



Paradigma Lógico

Módulo 3: Estructuras de datos. Individuos simples y compuestos.

**por Fernando Dodino
Carlos Lombardi
Nicolás Passerini
Daniel Solmirano**

revisado por Matías Freyre

**Versión 2.0
Diciembre 2016**

Contenido

[1 Abstracciones para modelar información](#)

[2 Individuos simples](#)

[2.1 Números](#)

[2.2 Strings](#)

[3 Pattern matching](#)

[4 Individuos compuestos](#)

[4.1 Listas](#)

[4.2 Operaciones básicas sobre listas](#)

[4.2.1 length](#)

[4.2.2 member](#)

[4.2.3 append](#)

[4.2.4 nth/3](#)

[4.2.5 last/2](#)

[4.2.6 reverse/2](#)

[4.2.7 Otros predicados interesantes](#)

[4.3 Pattern matching con listas](#)

[4.4 Functores](#)

[4.5 Pattern matching con functores](#)

[5 Ejemplo integrador](#)

[5.1 Nombre de un alumno](#)

[5.2 Nota de un alumno](#)

[5.3 Alumnos sacrificados](#)

[6 Resumen](#)

1 Abstracciones para modelar información

Más allá del paradigma en el que estemos trabajando, siempre es importante poder modelar información de nuestro dominio. Ejemplo: queremos modelar un viaje en un transporte público de larga distancia.

- la cantidad de pasajes que saco es un número (entero)
- el precio del viaje es otro número (que posiblemente admita decimales)
- el día en que viajo es una fecha
- si el boleto incluye ida y vuelta, es un sí/no (booleano)
- la persona que viaja, hasta el momento la modelamos como un individuo simple: juan, macarena, tobias, etc.
 - otra opción sería modelarlo como un String, encerrando el nombre entre comillas dobles: "Tobías Sandler", "Macarena Lepera", etc.
- el viaje en sí podría formar una estructura compuesta de varios átomos
- o podría modelar varios viajes, y necesitar la noción de conjunto

En el presente capítulo nos dedicaremos a estudiar las estructuras de datos presentes en el lenguaje Prolog.

2 Individuos simples

Ya hemos trabajado con individuos anteriormente:

```
pastas(ravioles).  
come(juan, fideos).
```

ravioles, juan, fideos marcan lo que conocíamos hasta el momento como un individuo, que es un tipo de dato simple: es atómico, no se puede descomponer en otros elementos.

Estos individuos participan en los predicados, y podemos compararlos:

? juan = pepe	? juan = juan	? tobi \= ravioles
false	true	true (son distintos)

2.1 Números

Los números se utilizan como literales:

```
ingrediente(1, pollo).  
nota(salvatelli, 8).
```

En este caso tenemos 1 y 8 como valores literales. Los números, además de poder compararse por igualdad/desigualdad definen

- operadores aritméticos: la suma (Valor + 2), la resta (Valor - Monto), la multiplicación ($3 * 4$), la división (Numero / 2), el valor absoluto ($\text{abs}(-2)$), entre otros
- operaciones de comparación por orden: $<$, $=$, $>$, $>=$

