# Programación Declarativa, 2023-2
# Nota de clase 03: Introducción a PROLOG

Manuel Soto Romero

Facultad de Ciencias UNAM

PROLOG es un lenguaje de programación que permite representar y usar el conocimiento que se tiene en un contexto determinado. Los programas de este paradigma están basados en mecanismos formales como son la Lógica de Primer Orden y las Cláusulas de Horn que revisamos en la Nota de Clase 2.

En esta nota se presentan los conceptos necesarios para tener un primer acercamiento al lenguaje de programación PROLOG, permitiendo que se conozca la estructura básica de un programa, que se analice cómo realizar consultas y que se conozcan algunos elementos básicos como los términos, hechos, consultas y principalmente la definición de predicados mediante reglas de inferencia así como el uso de estructuras de datos, expresiones aritméticas y definiciones de operadores.

## Tipos básicos de PROLOG

El único tipo de dato de PROLOG es el *término*. Todo término es una secuencia de caracteres que permite generar una expresión. Existen cuatro tipos de términos: *átomos*, *números*, *variables* y *términos compuestos*. Algunos autores consideran también términos a las listas[1].

### Átomos

Son considerados los términos más pequeños y existen tres tipos: aquellos formados por letras y números, los que se forman mediante signos especiales y aquellos delimitados por comillas simples (') que pueden incluir espacios en blanco. La única restricción es que éstos siempre deben iniciar con una letra minúscula. [2] Por ejemplo:

```
juanito    m    esto_es_un_atomo    x1    ***
                'esto es un atomo'
```

### Números

El tipo de números depende por lo general del intérprete que se esté usando. En el caso de SWI-PROLOG, se tienen números enteros y flotantes. [2] Por ejemplo:

```
1729    18.35    -502
```

### Variables

Se componen por letras, números y símbolos de guión bajo. La única restricción es que éstos siempre deben iniciar con una letra mayúscula o un guión bajo. El guión bajo es por sí mismo una variable anónima y se utiliza cuando el patrón de ésta no es de gran importancia. [2] Por ejemplo:

```
X    EstoEsUnaVariable    _var1    _
```

### Términos compuestos

Se componen de un *funtor* (nombre) y un número variable de *parámetros*. El funtor debe ser un átomo descriptivo, mientras que los parámetros pueden ser cualquier término válido, incluso otro término compuesto, se separan por comas y se delimitan por paréntesis. [2] Por ejemplo:

```
funtor(a, b)    'funtorcito'(termino)    g(f(a, b),
                                             h(c, d))
```

***Observación***

*El funtor debe ir seguido del paréntesis inicial que delimita a los parámetros (sin espacios), ya que en caso contrario se muestra un error.*

## Hechos, consultas y reglas

Programar en PROLOG consiste en definir un conjunto de hechos (también llamados axiomas o base de conocimientos) y realizar consultas sobre éstos. De igual manera se pueden definir reglas de inferencia que permitan realizar consultas más complejas dada una base de conocimientos, éstos se definen en un archivo de texto con extensión `.pl` que posteriormente es cargado en el intérprete. [3]

### Hechos

Sintácticamente, un hecho es un predicado seguido de un punto. Se utilizan para representar axiomas, es decir, se consideran verdaderos. [2]

***Ejemplo***

El Código 1 muestra un conjunto hechos que definen distintas relaciones que indican si una persona es más alta que otra. Se define la base de conocimientos en el archivo `altos.pl`.

```
alto(manuel, adriana).
alto(ricardo, manuel).
alto(javier, ricardo).
alto(alejandro, adriana).
alto(javier, alejandro).
```

Listado de código 1: Ejemplo de hechos

Para mejorar la legibilidad de los programas se incluyen *comentarios* que describen el propósito del código u otros detalles de documentación. Existen dos tipos de comentarios: de una línea delimitados por `%` y de varias líneas, delimitados por `/*` y `*/`.

