Predicados aritméticos

- Hasta ahora hemos visto la notación de Peano para la aritmética (cero y sucesor), pero en programas reales esto es poco práctico.
- En Prolog los números son constantes del lenguaje.
- Además, se incorpora de forma estándar el predicado **is/2** para evaluar una expresión aritmética formada por números y operadores ($+, -, *, /, \hat{}, sin/1, cos/1, log/1$, etc.):

```
?- X is 2+3*5.

X = 17
```

- `is/2` requiere que los argumentos estén instanciados adecuadamente:
  - El primer argumento debe ser una variable o un término que represente un número.
  - El segundo argumento debe ser un término que represente una **expresión aritmética evaluable**
- Otros predicados aritméticos son: $</2$, $>/2$, $=</2$, $>=/2$, $=:=/2$, $=\backslash=/2$

=:= evalua numéricamente ambos lados y si coinciden da True

== evalua sintácticamente ambos lados y si son idénticos da True

Predicados aritméticos (cont.)

- `is/2` y los demás predicados aritméticos no son reversibles: `6 is X+X` no es correcto si `X` es una variable libre.
- Pero sí es correcto: `?- Y = 3*4+2, X is Y/2.`
- Ejercicios:
  - Diseña un predicado que calcule el factorial de un número.
  - Diseña un predicado que sume 1 a todos los elementos de una lista.
  - Diseña un predicado que, dadas dos listas de números de igual longitud, devuelva una lista resultado de sumar los elementos de las listas anteriores dos a dos.
  - Diseña un predicado que, dado un árbol binario de números enteros positivos, obtenga el valor máximo.

Predicados metalógicos. Comprobación de tipo

- Los predicados *metalógicos* son aquellos que no se pueden representar en lógica de primer orden.
- En Prolog existen predicados para determinar el tipo de datos de un término:
  - `integer(X)` tiene éxito si `X` está instanciado a un número entero.
  - `float(X)` tiene éxito si `X` está instanciado a un número en coma flotante.
  - `number(X)` tiene éxito si `X` está instanciado a un número.
  - `atomic(X)` tiene éxito si `X` está instanciado a una constante numérica o no numérica.
    Ejemplo: los objetivos `atomic(f(3))` y `atomic(3+4)` fallan.
  - `atom(X)` tiene éxito si `X` está instanciado a una constante no numérica (en el lenguaje Prolog las constantes se denominan *átomos*).
- No se pueden utilizar para **generar** constantes: si el argumento es una variable libre, fallan.

  Obs: atom([ ]) da fallo pero atomic([ ]) da éxito

Predicados metalógicos. Comprobación de tipo (cont.)

- Por ejemplo, se pueden utilizar los predicados de comprobación de tipo para definir un predicado `suma/3` reversible:

```
suma(A,B,C):- number(A), number(B), C is A+B.
suma(A,B,C):- number(A), number(C), B is C-A.
suma(A,B,C):- number(B), number(C), A is C-B.
```

- Sin embargo, `suma(X,Y,10)` falla si `X` e `Y` son variables.
- **Ejercicios:**
  - Define el predicado `sumanum/2` que proporcione la suma de los elementos numéricos de una lista que puede tener cualquier tipo de términos en sus elementos.
    - ¿Qué ocurre si pedimos todas las soluciones de `sumanum([3,a,4])`? (Más adelante veremos la forma de solucionarlo).
  - Define el predicado `listapos/2` que dada una lista de números proporcione la lista de los números positivos. ¿Qué ocurre si pedimos todas las soluciones?.
  - Define el predicado `particion(P,L,Men,May)` que, dada una lista de números `L` proporciona en `Men` la lista de elementos menores o iguales a `P` y en `May` los elementos mayores a `P`. Utilízalo para definir `quicksort/2`.

