Prof. Carlos E. Cimino (3.601)
Prof. Dr. Diego Corsi (3.602)
Prof. Gastón Larriera (3.603)

Trabajo Práctico

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.

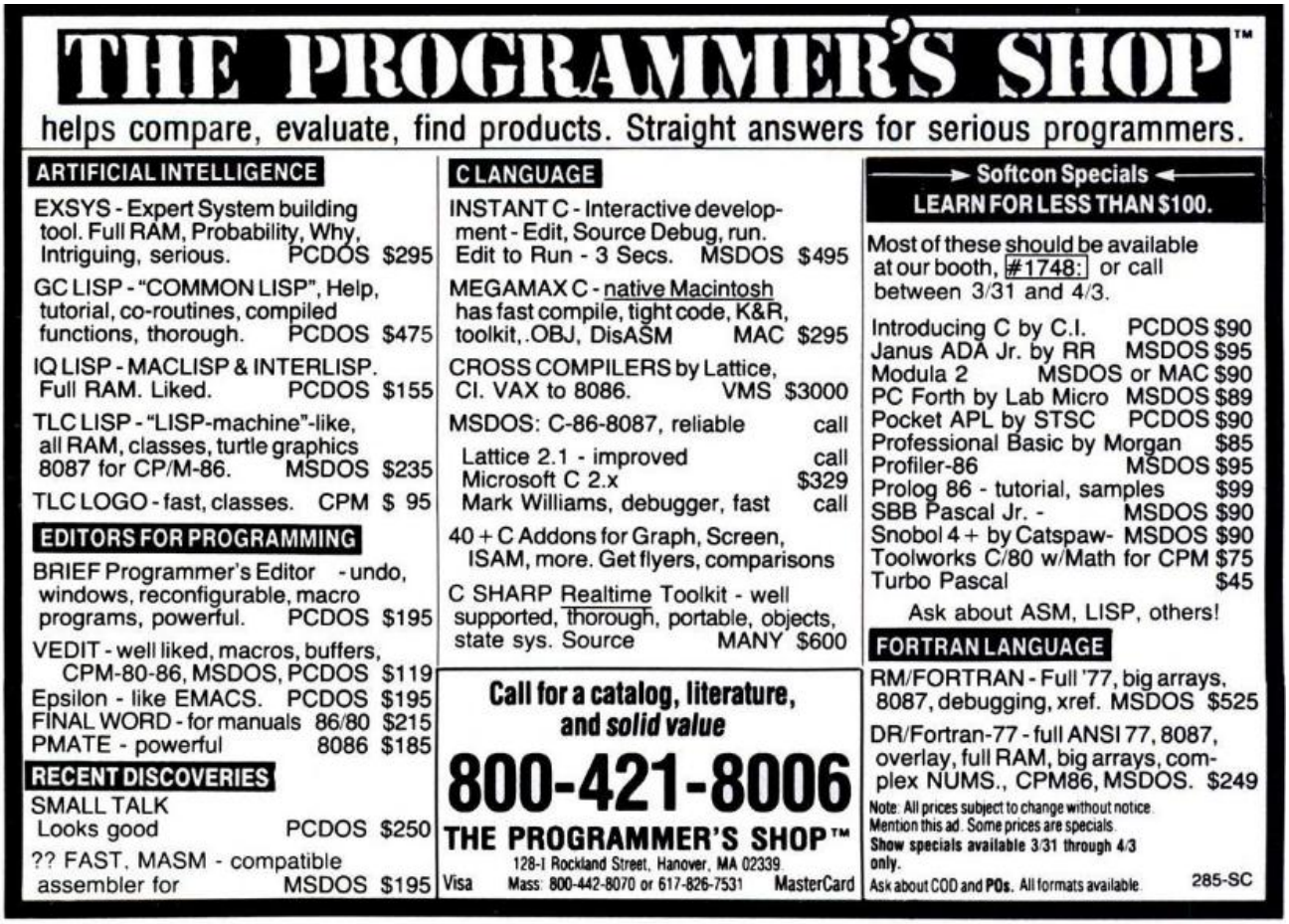
Summary

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 un intérprete que corra en la JVM. Por ello, el lenguaje elegido para su implementación es **Clojure**.

