Justificacion de patrones de instanciacion y Reversibilidad en prolog

%prefijoHasta(?X, +L, ?Prefijo)
prefijoHasta(X, L, Prefijo) :- append(Prefijo, [X | _], L).

Si L no está instanciada, no hay manera de instanciarla, ya que el segundo argumento del append tiene un sufijo sin instanciar, y el tercero es la misma L que no está instanciada.
Si L está instanciada, tanto X como Prefijo pueden estar o no instanciados, porque append se encarga de instanciarlos si no lo están, o de verificar que coincidan con un prefijo de L y el elemento siguiente si ya están instanciados.

%desde(+X, -Y)
desde(X, X).
desde(X, Y) :- desde(X, Z), Y is Z + 1.

Si X no está instanciada, desde(X,Y) va a arrojar el resultado Y = X, y luego al entrar en la segunda cláusula va a arrojar un error al intentar realizar una operación aritmética sobre Z sin instanciar.
Si Y está instanciada, va a tener éxito si Y >= X, pero luego (o siempre, si Y < X) se va a colgar porque va a seguir generando infinitos valores para Z y comparando sus sucesores con Y, lo cual nunca va a tener éxito.

%desde2(+X,?Y)
desde2(X,X).
desde2(X,Y) :- nonvar(Y), X < Y.
desde2(X,Y) :- var(Y), desde2(X,Z), Y is Z + 1.

X debe estar instanciada por el mismo motivo que en desde/2.
Si Y no está instanciada, instancia Y en X para el primer resultado, y luego entra por la tercera cláusula y va generando infinitos valores para Y.
Si Y está instanciada, entra por la primera cláusula si es igual a X, y por la segunda en caso contrario, haciendo una comparación entre dos variables ya instanciadas, lo cual funciona correctamente.

%preorder(+A, ?L)
preorder(Nil, []).
preorder(Bin(I, R, D),[R | L]) :- preorder(I, LI), preorder(D, LD), append(LI, LD, L).

Si A no está instanciado, funciona para el caso L = [], porque solo unifica con la primera cláusula. Pero para L no vacía o no instanciada va a entrar (eventualmente) en la segunda cláusula, y va a llamar a preorder con dos variables sin instanciar, lo cual solo le va a permitir generar árboles Nil y listas vacías para luego volver a llamarse infinitamente con argumentos sin instanciar.
Si A está instanciado y L no, instancia L con [] si A es Nil, y en caso contrario calcula los respectivos preorders con I y D ya instanciados y los junta con append.
Si ambos argumentos están instanciados, utiliza unificación y/o append para verificar que L coincida con el preorder de A.
Predicados Prolog
Predicados: =, sort, msort, length, nth1, nth0, member, append, last, between, is_list, list_to_set, is_set, union, intersection, subset, subtract, select, delete, reverse, atom, number, numlist, sum_list, flatten
Operaciones extra-lógicas: is, \=, ==, =:=, =\=, >, <, =<, >=, abs, max, min, gcd, var, nonvar, ground, trace, notrace, make, halt

p(+A) indica que A debe proveerse instanciado.
p(-A) indica que A no debe estar instanciado.
p(?A) indica que A puede o no proveerse instanciado.

var(A) tiene ´exito si A es una variable libre.
nonvar(A) tiene ´exito si A no es una variable libre.
ground(A) tiene ´exito si A no contiene variables libres.

desde(X, X).
desde(X, Y) :- N is X+1, desde(N, Y).

Un predicado que usa el esquema G&T se define mediante otros dos:
pred(X1,...,Xn) :- generate(X1, ...,Xm), test(X1, ...,Xm).
Esta division de tareas implica que:
generate(...) deber´a instanciar ciertas variables.
test(...) deber´a verificar si los valores intanciados pertenecen a
la solucion, pudiendo para ello asumir que ya esta instanciada.

```
iesimo(0,[X|_],X).
iesimo(I,[_|XS],X) :- iesimo(I2,XS,X), I is I2 + 1.

desde2(X,X).
desde2(X,Y) :- var(Y), N is X+1, desde2(N,Y).
desde2(X,Y) :- nonvar(Y), X < Y.

pmq(X,Y) :- between(0,X,Y), Y mod 2 =:= 0.

paresSuman(S,X,Y) :- S1 is S-1, between(1,S1,X), Y is S-X.

generarPares(X,Y) :- desde2(2,S), paresSuman(S,X,Y).

coprimos(X,Y) :- generarPares(X,Y), gcd(X,Y) =:= 1.


altaMateria(plp).
altaMateria(aa).
altaMateria(metnum).

liviana(plp).
liviana(aa).
liviana(eci).

obligatoria(plp).
obligatoria(metnum).

leGusta(M) :- altaMateria(M).
leGusta(M) :- liviana(M).

hacer(M) :- leGusta(M), obligatoria(M).

hacerV2(M) :- setof(X,(leGusta(X),obligatoria(X)),L),
member(M,L).

% corteMasParejo(+L,-L1,-L2)
corteMasParejo(L,L1,L2) :- unCorte(L,L1,L2,D),
not((unCorte(L,_,_,D2), D2 < D)).
unCorte(L,L1,L2,D) :- append(L1,L2,L), sumlist(L1,S1),
sumlist(L2,S2), D is abs(S1-S2).
%
```

