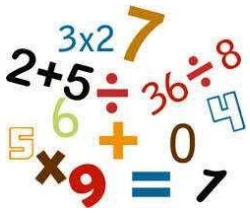
Aritmética

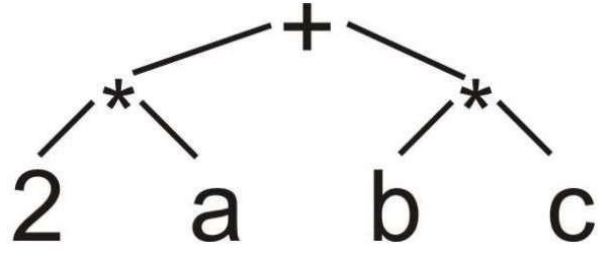


Prolog es más indicado para resolución de problemas simbólicos, pero también ofrece soporte a la aritmética.

Podemos utilizar dos notaciones para representar expresiones en Prolog:

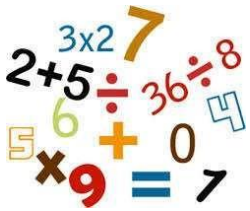
- Infija: 2 * a + b * c
- Prefija: +(*(2, a), *(b, c))

Prolog trata las representaciones de forma equivalente, pues, internamente utiliza árboles para representar expresiones. Así, basta cambiar el orden de encaminamiento para obtener una u otra forma.



Árbol para la expresión 2 ∗ a + b ∗ c

Aritmética



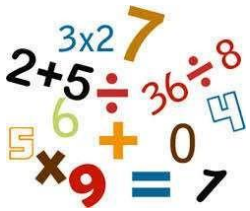
Son ofrecidos diversos predicados para operaciones aritméticas, algunos de estos son:

Operador	Significado
+, -, *, /	Realizar suma, subtracción, multiplicación y división, respectivamente
is mod	atribuye una expresión numérica a una variable
^	Obtener el resto de la división Calcular potenciación
cos, sin, tan	Función coseno, seno ...
exp	exponenciación
ln, log	logaritmo natural y logaritmo
sqrt	raiz

Existen predicados de conversión, tales como:

- integer(X), convierte X para entero;
- float(X), converte X para punto flotante.

Aritmética



Prolog también posee predicados para comparación, los operadores son:

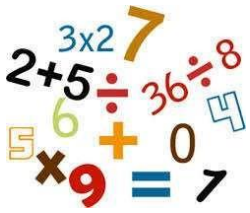
Operador	Significado
>	Mayor que
<	Menor que
>=	Mayor o igual a
=<	Menor o igual a
=:=	Igual
\=	diferente
\+	Negación – retorna exito si el predicado fuera falso y viceversa.

Los operadores = y =:=, realizan diferentes tipos de comparación:

- =, verifica si los “objetos” son iguales, o asigna valores para las variables;
- =:=, evalúa si los valores son iguales.

Unificación de termino.

Aritmética



Ejemplo de uso de = y =:=

?- 1 + 2 = 2 + 1. No
?- 1 + 2 =:= 2 + 1. Yes
?- 1 + A = B + 2. A = 2 B = 1 ; No
?- A =:= 1. ERROR
?- a \=a. false
?- 1 + 2 =:= +(1, 2). true
?- 1 + 2 =:= +(1, 4). false
?- X is sqrt(9). X = 3.0

```
dona(a,a).
dona(a,ab).

?- dona(a,a).

true.

?- \+dona(a,a).
false.
```

Negación.

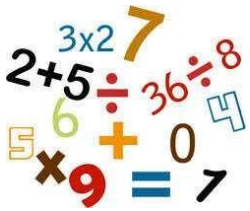
Observación sobre el operador **is**:

```
suma(A,B,Suma) :- Suma is A + B.
 ?- suma(1,2,X).
X = 3;
?- suma(1,2,3).
true ;

?- suma(1,X,3).
ERROR:
```

Todas las variables del lado derecho del operador is, deben estar instanciadas, o sea, deben tener valores asociados a ellas.

