Módulo 1: Introducción al Paradigma Lógico

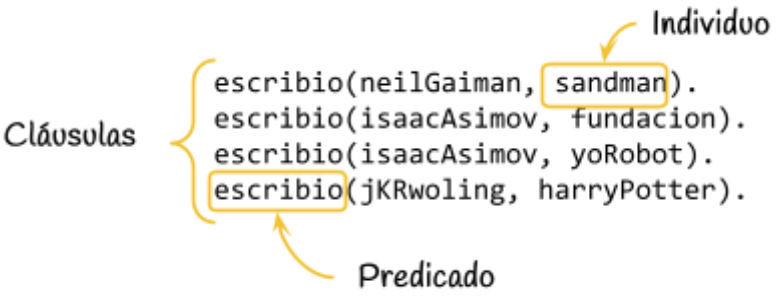
Base de Conocimientos: define el alcance, lo que forma parte del universo conocido (cláusulas que componen a un predicado, pueden ser hechos o reglas). Lo que esta afuera se asume falso según el ppio de universo cerrado. La base de conocimiento se conforma de predicados, estos relacionan individuos.

EJEMPLO:

```
pastas(ravioles).
pastas(fideos).
```

Hecho: declaran verdades por extensión. Son afirmaciones, las asumimos como verdaderas (axioma). Cada uno es una cláusula y no requieren antecedentes.

Individuo: elto que forma parte del universo posible de los predicados. *Por ej:* `ravioles` o `fideos`.



Aridad: cant de individuos que tiene un predicado. Si tiene aridad 1, se trata de una *propiedad*; si tiene 2+ se trata de una *relación*. *Por ej:* pastas tiene aridad 1 y se denota pastas/1.

EJEMPLO:

```
padre(homero, bart).
padre(homero, lisa).
padre(homero, maggie).
padre(ned, rod).
padre(ned, todd).

esInteligente(lisa).

hermano(Persona1, Persona2):-
   padre(Padre, Persona1),
   padre(Padre, Persona2).
```

- padre/2 es un predicado. Es una relación entre 2 individuos y está compuesto por 5 cláusulas, que son hechos.
- esInteligente/1 es un predicado. Es una prop y está compuesta por 1 cláusula, que es un hecho.
- hermano/2 es un predicado. Es una relación entre 2 individuos y está compuesta por 1 cláusula, que es una regla.
- homero, bart, lisa, maggie, ned, rod y todd son átomos.
- Persona1, Persona2 y Padre son variables.

Predicados Monádicos / Propiedades: aquellos con un solo arg, expresan características o atributos de individuos.

Predicados Políadicos / Relaciones: aquellos con más de un arg, expresan relaciones entre individuos. Conviene escribir las defs de un mismo predicado de forma contigua, y no intercalar con otros predicados.

EJEMPLO:

```
come(juan, ravioles).
come(brenda, fideos).
gusta(brenda, fideos).
```

Ppio de Universo Cerrado

Hay dos opciones cuando se define una base de conocimientos:

- Ppio de Universo Abierto: considerar 3 estados posibles: cierto, falso y desconocido.
- Ppio de Universo Cerrado: considerar todo lo desconocido como falso, por ej: Prolog. Todos los hechos que no se encuentran en la base de conocimiento, se consideran falsos.

Aplicación: aquí son soluciones declarativas porque

- Declaro conocimiento a través de predicados, características de los individuos o las relaciones entre ellos.
- En las consultas, un motor de inferencia permite sacar conclusiones a partir de ese conocimiento. Es él, el que termina construyendo el algoritmo para resolver las consultas. Encuentra las soluciones mediante *backtracking*.

Definición por Extensión: un cto de hechos para el mismo predicado forman su def por extensión.

EJEMPLO:

```
animal(tigre).
animal(oso).
animal(elefante).
```

Es como decir *Animales = {tigre, oso, elefante}.*

Desventaja:

- Requiere la enumeración de todos los eltos que componen el cto. Tedioso para ctos grandes.
- No brinda info referente al cto más allá de sus eltos directos.

Consultas: al ejecutar un prog lógico, en la consola podemos hacer consultas que toman como entrada las defs de la base de conocimiento para extraer conclusiones. Se agrega el punto para delimitar el final de una consulta. Las hacemos para probar nuestro prog. No devuelven nada, sino que tienen un valor de verdad (cierta o falsa).

