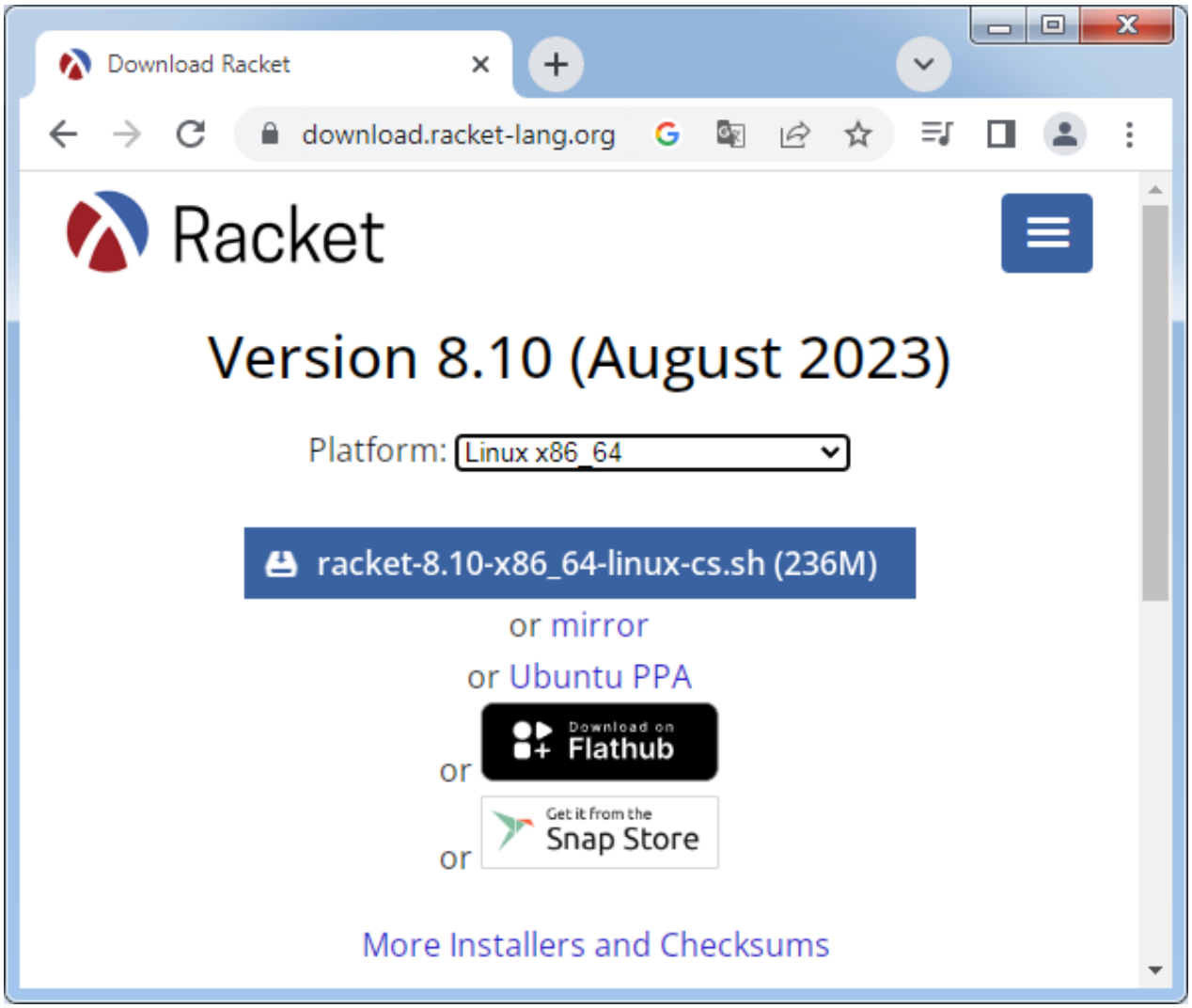
Trabajo Práctico (Individual y Obligatorio)

Intérprete de Racket en Clojure

Racket es un dialecto de la familia de lenguajes de programación de Lisp. Fue creado durante la década de 1990 y ha tenido una notable infuencia sobre lenguajes surgidos posteriormente, como Clojure y Rust. Su versión más reciente, la 8.10, fue lanzada en agosto de 2023.



