PROLOG

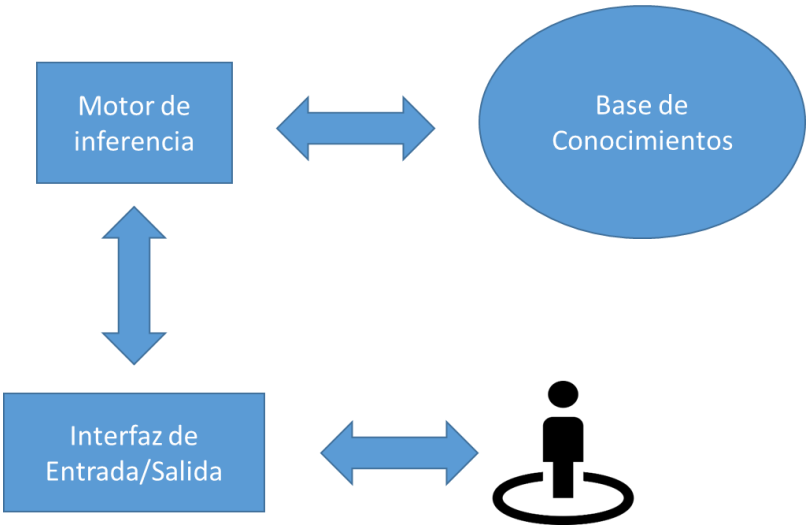
El Prolog es un lenguaje que implementa el paradigma de la programación lógica. Prolog significa "Programación en lógica". Fue creado en la universidad de Provenza (Marsella).

Es bastante diferente de los lenguajes que en general estamos acostumbrados a utilizar que suelen ser orientados a objetos o procedurales.

En los lenguajes tradicionales normalmente, dado un algoritmo, programar consiste en codificar un algoritmo que permite llegar a la solución de un problema. Cuando el programador escribe el programa está definiendo el camino para llegar a la solución del problema. El prolog el programador solo propone el objetivo al que quiere llegar y es prolog quien determina el camino para llegar a dicho objetivo.

Modo de trabajo de prolog

El prolog está formado por 3 bloques, de un modo similar a un sistema experto, el prolog está formado por una base de conocimientos, un motor de inferencia y una interfaz de entrada y salida que se comunica con el usuario.



La base de conocimientos es una base de datos en memoria. El motor de inferencia es un conjunto de algoritmos, incluido s en prolog , que permiten recorrer esa base de conocimientos y obtener

las soluciones los problemas que se la plantean al prolog a través de la interfaz de entrada y salida.

Programar en prolog consiste, en realidad, en incorporar conocimiento a la base.

Prolog genérico

Debido a que existen diversas versiones de prolog. Primero hablaremos de prolog en general, un pseudocódigo de programación lógica.

El lenguaje Prolog implementa el paradigma de la programación lógica.

La desventaja de este lenguaje, respecto de otros, es que el código generado por Prolog no va a ser muy eficiente.
La ventaja es que los tiempos de desarrollo son menores y el fuente es más pequeño, en comparación de C o JAVA por ejemplo.

Para comenzar tomemos un ejemplo común en todo curso de prolog, que es el ejemplo que consiste en incorporar a la base de conocimientos información acerca personas y la relación de parentesco entre ellas, para realizar consultas sobre esta información.

Por ejemplo queremos incorpora a la base de conocimientos de prolog que "Juan es padre de Pedro", deberíamos incorporar a la base un conocimiento como el siguiente

Juan es padre de Pedro

"Juan es padre de pedro" es una proposición lógica. En general puede ser verdadera o falsa. La incorporamos a la base conocimientos del prolog, porque en el problema que queremos modelizar es verdadera.

Para el prolog, todo lo que está en la Base de Conocimientos es verdadero.

Con esta infinitamente pequeña base de conocimientos que tenemos incorporada al prolog ya le podemos hacer algunas consultas elementales.

Por ejemplo, le preguntamos al Prolog

¿Es Juan padre de Pedro?

Entonces Prolog revisa la base de conocimientos y como la encuentra devuelve como respuesta verdadero ("SI").

Ahora supongamos que el preguntamos

¿Es *Juan padre de María?*

La respuesta a esta pregunta podría ser “no lo sé”, porque en la base de conocimientos no hay nada que diga que sí, pero tampoco hay nada que diga que no. Sin embargo para prolog la base de conocimientos es como el conjunto universal, todo lo que está en ella es verdadero lo que no está es falso.

Entonces en este caso devuelve falso, ya que no lo encuentra en la base de conocimientos.

¿Que pasaría si le preguntásemos?

¿Es Juan tío de Pedro?

En este caso como no encuentra la relación definida en la base devuelve Error. En el caso anterior no encuentra al elemento, en este caso no encuentra la relación.

“Tío de “ y “Padre de“ son relaciones, “Juan” , “Pedro ” son elementos

Ahora hagámosle otro tipo de consulta al prolog

¿Cuáles son los X / Juan es padre de X?

En este caso tiene una variable sin valor, en este caso no solo le estamos pidiendo que responda verdadero o falso, sino que además en el caso de que la respuesta sea verdadero, también me devuelve el valor, o los valores , para esa variable tal que hagan que la respuesta sea verdadero. Entonces en este ejemplo si encuentra algún hijo me devuelve Verdadero(“Si”) y los nombres de todos los hijos encontrados, de los contrario me devuelve falso (“No”).

