

**UNIVERSIDAD ANDRÉS BELLO**

**FACULTAD DE INGENIERÍA**

**INGENIERÍA CIVIL INFORMÁTICA**

**Trabajo 4 – Prolog/Lisp**

**ANDRÉS EDUARDO VALENZUELA GONZÁLEZ**

**SANTIAGO - CHILE**

**OCTUBRE, 2017**

Contenido

[**1.** **Introducción** 3](#_Toc495622399)

[**2.** **Ejecución** 3](#_Toc495622400)

[2.1. Errores 4](#_Toc495622401)

[2.2. Solución propuesta 5](#_Toc495622402)

[**3.** **Valores mas representativos** 6](#_Toc495622403)

[**4.** **Cálculo y análisis de resultados** 6](#_Toc495622404)

[4.1. Cluster 1 (c0): 7](#_Toc495622405)

[4.2. Cluster 2 (c1): 8](#_Toc495622406)

[4.3. Cluster 3 (c2): 8](#_Toc495622407)

[4.4. Cluster 4 (c3): 8](#_Toc495622408)

[4.5. Análisis de resultados 9](#_Toc495622409)

[**5.** **DBSCAN** 9](#_Toc495622410)

[**6.** **Conclusiones** 11](#_Toc495622411)

# **Introducción a Lisp**

En los comienzos de la era computacional resultaba impensable el uso de computadoras para uso doméstico, dado a lo cual no se disponía de una amplia gama de lenguajes de programación. Uno de los primeros (y más conocido hoy en día) lenguajes de alto nivel creado fue *FORTRAN* para propósito general (Desarrollado por *IBM* en el año 1957). Luego, en 1958 fue desarrollado *Lisp* (**LIS**t **P**rocessor) en el Instituto Tecnológico de Massachusetts por el ingeniero y ganador del premio Turing *John McCarthy* con el propósito de representar una notación matemática practica para los programas de computadoras basada en el cálculo *lambda* (de *Alonzo Church*). Mencionado lenguaje se convirtió en el preferido por investigadores de IA (*Inteligencia Artificial)*. También destaca por ser el lenguaje pionero en incluir estructuras de datos tales como los arboles binarios, el almacenamiento automático, tipos de datos dinámicos y compiladores auto contenidos. Pero el elemento fundamental de este lenguaje son las *Listas* (tanto los programas como sus tipos de datos son listas, de ahí su acrónimo List Processor) ya que, para la *IA* es el hecho de que el código y los datos tengan el mismo tratamiento como listas lo que hace especialmente sencillo escribir programas capaces de escribir otros programas según las circunstancias.

# **Introducción a Prolog**

Junto con *LISP*, *Prolog* es un lenguaje utilizado ampliamente en el campo de la inteligencia artificial.  
Creado por Alain Colmerauer a principio de los años 70 en la Universidad de Aix-Marseille I (Marseille, Francia) con el propósito de clasificar lenguajes naturales, cambio su objetivo a ser un lenguaje de programación único por ser declarativo (en donde se declaran/especifican un conjunto de condiciones, proposiciones, afirmaciones, restricciones o transformaciones que describen el problema y detallan su solución) y semi-interpretado, esto significa que también posee un compilador. Todas las sentencias se presentan en forma de reglas y hechos.

La participación de estos lenguajes dentro del campo de la inteligencia artificial es irrefutable, además de haber sido los fundadores de muchas bases y conceptos necesarios para entender el área de *IA*.

Para entender la extensión de este informe, se declara necesario el conocimiento base respecto a estos lenguajes de programación para poder comprender y analizar los siguientes ejemplos.

# **Desarrollo del problema**

Para desarrollar conocimiento y convocar la practica ante ambos lenguajes de programación fueron propuestos 8 ejercicios, los cuales solicitan conocimientos tanto en el uso de estructura de datos como el diseño de algoritmos recursivos.

Para Lisp se solicita la resolución de los siguientes ejercicios:

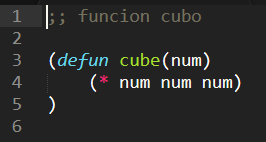
1. Una función a la que se le entregue un numero N y retorne el cubo de ese mismo número.
2. Una función a la que se le entregue un numero N y retorne su valor factorial.
3. Tres funciones que reciban por parámetro una lista y que devuelvan respectivamente el primer, segundo y tercer elemento de mencionada lista.
4. Una función que retorne el promedio cuadrado de dos números.
5. Una función que calcule y resuelva la ecuación cuadrática de la forma .
6. Una función que reciba una lista y la rote un elemento hacia la izquierda.

Se solicita además implementar las funciones en Prolog:

1. Una función que calcule la longitud de una lista.
2. Una función que entregue el número más grande dentro de un árbol.
3. Una función que verifique si un elemento está o no en una lista.

