



Trabajo N° 2

“Introducción a LISP y PROLOG”

Sistemas Inteligentes

Sergio Troncoso Duque
18/04/2017

Introducción

Antiguamente el uso de computadoras en el área industrial era escaso, impensable para el uso doméstico. A raíz de esto existían muy pocos lenguajes de programación, el más popular de la época siendo Fortran creado por IBM para el uso de los supercomputadores de la época. Se crea entonces un año después, en base a la notación matemática de Alonzo Church, el lenguaje LISP.

El lenguaje Lisp (o LISP) fue uno de los pioneros de muchas ideas relacionadas con la ciencia de la computación como la estructura de datos árboles, sentencias condicionales, funciones de alto orden y recursión. Su nombre se basa en “**LISt Processor**” ya que la lista enlazada es su principal estructura de dato. Hoy en día existen muchos tipos de dialecto de Lisp, los más utilizados son Common Lisp y Scheme. Basado altamente en una sintaxis de paréntesis, utilizando las s-expression (symbolic expressions) para la creación de los programas. Las expresiones simbólicas funcionan como árboles donde una expresión es una hoja y se pueden ir anidando para crear un programa con varias expresiones. Además es considerado uno de los pioneros en el uso de macros, formando lo que se conocen como DSL (domain-specific language).

El trabajo que se llevo a cabo con este lenguaje se expande a través de muchas disciplinas, pero tiene una mayor presencia en el área de la inteligencia artificial. También destaca en este tópico el lenguaje de programación lógico llamado Prolog.

Creado por Alain Colmerauer, este lenguaje de programación es prácticamente único por ser declarativo, es decir, todas las sentencias se presentan en forma de reglas y hechos. El lenguaje sigue siendo uno de los más populares y se ha usado para probar teoremas, creación de sistemas expertos (toma de decisiones por parte de un computador) y es el padre de las técnicas de procesamiento de lenguajes naturales.

La implicancia de estos lenguajes dentro de la inteligencia artificial es indudable, ya que son los fundadores de muchas bases y conceptos que son necesarios para entender la disciplina. Se hace necesario que se tengan conocimientos básicos de estos lenguajes de programación para entender estos conceptos y poder entender completamente lo que es la inteligencia artificial.

Desarrollo del problema

Para la obtención de conocimientos de ambos lenguajes de programación, se deben realizar 8 ejercicios de programación los cuales ejercitan tanto el uso de estructura de datos como el diseño de algoritmos en base a recursión.

Para el lenguaje LISP se deben realizar los siguientes ejercicios:

- 1) Una función que a la que se le entregue un número N y devuelva el cubo de ese número.
- 2) Una función que a la se le entregue un número N y devuelva el factorial de es número.
- 3) Tres funciones a las que se le entregue una lista, y devuelvan el primer elemento, el segundo y el tercero correspondientemente.
- 4) Una función que devuelva el promedio del cuadrado de dos números.
- 5) Una función que resuelva una ecuación cuadrática de la forma $Ax^2 + Bx + C$ cuando se le entreguen los valores de A, B, C.
- 6) Una función que reciba una lista y rote su primer elemento a la izquierda.

En cambio, para el lenguaje Prolog se deben realizar los siguientes programas:

- 1) Una función que reciba una lista y entregue la longitud de tal lista.
- 2) Una función que reciba un árbol y entregue el número más grande este.
- 3) Una función que reciba una elemento y verifique si está en una lista.

Análisis de Resultados

Ejercicio 1: Función cubo

El código de este ejercicio es el siguiente:

```
1 (defun cubo (X)
2   (* X X X)
3 )
```

Figura 1. Código función cubo

En la primera línea definimos el nombre de la función, usando la palabra reservada del lenguaje **defun**, se le llama a la función cubo la cual recibe el parámetro X. Luego en la línea 2, se llama a la función algebraica de la multiplicación a la cual se le dan tres parámetros, todos el valor X, para formar la multiplicación $X \cdot X \cdot X$ que es igual al cubo de X.

