Ingeniería Investigación y Tecnología, ISSN en trámite
VI. 3. 177-186, 2005 (artículo arbitrado)

Sobre el uso de *Logo* en inteligencia artificial

M.A. Murray-Lasso
Unidad de Enseñanza Auxiliada por Computadora
Departamento de Ingeniería de Sistemas. División de Ingeniería Mecánica e Industrial
Facultad de Ingeniería, UNAM
E-mail: mamurray@servidor.unam.mx

(recibido: agosto de 2004; aceptado: abril de 2005)

Resumen

Por ser un lenguaje derivado de LISP, el lenguaje *Logo*, no obstante de tener fama de ser un lenguaje para niños, es un lenguaje adecuado para escribir programas de inteligencia artificial. El artículo se propone establecer lo anterior. En vez de hacerlo escribiendo un proyecto real de inteligencia artificial implementando en *Logo*, lo que llevaría a un artículo muy largo, se opta por tomar de la literatura de inteligencia artificial un par de ejemplos cortos elaborados en LISP, transcribirlos a *Logo* y hacer comparaciones entre los programas en ambos lenguajes. Se espera con esto resaltar la similitud entre *Logo* y LISP. Debido a la gran variedad de versiones de LISP existentes, y a que muchas versiones tienen implementadas características foráneas al origen y filosofía del lenguaje, se escogió una versión "pura" de LISP para hacer la comparación.

Descriptores: Logo, LISP, inteligencia artificial, programa, lenguaje de programación.

Abstract

Since the Logo language was derived from LISP, the classical language of artificial intelligence, Logo, in spite of its reputation as a language for children, is adequate for writing artificial intelligence programs. The purpose of the article is to establish this fact. Instead of doing it by exhibiting a real artificial intelligence project implemented in Logo, which would result in a long paper, we choose to select from the artificial intelligence literature a pair of short examples implemented in LISP, transcribe them to Logo, and compare the programs in both languages. It is hoped that this will highlight the similarity between Logo and Lisp. Due to the great variety of avaible LISP versions, and to the fact that many versions have implemented characteristicas foreign to the origin and philosophy of the language, a "pure" version of LISP has been selected to carry out the comparison.

Keywords: *Logo, LISP, artificial intelligence, program, programming language.*

Introducción

El Lenguaje *Logo* es un lenguaje diseñado para la educación a todos los niveles, el cual ha adquirido la reputación de ser un lenguaje para niños con el que se pueden hacer dibujos interesantes y aprender conceptos geométricos. En realidad, se trata de un lenguaje completo, al igual que otros lenguajes como *FORTRAN, BASIC, C o Pascal,* aunque con diferente estilo y propósitos, que en algunas versiones llega a tener del orden de 700 o más instrucciones diferentes y que en algunas versiones tiene un enfoque orientado a objetos y facilidades capaces de manejar multimedios. El lenguaje fue desarrollado comenzando alrededor de 1970, entre el Laboratorio de Inteligencia Artificial del MIT y la empresa Bolt, Beranek & Newman. Al principio, el lenguaje corría en una DEC PDP-10 y tenía una tortuga robot que se desplazaba en el piso, recibiendo órdenes de avanzar,

retroceder, girar a la derecha o a la izquierda, subir o bajar una pluma con la cual dibujaba sobre un papel colocado en el piso al desplazarse. Con la aparición de las microcomputadoras, la tortuga robot fue sustituida por una tortuga dibujada en la pantalla de la computadora y se pudieron agregar algunas instrucciones como ***MUESTRATORTUGA*** y ***ESCONDETORTUGA*** que hacen que la tortuga esté o no visible para evitar que estorbe la visión del dibujo en la pantalla. En algunas versiones de *Logo* se pueden manejar varias tortugas, y en las más poderosas, la cantidad es esencialmente ilimitada (restringida solamente por la capacidad de memoria de la máquina). Aunque el lenguaje ha adquirido su mayor popularidad con grupos de estudiantes a nivel primaria y secundaria, en algunos lugares como MIT se le utiliza a nivel licenciatura y posgrado, debido a que además de poder ser útil en la enseñanza de la geometría elemental, la tortuga y sus habilidades pueden también orientar en conceptos más avanzados como Geometría de Riemann, Teoría de Relatividad, Dinámica de partículas y de cuerpos sólidos, simulación, robótica y animación, por mencionar sólo algunos. Por otra parte, como tiene instrucciones para manejar listas ligadas con las que se pueden representar estructuras de datos como árboles y redes, y tiene operadores aritméticos, lógicos y relacionales, así como instrucciones especiales para manejo de textos largos y de música, se puede utilizar para la enseñanza de muchos otros temas incluyendo gramática, literatura, historia, ciencias sociales, química, matemáticas, música y computación.

