Clase 4

Objetivos

- Introducir **predicados del sistema** Prolog para aritmética y relaciones de orden
- Aritmética y listas
- Acumuladores y predicados ***tail-recursive*** , mejoras en eficiencia
- Documentación: patrones de instanciación de argumentos

Aritmética en Prolog

- Predicado especial del sistema para realizar operaciones aritméticas y lógicas.
- Desde el punto de vista lógico, siempre verdadero.
- Interesa el “efecto de borde”, la realización de cálculos.
- Es el predicado **is**

Aritmética	Prolog
2 + 3 = 5	?- 5 is 2+3.
3 x 4 = 12	?- 12 is 3∗4.
5 – 3 = 2	?- 2 is 5-3.
3 – 5 = -2	?- -2 is 3-5.
4 : 2 = 2	?- 2 is 4/2.
1 es el resto de dividir 7 entre 2	?- 1 is mod(7,2).

Ejemplos de consultas

```
?- 10 is 5+5.
yes

?- 4 is 2+3.
no

?- X is 3 * 4.
X=12
yes

?- R is mod(7,2).
R=1
yes
```

Ejemplos de predicados

```
masTresYPorDos(X, Y):-
    Y is (X+3) * 2.
```

Predicados con aritmética

```
masTresYPorDos(X, Y):-
    Y is (X+3) * 2.
```

```
?- masTresYPorDos(1,X).
X=8
yes

?- masTresYPorDos(2,X).
X=10
yes
```

Cómo funciona

- Es importante saber que +, -, / y * NO realizan operaciones aritméticas

- Expresiones tales como 3+2, 4-7, 5/5 son términos complejos Prolog
  - Functor: +, -, /, *
  - Aridad: 2
  - Argumentos: enteros
  - El operador es infijo

Cómo funciona

```
?- X = 3 + 2.
```

Cómo funciona

```
?- X = 3 + 2.
X = 3+2
yes

?-
```

Cómo funciona

```
?- X = 3 + 2.
X = 3+2
yes

?- 3 + 2 = X.
```

Cómo funciona

```
?- X = 3 + 2.
X = 3+2
yes

?- 3 + 2 = X.
X = 3+2
yes

?-
```

El predicado is/2

- Para forzar que Prolog realmente evalúe expresiones aritméticas, tenemos que usar el predicado del sistema

  is

  tal como se hizo en los ejemplos
- Es una instrucción a Prolog para que realice evaluación.
- Tiene algunas restricciones.

El predicado is/2

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

El predicado is/2

```
?- X is 3 + 2.
X = 5
yes

?-
```

El predicado is/2

```
?- X is 3 + 2.
X = 5
yes

?- 3 + 2 is X.
```

El predicado is/2

```
?- X is 3 + 2.
X = 5
yes

?- 3 + 2 is X.
ERROR: is/2: Arguments are not sufficiently instantiated

?-
```

El predicado is/2

```
?- X is 3 + 2.
X = 5
yes

?- 3 + 2 is X.
ERROR: is/2: Arguments are not sufficiently instantiated

?- Result is 2+2+2+2+2.
```

El predicado is/2

```
?- X is 3 + 2.
X = 5
yes

?- 3 + 2 is X.
ERROR: is/2: Arguments are not sufficiently instantiated

?- Result is 2+2+2+2+2.
Result = 10
yes

?-
```

Restricciones en el uso de is/2

- La evaluación se realiza sobre la expresión del lado derecho de ***is***
- Se pueden usar variables en el lado derecho de ***is***

  pero cuando la evaluación efectivamente se realiza las variables deben estar instanciadas en un valor numérico.
- El término Prolog del lado derecho debe ser una expresión aritmética bien formada.

Notación

- 3+2, 4/2, 4-5 son términos Prolog comunes en notación amigable: **3+2** es en realidad **+(3,2)**
- El predicado **is** es también un predicado Prolog común (predefinido).

Notación

– 3+2, 4/2, 4-5 son términos Prolog comunes en notación amigable: **3+2** es en realidad **+(3,2)**

– El predicado **is** es también un predicado Prolog común (predefinido).

```
?- is(4,2+2).
true
```

Otro predicado de evaluación aritmética

- Además de ***is***, hay otro predicado predefinido que realiza evaluación aritmética.
- Se escribe **=:=** , es infijo.
- Permite tener expresiones aritméticas a ambos lados del símbolo de predicado.
- Como ocurre con ***is***, si hay variables estas deben estar instanciadas al momento de la evaluación.