En la siguiente imagen podemos ver su correcto funcionamiento:

```
* (load "tarea2-1.lisp")

T
* (cubo 3)

27
```

Figura 2. Resultado Ejercicio 1

Ejercicio 2: Función factorial

El código de este ejercicio es el siguiente:

```
1 (defun factorial (N)
2   (cond ((zerop N) 1)
3         (t (* N (factorial (- N 1)))))
4   )
5 )
6 )
7 )
8 )
```

Figura 3. Código Ejercicio 2

En la primera línea se vuelve a utilizar `defun` para la creación de la función la cual llamaremos `factorial`. Luego en la línea 2 se utiliza la macro `cond` la cual permite generar una condición en base a un `form`, en este caso, se pregunta a través de la función `zerop` si el número `N` es igual a cero, lo cual entrega un booleano en donde si es verdad se devuelve 1, si es falso se continua en la línea 3 donde `t` nos indica que se debe entregar, al cual se le da la función multiplicación con dos parámetros, `N` (nuestro número) y recursivamente, la función creada `factorial` pero como parámetro inicial se le entrega la resta `N - 1`.

En la siguiente imagen podemos ver su correcto funcionamiento:

```
* (load "tarea2-2.lisp")

T
* (factorial 0)

1
* (factorial 1)

1
* (factorial 5)

120
```

Figura 4. Funcionamiento Ejercicio 2

Ejercicio 3: Funciones de lista

El código de este ejercicio es el siguiente:

```
1 (defun primero (L)
2   (car L)
3 )
4
5 (defun segundo (L)
6   (car (cdr L))
7 )
8
9 (defun tercero (L)
10  (car (cdr (cdr L)))
11 )
```

Figura 5. Código Ejercicio 3

Para las tres funciones se utilizaron dos funciones de LISP. En la primera función que se creo se utiliza la función **car**, la cual entrega el primer valor de una lista. Luego en la segunda función se utiliza tanto la función **car** como la función **cdr**, la cual retorna el resto de la lista, es decir, todo lo que sigue al primero. Por ende lo que primero se obtiene es el resto de la lista, del segundo ítem hasta el n-esimo, y luego se obtiene el primer elemento de esta sublista, que sería para nosotros el segundo elemento de la lista general. Para el tercer elemento se utiliza el mismo método, se “remueven” los elementos con **cdr** y se obtiene el primer elemento de la lista que queda con **car**.

En la siguiente imagen podemos ver su correcto funcionamiento:

```
* (load "tarea2-3.lisp")

T
* (primero '(A B C) )

A
* (segundo '(A B C) )

B
* (tercero '(A B C) )

C
```

Figura 6. Funcionamiento Ejercicio 3

Ejercicio 4: Función el promedio del cuadrado

El código de este ejercicio es el siguiente:

```
1  (defun prom (num1 num2)
2    (/
3      (+ (* num1 num1) (* num2 num2)) 2.0
4    )
5  )
```

Figura 7. Código Ejercicio 4

El código siguiente sigue las mismas directrices que los códigos anteriormente mostrados. Se define el nombre **prom** con **defun**, se le entregan dos parámetros a esta función. Dentro de la función se ejecuta la función división, la cual se le entregan dos parámetros, el segundo (el divisor) es 2.0, tiene estos decimales para que el resultado que entregue la función también los tenga. El primero es una función suma, que recibe dos funciones como parámetros, ambas multiplicaciones, que generan el cuadrado de ambos números.