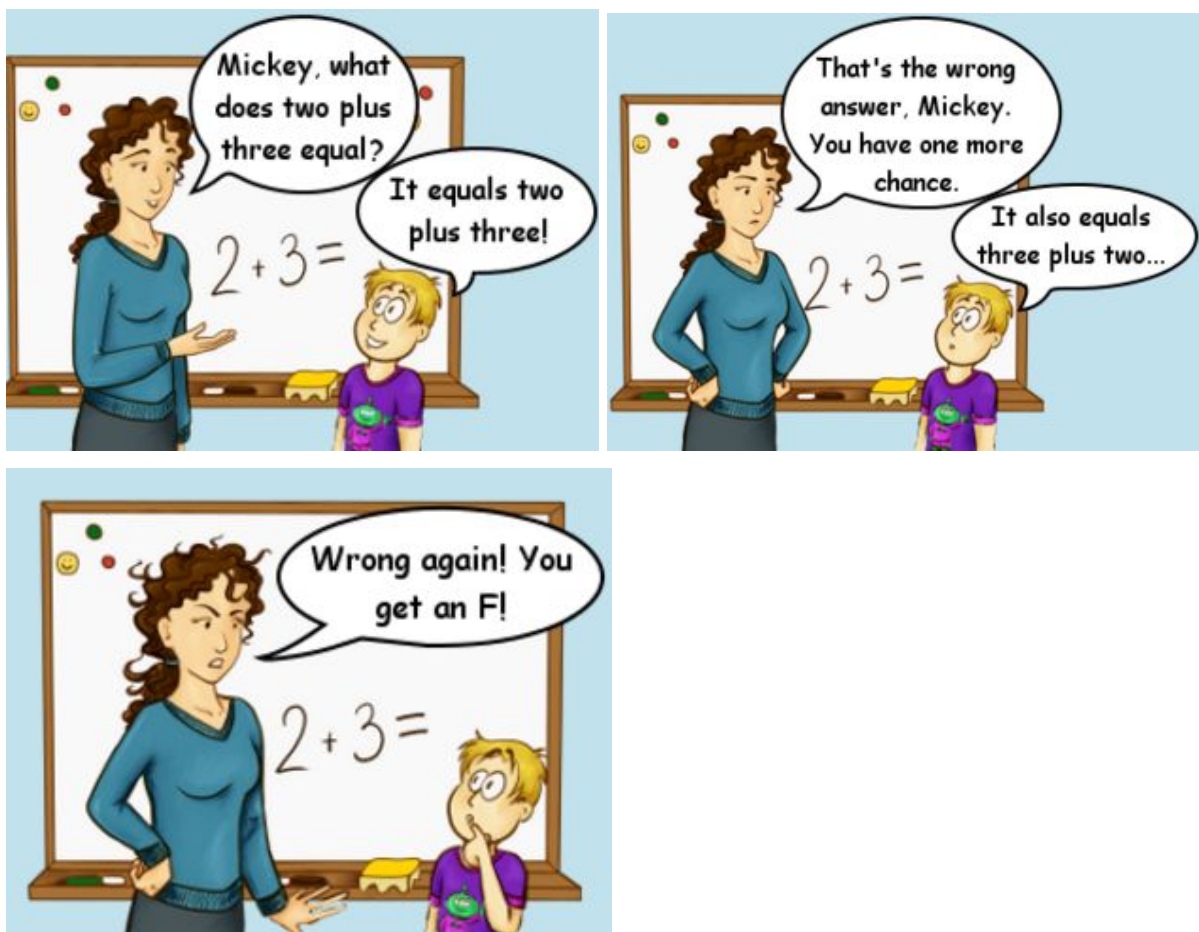
Ahora bien, si pensamos en una regla que relacione un número con su valor siguiente, tenemos que pensar en

`siguiente(N, Siguiente):- ...`

donde Siguiente debería ser... $N + 1$. Pero no podemos escribirlo como

`siguiente(N, Siguiente):- Siguiente = N + 1.`

Esto nos recuerda a este chiste de Cartesian Closed Comic¹, que muestra lo frustrante que puede ser confundir igualdad con reducción en matemática:



¹ <https://ro-che.info/ccc/18>

De hecho si prueban en Prolog lo siguiente:

```
? 5 = 2 + 3
```

```
false // el número 5 no es igual a la expresión 2 + 3
```

```
? 2 + 3 = 3 + 2
```

```
false // la expresión 2 + 3 no es igual a la expresión 3 + 2
```

```
? 2 + 3 = 2 + 3
```

```
true // son las mismas expresiones
```

Volviendo a la regla siguiente/2, debemos utilizar el operador `is/2`, que relaciona un número (del lado izquierdo) con una operación (del lado derecho) que se evalúa.