Otro predicado de evaluación aritmética

```
1 ?- 1+2 =:= 3.
true.

2 ?- 1+2 =:= X.
ERROR: =:=/2: Arguments are not
    sufficiently instantiated

3 ?- X =:= 1+2.
ERROR: =:=/2: Arguments are not
    sufficiently instantiated

4 ?- 2+3=:=4+1.
true.

5 ?- X=2,X+X=:=2*X.
X = 2.
```

Aritmética y listas

Cuál es el largo de una lista?

- La lista vacía tiene largo 0;
- El largo de una lista no vacía, L=[C|R], es el largo del resto R más 1.

Largo de una lista

```
largo([],0).
largo([_|L],N):-
    largo(L,X),
    N is X + 1.
```

```
?-
```

Largo de una lista

```
largo([],0).
largo([_|L],N):-
    largo(L,X),
    N is X + 1.
```

```
?- largo([a,b,c,d,e,[a,x],t],X).
```

Largo de una lista

```
largo([],0).
largo([_|L],N):-
    largo(L,X),
    N is X + 1.
```

```
?- largo([a,b,c,d,e,[a,x],t],X).
X=7
yes
?-
```

Acumuladores

- Existe otro modo de encontrar el largo de una lista
  - Trabajando con un acumulador
  - Un acumulador es una variable que almacena resultados intermedios
  - Se podría decir que permite hacer algo lo más parecido a un programa iterativo en Prolog

largoAc/3

El predicado largoAc/3 tiene los siguientes argumentos

- Una lista
- El largo de esa lista, un natural (típicamente una variable al invocarse)
- Un acumulador, mantiene los resultados intermedios

largoAc/3

El acumulador de largoAc:

- El valor inicial es 0.
- Se suma 1 cada vez que se procesa recursivamente el resto de la lista.
- Cuando se llega a la lista vacía, el acumulador contiene el largo de la lista.
- Se transfiere el valor del acumulador al *top level* mediante una variable que se mantiene incambiada en todos los pasos de recursión.

Largo de una lista con acumulador

```
largoAC([],Ac,Largo):-
    Largo = Ac.

largoAc([_|L],AcAnt,Largo):-
    Ac is AcAnt + 1,
    largoAc(L,Ac,Largo).
```

```
?-
```

Largo de una lista con acumulador

```
largoAC([],Ac,Largo):-
    Largo = Ac.

largoAc([_|L],AcAnt,Largo):-
    Ac is AcAnt + 1,
    largoAc(L,Ac,Largo).
```

suma 1 al acumulador cada vez que se procesa el resto

```
?-
```

Largo de una lista con acumulador

```
largoAC([],Ac,Largo):-
    Largo = Ac.

largoAc([_|L],AcAnt,Largo):-
    Ac is AcAnt + 1,
    largoAc(L,Ac,Largo).
```

Al llegar a la lista vacía el acumulador contiene el largo de la lista

```
?-
```

Largo de una lista con acumulador

```
largoAC([],Ac,Ac).


largoAc([_|L],AcAnt,Largo):-
    Ac is AcAnt + 1,
    largoAc(L,Ac,Largo).
```

Una mejora, la igualdad se resuelve por unificación

```
?-
```

Largo de una lista con acumulador

```
largoAC([],Ac,Ac).

largoAc([_|L],AcAnt,Largo):-
    Ac is AcAnt + 1,
    largoAc(L,Ac,Largo).
```

Invocación:
largoAc(L,0,Largo),
Inicialmente el acumulador es 0

```
?-
```

Largo de una lista con acumulador

```
largoAC([],Ac,Ac).

largoAc([_|L],AcAnt,Largo):-
    Ac is AcAnt + 1,
    largoAc(L,Ac,Largo).
```

```
?-
```

Largo de una lista con acumulador

```
largoAC([],Ac,Ac).

largoAc([_|L],AcAnt,Largo):-
    Ac is AcAnt + 1,
    largoAc(L,Ac,Largo).
```

```
?-largoAc([a,b,c],0,Largo).
Largo=3
yes

?-
```

Arbol de búsqueda para largoAc

?- largoAc([a,b,c],0,Largo).

```
largoAc([ ],Ac,Ac).

largoAc([_|L],AcAnt,Largo):-
    Ac is AcAnt + 1,
    largoAc(L,Ac,Largo).
```

Arbol de búsqueda para largoAc

?- largoAc([a,b,c],0,Lar).

