Trabajo Práctico (Individual y Obligatorio)

Intérprete de TLC-LISP en Clojure

TLC-LISP es un dialecto extinto de LISP. Fue lanzado en la década de 1980, como se describe en este artículo de InfoWorld del 19 de enero de 1981 (Vol. 3, No. 1):

TLC-LISP from the LISP Company

By Jonathan Sachs

The LISP Company (TLC) has introduced a LISP for Z80 computers running CP/M in 48K RAM or more. LISP is unlike "classical" languages such as FORTRAN or BASIC in both appearance and in function. Other languages are designed to manipulate individual items of data; LISP excels at processing lists of data. LISP's list-processing ability distinguishes it from most other high-level languages.

LISP can create and change data elements at run time. For example, a LISP program can read a word from the terminal, create a variable with that word as its name, and then build a list of arbitrary shape as the varaiable's value.

LISP stores a program and its data during execution in exactly the same way, so a program can "write itself" as it goes along. These characteristics make LISP a natural choice for writing programs that process patterns of any sort.

However, LISP was designed by and for researchers in mathematical logic and it looks strange to people without background in that area.

To get an experienced LISP programmer's perspective on TLC-LISP, I showed the system to Steven Gadol, a software researcher at Hewlett-Packard. His comments follow:

"It's a very competent implementation. It has a substantial subset of the features found in MACLISP [most widely used in artificial-intelligence research]. It has all the necessary basic functions for writing useful programs. The dialect corresponds closely to the ones used in current books on the use of LISP.

"I noticed there was no PROG feature, although a combination of the LET feature and PROGN gives you approximately the same effect.

"One major shortcoming is the lack of a built-in LISP editor. LISP is supposed to be a highly interactive language; you write a little piece of code, try it out, then write another piece of code. Without an editor that understands LISP syntax and formats the source program accordingly, LISP becomes hard to edit and, therefore, hard to write."

Functionality

On a microcomputer, TLC-LISP is intended for programmers who don't have the traditional academic background of LISP users. Things work as the documentation describes them, and the processor appears to contain few bugs. However, there are some rough edges, such as standard control keys that are not supported and error messages that are misspelled; but these do not seriously interfere with the usefulness of the language.

InfoWorld
Software Report Card

TLC-LISP
The LISP Company

	Unacceptable	*Poor*	*Fair*	*Good*	*Excellent*
Functionality					■
Documentation			■		
Ease of Use					■
Error Handling				■	
Support					

System Requirements

- 48K Cromemco system with CDOS monitor or
- Any 48K Z80 system with CP/M monitor

Price: ***$250*** for diskette & manual
$20 for manual alone

The LISP Company
Box 487
Redwood Estates, CA 59044

Execution speed is also reasonable; small programs compile and run instantaneously. Since TLC-LISP programs can also read and write CP/M files, programs in LISP communicate readily with programs in other languages.

TLC-LISP has some substantial shortcomings. First, it uses 16-bit integer arithmetic. The reason for this limitation is beyond my understanding; 16-bit numbers are insufficient for many kinds of problems. In a language like LISP, they save very little in program size or speed.

Equally dismal is the limit of 255 characters on string length. Again, it's a major restriction in some applications and saves little space.

Program size restrictions are inherent in microcomputer hardware. TLC-LISP occupies about 30K; on a 48K machine, it leaves about 10K for program and data. That's large enough for some uses, but LISP lends itself to many kinds of problems that come in larger packages, and 10K is likely to be limiting. TLC promises a version that will expand available memory, hardware permitting, to 32K for data plus 160K for programs. Versions for the 16-bit microcomputers are also promised.

Documentation

The 150-page manual has an index and is organized into three parts: introduction, examples, and language reference. The introduction is about 50 pages of theoretical background, most of which will seem like gobbledygook unless you already know LISP. The examples section shows how to write LISP functions and use the language. The treatment is much too cursory for a tutorial; 95% of the language is left for the reader to discover on his own.

The manual has a persistent problem: many functions and concepts are used before they are explained. However, the introduction refers you to a few books that TLC indicates will be more helpful. I recommend that any novice user first find a good book on LISP.