Predicados metalógicos. Inspección de estructuras

- Las estructuras que hemos manejado hasta ahora (por ejemplo, `arbol(a,void,void)`) se pueden manejar en Prolog de forma genérica.
- Existen tres predicados que permiten descomponer y construir estructuras dinámicamente: `functor/3`, `arg/3` y el operador `'=..'`.
- `functor(T,Fn,Ar)` tiene éxito si el término `T` tiene nombre de functor `Fn` y aridad `Ar`. Permite varios modos de uso:
  - Si `T` está instanciado a una estructura, unifica `Fn` con el nombre de functor y `Ar` con su aridad.
  - Si `Fn` y `Ar` están instanciados a una constante y un número natural, unifica `T` con ese nombre de functor y aridad.
- Ejemplos:

```
?- functor(arbol(A,B,C),Fn,Ar).
Fn = arbol,
Ar = 3.
```

```
?- functor(T,arbol,3).
T = arbol(_G236, _G237, _G238).
```

- ¿Cuál es el resultado del objetivo `functor([3,4],Fn,Ar)`?.

Predicados metalógicos. Inspección de estructuras (cont.)

- `arg(N,T,Arg)` tiene éxito si el término `T` tiene en el argumento `N` el término `Arg`. Modos de uso:
  - Si `N` está instanciado a un número natural y `T` a una estructura (con aridad `N` al menos), unifica `Arg` con el `N`-ésimo argumento de `T`.
  - (exclusivo de SWI) Si `T` está instanciado a una estructura, unifica en *backtracking* `N` y `Arg` con los distintos argumentos de `T`.
- Ejemplos:

```
?- arg(2,progenitor(juan,sara),V).
V = sara
?- arg(N,progenitor(juan,sara),V).
N = 1
V = juan ;
N = 2
V = sara
```

- ¿Qué devuelve el objetivo `arg(N,[a,b,c(6)],V)`?

Predicados metalógicos. Inspección de estructuras (cont.)

- `T =.. Lista` tiene éxito si `Lista` es una lista que contiene como primer elemento el nombre del functor del término `T` y como resto de los elementos los argumentos del término `T`.

$$\underbrace{<functor>\,(arg_1, \ldots, arg_n)}_{\text{Término}} \quad =.. \quad \underbrace{[<functor>, arg_1, \ldots, arg_n]}_{\text{Lista}}$$

- Modos de uso:
  - Si `T` está instanciado a una estructura, unifica `Lista` con el nombre del functor y los argumentos de `T`.
  - Si `Lista` está unificado a una lista (y el primer argumento es un átomo de Prolog: una constante no numérica), unifica `T` con el término tal que su functor es el primer elemento de `Lista` y sus argumentos son los demás elementos de `Lista`.
- Ejemplos:

```
?- progenitor(juan,sara) =.. L.
L = [progenitor, juan, sara]

?- f(a,b(d)) =.. L.
L = [f, a, b(d)]
```

```
?- T =.. [f, a, b(d)].
T = f(a, b(d))

?- [1,2] =.. L.
L = ['.', 1, [2]]
```

Predicados metalógicos. Inspección de estructuras (cont.)

- **Ejercicios:**
  - Define un predicado que, dado un término, obtenga la lista de los componentes atómicos (constantes, numéricas o no) del término. Por ejemplo, `term_atomic(arbol(1,arbol(a,void),void),L)` debe unificar `L` con la lista `[1,a,void,void]`.
    - ¿Qué ocurre con la consulta `term_atomic(arbol(X,Y,void),L)`?
  - Define un predicado que determine si un término es subtérmino de otro. Por ejemplo, `subterm(b,arbol([a,b,c,d],void,void))` tiene éxito, y `subterm(X,arbol([a,b,c,d],void,void))` debe devolver todos los subtérminos del segundo argumento.
    - ¿Qué ocurre con la consulta `subterm(d,arbol([a,b,c,d],X,void))`?

Predicados metalógicos. Estado de las variables

- Existen predicados metalógicos para consultar el estado de instanciación de las variables:
  - `var(X)` tiene éxito si `X` es una variable libre.
  - `nonvar(X)` tiene éxito si `X` **no** es una variable (pero puede contener variables).
  - `ground(X)` tiene éxito si `X` no contiene variables libres.
- También se pueden comparar términos entre sí:
  - `X == Y` si los términos `X` e `Y` son **idénticos** (con las mismas variables).
  - `X \== Y` si los términos `X` e `Y` **no** son idénticos.
  - `X @< Y`, `X @> Y` `X @=< Y` y `X @>= Y` permiten comparar términos (en orden alfabético): `f(a) @< f(b)` tiene éxito y `f(b) @> g(a)` falla (La comparación de variables depende del sistema).
- Ejercicios:
  - Modifica el predicado `term_atomic(Term,L)` para que no se produzca un error cuando `Term` contenga variables.
  - Modifica el predicado `subterm(Sub,Term)` para que no instancie variables de los argumentos.