Implementar un predicado per´ımetro(?T,?P) que es verdadero cuando T es un tri´angulo y P es su per´ımetro. No se deben generar
resultados repetidos (no tendremos en cuenta la congruencia entre tri´angulos: si dos tri´angulos tienen las mismas longitudes, pero en diferente orden, se considerar´an diferentes entre s´ı). El predicado debe funcionar para cualquier instanciaci´on de T y P (no es necesario que funcione para tri´angulos parcialmente instanciados).
Implementar un generador de tri´angulos v´alidos, sin repetir resultados: tri´angulo(-T).
%

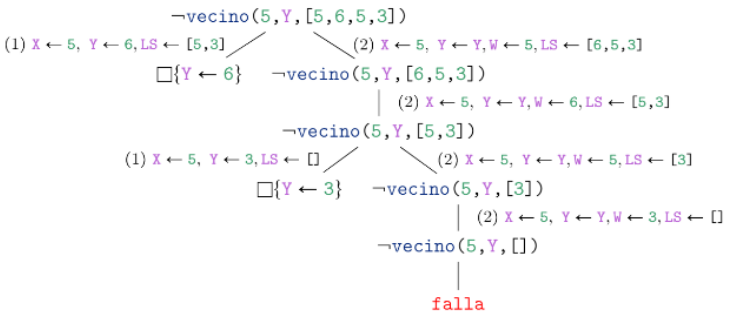
```
esTriangulo(tri(A,B,C)) :- A < B+C, B < A+C, C < B+A.


perimetro(tri(A,B,C),P) :- ground(tri(A,B,C)),
esTriangulo(tri(A,B,C)), P is A+B+C.
perimetro(tri(A,B,C),P) :- not(ground(tri(A,B,C))),
armarTriplas(P,A,B,C), esTriangulo(tri(A,B,C)).

armarTriplas(P,A,B,C) :- desde2(3,P), between(0,P,A), S is
P-A, between(0,S,B), C is S-B.

triangulos(T) :- perimetro(T,_).

%%Arbol de vecino:
vecino(X, Y, [X|[Y|Ls]]). vecino(X, Y, [W|Ls]) :- vecino(X, Y,
Ls).
vecino(5, Y, [5,6,5,3]):
```




```
%

desde(X,X).
desde(X,Y) :- N is X+1, desde(N,Y).


desde2(X,X).
desde2(X,Y) :- var(Y), N is X+1, desde2(N,Y).
desde2(X,Y) :- nonvar(Y), X < Y.
```

```
%TP
%% tablero(+Filas,+Columnas,-Tablero)
% lista(+Largo,-Lista)
lista(0,[]).
lista(N,[X|XS]):- length([X|XS], N), lista(Z ,XS) , X= _ , Z is N-1 .
tablero(0,_,[]).
tablero(F,C,[X|XS]):- length([X|XS],F) , tablero(F1 , C , XS) , X = X1 , lista(C,X1) , F1 is F-1 .


%% ocupar(+Pos,?Tablero)

% elementoTablero(+Pos,?Tablero,?Elemento)
elementoTablero(pos(0,0),[[X|_]|_],X).
elementoTablero(pos(0,C),[[_|XS]|YS],E):- 0 \= C , C1 is C-1 , elementoTablero(pos(0,C1),[XS|YS],E) .
elementoTablero(pos(F,C),[_|XS],E):- 0 \= F ,  F1 is F-1 , elementoTablero(pos(F1,C),XS,E).

ocupar(pos(F,C),T) :- elementoTablero(pos(F,C),T,ocupada).

%% Ejercicio 3
%% vecino(+Pos, +Tablero, -PosVecino)

vecino(pos(X,Y),[T|Ts],pos(F,Y)):- F is X+1, length([T|Ts],P), between(0,P,F).
vecino(pos(X,Y),[T|Ts],pos(F,Y)):- F is X-1, length([T|Ts],P), between(0,P,F).
vecino(pos(X,Y),[T|_],pos(X,J)):- J is Y+1, length(T,P), between(0,P,J).
vecino(pos(X,Y),[T|_],pos(X,J)):- J is Y-1, length(T,P), between(0,P,J).

 vecinoLibre(+Pos, +Tablero, -PosVecino)

vecinoLibre(pos(X,Y),[T|Ts],V) :- vecino(pos(X,Y),[T|Ts],V) , elementoTablero(V,[T|Ts],E),var(E).

%% camino(+Inicio, +Fin, +Tablero, -Camino) .
%caminoValido(+Inicio,+Fin,+Tablero,+Visitados,-Camino)
caminoValido(F,F,_,_,[]).
caminoValido(I,F,T,V,[P|C]):- not(I=F),vecinoLibre(I,T,P),not(member(P,V)),caminoValido(P,F,T,[P|V],C).
%camino(I,I,_,[]).
camino(I,F,T,[I|C]):- caminoValido(I,F,T,[I],C).

%% 5.1. Analizar la reversibilidad de los parámetros Fin y Camino justificando adecuadamente en cada
%% caso por qué el predicado se comporta como lo hace

% Analizando la reversibilidad de F, F al no venir instanciada unificara con la primera linea de camino valido,
% pero al solicitar otra solución en la siguiente linea ya no unificará,
% ya que "not(_ = F)" retornara siempre False. Por lo tanto, F no es reversible

%  En el caso de C, si esta instanciado, va a dar True si cada elemento efectivamente cumple con las condiciones que cada posicion del
%  tiene que cumplir en un camino valido, y que empieza y termine donde lo tiene que cumplir. Caso contrario da false
%  Por lo tanto C es reversible

%% camino2(+Inicio, +Fin, +Tablero, -Camino).

% desdeHasta(+Tablero , -Largo)
largoTablero([TH|TT],L) :- length([TH|TT],F) , length(TH,C) , L is F*C.

% desdeMenoresCaminos(+Tablero, -Camino)
desdeMenoresCaminos(T,C) :- largoTablero(T,F),
                            between(0,F,L),
                            length(C,L).

%camino2(I,I,_,[]).
camino2(I,F,T,[I|C]):- desdeMenoresCaminos(T,[I|C]), caminoValido(I,F,T,[I],C).