# **Analisis de Resultados**

* 1. Función cubo:   
     Se presenta el siguiente código:



En la primera línea se definen los comentarios. Siguiendo por la tercera línea se ubica la cabecera de la función, la cual se define con la palabra reservada *defun*, a continuación el nombre de la función y luego los parámetros que esta recibirá.  
El cuerpo de la función (línea 4) se compone de la típica notación polaca de Lisp (o más conocida por notación prefija, en la cual se tipea primero el operador y seguidos los operandos como muestra la imagen). En este ejercicio se vislumbra la multiplicación del numero ingresado por parámetro tres veces por si mismo.

En la siguiente imagen podemos ver su correcto funcionamiento:

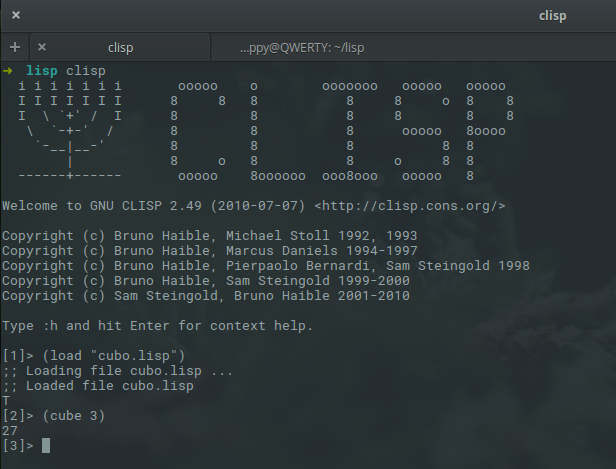
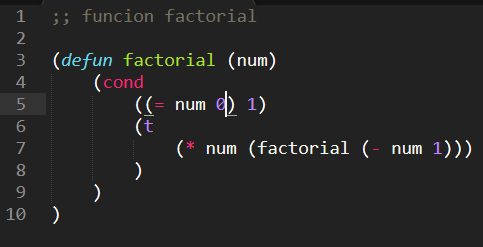


Ilustración 1.- Función cubo

* 1. Función factorial:

Se presenta el siguiente código:



En la línea 3 se vuelve a utilizar la palabra reservada *defun* para definir una nueva función, seguida por el nombre de esta y los parámetros a recibir.  
En la línea 4 se puede visualizar el uso de la macro *cond*, la cual permite evaluar la condición de la línea 5. Si esta se cumple, se retorna 1, pero de ser falsa se continua en la línea 6 en donde ***t***indica lo que debe ser retornado por la función, en este caso seria la multiplicación entre el numero recibido por parámetro y la llamada de la función actual reduciendo el numero en una unidad. Debido a este decremento, el numero cumplirá luego con la condición de termino *(= num 0)* y luego se multiplicaran los valores retornados.

A continuación se verifica que el código cumple con su propósito:

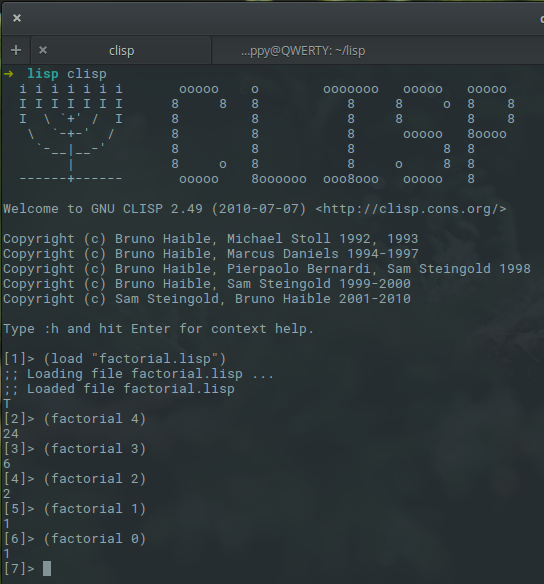
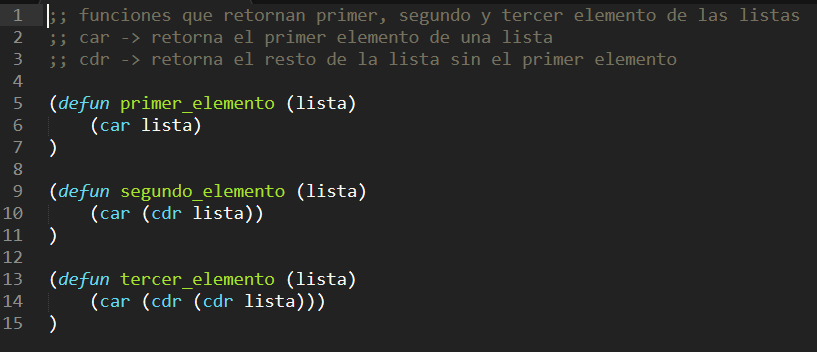


Ilustración 2.- Función factorial

* 1. Funciones primer, segundo y tercer elemento de una lista:

Se presenta el siguiente código:



Se parte mencionando que la macros *car* corresponde a una función que retorna el primer elemento de la lista que se le entregue por parámetro. Luego, la macros *cdr* corresponde a una función que retorna el resto de la lista (la misma lista sin el primer elemento), lo cual resulta bastante conveniente en este ejercicio.

