**SCHEME**

(displayln "Hello World!")

(define x 5)

(define (say-hello)

(displayln "Hello World from the function!"))

(define (greet name)

(if (string? name)

(displayln (string-append "Hello " name "!"))

(displayln "Input is not a string!")))

(define (greet-only-luca name)

(if (eq? name "Luca")

(displayln (string-append "Hello " name "!"))

(displayln "You are not Luca :(((")))

; Bad practice! We can run out of memory

(define (factorial n)

(if (zero? n)

1

(\* n (factorial (- n 1)))))

; Tail recursion!!!

(define (factorial-tail-rec n acc)

(if (zero? n)

acc

(factorial-tail-rec (- n 1) (\* acc n))))

; Just a label

(define (factorial-2 n)

(let factorial-loop ((curr n)

(acc 1))

(if (zero? curr)

acc

(factorial-loop (- curr 1)(\* acc curr)))))

(define (divisible-by? n m)

(zero? (modulo n m)))

(define (fizzbuzz n)

(cond ((divisible-by? n 15) "FizzBuzz")

((divisible-by? n 5) "Buzz")

((divisible-by? n 3) "Fizz")

(else n)))

(define numbers-0 (range 1 31)) ; Like in Python

(map fizzbuzz numbers-0)

(define numbers-1 (list 1 2 3 4 5))

(define numbers '(1 2 3 4 5))

(define quasi-numbers `(1 2 ,(+ 1 4) 4 5))

; Backtick, unquoting

(map (lambda (x)(+ x 1)) numbers)

(apply + numbers)

(foldl + 0 numbers)

(foldl \* 1 numbers) ; Factorial

(foldl \* 1 (range 1 10))

(define z 3)

(displayln (string-append "z = " (~a z)))

; ~a to convert number to string

(cons 2 (cons 1 '()))

(cons 1 2) ; NOT a proper list

(define (reverse-list lst)

(define (reverse-list-helper x acc)

(displayln (string-append "x = " (~a x) "; acc = " (~a acc)))

(if (null? x)

acc

(reverse-list-helper(cdr x)

(cons (car x) acc))))

(reverse-list-helper lst '()))

(empty? '())

(pair? '())

(pair? '(1))

(define a 'apple)

(set! a 'banana) ; To change the variable value!

(define x 5)

(define (change-value x)

(set! x 2)

(displayln x))

(change-value x)

x ; x will not be overwritten, because in raket we pass by value

(define v (vector 1 2 3)) ; we can't use #(1 2 3) because this way of definition is immutable

(define (change-vector v)

(vector-set! v 1 0) ; change item '2' to '0'

(displayln v))

(change-vector v) ; Values are changing because we are passing a reference!

(define k 10)

(define (f)

k)

(define (g)

(define k 5)

(f))

(g) ; Returns '10' because Statically scoped

(define ada "Ada")

ada

(let ((bob "Bob"))

(displayln ada)

(displayln bob))

; We cannot call 'bob' since it's not in the scope

; bob => not defined

; Named let

(define (loop-ten-times)

(let loop ((i 0))

(when (< i 10)

(displayln (string-append "Loop " (~a (+ i 1)) " times"))

(loop (+ i 1)))))

(define (loop-ten-times-alt)

(let ((i 0))

(let loop ()

(when (< i 10)

(displayln (string-append "Loop-alt " (~a (+ i 1)) " times"))

(set! i (+ i 1))

(loop)))))

(define (split-sum x . xs)

(displayln x)

(displayln xs)

(+ x

(apply + xs)))

; xs is a list, for example, '(2 3 4 5)

; We use 'apply' because 'x' cannont be used for a list in a form '(1 2 3)

(split-sum 1 2 3 4 5)

(define (list-flatten lst)

(cond ((null? lst) lst)

((not (list? lst)) (list lst))

(else (append (list-flatten (car lst))(list-flatten (cdr lst))))))

(list-flatten '(1 (2 (3 4) 5 6) (7 8)))

;; STRUCTURES

(struct person

(name

(age #:mutable)))

(define p1 (person "Ada" 25))

(define p2 "Bob")

(person? p1)

(person? p2)

(set-person-age! p1 26)

(person-age p1)

(struct node

((value #:mutable)))

(struct binary-node node

(left

right))

(define (leaf? n)

(and (node? n) (not (binary-node? n))))

(define a-tree (binary-node 2 (binary-node 3 (node 4) (node 2)) (node 1)))

(define (print-tree n)

(displayln (node-value n))

(unless (leaf? n)

(print-tree (binary-node-left n))

(print-tree (binary-node-right n))))

(print-tree a-tree)

; Apply is a higher-order function for trees

; applies a function f(x) to the value x of each tree node

(define (tree-apply f n)