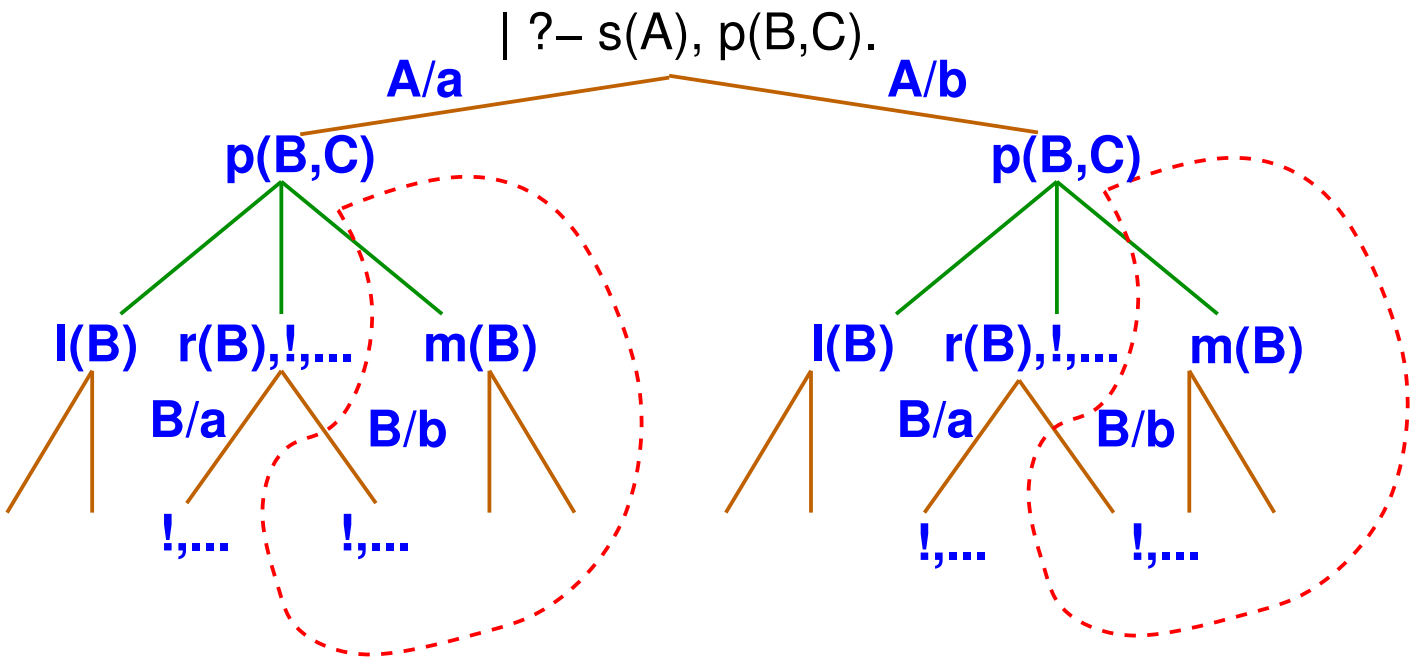
Control en Prolog: el corte (!)

- El predicado `!` (leído *corte*) proporciona control sobre el mecanismo de backtracking de Prolog:
  - Siempre tiene éxito,
  - pero tiene el **efecto lateral** de podar todas las elecciones alternativas en el nodo correspondiente en el árbol de búsqueda.
- Este predicado afecta al **comportamiento operacional** de Prolog.
  - es en cierto sentido un *predicado impuro* (ajeno a la lógica)
  - pero es un predicado muy útil... casi fundamental para el programador de Prolog

Control en Prolog: el corte (cont.)

```
s(a).          p(X,Y):- l(X), ...          r(a).
s(b).          p(X,Y):- r(X), !, ...       r(b).
               p(X,Y):- m(X), ...
```

- Supongamos que se realiza la siguiente consulta:



- La segunda alternativa de `r(X)` no se considera.
- La tercera cláusula de `p/2` no se considera.