As a reference for someone who already knows LISP, the manual looks good. The language-reference section describes everything thoroughly and clearly, with many examples.

Ease of Use

The TLC-LISP disk contains LISP in object-code, some machine-readable documentation, and several LISP programs, which are both useful programmer's tools and samples of code. The program requires no installation. All that's necessary is to copy it off the disk.

Error Handling

The error-handling facility in TLC-LISP is basic, but adequate. When you do something wrong, you get a run-time message telling you what it is.

continued on page thirty-three

LISP

continued from page fourteen

LISP includes words to find out what function you were executing when an error occurred and what argument of the function you were evaluating. According to the manual, you can use these words to build your own error-handling routines and debuggers. I tried it and it didn't work. I couldn't determine whether the fault was mine or LISP's.

Support

Direct support from TLC is rather thin. My version of the manual contains no phone number; evidently TLC expects you to conduct all business by mail.

Because I happened to know TLC's phone number (it's the home phone of TLC-LISP's author), I tried calling up with some questions. I called at all hours of the day and night over about a week, but the phone was never answered—not even by a machine.

I did speak to people at TLC before beginning this review and I found them friendly and helpful. I assume that they would handle correspondence in the same spirit.

Summary

LISP is an effective tool for solving many kinds of problems that classical programming languages don't solve well. It's right for you if you want an easy way to process any amount of data whose elements interrelate in complex and changing ways.

LISP is not notably compact or efficient. If that bothers you, you should use a low-level language such as assembler or C. If you do, you'd better be prepared to spend months rather than days writing your program. Also, consider that you may save time by writing your program in LISP first, and then rewriting it in something else after you understand the problem thoroughly.

If you have a computer that can run TLC-LISP, it's an excellent way to start. It's not too expensive, it's powerful, and it's well written. ■

Fue utilizado en el campo de la inteligencia artificial, como se puede ver en este anuncio de 1985:

THE PROGRAMMER'S SHOP™

helps compare, evaluate, find products. Straight answers for serious programmers.

ARTIFICIAL INTELLIGENCE

EXSYS - Expert System building tool. Full RAM, Probability, Why, Intriguing, serious. PCDOS $295

GC LISP - "COMMON LISP", Help, tutorial, co-routines, compiled functions, thorough. PCDOS $475

IQ LISP - MACLISP & INTERLISP. Full RAM. Liked. PCDOS $155

TLC LISP - "LISP-machine"-like, all RAM, classes, turtle graphics 8087 for CP/M-86. MSDOS $235

TLC LOGO - fast, classes. CPM $ 95

EDITORS FOR PROGRAMMING

BRIEF Programmer's Editor - undo, windows, reconfigurable, macro programs, powerful. PCDOS $195

VEDIT - well liked, macros, buffers, CPM-80-86, MSDOS, PCDOS $119

Epsilon - like EMACS. PCDOS $195

FINAL WORD - for manuals 86/80 $215

PMATE - powerful 8086 $185

RECENT DISCOVERIES

SMALL TALK Looks good PCDOS $250

?? FAST, MASM - compatible assembler for MSDOS $195

C LANGUAGE

INSTANT C - Interactive development - Edit, Source Debug, run. Edit to Run - 3 Secs. MSDOS $495

MEGAMAX C - native Macintosh has fast compile, tight code, K&R, toolkit, .OBJ, DisASM MAC $295

CROSS COMPILERS by Lattice, CI. VAX to 8086. VMS $3000

MSDOS: C-86-8087, reliable call

Lattice 2.1 - improved call

Microsoft C 2.x $329

Mark Williams, debugger, fast call

40 + C Addons for Graph, Screen, ISAM, more. Get flyers, comparisons

C SHARP Realtime Toolkit - well supported, thorough, portable, objects, state sys. Source MANY $600

Call for a catalog, literature, and *solid value*

800-421-8006

THE PROGRAMMER'S SHOP™

128-I Rockland Street, Hanover, MA 02339

Visa Mass: 800-442-8070 or 617-826-7531 MasterCard

Softcon Specials

LEARN FOR LESS THAN $100.

Most of these should be available at our booth, #1748: or call between 3/31 and 4/3.

