1 El proceso de Backtracking

1.1 Backtracking

En esta práctica se examina en detalle cómo Prolog busca todas las soluciones a un objetivo.

Prolog responde a una pregunta hecha por el programador. Esta pregunta contiene una conjunción de objetivos que se deben satisfacer.

Un hecho puede hacer que un objetivo se cumpla inmediatamente, mientras que una regla sólo puede reducir la tarea a la de satisfacer una conjunción de subobjetivos. Si no se puede satisfacer un objetivo, se iniciará un proceso de backtracking. Este proceso consiste en intentar satisfacer los objetivos buscando una forma alternativa de hacerlo.

En nuestro prolog, se puede seguir el proceso de backtracking paso a paso. Basta con escribir la palabra **trace** al inicio del programa y pulsar la tecla **F10** para realizar el siguiente paso. La traza hace que el entorno pare el programa después de cada paso y diga en la ventana **Trace** lo que ha hecho, dónde está y cuáles son los valores de las variables. En esta ventana aparece uno de los siguientes mensajes:

- **CALL:** Cuando intenta satisfacer un predicado.
- **RETURN:** Cuando un predicado se satisface.
- **FAIL:** Cuando un predicado no se satisface.
- **REDO:** Cuando se inicia el proceso de backtracking.

1.2 El predicado corte y el predicado fail

El **corte** es un predicado predefinido y sin argumentos. Se escribe con el signo de exclamación "!" y se utiliza para impedir que se inicie el proceso de backtracking. Como objetivo se cumple inmediatamente pero no puede satisfacerse de nuevo. Su utilización puede ser muy importante para que el programa funcione más rápido y no malgaste memoria y tiempo en intentar satisfacer objetivos que podemos decir de antemano que nunca contribuirán a una solución.

El predicado **fail** es también un predicado predefinido, sin argumentos y siempre fracasa como objetivo, lo que provoca que se desencadene inmediatamente el proceso de backtracking.

Los predicados corte y fail se combinan para indicar situaciones erróneas, es decir, para decir que se ha fallado y que no busque más soluciones.

1.3 Ejercicios propuestos

1.3.1 Sobre el predicado corte

1. Dado el programa:

   ```
   hola(3):- !.
   hola(6).
   hola(X):- Y=X-1, hola(Y), write(Y).
   ```

   Contesta a los siguientes objetivos sin ejecutar el programa:

   (a) **hola(4)**
   (b) **hola(9)**
   (c) **hola(X)**

   Comprueba si tus respuestas son correctas realizando una traza del programa.

2. Considera el siguiente programa:

   ```
   a(X):- b(X).
   a(X):- f(X).
   b(X):- g(X), !, v(X).
   b(X):- X=4, v(X).
   g(1).
   g(2).
   g(3).
   v(_).
   f(5).
   ```

   Ejecuta con una traza **a(X)**, y analiza si el resultado obtenido está en contradicción con el que sale al ejecutar los objetivos **a(2)**, **a(3)** y **a(4)**. ¿Cuál sería la respuesta si quitásemos el corte que aparece en la tercera línea?

3. Se considera el siguiente programa:

   ```
   p(X):- q(X), r(X).
   q(a).
   r(a).
   q(b).
   r(X):-s(X).
   s(b).
   ```

Si se ejecuta el objetivo **p(X)** se obtiene como respuesta **X=a** y **X=b**. Modificar el programa insertando el predicado corte en el lugar adecuado para que el sistema conteste sólo **X=a**.

1.3.2 La sucesión de Fibbonacci

La sucesión de Fibbonacci se define como:

$$f(0) = f(1) = 1,$$
$$f(n) = f(n-1) + f(n-2).$$

Escribe un predicado F(N,M) en Prolog de forma que $M = f(N)$. Este predicado debe de funcionar para los siguientes casos:

1. Dado un número natural N, que sea capaz de generar M.
2. N y M vienen instanciados y debe constestar si/no.
3. Ambas variables vienen sin instanciar y debe de genrar todos los pares N y M.

