg arg ...)
     ((dispatch object 'msg) object arg
         ...))))

(define Glenn (son-of Pino))
(!! Glenn name "Glenn")
(!! Glenn age 50)
(--> Glenn hello)
; (--> Glenn boh) ; => error: Unknown
    message boh

; Implementation of call-by-need
```

```
; promise
(struct promise (
    proc
    value?
    ) #:mutable)

; delay (macro since we must not evaluate
    expr)
(define-syntax delay
  (syntax-rules ()
    ((_ (expr ...))
     (promise (lambda () (expr ...))
              #f))))

; force evaluation
(define (force prom)
  (cond
    ((not (promise? prom)) prom)
    ((promise-value? prom) (promise-proc
        prom))
    (else
      (set-promise-proc! prom ((promise-proc
          prom)))
      (set-promise-value?! prom #t)
      (promise-proc prom))))

(define (infinity)
  (+ 1 (infinity)))
(define lazy-inf (delay (infinity)))
(define (fst x y) x)

; (fst 3 (infinity)) ; => doesn't terminate
(force (fst 3 lazy-inf)) ; => 3
(fst 3 lazy-inf) ; => 3

; Currying
(define (sum-square x)
  (lambda (y)
    (+ (* x x) (* y y))))

(define ((sum-square2 x) y)
  (+ (* x x) (* y y)))

((sum-square 1) 2) ; => 5
((sum-square2 2) 3) ; => 13
```

```haskell
module Haskell where

-- Types
-- 5 :: Integer
-- 'a' :: Char
-- inc :: Integer -> Integer
-- [1,2,3] :: [Integer]  -- equivalent to 1:(2:(3:[]))
-- ('b',4) :: (Char, Integer)

-- Function
inc n = n + 1
-- match top to bottom
len :: [a] -> Integer
len [] = 0
len (x:xs) = 1 + len xs

-- Lambda
lambda = \x y -> 1+x+y

-- Lists
l = [1,2,3]
elem 2 l -- => True
l2 = [4,5,6]
l3 = l ++ l2 -- => [1,2,3,4,5,6]
h = head [1,2,3] -- => 1
t = tail [1,2,3] -- => [2,3]

-- Union and struct
data Bool' = False' | True'
data Pnt a = Pnt a a
-- Pnt 2.3 5.7 is a value
-- Pnt Bool is a type
data Tree a = Empty | Leaf a | Branch (Tree a) (Tree a)
aTree = Branch (Leaf 'a') (Branch (Leaf 'b') (Leaf 'c'))
data List a = Null | Cons a (List a)
-- equivalent to data [a] = [] | a : [a]
data Point = Point {pointx, pointy :: Float}
-- p = Point 1.0 2.0  --  pointx p => 1.0
type String = [Char] -- type alias

fringe :: Tree a -> [a]
fringe (Leaf x) = [x]
fringe (Branch left right) = fringe left ++ fringe right

-- Type class
class Listoid l where
    listoidcons :: a -> l a -> l a
    listoidunit :: a -> l a
    listoidappend :: l a -> l a -> l a
    listoidfirst :: l a -> a
    listoidlast :: l a -> a
    listoidrest :: l a -> l a
data LL a = Head a (LL a) | Node a (LL a) | Tail a deriving Show
lconcat (Tail a) (Head l xs) = Node a xs
lconcat (Node a xs1) r = Node a (lconcat xs1 r)
lconcat (Head _ xs1) r@(Head l xs2) = Head l (lconcat xs1 r)

instance Listoid LL where
    listoidcons a (Head l xs) = Head l (Node a xs)
    listoidunit a = Head a (Tail a)
    listoidappend l r = lconcat l r
    listoidfirst (Head _ (Node a _)) = a
    listoidfirst (Head _ (Tail a)) = a
    listoidlast (Head l _) = l
    listoidrest (Head l (Node _ rest)) = Head l rest
    listoidrest (Head _ (Tail _)) = error "listoidrest on unit"

-- map :: (a -> b) -> [a] -> [b]
mymap f [] = []
mymap f (x:xs) = f x : mymap f xs
-- partially applied infix operator, also: (+ 1) (+)
r1 = map (1 +) [1,2,3] -- => [2,3,4]
-- foldl
foldl' f z [] = z
foldl' f z (x:xs) = foldl' f (f z x) xs
r10 = foldl (+) 0 [1,2,3] -- => ((0+1)+2)+3
-- foldr
foldr' f z [] = z
foldr' f z (x:xs) = f x (foldr' f z xs)
r11 = foldr (+) 0 [1,2,3] -- => 1+(2+(3+0))
-- concat l = foldr (++) [] l
r12 = concat [[1,2],[3],[4,5]] -- => [1,2,3,4,5]