Prof. Carlos E. Cimino (3.601)
Prof. Dr. Diego Corsi (3.602)
Prof. Gastón Larriera (3.603)

Deberá poder cargarse y correrse el siguiente **Sistema de Producción**:

jarras.lsp

```
(load 'breadth)

(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))
         )
)

(de jarra5 (x) (first x))

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

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)
```

Así como también el siguiente ejemplo:

ejemplo.lsp

```
(de sumar (a b) (+ a b))

(de restar (a b) (- a b))

(setq x 1)

(setq y 2)

(sumar 3 6)

(restar 4 5)

y

(de C (LF X)
   (if (null LF)
       nil
       (cons ((first LF) X) (C (rest LF) X))
   )
)

(C (list first rest list) '((1 2 3)(4 5 6)(7 8 9)))

(de cargarR ()
  (prin3 "R: ")(setq R (read))(prin3 "R * 2: ")(prin3 (+ R R))(terpri))

(de recorrer (L)
  (recorrer2 L 0))

(DE recorrer2 (L i)
  (COND
    ((NULL (rest L)) (list (first L) i))
    (T (prin3 (list (first L) i))(setq D (+ i 1)) (terpri)(recorrer2 (REST L) D))))

(de compa (a b)
  (if (equal a b) (setq m 5) (exit)))
```

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
```

La función principal del intérprete (**repl**) deberá estar implementada de la siguiente manera:

tlc-lisp.clj

```
(declare evaluar)
(declare aplicar)
(declare controlar-aridad)
(declare igual?)
(declare cargar-arch)
(declare imprimir)
(declare actualizar-amb)
(declare revisar-f)
(declare revisar-lae)
(declare buscar)
(declare evaluar-cond)
(declare evaluar-secuencia-en-cond)

; REPL (read–eval–print loop).
; Aridad 0: Muestra mensaje de bienvenida y se llama recursivamente con el ambiente inicial.
; Aridad 1: Muestra >>> y lee una expresion y la evalua
; Si la 2da. posicion del resultado es nil, retorna true (caso base de la recursividad).
; Si no, imprime la 1ra. pos. del resultado y se llama recursivamente con la 2da. pos. del resultado.
defn repl
  ([]
    (println "Interprete de TLC-LISP en Clojure")
    (println "Trabajo Practico de Programacion III")
    (println "Inspirado en:")
    (println "  TLC-LISP Version 1.51 for the IBM Personal Computer")
    (println "  Copyright (c) 1982, 1983, 1984, 1985 The Lisp Company") (flush)
    (repl '( add add append append cond cond cons cons de de env env equal equal eval eval exit exit
             first first ge ge gt gt if if lambda lambda length length list list load load lt lt nil nil not not
             null null or or prin3 prin3 quote quote read read rest rest reverse reverse setq setq sub sub
             t t terpri terpri + add - sub)))
  ([amb]
    (print ">>> ") (flush)
    (try (let  [res (evaluar (read) amb nil)]
              (if  (nil? (fnext res))
                   true
                   (do (imprimir (first res)) (repl (fnext res)))))
         (catch Exception e (println) (print "*error* ") (println (get (Throwable->map e) :cause)) (repl amb))))
)
```

Las 12 funciones restantes se deberán implementar completando las siguientes descripciones:

tlc-lisp.clj (cont.)

```
; Evalua una expresion usando los ambientes global y local. Siempre retorna una lista con un resultado y un ambiente.
; Si la evaluacion falla, el resultado es una lista con '*error* como primer elemento, por ejemplo: (list '*error* 'too-many-args) y el ambiente es el ambiente global.
; Si la expresion es un escalar numero o cadena, retorna la expresion y el ambiente global.
; Si la expresion es otro tipo de escalar, la busca (en los ambientes local y global) y retorna el valor y el ambiente global.
; Si la expresion es una secuencia nula, retorna nil y el ambiente global.
; Si el primer elemento de la expresion es '*error*, retorna la expresion y el ambiente global.
; Si el primer elemento de la expresion es una forma especial o una macro, valida los demas elementos y retorna el resultado y el (nuevo?) ambiente.
; Si no lo es, se trata de una funcion en posicion de operador (es una aplicacion de calculo lambda), por lo que se llama a la funcion aplicar,
; pasandole 4 argumentos: la evaluacion del primer elemento, una lista con las evaluaciones de los demas, el ambiente global y el ambiente local.
(defn evaluar [expre amb-global amb-local]  ■■■■■  )