1.3.3 La función de D'Ackerman

Construye un predicado que defina la función de D'Ackerman que viene dada por:

$$\begin{cases} A(0,n) = n+1, & n \geq 0 \\ A(m,0) = A(m-1,1), & m > 0 \\ A(m,n) = A(m-1, A(m,n-1)) & m,n > 0 \\ \text{no existe solución}, & m < 0, n < 0. \end{cases}$$

1.3.4 El predicado intersección

Se considera el siguiente programa:

```
pertenece(X, [X | _ ]).
pertenece(X, [_ | Y ]):- pertenece(X,Y).
inter( [ ], [ ], _ ).
inter( [X | Y], [X | L1], L2):- pertenece(X, L2), inter(Y, L1, L2).
inter(Y, [ _ | L1 ], L2) :- inter(Y, L1, L2).
```

Se desea que el predicado **inter** funcione de la manera siguiente:

Goal: inter(X,[1,2,3],[2,3,4,5])

X=[2,3]

Goal: inter([2,3],X,[2,3,4,5])

X=[2,3]

1. Usa la traza para saber por qué el resultado obtenido por el programa no es el deseado.

2. Añade un corte en el lugar adecuado para solventar el problema.

2 Listas y recursión

2.1 Listas

La *lista* es una manera fácil de trabajar en situaciones donde es difícil o imposible predecir el número de datos a ser guardados o manipulados.

En prolog, una lista es un objeto que contiene un número arbitrario de otros objetos, llamados *elementos* de la lista. Los elementos de una lista pueden ser de cualquier tipo de dato, incluso pueden ser a su vez listas. He aquí un ejemplo de una lista de enteros:

$$[1, 2, 3, 4, 5]$$

La lista que no contiene ningún elemento es la lista vacía: [ ].

Las listas se manipulan dividiéndolas en una *cabeza* y una *cola*. Una lista, es realmente un objeto compuesto recursivo, que consiste en la cabeza que es el primer elemento y en la cola que es a su vez una lista que contiene todos los elementos excepto el primero.

Como una variable que no esté instanciada se puede unificar con cualquier objeto, también lo puede hacer con una lista. Debido a que una operación común con las listas es separarla en su cabeza y su cola, existe una notación especial en Prolog para representar la lista con cabeza X y cola Y. Esto se escribe $[X|Y]$. Una expresión con esta forma instanciará X a la cabeza de una lista e Y a la cola de la lista.

2.2 Predicados con listas

La forma de trabajar con listas consiste en tomar la cabeza de la lista y luego hacer la misma operación con la cola de la lista. Esta forma de trabajar utiliza dos reglas, una recursiva y un criterio de terminación. Existen tres criterios de terminación muy importantes:

1. **Cuando la lista es vacía.**

   El esquema general de todos los predicados que tienen como criterio de terminación la lista vacía es:

   $predicado([\,]) :- procesar([\,]).$
   $predicado([Cabeza|Cola]) :- procesar(Cabeza), predicado(Cola).$

2. **Cuando un elemento en concreto es encontrado.**

   El esquema general es:

$predicado([Elemento, [Elemento|\_]]) :- procesar(Elemento).$
$predicado(Elemento, [Cabeza|Cola]) :- procesar(Cabeza),$
$predicado(Elemento, Cola).$

3. **Cuando una posición en conreto es encontrada.**

   El esquema general es:

   $predicado([1, Elemento, [Elemento|\_]]) :- procesar(Elemento).$
   $predicado(P, Elemento, [\_|Cola]) :- P1 = P - 1, predicado(P1,$
   $Elemento, Cola).$

2.3 Ejercicios propuestos

2.3.1 Predicados elementales sobre listas

Escribe los siguientes predicados:

1. **primero(X,Lista)**

   X es el primer elemento de la lista.

2. **ultimo(X,Lista)**

   X es el último elemento de la lista.

3. **elemento_par(X,Lista)**

   Es cierto si X ocupa una posición par en la lista.

4. **cuantas(X,Lista,N)**

   X aparece N veces en la lista.

5. **maximo(X,Lista)**

   X es el máximo de la lista.

6. **minimo(X,Lista)**

   X es el minimo de la lista.

7. **sumadelosdemas(Lista)**

   Dada una lista de enteros **L**, se satisface si existe algún elemento en **L** que es igual a la suma de los demás elementos de **L** y falla en caso contrario.

8. **pescalar(L1,L2,P)**

   P es el producto escalar de los vectores **L1** y **L2**. Los dos vectores vienen dados por las dos listas de enteros **L1** y **L2**. El predicado debe fallar si los dos vectores tienen una longitud distinta.