Num `is` Operacion

Pueden probar estas consultas:

```
? 4 is 2 * 2.
```

```
true // se puede unificar 4 a la expresión evaluada 2 * 2
```

```
? 4 is 24 - 6.
```

```
false // 4 no es unificable a 18
```

```
? Z is 2 * 2.
```

```
Z = 4 _ // existe un individuo que satisface la operación 2 * 2, que se evalúa como 4
```

Definimos entonces el predicado siguiente/2:

```
siguiente(N, Siguiente):- Siguiente is N + 1.
```

Esto nos permite hacer estas consultas:

```
? siguiente(2, 3).
```

```
true
```

```
? siguiente(4, Numero).
```




```
Numero = 5 _
```

Pero no podemos dejar variables sin ligar del lado derecho de una expresión en el operador `is/2`:

```
? siguiente(Numero, 4)
```

```
ERROR: is/2: Arguments are not sufficiently instantiated
```

La expresión `N + 1` contiene a la variable `N` sin unificar, y no está permitido dejar incógnitas libres en la parte derecha del operador `is/2`.

? 5 is 3 + 2  ? Numero is 3 + 2  ? 5 is Numero + 2 

En Prolog podemos usar indistintamente números enteros y decimales, mediante literales:

```
?- 6 > 3.2.
true.
```

Lo que no podemos es consultar los individuos mayores a un número:

```
?- Numero > 3.2.
ERROR: >/2: Arguments are not sufficiently instantiated
```

Claro, tiene sentido porque no conocemos el universo de individuos que participan de la relación con el `3.2`².

En resumen, hay restricciones de inversibilidad cuando trabajamos con estos predicados

- Comparaciones (`>`, `<`, `>=`, `<=`, `=`)
- Operaciones aritméticas (`+`, `-`, `*`)
- Operador `is/2`

2.2 Strings

También podemos trabajar con cadenas de caracteres, sobre todo cuando necesitamos modelar un individuo que tiene espacios:

```
escritor("Jorge Luis Borges").
escritor("Julio Cortázar").
escritor("Elsa Bornemann").
```

3 Pattern matching

La búsqueda exacta de patrones al enviar argumentos o pattern matching que hemos visto en el Paradigma Funcional también lo encontramos en el Paradigma Lógico, pero debemos estar al tanto de la diferencia entre una función y una relación.

Consideremos el siguiente ejemplo en Haskell:

² Nuevamente, para el lector curioso el análisis completo lo puede ver en la Unidad 6: Inversibilidad

```
valor 0 = 1
valor numero = numero
```

Sabemos que no se puede consultar

```
Main> valor x
```

Pero sí

```
Main> valor 0
1
```

El patrón encaja en la primera expresión posible: en este caso el patrón es el valor 0. El segundo patrón no se considera, dado que en Haskell es importante respetar el concepto de unicidad que tiene la función matemática.

Veamos el mismo ejemplo en Prolog

```
valor(0, 1).
valor(Incognita, Incognita).
```

Al hacer la consulta

```
?- valor(0, Cual).
Cual = 1 ;
Cual = 0.
```

Vemos que 0 unifica:

- con el valor 0 de la primera definición del predicado valor/2
- también con la variable Incognita, que matchea cualquier valor.

¿Por qué se produce esto? Porque una relación es más abarcativa que una función, entonces el motor prueba unificar cada uno de los patrones existentes. Entonces vemos que si bien tanto el paradigma funcional como el lógico tienen pattern matching, en el primero se devuelve la primera expresión donde el patrón coincida, mientras que en lógico por la naturaleza del paradigma se buscan todos los individuos que satisfacen un predicado.

Prolog no hace chequeo de tipos, cualquier individuo encaja como patrón de una variable sin ligar:

```
?- valor("cero", Cual).
Cual = "cero".
```

Es decir, la variable Incognita puede ser cualquier individuo, un número, un string, un booleano, etc.