; Aplica una funcion a una lista de argumentos evaluados, usando los ambientes global y local. Siempre retorna una lista con un resultado y un ambiente.
; Si la aplicacion falla, el resultado es una lista con '*error* como primer elemento, por ejemplo: (list '*error* 'arg-wrong-type) y el ambiente es el ambiente global.
; Aridad 4: Recibe la func., la lista de args. evaluados y los ambs. global y local. Se llama recursivamente agregando 2 args.: la func. revisada y la lista de args. revisada.
; Aridad 6: Si la funcion revisada no es nil, se la retorna con el amb. global.
; Si la lista de args. evaluados revisada no es nil, se la retorna con el amb. global.
; Si no, en caso de que la func. sea escalar (predefinida o definida por el usuario), se devuelven el resultado de su aplicacion (controlando la aridad) y el ambiente global.
; Si la func. no es escalar, se valida que la cantidad de parametros y argumentos coincidan, y:
; en caso de que se trate de una func. lambda con un solo cuerpo, se la evalua usando el amb. global intacto y el local actualizado con los params. ligados a los args.,
; en caso de haber multiples cuerpos, se llama a aplicar recursivamente, pasando la funcion lambda sin el primer cuerpo, la lista de argumentos evaluados,
; el amb. global actualizado con la eval. del 1er. cuerpo (usando el amb. global intacto y el local actualizado con los params. ligados a los args.) y el amb. local intacto.
(defn aplicar
  ([f lae amb-global amb-local]  ■■■■■  )
  ([resu1 resu2 f lae amb-global amb-local]  ■■■■■  )
)

; Controla la aridad (cantidad de argumentos de una funcion).
; Recibe una lista y un numero. Si la longitud de la lista coincide con el numero, retorna el numero.
; Si es menor, retorna (list '*error* 'too-few-args).
; Si es mayor, retorna (list '*error* 'too-many-args).
(defn controlar-aridad [lis val-esperado]  ■■■■■  )

; Verifica la igualdad de dos simbolos.
; Recibe dos simbolos a y b. Retorna true si se deben considerar iguales; si no, false.
; Se utiliza porque TLC-LISP no es case-sensitive y ademas no distingue entre nil y la lista vacia.
(defn igual? [a b]  ■■■■■  )

; Carga el contenido de un archivo.
; Aridad 3: Recibe los ambientes global y local y el nombre de un archivo
; (literal como string o atomo, con o sin extension .lsp, o el simbolo ligado al nombre de un archivo en el ambiente), abre el archivo
; y lee un elemento de la entrada (si falla, imprime nil), lo evalua y llama recursivamente con el (nuevo?) amb., nil, la entrada y un arg. mas: el resultado de la evaluacion.
; Aridad 4: lee un elem. del archivo (si falla, imprime el ultimo resultado), lo evalua y llama recursivamente con el (nuevo?) amb., nil, la entrada y el resultado de la eval.
(defn cargar-arch
  ([amb-global amb-local arch]
    (let [nomb (first (evaluar arch amb-global amb-local))]
      (if (and (seq? nomb) (igual? (first nomb) '*error*))
        (do (imprimir nomb) amb-global)
        (let [nm (clojure.string/lower-case (str nomb)),
              nom (if (and (> (count nm) 4)(clojure.string/ends-with? nm ".lsp")) nm (str nm ".lsp")),
              ret (try (with-open [in (java.io.PushbackReader. (clojure.java.io/reader nom))]
                         (binding [*read-eval* false] (try (let [res (evaluar (read in) amb-global nil)]
                                                             (cargar-arch (fnext res) nil in res))
                                                           (catch Exception e (imprimir nil) amb-global))))
                       (catch java.io.FileNotFoundException e (imprimir (list '*error* 'file-open-error 'file-not-found nom '1 'READ)) amb-global))]
          ret))))
  ([amb-global amb-local in res]
    (try (let [res (evaluar (read in) amb-global nil)] (cargar-arch (fnext res) nil in res))
         (catch Exception e (imprimir (first res)) amb-global)))
)
```

tlc-lisp.clj (cont.)