EJEMPLO:

```
? pastas(ravioles).
true
? pastas(fideos).
true
```

Consultas sobre Predicados Poliádicos

- Deben respetar la cant de argumentos con la que se definió el predicado.
- Imp el orden de los argumentos. Las relaciones no son bidireccionales por defecto, imp a la hora de definir; más que nada porque el sentido de los args se los damos nosotros.

Módulo 2: Conceptos

Variables: permiten modelar una incógnita. Podemos decir:

- Los individuos están *instanciados*. Comienzan con una minúscula y son valores.
- Las variables están *libres*. Comienzan con mayúscula y son incógnitas.

EJEMPLO ANTERIOR:

```
pastas(ravioles).
pastas(fideos).
```

Al consultar, podemos preguntar si algún fideo en particular es una pasta, o qué pastas hay en nuestra base de conocimientos.

```
? pastas(Pasta).
Pasta = ravioles _
```

`Pasta` sería la incógnita. Usarla indica que no conocemos los individuos que satisfacen la relación pastas/1, y los queremos determinar. La consulta se podría leer: *"¿cuáles son los individuos que satisfacen el predicado pastas/1?"*

Variables Anónimas: si el obj es determinar si existe algún individuo que cumpla la relación, pero no es necesario saber cuál, usamos el guión bajo (_) → variable anónima.

EJEMPLO:

```
? pastas(_).
true
```

Múltiples Soluciones: el motor de inferencia no solo infiere si una relación se satisface para cierto individuo, sino también cuál es el universo de individuos que la cumplen. Para obtener más soluciones, se puede presionar <;>, <n> *(next)*, <espacio> o <tab>; para detener la búsqueda <.> o <Enter>. Al final de todo aparece `false`, ya que no puede encontrar más individuos que cumplan la relación dada.

EJEMPLO:

```
? come(Persona, ravioles).
Persona = juan ;
Persona = melina ;
false.
```

Tipos de Consulta

- **Consultas Individuales:** para determinar si una relación o prop específica se satisface o no, instanciando todos sus args. Se verifican `true` o `false`.

```
? come(juan, ravioles).
? pastas(bohio).
? pastas(_).
```

- **Consultas Existenciales:** para conocer los individuos que satisfacen una relación; al menos uno de sus args está libre. El motor de inferencia unifica los posibles individuos que satisfacen la relación, si existen.

```
? come(juan, Comida).
? come(Persona, Comida).
? pastas(Pasta).
```

Inversibilidad: prop de un predicado que admite consultas individuales Y existenciales (podemos consultar con incógnitas en sus args). Para los hechos no hay restricciones, así que, por ej, come/2 y pastas/1 son totalmente inversibles.

Hay predicados donde no es posible hacer consultas existenciales para todos sus args, pero si son inversibles para el 1er, 2do, ..., argumento.

Unificación: en Prolog no existe el concepto de Asignación, sino Unificación → una variable se resuelve como incógnita con uno o más valores.

Las consultas NO DEVUELVEN NADA; pueden satisfacerse o no, encontrando individuos que cumplen el predicado, pero un predicado no "devuelve" valores. No son expresiones. Por esta razón, `x = x + 1` es una **condición**, no una asignación (estaría mal incluso, debería ser `x is x + 1`).

★ Regla: otro tipo de predicado. Declaran verdades por compresión. Son implicaciones, su valor de verdad depende del valor de verdad de su(s) antecedente(s). Tiene:

- Uno o más antecedentes (condiciones previas)
- Un consecuente.

Si se cumplen los antecedentes, se satisface el consecuente. Lo que en lógica se escribe $p \Rightarrow q$, en sintaxis prolog se escribe: `q :- p` (*consecuente :- antecedente(s)*). Esta sintaxis se llama *Cláusula de Horn*.

EJEMPLO:

1) Todo humano es mortal.	*(cuantificación universal)*	`mortal(Persona):-` `  humano(Persona),` `  humano(socrates).`
2) Sócrates es humano.	*(cuantificación particular)*	
3) Sócrates es mortal.	*(cuantificación particular)*	

No tiene sentido escribir `mortal(sócrates)` porque se infiere de la base de conocimientos (ventaja).

