

En el paradigma lógico se trabaja con el **principio de Universo Cerrado**, esto quiere decir que se asume como falso todo aquello que no puedo probar como verdadero, por lo tanto, está por fuera de la base de conocimiento y no podemos demostrar su veracidad. La forma de decir que algo es falso es no decir nada.

Cada programa de Prolog es una **base de conocimiento**: En ella se escribe todo lo que se sabe (SIEMPRE es VERDADERO) en forma de predicados. Se compone de **cláusulas** que definen **predicados** partiendo de los individuos de los que queremos hablar.

Las **cláusulas** son sentencias/unidades de información de una base de conocimiento. Deben terminar con un punto (.) (comienza con el nombre). Un predicado puede tener o no, más de una Cláusula. Cada cláusula puede ser un hecho o una regla.

Un **individuo** es cualquier entidad sobre la cual nos interese estudiar sus características o relaciones con otros individuos.

→ Individuos Simples:

- >Átomos, son las entidades que comienzan con minúscula y son indivisibles, piezas de información del dominio;
- >Números.

→ Individuos Compuestos:

- >Listas;
- >Funtores, son individuos que nos permiten agrupar otros individuos para formar una abstracción más compleja, tienen un nombre y una aridad determinada.

Los predicados pueden ser propiedades o relaciones, las **propiedades** son de aridad 1 y expresan características de los individuos, mientras que las **relaciones** tienen una aridad mayor a 1 y expresan una relación entre individuos. A su vez los predicados tienen formas distintas, ya que pueden ser **hechos** o **reglas**, los predicados pueden tener múltiples sentencias que pueden ser hechos o reglas.

Un **hecho** hace una afirmación incondicional (no depende de ninguna condición para ser cierta), generalmente sobre un individuo particular. (No contiene el símbolo ":-") Siempre son verdaderos. Los hechos me permiten definir por extensión el conjunto de individuos que tienen una característica.

Una **regla** posee un cuerpo, define que si se cumplen ciertas condiciones, entonces un predicado se verifica para ciertos individuos. Su valor de verdad depende de otros predicados. Van con ":-".

```

Hecho
humano(sócrates).
} Predicados
regla
mortal(Alguien):- humano(Alguien).

```

IDAÑEZ, LUCIA MARÍA 🐱

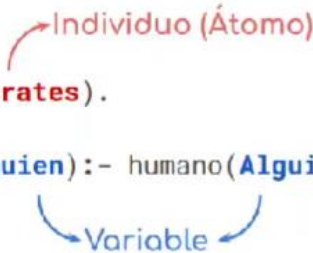
Esta manera de estructurar la información me va a dejar escribir cualquier dominio que se me ocurra, solo establezco conexiones entre los datos que sé que voy a tener.

Se trabaja con dos formas de parámetros o elementos:

->Átomos.

->Variables, determina si algo es un valor (se escriben con mayúscula), son las “incógnitas de las ecuaciones”.

Las variables se pueden ligar, estas toman un valor y se unifican, una vez que estas se unifican, toman el mismo valor para todas sus instancias.


humano(**sócrates**).
mortal(**Alguien**):- humano(**Alguien**).

Para colocar múltiples restricciones en una regla debemos colocar una “,” entre las restricciones que indica que se deben cumplir simultáneamente.

Consultas, son aquellas que realizamos para obtener información sobre nuestra base de conocimiento.

Las consultas pueden ser individuales, existenciales o variables, dependiendo de cada consulta recibiremos una o varias respuestas.

Las consultas INDIVIDUALES son aquellas que se realizan sobre un individuo en particular.

Ejem.:

```
?- pokemon(squirtle, agua).  
true.
```

Las consultas VARIABLES son aquellas que nos permiten trabajar con las incógnitas, nos devuelven los valores que hacen verdadero al predicado.

Ejem.:

```
?- pokemon(squirtle, Tipo).  
Tipo = agua.
```

“Tipo” es la variable, al realizar la consulta nos devuelve el valor que hace verdadero al predicado

El **Tipo** que hace verdadero al predicado **pokemon squirtle** es **agua**.

IDAÑEZ, LUCIA MARÍA 🐾

Las consultas EXISTENCIALES son aquellas que nos indican si existe ALGÚN INDIVIDUO el cual haga verdadero al predicado. La/s variable/s por la cual/es pregunto van con guion bajo. Devuelve True/False.

Ejem.:

```
?- pokemon(_, electrico).  
true.
```

Se coloca el _ para la consulta, esto se traduce como:
Existe algún pokemon que sea eléctrico

Ejem. De una regla:

Queremos definir los pokemon que ganan en una batalla, para ello el tipo de un pokemon debe ser super efectivo contra el tipo de otro pokemon.