En caso de que la pregunta incluya variables entonces, si es verdadero me devuelve los posibles valores resultantes.

Continuando con el ejemplo definimos nuevas relaciones:

Pedro es hijo de Juan

Podría agregar esta relación. Pero si tengo muchos ***es padre de*** y defino ***es hijo de*** de esta forma, estaría duplicando el contenido de la base de conocimientos. No estaría agregando información nueva, sino que solo estaría aumentando el tamaño de la base de conocimientos.

Podemos buscar una forma genérica para definir ***es hijo de*** en función ***es padre de***

$\forall$ ***X,Y si X es hijo de Y*** $\Leftarrow$ ***Y es padre de X***

Es más prolijo definir todas las relaciones por ejemplo ***es padre de*** con los nombres de las personas y luego definir las demás elaciones en forma genérica.

X es abuelo de Y $\Leftarrow$ ***X es padre de Z*** $\wedge$ ***Z es padre de Y***

X es hermano de Y $\Leftarrow$ ***Z es padre de X*** $\wedge$ ***Z es padre de Y***

$\forall$ X,Y no se escribe, sino que se asume que esta implícito.

Supongamos que hacemos la siguiente consulta.

¿Es Pedro hermano de Pedro?

La repuesta va a ser ***"SI"***

Si bien en la vida real puede haber dos personas con el mismo nombre, en este ejemplo vamos a tomar el nombre como el identificador unívoco de cada persona, o sea no puede haber dos personas con el mismo nombre.

Por lo tanto la respuesta es errónea, porque una persona no es hermano de si mismo.

Para asegurarnos que no tome a una persona como hermano de sí mismo, agregamos esta condición:

X es hermano de Y $\Leftarrow$ ***Z es padre de X*** $\wedge$ ***Z es padre de Y*** $\wedge$ ***X<>Y***

Ahora, pata seguir completando este ejemplo, vamos a definir la relación antecesor.

Llamamos antecesora a alguien que está por encima en el árbol genealógico, o sea ancestro.

Primero podríamos poner

X antecesor Y $\Leftarrow$ ***X es padre de Y***

Es correcto, pero no es suficiente para definir en forma completa antecesor.

Podríamos agregar

X antecesor Y $\Leftarrow$ ***X es abuelo de Y***

Sigue siendo correcto, pero no suficiente.

Podría seguir agregando ***bisabuelo***, etc. Pero así nunca voy terminar de cubrir todos los casos.

La solución consiste en hacerlo en forma recursiva.

X antecesor Y $\Leftarrow$ ***X padre de Z*** $\wedge$ ***Z antecesor de Y***

Si pongo ***Z es hijo de X***, en lugar de ***X es padre de Z***, primero tiene que resolver la del hijo y luego la del padre y recién ahí es cierta. Sino resuelve una sola. La relación del abuelo podría sacarla, pero si no la saco tiene más información y puede resolver más rápido.

Finalmente la definición completa de antecesor seria

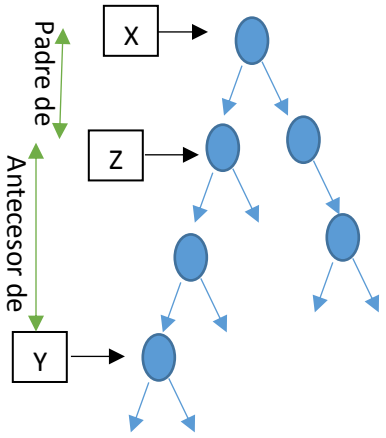
X antecesor Y $\Leftarrow$ ***X es padre de Y*** $\vee$ ***X padre de Z*** $\wedge$ ***Z antecesor de Y***

También se puede escribir de esta forma

X antecesor Y $\Leftarrow$ ***X es padre de Y***

X antecesor Y $\Leftarrow$ ***X padre de Z*** $\wedge$ ***Z antecesor de Y***

Estas dos líneas se pueden interpretar como unidas por un ***or***, porque la primera sola es verdadera, la segunda sola también es verdadera, el conjunto de ambas también es verdadero. Esta combinación es la tabla de verdad del or.



En la Base de Conocimientos de Prolog, mas redundancia implica que la base será más grande y tendrá mayor velocidad en la resolución de problemas. Por el contrario menos redundancia implica la base más chica y menor velocidad.

Al igual que en otros lenguajes recursivos todo función recursiva debe tener una condición de fin, o sea e tiene que cumplir que por lo menos para un caso la función no se invoque a sí misma.

Adema la sucesión de invocaciones recursivas de be converger a la condición de fin. ***Antecesor de*** cumple ambas condiciones.

En muchos lenguajes de programación la recursividad es algo opcional, poco usado en la vida real, en prolog es prácticamente indispensable, porque al no existir sentencias iterativas como while o for, la única forma de realizar una iteración es mediante la recursividad.

El motor de inferencia del prolog recorre la base de conocimientos desde el comienzo hasta el fin, o sea en el orden en que fueron incorporados los conocimientos.