Debido a que *Logo* es descendiente de *LISP*, uno de los lenguajes preferidos de los investigadores en inteligencia artificial, tiene muchas características en común con él. Por esta razón, en este artículo trataremos de exhibir con base en ejemplos, que *Logo* es un lenguaje adecuado para llevar a cabo estudios e investigaciones en inteligencia artificial. Lo que haremos será exhibir una serie de programas en *Logo* con comentarios que hacen la misma labor que otros programas escritos en *LISP* tomados de fuentes bibliográficas del campo de la inteligencia artificial. Dado que existen muchas versiones de *LISP* y de *Logo, se* han tomado como referencias *muLisp* y *LogoWriter,* por tener ambas versiones disponibles para probar los ejemplos.

El muLISP "Puro"

Los lenguajes que se utilizan en la práctica pueden tener manuales de referencia que ocupan varios cientos de páginas, razón por la cual no se pueden reseñar en un artículo corto como este. Muchas de las instrucciones no son fundamentales y se tienen disponibles para comodidad de los programadores, evitándoles o simplificándoles su trabajo. Por ejemplo, si se quieren intercambiar los valores de dos variables, basta tener la instrucción de asignación para que con tres instrucciones se haga la operación. Es decir, para intercambiar los valores de las variables ***A*** y ***B*** se puede escribir: ***T:= A, A:=B, B:=T.*** Lo anterior se simplifica si se cuenta con la función ***SWAP (A,B)***, que en una sola instrucción logra el mismo resultado y ahorra la introducción de la variable temporal auxiliar ***T.***. Si nos concentramos en las instrucciones realmente elementales, se podrá establecer el parecido y la equivalencia entre *LISP* y *Logo* en un espacio relativamente corto. La versión más recortada de *muLISP* conocida como *muLISP Puro* (The Software House, 1984) reconoce como primitivas fundamentales o básicas, a las instrucciones ***CAR, CDR, CONS, EQ,*** y ***ATOM.*** Tanto *LISP* como *Logo* manejan como principal estructura de datos la *lista,* la cual se representa como un renglón de elementos encerrados entre paréntesis ( ) para *LISP,* o entre corchetes [ ] para *Logo.* Los elementos de una lista pueden ser otras listas, las cuales tienen el mismo tipo de representación. El elemento con estructura más simple de LISP es un *átomo*, concepto que corresponde al de *nombre* en *Logo.* (En *Logo,* los números también son a la vez nombres).

La función **CAR** de *LISP* toma una lista como argumento que le extrae y produce como salida el primer elemento de la misma, cualquiera que sea su naturaleza. La correspondiente función en *Logo* se llama **PRIMERO** que se abrevia **PR** y hace exactamente la misma función. Cabe hacer aquí la aclaración que el lenguaje *Logo,* por su propósito educativo, generalmente se utiliza en el lenguaje propio de cada país, razón por la cual se presenta en español. El *LISP,* por otra parte, se utiliza en inglés.

La función **CDR** de *LISP* toma una lista como argumento y produce como salida la misma lista,

menos el primer elemento. La correspondiente función en *Logo* se llama **MENOSPRIMERO** y se abrevia **MPR.** Los nombres CAR y CDR tienen un origen histórico, basados en secciones de instrucciones de la máquina IBM 704, primera en la que se implementó *LISP.* La tradición ha conservado estos nombres y sería difícil desterrarlos. Por otra parte, *Logo*, siendo un lenguaje más reciente, utiliza palabras que tienen un significado asociado con la función que desempeñan.

La función **CONS** de *LISP,* a la inversa de las funciones **CAR** y **CDR** que seccionan listas, sirve para construir listas a partir de pedazos; toma como argumentos un átomo y una lista y los une de manera que el átomo se convierte en el **CAR** de la lista y la lista se convierte en el **CDR** de la misma, o lo que es lo mismo, a la lista le añade por el principio como un primer elemento, el átomo. La función corespondiente en *Logo* es **PONPRIMERO,** que se abrevia **PPR.** (Se está ignorando la posibilidad de hacer un **CONS** con dos átomos, en cuyo caso se obtiene un "par con punto", concepto que no se trata aquí).

