

# Programación Lógica

Fecha: 15/05/2017

Vimos el programa de la materia.

**Tema hablado al ver el programa de la materia:**

La **inteligencia artificial** se basa en programación lógica y las redes neuronales. La programación lógica se utiliza en los juegos, y se puede recalcar a Deep Blue el cual es un simulador de ajedrez. Otra cosa a tomar en cuenta es el Machine Learning a la hora de hablar de la inteligencia artificial.

**Libro que utilizaremos:** Programación Lógica de Julián Pascual.

## Evaluación

Actividades	Unidades a evaluar	Calificación
Parcial 1 (14 junio)	1, 2	20%
Portafolio de ejercicios I	3, 4	15%
Parcial 2 (10 julio)	3, 4	20%
Portafolio de ejercicios 2	3, 4	15%
Proyecto final	Todo el material	30%

La versión de Prolog que usaremos es: SWI-PROLOG

Si le preguntamos a Prolog:

```
[5 is 2 + 3.] → true
```

$[4 \text{ is } 2 + 3.] \rightarrow \text{false}$

El responde según la lógica true o false. Al final de cada enunciado debemos poner un punto (.) para indicar que terminamos el enunciado.

Si la lógica no está creada nos dirá "undefined procedure" pero Si creamos una lógica que diga:

espadre(juan, miguel).  
 espadre(juan, pedro).  
 esmadre(maria, miguel).

Le damos a compilar y a make. Y cuando preguntamos otra vez que si espadre(juan, miguel) nos dá como resultado true.

Si creamos una lógica que diga espadre(jean, oscar) nos dirá que es falso porque no "espadre" existe pero no existe es espadre de esas dos personas, así que nos dirá que es falso.

Podemos crear un predicado al decir lo siguiente:

espadre(juan, X), donde X puede seguir miguel o pedro ya que son las dos personas con la que juan está relacionado en la lógica de arriba.

Esto es posible ya que  $\exists x | \text{espadre}(\text{juan}, x)$  es decir que existe un X tal que x existe en espadre con juan.

## Fecha: 17/05/2017

La mayor parte de la clase se dió a través de una diapositiva, esta estará ubicada en la Plataforma Virtual de Aprendizaje.

Aclaraciones y/o ejemplos dados:

**Consecuencia lógica:** Si la conclusión es verdadera siempre que las premisas son verdaderas.

Lógica de proposiciones:

**Proposición:** Afirmación que tiene un único valor de verdad. En otras palabras, hechos.

**Ejemplos:**

La luna es de queso  $\rightarrow$  Es una proposición  
 Cual es tu edad?  $\rightarrow$  No es una proposición  
 Compra un café  $\rightarrow$  No es una proposición  
 $2 + 5 = 7 \rightarrow$  Es una proposición  
 $x + 2 = 5 \rightarrow$  Como depende del valor de x, no es una proposición

Otro ejemplo:

A = Pedro es dominicano (proposición)

Predicado: ser Dominicano: Dom(X)

Si queremos poner que Pedro es Dominicano ponemos: Dom(Pedro)

María es Dominicana: Dom(Maria)

### Ejemplos de Prolog

espadre(juan, pedro)

espadre = predicado

juan, pedro = argumentos

Los argumentos pueden ser términos o objetos

espadre / 2 → Predicado con dos argumentos

espadre / 1 → Predicado con 1 solo argumento

suma (3, 5)

suma (3, 7 - 2)

### Ejercicio rápido #1

Traducirlo a como si programamos Prolog en Prolog las siguientes proposiciones:

1.- Juan vive en la misma casa que Chucho → mismacasa(juan, chucho) →  
predicado(argumento1, argumento2)

**NOTA:** Los predicados deben de comenzar con minúscula y los argumentos también deben comenzar con minúscula.

2.-  $A + B = C \rightarrow \text{esigual}(\text{suma}(a, b), c) \rightarrow A + B = 7 \rightarrow \text{esigual}(\text{suma}(a, b), 7)$ .

3.-  $f(A) \rightarrow f(A)$

4.- Ana 17 años, Erika 19 años, Julia 18 años → años(ana, 17), años(erika, 19), años(julia, 18)

**NOTA:** En prolog la coma representa el conector lógico "∧" (y). También el punto y coma representa el conector lógico "∨" (o)

5.- Ana, Erika, Julia van a la universidad → univ(ana), univ(erika), univ(julia)

6.- Alguien va a la universidad  $\rightarrow \text{univ}(X)$ , en este caso  $X$  es una variable.

Conectivos lógicos en Prolog

$Y(\wedge) \rightarrow \text{coma } (,)$

$O(v) \rightarrow \text{punto y coma } (;)$

Si entonces( $\rightarrow$ )  $\rightarrow q:p \rightarrow$  Se pone primero el consecuente y después el antecedente.

Negación( $\sim$ )  $\rightarrow \text{not}(p)$ , donde  $p$  es cualquier función, por ejemplo  $\text{not}(\text{hermano}(\text{robin}, \text{juan}))$  donde esto nos dice que Robin y Juan no son hermanos.

Ejercicio rápido #2:

Expresar en forma lógica

1.- Todos los alumnos deben matricularse para asistir a la asignatura PL  $\rightarrow$   
Respuesta:

$\forall x \text{ Alumno}(x) \rightarrow \text{PL}(x)$

#### TAREA:

Investigar lo que es una fórmula abierta y una fórmula cerrada.

Una **función abierta** es una función entre dos espacios topológicos cuando la imagen de un conjunto abierto es un conjunto abierto. Es decir, una función  $f: X \rightarrow Y$  es abierta si para cualquier conjunto abierto  $U$  en  $X$ , la imagen  $f(U)$  es abierta en  $Y$ . Asimismo, una **función cerrada** cumple que la imagen de un conjunto cerrado es un conjunto cerrado.

**Fecha: 19/05/2017**

Teléfono del profesor Moronta  $\rightarrow$  (809)-889-1612

Las tutorías serán los lunes, miércoles y viernes en la tarde.

**ACTIVIDAD #1:** En grupo de 3 personas, conversar durante 15 minutos, identificando un compañero:

Nombre: Ricardo José Rosario

Ciudad: Santiago de los Caballeros

Pasatiempos: Programar, jugar videojuegos, leer, escribir, ver series y hacer ejercicios

Experiencia Laboral: Ninguna

Asignatura Favorita: Programación Aplicada

Profesor Incidente y su Asignatura: Estructura de Datos - Roberto Abreu

Nombre: Jean Melvin Lemoine  
 Ciudad: Santiago de los Caballeros  
 Pasatiempos: Programar, jugar videojuegos, escuchar música y hacer ejercicios  
 Experiencia Laboral: Ninguna  
 Asignatura Favorita: Programación I  
 Profesor Incidente y su Asignatura: Programación I - Daniel Pagés

Nombre: Oscar Dionisio Núñez Siri  
 Ciudad: Santiago de los Caballeros  
 Pasatiempos: Escuchar música, ver anime, leer manga, jugar videojuegos y programar  
 Experiencia Laboral: Ninguna  
 Asignatura Favorita: Programación Aplicada  
 Profesor Incidente y su Asignatura: Base de Datos I - Máximo Pérez

**COSAS A VER:** Cosas para la semana que viene (viernes):

- a) Implementación de bases de conocimientos sobre el documento de Excel que llenamos.
- b) Un cuestionario o quiz de algunas cosas básicas sobre Prolog.