Un programa escrito en prolog, tiene menos líneas de código que uno escrito en un lenguaje procedural u objetos, como C o JAVA, por ejemplo.

Hechos y Reglas

Retomemos los ejemplos anteriores y reescribamos las dos siguientes líneas.

Juan es padre de Pedro

A es hijo de B $\Leftarrow$ ***B es padre A***

Notamos que en la primer línea se trabaja con constantes, en la segunda se trabaja con variables.

Además en la primera no tenemos un implica mientras que en la segunda si.

A las líneas que tiene un formato similar a la primera se las llama hechos. A las que tienen un formato como la segunda se las llama reglas. Lo que realmente diferencia al hecho de la regla es la presencia o no del implica, pero en la gran mayoría de los casos en las reglas se trabaja con variables y en los hechos con constantes.

Juan es padre de Pedro (es un hecho)

A es hijo de B $\Leftarrow$ ***B es padre A*** (es una regla)

Tanto a los hechos como a las reglas se los llama cláusulas

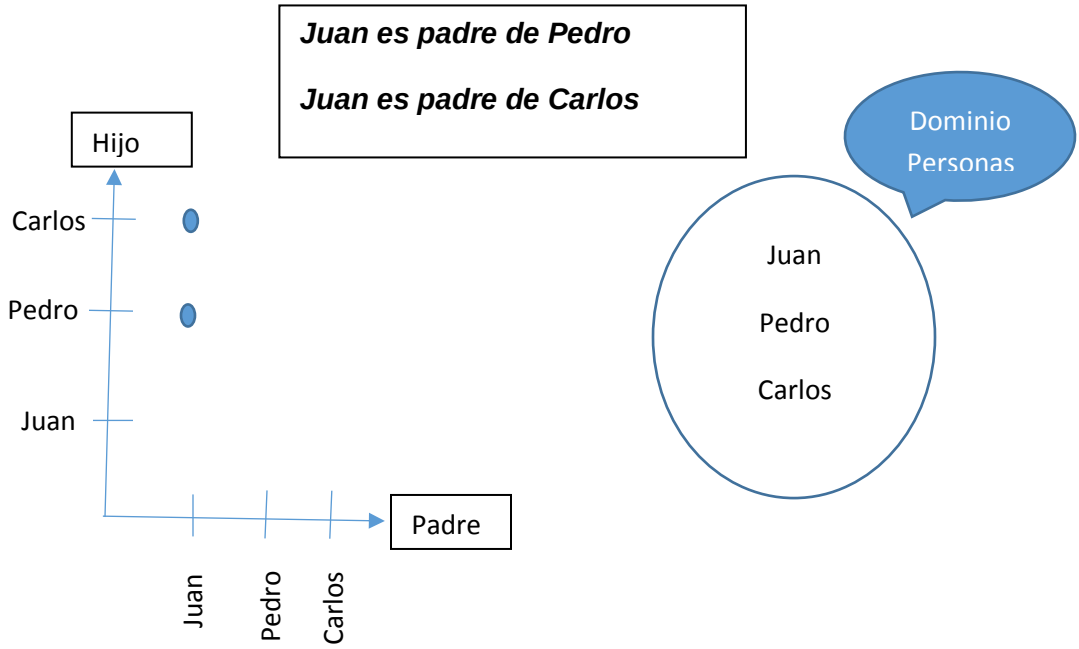
Todas las relaciones que hemos definido se pueden interpretar desde un punto de vista de programación como funciones booleanas (porque retornan verdadero o falso) y que se aplican a distintos dominios (en este ejemplo, hasta ahora, todos los dominios son personas). A estas funciones booleanas se las llama ***predicados.***

Las clausulas nos permiten definir a los predicados. O sea tanto los hechos como las reglas nos permiten definir a los predicados, pero los hechos además definen los dominios.

Por ejemplo al escribir

Juan es padre de Pedro

Se está definiendo parte del predicado ***padre de*** y además se está diciendo que ***Juan*** y ***Pedro*** son parte del dominio persona.



Cuando se define un predicado usando hechos se lo está definiendo punto a punto. Cada hecho representa un punto de la función. Cuando se lo define utilizando reglas se lo define mediante una condición genérica.

Definir un predicado usando hechos es como definir un conjunto por extensión. Definir un predicado usando reglas es como definir un conjunto por abstracción (comprensión).

Todo conocimiento que se coloque en la Base de Conocimientos se lo llama clausula (tanto hechos como reglas). Entonces el conjunto de cláusulas permite definir predicados. Las clausulas pueden ser hechos o reglas. Un conjunto de cláusulas me definen un predicado. Los predicados son funciones boolenas.

Un hecho es en realidad es un caso particular de una regla. Es un regla en la que el implicando es siempre verdadero.

Juan es padre de Pedro $\Leftarrow$ ***verdadero***

El formato genérico de un clausula es

$$p_n \Leftarrow \bigwedge p_i$$

El formato genérico de definición de un predicado es

$$p_n \Leftarrow \bigvee \bigwedge p_i$$

Instalación de SWI-Prolog

La versión de prolog que vamos a utilizar en el SWI-Prolog

Se puede descargar de

https://www.swi-prolog.org/Download.html

Eligiendo la versión adecuada para cada sistema operativo del ultimo Stable reléase. Es conveniente agregar el path de SWI Prolog al path del SO.