Reglas Compuestas

CONJUNCIONES (AND)	DISYUNCIONES (OR)
≫ Conector lógico AND se representa con una coma. ≫ Representa reglas del estilo $p \wedge q \Rightarrow r$.	≫ Se repite la cláusula en dos reglas diff. ≫ Representa reglas de tipo $p \vee q \Rightarrow r$.

EJEMPLO: *"Cualquier docente que vive en Lanús es afortunado"*

```
viveEn(tefi, lanus).
viveEn(gise, lanus).
viveEn(alf, lanus).
viveEn(dodain, liniers).
```

```
docente(alf).
docente(tefi).
docente(gise).
docente(dodain).
```

```
afortunado(Persona):-
      docente(Persona),
      viveEn(Persona, lanus).
```

Entonces

≫ La variable persona se unifica para todos los individuos que satisfacen docente/1.

≫ Una vez unificada la variable, al tratar de satisfacer viveEn/2 ya no hay incógnitas: Persona es un valor conocido (el encontrado como solución de docente(Persona)), y lanús es un individuo.

≫ Este predicado es inversible.

EJEMPLO: *"Si una persona es docente o vive en Lanús, es afortunada"*

```
afortunado(Persona):-
     docente(Persona).

afortunado(Persona):-
     viveEn(Persona, lanus).
```

Si se cumple cualquiera de las dos ramas (docente/1 o viveEn/2), la regla afortunado/1 se satisface.

Si pensamos en afortunado/1, docente/1 y viveEn/2 como componentes, vemos que afortunado depende de las defs de docente y viveEn, es decir hay un grado de acoplamiento entre ellos. Cualquier modificación en docente o viveEn impactan a afortunado.

Definición por Comprensión: definir un cto a través de una regla.

Módulo 3: Estructuras de Datos e Individuos

★ **Individuos Simples:** Tipo de dato simple, atómico (no puede descomponerse en otros eltos). Participan en los predicados y podemos compararlos.

EJEMPLO: ravioles, juan y fideos son individuos.

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

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

Números: se utilizan como valores literales (enteros y decimales).

EJEMPLO: ravioles, juan y fideos son individuos.

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

?- 6 > 3.2.
true.
```

Además de poder compararse por igualdad/desigualdad, definen:

- Operadores Aritméticos: suma *(valor + n)*, resta *(valor – n)*, multiplicación *(valor * n)* y división *(valor / n)*, valor absoluto *(abs(n))*, etc.
- Ops de Comparación por orden: < , =<, >, >=.

Con ellos y con is/2 sufrimos restricciones de inversibilidad, por ej no podemos consultar individuos mayores a un nro.

is/2: Relaciona un nro (lado izq) con una op (lado der) que se evalúa → nro is op. No puede haber una cuenta en la izq, y tampoco pueden ser las dos variables.

EJEMPLO:

```
? 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
```

EJEMPLO:

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

? siguiente(2, 3).
true

? siguiente(4, Numero).
Numero = 5 _

? siguiente(Numero, 4)
ERROR: is/2: Arguments are not sufficiently instantiated
```

String: cadena de caracteres, útil al modelar un individuo que tiene espacios.

EJEMPLO:

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

Pattern Matching: tanto funcional como lógico lo poseen, sin embargo su diff es que una relación es más abarcativa que una función (no tiene tantas limitaciones).

En el 1ero, se devuelve la 1era expresión que coincida con el patrón, mientras que en el 2do se buscan todos los individuos que satisfacen el predicado (unificar cada uno de los patrones existentes). Además, en lógico no se sufren restricciones de tipos, una incógnita puede ser un individuo, un nro, un string, un booleano, etc.