(set-node-value! n (f (node-value n)))

(unless (leaf? n)

(begin

(tree-apply f (binary-node-left n))

(tree-apply f (binary-node-right n)))))

(tree-apply add1 a-tree)

(println "---")

(print-tree a-tree)

(tree-apply (lambda (x) (+ 5 x)) a-tree)

(println "---")

(print-tree a-tree)

; Closure

(define (make-counter)

(let ((count 0))

(lambda ()

(set! count (+ 1 count))

count)))

(define counter1 (make-counter)) ; Two independent counters

(define counter2 (make-counter))

(displayln (counter1))

(displayln (counter1))

(displayln (counter2))

; MACROS

(define (say-hello . people) ; to pass a list

(displayln (string-append "Hello " (string-join people))))

(say-hello "Ada" "Bob" "Carl")

; In a form of a macro

(define-syntax hello

(syntax-rules () ; In this list we put the literals that should not be binded (decorative ones)

((\_ names ...) ; 'names ...' matches a sequence of items

(displayln (string-append "Hello " (string-join (list names ...)))))))

(hello "Lucus" "Uca" "Bibi Lu")

(define-syntax while

(syntax-rules (do) ; In this list we put the literals that should not be binded (decorative ones)

((\_ cond do body ...)

(let loop ()

(when cond

(begin

body ...

(loop)))))))

(displayln "while-do loop")

(define i 0)

(while (< i 5) do

(set! i (+ 1 i))

(displayln i))

; (for x in <list> <body>)

(define-syntax for

(syntax-rules (in)

((\_ item in lst body ...)

(begin

(unless (list? lst)

(error "Not a list"))

(let loop ((item (car lst))

(rest (cdr lst)))

(begin

body ...

(unless (null? rest)

(loop (car rest)(cdr rest)))))))))

(displayln "for-in loop")

(for x in '(1 2 3 4 5) (displayln x))

;; (for i in "pizza" (displayln x))

; RECURSIVE MACROS

; you can call the macro itself after you define it.

; useful if you have multiple syntax rules that match different conditions

(define-syntax say

(syntax-rules (hello goodbye)

((\_ hello) (displayln "hello"))

((\_ goodbye) (displayln "goodbye"))

((\_ ...) (displayln "whatever...")))) ; catch all case

(say hello)

(define (list-fruits)

(call/cc

(lambda (k)

(displayln "apple")

(k (displayln "banana"))

(displayln "carrot"))))

(list-fruits)

(define (lst) "item")

(lst)

; WHILE WITH A BREAK

(define-syntax while-break

(syntax-rules (break-id:); We need 'break-id' because of hygienic macros

; we can't call 'break' from outside the syntax-rule

((\_ cond break-id: break body ...)

(call/cc (lambda (break)

(let loop ()

(when cond

(begin body ...)

(loop))))))))

(define y 5)

(while-break (> y 0) break-id: stop ; Any custom name here

(when (= y 2) (stop))

(set! y (- y 1))

(displayln y))

(define \*exit-store\* '()) ; Global variable

(define (break v)

((car \*exit-store\*) v))

; FOR WITH A BREAK

(define-syntax for

(syntax-rules (from to do)

((\_ var from min to max do body ...)

(let\* ((min1 min)

(max1 max)

(inc (if (< min1 max1) + -)))

(call/cc (lambda (k)

(set! \*exit-store\* (cons k \*exit-store\*))

(let loop ((var min1))

body ...

(unless (= var max1)

(loop (inc var 1))))))

(set! \*exit-store\* (cdr \*exit-store\*))))))

(displayln "FOR")

(for i from 1 to 10 do

(displayln i)

(when (= i 5) ( break #t)))

; WHILE WITH A BREAK

(define-syntax while-2

(syntax-rules (do)

((\_ cond do body ...)

(begin

(call/cc (lambda (k)

(set! \*exit-store\* (cons k \*exit-store\*))

(let loop ()

(when cond

body ...

(loop)))))

(set! \*exit-store\* (cdr \*exit-store\*))))))

(define a 5)

(displayln "WHILE")

(while-2 (> a 0) do

(displayln a)

(set! a (- a 1))

(when (= a 2) (break #t)))

\*exit-store\*

; EXCEPTIONS

(define \*handlers\* '())

; Utility functions

(define (push-handler proc)

(set! \*handlers\* (cons proc \*handlers\*)))

(define (pop-handler)

(let ((head (car \*handlers\*)))

(set! \*handlers\* (cdr \*handlers\*))

head))

(define (throw x) ; error simulator!

(if (pair? \*handlers\*)

((pop-handler) x)

(apply error x))) ; if list of processes is empty

(push-handler displayln)

(throw 5)

;(throw 5)

(define-syntax try

(syntax-rules (catch)

((\_ expr ... (catch exception-id exception-body ...))