Para utilizarlo se debe entrar a SWI - Prolog

Para escribir un programa (definir la base de conocimientos) debe ir a ***File new y luego e***legir la carpeta y el nombre de archivo del programa escribir.

Se abrirá otra ventana con el editor para escribir el programa.

En el programa se escriben las clausulas (se definen los predicados)

Cada vez que se desea grabar se deberá ir a ***File -*** S***ave buffer***

En la primer ventana es donde se escribe la consulta que se le va a realizar al prolog

Para poder acceder a los predicados definidos en la segunda ventana se debe ir ***File -Consult*** o ***File - reload modified files.***

Cada vez que se modifique algo en la segunda ventana (predicados) , para la primer ventana tome los cambios se debe volver ir a ***File -Consult*** o ***File -.*** Con ***reload modified files*** se recargan los cambios hechos en el archivo asociado previamente y con Consult se elige un archivo para leer.

Para abrir y editar un archivo ya grabado debe ir a File - Edit

Código en SWI - Prolog

Volviendo al ejemplo anterior

En la ventana de predicados se puede escribir

padre_de(juan, pedro).
padre_de(juan, carlos).

Al utilizar la sintaxis de SWI-Prolog se debe escribir primero el nombre del predicado y entre paréntesis sus argumentos. Toda clausula finaliza con un punto. El prolog es case-sensitive. Las cadenas de caracteres, si comienzan con minúscula se pueden escribir sin delimitadores.

Una regla se define de la siguiente forma

hijo_de(X,Y):- padre_de(Y,X).

Implica se escribe como ***:-*** .
Las Variables van en mayúsculas. Si la cadena empieza con una letra minúscula no hace falta colocar delimitadores. Pero Juan es interpretado como variable, por lo tanto si se quiere colocar el nombre con mayúscula, entonces se deben usar delimitadores.

Si se definen varias cláusulas (más de una) para un predicado, estas deben estar agrupadas, sino prolog emite un warning. El Prolog hace el OR de esas cláusulas para definir si la invocación al predicado da verdadero o falso.

abuelo_de(X,Y) :- padre_de(X,Z), padre_de(Z,Y).

La sintaxis para en and es la coma "***,".***

En la primer ventana se coloca la consulta que se le hace al prolog, utilizando la misma sintaxis que en la definición de predicados. Siempre termina con un punto. Por ejemplo

padre_de("juan", pedro).

Va a retornar ***true***

padre_de("juan", luis).

Va a retornar ***false***

Estas consultas, que son las mas simples, se responden solo con ***true*** o ***false***. Para realizar consultas que retornen valores se deben utilizar variables. Por ejemplo

padre_de(juan,Y).

Va a retornar ***true***

Y = pedro .

Otra consulta.

padre_de(X,Y).

Va a retornar ***true***

X = juan,

Y = pedro .

Si la respuesta es true, retorna los valores para la o las variables tal que hagan verdadero al predicado.

En prolog las variables siempre comienzan con una letra mayúscula.

Operadores aritméticos y relacionales

Estos son los operadores aritméticos de prolog.

+
-
*
/
mod (resto)
div (cociente)

Estos son los operadores relaciones de prolog.

>
<
>=
=<
== (Compara)

=\=(Compara)
= (unifica)
is (asigna/compara)

Ejemplo de uso de los operadores:

Como ejemplo se define el siguiente predicado, muy elemental, que solo suma los números de los dos primeros parámetros y deja el resultado en el tercero. En prolog todas las funciones son booleanas por lo tanto los resultados se deben retornar mediante los parámetros.

sumar(X, Y, Z) :- Z is X +Y.

Se le hace la siguiente consulta a prolog.

sumar (1, 3, X).

1 y 3 son los valores de entrada, mientras que X es la variable de salida. En Prolog se determina si un parámetro es de entrada o salida, en tiempo de ejecución.

En prolog una variable puede tener dos estados, libre o ligada. Una variable es libre cuando no tiene ningún valor asignado. Una variable es ligada cuando tiene un valor.

Prolog, para evaluar una consulta se va a fijar si cada parámetro tiene un valor asignado o no. En el primer caso la variable será ligada, en el segundo será libre.

En Prolog no existen variables globales. Las variables son locales a cada cláusula.

Si se pone como consulta

X is 1, Y is 2, sumar (X, Y, Z).

Toda variable nace libre. Al colocar ***X is 1 X*** toma le valor ***1.*** Algo similar ocurre con ***Y***. Al llegar a ***sumar***, ***X*** e ***Y*** son ligadas y ***Z*** es libre.

Cuando existe una variable libre se tiene un grado de libertad. Entonces cuando tiene un grado de libertad prolog le da un valor a esa variable, tal que haga verdadera la expresión. En este ejemplo X tomará el valor 4.

En el caso de escribir como consulta

sumar (1, 3, 4).

Como no existe un parámetro de salida, porque todas las variables son ligadas, da verdadero o falso.

Cuando un predicado recibe como parámetro una constante o una variable ligada el parámetro es de entrada. Cuando recibe una variable libre es de salida. La primera vez que aparece una variable, nace libre, una vez que se liga la variable a un valor sigue con ese valor hasta el final de su cláusula (a menos que se produzca backtracking, que se verá mas adelante).