Los programas escritos en el lenguaje Racket se pueden correr en el REPL (*read-evaluate-print-loop* del sistema Racket o en un IDE (*integrated development environment*) denominado DrRacket.

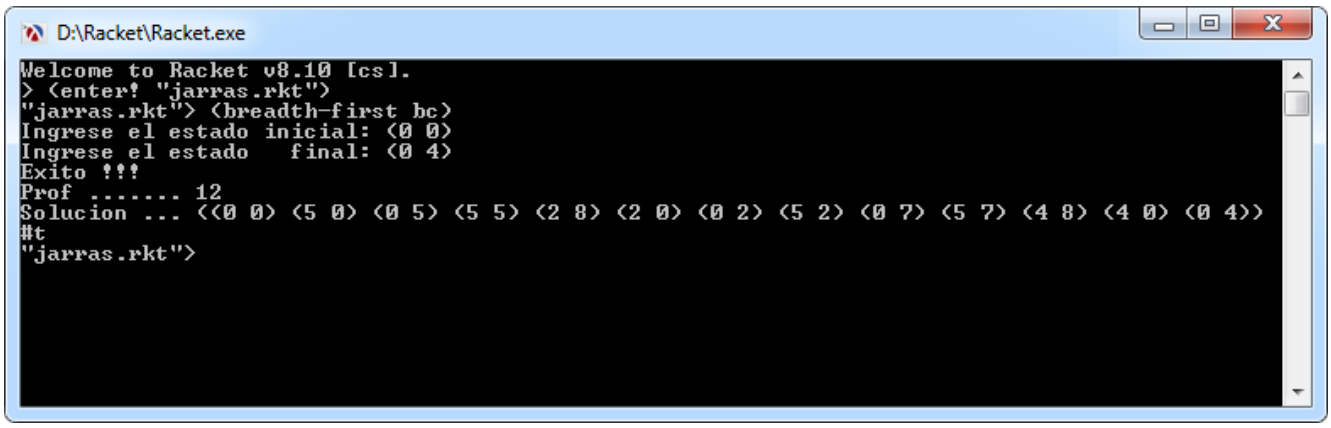
En *The Racket Manifesto*, los autores Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy y Sam Tobin-Hochstadt señalan que *"Racket is a programming language for creating new programming languages"*.

Para darle aún más portabilidad a este sistema, el objetivo del presente trabajo es construir un intérprete de Racket que corra en la JVM (*Java Virtual Machine*). Por ello, el lenguaje elegido para su implementación es **Clojure**.

Deberá poder cargarse y correrse el siguiente **Sistema de Producción**, que resuelve el problema de obtener 4 litros de líquido utilizando dos jarras lisas (sin escala), una de 5 litros y otra de 8 litros.

jarras.rkt

```
#lang racket

;;;;;;;;;;;;;;;;;;; DEFINIR LA BASE DE CONOCIMIENTO ;;;;;;;;;;;;;;;;;;;

(define (jarra5 x) (car x))

(define (jarra8 x) (car (cdr x)))

(define bc '(
      (lambda (x) (if (< (jarra5 x) 5) (list 5 (jarra8 x)) x))
      (lambda (x) (if (> (jarra5 x) 0) (list 0 (jarra8 x)) x))
      (lambda (x) (if (>= (- 5 (jarra5 x)) (jarra8 x)) (list (+ (jarra5 x) (jarra8 x)) 0) x))
      (lambda (x) (if (< (- 5 (jarra5 x)) (jarra8 x)) (list 5 (- (jarra8 x) (- 5 (jarra5 x)))) x))
      (lambda (x) (if (< (jarra8 x) 8) (list (jarra5 x) 8) x))
      (lambda (x) (if (> (jarra8 x) 0) (list (jarra5 x) 0) x))
      (lambda (x) (if (>= (- 8 (jarra8 x)) (jarra5 x)) (list 0 (+ (jarra8 x) (jarra5 x))) x))
      (lambda (x) (if (< (- 8 (jarra8 x)) (jarra5 x)) (list (- (jarra5 x) (- 8 (jarra8 x))) 8) x))
   )
)

;;;;;;;;;;; DEFINIR EL ALGORITMO DE BUSQUEDA DE LA SOLUCION ;;;;;;;;;;;;;

(define inicial '())

(define final '())

(define (breadth-first bc)
   (display "Ingrese el estado inicial: ") (set! inicial (read))
   (display "Ingrese el estado   final: ") (set! final (read))
   (cond ((equal? inicial final) (display "El problema ya esta resuelto !!!") (newline) (breadth-first bc))
         (#t (buscar bc final (list (list inicial)) '()))))

(define (buscar bc fin grafobusq estexp)
    (cond ((null? grafobusq) (fracaso))
          ((pertenece fin (car grafobusq)) (exito grafobusq))
          (#t (buscar bc fin (append (cdr grafobusq) (expandir (car grafobusq) bc estexp))
                (if (pertenece (car (car grafobusq)) estexp) estexp (cons (car (car grafobusq)) estexp))))))

(define (expandir linea basecon estexp)
   (if (or (null? basecon) (pertenece (car linea) estexp))
       '()
       (if (not (equal? ((eval (car basecon)) (car linea)) (car linea)))
          (cons (cons ((eval (car basecon)) (car linea)) linea) (expandir linea (cdr basecon) estexp))
          (expandir linea (cdr basecon) estexp))))

(define (pertenece x lista)
   (cond ((null? lista) #f)
         ((equal? x (car lista)) #t)
         (else (pertenece x (cdr lista)))))

(define (fracaso)
   (display "No existe solucion") (newline) #t)

(define (exito grafobusq)
   (display "Exito !!!") (newline)
   (display "Prof ..... ") (display (- (length (car grafobusq)) 1)) (newline)
   (display "Solucion ... ") (display (reverse (car grafobusq))) (newline) #t)
```