/ \

```
largoAc([ ],Ac,Ac).

largoAc([_|L],AcAnt,Largo):-
    Ac is AcAnt + 1,
    largoAc(L,Ac,Largo).
```

Arbol de búsqueda para largoAc

?- largoAc([a,b,c],0,Lar).

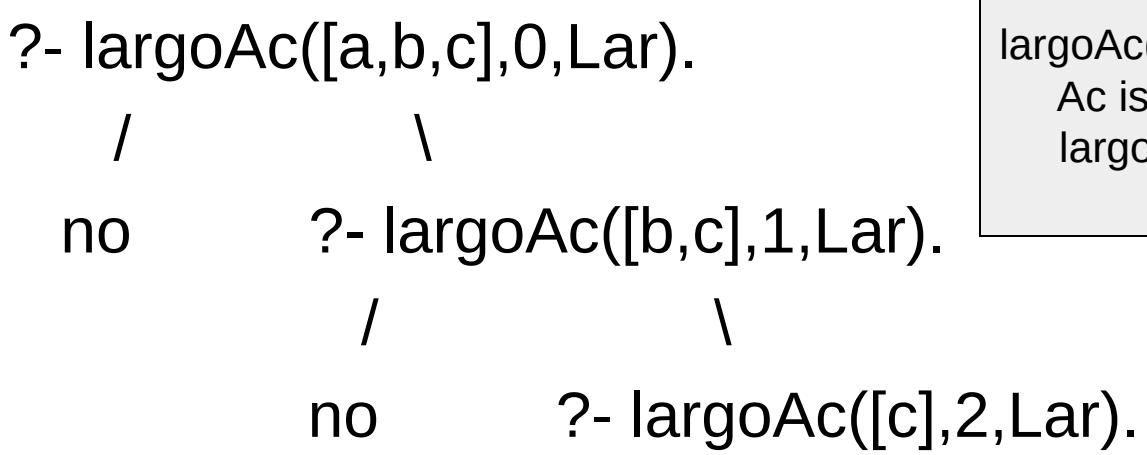
/ \

no ?- largoAc([b,c],1,Lar).

```
largoAc([ ],Ac,Ac).

largoAc([_|L],AcAnt,Largo):-
    Ac is AcAnt + 1,
    largoAc(L,Ac,Largo).
```

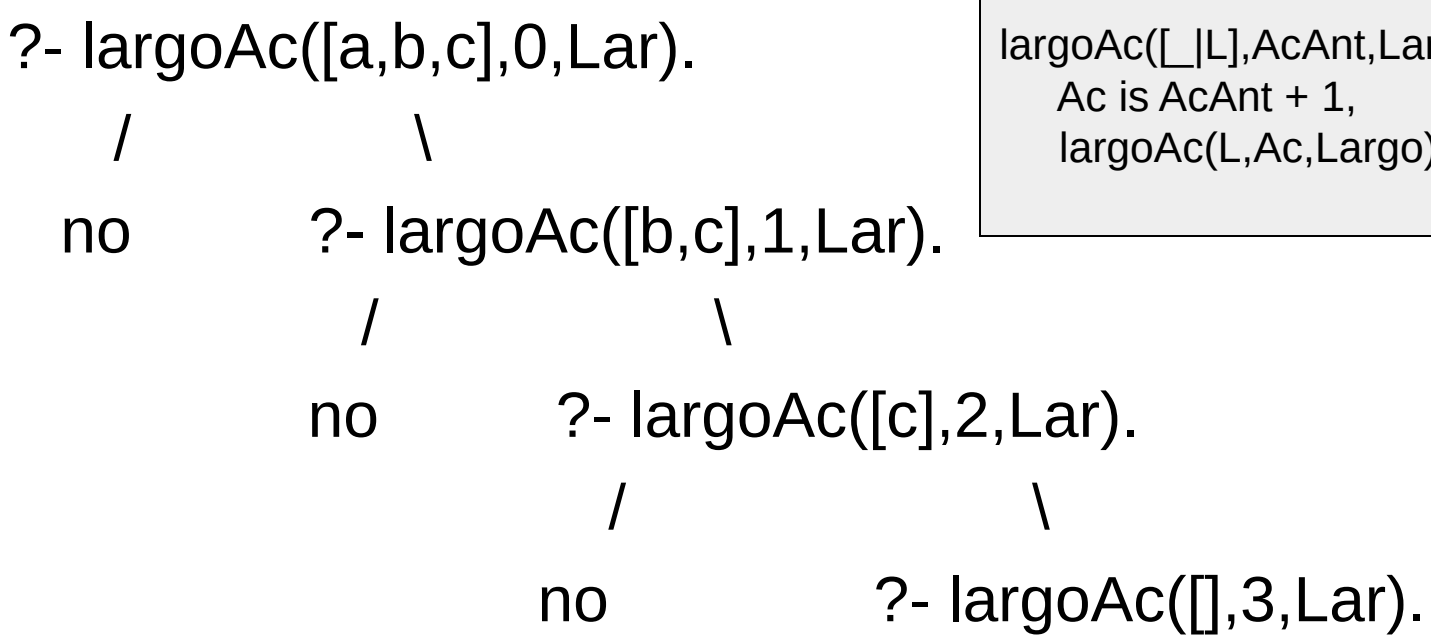
Arbol de búsqueda para largoAc

```
largoAc([ ],Ac,Ac).

largoAc([_|L],AcAnt,Largo):-
    Ac is AcAnt + 1,
    largoAc(L,Ac,Largo).
```