En base a esto definimos una serie de cláusulas:

```
% superEfectivo(TipoQueGana, TipoQuePierde).  
superEfectivo(agua, fuego).  
superEfectivo(fuego, hierba).  
superEfectivo(hierba, agua).
```

Nos determina qué tipo es super efectivo contra otro tipo.

Ahora definimos la REGLA ganaEnBatalla la cual nos va a permitir realizar consultas.

Un pokemon gana en batalla cuando su tipo es super efectivo contra el del otro:

ganaEnBatalla (PokemonQueGana, PokemonQuePierde) :-

Declaramos las variables de la regla:
PokemonQueGana, PokemonQuePierde

pokemon (PokemonQueGana, TipoQueGana),

pokemon (PokemonQuePierde, TipoQuePierde),

superEfectivo (TipoQueGana, TipoQuePierde).

Como necesitamos los tipos, preguntamos si para el **PokemonQueGana** su tipo es el **TipoQueGana**. Repetimos esto con el **PokemonQuePierde**.

Preguntamos si el **TipoQueGana** es super efectivo contra el **TipoQuePierde**.

Concatenamos todas las condiciones mediante una " , ".

→ Consultas:

Individual

```
?- ganaEnBatalla(charmander, squirtle).  
false.
```

Variable

```
?- ganaEnBatalla(PokemonGanador, PokemonPerdedor).  
PokemonGanador = bulbasaur,  
PokemonPerdedor = squirtle ;  
PokemonGanador = squirtle,  
PokemonPerdedor = charmander ;  
PokemonGanador = charmander,  
PokemonPerdedor = bulbasaur ;  
PokemonGanador = charmander,  
PokemonPerdedor = ivysaur ;  
PokemonGanador = charmander,  
PokemonPerdedor = venusaur ;  
PokemonGanador = ivysaur,  
PokemonPerdedor = squirtle ;  
PokemonGanador = venusaur,  
PokemonPerdedor = squirtle ;  
false.
```

Todas las formas en las que ganaEnBatalla es verdadero.

IDAÑEZ, LUCIA MARÍA 🐱

Para esto Prolog utiliza algo que llamamos **backtracking**, este es el proceso para encontrar múltiples valores por fuerza bruta, es decir va a buscar todas las combinaciones posibles de elementos que satisfagan la respuesta, va a deshacer y volver para atrás.

Existencial

```
?- ganaEnBatalla(charmander, _).  
true ;  
true ;  
true ;  
true.
```

Para agregar otra cláusula a la regla, se realiza de la siguiente manera:
Un pokemon legendario gana todas las batallas.

```
ganaEnBatalla(PokemonQueGana, PokemonQuePierde) :-  
    pokemon(PokemonQueGana, TipoQueGana),  
    pokemon(PokemonQuePierde, TipoQuePierde),  
    superEfectivo(TipoQueGana, TipoQuePierde).  
  
ganaEnBatalla(PokemonLegendario, PokemonQuePierde) :-  
    legendario(PokemonLegendario).
```

Reescribimos la cabeza de la regla,
y debajo su condición.

En este caso PokemonQuePierde no está siendo utilizada, se puede reemplazar con un `_`, sin embargo al momento de hacer la consulta variable, Prolog no es capaz de dar una respuesta ya que no puede deducir los infinitos valores que cumplen esa condición debido a que no tiene una restricción.

```
?- ganaEnBatalla(mew, Ganador).  
true.
```

mew es un pokemon legendario y al preguntar a qué pokemon le gana mew, Prolog responde **True** como si fuera una consulta Individual.

Para estos casos debemos restringir los valores que pueden hacer que esa condición se cumpla.

```
ganaEnBatalla(PokemonLegendario, PokemonQuePierde) :-  
    legendario(PokemonLegendario),  
    pokemon(PokemonQuePierde, _).
```

Se restringe a que
PokemonQuePierde sea
un pokemon.

Se consulta:

```
?- ganaEnBatalla(mew, Perdedor).  
Perdedor = pikachu ;  
Perdedor = bulbasaur ;  
Perdedor = squirtle ;  
Perdedor = charmander ;  
Perdedor = ivysaur ;  
Perdedor = ivysaur ;  
Perdedor = venusaur.
```

Los predicados también pueden ser del tipo:

Nivel < 20.

Costo > 100.

Siempre y cuando la variable ya esté ligada al realizar la comparación. En caso de que no esté ligada, Prolog evaluará infinitos números menores o mayores al número a comparar y nos dará un ERROR.

Se debe tener en cuenta el orden en que se colocan los predicados.