Además, al cargar el archivo `demo.rkt` (publicado en el Aula Virtual de la cátedra en el Campus), se deberá obtener la siguiente salida por pantalla:

```
Welcome to Racket v8.10 [cs].
> (enter! "demo.rkt")

**************************************************
*                   Racket 2023                  *
* Demo de definicion y uso de variables y funciones *
**************************************************

Definicion de variables
-----------------------
> (define u 'u)
> (define v 'v)
> (define w 'w)

Las variables ahora estan en el ambiente.
Evaluandolas se obtienen sus valores:
> u
u
> v
v
> w
w

Una vez definida una variable, con set! se le puede
cambiar el valor:
> (define n 0)
> n
0
> (set! n 17)
> n
17

Definicion de funciones
-----------------------
> (define (sumar a b) (+ a b))
> (define (restar a b) (- a b))
```

```
Las funciones ahora estan en el ambiente.
es posible aplicarlas a valores formando expresiones
que evaluadas generan resultados:
> (sumar 3 5)
8
> (restar 12 5)
7

Racket es un lenguaje de ambito lexico (lexically scoped):
> (define x 1)
> (define (g y) (+ x y))
> (define (f x) (g 2))
> (f 5)
3
[En TLC-Lisp -dynamically scoped- daria 7 en lugar de 3]

Aplicacion de funciones anonimas [lambdas]
------------------------------------------

Lambda con cuerpo simple:
> ((lambda (y) (+ 1 y)) 15)
16

Lambda con cuerpo multiple:
> ((lambda (y) (display "hola!") (newline) (+ 1 y)) 5)
hola!
6

Lambda con cuerpo multiple y efectos colaterales [side effects]:
> ((lambda (a b c) (set! u a) (set! v b) (set! w c)) 1 2 3)
#<void>

Los nuevos valores de las variables modificadas:
> u
1
> v
2
> w
3

Aplicacion parcial:
> (((lambda (x) (lambda (y) (- x y))) 8) 3)
5

El mismo ejemplo anterior, ahora definiendo una funcion:
> (define p (lambda (x) (lambda (y) (- x y))))
> (p 8)
#<procedure:F:/Racket/demo.rkt:118:22>
> ((p 8) 3)
5
```

```
Definicion de funciones recursivas [recorrido lineal]
-----------------------------------------------------

Funcion recursiva con efecto colateral
[deja en la variable d la cantidad de pares]:
> (define (recorrer l)
    (recorrer2 l 0))
> (define d 0)
> (define (recorrer2 l i)
    (cond
      ((null? (cdr l)) (set! d (+ 1 d)) (list (car l) i))
      (#t (display (list (car l) i)) (set! d (+ i 1)) (newline) (recorrer2 (cdr l) d))))
> (recorrer '(a b c d e f))
(a 0)
(b 1)
(c 2)
(d 3)
(e 4)
(f 5)
> d
6

Definicion de funciones recursivas [recorrido "a todo nivel"]
-------------------------------------------------------------

Existencia de un elemento escalar en una lista:
> (define (existe? a l)
    (cond
      ((null? l) #f)
      ((not (list? (car l))) (or (equal? a (car l)) (existe? a (cdr l))))
      (else (or (existe? a (car l)) (existe? a (cdr l))))))
> (existe? 'c '(a ((b) ((d c) a) e f)))
#t
> (existe? 'g '(a ((b) ((d c) a) e f)))
#f

Eliminacion de un elemento de una lista:
> (define (eliminar dat li)
    (cond
      ((null? li) li)
      ((equal? dat (car li)) (eliminar dat (cdr li)))
      ((list? (car li)) (cons (eliminar dat (car li)) (eliminar dat (cdr li))))
      (else (cons (car li) (eliminar dat (cdr li))))))
> (eliminar 'c '(a ((b) ((d c) a) c f)))
(a ((b) ((d) a) f))
> (eliminar '(1 2 3) '(a ((b) (((1 2 3) c) a) c f)))
(a ((b) ((c) a) c f))

Profundidad de una lista:
> (define (profundidad lista)
    (if (or (not (list? lista)) (null? lista)) 0
```

```