4 Individuos compuestos

4.1 Listas

Las listas representan una serie de elementos ordenados, que pueden repetirse.

- `[]` representa una lista sin elementos, o lista vacía
- `[borges]` representa una lista con un solo elemento
- `[1, 4, 9, 10]` representa una lista con 4 elementos, donde el primer elemento es 1
- En una lista pueden convivir todo tipo de individuos: `[8, hermanos, 6.0, "jeje"]` aunque hay que ver qué hacemos con una lista así

La lista es una estructura recursiva definido de la siguiente manera:

- el caso base es la lista vacía, que se denota `[]`
- el caso recursivo es una lista con al menos un elemento, que se divide en el primer elemento o cabeza y el resto llamado cola, que es una lista (aquí vemos la definición recursiva). Se denota con el patrón: `[Cabeza|Cola]`

En el ejemplo anterior: `[1, 4, 9, 10]` es una lista cuya cabeza es 1 y cola es `[4, 9, 10]`.

4.2 Operaciones básicas sobre listas

A continuación vamos a aprender algunos predicados que serán útiles a la hora de trabajar con listas. No nos interesa entender cómo se implementan, sino saber utilizarlos convenientemente para resolver requerimientos.

4.2.1 length

`length/2` es un predicado que relaciona una lista con su longitud.

```
?- length([picasso, vanGogh, dali], 3).  
true.
```

```
?- length([picasso, vanGogh, dali], Cantidad).  
Cantidad = 3.
```

Si solo conozco la longitud de una lista, ¿tiene sentido esta consulta?


```
?- length(Lista, 3).
```

Bueno, eso depende de si puedo conocer los individuos que forman el universo de elementos posibles para esa lista. En general, length/3 se puede considerar como un predicado **inversible**.

4.2.2 member

Member/2 verifica si un elemento está en una lista. Algunas consultas posibles:

```
?- member(5, [5, 8]).
```

```
true
```

```
?- member(Valor, [5, 8]).
```

```
Valor = 5 ;
```

```
Valor = 8.
```

```
?- member(4, [5, 8]).
```

```
false
```

Si la lista es una incógnita, el predicado member/2 pierde sentido:

```
?- member(4, Lista).
```

La lista podría ser cualquier lista que tenga un 4...

Member/2 es un predicado inversible para el primer argumento.

4.2.3 append

El predicado append/3 permite relacionar dos listas con la lista concatenada.

```
?- append([1, 3], [2], Resto).
```

```
Resto = [1, 3, 2].
```

```
?- append([1, 3], [2], [1, 2, 3]).
```

```
false.
```

```
?- append([1, 3], SegundaLista, [1, 3, borges]).
```

```
SegundaLista = [borges].
```

```
?- append(PrimeraLista, SegundaLista, [1, 3, borges]).
```

```
PrimeraLista = [],
```

```
SegundaLista = [1, 3, borges] ;
```

```
PrimeraLista = [1],
```

```
SegundaLista = [3, borges] ;
```

```

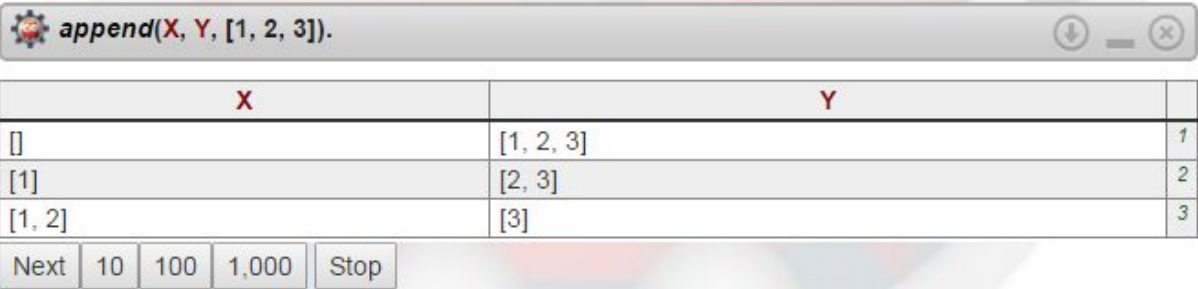
PrimeraLista = [1, 3],
SegundaLista = [borges] ;
PrimeraLista = [1, 3, borges],
SegundaLista = [] ;
false.