Clase 29/06

Unificación vs. Asignación

La asignación sucede cuando es posible reemplazar el valor de una variable por otro (no hay en lógico). Las variables en el paradigma lógico se asemejan a la idea de variable matemática, y el mecanismo por el cual se le dan valores a las variables se llama unificación. Cuando una variable que no tiene ningún valor pasa a tenerlo vamos a decir que dicha variable ha sido ligada, en caso contrario la variable se encuentra sin ligar o no ligada. Una variable siempre que tome un valor, va a tener siempre el mismo valor dentro del predicado/ la consulta, ya que se unifica. Pero dos variables diferentes también pueden unificarse con un mismo valor, por lo que es necesario aclarar si queremos que dos variables sean diferentes dentro de nuestro predicado.

En Prolog no se utiliza el “=” sino que se utiliza el **is** el cual sirve para las cuentas aritméticas.

Ej.:

Tenemos la siguiente fuente de conocimiento:

```
% puntos(Pokemon, Ataque, Defensa).
puntos(bulbasaur, 5, 5).
puntos(squirtle, 6, 4).
puntos(charmander, 3, 7).
puntos(pikachu, 50, 30).
puntos(charizard, 60, 35).
```

Queremos calcular el poder de cada Pokemon, el cual es la suma del ataque y defensa, excepto mewtwo que el poder es de 1000.

```
poder(Pokemon, Poder) :-
    puntos(Pokemon, Ataque, Defensa),
    Poder is Ataque + Defensa.
```

Para mewtwo podemos definir directamente un predicado que especifique que su poder es igual a 1000, esto quiere decir que estaríamos ligando las variables.

```
% Asignación \= unificación (unificar variables, ligar variables).
poder(mewtwo, 1000).
```

Mediante esto estamos definiendo que utilizamos **IS** únicamente para cuentas aritméticas, ya que este es el predicado que “sabe” hacer cuentas es decir sumas, restas multiplicación y división, de otra manera estaríamos indicando una asignación, es decir dar valor algo, mientras que en lógico nosotros ligamos y unificamos variables. En lógico, no existe el concepto de asignación sino el de unificación, una vez que una variable unifica/ se liga a un valor, este no puede cambiar.

Ej.:

Modificamos la regla anterior, estableciendo que para cualquier pokemon legendario su poder es 1000.

```
poder(Legendario, 1000) :-
    legendario(Legendario).
```

En este caso, no ligamos ningún pokemon al antecedente sino que dejamos la variable “Legendario” sin ligar, por lo tanto al momento de hacer la consulta la variable se ligará y en caso de que cumpla con el consecuente (todo aquello que está después del “:-”), dará como resultado 1000.

Otro ej.:

```
% diferenciaEntreEvoluciones/2
% diferenciaEntreEvoluciones(Pokemon, Diferencia).
% Relaciona a un Pokemon y cuánto pasa entre que evoluciona en una cosa y otra.

diferenciaEntreEvoluciones(Pokemon, Diferencia) :-
    evolucion(_, Nivel, Pokemon),
    evolucion(Pokemon, OtroNivel, _),
    Diferencia is OtroNivel - Nivel.
```

INVERSIBILIDAD

Si puedo hacer una consulta existencial en una de las variables del predicado, entonces esa variable es inversible.

Existen tres tipos de inversibilidad para un predicado:

- Predicado totalmente inversible,
- Predicado parcialmente inversible,
- Predicado no inversible.

En un predicado totalmente inversible si puedo hacer consultas variables para todos sus parámetros.

Un predicado parcialmente inversible solo se puede hacer consultas variables para algunos de sus parámetros, es decir debe tener al menos una variable ligada.

Un predicado es no inversible si no puedo hacer consultas variables para ninguno de sus parámetros, este tipo de predicado solo soporta preguntas del tipo individual (solo nos indica True o False).

Ej.:

Predicado totalmente inversible.

```
?- poder(Pokemon, Poder).
Pokemon = bulbasaur,
Poder = 10 ;
Pokemon = squirtle,
Poder = 10 ;
Pokemon = charmander,
Poder = 10 ;
Pokemon = pikachu,
Poder = 80 ;
Pokemon = charizard,
Poder = 95 ;
Pokemon = mew,
Poder = 1000 ;
Pokemon = mewtwo,
Poder = 1000.
```

poder es un predicado totalmente inversible, ya que el predicado es capaz de darnos todos los valores posibles que hacen lo verdadero.

Existen ciertas cuestiones que traen posibles problemas de inversibilidad, entre estas cuestiones están la aritmética, las comparaciones, los **not**, la recursividad, los forall, los funtores y polimorfismos.

En el caso de la **aritmética**, todas las variables a la derecha del **is** deben llegar ligadas al **is**, es decir un predicado no es inversible en caso de que al llegar a la cuenta a la derecha del **is** haya una variable sin ligar.