Aritmética –Ejercicio



1) Cree una regla en Prolog que pida en consola un número entero e imprima en pantalla si el número es mayor que 100 o si es menor o igual a 100.
2) Suponga los siguientes hechos:

```
nota(juan,10.0). nota(maria,12.0). nota(joana,16.0). nota(mariana,18.0).
nota(claudia,17.0). nota(jose,13.0).  nota(jaoquin,9.0).  nota(mara,8.0).
nota(mary,20.0).
```

Considerando que: Nota de 12.0 a 20.0 = Aprobado.
Nota de 10.0 a 11.9 = Recuperación.
Nota de 0.0 a 9.9 = Reprobado.

Escriba una regla para identificar la situación de un determinado alumno.

Reglas recursivas

La recursión es uno de los elementos más importantes del lenguaje Prolog, este concepto permite la resolución de problemas significativamente complejos de manera relativamente simple.

Una regla es recursiva si su condición depende de la misma, tal como:

a(X) :- b(X), a(X).

La importancia del uso de recursión puede ser ilustrada en la implementación de la relación descendiente(x, y), significando que “x es descendiente de y”.

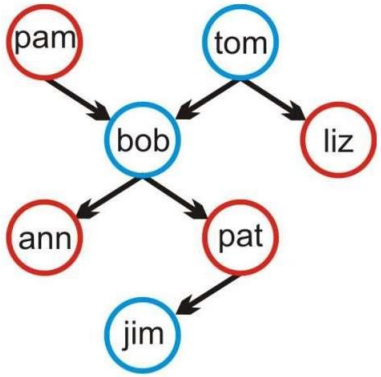
Un conjunto de reglas con el mismo nombre es denominado **procedimiento**.

Reglas recursivas

Ejemplo sin recursión (descendencia genealógica):

Para padres, abuelos, bisabuelos está correcto, pero y para tatarabuelos? Esa solución es limitada y trabajosa.

```
descendiente(X,Y) :- progenitor(Y,X).
descendiente(X,Y) :- progenitor(Y,Z), progenitor(Z,X).
descendiente(X,Y) :- progenitor(Y,Z), progenitor(Z,W), progenitor(W,X)
...
```



Ejemplo con recursión (descendencia genealógica):

Solución simple y trata todos los niveles de la jerarquía, pero recuerde, nunca olvide del caso base antes de la llamada recursiva.

```
descendiente(X,Y) :- progenitor(Y,X).
descendiente(X,Y) :- progenitor(Y,Z),descendiente(X,Z).
```

Reglas recursivas

Ejemplo con recursión (cálculo del factorial):

```
factorial(5) = factorial(4) * 5
             = (factorial(3) * 4) * 5
             = ((factorial(2) * 3) * 4) * 5
             = (((factorial(1) * 2) * 3) * 4) * 5
             = ((((factorial(0) * 1) * 2) * 3) * 4) * 5
             = ((((1 * 1) * 2) * 3) * 4) * 5
```

```
factorial(0,1). % caso base
factorial(N,F) :-
    N > 0,
    N1 is N-1,
    factorial(N1,F1),
    F is N * F1.

?- factorial(3,R).  R
= 6 ;
```

Corte

El retroceso (backtracking) es un proceso por el cual todas las alternativas de solución para una dada consulta son intentadas exhaustivamente.

En Prolog, el retroceso es automático.

Sin embargo, es posible controlarlo a través de un predicado especial llamado corte, denotado por ! .

Visto como una cláusula, su valor es siempre verdadero. Su función es provocar un efecto colateral que interfiere en el procesamiento patrón de una consulta.

Corte

Puede ser usado en cualquier posición en el lado derecho de una regla.

El corte es adecuado a las situaciones donde reglas diferentes son aplicadas en casos mutuamente exclusivos. Hace que el programa se torne más rápido y ocupe menos memoria.

Cuando colocado al final de una cláusula que define un predicado, evita que las cláusulas abajo de esa, relativas al mismo predicado, sean usadas en el backtracking.