#### Notas para repaso del quiz:

- 1) Todos los comandos que se hagan en Prolog deben terminar en un punto(.). **Ejemplo:** esPato(oscar). ← Se terminó en punto
- 2) Las variables comienzan con letras en mayúsculas o con una rayita abajo, **Ejemplo:** esPato(X) o esPato(\_x).
- 3) Se utiliza el operador "is" para decir que algo es otra cosa. **Ejemplo:** 5 is 2 + 3 o 43 is 43.
- 4) Se utiliza el operador not para negar algo. **Ejemplo:** not(false) devolverá true.
- 5) Los operadores lógicos son: coma(,) para la Y(^), punto y coma para la O(v) y (-) para sí entonces, en el caso de si entonces se pone la conclusión antes del antecedente, es decir q:-p en vez de p:-q.
- 6) halt. → se utiliza para salir del entorno de prolog
- 7) edit(archivo). → se utiliza para invocar al editor de archivos de Prolog.
- 8) consult(archivo). → se utiliza para consultar el fichero, es decir, traerlo a memoria. Otra forma de hacer esto es poner el nombre del archivo entre corchetes. Ejemplo: [patos] ← esto llamará a memoria al archivo patos. Llamar a memoria es poder hacer consultas con ese archivo, es decir, poder preguntar cosas como → esPato(oscar) y que devuelva true.
- 9) help(ayuda) → solicita ayuda al entorno de Prolog.
- 10) make. → consulta los ficheros que cambiaron desde la última consulta.
- 11) listing. → muestra todos los predicados o funciones de la base de conocimiento.
- 12) listing(predicado) → muestra todos los predicados que se hayan hecho de

- este predicado.
- 13) `apropos(palabra)` → busca información sobre la palabra introducida.
  - 14) Para poner una base de conocimiento básica se entra a SWI-Prolog, le damos a File > New, ponemos el nombre del documento y ahí nos abrirá la base de conocimiento y podemos poner cualquier regla que después de ponerla nos dará true. Por ejemplo si no ponemos en la base de conocimiento → `esPato(oscar)`, cuando preguntemos `esPato(oscar)` nos dará true.
  - 15) Si queremos saber cuales son las relaciones entre un valor y otro en una función, podemos poner `esPadre(oscar, X)` → Nos dirá quién es X, es decir cuales son los hijos de Oscar, pero solo dirá uno de los hijos, si queremos que diga todos los hijos debemos de escribir punto y coma(;) para que vayan apareciendo uno a uno.
  - 16) `pwd.` → nos dirá el directorio en el que estamos ahora mismo almacenando los archivos de Prolog.
  - 17) `ls.` → nos dirá los archivos de Prolog (.pl) que hay en ese directorio.
  - 18) `cd('ruta')` → cambia el directorio actual de prolog al puesto dentro de las comillas. **Ejemplo:** `cd('c:/users/oscar/desktop')` y eso cambiará nuestro directorio actual. Algo que debemos notar aquí es que los slashes se ponen inclinados a la derecha no al revés.
  - 19) Los comentarios en prolog se ponen con un signo de porcentaje. **Ejemplo:** `% Esto es un comentario %`.
  - 20) El nombre que tiene una función se llama functor o funtor y la cantidad de parámetros que esta recibe se llama árida. **Ejemplo:** `esPato/1` donde `esPato` es el functor y 1 es la árida.

Fecha: 22/05/2017

La mayor parte de la clase fue explicada a través de una diapositiva la cual fue subida a la Plataforma Virtual de Aprendizaje.

En el primer ejemplo ilustrativo visto en la diapositiva, podemos ver las relaciones de padres y madres.

Algunas relaciones que se pueden sacar de este ejemplo ilustrativo:

- 1.- Antonio es padre de Carlos → `esPadre(antonio, carlos)`.
- 2.- María es madre de Carlos → `esMadre(maria, carlos)`.
- 3.- María es madre de Eva → `esMadre(maria, eva)`

***De estas se pueden inferir que:***

- Los padres de Carlos son Antonio y Maria
- Carlos es hijo de Antonio y María.
- Carlos y Eva son hermanos
- Antonio y María son abuelos de Fernando, Silvia y Emilio.

**PREGUNTA:** Porqué Antonio y María son abuelos?

Porque el padre de Fernando es Carlos y el padre de Carlos es Antonio y así sucesivamente para todos los hijos de Carlos, Elena, Eva y David.

También podríamos decir que por la posición en que se encuentran los hijos en el árbol, es decir que como los hijos de Antonio y María son Carlos y Eva y estos tienen a sus hijos, los hijos de los hijos serán sus nietos y por lo tanto los nietos tienen a sus abuelos que son Antonio.

Para hacer esto en Prolog sería:

esabuelo(S) :- espadre(S, X) , espadre(X, Y).

esabuelo(S) :- espadre(S, X), esmadre(X, Y).

esabuela(S) :- esmadre(S, X), espadre(X, Y).

esabuela(S) :- esmadre(S, X), esmadre(X, Y).

**Otra forma de hacerlo es:**

esprogenitor(antonio, C).

esprogenitor(maria, C).

esabuelx(X) :- esprogenitor(X, Y), esprogenitor(Y, Z).

esabuelo(X) :- esabuelx(X), esHombre(X).

esabuela(X) :- esabuelx(X), esmujer(X)

**Otras consultas que podemos sacar son:**

esabuelo(x, z) :- espadre(x, y), es padre(y, z) → **Esto dirá si tal persona es abuelo de otra, no solo si una persona es abuelo.**

Link del tutorial que vimos sobre algunas cosas: <http://lpn.swi-prolog.org>

Si queremos poner un nombre de un argumento en mayúscula, ponemos la palabra entre comillas → esPadre('Oscar'), ya que si lo ponemos así: esPadre(Oscar) nos dará error.

**EJEMPLO:**

Base de consulta:

woman(mia).

woman(jody).

woman(yolanda).

playsAirGuitar(jody).

party.

Con esto solo probamos que debería de dar cuando lo ponemos en SWI-PROLOG.

### Base de consulta #2:

- 1 - happy(yolanda).
- 2 - listens2Music(mia).
- 3 - listens2Music(yolanda) :- happy(yolanda).
- 4 - playsAirGuitar(mia) :- listens2Music(mia).
- 5 - playsAirGuitar(yolanda) :- listens2Music(yolanda).

La forma de argumento que Prolog utiliza para calcular la respuesta es Modus Ponens.

p  
 $p \rightarrow g$   
 -----  
 g

**Mia toca la guitarra?** → Si, ya que si mia escucha música entonces toca la guitarra.

**Yolanda toca la guitarra** → Si, ya que si yolanda está feliz entonces ella está escuchando musica y si está escuchando música entonces toca la guitarra.

Definición de que una persona esté celosa, si tenemos los siguientes predicados:

loves(vicent, mia)

loves(marsellus, mia)

**estaCeloso(S, Y) :- loves(S, X), loves(Y, X)** → donde S sería vincent, X sería mia y Y sería marsellus.

**Fecha: 24/05/2017**

Progenitor(X, Y) → Busca todos los X, Y tal que X sea un progenitor de Y.

Un ejemplo que podemos recordar de la clase pasada es este:

abuelo(x) :- progenitor(x, y), progenitor(y, z) → Esto es la definición de lo que es un abuelo en Prolog.

Ahora hagamos la representación de **hermanos, tios y primos**:



Definición de hermano:

hermano(x, y) :- progenitor(z, x), progenitor(z, y),  $X \neq Y$

**NOTA:**  $X \neq Y \rightarrow$  significa X sea diferente de Y.

Definición de tío

tio(x, y) :- progenitor(z, y), hermano(z, x).

Otras formas de hacer esto es:

tio(Y, Z) :- hermano(X, Y), esPadre(X, Z).  
 tia(Y, Z) :- hermano(X, Y), esMadre(X, Z).  
  
 tio(X, Y) :- progenitor(Z, Y), hermano(Z, A), esposos(A, X).  
  
 tio(X, Y) :- hermano(X, Z), hijo(Y, Z).

Ejemplo de tío:

tio(alberto, juan) :- progenitor(felix, juan), hermano(felix, alberto).

tio(X, emilio)  $\rightarrow$  Nos dirá cuales son los tíos de emilio.

tio(emilio, X)  $\rightarrow$  Nos dirá los sobrinos de emilio.

Definición de primo:

primo(x, y) :- progenitor(z, x), tio(z, y).

Desarrollar dos reglas legusta/2 para expresar las siguientes situaciones:

De esta base de conocimiento:

legusta(maria, pasta).  
 legusta(juan, carne).  
 legusta(juan, vino).  
 legusta(jose, pasta).  
 legusta(jose, carne).  
 legusta(jose, cerveza).

1.- A Juan le gusta todo lo que le gusta Maria

legusta(juan, legusta(maria, X)).

2.- A Juan le gusta (cae bien) todo aquel al que le gusta la pasta.

legusta(juan, legusta(X, pasta)).

Fecha: 26/05/2017

Hicimos la actividad de subir 6 preguntas al foro, hicimos la actividad del documento de la sesión 2 y hicimos un quiz.