EJEMPLO:

FUNCIONAL	LÓGICO
`valor 0 = 1` `valor numero = numero` No se podría consultar `> valor x` Pero sí `> valor 0 → 1` El patrón encaja en la 1era expresión posible. El segundo no se considera, dado que no cumpliría unicidad.	`valor(0, 1).` `valor(Incognita, Incognita).` `?- valor(0, Cual).` `Cual = 1 ;` `Cual = 0.`

★ Individuos Compuestos

Lista: representa una serie de eltos ordenados, que pueden repetirse. Pueden convivir todo tipo de individuos, por ej: *[8, hermanos, 6.5, "hola"]*. Es una estructura recursiva definida de la sgte manera:

- Caso base → [ ]
- Caso recursivo → lista con al menos un elto, que se divide en cabeza (1er elto) y cola (el resto, otra lista). Se denota con el patrón: [Cabeza | Cola]

EJEMPLO: [1 , 4, 9, 10] donde la cabeza es 1, y la cola es [4, 9, 10]

OPERACIONES BÁSICAS SOBRE LISTAS	
length/2	Relaciona una lista con su longitud. `?- length([picasso, vanGogh, dali], 3).` `true.`

	`` ?- length([picasso, vanGogh, dali], Cantidad). Cantidad = 3. ```
member/2	Verifica si un elto está en una lista. Solo es inversible para el 1er arg, no tendría sentido que sea completamente inversible. ``` ?- member(5, [5, 8]). true ?- member(Valor, [5, 8]). Valor = 5 ; Valor = 8. ?- member(4, [5, 8]). false ```
append/3	Relaciona dos listas con su lista concatenada. Es inversible y tiene sentido para estos casos: 1A, 2A, 3A, 1A y 2A. ``` ?- 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. ```
nth/3	Relaciona un elto con la posición que ocupa en una lista. Hay dos alternativas posibles: nt0/3 (índice parte de 0) y nth1/3 (índice parte de 1). ``` ?- 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 . ```
last/2	Relaciona una lista con su último elto.

	?- last([1, 2, 3, 4, 5], Elemento). Elemento = 5. ?- last([1, 2, 3, 4, 5], 4). false
reverse/2	Verifica si los eltos 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]
otros	≫ sumlist/2: relaciona una lista de nros con el total que suman. ?- sum_list([7, 2, 6], Total). Total = 15. ≫ list_to_set/2: relaciona una lista de eltos repetidos con un cto sin repetidos. ?- list_to_set([2, 3, 2, 2, 3], Conjunto). Conjunto = [2, 3]. ≫ max_member/2: relaciona el mayor de una lista de eltos. También existe min_member/2. ?- max_member(Mayor, [borges, auster, tolstoi]). Mayor = tolstoi. ≫ subset/2: relaciona un subcto con su cto de eltos. ?- subset([1, 2], [1, 2, 3]). true. ≫ También hay ops de ctos: intersection/3, union/3, subtract/3, etc.

PM con Listas

PATTERN	EXPLICACIÓN
[ ]	Lista vacía.
[Elemento]	Lista con un solo elemento ≫ [8] → patrón se satisface. ≫ [] → patrón no se satisface. ≫ [1,2] → patrón no se satisface.
[Cabeza | Cola]	Lista con 1+ eltos. ≫ [1, 2, 3] → cabeza es 1, cola es [2, 3]. ≫ [1] → cabeza es 1, y cola es [ ] ≫ [ ] → patrón no coincide.

Functores: permiten agrupar info relacionada, denotan un individuo compuesto (una abstracción). Útil cuando tenemos eltos heterogéneos, relacionados de cierta manera.

≫ No se trata de un predicado, ya que no tiene valor de verdad. No puede hacerse consultas con él.

- Tiene un nombre y una aridad.
- Puede relacionar átomos, otros functores o listas.

EJEMPLO:

```
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 (individuo simple) y fecha/3 (functor de aridad 3).

PM con Functores: Cumplen el PM común, y además se puede generar PM sobre los átomos de un functor.

EJEMPLO:

- Común:

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

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

- Nuevo: dado 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 una 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
```

