

Repaso rápido de conceptos:

```
borracho(Persona):-tomo(Persona, Cantidad),
                   resiste(Persona, Resistencia),
                   Cantidad > Resistencia.
```

Borracho, ¿es inversible? Si tomó y resiste son hechos, seguro que es inversible.

Si tomó o resiste son reglas, necesitamos que el primer argumento de ambas reglas puedan ser incógnitas.

¿Dónde se produce unificación?

Si pregunto `borracho(pity)`, unifico `X` con `pity`.

Recordar unificación vs. asignación, unificación relacionado con pattern matching y con backtracking (encontrar todas las soluciones a un problema).

¿Un ejemplo donde se aplique el principio de universo cerrado? Si yo pregunto `borracho(marley)` y no tengo la cantidad de litros que tomó Marley, no puedo afirmar que esté borracho, por lo tanto PROLOG asume que la respuesta a `borracho(marley)` es no.

¿Cómo funciona la negación? Por falla. Esto se relaciona con:

- *Principio de universo cerrado*: si algo no está en la base es porque se presume falso, entonces es un contrasentido escribir las condiciones que no se cumplen:

```
noEsCopado(fer).
noEsCopado(dani).
noEstanCasados(fer, carlaConte).
etc.
```

- *Inversibilidad*: para probar que algo no se cumple tengo que conocer qué cosa no debe estar en la base (no puede ser incógnita lo que no esté, por eso necesito tener ligadas a mis variables cuando pregunto en negativo)

¿Recursividad? Un predicado que se relaciona consigo mismo. "Los amigos de mis amigos son mis amigos"

Listas

```
padre(ana, nico).
padre(ana, silvia).
padre(ana, mariano).
padre(fer, chiara).
padre(fer, melina).
```

- **Motivación**: me pregunto cuántos hijos tiene una persona. Como PROLOG me da las soluciones de a una no me sirve, entonces necesito un mecanismo para trabajar con "el conjunto de todas las soluciones" a la vez.
- Un par de ejemplos:

¿Cómo escribo "el conjunto de todos los hijos de ana"?

```
?- findall(X, padre(ana, X), Hijos).
X = _G358
Hijos = [nico, silvia, mariano]
```

Hijos = { x / x es hijo de Ana }

¿Y si quiero los hijos varones?

```
? findall(X, (padre(ana, X), varon(X)), HijosVarones).
HijosVarones = [nico, mariano]
```

HijosVarones = { x / x es hijo de Ana y x es varón }

También puedo poner una variable en lugar de ana:

```
? findall(X, padre(Padre, X), Hijos).
X = _G370
Padre = _G369
Hijos = [nico, silvia, mariano, chiara, melina]
```

- Ahora, ¿qué es eso entre corchetes?
Un conjunto o lista (a esta altura no nos interesa la diferencia).
- Puede tener dos formas:
 - Lista vacía []
 - Cabeza y cola [Cabeza|Cola]
 - Es una estructura recursiva, porque la cola es a su vez una lista.
 - Si pongo [1,2,3] es un shortcut.
 - ¿Cómo está formada la lista [1,2,3]? (la cabeza es 1 y la cola [2,3], etc). Tenemos las sogas para volver sobre el ejemplo, pero creo que ya se van a acordar.

Predicados recursivos sobre listas

Si digo que una lista es lista vacía o cabeza y cola, entonces definamos la longitud de cada una.

- ¿Cuál es la longitud de la lista []? 0
- ¿Y de [H|T]? 1 + la longitud de T.

Genéricamente:

La longitud de una lista vacía es 0.

La longitud de una lista que no es vacía es 1 + la longitud de su cola. ¡Nos quedó un predicado recursivo!

Trasladándolo a lógico:

```
length([], 0).
length([X|Xs], Longitud):-length(Xs, LongitudCola), Longitud is LongitudCola + 1.
```

- Como la lista es una estructura recursiva una forma de hacer cosas sobre una lista es a través de predicados recursivos.
- Muchas veces el caso base para una resolución recursiva es la lista vacía (no siempre).
- No es la única forma de trabajar sobre listas. Acabamos recién de conocer el predicado findall que genera una lista con las soluciones a una consulta. Veamos nuevamente la cláusula findall:

```
?- findall(X, padre(ana, X), Hijos).
```

¿Cómo accedo a cada elemento? ya no hay separación en cabeza y cola, conozco menos del algoritmo → tiene mayor grado de declaratividad.

Ciertos compiladores de Lógico exigen que cuando yo defina una variable en un predicado la tenga que usar, entonces cuando no uso X (porque no me interesa), defino una variable “anónima” con underscore:

```
longitud([],0).
longitud([_|Xs], Longitud):-longitud(Xs, LongitudCola),
                           Longitud is LongitudCola + 1.
```

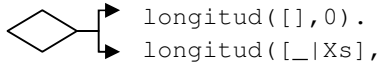
Por eso

? longitud([1, 2, 3], X)

Con un poco de suerte me devuelve 3.

Pattern Matching sobre Listas

- ¿Qué es lo que hace que siempre sepa cuál de las dos reglas usar en el length?