Not

Es un predicado de aridad 1 de orden superior. Las variables que aparecen en la parte de la cláusula deben llegar ligadas, a menos que sean variables afectadas por un “para ningún”, en este caso deben llegar libres.

Los **not** nos propician casos de no inversibilidad, ya que al no llegar ligados al momento de evaluar, entonces hay infinitos valores para los cuales ese predicado se cumpla, sin importar si existen casos dentro de nuestro universo para los cuales se cumple.

Ej.:

Tenemos el caso de `evolucionarRápido`, el cual nos indica que un pokemon evoluciona rápido si su nivel de evolución es menor a 20. Por lo tanto, queremos saber aquellos pokemon que no evolucionan rápido.

```
?- evolucionarRápido(charizard).
false.

?- evolucionarRápido(felipe).
false.

?- not(evolucionarRápido(charizard)).
true.

?- not(evolucionarRápido(bulbasaur)).
false.

?- not(evolucionarRápido(felipe)).
true.

?- not(evolucionarRápido(hola)).
true.

?- not(evolucionarRápido(banana)).
true.
```

En este caso tenemos aquel caso que no evoluciona rápido, **charizard**, por lo tanto es un valor que cumple con **not(evolucionarRápido charizard)**, sin embargo al preguntar por aquellos pokemon que no evolucionan rápido, estos son infinitos por lo tanto Prolog no es capaz de determinarnos todos aquellos valores para los cuales el predicado es válido y nos da “False”.


```
?- not(evolucionarRápido(Pokemon)).
false.
```

El orden de los predicados es importante en este caso, ya que la variable que está en la cláusula dentro del not debe llegar ligada, de esta manera tendremos un conjunto de valores finito (un valor unificado) sobre el cual fijarme si se cumple o no se cumple la regla, cuando el conjunto es reducido es más sencillo comprobar si se cumple o no la regla, si el conjunto es infinito no será posible para Prolog determinar la validez del predicado.

EJ.:

```
% entrenadorVencible/1
% Un entrenador es vencible cuando:
% - Tiene un pokemon de tipo insecto; o
% - Tiene un pokemon con poder menor a 50; o
% - Tiene un pokemon que no evoluciona rápido

equipo(esteban, charmander). % esteban es invencible
equipo(felipe, caterpie). % felipe es vencible porque tiene un insecto.
equipo(fede, squirtle). % fede es vencible porque tiene uno con poder menor a 50.
equipo(tomas, wartortle). % tomás es vencible porque tiene uno que *no* evoluciona rápido.

entrenadorVencible(Entrenador) :-
    equipo(Entrenador, Pokemon),
    pokemonVencible(Pokemon).

entrenadorVencible(Entrenador) :-
    equipo(Entrenador, Pokemon),
    poder(Pokemon, Poder),
    Poder < 50.

entrenadorVencible(Entrenador) :-
    equipo(Entrenador, Pokemon),
    not(evolucionarRápido(Pokemon)).
```

En este caso estamos repitiendo lógica.

Delegamos la parte de que un entrenador tiene un pokemon, esta es la parte que queremos que pase todas las veces. Aquello que varía es la debilidad de cada pokemon que posee el entrenador, ya que un pokemon es vencible si cumple alguna de las condiciones establecidas.

```
entrenadorVencible(Entrenador) :-
    equipo(Entrenador, Pokemon),
    pokemonVencible(Pokemon).

pokemonVencible(Pokemon) :-
    pokemon(Pokemon, insecto).

pokemonVencible(Pokemon) :-
    poder(Pokemon, Poder),
    Poder < 50.

pokemonVencible(Pokemon) :-
    not(evolucionarRápido(Pokemon)).
```

pokemonVencible no es inversible debido a que en su tercera cláusula tiene **not(evolucionarRápido(Pokemon))** el cual trae problemas de inversibilidad ya que al hacer una consulta **Pokemon** llega sin ligar. Sin embargo, entrenadorVencible es inversible, a pesar de utilizar pokemonVencible, ya que utiliza el predicado con la variable Pokemon ligada, la cual procede de equipo(Entrenador, Pokemon).

Clase 10/07

Hasta el momento vimos todos los predicados en términos de existencia, esto quiere decir que no aplicamos lo que se llama “para todo...” sino “existe al menos uno...”.

Para realizar un “para todo” en cuestiones de existencia, debemos aplicar la doble negación de los predicados.

Ej.:

Realizamos el siguiente predicado:

Para todo pokemon de un entrenador, ese pokemon es de fuego.