Otro ejemplo.

sumar (1, X, 4)

Se tiene un grado de libertad, en teoría debería dar un valor a X tal que sea verdadero, pero da un error porque ***is*** no acepta una variable libre como operando izquierdo, si como operando derecho.

El operador **==** da falso si se lo aplica a variables libres.

El operador **=\=** da error cuando se lo aplica a variables libres. Es equivalente a ***not(==).***

El operador **=** acepta variables libres en ambos operandos.

En este otro ejemplo

sumar (X,Y, 4).

Se tienen dos grados de libertad en una expresión. Existen infinitas soluciones, cualquier Prolog daría error.

Si se quisiera definir un predicado para incorporar a la base de conocimientos información acerca de autos, se podría escribir.

autos(renault, 2020, azul,2000).

autos(fiat, 2018, rojo, 25000).

autos(fiat, 2019, blanco, 15000).

Donde son todos hechos.

Como consulta se puede poner

autos(Marca, Modelo, Color, Km).

Dará como resultado todos los atributos del primer auto que encuentre en la base de conocimientos.

Marca = renault,

Modelo = 2020,

Color = azul,

Km = 2000

Para filtrar el resultado, se podría escribir por ejemplo

autos(Marca, Modelo, Color, Km), Km>20000.

El resultado será

Marca = fiat,

Modelo = 2018,

Color = rojo,

Km = 25000

Mostrará el primer auto que cumple la condición de tener mas de 20000 km. Con ***Km<*** estamos filtrando por filas en la base de Conocimientos.

Se filtra por filas cuando se ponen condiciones en las variables.

El and (,) del prolog es un and secuencial, se evalúa de izquierda a derecha, si un término es falso no se sigue evaluando, porque en el and basta que un término sea falso para que el resultado sea falso. Se evalúa hasta encontrar un falso o hasta que termine la cláusula. Si llega hasta el final de las clausula el resultado es verdadero sino es falso.

Para filtrar por columnas se debe recurrir a las variables anónimas.

Variable anónima

Una Variable anónima es una variable que no ocupa lugar en memoria. En predicados recursivos al utilizar variables anónimas se logra reducir el uso de memoria. La sintaxis para la variable anónima es "_" . Por ejemplo al escribir

autos(Marca, _, _, Km).

Devolverá

Marca = renault,

Km = 200

Al utilizar variables anónimas se filtra por columnas, en este ejemplo del autos solo muestra Marca y Kilometraje. Se colocaron variables anónimas en las columnas de las cuales no se necesita conocer su valor.

Backtracking

Volviendo al ejemplo de las relaciones de parentesco entre personas. Se busca definir un predicado que imprima los nombres de todos los hijos de una persona.

Se define una primera versión del predicado de la siguiente forma.

lista_hijos(X):- padre_de(X,Y), write(Y).

Si se hace la invocación

lista_hijos(juan).

La variable ***X*** se ligará al valor ***juan*** desde el momento en que comienza a evaluarse el predicado. La variable ***Y*** comenzará siendo libre.

Al evaluar el primer término ***padre_de(X,Y)***, el primer argumento será de entrada (porque recibe una variable ligada) y el segundo será de salida (porque recibe una variable libre). Como existe en la base se conocimientos un hecho que define una relación ***padre_de*** con ***juan*** como primer argumento (de acuerdo a los hechos

incorporados previamente), el resultado de dicha evaluación es verdadero y además como el segundo argumento recibe una variable libre liga a esta variable (***Y***) con el valor ***pedro***. Como el primer término da verdadero, prolog evalúa el segundo que es ***write***.

El predicado ***write*** es un predicado standard de prolog que lo que hace, como su nombre lo indica es simplemente imprimir lo que recibe como parámetro, que debe ser o una constante o una variable ligada, como en este caso.

El resultado será

pedro

Dará el nombre del primer hijo que encuentre tal que haga que el predicado sea verdadero. Pero para obtener los nombres de todos los hijos de ***juan*** habrá que recurrir al backtracking, que es un algoritmo que está implementado en el motor de inferencia de prolog.

Si se reescribe al regla de la siguiente forma.

lista_hijos(X):- padre_de(X,Y), writeln(Y), fail.

El resultado sería

pedro

carlos

El predicado ***writeln*** es similar ***write***, la única diferencia es que imprime un fin de línea al final de modo que el próximo write(ln) sea impreso en la línea siguiente.

La constante ***fail*** es una constante que vale falso (también se puede interpretar como predicado standard que retorna siempre falso).

Como el and del Prolog es secuencial de izquierda a derecha, al recorrer una lista de términos unidos por and sigue evaluando mientras sea verdadero, al encontarr de primer falso termina el and y comienza a actuar al backtracking. El backtracking hace que prolog retroceda para tratar de encontrar otro valor que haga que de verdadero. El prolog (debido al backtracking) siempre trata de lograr que el predicado resulte verdadero. Para lograr esto retrocede (hacia la izquierda) hasta encontrar un punto en que alguna variable fue ligada, en este ejemplo en ***padre_de***

se ligó la variable ***Y***. En este punto prolog desliga la variable y al liga con un nuevo valor (siempre que exista otro valor posible que haga verdadero el término) y vuelve a avanzar hacia la derecha. Si se encuentra con oro falso (en este ejemplo va a ocurrir el llegar al ***fail***) vuelve a retroceder.