Fecha: 29/05/2017

**Tema de hoy** → Unificación y Búsqueda de pruebas

Recordando de Lógica Computacional:

Regla:  $p(x) \rightarrow Q(x) \wedge y$

Sustitución:

Caso #1:

$x := a \rightarrow x$  ahora es  $a$

$y := b \rightarrow y$  ahora es  $b$

Entonces queda así:

$p(a) \rightarrow Q(a) \wedge b$

Caso #2:

$x := m \wedge n$

$y := b$

Queda así:

$p(m \wedge n) \rightarrow Q(m \wedge n) \wedge b$

Caso #3:

$x := M(a, b)$

$y := R(a) \rightarrow S(n)$

$p(M(a, b)) \rightarrow Q(M(a, b)) \wedge R(a) \rightarrow S(n)$

Prolog:

Un término puede ser:

- constante o átomos
- variables
- términos complejos (funciones, relaciones, etc).

Función de comparación:

Es igual  $a \rightarrow (a, a)$ . o  $a = a$ .  $\rightarrow$  En ambos casos dirá true.

Casos especiales

$2 = 2 \rightarrow \text{true}$

$'2' = 2 \rightarrow \text{falso}$   $\rightarrow$  ya que un átomo no es una constante, constante = 2, átomo = '2'.

$'mia' = mia \rightarrow \text{true}$

$'Mia' = mia \rightarrow \text{false}$

Ejemplo de unificación

?- bread = bread.  
**true.**

?- 'Bread' = bread.  
**false.**

?- 'bread' = bread.  
**true.**

?- Bread = bread.  
 Bread = bread.

?- bread = sausage.  
**false.**

?- food(bread) = bread.  
**false.**

?- food(bread) = X.  
 X = food(bread).

?- food(X) = food(bread).  
 X = bread.

?- food(bread, X) = food(Y, sausage).  
 X = sausage,  
 Y = bread.

?- food(bread, X, beer) = food(Y, sausage, X)  
 |  
**false.**

?- food(bread, X, beer) = food(Y, kahuna\_burger).  
**false.**

?- food(X) = X.  
 X = food(X).

?- meal(food(bread), drink(beer)) = meal(X, Y).  
 X = food(bread),  
 Y = drink(beer).

?- meal(food(bread), X) = meal(X, drink(beer)).  
**false.**

?-

Si decimos  $X = mia$ ,  $X = vincent$ , nos dirá false ya que X no puede tomar dos valores a la vez.

Si decimos  $k(s(g), Y) = k(X, t(k))$ , nos dará que  $X = s(g)$ ,  $Y = t(k)$ .

Si decimos  $loves(X, X) = loves(marcellus, mia)$  nos dirá false ya que no son las mismas X, es decir que marcellus no es mia.

Si decimos  $\text{father}(X) = X \rightarrow$  Esto podría ser verdad si  $X = \text{father}(X)$ , pero si sustituimos la  $X$  nos daría  $\text{father}(\text{father}(X)) = X$  y esto dirá que  $X = \text{father}(\text{father}(X))$  y nos daría  $\text{father}(\text{father}(\text{father}(X))) = X$ , por lo tanto **esto es un ciclo infinito**.

Existe una función llamada **unify\_with\_occurs\_check** la cual recibe dos parametros y nos dice si uno puede ser igual al otro. Un ejemplo de esto sería `unify_with_occurs_check(father(X), X)` nos daría false ya que como se dijo arriba se haría un ciclo infinito y nunca se igualaría.

Para saber si una linea es vertical podemos probar con la siguiente regla:

`vertical(line(point(X, Y), point(X, Z))).`

Si probamos con `vertical(line(point(2, 3), point(2, 5)))`.  $\rightarrow$  Nos da true debido a la unificación, es decir que la variable  $X1 = 2$ , coincide con la variable  $X2 = 2$ .

Esta forma es mucho mejor así que crear una consecuencia, es decir poner muchas cosas dentro de otra ademas de poner una cosa consecuencia de otra.

Si probamos con `horizontal(line(point(2, 3), P))`. Nos dará como resultado que  $P = \text{point}(\_888, 3)$ , lo cual nos dice que  $P$  debe ser un punto que en la primera posición tenga cualquier numero y que en la segunda posición tenga un 3.

```
loves(vincent, mia).
loves(marcellus, mia).
```

**jealous(A,B) :- loves(A, C), loves(B, C).**

Si decimos `jealous(vincent, marcellus)` nos dará true, mientras que si ponemos algo diferente a vincent o marcellus dentro nos dará false ya que no tenemos hechos con mas nombres de personas que aman a otras personas.

```
jealous(X, Y).
1  $\Rightarrow X = Y, Y = \text{vicent};$ 
2  $\Rightarrow X = \text{vicent}, Y = \text{marcellus};$ 
3  $\Rightarrow X = \text{marcellus}, Y = \text{vicent};$ 
4  $\Rightarrow X = Y, Y = \text{marcellus};$ 
false.
```

Para arreglar esto ponemos: `jealous(A, B) :- loves(A, C), loves(B, C), A \= B`  $\rightarrow$  esto solo quita la primera y la ultima respuesta. Y para arreglar esto le ponemos `A @< B` al final ademas de `A \= B`.

Fecha: 31/05/2017

Para construir una oración en Prolog:

Podemos decir en un predicado:

sentence(every, criminal, eats, a, burger), pero esto lo tomamos de que las palabras deben tener la siguiente gramática:

```
word(determiner, every).
word(noun, criminal).
word(verb, eats).
word(determiner, a).
word(noun, burger).
```

Ya que sentence es una regla que dice que:

```
sentence(word1, word2, word3, word4, word5) :- word(determiner, word1),
word(noun, word2), word(verb, word3), word(determiner, word4), word(noun,
word5).
```

Lo cual se traduce al español diciendo lo siguiente → una oración será una oración si la palabra1 es un determinante y la palabra dos es un sustantivo y la palabra3 es un verbo y la palabra4 es un determinante y la palabra5 es un sustantivo.

### Listas en Prolog:

Son elementos que van uno detrás de otro, la cual tiene una cabeza (que es el primer elemento) y la cola (que es el ultimo elemento).

Lista: a, b, c, d, e → donde a es la cabeza, los otros son elementos normales y e es la cola.

Lista en Prolog: [cabeza | lista] → es decir que ponemos quien será la cabeza y después podemos poner otra lista de elementos.

**Ejemplo:**

**$[X|Y] \rightarrow X$  es la cabeza (termino),  $Y$  es el resto de la lista (una lista).**

$[a, [b,c,d,e]] \rightarrow$  Esta es una lista que tiene los elementos a, b, c, d, y e. Pero esto es recursivo ya que b es la cabeza de la segunda lista y c la cabeza de la tercera lista y así sucesivamente.

¿Cómo saber si X es miembro de una lista?

$\text{miembro}(X, [X|Y]). \rightarrow X$  es parte de la lista si es el primer elemento de la lista.

$\text{miembro}(X, [Y|Z]) :- \text{miembro}(X, Z). \rightarrow X$  es miembro del resto de la lista, la cual es Z.

Si probamos en Prolog lo siguiente:

```
?- miembro(pedro, [maria, pedro, juan, ana]).
true
```

Nos dará true porque pedro existe, si le ponemos punto y coma(;) el va a seguir buscando si pedro está en el resto de la lista, hasta que llegue al final de la lista.

Si ponemos  $\text{miembro}(X, [maria, pedro, juan, ana]).$  nos daría:

$X = maria;$

$X = pedro;$

$X = juan;$

$X = ana.$

Es decir que X será cada uno de los elementos de la lista, ya que es una variable.

Para que no nos sala el error  $\rightarrow$  Singleton Variables X, Y o algo así, ponemos las variables que salen en el error como variables mudas con rayita abajo(\_), así:

```
miembro(X, [X|_]).
miembro(X, [_|Z]) :- miembro(X, Z).
```

Y ya dejará de dar el error.

Si queremos concatenar dos listas podemos utilizar lo siguiente:

```
concatenar([], L, L).
concatenar([X|Y], Z, [X|U]) :- concatenar(Y, Z, U).
```

**Ejemplo:**

```
?- concatenar([a, b, c], [d, e, f], X).
X = [a, b, c, d, e, f].
```

**Fecha: 2/06/2017**

Hicimos 3 actividades

- a) Determinar si existe una conexión entre dos aristas.
- b) Determinar si hay una descendencia entre dos familiares o personas normales.
- c) Realizar un documento sobre la formulación de correos y matriculas y la utilización de listas.