Lo realizamos en términos matemáticos:

$$\forall \text{ pokemon} : (\text{pokemon}, \text{entrenador}) \rightarrow (\text{pokemon}, \text{fuego})$$

$$\forall x : p(x) \rightarrow q(x)$$

Sin embargo, esto es lo mismo que decir:

```
% ~(\forall x : p(x) => q(x))
% ~(\forall x : ~p(x) \vee q(x))
% ~(\exists x : p(x) ^ ~q(x))
```

Esto significa: No existe un pokemon de un entrenador tal que ese pokemon NO sea de fuego.

Orden Superior: Un predicado es de orden superior si recibe una consulta (otro predicado) como parámetro. Representamos esto en Prolog:

```
tieneTodosPokemonDeFuego(Entrenador) :-
    not( (equipo(Entrenador, Pokemon), not(pokemon(Pokemon, fuego))) ).
```

Consulta Individual:

```
?- tieneTodosPokemonDeFuego(esteban).
true.

?- tieneTodosPokemonDeFuego(felipe).
false.
```

Sin embargo, al preguntar esto de forma variable, da como resultado “False”, a pesar de que hay individuos en nuestro universo que hacen verdadero al predicado, esto nos indica que el predicado no es inversible.

En un predicado de orden superior, la variable por la que queremos consultar debe llegar ligada. En el caso visto, la variable que debe llegar ligada es el entrenador, ya que el **para todo** es para el Pokemon, esto quiere decir que es PARA TODO pokemon DE UN ENTRENADOR (indica que el Entrenador debe estar ligado). Esto se soluciona limitando los valores de los entrenadores.

```
tieneTodosPokemonDeFuego(Entrenador) :-
    equipo(Entrenador, _),
    not( (equipo(Entrenador, Pokemon), not(pokemon(Pokemon, fuego))) ).
```

Limito los valores para la variable “Entrenador”

De esta manera el predicado va a ser inversible.

FORALL:

El **forall** es un predicado de un orden superior de aridad 2, donde el primer parámetro es el antecedente y el segundo parámetro es el consecuente. No es inversible para ninguna de sus dos aridades.

forall(UNIVERSO, CONDICIÓN)

El UNIVERSO es la consulta existencial que me genera los valores sobre los cuales trabajar, y la condición es la consulta me genera los valores de verdad para saber si se cumple o no esa condición para cierto valor (generado anteriormente).

Ej.:

Realizamos un predicado donde queremos que para todos los pokemon de un entrenador, ese pokemon es de fuego.

```
tieneTodosPokemonDeFuego2(Entrenador) :-
    equipo(Entrenador, _),
    forall( equipo(Entrenador, Pokemon), pokemon(Pokemon, fuego) ).
```

Limito los valores para la variable “Entrenador”

El **antecedente** del predicado es que es el pokemon de un entrenador

El **consecuente** del predicado es que ese pokemon debe ser de fuego

IDAÑEZ, LUCIA MARÍA 🐱

```
tieneTodosPokemonDeFuego2(Entrenador) :-  
    equipo(Entrenador, _),  
    forall( equipo(Entrenador, Pokemon), pokemon(Pokemon, fuego) ).  
%  
%           ^           ^  
%   variable ligada   variable libre   I  
%   (actúa como UN)   (actúa como TODOS)
```

Vamos a LIGAR aquella variable que actúe como UN en el predicado.

Ligar una variable hace que actúe como UN y no ligar una variable hace que actúe como TODOS.

```
% Saber si todos los pokemones de cierto tipo son vencibles.  
% pokemonVencible(Pokemon).  
  
todosLosPokemonSonVencibles(Tipo) :-  
    tipo(Tipo),  
    forall(pokemon(Pokemon, Tipo), pokemonVencible(Pokemon)).  
%%  
%%           ^           ^  
%%   variable libre   variable ligada  
%%   (actúa como TODOS) (actúa como UNO)  
  
tipo(Tipo) :-  
    pokemon(_, Tipo).
```

Un Tipo es Tipo si existe un pokemon que sea de ese Tipo.

```
%% Saber si un entrenador es malo, lo cual ocurre si no tiene pokemones legendarios.  
%% Para todos los pokemones del entrenador, ninguno es legendario.  
  
entrenadorMalo(Entrenador) :-  
    entrenador(Entrenador),  
    forall( equipo(Entrenador, Pokemon), not(legendario(Pokemon)) ).  
%%  
%%           ^           ^  
%%   variable ligada   variable libre  
%%   (actúa como UNO)   (actúa como todos)  
  
entrenador(Entrenador) :-  
    equipo(Entrenador, _).
```

FUNCTORES

Los **functores** son **individuos** compuestos que tienen nombre, parámetros y la forma de un predicado pero **NO LO SON**, y se usan para agrupar en un solo individuo cosas que tengan distinta forma pero que sigan representando el mismo individuo.

Los ponemos en el predicado para que representen la información que queremos.