En la siguiente imagen podemos ver su correcto funcionamiento:

```
* (load "tarea2-4.lisp")

T
* (prom 2 5)

14.5
* (prom 4 9)

48.5
```

Figura 8. Funcionamiento Ejercicio 4

Ejercicio 5: Función cuadrática

El código de este ejercicio es el siguiente:

```
1 (defun cuadra (A B C)
2   (setq x1 ( / (+ (* -1 B) (sqrt (- (* B B) (* 4 A C)))) (* 2 A)))
3   (setq x2 ( / (- (* -1 B) (sqrt (- (* B B) (* 4 A C)))) (* 2 A)))
4   (format t "x1 = ~2d x2 = ~2d" x1 x2)
5
6 )
```

Figura 9. Código Ejercicio 5

Siguiendo con la misma lógica de los ejercicios anteriores, en la primera línea definimos la función como **cuadra**, luego en las líneas 2 y 3 realizamos una serie de funciones anidadas para resolver la fórmula para la solución de la ecuación cuadrática. Se utiliza **sqrt** para calcular la raíz cuadrada, se multiplica B por un -1 para obtener su valor negativo. El único cambio que tienen ambas variables que definimos con **setq** es que la segunda función es una suma en vez de una resta. Finalmente, en la línea 5 utilizamos **format** para mostrar por pantalla ambas variables (x1 y x2) por pantalla.

En la siguiente imagen podemos ver su correcto funcionamiento:

```
* (cuadra 1 -7 12)
x1 = 4.0 x2 = 3.0
NIL
```

Figura 10. Funcionamiento Ejercicio 5

Ejercicio 6: Función rotar a la izquierda

El código de este ejercicio es el siguiente:

```

1 (defun izquierda (L)
2   (append
3     (cdr L) (list
4       (car L)
5     )
6   )
7 )

```

Figura 11. Código Ejercicio 6

En la primera línea definimos la función con nombre **izquierda** y definimos que recibe una lista L. Luego en la línea 2 utilizamos la función **append** para unir dos listas, pero ¿Cuáles dos listas? En la línea 4 utilizamos **car** para obtener el primer elemento de L, que es el elemento que queremos rotar al final de la lista, luego aplicamos la función **list** la cual genera una lista en base a un elemento o elementos ya que **car** por si solo nos entrega un ítem y no una lista. La otra lista está en la línea 3, donde utilizamos **cdr** para obtener todo lo demás de L. Al final unimos estas dos listas para generar la lista con el elemento rotado a la izquierda.

En la siguiente imagen podemos ver su correcto funcionamiento:

```

* (load "tarea2-6.lisp")

T
* (izquierda '(A B C D))

(B C D A)
* (izquierda '(B C D A))

(C D A B)
* (izquierda '(C D A B))

(D A B C)
* (izquierda '(D A B C))

(A B C D)

```


Figura 12. Funcionamiento Ejercicio 6

Ejercicio 7: Función longitud de una lista

El código de este ejercicio es el siguiente:

```
1 longitud([],0).  
2  
3 longitud(_|L,N):-  
4     longitud(L,N1),N is N1 + 1.
```

Figura 13. Código Ejercicio 7

En la primera línea impone la primera regla, la cual es si se entrega una lista vacía el valor de N es 0. Para la tercera línea la segunda regla dice que si se entrega una lista, aunque se denota de la forma `_|L`, donde L es la cola de la lista o mejor dicho el resto de la lista y `_` ignora la cabeza de esta. Luego se coloca `:-` para denotar una condición al ocurrir esta regla, la cual se describe en la línea cuatro. Acá recursivamente llamamos a nuestra regla nuevamente pero le entregamos la cola y un valor N1, además se entrega dice que N es ahora $N1 + 1$. El resultado es que se le entregara la cola siempre a la regla de la línea 3 hasta que caiga en la primera regla que propusimos, además el valor de N sumara uno por cada iteración partiendo desde 0, entregándonos el valor total del largo de la lista.

En la siguiente imagen podemos ver su correcto funcionamiento:

```
?- consult('tarea2-7.pro').  
true.  
  
?- longitud([a,b,c,d,e,f,g,h],N).  
N = 8.
```

Figura 14. Funcionamiento Ejercicio 7

Ejercicio 8: Función el número más grande en un árbol

El código de este ejercicio es el siguiente:

```

1  maximo(nil,0).
2
3  maximo(t(I,R,D),M):-
4      maximo(I,MI),
5      maximo(D,MD),
6      M1 is max(MI,MD),
7      M is max(R,M1).