La función **EQ** de *LISP* es un caso de una función predicado, la cual sirve para hacer pruebas lógicas. Normalmente, dichas funciones producen como resultado, *cierto* o *falso.* La función toma como argumentos dos átomos y produce *cierto* (que en *LISP* se representa con **T**) si los átomos son iguales y *falso* si no lo son. El resultado *falso* se representa en *LISP* con la palabra especial **NIL,** la cual también se utiliza para representar a la lista vacía sin elementos y que también se puede escribir ( ). En *Logo,* la correspondiente función lógica se llama **IGUAL?** y se le puede aplicar tanto a átomos como a listas. (En *LISP* también existe una función de predicado **EQUAL** que se le aplica tanto a átomos como a listas. Existen ciertas diferencias sutiles entre **EQ** y **EQUAL** de *LISP* en cuyos detalles no vamos a entrar). Otra diferencia entre *LISP* y *Logo* es que *Logo* maneja los operadores numéricos +, –, *, /; y los operadores relacionales <, >, = en notación infijo (además de hacerlo en notación prefijo como *LISP,* por lo que se puede aplicar una prueba lógica con expresiones parecidas a las de FORTRAN. Por ejemplo, **si a = b** <*tal cosa*>. Este tipo de detalles los comentaremos más a fondo en las ilustraciones posteriores.

La función predicado **ATOM** de *LISP* toma un objeto como parámetro y regresa **T** si el objeto es un átomo y **NIL** si no lo es. En *Logo* se utiliza la función lógica **PALABRA?** con igual efecto que en *LISP,* excepto que contesta *cierto* o *falso.*

Con las cinco funciones primitivas básicas de *muLISP Puro* que se describen anteriormente, se pueden escribir otras funciones importantes (esencialmente todas las demás). Como ejemplo se exhibirán dos o tres de ellas. Utilizaremos la función **DEFUN** que proviene de las palabras DEfine FUNction y que tiene la siguiente forma general:

```
(DEFUN nombre (LAMBDA (arg1 arg2 ... )
           tarea1
           tarea2
              . . .   ))
```

Los átomos <arg1>, <arg2>, ... son los nombres formales de los argumentos o parámetros que se utilizan para referirse a los argumentos concretos de la función. El cuerpo de la función contiene una o más tareas.

La función **NULL** es una función importante de *LISP* que se puede utilizar para determinar si una lista está vacía. Su definición en términos de las cinco primitivas básicas es la siguiente:

```
(DEFUN NULL (LAMBDA (OBJ)
     (EQ OBJ NIL) ))
```

Si le aplicamos la función **NULL** a algunas listas obtenemos lo que se muestra a continuación (lo indentado es la contestación de *LISP*)

```
$  (NULL '(A B C))
     NIL
$  (NULL ( ))
     T
```

En la interacción que se muestra aparece el símbolo $ que es una *invitación* (en inglés se llama *prompt*) de *LISP* para que tecleemos una expresión simbólica entre paréntesis, seguida por la opresión de la tecla de retorno del carro para que *LISP* nos la interprete y evalúe. Los resultados producidos por el lenguaje están indentados. Todas las funciones en *LISP* producen algo, esta es una diferencia

con *Logo* en el cual hay *funciones* como las de *LISP* y *comandos,* cuyo trabajo se logra por medio de efectos colaterales, tales como que la tortuga dibuje una línea o cambie el color del fondo de la pantalla. Otra cosa que notamos es la aparición del apóstrofe antes de la lista (A B C). *LISP* interpreta la primera palabra de cualquier lista como una función y las siguientes palabras como los argumentos de la función. Sin embargo, a veces alguno de los argumentos es una lista de objetos y no queremos que se interprete al primer elemento de dicha lista como una función. Para lograr esto, se antecede la lista de un apóstrofe que indica que hay que tomar la lista literalmente como lo que se ve. Este es el caso de la lista (A B C), pues no queremos que *LISP* crea que A es una función y B y C sus parámetros. En *Logo* se usa una convención similar, pero en vez de un apóstrofe se usa un par de comillas " . En *Logo,* las funciones son cadenas de símbolos que no necesariamente aparecen como el primer elemento de una lista rodeada de corchetes. Por lo tanto, cualquier cadena de símbolos que no incluye los símbolos " : [ ] ( ) | y que no coincide con las palabras reservadas del lenguaje, es interpretado en *Logo* como un comando o función. Los parentesis circulares ( ) se utilizan en *Logo* para delimitar algunas funciones que tienen un número indeterminado de parámetros y para cambiar las prioridades de las diversas operaciones, sobretodo en expresiones en infijo. No se utilizan para listas, ya que para ellas se utilizan los corchetes [ ]. Las variables siempre van precedidas, ya sea de *:* que denota valor, o de *"* que denota tomar los símbolos literalmente. Los espacios vacíos son importantes como separadores, tanto en *LISP* como en *Logo,* pero da lo mismo un espacio que varios.