***Ejemplo***

El Código 2 muestra una base de conocimientos de la estatura de un conjunto de personas haciendo uso de comentarios que ayudan a leer y documentar el código (líneas 1 a 4).

```
/* Ejemplo: Base de conocimientos de la
    estatura de un conjunto
   de personas. */

% Base de conocimientos.

alto(manuel, adriana).
alto(ricardo, manuel).
alto(javier, ricardo).
alto(alejandro, adriana).
alto(javier, alejandro).
```

Listado de código 2: Base de conocimientos con comentarios

El primer comentario del Código incluye dos líneas (líneas 1 y 2) y describe el propósito general del programa, mientras que

el segundo comentario (línea 4) consta de una línea e indica dónde comienza la base de conocimientos.

### Consultas

Una vez que se ha definido y cargado la base de conocimientos, es posible realizar preguntas o *consultas* sobre el contenido de la misma. El apuntador (*prompt*) de Prolog `?-` indica que el intérprete está listo para realizar consultas.

***Ejemplo***

Consultas sobre la base de conocimientos definida en el Código 1.

El primer paso es cargar el programa con extensión `.pl` para ello se tienen el siguiente comando:

```
?- [altos].
```

Cada que se realiza un cambio en el programa fuente es necesario volver a cargar el mismo en el intérprete.

**Consulta:** ¿Es más alto Ricardo que Manuel?

```
?- alto(ricardo,manuel).
true.
```

**Consulta:** ¿Es más alta Adriana que Ricardo?

```
?- alto(adriana,ricardo).
false.
```

**Consulta:** ¿Es más alto Javier que Manuel?

```
?- alto(javier,manuel).
false.
```

En el Ejemplo 3.3, al realizar una consulta, ésta siempre regresa dos posibles valores, `true` o `false` (verdadero o falso respectivamente). Devuelve el valor de verdadero cuando existe un hecho en la base de conocimientos que satisfaga la consulta y falso en otro caso, sin embargo, existen otro tipo de consultas cuyo propósito es obtener información particular de la base de conocimientos y no sólo si un predicado es falso o verdadero.

***Ejemplo***

¿Qué personas son más altas que Adriana?

```
?- alto(X,adriana).
X = manuel ;
X = alejandro
```

En el Ejemplo 3.4 se realiza una consulta para determinar qué personas son más altas que `adriana`. Para obtener respuesta a este tipo de consultas se usan variables en las consultas (en este caso `X`) y el intérprete busca todos los posibles valores para ésta. A este proceso se le conoce como resolución con unificación de variables. Para mostrar cada posible resultado se debe teclear un punto y coma (`;`) después de obtener una respuesta y para detener la búsqueda de valores se teclea un punto (`.`).

## Reglas de inferencia

En los ejemplos anteriores, se puede pensar que faltan respuestas sobre algunas consultas. Por ejemplo, en la consulta `alto(X,adriana)` la razón dice que también deben aparecer `javier` y `ricardo`, sin embargo, no fue así. De la misma forma, podría parecer que la consulta `alto(javier,manuel)` debe devolver `true` pues si `javier` es más alto que `ricardo` y a su vez `ricardo` es más alto que `manuel`, en particular, `javier` es más alto que `manuel`.

Este tipo de razonamiento es conocido como *regla de inferencia* y como su nombre lo dice, permite inferir datos o respuestas a partir de un conjunto de hechos definidos previamente.

***Ejemplo***

El Código 3 muestra las reglas de inferencia que definen una regla transitiva para la relación `alto`, es decir, si una persona `X` es más alta que otra persona `Z` y la persona `Z` es más alta que otra persona `Y`, entonces `X` es más alta que `Y`. [2, 3] Estas dos reglas en conjunto forman el predicado `mas_alto`. Se trata de un predicado recursivo en el cual, el caso base usa la definición antes definida y el caso recursivo, verifica si existe un término `Z` tal que `Z` sea más alto que `Y` de forma recursiva.

```
% Predicado que indica si una persona X
    es más alta que otra Y.
mas_alto(X,Y) :- alto(X,Y).
mas_alto(X,Y) :- alto(X,Z), mas_alto(Z,
   Y).
```

Listado de código 3: Base de conocimientos con comentarios

En el Ejemplo 3.5, el símbolo `:-` se lee como “si” y las comas entre cada regla de inferencia se leen como “y”. Estas reglas de inferencia son equivalentes a la notación formal:

$$\frac{alto(X,Y)}{mas\_alto(X,Y)} \qquad \frac{alto(X,Z) \quad mas\_alto(Z,Y)}{mas\_alto(X,Y)}$$

La parte abajo de estas reglas de inferencia es llamada conclusión y la parte de arriba se conforma de las premisas. [3] De esta forma, para poder concluir `mas_alto(X,Y)` hay que probar que existe el hecho `alto(X,Y)` o que a partir de `alto(X,Z)` y `mas_alto(Z,Y)` se puede llegar a algún hecho. Dicho de otro modo, si no se puede construir una prueba con la conclusión y premisas, PROLOG reportará `false`.