-- zip :: [a] -> [b] -> [(a,b)]
r7 = zip [1,2,3] "ciao" -- => [(1,'c'), (2,'i'), (3,'a')]

-- Composition
dd = (*2) . (1+) -- dd(x) = (*2)(1+)(x) = 2*(1+x)
r2 = dd 6 -- => 14
-- $ operator for avoiding parentheses
r3 = (10*) $ 5+3 -- => 80

-- Never-ending Computations
numsFrom n = n : numsFrom (n + 1)
squares = map (^2) (numsFrom 0)
r4 = take 5 squares -- => [0,1,4,9,16]
r5 = [1,1..] -- => [1,1,1,1,...]
r6 = [6..] -- => [6,7,8,9,...]

-- List Comprehensions
r8 = [(x,y) | x <- [1,2], y <- [3,4]] -- => [(1,3),(1,4),(2,3),(2,4)]
fib = 1 : 1 : [a+b | (a,b) <- zip fib (tail fib)]

-- Pattern Matching
sign x | x > 0 = 1
       | x == 0 = 0
       | x < 0 = -1
mytake 0 _ = []
mytake _ [] = []
mytake n (x:xs) = x : mytake (n-1) xs
mytake2 m ys = case (m,ys) of
    (0,_) -> []
    (_,[]) -> []
    (n,x:xs) -> x : mytake2 (n-1) xs
```

```
takeWhile' _ [] = []
takeWhile' p (x:xs) = if p x
    then x : takeWhile' p xs
    else []

-- let
r9 = let x = 3
         y = 12
     in x+y -- => 15
powerset set = powerset' set [[]] where
    powerset' [] out = out
    powerset' (e:set) out = powerset' set (out ++ [e:x | x <- out])

-- seq a b -- returns b only after a completed
-- $! :: (a -> b) -> a -> b
f $! x = seq x (f x)

-- infix operators
infixr 9 -*- -- right associative, max precedence
x -*- y = [(i,j) | i <- x, j <- y]

-- class Eq
instance (Eq a) => Eq (Tree a) where
    Leaf a == Leaf b = a == b
    (Branch l1 r1) == (Branch l2 r2) = (l1 == l2) && (r1 == r2)
    _ == _ = False
-- Eq defines automatically /=
-- Ord is subclass of Eq:
-- class (Eq a) => Ord a where
--    (<), (<=), (>=), (>) :: a -> a -> Bool
--    min, max :: a -> a -> Bool
-- only <= is required

-- class Show
instance Show a => Show (Tree a) where
    show (Leaf a) = show a
    show (Branch x y) = "<" ++ show x ++ " | " ++ show y ++ ">"

-- Maps
--import Data.Map
--exmap = let m = fromList [("nose", 11), ("emerald", 27)]
--            n = insert "rust" 99 m
--            o = insert "nose" 9 n
--        in (m ! "emerald", n ! "rust", o ! "nose") -- (27,99,9)
-- Arrays
-- exarr = let m = listArray (1, 3) ["alpha", "beta", "gamma"]
--             n = m // [(2, "Beta")]
--             o = n // [(1, "Alpha"), (3, "Gamma")]
--             in (m ! 1, n ! 2, o ! 1) -- ("alpha", "Beta", "Alpha")
-- // is for update/insert m // [(1, "Alpha")]

-- Foldable: a type that can be used with foldr
-- Only foldr needed
-- foldl f a bs = (foldr (\b g x -> g (f x b)) id bs) a
instance Foldable Tree where
    foldr f z Empty = z
    foldr f z (Leaf x) = f x z
    foldr f z (Branch l r) = foldr f (foldr f z r) l

-- Functor: a type that can be mapped
instance Functor Tree where
    fmap f Empty = Empty
    fmap f (Leaf x) = Leaf (f x)
    fmap f (Branch l r) = Branch (fmap f l) (fmap f r)
-- functor laws:
--  fmap id = id
--  fmap (f . g) = fmap f . fmap g

-- Applicative Functor:
--   pure a --> an instance of a container with a
--   fs <*> xs --> like fmap but instead of a function, a container
--                 with the functions
concatMap' f l = concat $ map f l
--instance Applicative [] where
--    pure x = [x]
--    fs <*> xs = concatMap' (\f -> map f xs) fs
tconc Empty t = t
tconc t Empty = t
tconc l r = Branch l r
tconcat t = foldr tconc Empty t
tconcatMap f t = tconcat $ fmap f t
instance Applicative Tree where
    pure x = Leaf x
    fs <*> xs = tconcatMap (\f -> fmap f xs) fs
-- replace each function f in fs with f(xs), obtaining a tree of
    trees,
-- then concatenate all those trees obtaining a tree