Un segundo ejemplo de una función útil que es primitiva en todas las versiones de *LISP* es la función **APPEND.** Esta función lo que hace es unir dos listas que se le dan como argumentos y producir una sola con los elementos ordenados de los argumentos, como elementos de la lista resultado. Así, si las listas son (A B C) y (D E F) el resultado sería la lista (A B C D E F) y si las listas son ((a b) (c d)) y (e f g h) el resultado sería la lista ((a b) (c d) e f g h). En otras palabras, le quita a cada una de las listas argumento los paréntesis exteriores, encadena todos los elementos y les agrega paréntesis exteriores a la cadena resultante. La definición de **APPEND** en términos de las primitivas básicas es

```
(DEFUN APPEND (LAMBDA (LST1 LST2)
     ((NULL LST1) LST2)
      (CONS (CAR LST1) (APPEND (CDR
LST1) LST2)) ))
```

Al aplicarle esta función a algunas listas obtenemos

```
$  (APPEND '(JUAN PEDRO JAVIER) '(
LUIS ANTONIO))
     (JUAN PEDRO JAVIER LUIS ANTONIO)
$  (APPEND '((A B) C D) '( ))
     ((A B) C D)
```

Notamos que con **APPEND** una lista vacía como argumento desaparece del resultado final, pues al quitarle los paréntesis no queda nada. También que en la definición de **APPEND** aparece la propia función **APPEND.** A esto se le llama *recursión.* Aunque parecería que la definición es circular, cada invocación de APPEND se hace con un argumento más sencillo, pues al usar el CDR le estamos quitando un elemento a la lista LST1, y eventualmente la lista LST1 quedará vacía, por lo que se aplicará la segunda línea de la definición. Tanto LISP como *Logo* son muy propensos a elegantes definiciones recursivas. Desgraciadamente la recursión no es un tema sencillo, lo cual junto con lo oscuro de los primeros manuales de LISP ha hecho que dicho lenguaje adquiriese injustamente la fama de ser un lenguaje difícil. Lo que sí admitimos es que es diferente la manera de pensar de los programadores de LISP (y *Logo*) al compararlos con el típico programador de FORTRAN, Pascal, BASIC o C.

En la definición de APPEND tenemos un ejemplo de lo que en muLISP Puro es una *tarea condicional,* que se distingue de las *tareas simples.* Una tarea simple es aquella cuyo CAR de la lista para ejecutarla es un átomo; por ejemplo, (NULL '(A B C)). Cuando el CAR de la lista correspondiente a la tarea no es un átomo como en ((NULL LST1) LST2), el CAR de una tarea condicional es el *predicado de la condicional.* Si el predicado regresa NIL, el valor de la tarea también es NIL y la evaluación del cuerpo

de la función continúa con la siguiente tarea, en caso de existir. Si el predicado regresa cualquier cosa diferente de NIL, las tareas restantes en el cuerpo de la función son ignoradas y la evaluación continúa usando el CDR de la tarea condicional como las tareas restantes.

Un tercer ejemplo de una función útil en LISP definida por medio de las cinco funciones primitivas básicas, es la función **REVERSE** que invierte el orden de los elementos de una lista y que es primitiva en cualquier versión de LISP. Su definición es la siguiente:

```
(DEFUN REVERSE (LAMBDA (LST)
     ((NULL LST) NIL)
     (APPEND (REVERSE (CDR LST)) (CONS
(CAR LST) NIL)) ))
```

Al aplicarle la función **REVERSE** a una lista se obtiene lo siguiente

```
$  (REVERSE '(1 2 3 4 5 6))
     (6 5 4 3 2 1)
```

Nótese que para definir **REVERSE** se utilizó **APPEND,** la cual fue definida previamente en términos de las funciones primitivas básicas. Conviene aclarar que la definición de **REVERSE** dada es muy ineficiente, pues llama a la función para cada elemento del argumento. Hay definiciones recursivas de **REVERSE** mucho más eficientes que la demostrada, ya que utilizan dos listas, una de ellas inicialmente vacía para irse llenando con los elementos de la lista a invertir en orden inverso. Por economía de espacio solamente se exhibe su definición; y para distinguirla de la anterior la llamaremos **REVERSE2.**

.

```
(DEFUN REVERSE2 (LAMBDA (LST1 LST2)
     ((NULL LST1) LST2)
     (REVERSE2 (CDR LST1) (CONS (CAR
LST1) LST2)) ))
```

Para encontrar una lista invertida, hay que ponerla como primer parámetro de **REVERSE2** y poner la lista vacía como segundo parámetro como se muestra. (En algunas versiones de *LISP* como *muLISP* se pueden omitir parámetros, y éstos toman el valor de la lista vacía como *default* — que es lo que se necesita en este caso — la lista vacía es a la vez un átomo).

```
$  (REVERSE2 '(1 2 3 4 5 6) '( ))
     (6 5 4 3 2 1)
```