        (if (> (+ 1 (profundidad (car lista))) (profundidad (cdr lista)))
            (+ 1 (profundidad (car lista)))
            (profundidad (cdr lista)))))
> (profundidad '((2 3)(3 ((7))) 5))
4
[el valor esperado es 4]

"Planchado" de una lista:
> (define (planchar li)
    (cond
      ((null? li) '())
      ((list? (car li)) (append (planchar (car li)) (planchar (cdr li))))
      (else (cons (car li) (planchar (cdr li))))))
> (planchar '((2 3)(3 ((7))) 5))
(2 3 3 7 5)

Definicion de funciones para "ocultar" la recursividad en la programacion funcional
--------------------------------------------------------------------------------

FILTRAR [selecciona de una lista los elementos que cumplan con una condicion dada]:
> (define (filtrar f l)
    (cond
      ((null? l) '())
      ((f (car l)) (cons (car l) (filtrar f (cdr l))))
      (else (filtrar f (cdr l)))))
> (filtrar (lambda (x) (> x 0)) '(5 0 2 -1 4 6 0 8))
(5 2 4 6 8)

REDUCIR [reduce una lista aplicando de a pares una funcion dada]:
> (define (reducir f l)
    (if (null? (cdr l))
        (car l)
        (f (car l) (reducir f (cdr l)))))
> (reducir (lambda (x y) (if (> x 0) (cons x y) y)) '(5 0 2 -1 4 6 0 8 ()))
(5 2 4 6 8)

MAPEAR [aplica a cada elemento de una lista una funcion dada]:
> (define (mapear op l)
    (if (null? l)
        '()
        (cons (op (car l)) (mapear op (cdr l)))))
> (mapear (lambda (x) (if (equal? x 0) 'z x)) '(5 0 2 -1 4 6 0 8))
(5 z 2 -1 4 6 z 8)

TRANSPONER [transpone una lista de listas]:
> (define (transponer m)
    (if (null? (car m))
        '()
        (cons (mapear car m) (transponer (mapear cdr m)))))
> (transponer '((a b c) (d e f) (g h i)))
((a d g) (b e h) (c f i))
```

```
IOTA [retorna una lista con los primeros n numeros naturales]:
> (define (iota n)
    (if (< n 1)
         '()
         (auxiota 1 n)))
> (define (auxiota i n)
    (if (equal? i n)
        (list n)
        (cons i (auxiota (+ i 1) n))))
> (iota 10)
(1 2 3 4 5 6 7 8 9 10)

Funciones implementadas usando las funciones anteriores
-------------------------------------------------------

Sumatoria de los primeros n numeros naturales:
> (define (sumatoria n) (reducir + (iota n)))
> (sumatoria 100)
5050
[El valor esperado es 5050]

Eliminacion de los elementos repetidos en una lista simple:
> (define (eliminar-repetidos li)
    (reverse (reducir (lambda (x y) (if (existe? x y) y (cons x y))) (reverse (cons '() li)))))
> (eliminar-repetidos '(a b c d e f g d c h b i j))
(a b c d e f g h i j)

Seleccion del enesimo elemento de una lista dada:
> (define (seleccionar n li)
    (if (or (< n 1) (> n (length li)))
        '()
        (car (car (filtrar (lambda (x) (equal? n (car (cdr x)))) (transponer (list li (iota (length li)))))))))
> (seleccionar 5 '(a b c d e f g h i j))
e

Aplicacion de todas las funciones de una lista a un elemento dado:
> (define (aplicar-todas lf x)
    (mapear (lambda (f) (f x)) lf))
> (aplicar-todas (list length cdr car) '((3 2 1)(9 8)(7 6)(5 4)))
(4 ((9 8) (7 6) (5 4)) (3 2 1))

Entrada de datos y salida del interprete
----------------------------------------

Carga de datos desde la terminal/consola:
> (define r 0)
> (define (cargar-r)
    (display "->r: ")(set! r (read))(display "r*2: ")(display (+ r r))(newline))
> (cargar-r)
->r: 7
r*2: 14

Para ver el ambiente [no funciona en Racket v8.10]: (env)
Para salir del interprete: (exit)
```