Las dos versiones de lista (vacía y cabeza | cola) responden a distintos patterns.

Lo que usa para elegir es pattern matching.

Para pensar: algunos ejercicios de matcheo

- Cuáles de los valores: 2, [], [2], [2,3], [4,5,6], [X], [2,X], [2|X]
- Matchean con: [H|T], [H], T, X, [2|T], [A, B]

Member: un elemento está en la lista

- si está en la cabeza de la lista
- si está en la cola.

El chiste de esta definición es que como la cola es una lista, la definición del predicado member termina siendo recursiva:

member(X, [X|_]).

member(X, [_|Z]):-member(X, Z).

¿Qué va a pasar si la lista está vacía? Evidentemente no puede haber ningún elemento en una lista vacía, por la definición de lista vacía. Como la lista vacía no es divisible en cabeza y cola, directamente no defino cláusulas y listo¹:

? member(5, [5, 8]) unifica con la primera cláusula (**Tip:** ¿qué sucede entonces, se siguen buscando otras soluciones o no?)

? member(5, [4, 6]) va a fallar, mientras que

? member(5, [4, 5]) va a encontrar al elemento 5 en la cola (por la segunda cláusula). Lo mismo pasará con:

? member(5, [4, 3, 5, 6]). El 5 está en la cola de esta lista.

Append: relaciona dos listas y su concatenación

? append([1, 3], [2], Resto)

Resto = [1, 3, 2]

El predicado queda definido append/3 (tiene aridad 3).

El caso base: Si la primera lista es vacía, append devuelve como lista resultante la segunda lista (esto incluye el caso particular en que la segunda lista también venga vacía).

El caso recursivo:

append([1, 3, 5], [2, 3], X) X = [1, 3, 5, 2, 3], o sea X = [1 | ?] ? = [3, 5, 2, 3]

Acá vemos que la cabeza de la primera lista (el elemento 1) forma parte de X, que es la tercera lista. Y no sólo forma parte... ¡es la cabeza de la tercera lista!

¹ ¿Con qué concepto está relacionado esta idea? ¡Exacto! Principio de Universo Cerrado, lo que no se cumple no me interesa que quede en la base de conocimientos.

Por otra parte, la cola de la tercera lista es [3, 5, 2, 3]

Eso no es otra cosa que la concatenación de la cola de la primera lista [3, 5] con la segunda lista [2, 3].

La concatenación es el predicado `append` que estamos resolviendo, de hecho es el mismo caso recursivo anterior, sólo que:

Hacemos `append` de [3, 5] con [2, 3], la tercera lista es: [3 | ¿?]. Nuevamente vemos que la cabeza de la primera lista es también la cabeza de la tercera lista. Eso sí, la cola de la tercera lista no es la cola de la primera lista: es la cola de la primera concatenada con la segunda lista...

Genéricamente:

```
append([], L2, L2).
```

```
append([X|L1], L2, [X|L3]) :- append(L1, L2, L3).
```

Esto se lee: concateno dos listas de la siguiente manera:

- si la primera lista es vacía, la tercera lista es igual a la segunda lista.
- si la primera lista tiene elementos, esos elementos forman parte de la tercera lista.

Ejercicios con predicados sobre listas

- Sumatoria: es similar al `length`.

“La sumatoria de una lista es:

0 si la lista es vacía o

la cabeza más la sumatoria de la cola”

- Mayor

“El mayor de una lista de un elemento es ese elemento o

El mayor de una lista es el mayor entre la cabeza y el mayor de la cola.”

- Seguimos con el promedio

Y como broche, mechamos con inversibilidad.

`Length`, ¿es inversible? Sí, porque no necesita unificar variables (aunque sin la lista unificada no tiene mucho sentido).

`Sumatoria`, ¿es inversible? No, porque no puede generar listas con expresiones aritméticas sin unificar.

`Append` es full inversible (aunque de nuevo no tiene mucho sentido si no están unificados "C" o "A y B"), yo puedo hacer:

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

```
A = []
```

```
B = [1, 2, 3] ;
```

```
A = [1]
```

```
B = [2, 3] ;
```

```
A = [1, 2]
```

```
B = [3] ;
```

```
A = [1, 2, 3]
B = [] ;
```

O bien

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

```
Y = [2, 3] ;
```

O bien ... etc ...

Acá se puede ver que una relación es mucho más potente que una función: no tuve que pensar de antemano qué tipo de consultas satisfacer, el motor se encarga de responder varias preguntas a la vez.

Desafío

Definir un predicado `sum2Elementos` que relaciona un número `N` con dos números `A` y `B`, tal que $A + B = N$

```
sum2Elementos(X, 5)
```

```
X = [1, 4] ;
X = [2, 3] ;
X = [3, 2] ;
X = [4, 1]
... etc...
```

```
numero(1).
numero(2).
numero(3).
numero(4).
numero(5).
numero(6).
numero(7).
numero(8).
numero(9).
```

```
sum2Elementos([A, B], N):- numero(A), numero(B), N is A + B.
```