```

Como vemos Prolog puede satisfacer cuáles son los individuos que cumplen esta relación de diferentes maneras.

El predicado `append/3` es inversible y tiene sentido para estos casos:

- el primer argumento como incógnita
- el segundo argumento como incógnita
- el tercer argumento como incógnita
- el primer y segundo argumento como incógnita:



The screenshot shows a Prolog interpreter window with the query `append(X, Y, [1, 2, 3]).` and a table of results. The table has three columns: X, Y, and a result index. The results are as follows:

X	Y	
[]	[1, 2, 3]	1
[1]	[2, 3]	2
[1, 2]	[3]	3

Below the table, there are buttons for 'Next', '10', '100', '1,000', and 'Stop'.

4.2.4 nth/3

Este predicado relaciona un elemento con la posición que ocupa en una lista. Tenemos dos predicados: `nth0/3` que toma el índice partiendo de 0 y `nth1/3` que considera el índice partiendo de 1 como primer elemento.

```

?- nth0(2, [borges, cortazar, bioy], Elemento).
Elemento = bioy.

```

```

?- nth0(Posicion, [borges, cortazar, bioy], cortazar).
Posicion = 1.

```

```

?- nth1(2, [borges, cortazar, bioy], Elemento).
Elemento = cortazar.

```

```

?- nth1(Posicion, [borges, cortazar, bioy, borges, marechal],
borges).
Posicion = 1 ;
Posicion = 4 .

```

4.2.5 last/2

Last relaciona una lista con su último elemento

```
?- last([1, 2, 3, 4, 5], Elemento).  
Elemento = 5.
```

```
?- last([1, 2, 3, 4, 5], 4).  
false
```

4.2.6 reverse/2

Este predicado se verifica si los elementos de una lista están al reverso en la segunda.

```
?- reverse(X, [1, 2, 3]).  
X = [3, 2, 1] .
```

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

```
?- reverse([1, 2, 3], X).  
X = [3, 2, 1]
```

4.2.7 Otros predicados interesantes

- sum_list/2: relaciona una lista de números con el total que suman esos números.

```
?- sum_list([7, 2, 6], Total).  
Total = 15.
```

- list_to_set/2: relaciona una lista de elementos repetidos con un conjunto sin repetidos.

```
?- list_to_set([2, 3, 2, 2, 3], Conjunto).  
Conjunto = [2, 3].
```

- max_member/2: relaciona el mayor de una lista de elementos. También existe min_member/2

```
?- max_member(Mayor, [borges, auster, tolstoi]).  
Mayor = tolstoi.
```

- subset/2: relaciona un subconjunto de un conjunto de elementos.

```
?- subset([1, 2], [1, 2, 3]).
true.
```

- y también tenemos operaciones de conjuntos: intersection/3, union/3, subtract/3. Recomendamos leer [este material](#).

4.3 Pattern matching con listas

Hemos visto anteriormente que al utilizar una variable como patrón de un predicado permite matchear cualquier individuo.

```
valor(0, 1).
valor(Incognita, Incognita).
```

Esto incluye las listas

```
?- valor([1, 5], Valor).
Valor = [1, 5].
```

Pero además las listas incorporan algunos patrones adicionales,

Pattern	Explicación
[]	una lista vacía. Para [], el patrón coincide. Para una lista con elementos, el patrón no coincide.
[Elemento]	una lista con un solo elemento Para [8], Elemento vale 8 (el número 8). Para [], el patrón no se satisface. Para [1, 2], el patrón no se satisface.
[Numero Numeros]	una lista con al menos un elemento que tiene cabeza y cola. Para [1, 2, 3], la cabeza es el elemento 1, y la cola = [2, 3]. Para [1], la cabeza es el elemento 1, y la cola [] (la lista vacía) Para [], el patrón no coincide.