Los functores son individuos, parecen predicados, pero no lo son.

Los functores siempre van a estar como parámetros.

Ej.:

```
% Haremos entonces un listado de ataques.  
% * Los especiales dicen su tipo y cuánto daño hacen.  
% * Los rápidos solamente dicen cuánto daño hacen.  
% * Los potentes no dicen nada, simplemente son potentes.  
  
% impactrueno, especial electrico que hace 20 de daño  
% voltio, especial electrico que hace 10 de daño  
% cabezazo, rapido y hace 50 de daño  
% coletazo, potente  
  
% ataque(Ataque, TipoDeAtaque).  
  
% Functor es un individuo compuesto que tiene la forma de un predicado PERO QUE NO LO ES  
% * especial(TipoPokemon, Daño)  
% * rapido(Daño)  
% * potente()  
  
ataque(impactrueno, especial(electrico, 20)).  
ataque(voltio, especial(electrico, 10)).  
ataque(cabezazo, rapido(50)).  
ataque(coletazo, potente()).
```

Functores

¿Cómo se usan los funtores?

Ej.: Queremos saber el daño que hace un pokemon cuando ataca con cierto ataque.

Para saber su daño cuando un pokemon ataca con un ataque, se sigue la siguiente lógica:

- * Si es especial y coincide en tipo, hace 50 de daño extra
- * Si es especial y no coincide en ningún tipo, hace 25 de daño extra
- * Si es rápido, sólo hace el daño del ataque
- * Si es potente, hace 100 de daño.

```
% * Si es especial y coincide en tipo, hace 50 de daño extra
atacar(Pokemon, NombreAtaque, Danio) :-
    ataque(NombreAtaque, especial(Tipo, DanioAtaque)),
    pokemon(Pokemon, Tipo),
    Danio is DanioAtaque + 50.

% * Si es especial y no coincide en ningún tipo, hace 25 de daño extra
atacar(Pokemon, NombreAtaque, Danio) :-
    ataque(NombreAtaque, especial(TipoAtaque, DanioAtaque)),
    pokemon(Pokemon, _),
    forall( pokemon(Pokemon, TipoPokemon), TipoPokemon \= TipoAtaque ),
    Danio is DanioAtaque + 25.

% * Si es rápido, sólo hace el daño del ataque
atacar(Pokemon, NombreAtaque, Danio) :-
    pokemon(Pokemon, _),
    ataque(NombreAtaque, rapido(Danio)).

% * Si es potente, hace 100 de daño.
atacar(Pokemon, NombreAtaque, 100) :-
    pokemon(Pokemon, _),
    ataque(NombreAtaque, potente()).
```

Ligamos las variables de Pokemon para que sea inversible.

POLIMORFISMO

El polimorfismo permite obtener soluciones más genéricas, que sean válidas para diferentes tipos de datos contemplando las particularidades de cada uno de ellos.

Ej.:

En el ejemplo anterior, existen líneas que se repiten múltiples veces, así como:

```
pokemon(Pokemon, _), ataque(NombreAtaque, especial(Tipo, DanioAtaque))
y la porción de línea ataque(NombreAtaque,...).
```

Por lo tanto, vamos a generar abstracciones para aquellas cosas que se repiten.

```
atacar(Pokemon, NombreAtaque, Danio) :-
    ataque(NombreAtaque, ClaseDeAtaque,
    esPokemon(Pokemon),
    danioCausadoPor(Pokemon, ClaseDeAtaque, Danio).
```

Como el segundo parámetro de **ataque** es un functor, podemos reemplazarlo por el nombre de una variable ya que estos son individuos

```
% pokemon/1 \= pokemon/2
esPokemon(Pokemon) :-
    pokemon(Pokemon, _).
```

El predicado **danioCausadoPor** es una abstracción de aquellas líneas que se repetían, por ende nos otorga el **Danio** que causa cada **Pokemon** al atacar con una **ClaseDeAtaque**.

Ahora en **danioCausadoPor** realizamos los distintos casos para las clases de ataque:

Clase 13/07/2022**LISTAS**

Las **listas** tienen el mismo formato que en Haskell, es decir que se representan entre corchetes [...]. Las listas en Prolog no tienen tipos, esto quiere decir que podemos “mezclar” números con strings.

Findall es un predicado de orden superior que tiene aridad 3, este predicado encuentra todos los que cumplen con una condición. Tiene la siguiente estructura:

findall(Elemento, Condición, Lista).

- ➔ Elemento: nos permite elegir la porción de respuesta que voy a querer generar, de qué voy a querer que esté conformada la lista.
- ➔ Condición: nos da el universo con el que vamos a querer trabajar, la condición que deben cumplir los elementos que van a conformar la lista.
- ➔ Lista: agrupa los elementos.