%% 6.1. Analizar la reversibilidad de los parámetros Inicio y Camino justificando adecuadamente en
%% cada caso por qué el predicado se comporta como lo hace.
% Cuando I si viene instanciada, ocurre un caso parecido al de F en camino, ya que unifica en el caso donde I es igual que F,
% pero luego al pedir la siguiente solucion da false por el "not(_ = I)". Entonces, I no es reversible.
% En C tambien es analogo al C de 5.1 , ya que devuelve True si cada elemento cumple las condiciones. En caso contrario da false.
% Por lo tanto C es reversible

%% caminoOptimo(+Inicio, +Fin, +Tablero, -Camino)
caminoOptimo(I,F,T,C):- camino(I,F,T,C), not((camino(I,F,T,C1) ,length(C,A),length(C1,B),B<A)).

%% caminoDual(+Inicio, +Fin, +Tablero1, +Tablero2, -Camino)

caminoDual(I, F , T1 , T2 , C):- camino(I,F,T1,C) , camino(I,F,T2,C).
```

```
sublistaMasLargaDePrimos(L,P) :-
sublistaDePrimosDeLong(L,P,Long),
not((sublistaDePrimosDeLong(L,_,Long2), Long2 > Long)).

sublistaDePrimosDeLong(L,P,Long) :- sublista(L,P),
soloPrimos(P), length(P,Long).

sublista(_,[]).
sublista(L,S) :- append(P,_,L), append(_,S,P), S \= [].

soloPrimos(L) :- not((member(X,L), not(esPrimo(X)))).

% esPrimo(+P)
esPrimo(P) :- P \= 1, P2 is P-1, not((between(2,P2,D),
mod(P,D) =:= 0)).


%?- listaDeÁrboles(L).
%L = [];

%L = [bin(nil,_,nil)]; ---> [1]
%L = [bin(nil,_,nil), bin(nil,_,nil)]; ---> [1,1]
%L = [bin(nil,_, bin(nil,_,nil))];  ---> [2]
%L = [bin(bin(nil _,nil),_,nil)];   ---> [2]
%L = [bin(nil,_,nil), bin(nil,_,nil), bin(nil,_,nil)]; --> [1,1,1]
%---> [1,2]
%---> [1,2]
%---> [2,1]
%---> [2,1]
%---> [3]

desde(X,X).
desde(X,Y) :- N is X+1, desde(N,Y).

listaDeArboles(L) :- desde(0,S), listaAcotadaDeArboles(S,L).

listaAcotadaDeArboles(0,[]).
listaAcotadaDeArboles(S,[X|XS]) :- between(1,S,Na),
                    arbolDeN(Na,X), S2 is S-Na,
                    listaAcotadaDeArboles(S2,XS).


arbolDeN(0,nil).
arbolDeN(N,bin(I,_,D)) :- N > 0, N2 is N-1,
paresQueSuman(N2,NI,ND), arbolDeN(NI,I),
arbolDeN(ND,D).

paresQueSuman(S,X,Y) :- between(0,S,X), Y is S-X.
```

```
%% Ejercicio 4

% juntar(?Lista1,?Lista2,?Lista3)
juntar([],Lista2,Lista2).
juntar([X|T1],Lista2,[X|T3]) :- juntar(T1,Lista2,T3).

%% Ejercicio 5

%% last(?L, ?U), donde U es el último elemento de la lista L
last([ X ], X).
last((_|T),Y) :- last(T,Y).

%mismaLongitud (+L,+L1)
mismaLongitud(L,L1) :- length(L,N) , length(L1,N).

%% reverse(+L, -L1), donde L1 contiene los mismos elementos que L, pero en orden inverso.
reverse([],[]).
reverse([X|XS],Y) :- mismaLongitud([X|XS],Y), reverse(XS,Z), append(Z, [X] , Y).


%% maxlista(+L , -M)  donde M es el maximo de cada lista.

maxlista([ X ] , X) .
maxlista([X,XS], X) :- maxlista(XS,Z), Z =< X.
maxlista([X,XS], Y) :- maxlista(XS,Y), X =< Y.

%% prefijo(?P , +L) donde S es el prefijo de la lista L
%prefijo([],[_]).
%prefijo([X],[X]).
prefijo([X],[X|_]).
prefijo([X|XS],[Y|YS]) :- prefijo(XS,YS) , X=Y.

%% sufijo(?S, +L), donde S es sufijo de la lista L.
%sufijo([],[_]).
%sufijo([X],[X]).
sufijo(X,Y) :- reverse(Y,Z), prefijo(Z2,Z) , reverse(Z2,X) .

%sublista(?S, +L), donde S es sublista de L.
sublista(X,Y) :- append(Z,_,Y) , append(_,X,Z) .

%pertenece(?X, +L), que es verdadero sii el elemento X se encuentra en la lista L. (Este predicado ya vienedefinido en Prolog y se llama member).
%pertenece(X,[X]).
pertenece(X,[X|_]).
pertenece(Y,[_|XS]) :- pertenece(Y,XS).


% Ejercicio 6

aplanar([],[]).
aplanar([ [] | T ], Res) :-
        aplanar(T, Res).
aplanar([ [X | T1 ] | T ], Res) :-
        aplanar([ X | T1 ], Y),
        aplanar(T, RecT),
        append(Y, RecT, Res).
aplanar([ X | T ], [X | Res]) :-
        not(is_list(X)),
        aplanar(T, Res).


% Ejercicio 7

%i. intersección(+L1, +L2, -L3)
%interseccionAux([L1H|L1T],L2,L3,V):- member(L1H,L2), member(L1H,L3) , not(member(L1H,V)) , interseccionAux (L1H,L2,L3,[L3H|V]).