Control en Prolog: el corte (cont.)

- Cuando se resuelve un objetivo p con una cláusula de la forma:

$$p :- \quad q_1, \ldots, q_n, !, r_1, \ldots, r_m$$

- Prolog intenta resolver los objetivos $q_1, \ldots, q_n$ normalmente (haciendo backtraking sobre cada uno de ellos si es necesario);
- pero una vez que se alcanza el corte ! se descartan automáticamente:
  - las alternativas que quedasen por explorar para $q_1, \ldots, q_n$
  - el resto de claúsulas para p que vengan a continuación de esta
- Sí que se puede hacer backtraking sobre $r_1, \ldots, r_m$

Control en Prolog: el corte (cont.)

- Por ejemplo, en el predicado `sumanum/2` que proporciona la suma de los elementos numéricos de una lista, se puede añadir un corte para evitar respuestas incorrectas:

```
sumanum([],0).
sumanum([X|Xs],N):- number(X),!, sumanum(Xs,N1), N is X+N1.
sumanum([_|Xs],N):- sumanum(Xs,N).
```

- Cuando se comprueba que el primer elemento de la lista es un número, se puede **descartar con seguridad** el caso en que no lo es (representado en la tercera cláusula de `sumanum/2`).
- El corte permite reducir el tamaño del árbol de búsqueda:

```
max(X,Y,X):- X>Y, !.
max(X,Y,Y):- X=<Y.
```

- Esto puede aumentar la eficiencia de algunos predicados.

Control en Prolog: el corte (cont.)

- Sin embargo, si se utiliza el corte para representar la semántica del programa se pueden producir soluciones incorrectas:

```
max(X,Y,X):- X>Y, !.
max(X,Y,Y).
```

- ¿Qué resultado proporciona la consulta `?- max(5,2,2).`?
- Otro ejemplo:

```
membercheck(X,[X|Xs]):- !.
membercheck(X,[_|Xs]):- membercheck(X,Xs).
```

- Sin embargo, este predicado no nos permite **reevaluar** el objetivo para distintos valores de `X`.
- `membercheck/2` sólo sirve para comprobar que un elemento dado está en la lista.
- Por ejemplo, `?- membercheck(X,[a,b,c]),membercheck(X,[b,c]).` no obtiene ninguna respuesta.

Control en Prolog: negación por fallo

- Utilizando el corte (y `fail/0`) es posible definir un predicado que tenga el comportamiento contrario a otro: que `p(X)` falle cuando `q(X)` tenga éxito y viceversa:

```
p(X):- q(X),!,fail.
p(_).
```

(`fail/0` es un predicado que nunca tiene éxito.)

- Esto se puede generalizar utilizando *orden superior* para definir la **negación**:

```
not(Goal):- call(Goal),!,fail.
not(_).
```

- `call/1` permite evaluar un objetivo que se le pasa como argumento.
- La negación en Prolog se debe entender como el **fallo finito** del objetivo.
- En Prolog, la negación está implementada con el operador `\+ Goal`.

Control en Prolog: negación por fallo (cont.)

- La negación de prolog es **negación por fallo**:
  - `not(P)` tiene éxito $\Leftrightarrow$ el árbol de resolución de P es finito y todos sus nodos son de fallo
  - `not(P)` tiene éxito $\Leftrightarrow$ P no es consecuencia lógica de las reglas del programa y esto es puede demostrar en tiempo finito
  - `not(P)` tiene éxito $\not\Leftrightarrow$ `not(P)` es consecuencia lógica de las reglas.
  - `not(P)` falla $\Leftrightarrow$ P tiene (algún) éxito en tiempo finito.
- La negación por fallo es solamente una aproximación a la negación lógica, y tiene restricciones de uso.
- En particular, la negación **nunca instancia variables** del objetivo negado. Por ejemplo:

```
estudiante(pepe).   estudiante(rufo).   casado(pepe).
estudiante_soltero(X):- estudiante(X),not(casado(X)).
```

- ¿Qué ocurre si se define así?

```
estudiante_soltero(X):- not(casado(X)),estudiante(X).
```

Control en Prolog: disyunción e *if-then-else*

- En Prolog, los objetivos del cuerpo de una cláusula se separan por comas, lo que se interpreta como su **conjunción**.
- Además, se puede utilizar el punto y coma para representar la **disyunción** de objetivos.
- Suele ser útil para evitar el uso de predicados auxiliares, y hacer más compactos los predicados con varias cláusulas con la misma cabeza.
- Por ejemplo:

```
membercheck(X,[Y|Xs]):- ( X = Y, ! ; membercheck(X,Xs)).
```

- Además, se utiliza para la definición de estructuras *if-then-else*

Control en Prolog: disyunción e *if-then-else* (cont.)