-- Monad: algebraic data type containing computation, it can be
    chained
-- to build ordered sequence
-- class Applicative m => Monad m where
--    -- chain 2 monads passing the output of the first to the second
--    (>>=) :: m a -> (a -> m b) -> m b
--    -- chain 2 monads ignoring the output of the first
--    (>>) :: m a -> m b -> m b
--    m >> k = m >>= \_ -> k
--    -- inject a value into the monad
--    return :: a -> m a
--    return = pure
--    -- fail with a message
--    fail :: String -> m a
--    fail = error

-- Monad laws:
--   return is the identity element:
--       (return x) >>= f  <=>  f x
--       m >>= return  <=>  m
--   associativity for binds
--       (m >>= f) >>= g  <=>  m >>= (\x (f x >>= g))

-- do notation
-- do e1 ; e2      <=>  e1 >> e2
-- do p <- e1; e2  <=>  e1 >>= \p -> e2
```

```
esp :: IO Integer
esp = do x <- return 4
         return (x+1) -- => 5
--instance Monad [] where
--    xs >>= f = concatMap f xs
--    fail _ = []
instance Monad Tree where
    xs >>= f = tconcatMap f xs
    fail _ = Empty
exmon :: (Monad m, Num r) => m r -> m r -> m r
exmon m1 m2 = do x <- m1
                 y <- m2
                 return $ x-y
r13 = exmon [10,11] [1,7] -- => [9,3,10,4]

-- State Monad
data State st a = State (st -> (st, a))
instance Functor (State st) where
    fmap f (State g) = State (\s -> let (s', x) = g s
                                    in (s', f x))
instance Applicative (State st) where
    pure x = State (\t -> (t, x))
    (State f) <*> (State g) =
        State (\state -> let (s, f') = f state
                             (s', x) = g s
                         in (s', f' x))
instance Monad (State state) where
    State f >>= g = State (\olds ->
                            let (news, value) = f olds
                                State f' = g value
                            in f' news)
runStateM :: State state a -> state -> (state, a)
runStateM (State f) st = f st
ex = runStateM
     (do x <- return 5
         return (x+1))
     333  -- => (333,6)
getState = State (\state -> (state, state))
putState new = State (\_ -> (new, ()))
ex' = runStateM
      (do x <- getState; return (x+1))
      333 -- => (333,334)
ex'' = runStateM
       (do x <- getState
           putState (x+1)
           x <- getState
           return x)
       333 -- => (334,334)
mapTreeM f (Leaf a) = do
    b <- f a
    return (Leaf b)
mapTreeM f (Branch l r) = do
    l' <- mapTreeM f l
    r' <- mapTreeM f r
    return (Branch l' r')
numberTree tree = runStateM (mapTreeM number tree) 100
    where number v = do cur <- getState
                        putState (cur+1)
                        return (v,cur)
logTree tree = runStateM (mapTreeM log tree) "Log\n"
    where log v = do cur <- getState
                     putState (cur ++ "[node] " ++ (show v) ++ "\n")
                     return v
```

```erlang
-module(test).
-compile(export_all).
%-export([add/2]).

% Builtins
% date() time() length([1,2,3]) size({1,2,3})
% atom_to_list(atom) => "atom"
% list_to_tuple([1,2,3]) => {1,2,3}
% tuple_to_list({1,2,3}) => [1,2,3]
% integer_to_list(1234) => "1234"

% lists module
% hd(List) -> Element -- Returns the first element of the list.
% tl(List) -> List -- Returns the list minus its first element.
% length(List) -> Integer -- returns the length of the list.
% all(Pred, List) -> bool()
% any(Pred, List) -> bool()
% append(List1, List2) -> List3 is equivalent to A ++ B.
% filter(Pred, List1) -> List2
%     lists:filter(fun(X) -> X =< 3 end, [3, 1, 4, 1, 6]). % Result
    is [3,1,1]
% flatten(DeepList) -> List -- Returns a flattened version of
    DeepList.
% foldl(F, I, L) = F(L_n, ... F(L_2, F(L_1, I)))
% foldr(F, I, L) = F(L_1, F(L_2, ... F(L_n, I)))
% foldl(Fun, Acc0, List) -> Acc1
%     Example: lists:foldl(fun(X, Y) -> X + 10 * Y end, 0, [1, 2,
    3]). % Result is 123
% foreach(Fun, List) -> void()
% map(Fun, List1) -> List2
%     Example: lists:map(fun(X) -> 2 * X end, [1, 2, 3]). % Result
    is [2,4,6]
% member(Elem, List) -> bool() -- Search Elem in List
% partition(Pred, List) -> {Satisfying, NonSatisfying}
% reverse(List1) -> List2
% seq(From, To) -> Seq -- [From, To], inclusive.
% seq(From, To, Incr) -> Seq
% sort(List1) -> List2 -- Returns a list containing the sorted
    elements of List1.
% unzip(List1) -> {List2, List3}
% zip(List1, List2) -> List3 -- L1 and L2 of the same length