***Ejemplo***

Una consulta para preguntar quién es más alto que `adriana`.

```
?- mas_alto(X,adriana).
X = manuel ;
X = alejandro ;
X = ricardo ;
X = javier ;
X = javier ;
false.
```

Análisis de la consulta:

1. *Se realiza la consulta* `mas_alto(X,adriana).`
2. *Se procede con la primera regla de inferencia:*

   ```
   mas_alto(X,adriana) :- alto(X,adriana)
   ```

   *Se tienen dos hechos que satisfacen* `alto(X,adriana)` *por lo que se obtiene*

   `X = manuel` y `X = alejandro`
3. *Se procede con la segunda regla de inferencia:*

   ```
   mas_alto(X,adriana) :- alto(X,Z),
         mas_alto(Z,adriana)
   ```

   *Para* `alto(X,Z)` *se obtiene:*

   ```
   X = manuel , Z = adriana
   X = ricardo , Z = manuel
   X = javier , Z = ricardo
   X = alejandro , Z = adriana
   X = javier , Z = alejandro
   ```

   *Para* `alto(Z,adriana)` *se obtiene:*

   ```
   Z = manuel
   Z = alejandro
   Z = ricardo
   Z = javier
   Z = javier
   ```

   *Por lo tanto:*

   ```
   X = ricardo
   X = javier
   X = javier
   ```

   Se aprecia cómo el término `javier` aparece en dos ocasiones. De la misma forma, el último resultado es falso pues uno de todos los posibles caminos falla. Esto ocurre debido a la forma en que PROLOG realiza las como vimos en la Nota de Clase 2.

## Manejo de listas

En PROLOG, las listas se definen recursivamente de la siguiente manera:

***Definición***

*Una lista es:*

- *La lista vacía es una lista y se representa por* `[]`.
- *Si* `C` *es un elemento de un conjunto cualquiera y* `R` *es una lista, entonces* `[C|R]` *lo es también. Llamamos a* `C` *la cabeza de la lista y a* `R` *el resto.*
- *Son todas.*

Revisamos en esta sección el uso de listas en la solución de problemas así como sus distintas formas de representación.

**Representación de listas**

*Representación simple*

En su forma más simple, una lista se representa mediante una secuencia de elementos separados por comas y delimitados por corchetes. Los elementos de una lista pueden ser cualquier término válido o incluso otra lista. [2]

```
[1,2,3,4,5]     [l,i,s,t,a]     [C,I,E,N,C,I,A]
                [[1,2],[3,4]]   []
```

*Representación interna*

Internamente, las listas se construyen mediante el *predicado punto* que recibe dos parámetros: uno para la cabeza de la lista y otro para el resto. Esta representación es más un formalismo y no suele usarse en programas reales. [2]

```
.(1,.(2,.(3,.(4,.(5,[])))))
.(.(1,.(2,[])),.(3,.(4,[])), [])
```

*Representación mediante patrones*

La definición de predicados recursivos se realiza mediante la especificación del *caso base* y el *caso recursivo*. En el caso de las listas el caso base es la lista vacía y el caso recursivo se realiza sobre listas que tienen cabeza y resto. Para definir este tipo de predicados se provee la representación de listas mediante el patrón `[C|R]` que permite obtener la cabeza y el resto de la lista de una forma sencilla. [2]

```
[1|[2,3,4,5]]     [l,i|[s,t,a]]     [C,I,E|[N,C,I,A]]
                   [[1,2]|[[3,4]]]
```

La segunda y tercera expresión, muestran cómo usar este patrón para listar los primeros elementos además de la cabeza.

**Predicados sobre listas**

A continuación se muestran ejemplos de definición de predicados que usan listas, con el fin de ilustrar el uso de patrón `[C|R]` y repasar el concepto de recursión.