Introducing C by C.I. PCDOS $90

Janus ADA Jr. by RR MSDOS $95

Modula 2 MSDOS or MAC $90

PC Forth by Lab Micro MSDOS $89

Pocket APL by STSC PCDOS $90

Professional Basic by Morgan $85

Profiler-86 MSDOS $95

Prolog 86 - tutorial, samples $99

SBB Pascal Jr. - MSDOS $90

Snobol 4 + by Catspaw- MSDOS $90

Toolworks C/80 w/Math for CPM $75

Turbo Pascal $45

Ask about ASM, LISP, others!

FORTRAN LANGUAGE

RM/FORTRAN - Full '77, big arrays, 8087, debugging, xref. MSDOS $525

DR/Fortran-77 - full ANSI 77, 8087, overlay, full RAM, big arrays, complex NUMS., CPM86, MSDOS. $249

Note: All prices subject to change without notice. Mention this ad. Some prices are specials. **Show specials available 3/31 through 4/3 only.**

Ask about COD and **POs**. All formats available. 285-SC

32 March 18, 1985 InfoWorld

Actualmente, TLC-LISP ya no se comercializa más y, en consecuencia, para utilizar el software existente desarrollado en él, se desea construir en este trabajo práctico un intérprete que corra en la JVM (*Java Virtual Machine*). Por ello, el lenguaje elegido para su implementación es **Clojure**.

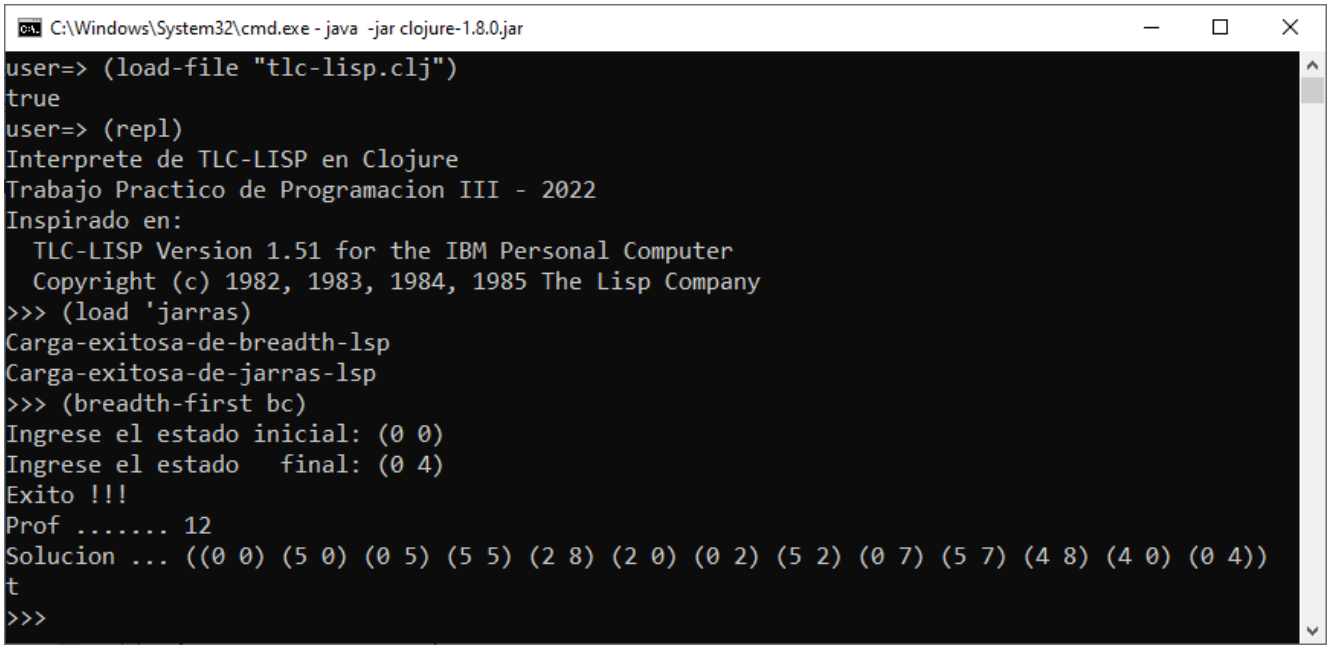
Deberá poder cargarse y correrse el siguiente **Sistema de Producción**, que resuelve el problema de obtener 4 litros de líquido utilizando dos jarras lisas (sin escala), una de 5 litros y otra de 8 litros:

jarras.lsp

```
(load 'breadth)

(de jarra5 (x) (first x))

(de jarra8 (x) (first (rest x)))

(setq bc '(
          (lambda (x) (if (lt (jarra5 x) 5) (list 5 (jarra8 x)) x))
          (lambda (x) (if (gt (jarra5 x) 0) (list 0 (jarra8 x)) x))
          (lambda (x) (if (ge (- 5 (jarra5 x)) (jarra8 x)) (list (+ (jarra5 x) (jarra8 x)) 0) x))
          (lambda (x) (if (lt (- 5 (jarra5 x)) (jarra8 x)) (list 5 (- (jarra8 x) (- 5 (jarra5 x)))) x))
          (lambda (x) (if (lt (jarra8 x) 8) (list (jarra5 x) 8) x))
          (lambda (x) (if (gt (jarra8 x) 0) (list (jarra5 x) 0) x))
          (lambda (x) (if (ge (- 8 (jarra8 x)) (jarra5 x)) (list 0 (+ (jarra8 x) (jarra5 x))) x))
          (lambda (x) (if (lt (- 8 (jarra8 x)) (jarra5 x)) (list (- (jarra5 x) (- 8 (jarra8 x))) 8) x))))

'Carga-exitosa-de-jarras-lsp
```

breadth.lsp

```
(de breadth-first (bc)
   (prin3 "Ingrese el estado inicial: ") (setq inicial (read))
   (prin3 "Ingrese el estado   final: ") (setq final (read))
   (cond ((equal inicial final) (prin3 "El problema ya esta resuelto !!!") (terpri) (breadth-first bc))
         (t (buscar bc final (list (list inicial)) nil))))

(de buscar (bc fin grafobusq estexp)
   (cond  ((null grafobusq) (fracaso))
          ((pertenece fin (first grafobusq)) (exito grafobusq))
          (t (buscar bc fin (append (rest grafobusq) (expandir (first grafobusq) bc estexp))
                                (if  (pertenece (first (first grafobusq)) estexp)
                                     estexp
                                     (cons (first (first grafobusq)) estexp))))))

(de expandir (linea basecon estexp)
   (if (or (null basecon) (pertenece (first linea) estexp))
      nil
      (if (not (equal ((eval (first basecon)) (first linea)) (first linea)))
          (cons (cons ((eval (first basecon)) (first linea)) linea) (expandir linea (rest basecon) estexp))
          (expandir linea (rest basecon) estexp))))

(de pertenece (x lista)
  (cond  ((null lista) nil)
         ((equal x (first lista)) t)
         (t (pertenece x (rest lista)))))

(de exito (grafobusq)
   (prin3 "Exito !!!") (terpri)
   (prin3 "Prof ..... ") (prin3 (- (length (first grafobusq)) 1)) (terpri)
   (prin3 "Solucion ... ") (prin3 (reverse (first grafobusq))) (terpri) t)

(de fracaso ()
   (prin3 "No existe solucion") (terpri) t)

'Carga-exitosa-de-breadth-lsp
```