Para extraer el segundo y tercer elemento de una lista se pueden utilizar combinaciones de CAR y CDR. Por ejemplo, si se quiere obtener el segundo elemento de una lista se puede definir una función llamada SEGUNDO y para el tercer elemento TERCERO, así como sigue:

```
(DEFUN SEGUNDO (LAMBDA (LST)
     (CAR (CDR LST)) ))
(DEFUN TERCERO (LAMBDA (LST)
     (CAR (CDR (CDR LST))) ))
$  (SEGUNDO '(1 2 3 4 5))
     2
$   (TERCERO '(1 (2 3) (4 5) 6 7))
    (4 5)
```

Las combinaciones funcionales (CAR (CDR LST)), (CAR (CDR (CDR LST))) y otras similares, se usan tan frecuentemente que todas las versiones de *LISP* las tienen como primitivas: a la primera se le llama CADR a la segunda CADDR, y en general, las funciones comienzan con la letra C y terminan con la letra R; asimismo, cada A intermedia representa CAR y cada D intermedia representa CDR. Normalmente se limita el número de letras intermedias a 3 o 4. Estas primitivas ejecutan más rápidamente que las combinaciones de CAR y CDR correspondientes, requieren menos trabajo de teclear y menos paréntesis, por lo que son preferibles. Usando estas primitivas, las definiciones de SEGUNDO y TERCERO quedarían

```
(DEFUN SEGUNDO (LAMBDA (LST)
     (CADR LST) ))
(DEFUN TERCERO (LAMBDA (LST)
     (CADDR LST) ))
```

Se han mostrado algunos ejemplos de cómo con las funciones primitivas básicas se pueden definir otras funciones primitivas de *LISP.* Dado que también se han dado las equivalentes funciones de *Logo*, no resulta demasiado difícil concluir que cualquier cosa que se pueda escribir en *LISP*

se puede traducir a *Logo* y viceversa, con traducciones que son muy similares en sintaxis y en forma de ejecutar, pero con diferentes nombres de funciones, de las cuales las de *Logo* son mas mnemónicas. Para ilustrar esta aseveración haremos la traducción a *LogoWriter* de algunos programas sencillos en *LISP* tomados de Barr y Feigenbaum (1982) y modificados ligeramente para que queden en *muLISP.*

Versiones en *Logo* de algunos Programas en LISP

Uno de los temas favoritos de los investigadores de inteligencia artificial es la solución de acertijos por medio de computadora. Entre las más elegantes soluciones de acertijos están las que se logran por medio de funciones recursivas. Un acertijo muy popular es el conocido como "La Torre de Hanoi," en el cual se tienen tres agujas y un conjunto de N discos de diámetros crecientes colocados los mayores hasta abajo y los menores arriba, como se muestra en la figura 1. El acertijo consiste en decir qué movimientos de discos hay que hacer para lograr colocar todos los discos en la aguja B en la misma forma que están actualmente en la aguja A, con la condición de que durante los movimientos nunca quede un disco mayor arriba de uno menor.

La solución recursiva del acertijo se razona de la siguiente manera:

1. Supóngase que se tiene una estrategia válida para mover los primeros N–1 de los discos de la aguja en la que están, a la aguja que se quiera, utilizando como aguja auxiliar la aguja restante.

2. Entonces lo que se hace para mover los N discos de la aguja A a la B es pasar con la estrategia los primeros N–1 discos de la aguja A a la C, utilizando la aguja B como auxiliar; mover el disco N de la aguja A a la B y finalmente mover con la estrategia los primeros N–1 discos de la aguja C a la B, utilizando la aguja A como auxiliar.

La estrategia descrita es esencialmente la que utiliza la inducción matemática, que dice que si se puede pasar del caso N–1 al caso N y puedo resolver el caso N = 1, entonces tengo la solución para todos los enteros positivos.

A continuación, se muestra la solución del acertijo de la Torre de Hanoi por medio de un programa en *LISP* tomado de Barr y Feigenbaum (1982).

```
(DEFUN MOVETOWER (LAMBDA (DiskList
PegA PegB PegC)
     ((NULL DiskList) NIL)
      (MOVETOWER (CDR DiskList) PegA
PegC PegB)
     (PRINT (LIST 'Move (CAR DiskList)
'from PegA 'to PegB))
      (MOVETOWER (CDR DiskList) PegC
PegB PegA)  ) ))
```

La ejecución del programa para resolver el acertijo de N = 3 discos se muestra a continuación:

```
(MOVETOWER '(Disk3 Disk2 Disk1)  'A 'B
'C)

     (Move Disk1 from A to B)
     (Move Disk2 from A to C)
     (Move Disk1 from B to C)
     (Move Disk3 from A to B)
     (Move Disk1 from C to A)
     (Move Disk2 from C to B)
     (Move Disk1 from A to B)
     NIL
```