EJEMPLO INTEGRADOR:

```
% natacion: estilos (lista), metros nadados, medallas
practica(ana, natacion([pecho, crawl], 1200, 10)).
practica(luis, natacion([perrito], 200, 0)).
practica(vicky,
   natacion([crawl, mariposa, pecho, espalda], 800, 0)).

% fútbol: medallas, goles marcados, veces que fue expulsado
practica(deby, futbol(2, 15, 5)).
practica(mati, futbol(1, 11, 7)).
```

```
% rugby: posición que ocupa, medallas
practica(zaffa, rugby(pilar, 0)).

/* Nadadores son aquellos que practican deporte, donde ese deporte es la natación.
Primero, trabajamos con el predicado práctica, para luego hacer PM con el functor y
así poder hacer consultas inversibles. */

nadadores(Quien):-
practica(Quien, Deporte),
nadador(Deporte).

nadador(natacion(_, _, _)). %PM con el functor

/* Medallas es completamente inversible, porque practica/2 funciona como predicado
generador: permite conocer el universo de deportistas.

En rojo, Alguien y Medallas pueden ser incógnitas, pero el predicado práctica es un
hecho, por lo que liga las variables Alguien (en verde) y Deporte (en amarillo).

Luego, buscamos satisfacer la consulta cuantasMedallas/2, con el primer arg
instanciado. Entonces, este predicado es inversible para el 2do arg: si l 1ero está
instanciado, es posible resolver el 2do como incógnita. */

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

cuantasMedallas(natacion(_, _, Medallas), Medallas). %PM con el functor
cuantasMedallas(futbol(Medallas, _, _), Medallas).   %PM con el functor
cuantasMedallas(rugby(_, Medallas), Medallas).       %PM con el functor
```

Módulo 4: Predicados de Orden Superior

Predicados de Orden Superior: reciben predicados de args.

Si queremos volverlos inversibles, debemos fijar el dominio para los args (*"el primer arg no puede ser cualquier cosa, debe ser un ..."*). Lo hacemos a través de un **predicado generador**, que debe ser inversible, y genera el universo de sus individuos posibles. Se hace en:

- NOT → porque no es posible determinar los individuos que no satisfacen un predicado si no conocemos todo el universo.
- FINDALL y FORALL → porque debemos estar atentos a las variables libres y ligadas que participan en ellas.

EJEMPLO:

```
yeta(Numero):-
not(juega(_, Numero)).

yeta(Numero):-
```

```
numeroRuleta(Numero),
not(juega(_, Numero)).

perroComun(Perro):-
perro(Perro),
not(perroFamoso(Perro)).

terminoAnio(Alumno, Anio):-
     alumno(Alumno),
     anio(Anio),
     forall(materia(Materia, Anio), aprobo(Alumno, Materia)).
```

★ **not/1:** permite invertir el valor de verdad de un predicado. *Si se quiere negar la ocurrencia de dos predicados, se los debe colocar con paréntesis después del not.*

EJEMPLO:

```
?- not((10 > 3, 10 > 2).
false.
```

Con él, para volver inversible un predicado hay dos formas: mediante un predicado generador, o definir el predicado por extensión (transformando un predicado negativo en uno asertivo). En ambos casos, la desventaja es que se desconoce el cto universal.

En vez de implementar perroComun/1 como regla, se genera una serie de hechos: `perroComun(chicho)`, `perroComun(sultan)`, *etc.*

★ **forall/2:** verifica que una condición se cumpla para todas las variables posibles.

EJEMPLO: *"Un alumno terminó un año si aprobó todas las materias de ese año."*

```
materia(algoritmos, 1).
materia(analisisI, 1).
materia(pdp, 2).
materia(proba, 2).
materia(sintaxis, 2).

alumno(Alumno):-nota(Alumno, _ , _).
%def for comprensión
```

```
nota(nicolas, pdp, 10).
nota(nicolas, proba, 7).
nota(nicolas, sintaxis, 8).
nota(malena, pdp, 6).
nota(malena, proba, 2).
nota(raul, pdp, 9).
```

≫ 1era Forma: No inversible.

```
terminoAnio(Alumno, Anio):-
     forall(materia(Materia, Anio),
          (nota(Alumno, Materia, Nota), Nota >= 6)).
```

≫ 2da Forma: Inversible (solo 1er arg). Se puede preguntar qué alumnos aprobaron todas las materias de **un** año. Si se fuera a hacer `termino(Alumno, Anio)`, se obtendrán los alumnos que aprobaron **todas** las materias de **todos** los años (se recibieron).

```
terminoAnio(Alumno, Anio):-
      alumno(Alumno),
      forall(materia(Materia, Anio), aprobo(Alumno, Materia)).
```