**Fecha: 5/06/2017**

Conceptos básicos de una lista en Prolog

$[X | Y] \rightarrow$  Donde X es la cabeza y Y es el resto de la lista.

Vemos que hay recursividad en la función miembro porque esta llama a sí misma en su segunda forma de predicado.

```
miembro(X, [X|_]).
miembro(X, [_|Z]) :- miembro(X, Z).
```

Su forma de parar es cuando las cabezas se hayan desplazado hasta que la cabeza sea el elemento buscado, si esto nunca pasa significa que no se encuentra el elemento en la lista.

Podemos tener las siguientes reglas:

```
piezasde(X, [X]) :- piezabasica(X).
piezasde(X, P) :- emsamblaje(X, unir(Subpiezas)), listapiezasde(Subpiezas, P).
```

Aquí podemos ver que una pieza puede ser simplemente una pieza básica o un emsamblaje de piezas que se guarda en una lista de piezas, la cual se declara así:

```
listapiezasde([], []).
listapiezasde([P|Resto], Total) :-
    piezasde(P, Piezascabeza),
    listapiezasde(Resto, Piezasresto),
    concatenar(Piezascabeza, Piezasresto, Total).
```

Podemos hacer la función factorial de la siguiente manera:

```
fact(0, 1). → Significa que factorial de 0 es 1.
fact(N, Y) :- M is N-1, fact(M, Z), Y is N*Z.
```

Podemos hacer lo siguiente con esta función:

**Prueba #1:**

```
?- fact(5, X).
X = 120 .
```

**Prueba #2:**

```
?- fact(7, X).
X = 5040 .
```

La forma en que se determinan los factoriales es la siguiente:

```
fact(1, X).
```

Por la regla 2 obtenemos  $\rightarrow \text{fact}(1, X) :- N = 1, Y = X$ .

M es 0, Y es  $1 * Z$ .

Y nos quedamos con  $\text{fact}(0, Z)$ . Y por la regla 1 obtenemos que  $z = 1$ , y volvemos hacia atrás y ponemos 1 donde esté la Z. Y nos da como resultado que X es igual a 1 y así es que encuentra el resultado.

Otra forma de hacerlo es:

```
fact(X, Y) :- fact_aux(X, 1, Y).
```

```
fact_aux(0, Y, Y).
```

```
fact_aux(N, Y, Z) :- U is N*Y, M is N-1, fact_aux(M, U, Z).
```

Y nos dá los mismos resultados:

```
?- fact(7, X).
X = 5040
```

Esto funciona de la siguiente manera:

```
fact(0, X)
```



Por la regla  $1 \rightarrow x = 0, y = X$ ,  
 Después `fact_aux(0, 1, X)`. Y por la regla 2 unifica y dice:  
 $Y = 1, Y = X$ , por lo tanto  $X = 1$  y devuelve el 1.

`fact(2, X)`.

Por la regla  $1 \rightarrow X = 2, Y = X$

`fact_aux(2, 1, Y)  $\rightarrow U = 2 * 1 = 2, M = 2 - 1 = 1$ .`


`fact_aux(1, 2, Z)  $\rightarrow U = 1 * 2 = 2, M = 1 - 1 = 0$`

`fact_aux(0, 2, Z)  $\rightarrow Z = 2$`  y como  $Z = 2$ , se reemplaza el valor de  $Y$  con 2 y como  $Y = X$  arriba significa que el factorial de 2 es 2.

**TAREA:** En la PVA nos dejarán la función de invertir una lista. Revisar ese código:

```
concatenar([], L, L).
concatenar([X|Y], Z, [X|U]) :- concatenar(Y, Z, U).
```


Esta nos da este resultado:

 `inversa([x, y, z], X).`  
 $X = [z, y, x]$ 

Y este:

```
inversa(X, Y):- inversa(X, Y, []).
inversa([X|Z], Y, T):- inversa(Z, Y, [X|T]).
inversa([], T, T).
```

Esta nos dá este resultado:

 `inversa([a, b, c, d], X).`  
 $X = [d, c, b, a]$ 

**PREGUNTA:** ¿Cuál es la más eficiente y porqué?

La segunda es más eficiente ya que no utiliza otras funciones que la

ayuden a realizar una parte de la misma función, mientras que la primera utiliza el predicado concatenar y la segunda concatena usando una version diferente del mismo predicado, por lo cual Prolog lo correrá más rápido. Ya que Prolog evalúa todas las formas en la que se cumple una inversa primero, en vez de concatenar.

Fecha: 7/06/2017

Probamos el algoritmo de la inversa de una lista e hicimos un algoritmo que determinara el último elemento de una lista.

**Sacar el último elemento de la lista:**

```
ultimo([X|[]], X).
ultimo([_|Z], X) :- ultimo(Z, X).
```

Ejemplo de unificación

```
?- [Cabeza | Cola] = [juan, come, pan].
Cabeza = juan,
Cola = [come, pan].

?- [alguien escribe | Cola] = [Cabeza | estas, notas].
ERROR: Syntax error: Operator expected
ERROR: [alguien
ERROR: ** here **
ERROR: escribe | Cola] = [Cabeza | estas, notas] .
?- [sevilla, unificar] = [sevilla, [Ciudad]].
false.

?- [nuevo] = [Cabeza, Cola].
false.

?- [nuevo] = [Cabeza | Cola].
Cabeza = nuevo,
Cola = [].
```

Fecha: 9/06/2017

Hicimos los ejercicios de la unidad 4 y 5 de la página Learn Prolog Now.

Fecha: 12/06/2017

Vimos la diapositiva de aritmética y entrada y salida en Prolog. Donde vimos

todos los tipos de operadores en Prolog.

$3 + 1 = 2 + 2 \leftarrow$  el  $"=:"$  evalúa en ambos lados.  $\rightarrow$  true

$3 \text{ is } 2 + 1 \rightarrow$  true

$3 == 2 + 1 \rightarrow$  false

Ejemplo de unificación:

```
?- X is 3 + 4.  
X = 7.
```

```
?- X + Y = 3 + 5.  
X = 3,  
Y = 5.
```

```
?- X = 3 + 5.  
X = 3+5.
```

```
?- X is 8, X is 3 + 5.  
X = 8.
```

```
?- 3 == 1 + 2.  
false.
```

```
?- 3*3 =:= 9.  
true.
```

```
?- X*Y = 9*a.  
X = 9,  
Y = a.
```

```
?- X is 8, X = 3 + 5.  
false.
```

```
?- a = juan, X*Y = 9*a.  
false.
```

```
?- a = 4, X*Y = 9*a.  
false.
```

Da falso porque se está comparando a la letra "a" que es un **átomo** con "juan" que es otro átomo tirará false.

```
?- a = juan.  
false.
```

```
?- a = 4.  
false.
```

```
?- a is 4, X*Y = 9*a.  
false.
```

Pero si A sería una variable si unificaría.

```
?- A = juan, X*Y = 9*A.
A = Y, Y = juan,
X = 9.
```

Si tenemos un predicado:

**precio(X, Y), Y < 1000.** → Nos dará todos los artículos X, que tengan un precio Y menor que 1000.

**precio(X, Y), Y >= 10000, Y <= 40000.** → Nos dará todos los artículos X que tengan un precio Y entre 10,000 y 40,000. Nota, el menor que se pone después del igual en Prolog.

**PROBLEMA:** Dada los siguientes hechos, cree una regla que permita saber el porcentaje de juegos ganados por persona.

```
ganados(juan, 7).
ganados(susana, 6).
ganados(pedro, 2).
ganados(rosa, 5).
ganados(rosa, 10).
jugados(rosa, 10).
jugados(juan, 13).
jugados(pedro, 3).
jugados(susana, 7).
```

**porcentajegananancia(X, Y) :-** ganados(X, G), jugados(X, J), Y is (G/J)\*100.

**Predicado write:**

```
?- write(x).
x
true.

?- write(X).
_2238
true.

?- write('X').
X
true.

?- X = 'texto', write(X).
texto
X = texto.

?- write("X").
X
true.
```

**Otros comandos:**

**nl** → hace un salto de línea.

**tab(X)** → escribe X espacios en blanco.

**display(X)** → muestra lo que está en texto sin evaluar o unificar.

**read(X)** → lee un termino y lo unifica con X.

Otras verlas en la diapositiva.