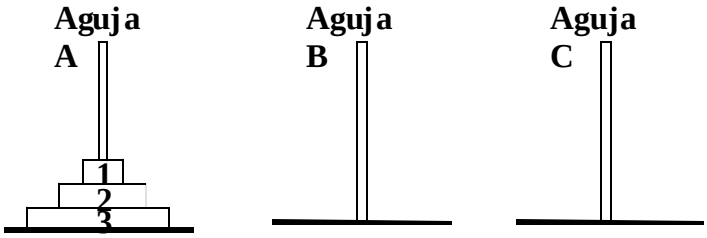

Figura 1

A continuación, se muestra la versión en *Logo* para que el lector pueda apreciar la similitud

```
para MOVETOWER :DiskList :PegA :PegB
:PegC
si no vacia? :Disklist [MOVETOWER mpr
:DiskList :PegA :PegC
     :PegB (escribe (lista "Mueve pr
:DiskList "de :PegA "a
     :PegB))MOVETOWER mpr :DiskList
:PegC :PegB :PegA]
fin
```

La correspondiente ejecución es la siguiente:

```
MOVETOWER [Disco3 Disco2 Disco1 "A "B
"C

Mueve Disco1 de A a B
Mueve Disco2 de A a C
Mueve Disco1 de B a C
Mueve Disco3 de A a B
Mueve Disco1 de C a A
Mueve Disco2 de C a B
Mueve Disco1 de A a B
```

El segundo programa es la función que casi invariablemente se utiliza para explicar la recursión en *LISP,* a saber, la función factorial.

```
(DEFUN FACTORIAL (LAMBDA (N)
     (COND ((EQ N 1) 1)
           (T (TIMES N (FACTORIAL
(DIFERENCE N 1)))) ) ))
```

La ejecución con el argumento N = 5 es

```
(FACTORIAL 5)
     120
```

En *Logo* el correspondiente programa es

```
para FACTORIAL :n
si :n = 1 [re 1]
re :n * FACTORIAL :n - 1
fin
```

La ejecución para n = 5 es

```
escribe FACTORIAL  5
120
```

Conviene hacer notar que los programas no revisan el rango del parámetro N o n, el cual debe ser un número entero positivo. Un programa profesional sí lo revisaría y además escribiría comentarios en caso de encontrar parámetros inválidos.

El tercer ejemplo es un programa para sustituir en una lista llamada Object, cada aparición de un átomo Old por un átomo New, sin importar la complejidad de la lista.

El programa que hace esto en LISP es el siguiente:

```
(DEFUN SUBSTITUTE (LAMBDA (Object Old
New)
 (COND ((ATOM Object)(COND ((EQUAL
Object Old) New)(T Object)
       (T (CONS (SUBSTITUTE (CAR
Object) Old New)
                (SUBSTITUTE (CDR
Object) Old New))) ) ))
```

Un ejemplo de ejecución sigue:

```
(SUBSTITUTE '(PLUS (TIMES A X) X) 'X
'(PLUS 2 3))
     (PLUS (TIMES A (PLUS 2 3)) (PLUS
2 3))
```

La traducción a *Logo* es la siguiente:

```
para SUBSTITUTE :Object :Old :New
si vacia? :object [re []]
siotro no lista? :Object  [siotro
:Object = :Old [re :New]
     [re :Object]] [re ppr SUBSTITUTE
pr :Object:Old :New
     SUBSTITUTE mpr :Object :Old :New]
fin
```

La ejecución correspondiente al ejemplo dado arriba es:

```
escribe SUBSTITUTE [PLUS [TIMES A X]
X] "X [PLUS 2 3]

PLUS [TIMES A [PLUS 2 3]] [PLUS 2 3]
```

Notamos que a diferencia de lo escrito por el programa en *LISP*, lo escrito por el programa en *Logo* no tiene los corchetes inicial y final de lista. Esto se debe a que la instrucción "escribe" despliega los elementos de una lista sin ponerle los corchetes inicial y final; en cambio, la instrucción "muestra" sí se los coloca.

La similitud entre los dos lenguajes debe ser evidente. Algunas de las diferencias tienen que ver con el hecho de que *LogoWriter* considera error tratar de sacar el primer elemento, o la lista menosprimero de una lista vacía, mientras que muLISP regresa NIL en ambos casos. Por esta razón, en *LogoWriter* hay que prever con más cuidado que ciertas listas no se vacíen para que la ejecución no se interrumpa por error.