Este proceso de avanzar y retroceder se llama backtracking. Si logra llegar al final de la cláusula (todos los términos verdaderos) el resultado es verdadero. Si no logra llegar o sea si termina del lado izquierdo (en el implica) el resultado es falso.

Si una cláusula tiene un ***fail*** el resultado será siempre falso, debido a que el ***fail*** siempre da falso y por lo tanto no se puede cruzar, como ocurre en este ejemplo.

Se coloca fail para forzar el backtracking y de esta forma traer todos los resultados posibles, es este ejemplo los nombres de todos los hijos.

El backtracking, cuando encuentra un falso, retrocede y busca otra solución. En este ejemplo al retroceder se encuentra con write(Y), como en ese término no se ligó ninguna variable al avanzar, sigue retrocediendo y encuentra padre_de(X,Y), en donde se ligó la variable ***Y*** al avanzar. En este punto si se puede obtener otra solución, entonces se desliga la variable ***Y*** se la vuelve a ligar con un nuevo valor. A partir de acá sigue avanzando, vuelve e imprimir y al volver a encontrarse con el fail vuelve a fallar y se repite el proceso. Cuando no existan más soluciones el termino ***padre_de*** va a dar falso y va a seguir retrocediendo hasta el implica por lo que el resultado final será falso.

Prolog ejecuta backtracking cuando se encuentra con un falso y la cláusula tiene al menos un grado de libertad.

En esta primera versión de este ejemplo se obtiene el resultado esperado, los nombres de todos los hijos, pero el resultado del predicado es falso. El hecho de que el resultado sea falso hace que en la consulta no pueda agregar otro término luego de este, porque nunca se va a evaluar, debido a que el and es consecutivo.

Par lograr que el predicado, además de imprimir los nombres de todos los hijos, de verdadero, se debería agregar otra clausula (o poner un or) de modo que de verdadero. falso or verdadero= verdadero. Quedaría

```
lista_hijos(X):- padre_de(X,Y), writeln(Y), fail.

lista_hijos(_).
```

De esta forma la primer clausula siempre da falso (por el fail), pero es la que realiza la impresión. La segunda siempre da verdadero, porque es un hecho. Por lo tanto el resultado final es siempre verdadero. En la segunda clausula se utilizó una

variable anónima pro que no interesa el valor de la variable. En la primer cláusula el valor de ***X*** se utiliza para obtener el nombre del hijo , pero en la segunda no.

pedro

carlos

true.

En una tercer versión del predicado el objetivo es que además de imprimir los nombres de los hijos, retorne verdadero en el caso que tenga al menos un hijo y falso ni no tiene ninguno. Entonces se reescribe el predicado de esta forma.

lista_hijos(X):- padre_de(X,Y), writeln(Y), fail.

lista_hijos(X):- padre_de(X, _).

En la clausula ***lista_hijos(X):- padre_de(X,_).*** Si no tiene ningún hijo da falso y si tiene al menos no da verdadero. Como no interesa el nombre del hijo, solo si tiene o no, se utiliza una variable anónima. Nuevamente la primera cláusula imprime y la segunda determina el valor de verdad.

Las variables son locales a cada cláusula. La variable ***X*** de la primera cláusula es una variable independiente de la ***X*** de la segunda cláusula.

Finalmente en la última versión de este ejemplo se agrega que imprime un mensaje indicando si tiene o no tiene hijos, manteniendo las especificaciones anteriores.

lista_hijos(X):- padre_de(X,Y), writeln(Y), fail.

lista_hijos(X):- padre_de(X,_), writeln("Tiene hijos").

lista_hijos(X):- not(padre_de(X,_)), writeln("No tiene hijos"), fail.

La primer clausula, como siempre, realiza la impresión y da falso.

En la segunda si tiene hijos, el primer término da verdadero y avanza hasta el segundo e imprime ***"Tiene hijos".***

En la tercera si no tiene hijos, el primer término da verdadero y avanza hasta el segundo e imprime ***"No tiene hijos".*** El fail de esta última cláusula se utiliza para que de falso.

El uso más común del ***fail*** es para forzar un backtracking, como en la primera cláusula.

Ejemplo factorial

Desarrollemos el factorial de N, entonces: n! = n.fact(n-1)

```
fact(0,1).

fact(N,R):- N>0, N1 is N-1, fact(N1, R1), R is R1*N.
```

Una possible consulta sería

```
fact(3,X).
```

El resultado

X=6

En prolog no existen sentencias de iterativas (como for o while), la única forma de iterar es mediante al recursividad. Para hacer un algoritmo recursivo que funcione correctamente se deben cumplir 2 condiciones

- Debe existir un caso base, o sea se debe cumplir que por lo menos para un valor de los datos el predicado no se vuelva a invocar a si mismo.
- La sucesión de invocaciones recursivas debe converger a ese caso base.

Si en la definición del predicado no se hubiese puesto la condición

N>0

y se hubiese hecho la siguiente invocación

fact (-1,X).