≫ 3er Forma: Totalmente inversible. De esta forma se puede preguntar qué alumnos aprobaron todas las materias de un año y en qué año lo hicieron.

```
terminoAnio(Alumno, Anio):-
     alumno(Alumno),
     anio(Anio),
     forall(materia(Materia, Anio), aprobo(Alumno, Materia)).

aprobo(Alumno, Materia):-
nota(Alumno, Materia, Nota),
Nota >= 6.
```

EJEMPLO

≫ Incluido: cto A está incluido en cto B si todos los eltos de A están en B.

```
incluido(A, B):-
forall(member(X, A), member(X, B)).
```

≫ Disjunto: cto A es disjunto de B si todos los eltos de A no están en B.

```
disjuntos(A, B):-forall(member(X, A), not(member(X, B)))
```

Not y Forall: *Si todos los individuos que cumplen p, también cumplen q* → podemos decir que no es verdad que exista un individuo que cumpla p que no cumpla q.

En términos lógicos: $\forall\ p(x) \Rightarrow q(x)$ equivale a $\nexists\ x / p(x) \wedge \neg q(x)$

Si todos los individuos que cumplen p, no cumplen q → no es verdad que exista un individuo que cumpla p y q a la vez.

En términos lógicos: $\forall\ p(x) \Rightarrow \neg q(x)$ equivale a $\nexists\ x / p(x) \wedge q(x)$

EJEMPLO: *"Un alumno estudioso es aquel que estudia para todas las materias..."*

```
materia(am1).
materia(sysop).
materia(pdp).
alumno(clara).
```

```
estudio(clara, am1).
estudio(clara, sysop).
estudio(clara, pdp).
estudio(matias, pdp).
```

`alumno(matias).` `alumno(valeria).` `alumno(adelmar).`	`estudio(matias, am1).` `estudio(valeria, pdp).`

```
estudioso(Alumno):-
      alumno(Alumno),
      forall(materia(Materia), estudio(Alumno, Materia)).
```

Se liga alumno y no materia porque quiero probar si **un** alumno estudia para **todas** las materias.

"No es verdad para un alumno estudioso que exista alguna materia en la que no haya estudiado".

```
estudiosoNot(Alumno):-
    alumno(Alumno),
    not((materia(Materia), not(estudio(Alumno, Materia)))).
```

Si ahora quiero buscar los alumnos difíciles, descriptos como *"aquellos que no estudian para ninguna materia"*, si se podría considerar la utilización del not:

```
dificil(Alumno):-
alumno(Alumno),
not(estudio(Alumno, _)).
```

★ **findall/3:** predicado que relaciona un individuo o variable, con una consulta y con el cto (lista) de los individuos que satisfacen a la consulta. El 2do arg puede ser cualquier tipo de consulta, no necesariamente un único predicado. Si se quiere hacer 1+, solo hay que encerrar entre paréntesis las cláusulas para no cambiar la aridad.

EJEMPLO:
```
findall(UnIndividuoOVariable, Consulta, Conjunto)
```

EJEMPLO: *¿Cuántos hijos tiene Homero?*
```
padre(homero,bart).
padre(homero,maggie).
padre(homero,lisa).

cantidadDeHijos(Padre, Cantidad) :-
      findall(Hijo, padre(Padre, Hijo), Hijos),
      length(Hijos, Cantidad).
```

Inversibilidad del Findall: el problema llega cuando se tiene más hechos en la base de conocimientos y se quiere volver a los predicados inversibles.

EJEMPLO ANTERIOR:
```
padre(homero,bart).
padre(homero,maggie).
```

```
padre(homero,lisa).
padre(juan, fede).
padre(nico, julieta).

?- cantidadDeHijos(Padre, Cantidad).
Cantidad = 5.
```

En este caso, hijos abarcaría a todos los que haya, no se distingue por padre → Hijos = [bart, maggie, lisa, fede, julieta]. Para solucionar el problema, hay que unificar las defs de Padre, previo al findall.