- Se puede definir la estructura condicional
  Si P(X) entonces Q(X) si no R(X) con el predicado siguiente:

```
siPentQsinoR(X):- P(X), !, Q(X).
siPentQsinoR(X):- R(X).
```

- En Prolog se puede utilizar la siguiente notación dentro del cuerpo de una cláusula para representar un *if-then-else*:

```
P(X) -> Q(X) ; R(X)
```

- Ejemplo:

```
sumanum([],0).
sumanum([X|Xs],N):-
        ( number(X) ->
          sumanum(Xs,N1),
          N is X+N1
        ;
          sumanum(Xs,N)
        ).
```

Orden superior. Predicados de agregación

- El predicado básico de orden superior es la *metallamada*, que en Prolog es `call/1`.
- `call(X)` evalúa el término `X` que recibe como argumento como si fuera un objetivo, y unifica las variables que pudiera contener apropiadamente.
- `X` **debe** estar instanciado a un término que represente un objetivo existente. Por ejemplo:

```
q(a).                    p(X) :- call(X).
?- p(q(Y)).
Y = a
```

- **Ejercicio**: define un predicado que, dado el nombre de un predicado de aridad 1 y una lista, tenga éxito si la evaluación de ese predicado para todos los elementos de la lista tiene éxito.
- En algunos sistemas como SWI, se puede utilizar orden superior simplemente utilizando una variable como objetivo. Por ejemplo:

```
?- X = append(A,B,[a,b]), X.
```

Orden superior. Predicados de agregación (cont.)

- Otros predicados definidos en Prolog que implementan orden superior se utilizan para **recolectar las respuestas de objetivos**. Son los **predicados de agregación**
- Existen tres predicados de agregación: `findall/3`, `setof/3` y `bagof/3`.
- `findall/3` permite obtener **todas las respuestas de un objetivo** en una lista.
- `findall(Term, Objetivo, Lista)` proporciona en `Lista` todas las instancias de `Term` que satisfacen la evaluación de `Objetivo`.
- Por cada una de las respuestas de `Objetivo`, `findall` unifica el término `Term` con las variables del objetivo y lo añade a `Lista`.
- Las variables libres de `Objetivo` que no aparecen en `Term` se suponen cuantificadas existencialmente.

Orden superior. Predicados de agregación (cont.)

- Por ejemplo, el siguiente objetivo proporciona los resultados:

```
?- subterm(X,f(a,g(b),3)).
X = f(a, g(b), 3) ;
X = 3 ;
X = g(b) ;
X = b ;
X = a ;
false.
```

- Utilizando `findall/3`, se obtiene:

```
?- findall(X,subterm(X,f(a,g(b),3)),L).
L = [f(a, g(b), 3), 3, g(b), b, a].
```

- El término del primer argumento puede contener varias variables:

```
?- findall(par(X,Y),append(X,Y,[a,b,c]),L).
L = [par([],[a,b,c]), par([a],[b,c]), par([a,b],[c]),
par([a,b,c],[])].
```

- También se pueden utilizar objetivos compuestos:

```
?- findall(X,(member(X,[1,a,3,c]),member(X,[3,4,c,b])),L).
```

Orden superior. Predicados de agregación (cont.)

- `setof(Term,Objetivo,Lista)` proporciona en `Lista` la lista **ordenada y sin repeticiones** de todas las instancias de `Term` que satisfacen `Objetivo`.
- Si `Objetivo` contiene variables distintas de las que aparecen en `Term`, devuelve en *backtracking* tantos resultados como posibles instanciaciones existen para dichas variables.
- Las variables de `Objetivo` que no aparecen en `Term` se pueden cuantificar existencialmente mediante el operador ^.
- Ejemplos:

```
?- setof((X,Y),descendiente(X,Y),L).
L = [(alfonso, carmen), (alfonso, javier), (alfonso,
maria), (alfonso, pedro), (alicia, javier)...].
```

Orden superior. Predicados de agregación (cont.)