En las líneas 5, 9 y 13 se visualiza el uso de la palabra reservada *defun*, el nombre de cada función y por ultimo el parámetro que recibe (nombre a convenir, puede ser el que uno guste. Por convención se escogio “lista”). Consecutivo a la línea 5, se ubica el uso de la macro *car* y luego la lista entregada por parámetro.

En la línea 10 se puede contemplar el uso de las macros *car* y *cdr*, de las cuales se considera su funcionalidad en sentido de derecha a izquierda, es decir, primero la macro *cdr* nos retorna la cola de la lista y luego la macro *car* extrae el primer elemento de esta cola (segundo elemento)

# **Cálculo y análisis de resultados**

Una vez interpretados los clusters y sus datos contenidos, se pueden realizar diversos cálculos tales como la precisión, el recall, F1-score, entropías y puritys respectivos.

1. Para calcular la entropía, se utilizara la siguiente fórmula propuesta por las ponencias del curso *Sistemas Inteligentes*:

*Sea | D | el total de datos del dataset, | Di | la cantidad de datos contenidos por cluster y cj la cantidad de datos por clase*

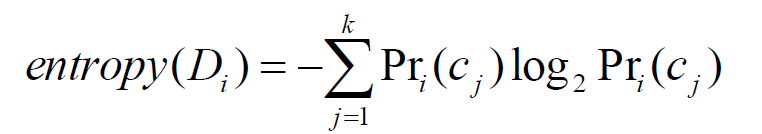


Ilustración 3 - Entropía por clase

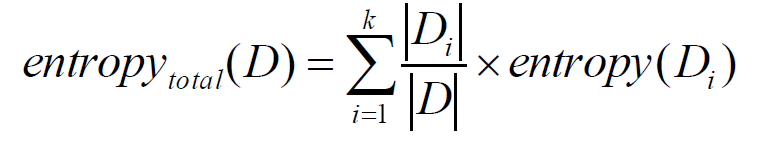


Ilustración 4 - Entropía total

1. Para calcular la pureza *(purity)* se utilizaron las siguientes formulas:

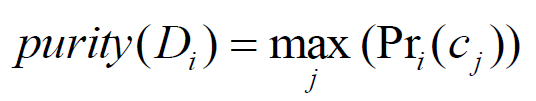


Ilustración 5 - Purity cluster

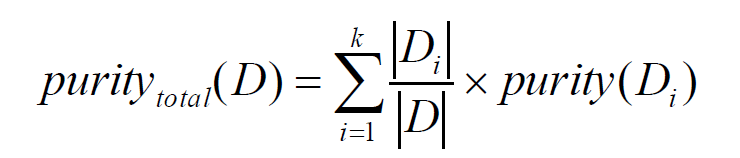


Ilustración 6 - Purity total

1. Para calcular la precisión, el recall y el F1 – Score de cada cluster se tiene:

### Cluster 1 (c0):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Undetermined | World | Non-USA | USA only |
| Precisión | *0.0845070422535* | *0.267605633803* | *0.0281690140845* | *0.619718309859* |
| Recall | *0.015* | *0.0475* | *0.005* | *0.11* |
| F1-score | *0.0254777070064* | *0.0806794055202* | *0.00849256900212* | *0.186836518047* |
| Entropía | *0.301249404404* | *0.508937641059* | *0.145063299141* | *0.427801155467* |
| Purity | *0.0845070422535* | *0.267605633803* | *0.0281690140845* | *0.619718309859* |
| Total datos | **30** | **95** | **10** | **220** |

### Cluster 2 (c1):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Undetermined | World | Non-USA | USA only |
| Precisión | *0.195402298851* | *0.256704980843* | *0.0766283524904* | *0.471264367816* |
| Recall | *0.0255* | *0.0335* | *0.01* | *0.0615* |
| F1-score | *0.0451127819549* | *0.0592658115878* | *0.0176912870411* | *0.108801415303* |
| Entropía | *0.460266334807* | *0.50360814563* | *0.283982980972* | *0.511506334948* |
| Purity | *0.195402298851* | *0.256704980843* | *0.0766283524904* | *0.471264367816* |
| Total datos | **51** | **67** | **20** | **123** |