Pese a tener un caso base, nunca llega a él, porque al ser comenzar como negativo y restar se aleja del cero, que es el caso base.

Si se realiza la siguiente invocación

fact (10000,R)

Haría ~10000 llamadas recursivas, suponiendo que no existen problemas numéricos daría el resultado. Pero si se agrega en la base de conocimientos

fact (8000 , …)

como un hecho, se acorta la búsqueda muchísimo, porque terminaría en 8000,por lo que haría solo ~2000 llamadas. Pero se estaría agregando redundancia a la base de conocimientos.

↑ Redundancia -> ↓Tiempo de búsqueda

↓Redundancia -> ↑Tiempo de búsqueda

Control del backtracking

Para analizar el funcionamiento del backtracking se definen las siguientes cláusulas

r1(a,b).

r1(a,c).

r2(b,d).

r2(b,e).

r2(c,f).

r2(c,g).

r2(c,h).

r3(d,i).

r3(d,j).

r3(f,k).

r3(g,l).

r4(i,m).

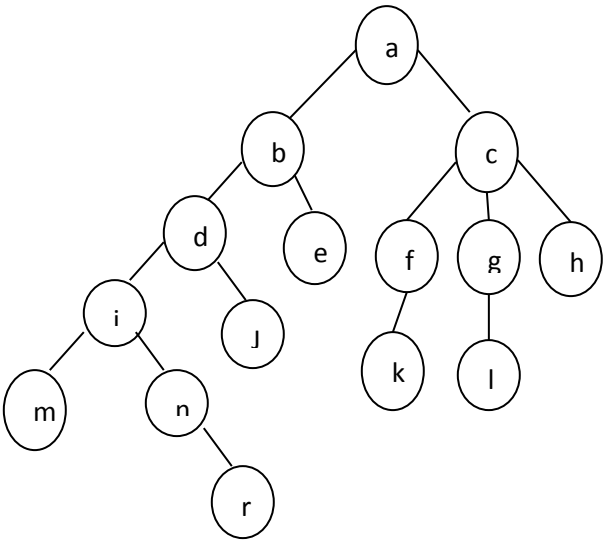
r4(i,n).

r5(n,r).

r(X):-r1(X,Y),writeln(Y),r2(Y,Z),writeln(Z),r3(Z,W),writeln(W),r4(W,U),writeln(U)

,r5(U,T),writeln(T) .

En este programa con los predicados r1, r2, r3, r4 y r5 se trata de representar un árbol como este.



Como consulta se escribe

r(a).

El prolog va a ir recorriendo el árbol por la rama izquierda y cuando se encuentra con que no hay ningún r5 en el nodo ***m***, o sea da falso la invocación ***r5(m, T),*** comienza el retroceso del backtracking, asciedende hasta ***i*** y comeinza a descender de nuevo, ahora si logra llegar hasta ***r*** . Como lego hasta el fina de la clausula termina y da verdadero. recorre ***b, d, I, m, n, r*** y termina.

Para que recorra todo el árbol se le debe poner un fail al final a ***r***, o sea.

r(X):-r1(X,Y),writeln(Y),r2(Y,Z),writeln(Z),r3(Z,W),writeln(W),r4(W,U),writeln(U)

,r5(U,T),writeln(T) , fail.

Como el ***fail*** nunca lo va a pasar da falso y busca todas las soluciones, o sea recorre todo el árbol.

El prolog avanza mientras da verdadero, cuando encuentra algún falso, retrocede buscando otro camino (otra solución), por el que pueda avanzar y ver si por ese otro camino logra ejecutar toda la sentencia. Y nunca vuelve a presentar un resultado que ya presentó con anterioridad.

Existe una forma de controlar el backtracking, para eso se utiliza el operador ***!*** llamado corte. Cuando el prolog avanza lo ignora, solo tiene efecto en el retroceso. Cuando retrocede provoca el corte del backtracking desde el punto en que se encuentra el operador hacia atrás.

Si se pone a final de r(X) la combinación ***!,fail***, hace que de falso y corta el backracking o sea desde el puno de vista del recorrido es como si no hubiese puesto el ***fail***. Si se lo pones en r2(b,d):-!. , entonces no busca la otra solución de este predicado, pero solo afecta a este predicado, no a ***r***.

La base de conocimientos siempre se recorre en el orden que están cargadas las cláusulas.

Problemas con la recursividad

Se define un predicado h, que se aplica a dos elementos, por ejemplo

h(a,b).

h(a,c).

h(b,d).

Si se quiere definir la propiedad reflexiva para este predicado, se podría escribir.

h(X,Y) :- h(Y,X).

Si se realiza la siguiente consulta

h(X,Y), write(X), writeln(Y),fail

Mostrará los resultados esperados, pero los repetirá infinitamente.

Este problema se da porque h es un predicado recursivo, tiene un caso base, pero debido al backtracking que produce por el fail, luego de pasar por el caso base también pasa por la rama recursiva, por lo tanto se sigue invocando indefinidamente.

Algo similar ocurriría si definiéramos la propiedad transitiva par este predicado

h(X,Y) :- h(X,Z), h(Z,Y).

Mostrará los resultados esperados, pero luego emitirá un mensaje de error indicando que se llenó el stack.