Así podemos escribir nuestro propio predicado head/2, que relaciona una lista con su primer elemento:

```
head([ Cabeza | _ ], Cabeza).
```

4.4 Functores

Los funtores permiten agrupar información relacionada.

```
nacio(karla, fecha(22, 08, 1979)).
compro(cliente(231024, "Nelson Pedernera"),
        producto(pirufio, 239, 1)).
```

nacio/2 es un predicado que relaciona dos átomos: karla que es un individuo simple, y fecha/3 que es un functor de aridad 3.

Un functor

- no es un predicado, si bien tienen formatos similares el functor no tiene un valor de verdad, no puedo preguntar ?- fecha(22, 08, 1979)
- tiene un nombre y una aridad que es la cantidad de individuos que lo componen
 - corolario: denota un individuo compuesto
 - dos funtores con distinta aridad representan dos abstracciones diferentes
 - es equivalente a la tupla de Funcional, pero el nombre o prefijo permite entender más claramente el origen de los individuos que participan del functor (le da mayor expresividad)
- es especialmente útil cuando necesitamos trabajar elementos heterogéneos, que tienen algo común pero diferente información entre sí
- puede relacionar átomos, otros funtores o listas

4.5 Pattern matching con funtores

Recordemos una vez más el ejemplo de pattern matching inicial:

```
valor(0, 1).
valor(Incognita, Incognita).
```

Sabemos que una variable se ajusta perfectamente a cualquier individuo, sea simple o compuesto:

```
?- valor(fecha(20, 2, 1988), Valor).
Valor = fecha(20, 2, 1988).
```

Además, podemos usar pattern matching sobre los átomos de un functor. Si tenemos este predicado:

```
nacio(karla, fecha(22, 08, 1979)).
nacio(sergio, fecha(14, 10, 1986))
```

Podemos consultar quién nació en 1986:

```
?- nacio(Quien, fecha(_, _, 1986)).
Quien = sergio.
```

O en qué año nació alguna persona:

```
?- nacio(_, fecha(_, _, Anio)).
Anio = 1979 ;
Anio = 1986.
```

Lo que no se puede es tratar de relacionar como variable el nombre del functor:

```
?- nacio(_, Functor(_, _, Anio)). // no funciona Functor
```

5 Ejemplo integrador

Dada la siguiente información de gente que realiza actividades deportivas:

```
% natacion: estilos (lista), metros nadados, medallas
practica(ana, natacion([pecho, crawl], 1200, 10)).

% fútbol: medallas, goles marcados, veces que fue expulsado
practica(deby, futbol(2, 15, 5)).

% rugby: posición que ocupa, medallas
practica(zaffa, rugby(pilar, 0)).
practica(luis, natacion([perrito], 200, 0)).
practica(vicky, natacion([crawl, mariposa, pecho, espalda], 800, 0)).
practica(mati, futbol(1, 11, 7)).
```

Aclaraciones:

- para la natación sabemos los estilos que nada, la cantidad de metros diarios que recorre, y la cantidad de medallas que consiguió a lo largo de su carrera deportiva
- para el fútbol primero conocemos las medallas, luego los goles convertidos y por último las veces que fue expulsado
- para el rugby, queremos saber la posición que ocupa y luego la cantidad de medallas obtenidas

5.1 Medallas obtenidas

Si queremos saber cuántas medallas tiene alguien, debemos relacionar primero una persona con el deporte que practica...

```
medallas(Alguien, Medallas):-
    practica(Alguien, Deporte),
    cuantasMedallas(Deporte, Medallas).
```