≫ 1era Opción: En la consulta, Bart cumple el predicado (porque es una persona) y tiene cero hijos (porque no están especificados). Esta opción es mejor.

```
cantidadDeHijos(Padre, Cantidad) :-
      persona(Padre),
      findall(Hijo, padre(Padre, Hijo), Hijos),
      length(Hijos, Cantidad).

?- cantidadDeHijos(bart, Cantidad).
Cantidad = 0
```

POR EXTENSIÓN	POR COMPRENSIÓN
`persona(bart).` `persona(lisa).  % ... etc.`	`persona(Papa) :- padre(Papa, _).` `persona(Hijo) :- padre(_, Hijo).`

≫ 2da Opción: En la consulta, Bart no cumple el predicado (porque no es padre).

```
cantidadDeHijos(Padre,Cantidad) :-
      padre(Padre, _),
      findall(Hijo, padre(Padre, Hijo), Hijos),
      length(Hijos, Cantidad).

?- cantidadDeHijos(bart, Cantidad).
false
```

EJEMPLO: Queremos saber

≫ ¿Cuántos minutos jugó un nene en total?

≫ ¿Cuántos juegos diff jugó?

```
jugoCon(tobias, pelota, 15).
jugoCon(tobias, bloques, 20).
jugoCon(tobias, rasti, 15).
jugoCon(tobias, dakis, 5).
jugoCon(tobias, casita, 10).
jugoCon(cata, muniecas, 30).
jugoCon(cata, rasti, 20).
jugoCon(luna, muniecas, 10).
```

```
juegosQueJugo(Nene, CantidadJuegos):-
    findall(Juego, jugoCon(Nene, Juego, _), Juegos),
    length(Juegos, CantidadJuegos).

minutosQueJugo(Nene, CuantosMinutos):-
nene(Nene),
findall(Minutos, jugoCon(Nene, _, Minutos), ListaMinutos).
```

Consultas:

```
?- juegosQueJugo(tobias, CuantosJuegos).
CuantosJuegos = 5.

?- minutosQueJugo(cata, Minutos).
Minutos = 50.
```

EJEMPLO INTEGRADOR:

```
tiene(juan, foto([juan, hugo, pedro, lorena, laura], 1988)).
tiene(juan, foto([juan], 1977)).
tiene(juan, libro(saramago, "Ensayo sobre la ceguera")).
tiene(juan, bebida(whisky)).
tiene(valeria, libro(borges, "Ficciones")).
tiene(lucas, bebida(cusenier)).
tiene(pedro, foto([juan, hugo, pedro, lorena, laura], 1988)).
tiene(pedro, foto([pedro], 2010)).
tiene(pedro, libro(octavioPaz, "Salamandra")).

premioNobel(octavioPaz).
premioNobel(saramago).
```

Determinamos que alguien es coleccionista si todos los eltos que tiene son valiosos:

- Un libro de un premio nobel es valioso.
- Una foto con más de 3 integrantes es valiosa.
- Una foto anterior a 1990 es valiosa.
- El whisky es valioso.

Planteamos el predicado coleccionista/1 (no inversible) y vale/1:

```
coleccionista(Alguien):-
   forall(tiene(Alguien, Cosa), vale(Cosa)).

vale(foto(Gente, _)):-
length(Gente, Cantidad),
Cantidad > 3.

vale(foto(_, Anio)):-
Anio < 1990.
vale(libro(Escritor, _)):-
premioNobel(Escritor).
```

```
vale(bebida(whisky)).
```

Se podrían hacer consultas individuales, pero no una existencial, porque ahí requeriría verificar que todas las personas tienen todos los eltos valiosos.

```
?- coleccionista(Quien).
true.
```

Para solucionarlo, se requiere utilizar un predicado generador sobre Alguien.

```
coleccionista(Alguien):-
   tiene(Alguien, _),
   forall(tiene(Alguien, Cosa), vale(Cosa)).
```

Módulo 5: Recursividad y Polimorfismo

Predicado Recursivo: aquel que en alguna de sus cláusulas se invoca a sí mismo. Para poder funcionar correctamente necesitan contar con un caso base que corta la recursividad.