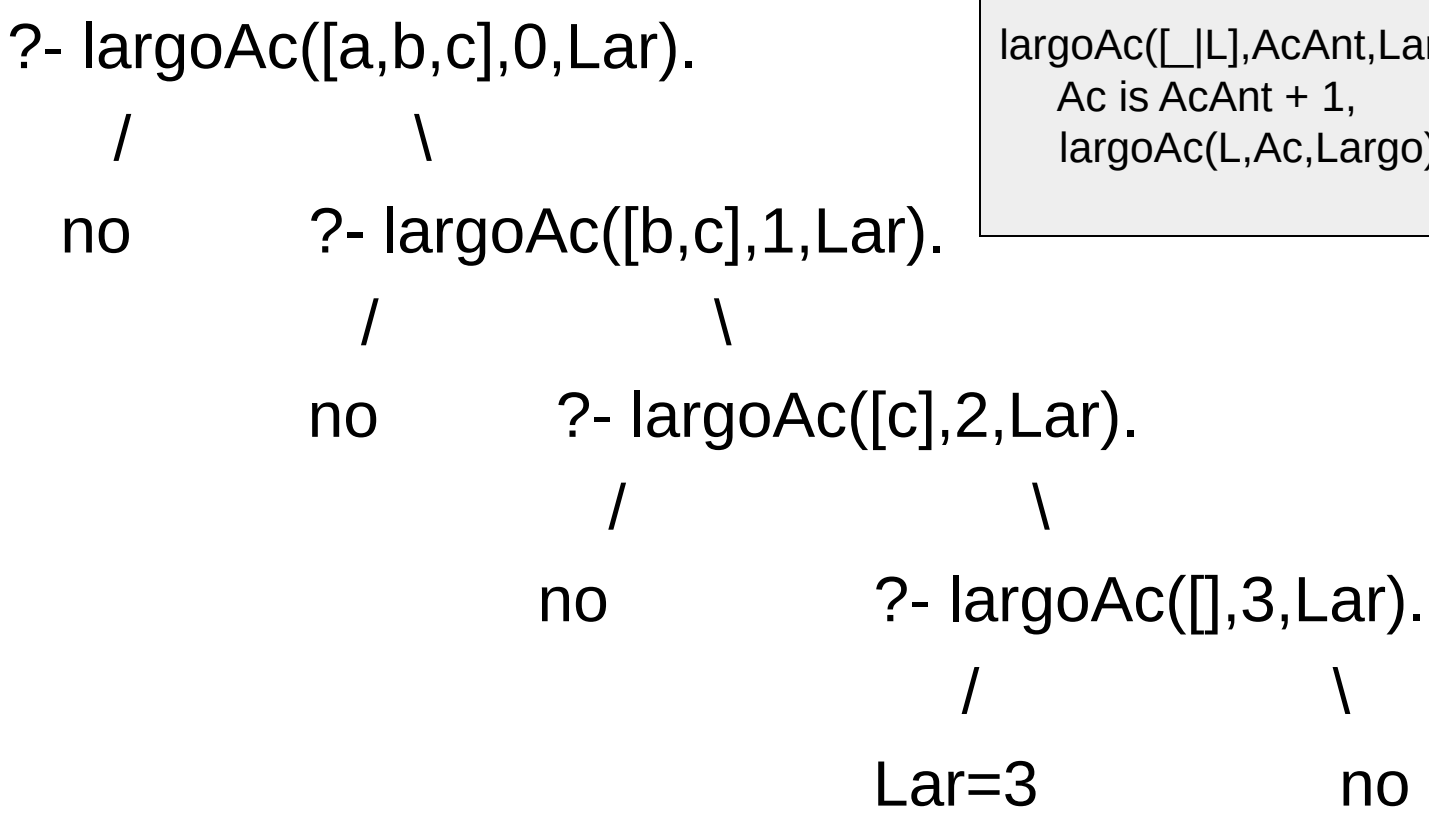
Arbol de búsqueda para largoAc

```
largoAc([ ],Ac,Ac).

largoAc([_|L],AcAnt,Largo):-
    Ac is AcAnt + 1,
    largoAc(L,Ac,Largo).
```