```

Figura 15. Código Ejercicio 7

En la primera línea definimos que el máximo es 0 si el valor dado por el árbol es nil (o null más comúnmente llamado). Luego en la línea siguiente creamos otra regla en donde le entregamos el árbol como estructura el cual recibe una raíz (R) , su nodo en izquierda (I) y su nodo en derecha (D). Luego en la línea 4 recursivamente recorre los nodos izquierdos del árbol y en la línea 5 los derechos de igual forma. En las líneas 6 y 7 utiliza la función **max** para obtener el valor mayor entre cada hoja (izquierda o derecha) y para calcular el valor final al ver cual es mayor con la raíz, correspondientemente.

En la siguiente imagen podemos ver su correcto funcionamiento:

```

?- consult('tarea2-8.pro').
true.

?- maximo(t( t( t(nil,2,nil), 6, t(nil,9,nil)), 18, t( t(nil,7,nil), 8, t(nil,27,nil))),M).
M = 27.

```

Figura 16. Funcionamiento Ejercicio 8

Ejercicio 9: Función pertenencia en una lista

El código de este ejercicio es el siguiente:

```

1  pertenece( [X|_],X).
2
3  pertenece( [_|L],X):-
4      pertenece(L,X).

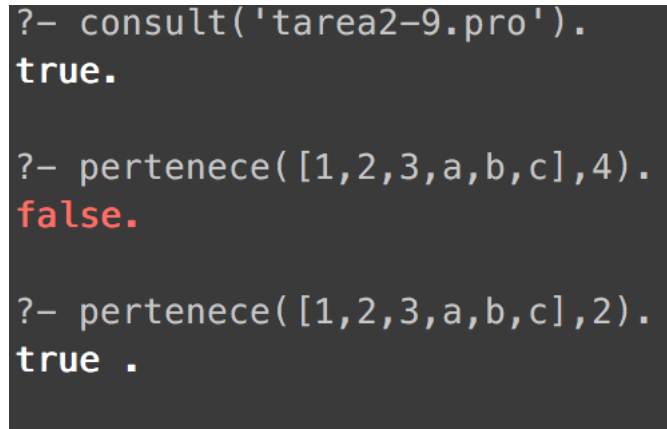
```

Figura 17. Código Ejercicio 9

En la línea 1 definimos la primera regla que es que verifique que la cabeza de la lista, es decir, que el primer elemento de la lista sea o no el elemento X que se entrega por parámetro. Luego la segunda regla utiliza la recursión para comprobar, al igual que en el ejercicio 7, se le entrega la cola de la lista y se le agrega una condición en la línea 4 que es la misma función pero evaluando solo la cola. Esto hará que revise todas las posiciones hasta

el último elemento, si el último elemento no igual a X devolverá falso pero si se encuentra con X en alguna posición entonces devolverá verdadero.

En la siguiente imagen podemos ver su correcto funcionamiento:



```
?- consult('tarea2-9.pro').
true.

?- pertenece([1,2,3,a,b,c],4).
false.

?- pertenece([1,2,3,a,b,c],2).
true .
```

Figura 17. Funcionamiento Ejercicio 9

Conclusiones

Luego de la realización de estos ejercicios se pudo concluir que ambos lenguajes de programación utilizan la recursividad como método para realizar acciones que otros lenguajes como C no utilizan.

También se identificó que las estructuras de datos de los lenguajes carecían de funciones propias como en otros lenguajes. En Prolog existían ciertas notaciones que podían facilitar la creación de reglas como se vieron en los ejercicios 7 y 9.

Habían ciertas similitudes entre Lisp y Prolog cuando se utilizaban funciones para retornar un valor. Aunque es mucho más difícil emular una función Lisp con una regla de Prolog que al revés.

Por último, estos lenguajes de programación al ser los pioneros y tener muchos años de desarrollo, existen muchas formas de realizar las funciones que se realizaron acá e incluso en el caso de Lisp, estas funciones pueden no funcionar en otros dialectos del mismo lenguaje o incluso el mismo intérprete. A pesar de eso, estos lenguajes son sumamente poderosos al crear