***Ejemplo***

El Código 4 muestra un predicado que indica si un elemento está contenido dentro de una lista. Se proporciona primero la regla para el caso base, pues en caso de cumplirse permite terminar la recursión evitando ciclos infinitos y una regla para el caso recursivo que descompone el problema en casos más pequeños.

```
% Predicado contiene(L ,X) que indica
   si el elemento X se encuentra
% contenido en la lista L.
contiene([E|_],E).
contiene([_|R],E) :- contiene(R,E).
```

Listado de código 4: Regla para determinar si un elemento está contenido en una lista

Nótese que no se define el caso para la lista vacía pues resulta ser falso y no es necesario proveer una regla para éste.

Análisis de casos del predicado:

***Caso 1:*** *La lista es vacía.*

```
?- contiene([],1).
false.
```

1. *Se realiza la consulta* `contiene([],1).`
2. *Para* `contiene([],1)` *no existe ningún hecho o regla que satisfaga la consulta y por lo tanto se detiene la búsqueda y se reporta* `false`.

***Caso 2:*** *El elemento se encuentra al inicio de la lista.*

```
?- contiene([1,2,3],1).
true ;
false.
```

1. *Se realiza la consulta* `contiene([1,2,3],1)`
2. *Se procede con la primera regla de inferencia.*

   `contiene([1,2,3],1).`

3. *La consulta es verdadera y por lo tanto se reporta* `true`.
4. *Otro camino falla, lo cual explica el segundo resultado con* `false`.

***Caso 3:*** *El elemento se encuentra en cualquier otra posición de la lista.*

```
?- contiene([1,2,3],2).
true ;
false.
```

1. *Se realiza la consulta* `contiene([1,2,3],2).`
2. *Se procede con la primera regla de inferencia:*

   `contiene([1,2,3],2).`

   *La consulta es falsa.*
3. *Se procede con la segunda regla de inferencia:*

   `contiene([_|[2,3],2) :- contiene([2,3],2)`

   *Para* `contiene([2,3],2)` *se tiene que la consulta es verdadera y por lo tanto se reporta* `true`.
4. *Otro camino falla, lo cual explica el segundo resultado con* `false`.

***Caso 4:*** *El elemento no se encuentra en la lista.*

```
?- contiene([1,2,3],4).
false.
```

1. *Se realiza la consulta* `contiene([1,2,3],4).`
2. *Se procede con la primera regla de inferencia:*

```
contiene([1,2,3],4).
```

*La consulta es falsa.*

3. *Se procede con la segunda regla de inferencia:*

```
contiene([_|[2,3]],4) :- contiene([2,3],4)
```

*Para* `contiene([2,3],4)` *se tiene que la consulta es falsa.*

4. *Se procede nuevamente con la segunda regla de inferencia:*

```
contiene([_|[3]],4) :- contiene([3],4)
```

*Para* `contiene([3],4)` *se tiene que la consulta es falsa.*

5. *Se procede nuevamente con la segunda regla de inferencia:*

```
contiene([_|[]],4) :- contiene([],4)
```

*Para* `contiene([],4)` *no existe ningún hecho que satisfaga la consulta y por lo tanto se detiene la recursión y se reporta* `false`.

***Ejemplo***

El Código 5 muestra un predicado para concatenar listas.

```
% Predicado concatena(X,Y,Z) que
   representa a Z como la
% concatenación de las listas X y Y.
concatena([],L2,L2).
concatena([C1|R1],L2,[C1|R3]) :-
   concatena(R1,L2,R3).
```

Listado de código 5: Regla para concatenar listas

La primera regla de inferencia, indica que al concatenar una lista vacía con cualquier otra, da como resultado la segunda lista. En la segunda regla se verifica antes de probar cualquier cosa que la cabeza de la primera y tercera lista sean iguales, de lo contrario, se reporta `false`.

```
?- concatena([1,2,3],[4,5,6],C).
C = [1,2,3,4,5,6].
?- concatena([1,2,3],[4,5,6],[1,2,3,4,5,6]).
true.
?- concatena([1,2,3],[4,5,6],[2,3,4,5,6]).
false.
```

## Listas diferencia

Una *estructura de datos incompleta* es una estructura que hace uso de variables para representar *huecos* que a su vez, representan partes de la estructura cuyo valor es desconocido. Las *listas diferencia* son estructuras de datos de este tipo y se forma a partir de una *lista abierta* y su hueco. Una lista abierta es una lista que tiene como resto una variable (hueco). Para representar lista diferencia se usa el símbolo - entre la lista abierta y su hueco.