Además, al cargar el archivo `demo.lsp` (provisto por la cátedra), se deberá obtener la siguiente salida por pantalla:

```
>>> (load 'demo)

*****************************************************
*                  TLC-LISP 2022                    *
* DEMO DE DEFINICION Y USO DE VARIABLES Y FUNCIONES *
*****************************************************

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

Type a letter or a digit and press Enter... 1

DEFINICION DE VARIABLES
-----------------------
> (setq u 'u)
u
> (setq v 'v)
v
> (setq w 'w)
w

LAS VARIABLES AHORA ESTAN EN EL AMBIENTE.
```

```
EVALUANDOLAS SE OBTIENEN SUS VALORES:
> u
u
> v
v
> w
w

Type a letter or a digit and press Enter... 1

UNA VEZ DEFINIDA UNA VARIABLE, CON SETQ TAMBIEN
SE LE PUEDE CAMBIAR EL VALOR:
> (setq n 0)
0
> (setq N 17)
17
> n
17

Type a letter or a digit and press Enter... 1

DEFINICION DE FUNCIONES
-----------------------
> (de sumar (a b) (add a b))
sumar
> (de restar (a b) (sub a b))
restar

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

Type a letter or a digit and press Enter... 1

TLC-LISP ES UN LENGUAJE DE AMBITO DINAMICO (DYNAMICALLY SCOPED):
> (setq x 1)
1
> (de g (y) (+ x y))
g
> (de f (x) (g 2))
f
> (f 5)
7
[En Scheme -lexically scoped- daria 3 en lugar de 7.]

Type a letter or a digit and press Enter... 1
```

```
APLICACION DE FUNCIONES ANONIMAS [LAMBDAS]
------------------------------------------

LAMBDA CON CUERPO SIMPLE:
> ((lambda (y) (+ 1 y)) 15)
16

LAMBDA CON CUERPO MULTIPLE:
> ((lambda (y) (prin3 'Hola!) (terpri) (+ 1 y)) 5)
Hola!
6

LAMBDA CON CUERPO MULTIPLE Y EFECTOS COLATERALES [SIDE EFFECTS]:
> ((lambda (a b c) (setq u a) (setq v b) (setq w c)) 1 2 3)
3

LOS NUEVOS VALORES DE LAS VARIABLES MODIFICADAS:
> u
1
> v
2
> w
3

Type a letter or a digit and press Enter... 1

APLICACION PARCIAL:
> (((lambda (x) (lambda (y) (- x y))) 8) 3)
-2

EL MISMO EJEMPLO ANTERIOR, AHORA DEFINIENDO UNA FUNCION:
> (setq p (lambda (x) (lambda (y) (- x y))))
(lambda (x) (lambda (y) (- x y)))
> (p 8)
(lambda (y) (- x y))
> ((p 8) 3)
-2

Type a letter or a digit and press Enter... 1

DEFINICION DE FUNCIONES RECURSIVAS [RECORRIDO LINEAL]
-----------------------------------------------------

FUNCION RECURSIVA CON EFECTO COLATERAL
[DEJA EN LA VARIABLE D LA CANTIDAD DE PARES]:
> (de recorrer (L)
    (recorrer2 L 0))
recorrer
> (setq D 0)
0
```

```
> (de recorrer2 (L i)
    (cond
      ((null (rest L)) (setq D (+ 1 D)) (list (first L) i))
      (t (prin3 (list (first L) i)) (setq D (+ i 1)) (terpri) (recorrer2 (rest L) D))))
recorrer2
> (recorrer '(x y z))
(x 0)
(y 1)
(z 2)
> d
3

Type a letter or a digit and press Enter... 1

DEFINICION DE FUNCIONES RECURSIVAS [RECORRIDO A TODO NIVEL]
-----------------------------------------------------------

EXISTENCIA DE UN ELEMENTO ESCALAR EN UNA LISTA:
> (DE EXISTE (A L)
    (COND
      ((NULL L) NIL)
      ((NOT (LISTP (FIRST L))) (OR (EQUAL A (FIRST L)) (EXISTE A (REST L))))
      (T (OR (EXISTE A (FIRST L)) (EXISTE A (REST L))))))
EXISTE
> (existe 'c '(a ((b) ((d c) a) e f)))
t
> (existe 'g '(a ((b) ((d c) a) e f)))
nil

Type a letter or a digit and press Enter... 1

ELIMINACION DE UN ELEMENTO DE UNA LISTA:
> (de eliminar (dat li)
    (cond
      ((null li) li)
      ((equal dat (first li)) (eliminar dat (rest li)))
      ((listp (first li)) (cons (eliminar dat (first li)) (eliminar dat (rest li))))
      (T (cons (first li) (eliminar dat (rest li))))))
eliminar
> (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))

Type a letter or a digit and press Enter... 1

PROFUNDIDAD DE UNA LISTA:
> (de profundidad (lista)
    (if (or (not (listp lista)) (null lista)) 0
        (if (gt (+ 1 (profundidad (first lista))) (profundidad (rest lista)))
            (+ 1 (profundidad (first lista)))
