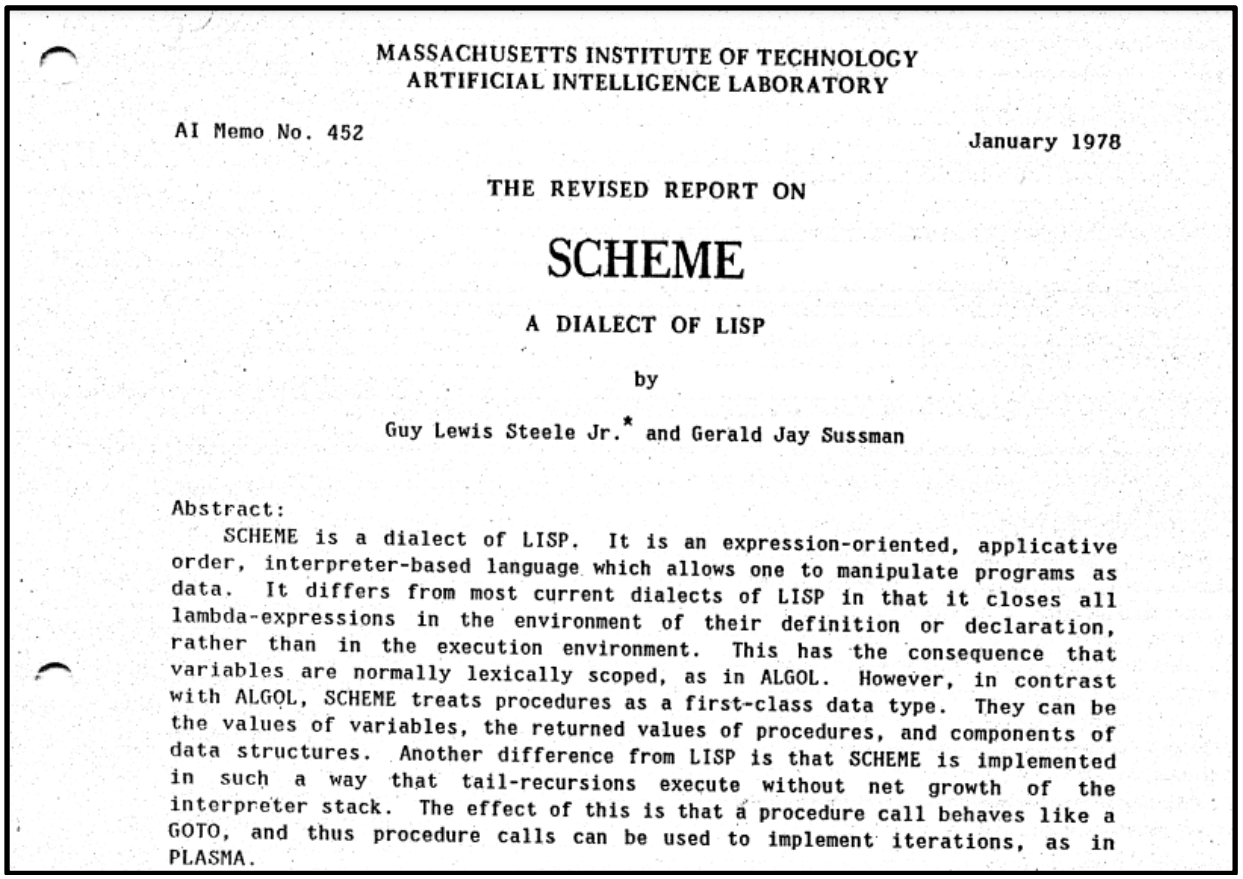
Trabajo Práctico (Individual y Obligatorio)

Intérprete de Scheme en Clojure

Scheme es un dialecto minimalista de la familia de lenguajes de programación de Lisp, creado durante la década de 1970 en el MIT AI Lab y publicado por sus desarrolladores, Guy L. Steele y Gerald Jay Sussman, a través de una serie de memorandos ahora conocidos como *Lambda Papers*. Fue el primer dialecto de Lisp en utilizar variables de ámbito léxico (*lexically scoped*).



MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo No. 452 January 1978

THE REVISED REPORT ON

SCHEME

A DIALECT OF LISP

by

Guy Lewis Steele Jr.* and Gerald Jay Sussman

Abstract:
SCHEME is a dialect of LISP. It is an expression-oriented, applicative order, interpreter-based language which allows one to manipulate programs as data. It differs from most current dialects of LISP in that it closes all lambda-expressions in the environment of their definition or declaration, rather than in the execution environment. This has the consequence that variables are normally lexically scoped, as in ALGOL. However, in contrast with ALGOL, SCHEME treats procedures as a first-class data type. They can be the values of variables, the returned values of procedures, and components of data structures. Another difference from LISP is that SCHEME is implemented in such a way that tail-recursions execute without net growth of the interpreter stack. The effect of this is that a procedure call behaves like a GOTO, and thus procedure calls can be used to implement iterations, as in PLASMA.

Existen dos estándares que definen este lenguaje de programación: el oficial de la IEEE y un estándar de facto conocido como *Revised n-th Report on the Algorithmic Language Scheme*, abreviado como *RnRS*, donde *n* es el número de la revisión. El más reciente es el *R7RS*, que fue publicado en 2013.

La filosofía minimalista de estos estándares conlleva ventajas e inconvenientes. Por ejemplo, escribir un intérprete completo de Scheme es más fácil que implementar uno de Common Lisp; empotrar Lisp en dispositivos con poca memoria también es más factible usando Scheme en lugar de Common Lisp. A los aficionados a Scheme les divierte mucho señalar que su estándar, con solo 50 páginas, es más corto que el índice del libro *Common Lisp: The Language*, de Guy L. Steele.

Scheme tiene una base de usuarios diversa debido a su estructura compacta y su elegancia. Sin embargo, también hay una gran divergencia entre las implementaciones prácticas, tanto que el *Scheme Steering Committee* lo ha llamado "el lenguaje de programación menos portable del mundo" (*the world's most unportable programming language*) y lo considera "una *familia* de dialectos", en lugar de un lenguaje único.

El objetivo del presente trabajo es construir un intérprete de Scheme que corra en la JVM. Por ello, el lenguaje elegido para su implementación es **Clojure**.

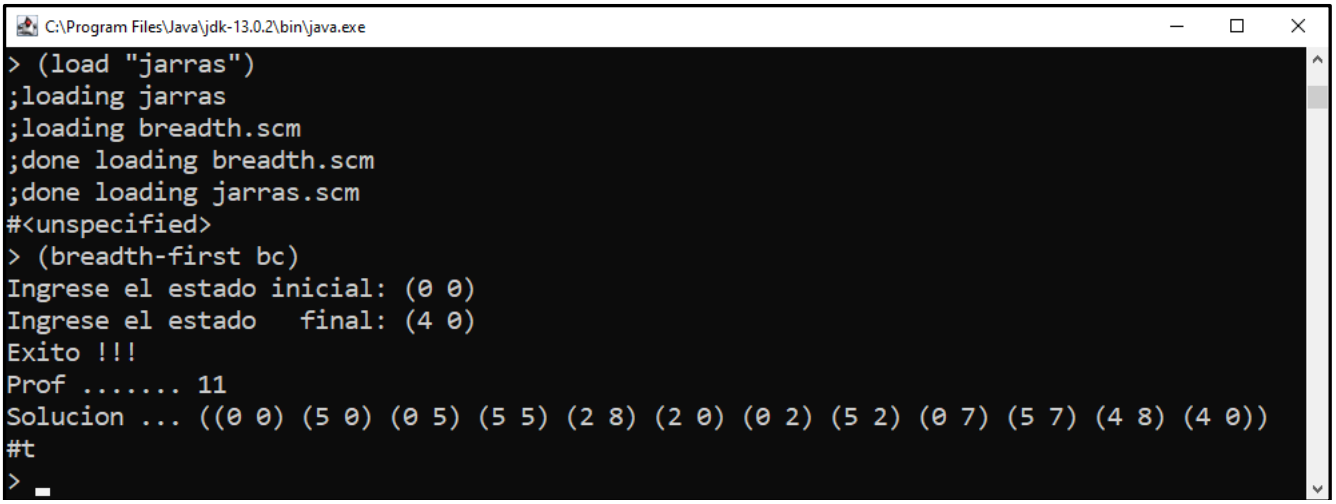
Deberá poder cargarse y correrse el siguiente **Sistema de Producción**, que resuelve el problema de obtener 4 litros de líquido utilizando dos jarras lisas (sin escala), una de 5 litros y otra de 8 litros.