% Variables (UpperCamelCase)
diff(A, B) ->
    C = A - B,
    C.

% Functions
add(A, B) ->
    A + B.

% Definition via pattern, note the ';'
fact(0) -> 1;
fact(N) when N > 0 -> N * fact(N-1).
area({square, Side}) -> Side * Side;
area({circle, Radius}) -> 3.14 * Radius * Radius;
```

```erlang
area({triangle, A, B, C}) ->
    S = (A + B + C) / 2,
    math:sqrt(S*(S-A)*(S-B)*(S-C)).
% Guards after `when`:
% is_number(X) is_integer(X) is_float(X) is_atom(X) is_tuple(X)
    is_list(X)
% X > Y + Z
% X =< Y  (not <=)
% A and B (or/not/...)
% X =:= Y  <=>  X === Y
% X == Y  <=>  equal with float-int coercion
% X =/= Y  <=>  X !== Y

% Case
casef(X) ->
    case lists:member(a, X) of
        true -> present;
        false -> absent
    end.
iff(X) ->
    if
        is_integer(X) -> integer;
        is_tuple(X) -> tuple;
        true -> boh
    end.

% Symbols (lowercase or 'escaped')
sym() ->
    ciao.

% Tuples
tuple() ->
    {123, abc, {a, b}}.

% List
list() ->
    [1, 2, 3] ++ [4, 5].
cons(X, L) ->
    [X | L].

% List comprehensions
comp() ->
    [{X,Y} || X <- [-1,0,1], Y <- [one, two], X >= 0].
%   => [{0,one},{1,one},{0,two},{1,two}]

% Matching
match() ->
    [A,B|C] = [1,2,3,4], % => A=1, B=2, C=[3,4]
    {D,E,_} = {1,2,3}. % D=1, E=2

% Maps
map() ->
    Map = #{one => 1, "Two" => 2, 3 => three},
    Map#{one := "uno"}, % return the updated, keep Map unchanged
    #{"Two" := V} = Map,
    V. % => 2
```

```erlang
% apply(Mod, Func, Args)
apply() ->
    apply(?MODULE, add, [1,2]). % => 3

% Lambda
lambda() ->
    fun(X) -> X*X end.
lambda2() ->
    (lambda())(9), % => 81
    lists:map(lambda(), [1,2,3]), % => [1,4,9]
    lists:foldr(fun add/2, 0, [1,2,3]). % => 6

% Processes and messages
% Between 2 processes there is FIFO ordering of messages
actor1() ->
    Pid2 = spawn(?MODULE, actor2, []),
    Pid2 ! {self(), foo}.

actor2() ->
    receive
        {From, Msg} -> io:format("Actor2 received ~w from ~w~n",
            [Msg, From])
    end.
actor2withTimeout(T) ->
    receive
        {From, Msg} -> io:format("Actor2 received ~w from ~w~n",
            [Msg, From])
    after
        T -> io:format("Timeout!") % T in ms
    end.

% Management of processes
startMaster() ->
    Pid = spawn(?MODULE, masterBody, []),
    register(master, Pid).
startWorker() ->
    spawn_link(?MODULE, workerBody, []).
workerBody() ->
    BadLuck = rand:uniform(100) =< 30,
    if
        BadLuck -> exit("Bye.");
        true -> master ! {done}
    end.
masterBody() ->
    startWorker(),
    startWorker(),
    startWorker(),
    handleExits().
handleExits() ->
    process_flag(trap_exit, true),
    receive
        {'EXIT', Pid, normal} ->
            io:format("Process ~p exited normally~n", [Pid]),
            handleExits();
        {'EXIT', Pid, Msg} ->
            io:format("Process ~p died with message: ~s~n",
                      [Pid, Msg]),
            spawn_link(?MODULE, workerBody, []),
            handleExits();
        {done} ->
            io:format("Worker done~n", []),
            handleExits()
    end.
```