            (profundidad (rest lista)))))
profundidad
```

```
> (profundidad '((2 3)(3 ((7))) 5))
4
[El valor esperado es 4.]

Type a letter or a digit and press Enter... 1

PLANCHADO DE UNA LISTA:
> (de planchar (li)
    (cond
      ((null li) ())
      ((listp (first li)) (append (planchar (first li)) (planchar (rest li))))
      (T (cons (first li) (planchar (rest li))))))
planchar
> (planchar '((2 3)(3 ((7))) 5))
(2 3 3 7 5)

Type a letter or a digit and press Enter... 1

DEFINICION DE FUNCIONES PARA OCULTAR LA RECURSIVIDAD
----------------------------------------------------

FILTRAR [SELECCIONA LOS ELEMENTOS QUE CUMPLAN UNA CONDICION DADA]:
> (de FILTRAR (F L)
    (COND
      ((null L) ())
      ((F (first L)) (CONS (first L) (FILTRAR F (rest L))))
      (T (FILTRAR F (rest L)))))
FILTRAR
> (filtrar (lambda (x) (gt x 0)) '(5 0 2 -1 4 6 0 8))
(5 2 4 6 8)

Type a letter or a digit and press Enter... 1

REDUCIR [REDUCE UNA LISTA APLICANDO DE A PARES UNA FUNCION DADA]:
> (de REDUCIR (F L)
    (IF (null (rest L))
        (first L)
        (F (first L) (REDUCIR F (rest L)))))
REDUCIR
> (reducir (lambda (x y) (if (gt x 0) (cons x y) y)) '(5 0 2 -1 4 6 0 8 ()))
(5 2 4 6 8)

Type a letter or a digit and press Enter... 1

MAPEAR [APLICA A CADA ELEMENTO DE UNA LISTA UNA FUNCION DADA]:
> (de MAPEAR (OP L)
    (IF (null L)
        ()
        (CONS (OP (first L)) (MAPEAR OP (rest L)))))
MAPEAR
> (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)
```

```
Type a letter or a digit and press Enter... 1


TRANSPONER [TRANSPONE UNA LISTA DE LISTAS]:
> (de TRANSPONER (M)
    (IF (null (first M))
        ()
        (CONS (MAPEAR first M) (TRANSPONER (MAPEAR rest M)))))
TRANSPONER
> (transponer '((a b c) (d e f) (g h i)))
((a d g) (b e h) (c f i))

Type a letter or a digit and press Enter... 1


IOTA [RETORNA UNA LISTA CON LOS PRIMEROS N NUMEROS NATURALES]:
> (de IOTA (N)
    (IF (LT N 1)
         ()
         (AUXIOTA 1 N)))
IOTA
> (de AUXIOTA (I N)
    (IF (EQUAL I N)
        (LIST N)
        (CONS I (AUXIOTA (+ I 1) N))))
AUXIOTA
> (IOTA 10)
(1 2 3 4 5 6 7 8 9 10)

Type a letter or a digit and press Enter... 1


FUNCIONES IMPLEMENTADAS USANDO LAS FUNCIONES ANTERIORES
------------------------------------------------------


SUMATORIA DE LOS PRIMEROS N NUMEROS NATURALES:
> (de sumatoria (n) (reducir + (iota n)))
sumatoria
> (sumatoria 10)
55
[El valor esperado es 55.]

Type a letter or a digit and press Enter... 1


ELIMINACION DE LOS ELEMENTOS REPETIDOS EN UNA LISTA SIMPLE:
> (de eliminar-repetidos (li)
    (reverse (reducir (lambda (x y) (if (existe x y) y (cons x y))) (reverse (cons () li)))))
eliminar-repetidos
> (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)

Type a letter or a digit and press Enter... 1
```

```
SELECCION DEL ENESIMO ELEMENTO DE UNA LISTA DADA:
> (de seleccionar (n li)
    (if (or (lt n 1) (gt n (length li)))
        ()
        (first (first (filtrar (lambda (x) (equal n (first (rest x)))) (transponer (list li (iota (length li)))))))))
seleccionar
> (SELECCIONAR 5 '(A B C D E F G H I J))
E

Type a letter or a digit and press Enter... 1

APLICACION DE TODAS LAS FUNCIONES DE UNA LISTA A UN ELEMENTO DADO:
> (de aplicar-todas (lf x)
    (mapear (lambda (f) (f x)) lf))
aplicar-todas
> (aplicar-todas (list length rest first) '((3 2 1)(9 8)(7 6)(5 4)))
(4 ((9 8) (7 6) (5 4)) (3 2 1))

Type a letter or a digit and press Enter... 1

ENTRADA DE DATOS Y SALIDA DEL INTERPRETE
---------------------------------------

CARGA DE DATOS DESDE LA TERMINAL/CONSOLA:
> (setq R 0)
> (de cargarR)
    (prin3 '->R: )(setq R (read))(prin3 'R*2: )(prin3 (+ R R))(terpri))
> (cargarR)
->R: 7
R*2: 14

PARA VER EL AMBIENTE [NO FUNCIONA EN TLC-LISP]: (env)
PARA SALIR DEL INTERPRETE: (exit)
Carga-exitosa-de-demo-lsp
>>>
```