%interseccion(L1,L2,L3):-interseccionAux(L1,L2,L3,[]).

% Ejercicio 12

% bin(izq, v, der),
vacio(nil).

raiz(bin(_, V, _),V).


altura(nil,0).
altura(bin(IZQ,_,DER),N) :- altura(IZQ,NI) , altura(DER,ND) , N is max(NI,ND) + 1 .

cantidadDeNodos(nil,0).
cantidadDeNodos(bin(IZQ,_,DER),N):- cantidadDeNodos(IZQ,NI) , cantidadDeNodos(DER,ND) , N is NI + ND + 1.

% ejemplo : funcion(bin(bin(bin(nil,2,nil),4,nil),5,bin(nil,6,nil)),X).

% Ejercicio 13

% inorder(+AB,-Lista)

inorder(nil,[]).
inorder(bin(IZQ,R,DER),L) :- inorder(IZQ,LI) , inorder(DER, LR) , append(LI,[R],L1) , append(L1,LR,L).

% no es mi sol
%arbolConInorder(-L, +AB)
arbolConInorder([], nil).
arbolConInorder(XS, bin(AI, X, AD)) :-
        reparto(XS, 2, [LI, [X | LD]]),
        arbolConInorder(LI, AI),
        arbolConInorder(LD, AD).
% ejemplo : arbolConInorder([2, 4, 5, 6],X).

% aBB(+T)


aBB(nil).
aBB(bin(nil,_,nil)).
aBB(bin(IZQ,R,nil)) :- raiz(IZQ,RI) , R >= RI , aBB(IZQ).
aBB(bin(nil,R,DER)):- raiz(DER,RD) , R =< RD , aBB(DER).
aBB(bin(IZQ,R,DER)):- raiz(IZQ,RI) , raiz(DER,RD) , R >= RI , R =< RD , aBB(IZQ) , aBB(DER).
```

```
% ejemplo aBB(bin(bin(bin(nil,2,bin(nil,3,nil)),4,nil),5,bin(nil,6,nil))).

% aBBInsertar(+X, +T1, -T2)
aBBInsertar(X,nil,bin(nil,X,nil)).
%aBBInsertar(X,bin(IZQ,X,DER),bin(IZQ,X,DER)). % por si son iguales , aunque no hace falta
aBBInsertar(X,bin(IZQ,R,DER),bin(IZQ2,R,DER)) :- X < R , aBBInsertar(X,IZQ,IZQ2).
aBBInsertar(X,bin(IZQ,R,DER),bin(IZQ,R,DER2)) :- X > R , aBBInsertar(X,DER,DER2).

% Ejercicio 15

desde2(X,X).
desde2(X,Y) :- var(Y), N is X+1, desde2(N,Y).
desde2(X,Y) :- nonvar(Y), X < Y.
todasLasFilasSumanLoMismo(XS) :- not((member(E1,XS), member(E2,XS) , E1 \= E2 , sumlist(E1,N1) , sumlist(E2,N2) , N1 \= N2 )).

cuadradoLat(N,M):- desde2(0,P), matrices(N,P,N,M).

matrices(_,_,0,[]).
matrices(N,P,C,[L|M]):- C > 0, generarLista(N,P,L) ,Y is C-1, matrices(N,P,Y,M).

generarLista(0,0 ,[]).
generarLista(N,P ,[X|XS]) :-
    N > 0, P >= 0,
    between(0, P, X),
    R is P - X, Y is N-1,
    generarLista(Y,R,XS).

%ahora con generate and test.
generarLista1(0,_ ,[]).
generarLista1(N,P ,[X|XS]) :-
    N > 0, P >= 0,
    between(0, P, X),
     Y is N-1,
    generarLista1(Y,P,XS).

cuadradoLat1(0,[]).
cuadradoLat1(N,M):- desde2(0,P), matrices1(N,P,N,M) , todasLasFilasSumanLoMismo(M).

matrices1(_,_,0,[]).
matrices1(N,P,C,[L|M]):- C > 0, generarLista1(N,P,L),Y is C-1, matrices1(N,P,Y,M).

% Parcial 1er cuat 2023

% palabra(+A,+N,-P)
palabra(_,0,[]).
palabra(A,N,[PH|PT]) :- N > 0 , I is N-1 , member(PH,A) , palabra(A,I,PT).

%frase(+A,-F).

generarFrases(_,0,[]).
generarFrases(A,P,[X|XS]):- between(1,P,Na), palabra(A,Na,X), S2 is P-Na, generarFrases(A,S2,XS).

frase(A,F) :- desde2(0,P), generarFrases(A,P,F).

% Estos ejs no son mios:

%borrar(+ListaOriginal, +X, -ListaSinXs)
borrar([], _, []).
borrar([X | T], X, ListaSinXs) :-
        borrar(T, X, ListaSinXs).
borrar([ Y | T], X, [ Y | Rec ]) :-
        X \= Y,
        borrar(T, X, Rec).
%sacarDuplicados(+L1, -L2)
sacarDuplicados([], []).
sacarDuplicados([ X | T], [ X | Rec ]) :-
        borrar(T, X, T1),
        sacarDuplicados(T1, Rec).
%concatenarTodas( +LL, -L)
concatenarTodas([], []).
concatenarTodas([ X | T], Res) :-
        concatenarTodas(T, Rec),
        concatenar(X, Rec, Res).
%todosSusMiembrosSonSublitas(+LListas, +L)
todosSusMiembrosSonSublitas([], _).
todosSusMiembrosSonSublitas([ X | XS], L) :-
        sublista(X, L),
        todosSusMiembrosSonSublitas(XS, L).
%reparto(+L, +N, -LListas)
reparto(L, N, LListas) :-
        length(LListas, N),
        todosSusMiembrosSonSublitas(LListas, L),
        concatenarTodas(LListas, L).
%repartoSinVacias(+L, -LListas)
repartoSinVacias(L, LListas) :-
        length(L, N),
        between(1, N, X),
        reparto(L, X, LListas),
        not(member([], LListas)).
```