Predicados con listas:

Length es de aridad 2, se cumple para la longitud de la lista. Es inversible solo para el segundo parámetro. Tiene la siguiente forma: `length([...], Longitud).`

Member se cumple si el elemento está en la lista. Tiene la siguiente forma: `member(Elemento, [...]).`

Sumlist se cumple para la suma de los elementos de la lista. Los elementos de la lista deben ser numéricos y es inversible solo para el segundo parámetro. Tiene la siguiente forma: `sumlist([...], ResultadoDeLaSuma).`

El findall nos limita en cuanto a orden.

Ej.: Queremos saber cuántos pokemones son de un tipo.

Este predicado se realiza mediante `findall`, ya que tenemos un universo sobre el cuál trabajar (pokemones) y queremos saber cuáles de ellos cumplen una condición (ser de un tipo). Luego, se agrupan en una lista de la cual sacamos su longitud.

```
% cuantosPokemonesSonDeUnTipo/2 -> Tipo, Cantidad.
cuantosPokemonesSonDeUnTipo(Tipo, Cantidad) :-
    tipo(Tipo),
    findall(Pokemon, pokemon(Pokemon, Tipo), Pokemones),
    length(Pokemones, Cantidad).|
```

Ligo la variable “Tipo”

Pokemon llega como variable libre al `findall` mientras que Tipo, llega ligada, esto quiere decir que encuentra **todos los pokemones de un tipo**.

SIEMPRE SE DEBE GENERAR LAS VARIABLES CON FINALL Y FORALL, ASÍ LLEGAN LIGADAS AL PREDICADO.

Ej.:

```
hayAlMenosTresDeUnTipo(Tipo) :-
    cuantosPokemonesSonDeUnTipo(Tipo, Cantidad),
    Cantidad >= 3.

hayMasDeCincoDeUnTipo(Tipo) :-
    cuantosPokemonesSonDeUnTipo(Tipo, Cantidad),
    Cantidad > 5.

hayExactamenteSiete(Tipo) :-
    cuantosPokemonesSonDeUnTipo(Tipo, 7).
```

```
% Para fijarme si...
% ninguno cumple:          not/1
% alguno cumple:          existencia (todo lo que vimos las primeras tres clases!)
% todos cumplen:          forall/2
% al menos dos cumplen:    existencia
% al menos N cumplen (N > 2): findall + length
% exatamente N cumplen (N > 2): findall + length
```

ilógico

Punto 1

```
todosSiguenA(Rey) :-
    personaje(Rey),
    not((personaje(Personaje), not(sigueA(Personaje, Rey))))).
```

```
sigueA(Alguien, Alguien).
sigueA(lyanna, jon).
sigueA(jorah, daenerys).
%% etc
```

En este caso estamos diciendo, mediante la línea **not((personaje(Personaje), not(sigueA(Personaje, Rey))))**, que **no existe un personaje tal que no siga al rey**, esto es lo mismo que decir que TODOS LOS PERSONAJES SIGUEN AL REY. Por lo tanto lo reescribimos como:

```
forall(personaje(Personaje), sigueA(Personaje, Rey)).
```

Punto 2 - Error de desaprobación CASI directa

```
baresCopados(Ciudad, Bares) :-
    findall(Bar, (puntoDeInteres(bar(CantVarCer), Ciudad), CantVarCer > 4), Bares).
```

```
museosCopados(Ciudad, Museos) :-
    findall(Museo, puntoDeInteres(museo(cienciasNaturales), Ciudad), Museos).
```

```
estadiosCopados(Ciudad, Estadios) :-
    findall(Estadio, (puntoDeInteres(estadio(Cap), Ciudad), Cap > 4000), Estadios).
```

```
ciudadInteresante(Ciudad) :-
    antigua(Ciudad),
    baresCopados(Ciudad, Bares),
    museosCopados(Ciudad, Museos),
    estadiosCopados(Ciudad, Estadios),
    length(Bares, CantidadBares),
    length(Museos, CantidadMuseos),
    length(Estadios, CantidadEstadiosCopados),
    CantidadLugaresCopados is CantidadBares + CantidadMuseos + CantidadEstadios,
    CantidadLugaresCopados > 10.
```

En este caso tenemos repetición de lógica en lo que son los **findall**, ya que realizamos la misma evaluación para tres casos distintos, y además tenemos error en la expresividad, debido a nombres de variables tales como **Cap**, **CantVarCer**, etc.