- Más ejemplos:

```
?- setof(X,descendiente(X,Y),L).
Y = carmen,
L = [alfonso, juan] ;
Y = javier,
L = [alfonso, alicia, juan, pedro, teresa]

?- setof(X,Y^descendiente(X,Y),L).
L = [alfonso, alicia, juan, pedro, teresa].
```

- `bagof(Term,Objetivo,Lista)` funciona de forma similar a findall, pero por defecto ninguna variable de `Objetivo` que no aparece en `Term` está cuantificada existencialmente:

```
p(a,c). p(b,e). p(a,b). p(b,d). p(a,b).
?- bagof(Y,p(X,Y),L).
L = [c,b,b],
X = a ? ;
L = [e,d],
X = b ?
```

Precedencia de operadores

- Anteriormente hemos visto que en Prolog existen operadores predefinidos: `is/2`, `^/2`, los operadores aritméticos, etc.
- Si no existieran, el objetivo `X is 3*4+Y/2` se debería escribir `is(X,+(*(3,4),/(Y,2)))`.
- Los operadores tienen tres características fundamentales:
  - La **posición del operador**: prefija (por ejemplo, `-3`), infija (`2 + 3`), o postfija.
  - La **precedencia**: `2 + 3 * 4` se lee como `2 + (3 * 4)`.
  - La **asociatividad**: define cómo se interpretan expresiones como `8 - 5 - 2` (asociativo *por la derecha*: `8 - (5 - 2)`).
- Las reglas de precedencia se pueden ignorar utilizando paréntesis.

Precedencia de operadores

- En prolog se pueden utilizar nuevos operadores mediante la directiva `op/3`:

```
:- op(Precedencia,PosAsoc,Nombre).
```

- **Nombre** es una constante Prolog.
- **Precedencia** es un número entre 0 y 1200 (los números más bajos representan **mayor** precedencia).
- **PosAsoc** es una constante de las siguientes:

`xf, yf`	posición postfija
`fx, fy`	posición prefija
`xfx, xfy, yfx, yfy`	posición infija

  - `f` representa la posición del operador,
  - en `x` solo pueden aparecer otros operadores con *menor* precedencia que `f`.
  - en `y` pueden aparecer operadores con precedencia *menor o igual* a `f` (para la asociatividad)

Precedencia de operadores (cont.)

- Algunos de los operadores estándar de SWI-Prolog:

Prec	PosAsoc	Operadores
1200	xfx	`-->, :-`
1200	fx	`:-, ?-`
1150	fx	`dynamic, discontiguous`
1100	xfy	`;, |`
1050	xfy	`->`
1000	xfy	`,`
900	fy	`\+`
700	xfx	`<, =, =.., =@=, =:=, =<, ==, =\=, >, >=, @<, @=<, @>, @>=, \=, \==, is`
600	xfy	`:`
500	yfx	`+, -, /\, \/, xor`
500	fx	`?`
400	yfx	`*, /, //, rdiv, <<, >>, mod, rem`
200	xfx	`**`
200	xfy	`^`
200	fy	`+, -, \`

- `':-'` y la coma también son operadores. ¿Qué efecto tiene esto sobre el if-then-else (`A -> B ; C`)?

Creación de nuevos operadores

- Por ejemplo, se podrían definir operadores para cambiar la sintaxis de if-then-else:

```
:- op(1000,fx,if).
:- op(1050,xfy,then).
:- op(1100,xfy,else).

else(if Condicion then Then,Else):-
     call(Condicion) -> call(Then) ; call(Else).
```

- Con esta definición se pueden definir predicados como:

```
P(X,Y) :- ..., (if X > 4 then Y = 10 else Y = 20).
```

- **Ejercicio:** Define un operador infijo `nand/2` y el predicado correspondiente que realice la operación lógica con el mismo nombre sobre objetivos Prolog, y que permita utilizar objetivos compuestos separados por comas en sus operandos sin necesidad de paréntesis, pero no disyunciones (que utilizan ';').