-- Ejercicio 1

-- a. 10 numberOfDigitsInBase: 2

-- 10 es objeto receptor
-- numberOfDigitsInBase es mensaje
-- 2 es colaborador

-- b. 10 factorial

-- 10 es objeto receptor
-- factorial es mensaje

-- c. (20 + 3) * 5

-- 20 es objeto receptor
-- + es mensaje
-- 3 es colaborador
-- (20 + 3) es objeto receptor
-- * es mensaje
-- 5 es colaborador

-- d. 20 + (3 * 5)

-- 20 es objeto receptor
-- + es mensaje
-- el resultado de (3 * 5) es colaborador
-- 3 es objeto receptor
-- * es mensaje
-- 5 es colaborador

-- e. December first, 1985

-- Pharo no entiende el mensaje December, por lo que no se puede ejecutar

-- f. 1 = 2 ifTrue: ['what!?']

-- 1 es objeto receptor
-- = es mensaje
-- 2 es colaborador

-- el resultado de (1 = 2) es objeto receptor
-- ifTrue: es mensaje
-- ['what!?'] es colaborador

-- g. 1@1 insideTriangle: 0@0 with: 2@0 with: 0@2.

-- 1@1 es objeto receptor
-- insideTriangle:with:with: es mensaje
-- 0@0, 2@0 y 0@2 son colaboradores

-- h. 'Hello World' indexOf: $o startingAt: 6

-- 'Hello World' es objeto receptor
-- indexOf:startingAt: es mensaje
-- $o y 6 son colaboradores

-- i. (OrderedCollection with: 1) add: 25; add: 35; yourself.

-- OrderedCollection with: 1 es objeto receptor
-- add: es mensaje
-- 25 es colaborador
-- add: es mensaje
-- 35 es colaborador
-- yourself es mensaje

-- j. Object subclass: #SnakesAndLadders instanceVariableNames: 'players squares turn die over' classVariableNames: '' poolDictionaries: '' category: 'SnakesAndLadders'

-- Object es objeto receptor (en este claso una clase)
-- subclass:instanceVariableNames:classVariableNames:poolDictionaries:category: es mensaje
-- #SnakesAndLadders, 'players squares turn die over', '', '', 'SnakesAndLadders' son colaboradores

-- Ejercicio 3

-- Mostrar expresiones válidas de Smallktalk que contengan los siguientes conceptos:

-- a) Objeto
-- 10

-- b) Mensaje unario
-- 10 factorial

-- c) Mensaje binario
-- 10 + 5

-- d) Mensaje keyword
-- 10 factorial printString

-- e) Colaborador
-- 10 + 5

-- f) Variable local
-- | x | x := 10

-- g) Asignación
-- x := 10

-- h) Símbolo
-- #hola

-- i) Carácter
-- $a

-- j) Array
-- #(1 2 3)

```
-- Ejercicio 4

-- Indicar el valor que devuelve cada una de las siguientes expresiones:

-- a) [:x | x + 1] value: 2
-- 3


-- b) [|x| x := 10. x + 12] value
-- 22


-- c) [:x :y | |z| z := x + y] value: 1 value: 2
-- 3


-- d) [:x :y | x + 1] value: 1
-- Exception: ArgumentsCountMismatch


-- e) [:x | [:y | x + 1]] value: 2
-- [:y | x + 1]


-- f) [[:x | x + 1]] value
-- [:x | x + 1]


-- g) [:x :y :z | x + y + z] valueWithArguments: #(1 2 3)
-- 6


-- h) [ |z| z := 10. [:x | x + z]] value value: 10
-- 20


-- ¿Cuál es la diferencia entre [|x y z| x + 1] y [:x :y :z| x + 1]?
-- En el primer caso hay tres variables locales, en el segundo caso hay tres argumentos


-- Ejercicio 5

-- Integer factorialsList
--   | list |
--   list := OrderedCollection with: 1.
--   2 to: self do: [:aNumber | list add: (list last) * aNumber ].
--   ^ list

-- a) factorialList: 10
-- el mensaje no tiene objeto receptor


-- b) Integer factorialsList: 10
-- la clase no entiende el mensaje


-- c) 3 factorialsList.
-- #(1 2 6 24)


-- d) 5 factorialsList at: 4
-- 24

-- e) 5 factorialsList at: 6
-- Exception: SubscriptOutOfBounds


-- Ejercicio 6

-- Mostrar un ejemplo por cada mensaje:

-- a) #collect:
-- #(1 2 3 4) collect: [:x | x + 1]
-- #(2 3 4 5)

-- b) #select:
-- #(1 2 3 4) select: [:x | x > 2]
-- #(3 4)

-- c) #inject: into: (how each element of the collection should be combined with the current accumulation to produce a new accumulation value)
-- "(#(1 2 3) inject: OrderedCollection new into: [ :a :e | a add: (e + 1). a ])
-- #(2 3 4)"

-- d) #reduce: (o #fold:)
-- #(1 2 3 4) reduce: [:x :y | x + y]
-- 10

-- e) #reduceRight: (it works like reduce but starts from the right like foldr in Haskell)
-- #(1 2 3 4) reduceRight: [:x :y | x - y]
-- -2

-- f) #do:
-- | accumulator | accumulator := 0. #(1 2 3 4) do: [:x | accumulator := accumulator + x]
-- 10


-- Ejercicio 7

-- SomeClass << foo: x
--  | aBlock z |
--  z := 10.
--  aBlock := [ x > 5 ifTrue: [ z := z + x. ^0 ] ifFalse: [ z := z = x. 5 ] ].
--  y := aBlock value.
--  y := y + z.
--  ^y.

-- a) obj foo: 4.
-- Instance of False did not understand #+

-- b) Message selector: #foo: argument: 5.
-- Es la instancia del mensaje foo con el argumento 5

-- c) obj foo: 10. (Ayuda: el resultado no es 20).
-- 0 (el resultado es 0 porque el bloque retorna 0 si x > 5 y no continúa con el resto de las instrucciones)


-- Ejercicio 8

-- a) #curry, cuyo objeto receptor es un bloque de dos parámetros,
-- y su resultado es un bloque similar al original pero currificado.

-- BlockClosure << curry
--   ^ [ :x | [ :y | self value: x value: y ] ] .


-- b) #flip, que al enviarse a un bloque de dos parámetros,
```

-- devuelve un bloque similar al original, pero con los parámetros en el orden inverso.

-- BlockClosure << flip
-- ^ [:x :y | self value: y value: x] .

-- c) #timesRepeat:, cuyo objeto receptor es un número natural y
-- recibe como colaborador un bloque, el cual se evaluará tantas veces como el número lo indique.

-- Integer << timesRepeat: aBlock
-- 1 to: self do: [aBlock value].

-- Ejercicio 9

-- Agregar a la clase BlockClosure el método de clase generarBloqueInfinito que devuelve un bloque b1 tal que:
-- b1 value devuelve un arreglo de 2 elementos #(1 b2),
-- b2 value devuelve un arreglo de 2 elementos #(2 b3),
-- ...,
-- bi value devuelve un arreglo de 2 elementos #(i bi+1)

-- BlockClosure class << generarBloqueInfinito
-- ^ [:x | #(x (self value: x + 1))] .

-- Ejercicio 10

-- i. Todo objeto es instancia de alguna clase y a su vez, estas son objetos.

-- [Verdadero]

-- ii. Cuando un mensaje es enviado a un objeto, el método asociado en la clase del receptor es ejecutado.

-- [Falso] El método asociado en la clase del receptor es ejecutado si el objeto entiende el mensaje, de lo
-- contrario se busca en la jerarquía de clases para encontrar la implementación del método correspondiente.

-- iii. Al mandar un mensaje a una clase, por ejemplo Object new, se busca en esa clase el método correspondiente.
-- A este método lo clasificamos como método de instancia.

-- [Falso] Al mandar un mensaje a una clase se busca el método de clase correspondiente.