El funcionamiento del intérprete en Clojure deberá imitar al TLC-LISP original, que puede correrse en un emulador de MS-DOS (por ejemplo, DOSBox).

En DOSBox, es necesario montar la ubicación donde esté grabado TLC-LISP. Por ejemplo:

En Windows, si la carpeta de TLC-LISP está en la raíz de D:

```
mount c d:\tlc-lisp
```

En Linux o Mac:

```
mount c ~/
```

Luego hay que cambiar a C: y ejecutar LISP

```
c:
lisp
```

OBS.: Si el teclado español no está bien configurado en DOSBox, ejecutar:

```
keyb sp
```

Características de TLC-LISP a implementar en el intérprete*

Valores de verdad

`nil`: significa *falso* y lista vacía
`t`: significa *verdadero*

Formas especiales y macros

`cond`: macro (evalúa múltiples condiciones)
`de`: macro (define función y la liga a símbolo)
`eval`: evalúa una lista
`exit`: sale del intérprete
`if`: forma especial (evalúa una condición)
`lambda`: macro (define una func. anónima)
`load`: carga un archivo
`or`: macro (evalúa mientras obtenga `nil`)
`quote`: forma especial (impide evaluación)
`setq`: forma especial (liga símbolo a valor)

(*) **env** no está disponible así en TLC-LISP

Funciones

`add`: retorna la suma de los argumentos
`append`: retorna la fusión de dos listas
`cons`: retorna inserción de elem. en cabeza de lista
`env`: retorna el ambiente
`equal`: retorna `t` si dos elementos son iguales
`first`: retorna la 1ra. posición de una lista
`ge`: retorna `t` si el 1° núm. es mayor o igual que 2°
`gt`: retorna `t` si el 1° núm. es mayor que el 2°
`length`: retorna la longitud de una lista
`list`: retorna una lista formada por los args.
`listp`: retorna `t` si un elemento es una lista
`lt`: retorna `t` si el 1° núm. es menor que el 2°
`not`: retorna la negación de un valor de verdad
`null`: retorna `t` si un elemento es `nil`
`prin3`: imprime un elemento y lo retorna
`read`: retorna la lectura de un elemento
`rest`: retorna una lista sin su 1ra. posición
`reverse`: retorna una lista invertida
`sub`: retorna la resta de los argumentos
`terpri`: imprime un salto de línea y retorna `nil`
`+`: equivale a `add`
`-`: equivale a `sub`

A los efectos de desarrollar el intérprete solicitado, se deberá partir de los siguientes materiales proporcionados por la cátedra:

- 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 TLC-LISP sin terminar, para completarlo. El mismo contiene dos funciones que deben ser terminadas y 23 funciones que deben desarrollarse por completo.
- Archivos `jarras.lsp`, `breadth.lsp` y `demo.lsp` 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 el trabajo práctico, se deberá entregar un **proyecto** compuesto por el código fuente del intérprete de TLC-LISP 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).