(call/cc (lambda(exit)

(push-handler (lambda (x)

(if (equal? x exception-id)

(exit (begin exception-body ...))

(throw x)))) ; else

(let ((res (begin expr ...)))

(pop-handler)

res))))))

(define (foo)

(displayln "Foo")

(throw "bad-foo"))

(try

(displayln "Before foo")

(foo)

(displayln "After foo")

(catch "bad-foo"

(displayln "I caught a throw")

#f))

; if we try to catch an unregistered exception,

; we'll have an error: contract violation

; NON-DETERMINISM

(define (is-sum-of sum)

(unless (and (>= sum 0) (<= sum 10))

(error "out of range" sum))

(let ((x (choose '(0 1 2 3 4 5)))

(y (choose '(0 1 2 3 4 5))))

(displayln (string-append (~a x) "+" (~a y) "=" (~a sum) "?"))

(if (= (+ x y) sum)

(list x y)

(begin

(displayln "is-sum-of fail")

(fail)))))

(define \*paths\* '())

(define (push-path x)

(set! \*paths\* (cons x \*paths\*)))

(define (pop-paths)

(let ((p1 (car \*paths\*)))

(set! \*paths\* (cdr \*paths\*))

p1))

(define (choose choices)

(if (null? choices)

(begin

(displayln "choice fail")

(fail))

(call/cc (lambda (k)

; save checkpoint

; backtrack to k, and choose again from the rest

(push-path (lambda ()

(k (choose (cdr choices)))))

(car choices))))) ; return the current choice

(define fail (lambda ()

(if (null? \*paths\*)

(error "no paths")

((pop-paths)))))

; double (()) because we're calling the returned continuation

(is-sum-of 6)

; Object-Oriented Programming

; With CLOSURES

#|

class Person {

public name: string

private age: int

// constructor

Person(name, age) {

this.name = name

this.age = age

}

int growOlder(years: int) {

this.age += int

}

}

Person bob = Person(Bob, 25)

bob.getName()

|#

(define (new-person ; constructor

initial-name

initial-age)

; attributes

(let ((name initial-name)

(age initial-age))

; methods

(define (get-name)

name)

(define (grow-older years)

(set! age (+ age years))

age)

(define (show)

(display "Name: ")

(displayln name)

(display "Age: ")

(displayln age))

; dispatcher - to handle calls to methods

(lambda (message . args)

(apply (case message

((get-name) get-name)

((grow-older) grow-older)

((show) show)

(else (error "unknown method")))

args))))

(define ada (new-person "Ada" 25))

(ada 'show)

(define bob (new-person "Bob" 27))

(bob 'get-name)

(bob 'grow-older 3)

(bob 'show)

; Inheritance

(define (new-superhero name age init-power)

(let ((parent (new-person name age)) ; inherits attrs/methods

(power init-power))

(define (use-power)

(display name)(display " uses ")(display power)(displayln "!"))

(define (show)

(parent 'show)

(display "Power: ")(displayln power))

(lambda (message . args)

(case message

((use-power) (apply use-power args))

((show) (apply show args))

(else (apply parent (cons message args)))))))

(define superman (new-superhero "Clark Kent" 32 "Flight"))

(superman 'show)

(superman 'grow-older 10)

(superman 'use-power)

; PROTOTYPE-BASED OBJECTS

; HASHMAP with attrs/method names as KEY and

; their value/implementation as VALUE

; We need to use MACROS because otherwise we cannot

; quote 'msg' correctly: if we quote 'msg' at runtime

; we are quoting 'msg' itself: 'msg

(define new-obj make-hash)

(define clone hash-copy)

; define-syntax-rule defines a mcro that binds single pattern

; SETTER

(define-syntax-rule (obj-set object msg new-val)

(hash-set! object 'msg new-val))

; GETTER

(define-syntax-rule (obj-get object msg)

(hash-ref object 'msg))

; SEND MESSAGE / USE METHOD

(define-syntax-rule (obj-send object msg arg ...)

((hash-ref object 'msg) ; retrieve method

object arg ...)) ; call method with the object itself as first argument, and any arg

(define carl (new-obj))

(obj-set carl name "Carl")

(obj-set carl show

(lambda (self)

(display "Name: ")(displayln (obj-get self name))))

(obj-set carl say-hi

(lambda (self to)

(display (obj-get self name))

(display " says hi to ")

(displayln to)))

(obj-send carl show)

(obj-send carl say-hi "Dan")

; New object that copies properties of carl

(define dan (clone carl))

(obj-set dan name "Dan")

(obj-send dan show)

(obj-set dan dance

(lambda (self)

(displayln "I'm Dan, I can dance")))

(obj-send dan dance)

(obj-send dan say-hi "Carl")