La diferencia con el caso anterior es que en este se tienen dos ramas recursivas, en el anterior solo una y al final. Cuando el prolog detecta que un predicado tiene una sola rama recursiva y esta se encuentra al final , no realiza el apilamiento de variables en cada llamada recursiva, por lo que no llena el stack.

Listas en Prolog

En Prolog no existen ni vectores ni matrices, pero si existen listas.

En las listas de Prolog la cantidad de elemento es variable.

Cada elemento se identifica por su posición en la lista.

Si se intercambian los elementos de la lista, la lista cambia.

Los elementos de una lista pueden ser de cualquier tipo.

Por ejemplo

[1,2, c, ”BA”, 8]

Es una lista válida en prolog.

En prolog las listas se delimitan con corchetes y sus elementos se separan por comas.

Un caso particular de lista e sla lista vacia. No tene ningún elemento y se ecribe

[ ]

Como cada elemento de la lista puede ser de cualquier tipo, también puede ser otra lista, o sea se tendría una lista de listas.

[1,[8,3,”a”],7]

Al primer elemento de toda lista no vacía se lo llama Cabeza de la lista.

El resto es la Cola de la Lista (la cola siempre es una lista).

En prolog existe un operador que permite separar la cabeza de la cola de la lista. Es el operador **|**

Por ejemplo si se escribe ***[Ca|Co],*** o sea una se colocan dos variables dentro de corchetes separadas por un **|**, al ligarse esta estructura con una lista en ***Ca*** quedaría la Cabeza y en ***Co*** la cola (incluyendo los corchetes)

Una lista en Prolog se puede definir de la siguiente forma. Una lista o bien es vacía o bien está formada por una cabeza y una Cola. La cola siempre es una lista. La cabeza puede ser un elemento atómico o una lista.

Se puede definir un predicado que recorra e imprima una lista elemente a elemento de la siguiente forma.

```
reclista([]).

reclista([Ca|Co]):-writeln(Ca),reclista(Co).
```

La primer cláusula es para ale caso de lista vacía (No se puede partir una lista vacía en cabeza y cola, y si se recibe una lista vacía da falso)

Como consulta se podría poner

reclista([1,2,3,4]).

Se obtendría como resultado

1

2

3

4

true

Para recorre un lista en orden inverso se puede definir el siguiente predicado

reclistainversa([]).

reclistainversa([Ca|Co]):- reclistainversa(Co), writeln(Ca).

Ejemplo: Definir un predicado que verifique si un elemento indicar si pertenece o no a una lista.

perte([]):-fail.

Debido a que el prolog interpreta lo que no está esta en la base de conocimientos como falso, se puede no poner esta cláusula

perte([Ca|Co],Ca).

perte([Ca|Co],E):-perte(Co, E).

Son variables de las cuales no se va a utilizar su valor, por lo tanto, se pueden definir como anónimas.

Entonces quedaría

perte([Ca|_],Ca).

perte([_|Co],E):-perte(Co, E).

Invocación

perte([1,2,3],2).

Ejemplo: Definir un predicado que cuente la cantidad de elementos de una lista.

cuenta([], 0).

cuenta([_|Co],C):- cuenta(Co,CC), C is CC+1.

cuenta([1,2,3],X), writeln(X).

Resultado

X=3

Ejemplo: Concatenar dos listas:

concatenar([],[],[]).

concatenar([], L, L).

concatenar(L, [], L).

concatenar([Ca|Co], L2, [Ca|Co3]):- concatenar(Co, L2, Co3).

Invocación

concatenar([1,2], [3,4], L)

Resultado:

L = [1,2,3,4]

Otra invocación posible

concatenar(X,Y, [1,2,3,4]), write(X), writeln(Y), fail.

En este caso retorna todas las forma posibles de obtener la lista [1,2,3,4]

El resultado seria

[][1,2,3,4]

[1,2,3,4][]

[1][2,3,4]

[1,2,3,4][]

[1,2][3,4]

[1,2,3,4][]

[1,2,3][4]

[1,2,3,4][]

[1,2,3,4][]

[1,2,3,4][]

[1,2,3,4][]

false.

Se observan resultados repetidos. Esto se debe a que existe redundancia en la definición de concatenar y por la tanto varios caminos posibles para llegar a una misma solución.

Para eliminar la redundancia se puede eliminar al primer cláusula, porque esta incluida en la segunda o en la tercera.

También se puede eliminar la tercer cláusula porque está incluida en la cuarta.

De este modo quedaría.

concatenar([],[],[]).

concatenar([], L, L).

concatenar(L, [], L).

concatenar([Ca|Co], L2, [Ca|Co3]):- concatenar(Co, L2, Co3).

El resultado sería.

[][1,2,3,4]

[1][2,3,4]

[1,2][3,4]

[1,2,3][4]

[1,2,3,4][]

false

El predicado findall

findall(X, p(X), L)

Este predicado se utiliza para generar listas. Crea una lista en el tercer parámetro, cuyos elementos son todos los posibles valores de X tal que hagan que p(X) sea verdadero, donde p es un predicado definido en la base de conocimientos.

O sea

$\forall X / p(X) \Rightarrow X$ pertenece L

findall (X, padre_de(juan, X), L).

retornaría en L una lista con los nombres de los hijos de ***juan***.