*Lista abierta:* `[1,2,3|X]`

*Lista diferencia:* `[1,2,3|X]-X`

***Ejemplo***

Un ejemplo clásico del uso de listas diferencia es la concatenación de listas. La concatenación de listas es un predicado ampliamente usado en el procesamiento de listas por lo que debería ser lo más eficiente posible, sin embargo, a partir del Ejemplo 3.8 es fácil ver que su complejidad es $O(n)$.

Usando listas diferencia, esta complejidad puede cambiar a $O(1)$. El Código 6 muestra una segunda implementación del predicado que relaciona dos listas con su concatenación.

```
% Predicado concatenaDif1(X,Y,Z) que
   representa a Z como la
% concatenación de las listas X y Y.
concatenaDif1(A-Ha,B-Hb,C-Hc) :- Ha = B
   , Hb = Hc, C = A.
```

Listado de código 6: Regla para concatenar listas diferencia

Análisis de consulta:

```
?- concatena([1,2,3|X]-X,[4,5,6|Y]-Y,Z).
X = [4,5,6|Y].
Z = [1,2,3,4,5,6|Y]-Y.
```

1. *Se realiza la consulta*
   `contiene([1,2,3|X]-X,[4,5,6|Y]-Y,Z)..`
2. *Se realizan las siguientes unificaciones:*

   *Sea* `Hc` *el hueco de* `Z`.

   `X = [4,5,6|Y]` *y por lo tanto la primera lista es* `[1,2,3,4,5,6|Y]`
   `Y = Hc`
   `Z = [1,2,3,4,5,6|Y]-Y`

De esta forma, el Código 7 muestra una definición sintetizada de la concatenación mediante listas diferencia:

```
% Predicado concatenaDif2(X,Y,Z) que
   representa a Z como la
% concatenación de las listas X y Y.
concatenaDif2(A-B,B-C,A-C).
```

Listado de código 7: Regla para concatenar listas diferencia

## Expresiones aritméticas

Como en la mayoría de lenguajes de programación, PROLOG provee mecanismos para trabajar con expresiones aritméticas. La diferencia con otros lenguajes es que se utiliza una representación que permita trabajar con las operaciones definidas mediante predicados. En esta sección revisaremos esta representación y la usaremos en la solución de distintos problemas.

### El operador `is`

Para verificar o calcular el valor de una expresión aritmética como $1+7$ o $(2\times 9)+(1/2)$ se usa el predicado `is`, el cual asocia el resultado de una expresión aritmética a una variable. Este predicado u operador (como es llamado comúnmente) puede usarse de manera prefija como cualquier otro predicado o de forma infija.

***Ejemplo***

Ejemplos de uso del operador `is`:

```
?- X is 1 + 7.
X = 8.
?- is(X,(2*9)+(1/2))
X = 18.5.
```

***Observación***

*El operador* `is` *no es conmutativo. Esto quiere decir que las expresiones de la forma* `X is 1+7` *no son equivalentes a* `1+7 is X`. *El operador* `is` *evalúa el operando derecho y lo relaciona con la variable izquierda. [2]*

**Operadores aritméticos**

Para operar con las expresiones aritméticas PROLOG provee varios operadores o predicados en su núcleo. Estos operadores se dividen el *funciones* y *relaciones*. Las funciones se usan para realizar cálculos, mientras que las relaciones se usan para comparar expresiones aritméticas. [2]

*Funciones*

En el siguiente cuadro se listan algunas de las principales funciones de `Prolog`.[1]

[1]Para revisar más funciones aritméticas, consultar el manual de referencia de SWI-PROLOG, disponible en: `https://www.swi-prolog.org/pldoc/doc_for?object=manual`.