Primero, para solucionar el error de repetición de lógica, agrupamos los predicados de **baresCopados**, **museosCopados** y **estadiosCopados** en un solo predicado llamada **"puntosDeInteresesCopadosEn"** que reciba la **Ciudad** y **PuntosDeInteres** de esta manera evitamos la repetición de lógica.

```
puntosDeInteresesCopadosEn(Ciudad, PuntosDeInteres) :-
    findall(PuntoDeInteres, puntoDeInteresCopadoEn(PuntoDeInteres, Ciudad),
    PuntosDeInteres).
```

```
puntoDeInteresCopadoEn(PuntoDeInteres, Ciudad) :-
    puntoDeInteres(PuntoDeInteres, Ciudad),
    puntoDeInteresCopado(PuntoDeInteres).
```

Mediante el predicado **"puntoDeInteresCopadoEn"** que recibe la ciudad y el PuntoDeInteres, se verifica que ese punto esté en la ciudad con el predicado **puntoDeInteres** donde el

Mediante el predicado **"puntosDeInteresesCopadosEn"** lo que hacemos es realizar una lista de **PuntosDeInteres** para luego sacar su longitud, luego tenemos el predicado **puntoDeInteresCopadoEn** que recibe la ciudad y el PuntoDeInteres (la variable ligada es la ciudad).

```
puntoDeInteresCopado(bar(VariedadesDeCerveza)) :-
    VariedadesDeCerveza > 4.
```

```
puntoDeInteresCopado(museo(cienciasNaturales)).
```

```
puntoDeInteresCopado(estadio(Capacidad)) :-
    Capacidad > 4000.
```

```
puntoDeInteresCopado(parque(CantidadDeArboles)) :-
    CantidadDeArboles > 5.
```

Condiciones que debe cumplir
para que un lugar sea copado.

Finalmente, definimos el predicado **ciudadInteresante**:

```
ciudadInteresante(Ciudad) :-
    antigua(Ciudad),
    puntosDeInteresCopadosEn(Ciudad, PuntosDeInteres),
    length(PuntosDeInteres, Cantidad),
    Cantidad > 10.
```

Punto 3 - Error de desaprobación directa

```
inFraganti(Delito, Delincuente) :-
    cometio(Delito, Delincuente),
    findall(Testigo, testigo(Delito, Testigo), Testigos),
    length(Testigos, Cantidad),
    Cantidad > 0.
```

Este error es de desaprobación directa, ya que es un error a nivel lógico, debido a que estamos utilizando:

```
findall(Testigo, testigo(Delito, Testigo), Testigos),
length(Testigos, Cantidad),
Cantidad > 0.
```

Solo para saber si hay algún testigo que presencié un delito, esto es una consulta EXISTENCIAL por lo tanto no hace falta realizar un **findall**.

```
inFraganti(Delito, Delincuente) :-
    cometio(Delito, Delincuente),
    testigo(Delito, _).
```

Punto 8 - Error de repetición de lógica - Puede llegar a ser de desaprobación

```
costoEnvio(Paquete, PrecioTotal) :-
    findall(PrecioItem, precioItemPaquete(Paquete, PrecioItem), Precios),
    sumlist(Precios, PrecioTotal).
```

```
precioItemPaquete(Paquete, Precio) :-
    itemPaquete(Paquete, libro(Precio)).
```

```
precioItemPaquete(Paquete, Precio) :-
    itemPaquete(Paquete, mp3(_, Duracion)),
    Precio is Duracion * 0.42.
```

```
precioItemPaquete(Paquete, PrecioOferta) :-
    itemPaquete(Paquete, productoEnOferta(_, PrecioOferta)).
```


IDAÑEZ, LUCIA MARÍA 🐱

Este es un error de repetición de lógica, se soluciona agrupando aquellos predicados en común.

En este caso **precioItemPaquete(Paquete, Precio)** se repite tres veces, por tanto vamos a extraer esa lógica y observar que solo se diferencian en la asignación del precio, por lo tanto creamos un predicado “**precioIndividual**” que abarque los diferentes casos.

```
precioIndividual(libro(Precio), Precio).
precioIndividual(productoEnOferta(_, Precio), Precio).
precioIndividual(mp3(_, Duracion), Precio) :-
    Precio is Duracion * 2.
```

De esta manera podemos extraer la repetición de lógica del predicado **precioItemPaquete**, entonces creamos otro en el cual apliquemos lo hecho hasta ahora

```
precioDeItemEnPaquete(Paquete, Precio) :-
    itemPaquete(Paquete, Item),
    precioIndividual(Item, Precio).
```

En este predicado, primero se verifica que haya un **Item** en el **Paquete**, de esta manera se separa el **functor** que utilizaremos para definir el **precioIndividual**.

Definimos las modificaciones:

```
costoEnvio(Paquete, PrecioTotal) :-
    findall(PrecioItem, precioDeItemEnPaquete(Paquete, PrecioItem), Precios),
    sumlist(Precios, PrecioTotal).
```