**Ejemplo:**

**read(U)** → y el usuario pone p(1,2) dirá entonces  $U = p(1, 2)$ . Lo que ponga el usuario debe terminar en un punto.

**read(U), X is U\*2, write('el numero es'), write(X).** → Nos dejará leer un número, lo multiplicará por 2 y lo guardará en X, imprimirá "el numero es" e imprimirá X.

**porcentajeganancia(X, Y) :- ganados(X, G), jugados(X, J), Y is (G/J)\*100, write(Y), write('%').** → Nos saca el porcentaje ganado entre las partidas jugadas y ganadas y lo escribe y escribe el simbolo de porcentaje.

Para el examen:

Saber unificación, recursividad, listas como recorrerlas (funcion miembro), factorial y concatenar listas y algo de lógica para hacer funciones nuevas, unificar variables y atomos y predicados y saber entender que es lo que hacen las funciones que dieron ultimas de read, write y todo eso.

Ejemplo de unificación

?- a \= a.

**false.**

?- 'a' \= a.

**false.**

?- A \= a.

**false.**

?- f(a) \= a.

**true.**

?- f(a) \= a.

**true.**

?- f(a) \= A.

**false.**

?- f(A) \= f(a).

**false.**

?- g(a, B, c)  
| \= g(A, b, c).

**false.**

?- g(a, b, c) \= g(A, C).

**true.**

?- f(X) \= X.

**false.**

Ejemplo de unificación:

?- [a,b,c,d] = [a,[b,c,d]].  
**false.**

?- [a,b,c,d] = [a|[b,c,d]].  
**true.**

?- [a,b,c,d] = [a,b,[c,d]].  
**false.**

?- [a,b,c,d] = [a,b|[c,d]].  
**true.**

?- [a,b,c,d] = [a,b,c,[d]].  
**false.**

?- [a,b,c,d] = [a,b,c|[d]].  
**true.**

?- [a,b,c,d] = [a,b,c,d,[]].  
**false.**

?- [a,b,c,d] = [a,b,c,d|[]].  
**true.**

?- [] = \_.  
**true.**

?- [] = [\_].  
**false.**

?- [] = [\_|[]].  
**false.**

Ejemplo de unificación:

?-  $X = 3 * 4$ .  
 $X = 3 * 4$ .

?-  $X$  is  $3 * 4$ .  
 $X = 12$ .

?-  $4$  is  $X$ .  
**ERROR: Arguments are not sufficiently instantiated**  
**ERROR: In:**  
**ERROR: [8] 4 is \_5520**  
**ERROR: [7] <user>**  
 ?-  $X = Y$ .  
 $X = Y$ .

?-  $3$  is  $1 + 2$ .  
**true.**

?-  $3$  is  $+(1, 2)$ .  
**true.**

?-  $3$  is  $X + 2$ .  
**ERROR: Arguments are not sufficiently instantiated**  
**ERROR: In:**  
**ERROR: [8] 3 is \_6458+2**  
**ERROR: [7] <user>**  
 ?-  $X$  is  $1 + 2$ .  
 $X = 3$ .

?-  $1 + 2$  is  $1 + 2$ .  
**false.**

?- is( $X, +(1, 2)$ ).  
 $X = 3$ .

?-  $3 + 2 = +(3, 2)$ .  
**true.**

?-  $*(7, 5) = 7 * 5$ .  
**true.**

Ejemplo de unificación:

?-  $\ast(7,5) = 7\ast5$ .  
**true.**

?-  $\ast(7,+(3,2)) = 7\ast(3+2)$ .  
**true.**

?-  $\ast(7,(3+2)) = 7\ast(3+2)$ .  
**true.**

?-  $7\ast3+2 = \ast(7,+(3,2))$ .  
**false.**

?-  $\ast(7,(3+2)) = 7\ast(+(3,2))$ .  
**true.**

Fecha: 14/06/2017

No se dió clase → Fecha de examen

Fecha: 16/06/2017

Se dió examen práctico en el laboratorio.

**Segundo parcial:** Fecha: 19/06/2017

**Programación avanzada en Prolog:** Corte explícito y predicados meta-lógicos

**NOTA:** Ver diapositiva de este tema.

1.- Corte explícito:

hombre(mateo).

hombre(luis).

mujer(eva).



`legusta(X, Y) :- hombre(X), mujer(Y).`

`legusta(X, Y) :- mujer(X), hombre(Y).`

`hombre(X) :- padre(Y, X), varon(X).`

`hombre(juan) → false`, ya que el busca `hombre(juan) = hombre(mateo) → falla`.

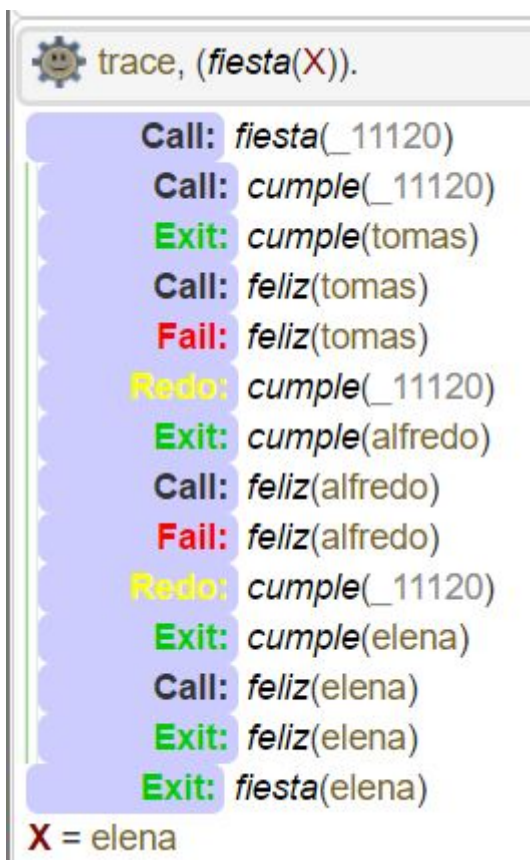
Entonces después hace backtracking y dice `hombre(juan) = hombre(luis)` y falla otra vez y como no hay otro hombre falla totalmente y da false.

`hombre(X).`

**Intenta unificar `hombre(X) = hombre(mateo)`**, entonces `X = mateo`, si le ponemos un punto y coma haciendo backtracking busca el próximo hombre.

```
fiesta(X) :- cumple(X), feliz(X).
cumple(tomas).
cumple(alfredo).
cumple(elena).
feliz(javier).
feliz(monica).
feliz(elena).
```

Trace en Prolog:




```
trace, (fiesta(X)).

Call: fiesta(_11120)
Call: cumple(_11120)
Exit: cumple(tomas)
Call: feliz(tomas)
Fail: feliz(tomas)
Redo: cumple(_11120)
Exit: cumple(alfredo)
Call: feliz(alfredo)
Fail: feliz(alfredo)
Redo: cumple(_11120)
Exit: cumple(elena)
Call: feliz(elena)
Exit: feliz(elena)
Exit: fiesta(elena)
X = elena
```

**Corte** → es un signo de admiración (!) se utiliza para impedir que se inicie el proceso de backtracking. Se utiliza cuando sabemos que no busque más resultados. Es como un break.

Sistema de notas:

```
nota(X, f) :- X < 60.
nota(X, d) :- X >= 60, X < 70.
nota(X, c) :- X >= 70, X < 80.
nota(X, b) :- X >= 80, X < 90.
nota(X, a) :- X >= 90.
```

 **trace**, (nota(65, a)).

**Call:** nota(65, a)

**Call:** 65>=90


**Fail:** 65>=90

**Fail:** nota(65, a)

**false**

Sistema de notas con corte:

```
nota(X, f) :- X < 60, !.
nota(X, d) :- X >= 60, X < 70, !.
nota(X, c) :- X >= 70, X < 80, !.
nota(X, b) :- X >= 80, X < 90, !.
nota(X, a) :- X >= 90, !.
```

 **trace**, (nota(65, X)).

**Call:** nota(65, \_12616)

**Call:** 65<60

**Fail:** 65<60

**Redo:** nota(65, \_12616)

**Call:** 65>=60

**Exit:** 65>=60

**Call:** 65<70

**Exit:** 65<70

**Exit:** nota(65, d)

**X** = d

```

trace, (nota(91, X)).
Call: nota(91, _12616)
Call: 91<60
Fail: 91<60
Redo: nota(91, _12616)
Call: 91>=60
Exit: 91>=60
Call: 91<70
Fail: 91<70
Redo: nota(91, _12616)
Call: 91>=70
Exit: 91>=70
Call: 91<80
Fail: 91<80
Redo: nota(91, _12616)
Call: 91>=80
Exit: 91>=80
Call: 91<90
Fail: 91<90
Redo: nota(91, _12616)
Call: 91>=90
Exit: 91>=90
Exit: nota(91, a)
X = a

```

### Ejercicio:

Pruebe el siguiente código:

$\text{max}(A, B, A) :- A \geq B.$

$\text{max}(A, B, B) :- B \geq A.$

```

max(3, 5, X).
X = 5

```

Trace:

```

trace, (max(3, 5, X)).
Call: max(3, 5, _11172)
Call: 3>=5
Fail: 3>=5
Redo: max(3, 5, _11172)
Call: 5>=3
Exit: 5>=3
Exit: max(3, 5, 5)
X = 5

```

Con corte:

```

max(A, B, A) :- A >= B, !.
max(A, B, B) :- B >= A.