```
; Imprime, con salto de linea, atomos o listas en formato estandar (las cadenas con comillas) y devuelve su valor. Muestra errores sin parentesis.
; Aridad 1: Si recibe un escalar, lo imprime con salto de linea en formato estandar (pero si es \space no lo imprime), purga la salida y devuelve el escalar.
; Si recibe una secuencia cuyo primer elemento es '*error*, se llama recursivamente con dos argumentos iguales: la secuencia recibida.
; Si no, imprime lo recibido con salto de linea en formato estandar, purga la salida y devuelve la cadena.
; Aridad 2: Si el primer parametro es nil, imprime un salto de linea, purga la salida y devuelve el segundo parametro.
; Si no, imprime su primer elemento en formato estandar, imprime un espacio y se llama recursivamente con la cola del primer parametro y el segundo intacto.
(defn imprimir
  ([elem]  ■■■■■  )
  ([lis orig]  ■■■■■  )
)

; Actualiza un ambiente (una lista con claves en las posiciones impares [1, 3, 5...] y valores en las pares [2, 4, 6...]
; Recibe el ambiente, la clave y el valor.
; Si el valor no es escalar y en su primera posicion contiene '*error*, retorna el ambiente intacto.
; Si no, coloca la clave y el valor en el ambiente (puede ser un alta o una actualizacion) y lo retorna.
(defn actualizar-amb [amb-global clave valor]  ■■■■■  )

; Revisa una lista que representa una funcion.
; Recibe la lista y, si esta comienza con '*error*, la retorna. Si no, retorna nil.
(defn revisar-f [lis]  ■■■■■  )

; Revisa una lista de argumentos evaluados.
; Recibe la lista y, si esta contiene alguna sublista que comienza con '*error*, retorna esa sublista. Si no, retorna nil.
(defn revisar-lae [lis]  ■■■■■  )

; Busca una clave en un ambiente (una lista con claves en las posiciones impares [1, 3, 5...] y valores en las pares [2, 4, 6...] y retorna el valor asociado.
; Si no la encuentra, retorna una lista con '*error* en la 1ra. pos., 'unbound-symbol en la 2da. y el elemento en la 3ra.
(defn buscar [elem lis]  ■■■■■  )

; Evalua el cuerpo de una macro COND. Siempre retorna una lista con un resultado y un ambiente.
; Recibe una lista de sublistas (cada una de las cuales tiene una condicion en su 1ra. posicion) y los ambientes global y local.
; Si la lista es nil, el resultado es nil y el ambiente retornado es el global.
; Si no, evalua (con evaluar) la cabeza de la 1ra. sublista y, si el resultado no es nil, retorna el res. de invocar a evaluar-secuencia-en-cond con la cola de esa sublista.
; En caso contrario, sigue con las demas sublistas.
(defn evaluar-cond [lis amb-global amb-local]  ■■■■■ )

; Evalua (con evaluar) secuencialmente las sublistas de una lista y retorna el valor de la ultima evaluacion.
(defn evaluar-secuencia-en-cond [lis amb-global amb-local]  ■■■■■ )

; Al terminar de cargar el archivo, se retorna true.
■■■■■
```

Algunos ejemplos de uso de las funciones anteriores:

repl

```
user=> (load-file "tlc-lisp.clj")
true
user=> (repl)
Interprete de TLC-LISP en Clojure
Trabajo Practico de Programacion III
Inspirado en:
  TLC-LISP Version 1.51 for the IBM Personal Computer
  Copyright (c) 1982, 1983, 1984, 1985 The Lisp Company
>>> _
```

evaluar