Arbol de búsqueda para largoAc

```
largoAc([ ],Ac,Ac).

largoAc([_|L],AcAnt,Largo):-
    Ac is AcAnt + 1,
    largoAc(L,Ac,Largo).
```



Predicado *wrapper* para la invocación

```
largoAC([],Ac,Ac).

largoAc([_|L],AcAnt,Largo):-
    Ac is AcAnt + 1,
    largoAc(L,Ac,Largo).

largo(Lista,Largo):-
     largoAc(Lista,0,Largo).
```

```
?-largo([a,b,c], X).
X=3
yes
```

Tail recursion

Porqué es mejor largoAc/3 que largo/2 ?

- largoAc/3 es *tail-recursive*, y largo/2 no

Diferencia:

- En predicados *tail-recursive* los resultados se calculan en el paso base.
- En los otros predicados recursivos, hay aún objetivos en el stack de ejecución al llegar al paso base.
- Mayor tamaño del stack de ejecución.
- El sistema optimiza.
- Debe haber determinismo en los predicados previos.

Comparación

No tail-recursive

```
largo([],0).

largo([_|L],N):-
    largo(L,X),
    N is X + 1.
```

Tail-recursive

```
largoAC([],Ac,Ac).

largoAc([_|L],AcAnt,Largo):-
     Ac is AcAnt + 1,
     largoAc(L,Ac,Largo).
```

Arbol de búsqueda para largo/2

?- largo([a,b,c], L).

```
largo([],0).

largo([_|L],N):-
    largo(L,X),
    N is X + 1.
```

Arbol de búsqueda para largo/2

?-largo([a,b,c], L).

/ \

no ?- largo([b,c],L1),
L is L1 + 1.

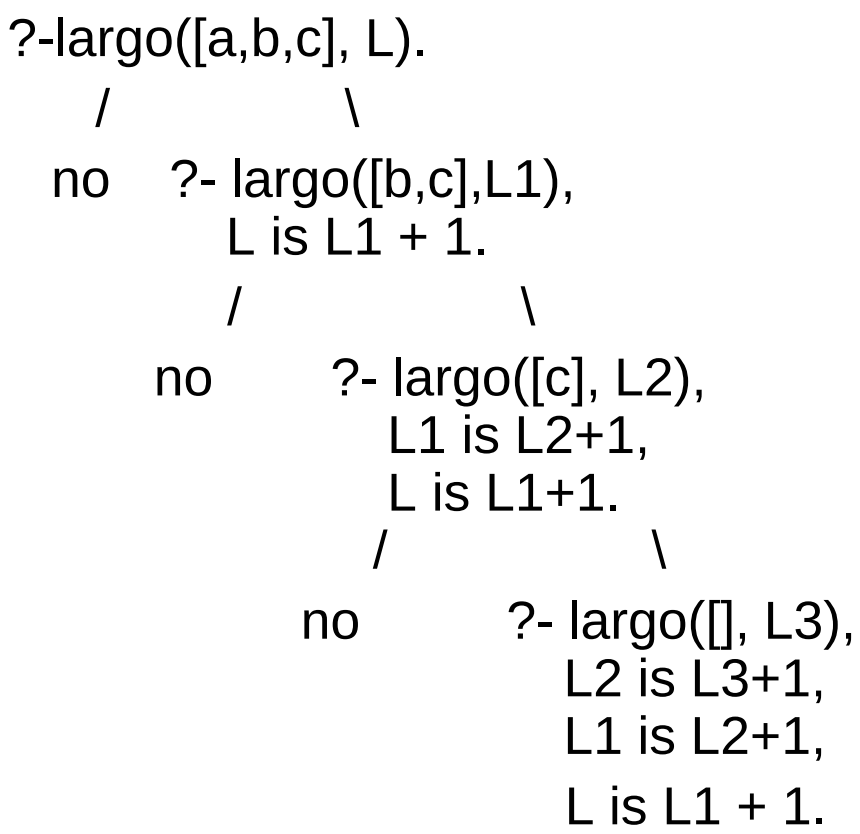
```
largo([],0).

largo([_|L],N):-
    largo(L,X),
    N is X + 1.
```