```

En la de abajo no es necesario. Ya que como se pone el corte si A es mayor a B ya cortará ahí y no seguirá. La única diferencia que hace es que corta antes de seguir preguntando para tirar el false si es que tirará false o algo.

Fecha: 21/06/2017

Nos entregaron los exámenes y el profesor atendió a las dudas y reclamaciones.

**Meta-predicados** (fuera de la lógica de su código) : sirven para verificar los tipos de datos.

var(X) → verifica que X es una variable.

atom(X) → verifica que X es un atomo.

integer(X) → verifica si es un entero

atomic(X) → verifica si X es un atomo o número.

float(X) → verifica si X es un número real.

number(X) → verifica si X es un número.

compound(X) → X está instanciada a una estructura.

isList(X) → verifica si X está instanciada a una lista.

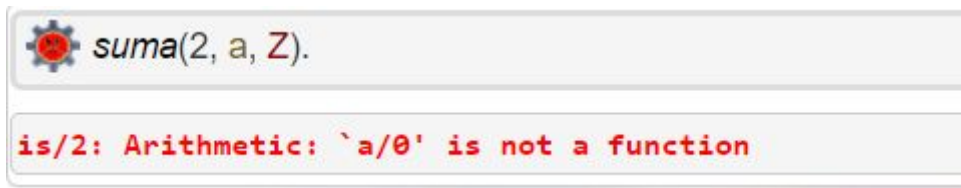
number(A), number(B), A >= B → Dos numeros tales que A sea mayor que B.

Qué pasa si A no es un numero? - Daría false.

Si ponemos por ejemplo:

`suma(X, Y, Z) :- Z is X + Y.`

y decimos `suma(2, a, z)` nos dará el siguiente error:



Pero si le ponemos `suma(X, Y, Z) :- number(X), number(Y), Z is X + Y.` Y lo probamos igualmente con `suma(2, a, Z)` nos dará false además del error ya que `number(Y)` da false, ya que "a" no es un number.

Fecha: 23/06/2017

Corregimos el examen en línea de todos los compañeros.

Fecha: 26/06/2017

**! → Corte → Previene el backtracking.**

**Predicado fail:** siempre produce un fracaso en la satisfacción del objetivo, desencadena el proceso de backtracking, es decir que forza el backtracking para que generen todas las posibles soluciones. Es decir que obliga a que falle, es decir, que de falso.

La combinación de **corte + fail** es igual a una excepción.

Ejemplo:

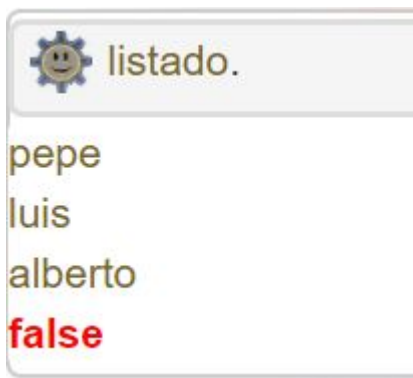
**regla(X) :- p(X), p2(Y), p3(Z), !, fail.** → Si ocurre todo esto entonces falla y sale, es decir que siempre que ocurra todos los predicados es decir que den true todos este terminará y fallará o sea que dará false y saldrá si todo esto pasa.

Otro ejemplo de esto sería:

```
padre(juan, pepe).
padre(juan, luis).
padre(juan, alberto).
listado :- padre(juan, X), write(X), nl, fail.
```

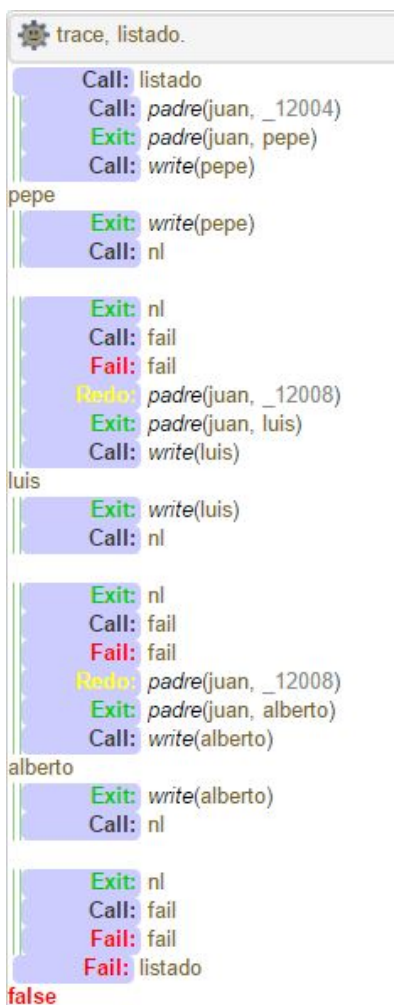
**NOTA:** nl significa salto de línea. Y el fail evita que el salga, ya que simplemente hace que repita.

Da como resultado esto:



```
listado.
pepe
luis
alberto
false
```

Y el trace da esto:



```
trace, listado.
Call: listado
Call: padre(juan, _12004)
Exit: padre(juan, pepe)
Call: write(pepe)
pepe
Exit: write(pepe)
Call: nl
Exit: nl
Call: fail
Fail: fail
Redo: padre(juan, _12008)
Exit: padre(juan, luis)
Call: write(luis)
luis
Exit: write(luis)
Call: nl
Exit: nl
Call: fail
Fail: fail
Redo: padre(juan, _12008)
Exit: padre(juan, alberto)
Call: write(alberto)
alberto
Exit: write(alberto)
Call: nl
Exit: nl
Call: fail
Fail: fail
Fail: listado
false
```

Si tenemos  $p(X) :- b(X), c(X), !, d(X), e(X).$   $\rightarrow b(X)$  y  $c(X)$  solo obtendrá un solo valor debido al corte mientras que  $d(X)$  y  $e(X)$  obtendrán todos los valores que


puede ser la variable x.

Probar el siguiente código con un trace:

```
s(X, Y) :- q(X, Y).
s(0, 0).
q(X, Y) :- i(X), j(Y).
i(1).
i(2).

j(1).
j(2).
j(3).
```

Dará como resultado:

 s(X, Y).