El último ejemplo que consideraremos tomado también de Barr y Feigenbaum (1982), tiene que ver con la manipulación de hechos y reglas. El conjunto de tres programas que se exhibieron, hace deducciones lógicas de una base de datos con aseveraciones representadas en *LISP* por medio de listas. Las aseveraciones son de dos tipos: hechos que indican que es verdadero cierto predicado sobre cierto objeto (por ejemplo (HOMBRE Sócrates) asegura que Sócrates es un hombre), y reglas generales que indican que un predicado implica a otro. Estas reglas quedarán representadas por listas de la forma (TODO predicado$_1$ predicado$_2$); por ejemplo (TODO HOMBRE MORTAL) asegura que todos los hombres son mortales.

El programa PROVE, cuyo listado en LISP aparece abajo, toma dos argumentos: una aseveración como (MORTAL Sócrates) y una base de datos en la forma de una lista de aseveraciones; así también regresa T si la declaración se puede deducir de la base de datos y NIL si no se puede. La función PROVE utiliza dos funciones auxiliares FINDASSERTION y PROVESIT. La primera peina la base de datos buscando, ya sea una aseveración igual a la que se quiere demostrar, o en su defecto, una aseveración que partiendo de un predicado se implique un segundo predicado más una aseveración que asegure que el segundo predicado sea verdadero. Por ejemplo, si se quiere demostrar (MORTAL Sócrates) y no estando dicha aseveración en la base de datos se encuentra (TODO HOMBRE MORTAL) y también (HOMBRE Sócrates), se puede deducir que como todo hombre es mortal y Sócrates es hombre, entonces Sócrates es mortal, que es lo que queríamos demostrar. La segunda función se encarga de los detalles, de analizar si se da la cadena de razonamiento mencionada.

```
$  (DEFUN PROVE (LAMBDA (Statement
DataBase)
     (FINDASSERTION DataBase) ))
$      PROVE

$  (DEFUN FINDASSERTION (LAMBDA
(RestOfDataBase)
     (COND ((NULL RestOfDataBase) NIL)
           ((OR (PROVESIT (CAR
RestOfDataBase))
          (FINDASSERTION (CDR
RestOfDataBase)) )) ) ))
$     FINDASSERTION

$  (DEFUN PROVESIT (LAMBDA (Assertion)
     (OR  (EQUAL Statement Assertion)
     (AND (EQUAL (CAR Assertion) 'ALL)
          (EQUAL (CADDR Assertion)
(CAR Statement))
          (PROVE (CONS (CADR
Assertion) (CDR Statement))  DataBase)
) ) ))
$     PROVESIT
```

Vamos a suponer el caso muy sencillo de que se quiere demostrar que es verdadero (MORTAL Sócrates) y que nuestra base de datos tiene las siguientes aseveraciones (TODO HOMBRE MORTAL) y (HOMBRE Sócrates), entonces lo que se debe ejecutar es

```
$ (PROVE '(MORTAL Sócrates) '((HOMBRE
Sócrates)(TODO HOMBRE   MORTAL)
     T
```

La versión en *LogoWriter* de las tres funciones es

```
para PROVE :Statement :DataBase
re FINDASSERTION :DataBase
fin

para FINDASSERTION :RestOfDataBase
si vacia? :RestOfDatabase [re "falso]
```

```
si PROVESIT pr :RestOfDataBase [re
“cierto]
si FINDASSERTION mpr :RestOfDataBase
[re “cierto]
re “falso
fin

para PROVESIT :Assertion
si igual? :Statement :Assertion [re
“cierto]
si no igual? pr :Assertion “ALL [re
”falso]
si (cuenta :Assertion) < 3 [re “falso]
si no igual? pr mpr mpr :Assertion pr
:Statement [re “falso]
si no PROVE ppr pr mpr :Assertion mpr
:Statement :DataBase [re “falso]
re “cierto
fin
```

En la versión de *LogoWriter* por el asunto de los errores al querer tomar el primer elemento o la menosprimer lista de una lista vacía, se tuvieron que expandir en el procedimiento PROVESIT las expresiones lógicas compuestas en expresiones sencillas, probando primero si la lista que representa la aseveración en la base de datos tiene menos de tres elementos, para que al momento de tomar el **pr mpr mpr** no se genere un error y se suspenda la ejecución. Fuera de detalles como este, la lógica en los dos trios de procedimientos *LISP* y *LogoWriter* es la misma y los detalles muy parecidos.