jarras.scm

```
(load "breadth.scm")
(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))))
```

breadth.scm

```
(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.scm` (publicado en el Aula Virtual de la cátedra en el Campus), se deberá obtener la siguiente salida por pantalla:

```
> (load "demo")
;loading demo

*****************************************************
*                   SCHEME 2021                     *
* DEMO DE DEFINICION Y USO DE VARIABLES Y FUNCIONES *
*****************************************************

OBS.: SCHEME NO DISTINGUE MAYUSCULAS DE MINUSCULAS.
      PARA APROVECHAR ESTA CARACTERISTICA, ALGUNAS
      VARIABLES Y FUNCIONES SE DEFINIERON A PROPOSITO
      EN MAYUSCULAS Y OTRAS EN MINUSCULAS.

DEFINICION DE VARIABLES
-----------------------
> (define u 'u)
#<unspecified>
> (define v 'v)
#<unspecified>
> (define w 'w)
#<unspecified>

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)
#<unspecified>
> (set! N 17)
#<unspecified>
> n
17


DEFINICION DE FUNCIONES
-----------------------
> (define (sumar a b) (+ a b))
#<unspecified>
> (define (restar a b) (- a b))
#<unspecified>

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



SCHEME ES UN LENGUAJE DE AMBITO LEXICO (LEXICALLY SCOPED):
> (define x 1)
#<unspecified>
> (define (g y) (+ x y))
#<unspecified>
> (define (f x) (g 2))
#<unspecified>
> (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)
#<unspecified>

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))))
#<unspecified>
> (p 8)
(lambda (y) (- (quote 8) y))
> ((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))
#<unspecified>
> (define D 0)
#<unspecified>
> (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))))
#<unspecified>
> (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))))))
#<unspecified>
> (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))))))
#<unspecified>
> (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)))))
#<unspecified>
> (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))))))
#<unspecified>
> (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)))))
#<unspecified>
> (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)))))
#<unspecified>
> (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)))))
#<unspecified>
> (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)))))
#<unspecified>
> (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)))
#<unspecified>
> (DEFINE (AUXIOTA I N)
    (IF (EQUAL? I N)
        (LIST N)
        (CONS I (AUXIOTA (+ I 1) N))))
#<unspecified>
> (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)))
#<unspecified>
> (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)))))
#<unspecified>
> (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)))))))))
#<unspecified>
> (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))
#<unspecified>
> (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 (cargarR)
    (display "->R: ")(set! R (read))(display "R*2: ")(display (+ R R))(newline))
> (cargarR)
->R: 7
R*2: 14

PARA VER EL AMBIENTE [NO FUNCIONA EN SCM version 5f2]: (env)
PARA SALIR DEL INTERPRETE: (exit)
;done loading demo.scm
#<unspecified>
> (exit)
Goodbye!
```

Características de Scheme 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
exit: sale del intérprete
if: evalúa una condición
lambda: define una func. anónima
load: carga un archivo
or: evalúa mientras no obtenga #f
quote: impide evaluación
set!: redefine un símbolo

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

Funciones

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

El intérprete a desarrollar deberá comportarse de manera similar a SCM version 5f2, Copyright (C) 1990-2006 Free Software Foundation.

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 compilación 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 Scheme sin terminar, para completarlo. El mismo contiene dos funciones que deben ser terminadas y 21 funciones que deben desarrollarse por completo.

• Archivos `jarras.scm`, `breadth.scm` y `demo.scm` 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 15/12/2021** (por e-mail a **dcorsi@fi.uba.ar**) un proyecto compuesto por el código fuente del intérprete de Scheme en Clojure y las pruebas de las funciones desarrolladas (como mínimo, se deberán incluir 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.