X = Y, Y = 1

X = 1,

Y = 2

X = 1,

Y = 3

X = 2,

Y = 1

X = Y, Y = 2

X = 2,

Y = 3

X = Y, Y = 0

Trace:

```

trace, s(X, Y).

Call: s(_11498, _11504)
Call: q(_11498, _11504)
Call: i(_11498)
Exit: i(1)
Call: j(_11504)
Exit: j(1)
Exit: q(1, 1)
Exit: s(1, 1)
X = Y, Y = 1
Redo: j(_11504)
Exit: j(2)
Exit: q(1, 2)
Exit: s(1, 2)
X = 1,
Y = 2
Redo: j(_11504)
Exit: j(3)
Exit: q(1, 3)
Exit: s(1, 3)
X = 1,
Y = 3
Redo: i(_11498)
Exit: i(2)
Call: j(_11504)
Exit: j(1)
Exit: q(2, 1)
Exit: s(2, 1)
X = 2,
Y = 1
Redo: j(_11504)
Exit: j(2)
Exit: q(2, 2)
Exit: s(2, 2)
X = Y, Y = 2
Redo: j(_11504)
Exit: j(3)
Exit: q(2, 3)
Exit: s(2, 3)
X = 2,
Y = 3
Redo: s(_11498, _11504)
Exit: s(0, 0)
X = Y, Y = 0

```

Si cambiamos el código cambiando la regla q, por el siguiente código:

$q(X, Y) :- i(X), !, j(Y).$

Resultado:

```

s(X, Y).

X = Y, Y = 1
X = 1,
Y = 2
X = 1,
Y = 3
X = Y, Y = 0

```



El trace nos dará lo siguiente:

```

trace, s(X, Y).

Call: s(_11648, _11656)
Call: q(_11648, _11656)
Call: i(_11648)
Exit: i(1)
Call: j(_11656)
Exit: j(1)
Exit: q(1, 1)
Exit: s(1, 1)
X = Y, Y = 1
Redo: j(_11656)
Exit: j(2)
Exit: q(1, 2)
Exit: s(1, 2)
Redo: j(_11656)
Exit: j(3)
Exit: q(1, 3)
Exit: s(1, 3)
Redo: s(_11648, _11656)
Exit: s(0, 0)
X = 1,
Y = 2
X = 1,
Y = 3
X = Y, Y = 0

```

Si cambiamos la regla q a : **q(X, Y) :- i(X), j(Y), !.**

Nos dará como resultado:

```

s(X, Y).

X = Y, Y = 1
X = Y, Y = 0

```

Si ponemos el corte al principio no pasa nada ya que el corta sin tener ningún valor X o Y en i y j.

Si tenemos el siguiente código:

```

enjoys(vincent,X) :- big_kahuna_burger(X),!,fail.
enjoys(vincent,X) :- burger(X).
burger(X) :- big_mac(X).

```

```

burger(X) :- big_kahuna_burger(X).
burger(X) :- whopper(X).
big_mac(a).
big_kahuna_burger(b).
big_mac(c).
whopper(d).

```

Preguntar:

enjoys(vincent, b).

```

⚙️ enjoys(vincent, b).
false

```

Trace:

```

⚙️ trace, enjoys(vincent, b).
Call: enjoys(vincent, b)
Call: big_kahuna_burger(b)
Exit: big_kahuna_burger(b)
Call: fail
Fail: fail
Fail: enjoys(vincent, b)
false

```

enjoys(vincent, c).

```

⚙️ enjoys(vincent, c).
true
false

```

Trace:

```

trace, enjoys(vincent, c).
Call: enjoys(vincent, c)
Call: big_kahuna_burger(c)
Fail: big_kahuna_burger(c)
Redo: enjoys(vincent, c)
Call: burger(c)
Call: big_mac(c)
Exit: big_mac(c)
Exit: burger(c)
Exit: enjoys(vincent, c)
true
Redo: burger(c)
Call: big_kahuna_burger(c)
Fail: big_kahuna_burger(c)
Redo: burger(c)
Call: whopper(c)
Fail: whopper(c)
Fail: burger(c)
Fail: enjoys(vincent, c)
false

```

**IMPORTANTE:** Cuando se pone corte con fallo debe estar arriba de las otras reglas. En este caso es para decir, no le gusta la hamburguesa X y todas las demás si.

`enjoys(vincent, X) →` devuelve false. El corte con fallo solo sirve para preguntas puntuales.

Otra forma de poner el enjoys es:

**`enjoys(vincent, X) :- burger(X), \+ big_kahuna_burger(X).`** → Esta regla dice: a vincent le gusta todas esas hamburguesas menos esta.

Y cuando preguntamos `enjoys(vincent, X)` nos dará como resultado:

```

enjoys(vincent, X).
X = a
X = c
X = d

```

Trace de esta regla:

```

🔍 trace, enjoys(vincent, X).

Call: enjoys(vincent, _11668)
Call: burger(_11668)
Call: big_mac(_11668)
Exit: big_mac(a)
Exit: burger(a)
Call: big_kahuna_burger(a)
Fail: big_kahuna_burger(a)
Redo: enjoys(vincent, a)
Exit: enjoys(vincent, a)

X = a
Redo: big_mac(_11668)
Exit: big_mac(c)
Exit: burger(c)
Call: big_kahuna_burger(c)
Fail: big_kahuna_burger(c)
Redo: enjoys(vincent, c)
Exit: enjoys(vincent, c)

X = c
Redo: burger(_11668)
Call: big_kahuna_burger(_11668)
Exit: big_kahuna_burger(b)
Exit: burger(b)
Call: big_kahuna_burger(b)
Exit: big_kahuna_burger(b)
Redo: burger(_11668)
Call: whopper(_11668)
Exit: whopper(d)
Exit: burger(d)
Call: big_kahuna_burger(d)
Fail: big_kahuna_burger(d)
Redo: enjoys(vincent, d)
Exit: enjoys(vincent, d)

X = d

```

¿Cuál es la diferencia entre este código:

$p :- a, b.$

$p :- \backslash + a, c.$

Y este código?

$p :- a, !, b.$

$p :- c.$


Que las excepciones se dan antes en el primero y en el segundo se dan después.

Si tenemos el siguiente código que pasaría si le preguntamos estas preguntas?

$p(1).$


$p(2) :- !.$

$p(3).$

  $p(X).$

$X = 1$

$X = 2$

  $p(X), p(Y).$

$X = Y, Y = 1$


$X = 1,$

$Y = 2$

$X = 2,$

$Y = 1$

$X = Y, Y = 2$


  $p(X), !, p(Y).$

$X = Y, Y = 1$

$X = 1,$

$Y = 2$

Trace:

  $\text{trace}, p(X).$

**Call:**  $p\_10952)$

**Exit:**  $p(1)$

$X = 1$

**Redo:**  $p\_10952)$

**Exit:**  $p(2)$

$X = 2$

```

trace, p(X), !, p(Y).

Call: p(_11628)
Exit: p(1)
Call: p(_11636)
Exit: p(1)
X = Y, Y = 1
Redo: p(_11636)
Exit: p(2)
X = 1,
Y = 2

```

Fecha: 28/06/2017

**Combinación de corte + fallo** → `!, fail`.

**Otra forma de expresarlo** → `\+`.

`vuela(X) :- pinguino(X), !, fail.`

`vuela(X) :- pajaro(X).`

Este código de arriba dice que todos los pajaros vuelan excepto los pinguinos.

Por lo tanto si tenemos:

`pinguino(p).`

`canario(c).`

`pajaro(p).`

`pajaro(c).`

Si preguntamos

`vuela(c).` → Dará **true**.

```

trace, vuela(p).

Call: vuela(p)
Call: pinguino(p)
Exit: pinguino(p)
Call: fail
Fail: fail
Fail: vuela(p)
false

```

Si le quitamos el corte a `vuela(X)` pasa lo siguiente:

`vuela(X) :- pinguino(X), fail.`



```

trace, vuela(p).
Call: vuela(p)
Call: pinguino(p)
Exit: pinguino(p)
Call: fail
Fail: fail
Redo: vuela(p)
Call: pajaro(p)
Exit: pajaro(p)
Exit: vuela(p)
true

```

Da true, y está mal debido a que `p` es un pinguino y no debería decir que vuela.

Pero esto lo podemos cambiar a una versión más simple del predicado:

**`vuela(X) :- pajaro(X), \+ pinguino(X).`** → Que dice igualmente todos los pajaros vuelan pero si es pinguino además de pajaro debe fallar. Debe ponerse siempre, lo general primero y la excepción después.

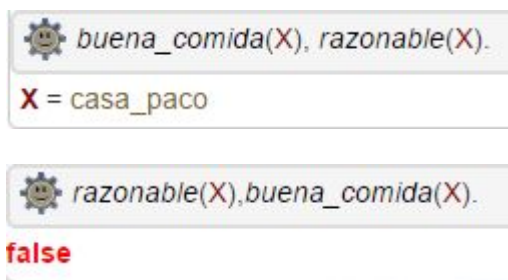
Si tenemos lo siguiente:

```

buena_comida(el_meson).
buena_comida(casa_paco).
caro(el_meson).
razonable(Restaurante) :- not(caro(Restaurante)).

```

Y hacemos las siguientes preguntas, dará como resultado:



```

buena_comida(X), razonable(X).
X = casa_paco

razonable(X), buena_comida(X).
false

```

La segunda pregunta da falso ya que en la primera el busca que buena\_comida la `X` es `casa_paco` y el sabe que es razonable pero en la segunda da falso debido a que `razonable(X)` no va a encontrar a nadie.