Al ejecutar la siguiente línea se obtiene lo que se muestra

```
escribe PROVE [MORTAL Sócrates]
[[HOMBRE Sócrates] [TODO HOMBRE
MORTAL]]

cierto
```

Si por otro lado, la aseveración a demostrar es [MORTAL Sócrates] y la base de datos fuera [HOMBRE Sócrates] [TODO ANIMAL MORTAL], en la corrida con estos datos los procedimientos contestarían falso. Asimismo, si se plantea la misma aseveración a demostrar y la base de datos fuese [HOMBRE Sócrates] [TODO HOMBRE ANIMAL] [TODO ANIMAL MORTAL], entonces la corrida contestaría *cierto*.

Conclusiones

El propósito de este artículo ha sido exhibir la similitud del lenguaje *Logo* representado por la versión *LogoWriter* con el lenguaje LISP, que a su vez, es representado por la versión muLISP. Para mostrar dicha similitud, se analizaron las funciones primitivas básicas de lo que se llama muLISP Puro y se hizo una correspondencia muy cercana con instrucciones de *LogoWriter*. Al mismo tiempo, se señalaron las similitudes de las estructuras de datos que manejan los dos lenguajes: listas, nombres y números. En general, las versiones de LISP que se tienen en el mercado tienen más funciones que las de *Logo*, pues el LISP es utilizado por profesionales de la computación, en particular, de la inteligencia artificial, mientras que *Logo* es utilizado en la enseñanza, frecuentemente por estudiantes de primaria y secundaria. No obstante, esperamos haber convencido al lector de que en esencia son equivalentes los dos lenguajes, cosa que no debe extrañarnos, pues ambos fueron creados por investigadores de inteligencia artificial. Hay algunas diferencias que, descartada la diferencia de número y variedad de funciones, inclina la balanza en favor de *Logo*, en opinión del autor. Entre estas diferencias, están el uso de operadores aritméticos con notación de infijo, como están acostumbrados los estudiantes de matemáticas. Se descarga el uso del paréntesis que es una de las críticas fuertes en contra de LISP que obliga a la cuenta cuidadosa de normalmente muchos paréntesis. En *Logo*, los paréntesis se utilizan para cambiar la prioridad de las operaciones en notación infijo y para manejar funciones con un número variable de argumentos. Para las listas, en *Logo* se utiliza el corchete. Otra de las ventajas de *Logo*, es que los nombres son más mnemónicos que en LISP y se tienen versiones en las lenguas locales. Finalmente, entre las ventajas importantes está la tortuga y las facilidades para hacer dibujos. Adicionalmente, casi todas las versiones de *Logo* tienen instrucciones para manejar notas musicales y algunas como *LogoWriter* tienen instrucciones para manejar textos largos.
La conclusión final es que *Logo* es un lenguaje adecuado para estudiar inteligencia artificial,

precisamente por su similitud con LISP que es el lenguaje de computadora de la inteligencia artificial por antonomasia.

Referencias

Barr A. y Feigenbaum E.A. (1982). The Handbook of Artificial Intelligence. *HeurisTech Press*, Vol. II, Stanford, CA.

The Software House (1984). *Tutorial System for muLISP-83 (software)*.

Bibliografía sugerida

Charniak E. y McDermott D. (1985). *Introduction to Artificial Intelligence.* Addison-Wesley Publishing Company, Reading, MA.

Logo Computer Systems, Inc. (1990). *Logo Writer: Guía de Referencia.* Macrobit Editores SA. de CV, México.

Winston P.H. (1977). *Artificial Intelligence.* Addison-Wesley Publishing Company, Reading, MA.

Semblanza del autor

Marco Antonio Murray-Lasso. Realizó la licenciatura en ingeniería mecánica-eléctrica en la Facultad de Ingeniería de la UNAM. El Instituto de Tecnología de Massachussetts (MIT) le otorgó los grados de maestro en ciencias en ingeniería eléctrica y doctor en ciencias cibernéticas. En México, ha laborado como investigador en el Instituto de Ingeniería y como profesor en la Facultad de Ingeniería (UNAM) durante 43 años; en el extranjero, ha sido asesor de la NASA en diseño de circuitos por computadora para aplicaciones espaciales, investigador en los Laboratorios Bell, así como profesor de la Universidad Case Western Reserve y Newark College of Engineering, en los Estados Unidos. Fue el presidente fundador de la Academia Nacional de Ingeniería de México; vicepresidente y presidente del Consejo de Academias de Ingeniería y Ciencias Tecnológicas (organización mundial con sede en Washington que agrupa las Academias Nacionales de Ingeniería) y secretario de la Academia Mexicana de Ciencias. Actualmente es jefe de la Unidad de Enseñanza Auxiliada por Computadora de la División de Ingeniería Mecánica e Industrial de la Facultad de Ingeniería de la UNAM, investigador nacional en ingeniería, consejero educativo del MIT y consultor de la UNESCO.