-- iv. Una Variable de instancia es una variable compartida por todas las instancias vivas de una clase, en caso
-- de ser modificada por alguna de ellas, la variable cambia.

-- [Falso] Una Variable de instancia es una variable que pertenece a una instancia de una clase, por lo que no es
-- compartida por todas las instancias vivas de una clase.

-- v. Las Variables de clase son accesibles por el objeto clase, pero al mismo tiempo también son accesibles y
-- compartidas por todas las instancias de la clase; es decir, si una instacia modifica el valor de dicha variable,
-- dicho cambio afecta a todas las instancias.

-- [Falso] Las Variables de clase son accesibles por el objeto clase pero no significa que los valores sean compartidos.
-- Si una instancia modifica el valor de una variable de clase, el cambio no afecta a las demás instancias.

-- vi. Al ver el código de un método, podemos determinar a qué objeto representará la pseudo-variable self.

-- [Verdadero] Representará al objeto que recibe el mensaje.

-- vii. Al ver el código de un método, podemos determinar a qué objeto representará la pseudo-variable super.

-- [Verdadero] Representará a la superclase de la clase que implementa el método.

-- viii. Un Método de clase puede acceder a las variables de clase pero no a las de instancia, y por otro lado,
-- siempre devuelven un objeto instancia de la clase receptora.

-- [Falso] Un Método de clase puede acceder a las variables de clase y no a las de instancia, pero no siempre devuelven
-- un objeto instancia de la clase receptora.

-- ix. Los métodos y variables de clase son los métodos y variables de instancia del objeto clase.

-- [Falso] Los métodos y variables de clase son los métodos y variables de instancia del objeto clase.

-- Ejercicio 11

-- Suponiendo que anObject es una instancia de la clase OneClass que tiene definido el método de instancia
-- aMessage. Al ejecutar la siguiente expresión: anObject aMessage...

-- i. ¿A qué objeto queda ligada (hace referencia) la pseudo-variable self en el contexto de ejecución del método que es invocado?

-- El objeto anObject recibe el mensaje aMessage. En la definición del método de la instancia aMessage, self hace referencia a anObject.

-- ii. ¿A qué objeto queda ligada la pseudo-variable super en el contexto de ejecución del método que es invocado?

-- La pseudo-variable super quedará ligada a la superclase de OneClass.

-- iii. ¿Es cierto que super == self? ¿Es cierto en cualquier contexto de ejecución?

-- No es cierto que super == self. En cualquier contexto de ejecución super hace referencia a la superclase de la clase
-- que implementa el método, mientras que self hace referencia al objeto que recibe el mensaje.

-- Ejercicio 12

-- Se cuenta con la clase Figura, que tiene los métodos perimetro y lados.

-- sumarTodos es un método de la clase Collection, que suma todos los elementos de la colección receptora.
-- El método lados debe devolver un Bag (subclase de Collection) con las longitudes de los lados de la figura.

-- Figura tiene dos subclases: Cuadrado y Círculo. Cuadrado tiene una variable de instancia lado, que representa
-- la longitud del lado del cuadrado modelado; Círculo tiene una variable de instancia radio, que representa
-- el radio del círculo modelado.

-- Se pide que las clases Cuadrado y Círculo tengan definidos su método perímetro. Implementar los métodos
-- que sean necesarios para ello, respetando el modelo (incompleto) recién presentado.

-- Observaciones: el perímetro de un círculo se obtiene calculando: 2 · π · radio, y el del cuadrado: 4 · lado.
-- Consideramos que un círculo no tiene lados. Aproximar π por 3,14.

-- Object subclass: #Figura
-- instanceVariableNames: ''.

-- Figura >> perimetro
-- ^((self lados) sumarTodos).

-- Figura >> lados
-- self subclassResponsibility.

-- Figura subclass: #Cuadrado
-- instanceVariableNames: 'lado'.

-- Cuadrado >> lados
-- ^Bag with: (4 * lado).

-- Cuadrado >> perimetro
-- ^4 * lado.

-- Figura subclass: #Circulo
-- instanceVariableNames: 'radio'.

-- Circulo >> lados
-- ^Bag new. "Los círculos no tienen lados, devolvemos un Bag vacío"

-- Circulo >> perimetro
-- ^2 * 3.14 * radio.

-- Ejercicio 13

-- Object subclass: Counter [

-- | count |
-- class >> new [
-- ^ super new initialize: 0.
--]

-- initialize: aValue [
-- count := aValue.
-- ^ self.
--]

-- next [
-- self initialize: count + 1.
-- ^ count.
--]

-- nextIf: condition [
-- ^ condition ifTrue: [self next] ifFalse: [count]
--]
--]

-- Counter subclass: FlexibleCounter [

-- | block |
-- class >> new: aBlock [
-- ^ super new useBlock: aBlock.
--]

-- useBlock: aBlock [
-- block := aBlock.
-- ^ self.
--]

-- next [
-- self initialize: (block value: count).
-- ^ count.
--]
--]