| Operación | Notación | Ejemplos |
|---|---|---|
| Suma | Infija o prefija | `1+7, +(1,7)` |
| Resta | Infija o prefija | `2-9, -(2,9)` |
| Producto | Infija o prefija | `1*8, *(1,8)` |
| División real | Infija o prefija | `3/5, /(3,5)` |
| División entera | Infija o prefija | `4//5, //(4,5)` |
| Potencia | Infija o prefija | `5**2, **(5,2)` |
| Máximo de dos números | Prefija | `max(6,1)` |
| Mínimo de dos números | Prefija | `min(3,1)` |
| Raíz cuadrada | Prefija | `sqrt(2)` |
| Seno | Prefija | `sin(9)` |
| Conversión de enteros a flotantes | Prefija | `float(1)` |
| Redondear un flotante al siguiente entero | Prefija | `round(8.5)` |

*Relaciones*

En el siguiente cuadro se listan los operadores relacionales para expresiones aritméticas.

| Operación | Notación | Ejemplos |
|---|---|---|
| Menor que | Infija o prefija | `1<7, <(1,7)` |
| Mayor que | Infija o prefija | `2>9, >(2,9)` |
| Menor o igual | Infija o prefija | `1=<8, =<(1,8)` |
| Mayor o igual | Infija o prefija | `3>=5, >=(3,5)` |
| Distinto de | Infija o prefija | `4=\=5, =\=(4,5)` |
| Igual que | Infija o prefija | `5=:=5, =:=(5,5)` |

***Ejemplo***

El Código 1, muestra las fórmulas para calcular el área y perímetro de triángulos.

Se usa el operador `is` en la definición de los tres predicados.

Para calcular el área de un triángulo con lados $a, b, c$ se usa la fórmula de Herón:

$$A = \sqrt{S(S-a)(S-b)(S-c)}$$

donde $S$ es el semiperímetro:

$$S = \frac{(a+b+c)}{2}$$

De esta forma, para probar que el área de un triángulo $T$ es $A$, primero se calcula el semiperímetro osando la regla `semiperimetro/2` y se usa el valor con el que fue unificada la variable S para obtener el área.

```
/* Ejemplo: Cálculo de área y perímetro
    de triángulos. */

% Base de conocimientos.

triangulo(a,lados(4,4,5)).
triangulo(b,lados(3,25,6)).

% Predicado perimetro(T,P) que
   relaciona al triangulo T con su
% perímetro P.
perimetro(T,P) :-
   triangulo(T,lados(A,B,C)),
   P is A + B + C.

% Predicado area(T,A) que relaciona al
   triángulo T con su
% área A.
area(T,A) :-
   triangulo(T,lados(X,Y,Z)),
   semiperimetro(T,S),
   A is sqrt(S*(S-X)*(S-Y),(S-Z)).

% Predicado semiperimetro(T,S) que
   relaciona al triangulo T con su
% semiperimetro S.
semiperimetro(T,S) :-
   triangulo(T,lados(A,B,C)).
   S is (A+B+C)/2.
```

Listado de código 8: Perímetro y área de triángulos

```
?- perimetro(a,P).
P = 13.
?- perimetro(b,P).
P = 54.
?- area(a,A).
A = 7.806247497997997.
?- area(b,A).
A = 36.
```

## Definición de operadores

En la sección anterior, se presentó una equivalencia entre expresiones de la forma `1+2` y `+(1,2)` la diferencia entre éstas dos expresiones es que la primera define un operador que puede ser infijo, prefijo o postfijo y la segunda es simplemente una regla definida en el núcleo del lenguaje o por el programador. Ya hemos visto cómo definir reglas, por lo tanto

en esta sección revisaremos cómo realizar la definición de un operador a partir de una regla usando sus reglas de precedencia y asociatividad.

**Precedencia y asociatividad**

Para definir un nuevo operador en PROLOG, hay que tomar en cuenta la precedencia y asociatividad del mismo.

*Precedencia*

Para definir el orden en que se evalúa una operación en una expresión que contiene dos o más operadores, se definen reglas de precedencia a los operadores. Si un operador $\square$ tiene mayor precedencia que otro operador $\bigtriangleup$, esto quiere decir que primero se evalúa la operación de $\square$ y después $\bigtriangleup$. Esto permite además, reducir el número de paréntesis en una expresión. Por ejemplo, en las expresiones aritméticas la multiplicación tiene mayor precedencia que la suma y por lo tanto la expresión $1+7\times 2$ se evalúa a 15.[2]

Para indicar la precedencia de operadores en PROLOG se usan números del 0 al 1200. Mientras más pequeño sea este número, mayor será la precedencia del operador. Por ejemplo, nuevamente en la expresión 1+7*2, el operador * tiene una precedencia de 400 y el operador + de 500.