Corte (Ejemplo)

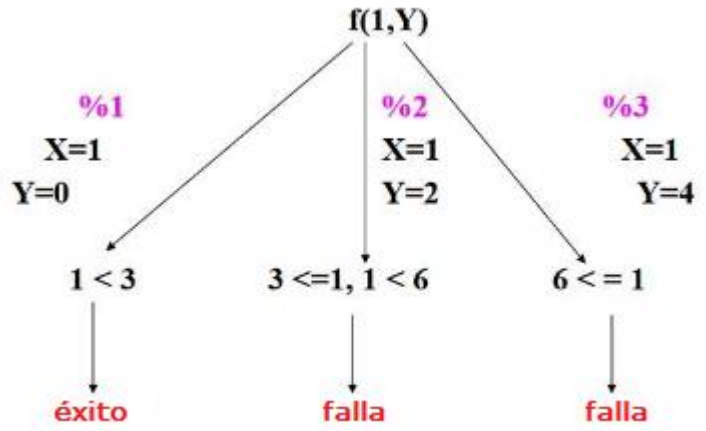
Construir un programa Prolog para implementar la función:

$$f(x) = \begin{cases} 0 \text{ si } x < 3 \\ 2 \text{ si } x >= 3 \text{ y } x < 6 \\ 4 \text{ si } x > 6 \end{cases}$$

```
f(X,0) :- X < 3.          %1
f(X,2) :- 3 =< X, X < 6.  %2
f(X,4) :- 6 =< X.         %3


?- f(1,Y).  Y = 0 ;
false.
```

Sabemos que las reglas son mutuamente exclusivas. Luego, para que testear las otras reglas?



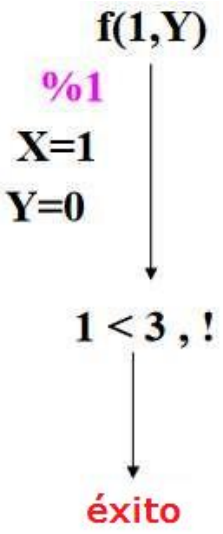
Corte (Ejemplo)

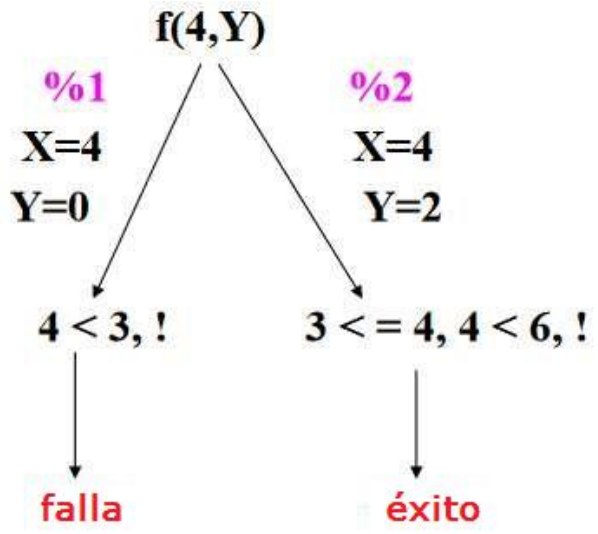
Construir un programa Prolog para implementar la función:

$$f(x) = \begin{cases} 0 \text{ si } x < 3 \\ 2 \text{ si } x >= 3 \text{ y } x < 6 \\ 4 \text{ si } x > 6 \end{cases}$$

```
f(X,0) :- X < 3, !.          %1
f(X,2) :- 3 =< X, X < 6, !. %2
f(X,4) :- 6 =< X.            %3


?- f(1,Y).  Y = 0 ;
false.
```





Ese tipo de corte es llamado de corte verde, si fuera retirado, el programa ejecuta la misma lógica. Altera apenas la eficiencia.