```
user=> (evaluar '(setq r 3) '(+ add) nil)
(3 (+ add r 3))

user=> (evaluar '(de doble (x) (+ x x)) '(+ add) nil)
(doble (+ add doble (lambda (x) (+ x x))))

user=> (evaluar '(+ 2 3) '(+ add) nil)
(5 (+ add))

user=> (evaluar '(+ 2 3) '(add add) nil)
((*error* unbound-symbol +) (add add))

user=> (evaluar '(doble 3) '(+ add doble (lambda (x) (+ x x))) nil)
(6 (+ add doble (lambda (x) (+ x x))))

user=> (evaluar '(doble r) '(+ add r 4 doble (lambda (x) (+ x x))) nil)
(8 (+ add r 4 doble (lambda (x) (+ x x))))

user=> (evaluar '((lambda (x) (+ x x)) 3) '(+ add) nil)
(6 (+ add))
```

aplicar

```
user=> (aplicar 'cons '(a (b)) '(cons cons) nil)
((a b) (cons cons))

user=> (aplicar 'add '(4 5) '(+ add r 5) nil)
(9 (+ add r 5))

user=> (evaluar '(doble r) '(+ add r 4 doble (lambda (x) (+ x x))) nil)
(8 (+ add r 4 doble (lambda (x) (+ x x))))

user=> (aplicar '(lambda (x) (+ x x)) '(4) '(+ add r 4 doble (lambda (x) (+ x x))) nil)
(8 (+ add r 4 doble (lambda (x) (+ x x))))
```

controlar-aridad

```
user=> (controlar-aridad '(a b c) 4)
(*error* too-few-args)

user=> (controlar-aridad '(a b c d) 4)
4

user=> (controlar-aridad '(a b c d e) 4)
(*error* too-many-args)
```

igual?

```
user=> (igual? nil 'NIL)          user=> (= nil 'NIL)
true                              false

user=> (igual? nil "NIL")         user=> (= nil "NIL")
true                              false

user=> (igual? nil ())            user=> (= nil ())
true                              false

user=> (igual? () 'NIL)           user=> (= () 'NIL)
true                              false
```

imprimir

```
user=> (imprimir "hola")
"hola"
"hola"

user=> (imprimir 5)
5
5

user=> (imprimir 'a)
a
a

user=> (imprimir \space)
\space

user=> (imprimir '(hola "mundo"))
(hola "mundo")
(hola "mundo")

user=> (imprimir '(*error* hola "mundo"))
*error* hola "mundo"
(*error* hola "mundo")
```

actualizar-amb

```
user=> (actualizar-amb '(+ add - sub) 'x 1)
(+ add - sub x 1)

user=> (actualizar-amb '(+ add - sub x 1 y 2) 'x 3)
(+ add - sub x 3 y 2)
```

revisar-f

```
user=> (revisar-f 'doble)
nil

user=> (revisar-f '(*error* too-few-args))
(*error* too-few-args)
```

revisar-lae

```
user=> (revisar-lae '(1 add first))
nil

user=> (revisar-lae '(1 add (*error* too-many-args) first))
(*error* too-many-args)
```

buscar

```
user=> (buscar '- '(+ add - sub))
sub

user=> (buscar 'doble '(+ add - sub))
(*error* unbound-symbol doble)
```

evaluar-cond

```
user=> (evaluar-cond nil '(equal equal setq setq) nil)
(nil (equal equal setq setq))

user=> (evaluar-cond '(((equal 'a 'b) (setq x 1))) '(equal equal first first) nil)
(nil (equal equal first first))

user=> (evaluar-cond '(((equal 'a 'b) (setq x 1)) ((equal 'a 'a) (setq y 2)))
'(equal equal setq setq) nil)
(2 (equal equal setq setq y 2))

user=> (evaluar-cond '(((equal 'a 'b) (setq x 1)) ((equal 'a 'a) (setq y 2)) (setq z 3)))
'(equal equal setq setq) nil)
(3 (equal equal setq setq y 2 z 3))
```

evaluar-secuencia-en-cond

```
user=> (evaluar-secuencia-en-cond '((setq y 2)) '(setq setq) nil)
(2 (setq setq y 2))

user=> (evaluar-secuencia-en-cond '((setq y 2) (setq z 3)) '(setq setq) nil)
(3 (setq setq y 2 z 3))
```

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)
exit: sale del intérprete
if: forma especial (evalúa una condición)
lambda: macro (define una func. anónima)
load: carga un archivo
or: macro (evalúa mientras no obtenga t)
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
eval: retorna la evaluación de una lista
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.
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