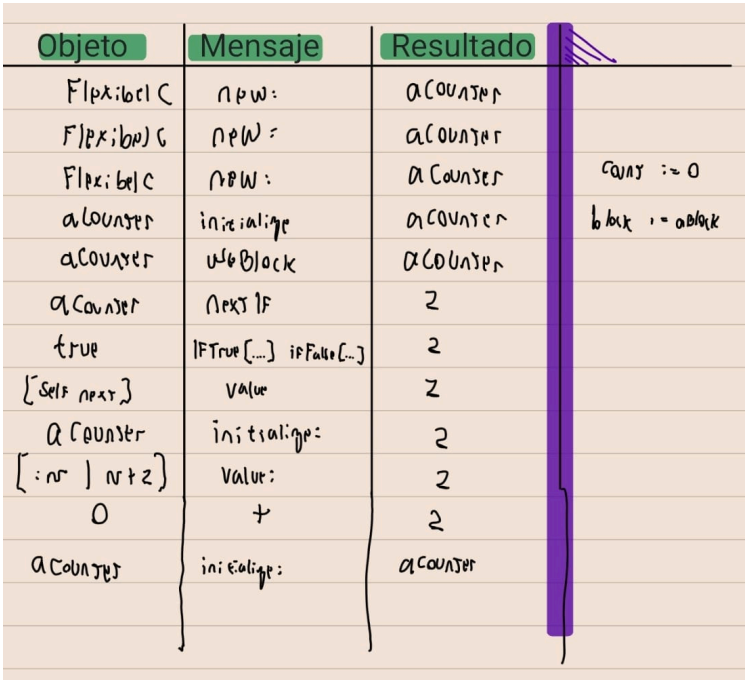
-- Considere la siguiente expresión: aCounter := FlexibleCounter new: [:v | v+2]. aCounter nextIf: true.

-- Se desea saber qué mensajes se envían a qué objetos (dentro del contexto de la clase) y cuál es el resultado
-- de dicha evaluación. Recordar que := y ^ no son mensajes. Recomendación, utilizar una tabla: "Objeto Mensaje Resultado"

-- Tenemos dos instrucciones, la primera crea una instancia de FlexibleCounter y la segunda envía el mensaje nextIf a la misma.

-- Objeto | Mensaje | Resultado
-- FlexibleCounter Class | new: | una instancia de FlexibleCounter
-- aCounter | nextIf: | 2

Mensajes comunes en colecciones:
add: agrega un elemento.
at: devuelve el elemento en una posici´on.
at:put: agrega un elemento a una posici´on.
includes: responde si un elemento pertenece o no.
includesKey: responde si una clave pertenece o no.
Paradigmas (de Lenguajes) de Programaci´on Clase pr´actica: Smalltalk 28 de junio de 2024 14 / 21
Colecciones
Mensajes m´as comunes
do: eval´ua un bloque con cada elemento de la colecci´on.
keysAndValuesDo: eval´ua un bloque con cada par clave-valor.
keysDo: eval´ua un bloque con cada clave.
select: devuelve los elementos de una colecci´on que cumplen un predicado (filter de funcional).
reject: la negaci´on del select:
collect: devuelve una colecci´on que es resultado de aplicarle un bloque a cada elemento de la colecci´on original (map de funcional).
detect: devuelve el primer elemento que cumple un predicado.
detect:ifNone: como detect:, pero permite ejecutar un bloque si no se encuentra ning´un elemento.
reduce: toma un bloque de dos o m´as par´ametros de entrada y hace
fold de los elementos de izquierda a derecha (foldl de funcional).



Metodos creados en Colecciones

```
minimo: aBlock
|b min |
min := 1000.
b:= self map1: aBlock.
b do: [ :each |  min > each ifTrue:[min := each ]].
^min.
map1: aBlock
|c|
c := OrderedCollection new.
self do: [ :each | c add: (aBlock value: each)].
^c
```

Minimo de Christian

```
minimo: aBlock
| minElement minValue |
self do: [:each | | val |
minValue ifNotNil: [
(val := aBlock value: each) < minValue ifTrue: [
minElement := each.
minValue := val]]
ifNil: ["first element"
minElement := each.
minValue := aBlock value: each].
].
^minElement
```

Dado el siguiente código:

```
drone := Drone newWith: [:n1 :n2 | {n1+1 . n2+1}].
drone avanzar.
```

se obtiene la tabla de seguimiento de abajo.

Objeto	Mensaje	Colaboradores	Ubicación del método	Resulta
Drone	newWith:	[:n1 :n2 ...]	Drone	aDrone
Drone	newWith:	[:n1 :n2 ...]	Robot	aDrone
Drone	new	-	Object	aDrone
aDrone	initWith:	[:n1 :n2 ...]	Robot	aDrone
aDrone	init	-	Drone	aDrone
aDrone	avanzar	-	Drone	aDrone
0	<	10	SmallInteger	True
true	ifTrue:	[z:=(z+1)]	True	1
[z:=(z+1)]	value	-	BlockClosure	1
0	+	1	SmallInteger	1
aDrone	avanzar	-	Robot	aDrone
[:n1 :n2 ...]	value: value:	0, 0	BlockClosure	#(1 1)
0	+	1	SmallInteger	1
0	+	1	SmallInteger	1
#(1 1)	at:	1	OrderedCollection	1
#(1 1)	at:	2	OrderedCollection	1