EJEMPLO: *¿Quiénes son mis ancestros?* Mis padres y los ancestros de mis padres.

```
padre(tatara, bisa).
padre(bisa, abu).
padre(abu, padre).
padre(padre, hijo).
```

```
ancestro(Padre, Persona):-
     padre(Padre, Persona).

ancestro(Ancestro, Persona):-
    padre(Padre, Persona),
    ancestro(Ancestro, Padre).
```

Esto se lee: "*ancestro es mi padre o bien el ancestro de mi padre*".

```
?-ancestro(tatara, Quien).
Quien = bisa ;
Quien = abu ;
Quien = padre ;
Quien = hijo ;
false.
```

EJEMPLO: *¿Quiénes son mis ancestros?* Mis padres y los ancestros de mis padres.

```
distancia(buenosAires, puertoMadryn, 1300).
distancia(puertoMadryn, puertoDeseado, 732).
distancia(puertoDeseado, rioGallegos, 736).
distancia(puertoDeseado, calafate, 979).
distancia(rioGallegos, calafate, 304).
distancia(calafate, chalten, 213).
```

```
kilometrosViaje(Origen, Destino, Kms):-
     distancia(Origen, Destino, Kms).

kilometrosViaje(Origen, Destino, KmsTotales):-
    distancia(Origen,PuntoIntermedio,KmsIntermedios),
    kilometrosViaje(PuntoIntermedio,Destino,KmsFinales
    ),
    KmsTotales is KmsIntermedios + KmsFinales.

totalViaje(Origen, Destino, Kms):-
   kilometrosViaje(Origen, Destino, Kms).

totalViaje(Origen, Destino, Kms):-
   kilometrosViaje(Destino, Origen, Kms).
```

EJEMPLO: *Factorial*

```
factorial(0,1).

factorial(N,F):-
 N > 0,
 Anterior is N-1,
 factorial(Anterior,F2),
 F is F2*N.
```

Recursividad con Listas

EJEMPLO: *Sumatoria*

```
sumatoria([],0).
sumatoria([Cabeza|Cola], S):-
    sumatoria(Cola,SCola),
    S is SCola + Cabeza.
```

EJEMPLO: *Último*

```
ultimo([E],E).
ultimo([_|Cola],Ultimo):-
          ultimo(Cola,Ultimo).
```

Polimorfismo: permite obtener soluciones más genéricas, que sean válidas para diff tipos de datos, contemplando las particularidades de cada uno de ellos. En gral, decimos que dos cosas son polimórficas cuando desde algún punto de vista comparten un "tipo", es decir pueden ser tratados indistintamente por quienes no les interesan los detalles donde difieren.

Si bien todos los predicados pueden recibir cualquier tipo de argumento en Prolog, muchas veces a partir de su uso se delimita un rango de tipos de valores con sentido. Aunque no genera errores colocar un tipo diff, al hacer consultas va a fallar porque no se puede unificar el valor recibido con lo esperado.

EJEMPLO: *"Se tiene 3 tipos de vehículos: autos, camiones y bicicletas. Un auto es viejo si su patente es menor a F, un camión es viejo si tiene más de 60000km o más de 10 años, una bicicleta es vieja si la fecha de fabricación es de año anterior al 2006"*

```
hoy(fecha(22,12,2008)).
vehiculo(auto("A2000")).
vehiculo(auto("H2342")).
vehiculo(camion(12000,2005)).
vehiculo(camion(70000,2003)).
vehiculo(camion(30000,1997)).
vehiculo(bici(fecha(30,10,2005))).
vehiculo(bici(fecha(20,12,2008))).
```

Alternativa No-Polimórfica:

```
autoViejo(Patente):-
    Patente > "F".
camionViejo(Kilometraje,_):-
    Kilometraje > 60000.
camionViejo(_,Anio):-
    hoy(fecha(_,_,AnioActual)),
    AnioActual - Anio > 10.
biciVieja(fecha(_,_,Anio)):-
    Anio < 2006.

forall(vehiculo(auto(Patente)), autoViejo(Patente)),
forall(vehiculo(camion(Kms, Anio)), camionViejo(Kms, Anio)),
forall(vehiculo(bici(Fecha)), biciVieja(Fecha)).
```

Alternativa Polimórfica:

```
esViejo(auto(Patente)):-
    Patente > "F".
esViejo(camion(Kilometraje,_)):-
    Kilometraje > 60000.
esViejo(camion(_,Anio)):-
    hoy(fecha(_,_,AnioActual)),
    AnioActual - Anio > 10.
esViejo(bici(fecha(_,_,Anio))):-
    Anio < 2006.

forall(vehiculo(Vehiculo), esViejo(Vehiculo)).
```