***Observación***

*La precedencia de un término siempre es de 0, es decir, siempre se evalúan primero. [2]*

*Asociatividad*

Esta propiedad permite decidir en caso de que se tenga al mismo operador más de una vez en una expresión, que o incluye paréntesis, cuál de las presencial del operador debe evaluarse primero. Por ejemplo, en la expresión $9-1-8$, ¿cuál de los dos operadores debe evaluarse primero?, ¿el de la izquierda o el de la derecha? Si se evalúa primero $9-1=8$ y luego se resuelve $8-8$ se obtendrá 0, mientras que si se evalúa primero $1-8=-7$ y luego se resuelve $9--7$ se obtendrá 16.

Para definir la precedencia de un operador en PROLOG se usan átomos que indican el patrón de asociatividad correspondiente y la notación que usa el operador. El átomo `f` indica la posición del operador y los átomos `x` y `y` indican su asociatividad. La `y` indica hacia qué lado se realiza, si ésta es omitida, significa que no se tiene una regla de asociatividad definida. [2]

| Patrón | Significado |
|---|---|
| `yfx` | Operador infijo que asocia a la izquierda |
| `xfy` | Operador infijo que asocia a la derecha |
| `xfx` | Operador infijo sin regla de asociatividad |
| `fy` | Operador prefijo que asocia |
| `fx` | Operador prefijo sin regla de asociatividad |
| `yf` | Operador postfijo que asocia |
| `xf` | Operador postfijo sin regla de asociatividad. |

Para conocer la precedencia y asociatividad de un operador puede usarse el predicado `current_op/3`. Este predicado asocia un operador con su precedencia y asociatividad.

***Ejemplo***

Precedencia y asociatividad del operador resta.

```
?- current_op(Precedencia,Asociatividad,-).
Precedencia = 200, Asociatividad = fy ;
Precedencia = 500, Asociatividad = yfx ;
?- X is 9 - 1 - 8.
X = 0.
```

Se muestran dos resultados pues este operador también puede usarse de manera prefija. Para el caso de la notación infija, se aprecia que la asociatividad se da hacia la izquierda, por lo tanto en PROLOG, la expresión $9-1-8$ se evalúa a 0.

**Definición de operadores**

Para definir un nuevo operador, se debe indicar primero el predicado que se asociará con éste y establecer su precedencia y asociatividad. Para hacer esto, se usa el predicado op/3. El primer parámetro indica la precedencia, el segundo el valor de asociatividad y el tercero el predicado asociado. [2]

***Ejemplo***

En el Código 1 se muestra la definición del operador `contiene`. La línea 5 define la precedencia y asociatividad del operador y las líneas 8 y 9 definen el predicado asociado al operador.

```
/* Ejemplo: Definición de operadores.
   */

% Definición de operadores.

:- op(300,xfx,contiene).

% Predicado contiene(X,L) que indica si
    el elemento X se encuentra
% contenido en la lista L.
contiene([E|_],E).
contiene([_|R],E) :- contiene(R,E).
```

Listado de código 9: Definición del operador contiene

Se define el operador `contiene` al inicio del programa para que éste sea cargado junto con el archivo. Podemos apreciar que se trata de una meta. La precedencia del operador es 300, el patrón de asocitatividad indica que es prefijo y que no hay regla definida, este operador se asocia a la regla `contiene` definida en Secciones anteriores.

```
?- [1,2,3] contiene 1
true.
```

## Referencias


[1] Manuel Soto, *Manual de Prácticas para la Asignatura de Programación Declarativa,* Facultad de Ciencias UNAM, Proyecto de Titulación, 2019.

[2] Ulle Endriss, *An Introduction to Prolog Programming,* Notas de clase, Institute for Logic, Language and Computation, Universiteit van Amsterdam, 2016.

[3] Favio E. Miranda, Lourdes del C. González, *Notas para el curso de Programación Funcional y Lógica,* Facultad de Ciencias UNAM, Revisión 2012-1.

[4] William F. Clocksin, Christopher S. Mellish, *Programming in Prolog*, Quinta edición, Springer, 2003.