... y luego trabajar con pattern matching en base a cada functor:

```
cuantasMedallas(natacion(_, _, Medallas), Medallas).
cuantasMedallas(futbol(Medallas, _, _), Medallas).
cuantasMedallas(rugby(_, Medallas), Medallas).
```

El predicado medallas es inversible en sus dos argumentos, porque practica/2 funciona como predicado **generador**: permite conocer el universo de deportistas:

```
medallas(Alguien, Medallas):-
    practica(Alguien, Deporte),
    cuantasMedallas(Deporte, Medallas).
```

En rojo vemos que Alguien y Medallas pueden ser incógnitas, pero el predicado practica es un hecho, por lo tanto liga las variables Alguien y Deporte (en verde). Luego buscamos satisfacer la consulta cuantasMedallas/2 con el primer argumento instanciado. Y como vemos a continuación el predicado cuantasMedallas/2 es inversible para el segundo argumento: si el primer argumento está instanciado es posible resolver el segundo argumento como incógnita

```
?- cuantasMedallas(rugby(hooker, 10), Medallas).
Medallas = 10.
```

```
?- cuantasMedallas(natacion([crawl, espalda, mariposa], 1800, 6),
Medallas).
Medallas = 6.
```

Por lo tanto, Medallas se puede conocer (pasa a estar en verde).

5.2 Buen deportista

Quiero saber si alguien es buen deportista

- en el caso de la natación, si recorren más de 1.000 metros diarios o nadan más de 3 estilos
- en el caso del fútbol, si la diferencia de goles menos las expulsiones suman más de 5
- en el caso del rugby, si son wings o pilares

```
buenDeportista(Alguien):- practica(Alguien, Deporte), esBueno(Deporte).
```

El predicado esBueno/1 trabaja una vez más con pattern matching...

```
esBueno(natacion(Estilos, _, _)):-
    length(Estilos, CantidadEstilos),
    CantidadEstilos > 3.
esBueno(natacion(_, Kms, _)):-Kms > 1000.
esBueno(futbol(_, Goles, Expulsiones)):-
    Valor is Goles - Expulsiones, Valor > 5.
esBueno(rugby(pilar, _)).
esBueno(rugby(wing, _)).
```

Las últimas dos cláusulas también pueden escribirse de esta manera:

```
esBueno(rugby(Puesto, _)):-member(Puesto, [pilar, wing]).
```

¿Qué pasaría si no trabajáramos con funtores? Necesitaríamos diferentes aridades para el predicado esBueno. En cambio...

```
?- buenDeportista(Alguien).
Alguien = ana ;
Alguien = deby ;
Alguien = zaffa ;
Alguien = vicky ;
false.
```

...estamos trabajando con individuos intercambiables o **polimórficos**. En algún punto rugbiers, futbolistas y nadadores se pueden trabajar en conjunto y eso para el negocio tiene sentido. Incluso, si incorporamos un deporte más

```
% nos interesa el handicap del polista,
% no tiene medallas porque no importa
practica(gise, polo(8)).
```


Alguien que practica polo es bueno si tiene un handicap mayor a 6. Incorporamos a la base esta definición:

```
esBueno(polo(Handicap)) :- Handicap > 6.
```

el que se beneficia es el predicado buenDeportista, que no va a verse impactado: la definición original sigue valiendo

```
buenDeportista(Alguien) :- practica(Alguien, Deporte), esBueno(Deporte).
```

6 Resumen

Dentro del Paradigma Lógico podemos modelar la información a través de individuos, que pueden ser átomos simples: un número, un string, un booleano, o bien átomos compuestos: como las listas y los funtores.

También hemos visto que el pattern matching -a diferencia de funcional- permite relacionar todos los valores que coincidan con un patrón, sea un valor específico, una incógnita general, o bien alguno de los formatos definidos para las listas o los funtores.