Corte (Ejemplo)

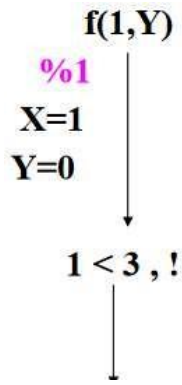
Construir un programa Prolog para implementar función:

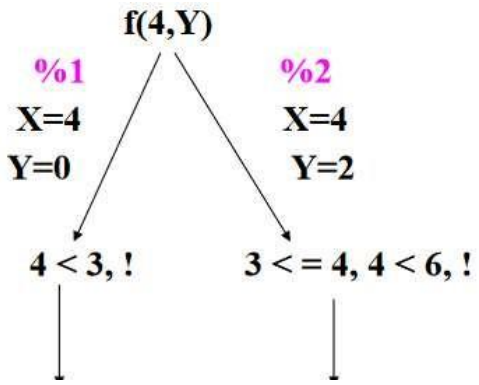
El corte puede dejar los programas más compactos, mas ahora tenemos un ejemplo de corte rojo, si fuera retirado, el programa no ejecutará de la misma forma.

$$f(x) = \begin{cases} 0 \text{ si } x < 3 \\ 2 \text{ si } x >= 3 \text{ y } x < 6 \\ 4 \text{ si } x > 6 \end{cases}$$

```
f(X,0) :- X < 3, ! .      %1
f(X,2) :- X < 6, ! .      %2
f(X,4).                   %3


?- f(1,Y).
Y = 0 ;
false.
```

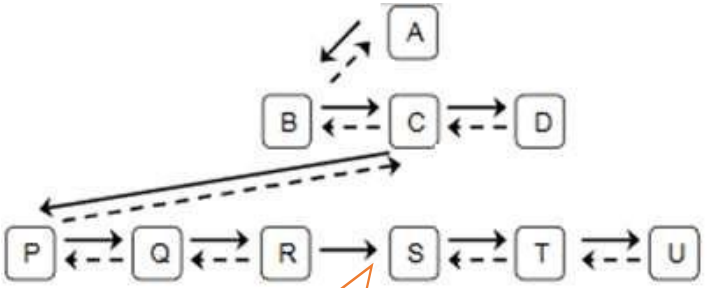




Corte (Ejemplo)

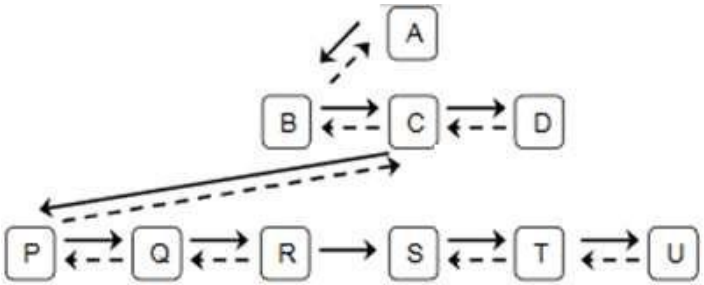
```
b.
d.
p.
q.
r.
s.
t.
u.
v.
c :- p, q, r, !, s, t, u.
c :- v.
a :- b, c, d.

?- a.
```



Cuando el corte es encontrado en medio de una regla, el comportamiento es un poco diferente del presentado en el slide anterior. ⊟ Backtracking sucederá solamente a la derecha del corte.

Corte (Ejemplo)



El corte afectará la ejecución de la meta C
Backtracking será posible dentro de la lista de metas P, Q, R; entre tanto, tan pronto el corte sea encontrado, todas las soluciones alternativas de P, Q, R son descartadas.
También será descartada la cláusula alternativa para la meta C :- V .
Mientras tanto, el backtracking todavía será posible entre la lista de metas S, T, U

La meta-padre de la cláusula conteniendo el corte es la meta C en la cláusula A :- B, C, D.

Por tanto, el corte afectará apenas la ejecución de la meta C; mas será 'invisible' de dentro de la meta A: backtracking automático continuará a existir entre a lista de metas B, C, D a pesar del corte dentro de la cláusula usada para satisfacer C