Crear los predicados `soltero/1` y `sinhijos/1`.

`hombre(juan).`

hombre(carlos).

mujer(maria).

mujer(laura).

padre(juan, carlos).

madre(laura, maria).

madre(laura, carlos).

esposo(juan, laura).

esposo(laura, juan).

soltero(X) :- hombre(X), not(esposo(X, \_)) ; mujer(X), not(esposo(X, \_)).

sinhijos(X) :- hombre(X), not(padre(X, \_)) ; mujer(X), not(madre(X, \_)).

### Pruebas:

 *sinhijos(carlos).*


true

 *soltero(carlos).*

true

 *soltero(juan).*


false

 *sinhijos(X).*

X = carlos

X = maria

false

 *soltero(X).*

X = carlos

X = maria

false



Fecha: 30/06/2017

Hicimos el ejercicio del documento de Rivero y comenzamos a hacer el algoritmo de Dijkstra en Prolog con lo que teníamos de las conexiones.

Fecha: 3/07/2017

**IMPORTANTE:** El examen se mueve al miércoles 12 de julio. Hay que confirmar el próximo miércoles.

### Manipulación de base de datos y recolección de soluciones

listing. → Saca los comandos puestos en memoria, tanto los hechos como las reglas, es decir que ve el contenido de la base de datos de Prolog.

Insertar un dato en la base de datos → assert(dato) → Ej: assert(happy(mia)), también se puede con reglas → assert(hermano(X, Y) :- padre(Z, X), padre(Z, Y)).

El assert dará true significando que agregó el dato a la base de datos de Prolog.

Para eliminar hechos o reglas de la base de datos es:

retract(hecho\_o\_regla) → retract(happy(mia)).

Si queremos borrar todos los que sean happy seria retract(happy(X)).

assertz → inserta al final.

asserta → inserta al principio,

assert → inserta por ahí.


:- dynamic lookup/3 → crea una función que se llame lookup que no se sabe cuál es el cuerpo de pero se sabe que está definido y pasará algo ahí.

Si tenemos el siguiente código, y lo probamos con los números 3 y 5 dará lo siguiente:

```
:- dynamic dato/3.
add_and_square(X, Y, Res) :- dato(X, Y, Res), !.


add_and_square(X, Y, Res) :-
    Res is (X + Y) * (X + Y),
    assert(dato(X, Y, Res)).
```

Ejercicio 11.2 y 11.3 de Learn Prolog Now:

 `add_and_square(3, 5, X).`  
**X = 64**

Y lo que hace es  $(3 + 5) * (3 + 5) = 8 * 8 = 64$ .


Si le damos un dato que no haya sido calculado como:

 `add_and_square(3, 3, 81).`  
**false**

Nos dará false debido a que no ha sido guardado el dato de que  $3*3 + 3*3$  es 81.

`retractall(dato(_,_,_))` → nos borrará todos los datos que tenemos. En diferencia al `retract(dato(X, Y, Z))` este no pregunta uno por uno como el de las variables que si queremos borrar ese dato.

`findall` → Nos muestra todas las soluciones de una regla en una lista. Se usa así → `findall(X, descend(martha, X), Z).` → Encuentra cuales son todas las descendencias de martha y guardalo en una lista. Se puede usar el `findall` junto así `findall((X, Y), descend(X, Y), Z).`

 `findall(X, descend(Y, X), Z).`  
**Z = [charlotte, caroline, laura, rose, caroline, laura, rose, laura, rose, rose]**


`bagof` funciona igual que el `findall` pero que lo muestra en grupo. osea `bagof(Child, descend(Mother, Child), List).`

 `bagof(X, descend(Y, X), Z).`  
**Y = caroline,**  
**Z = [laura, rose]**  
**Y = charlotte,**  
**Z = [caroline, laura, rose]**  
**Y = laura,**  
**Z = [rose]**  
**Y = martha,**  
**Z = [charlotte, caroline, laura, rose]**

Si hacemos `findlall(List, bagof(Child, descend(Mother, Child), List), Z).` Te hace una lista de lista con cada madre.

`setof(X, age(X, Y), Out)` → Saca la edad que tiene ciertas personas y te pone la lista de esas personas, en este caso saca la X que es el nombre de la persona que tiene una edad Y. Si ponemos `setof(X, Y^age(X, Y), Out)` nos dará la lista entera de todos los nombres sin ubicar por edad.


```

 setof(X, age(X, Y), Out).
Out = [hey, hola],
Y = 1
Out = [hola2],
Y = 3

```

---


```

 setof(X, age(X, 1), Out).
Out = [hey, hola]

```

Si hacemos setof con las edades o sea con los numeros, la organizará y quitará las repeticiones:


```

 setof(Y, X^age(X, Y), Out).
Out = [1, 3]

```

Esto saca los nombres de las personas:

```

 setof(X, Y^age(X, Y), Z).
Z = [hey, hola, hola2]

```

La base de conocimiento usada para sacar estos resultados fue:

```

age(hola, 1).
age(hey, 1).
age(hola2, 3).

```

### GRAMATICA EN PROLOG:

Podemos poner la gramática en Prolog de la siguiente manera:

```
s(Z):- np(X), vp(Y), append(X,Y,Z).
```

```
np(Z):- det(X), n(Y), append(X,Y,Z).
```

```
vp(Z):- v(X), np(Y), append(X,Y,Z).
```

```
vp(Z):- v(Z).
```

```
det([the]).
```

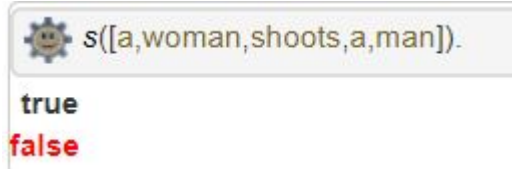
```
det([a]).
```

```
n([woman]).
```

```
n([man]).
```

v([shoots]).

Y probamos con:



Nos dará true ya que cada regla tiene su forma de verificar, es decir s significa sentence y np significa parte no verbal y vp parte verbal, y det significa determinando y n sustantivo y v verbo.

### 11.2 - Learn Prolog Now:

```
q(blob,blug).
q(blob,blag).
q(blob,blig).
q(blaf,blag).
q(dang,dong).
q(dang,blug).
q(flaf,blob).
```

findall(X,q(blob,X),List). → List = [blug, blag, blig].

findall(X,q(X,blug),List). → List = [blob, dang].

findall(X,q(X,Y),List). → List = [blob, blob, blob, blaf, dang, dang, flab].

```
bagof(X,q(X,Y),List). →
List = [blob, blaf],
Y = blag
List = [blob],
Y = blig
List = [flaf],
Y = blob
List = [blob, dang],
Y = blug
List = [dang],
Y = dong
```

setof(X,Y^q(X,Y),List). → List = [blaf, blob, dang, flab].

### 11.3 - Learn Prolog Now:

sigma(X, \_) :- X < 1, !, fail. → **Para evitar que explote con valores de X valores a 0.**

sigma(1, 1) :- !.

sigma(N, X) :- M is N-1, sigma(M, X2), X is X2 + N.

**Prueba:**

Lo que hace es sumar todos los numeros del 4 al 1 =  $4 + 3 + 2 + 1 = 10$ .

**Fecha: 5/07/2017**

Prolog para hoy

Avisos: Fecha de examen, %'s etc..

Actividad: Rally de programación

→ Realice el código de la función sigma del tutorial LPN de manera individual.

Para ganar necesita:

→ Que alguien verifique que su solución es válida.

→ Revisar 3 propuestas de otros 3 compañeros que no hayan trabajado con usted en esta actividad.

```
:- dynamic fact/2.
```

```
sigma(X, Y) :- fact(X, Y), !.
```

```
sigma(X, _) :- X < 0, !, fail.
```

```
sigma(0, 0) :- !.
```

```
sigma(N, X) :- M is N-1, sigma(M, X2), X is X2 + N, assert(fact(N, X)).
```

Personas a la que le corregí:

Adonis, Melissa, Bryan, Cesar, Carlos, Juan Thomas, Jhon y Emilio.

**Fecha: 7/07/2017**

**Nota:** Todavía no se ha dado esta clase, este contenido será actualizado en el momento en que se dé.