### Cluster 3 (c2):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Undetermined | World | Non-USA | USA only |
| Precisión | *0.269230769231* | *0.263736263736* | *0.0274725274725* | *0.43956043956* |
| Recall | *0.0245* | *0.024* | *0.0025* | *0.04* |
| F1-score | *0.044912923923* | *0.0439963336389* | *0.00458295142071* | *0.0733272227314* |
| Entropía | *0.509676675869* | *0.507120564258* | *0.142468861135* | *0.521260019917* |
| Purity | *0.269230769231* | *0.263736263736* | *0.0274725274725* | *0.43956043956* |
| Total datos | **49** | **48** | **5** | **80** |

### Cluster 4 (c3):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Undetermined | World | Non-USA | USA only |
| Precisión | *0.349417637271* | *0.278702163062* | *0.0357737104825* | *0.336106489185* |
| Recall | *0.21* | *0.1675* | *0.0215* | *0.202* |
| F1-score | *0.262336039975* | *0.209244222361* | *0.0268582136165* | *0.252342286071* |
| Entropía | *0.530058051979* | *0.513704912602* | *0.171891120065* | *0.528698767035* |
| Purity | *0.349417637271* | *0.278702163062* | *0.0357737104825* | *0.336106489185* |
| Total datos | **420** | **335** | **43** | **404** |

**Entropía total: 1.04835606386  
Purity total: 0.601**

### Análisis de resultados

Sorpresivamente la mayoría de los datos fue anexada al cuarto cluster. Se puede observar además que todos los clusters contienen datos de las cuatro clases evaluadas, de esto se puede inferir que, y por teoría, todos los datos dentro de un cluster cumplen con alguna característica en común que los difiere de los demás clusters (en este caso, la distancia vectorial mínima cercana al *seed*, se estima que es euclidiana).

Por otro lado, los resultados varían en gran medida debido al orden de los datos de entrada. Se entiende que el *seed* escogido es distinto por cada entrada y por lo tanto los clusters difieren en gran medida, por lo tanto es improbable que se repitan exactamente las mismas métricas para cada cluster con cada iteración con entrada randomizada.

# **DBSCAN**

Para esta sección se utilizó una implementación de DBSCAN distribuido por *Weka*, teniendo como entrada los perfiles vectorizados en *SVMLight* *Format* con orden randomizado.  
Lamentablemente no se pudo implementar el código sugerido desde el enunciado ya que se producía errores de referencias y tampoco se pudo obtener un resultado realista desde *Weka* ya que solo producía un cluster para todos los datos (incluso cambiando la función de distancia [euclidiana por defecto] y ajustando el épsilon a 6 unidades para obtener menos perfiles clasificados como “ruido”).

De todas maneras, resultaron gráficos de interés:







En estos gráficos se pueden apreciar ciertos *outliers*, que tan dispersos están los datos dentro de sus mismos conjuntos e incluso como pueden caer dentro del área otros clusters (cafés con naranjas y verdes con azules). Las clases se ordenan de abajo hacia arriba 1 = *Undetermined,* 2 = *Non-USA,* 3 = *World,* 4 = *USA only*.

# **Conclusiones**

¿Qué observa?

En el programa *yakmo* se puede apreciar un algoritmo sensible a la entrada e impredecible en cuanto a salida, pero se obtiene un resultado deseable e interpretable bajo ciertos estándares (nivel de verbosidad solicitado).

La implementación del DBSCAN no se pudo concretar con éxito en su totalidad y los resultados no fueron realistas ni interpretables (excepto por los gráficos, pero estos solo interpretan los datos ingresados y no una salida por clusters). De todas formas el ruido da bastante para discutir, es decir, se puede destacar el bajo nivel de agrupamiento (o la gran magnitud de distancia) entre los datos, la similitud entre datos de distintas clases e incluso los posibles ***outliers***.

¿Hay relación entre los clústeres generado por K-means y DBSCAN?

Existen diversas características que hacen la distinción entre los clústeres generados por K-means y DBSCAN, por ejemplo la densidad utilizada por DBSCAN, la *épsilon*, los *minPoints* y la clasificación de datos como ruido. K-means resulta más simple dado a que posiciona los vectores solo por distancia desde cada cluster.

¿Qué algoritmo es mejor?

DBSCAN es mejor que K-means debido a que no es necesario que se le especifique una cantidad de *clusters* a priori, puede o no tomar en cuenta los datos ruidosos, hace la distinción entre el ruido de los datos reales basándose en medidas que uno le proporciona (épsilon) y su calidad de clusterización se verá incrementada (o reducida) dependiendo de la métrica de distancia utilizada debido a la “maldición de la dimensionalidad”.

En cambio, K-mean solo asigna datos a clusters basándose en las distancias de cada vector hacia cada centroide (por ser simple no significa que sea el mejor, además, el programa *yakmo* solo disponía de una métrica de distancia).