Características de Racket a implementar en el intérprete*

Valores de verdad

#f: significa *falso* (**nil** es un sinónimo)
#t: significa *verdadero*

Formas especiales (*magic forms*)

cond: evalúa múltiples condiciones
define: define var o func. y la liga a símbolo
enter!: carga un archivo
eval: evalúa una lista
exit: sale del intérprete
if: evalúa una condición
lambda: define una func. anónima
or: evalúa mientras obtenga #f
quote: impide evaluación
set!: redefine un símbolo

Funciones

+: retorna la suma de los argumentos
-: retorna la resta de los argumentos
<: retorna #t si los números son < (ordenados)
>: retorna #t si los números son > (ordenados)
>=: retorna #t si los números son >= (ordenados)
append: retorna la fusión de las listas dadas
car: retorna la 1ra. posición de una lista
cdr: retorna una lista sin su 1ra. posición
cons: retorna inserción de elem. en cabeza de lista
display: imprime un elemento y retorna #<void>
env: retorna el ambiente
equal?: retorna #t si los elementos son iguales
length: retorna la longitud de una lista
list: retorna una lista formada por los args.
list?: retorna #t si el argumento es una lista
newline: imprime salto de línea y retorna #<void>
not: retorna la negación de un valor de verdad
null?: retorna #t si un elemento es ()
read: retorna la lectura de un elemento
reverse: retorna una lista invertida

(*) **env** no está disponible así en Racket

El intérprete deberá comportarse de manera similar a Racket v8.10.

A los efectos de desarrollar el intérprete solicitado, se deberá partir de los siguientes materiales proporcionados por la cátedra en el aula virtual de la materia en el campus FIUBA:

- Apunte de la cátedra: *Interpretación de programas de computadora*, donde se explican la estructura y el funcionamiento de los distintos tipos de intérpretes posibles, en particular la *interpretación recursiva*, que es la estrategia a utilizar en este trabajo práctico.
- Apunte de la cátedra: *Clojure*, donde se resume el lenguaje a utilizar para el desarrollo.
- Tutorial: *Pasos para crear un proyecto en Clojure usando Leiningen*, donde se indica cómo desarrollar un proyecto dividido en código fuente y pruebas, y su despliegue como *archivo jar*.
- Código fuente de un intérprete de Racket sin terminar, para completarlo. El mismo contiene dos funciones que deben ser terminadas y 20 funciones que deben desarrollarse por completo.
- Archivos `jarras.rkt` y `demo.rkt` para interpretar.

Con este trabajo práctico, se espera que las/los estudiantes adquieran conocimientos profundos sobre el proceso de interpretación de programas y el funcionamiento de los intérpretes de lenguajes de programación y que, a la vez, pongan en práctica los conceptos del paradigma de *Programación Funcional* vistos durante el cuatrimestre.

Para aprobar la cursada, se deberá entregar **hasta el 29/11/2023** (por e-mail a dcorsi@fi.uba.ar o a corsi@mail.com un **proyecto** compuesto por el código fuente del intérprete de Racket en Clojure y las pruebas de las funciones desarrolladas (como *mínimo*, las pruebas correspondientes a los ejemplos que acompañan el código fuente del intérprete sin terminar proporcionado por la cátedra).

Al momento de rendir la evaluación final de la materia, se deberá modificar el intérprete presentado en este trabajo práctico, incorporándole alguna funcionalidad adicional.