Arbol de búsqueda para largo/2

```
?-largo([a,b,c], L).
      /              \
   no    ?- largo([b,c],L1),
              L is L1 + 1.
            /              \
         no       ?- largo([c], L2),
                      L1 is L2+1,
                      L is L1+1.
```

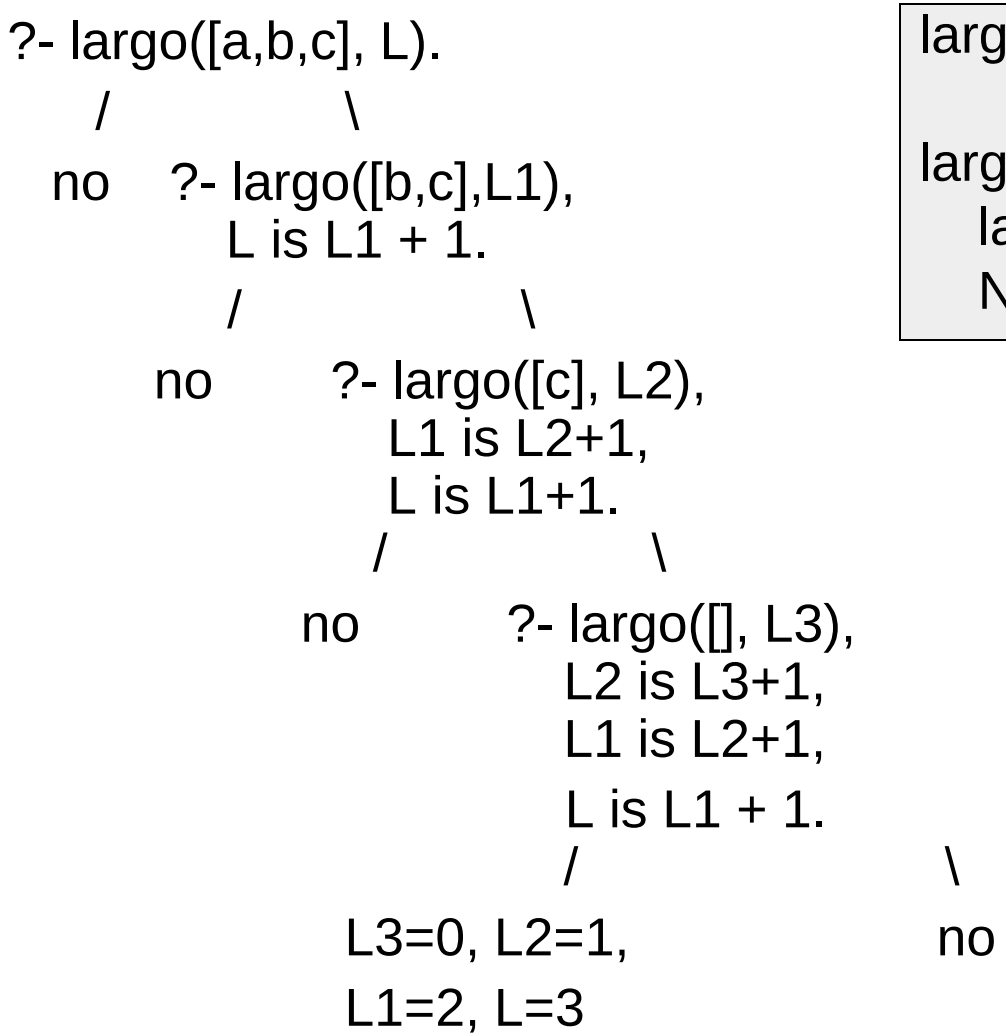
```
largo([],0).

largo([_|L],N):-
    largo(L,X),
    N is X + 1.
```

Arbol de búsqueda para largo/2




```
largo([],0).

largo([_|L],N):-
    largo(L,X),
    N is X + 1.
```

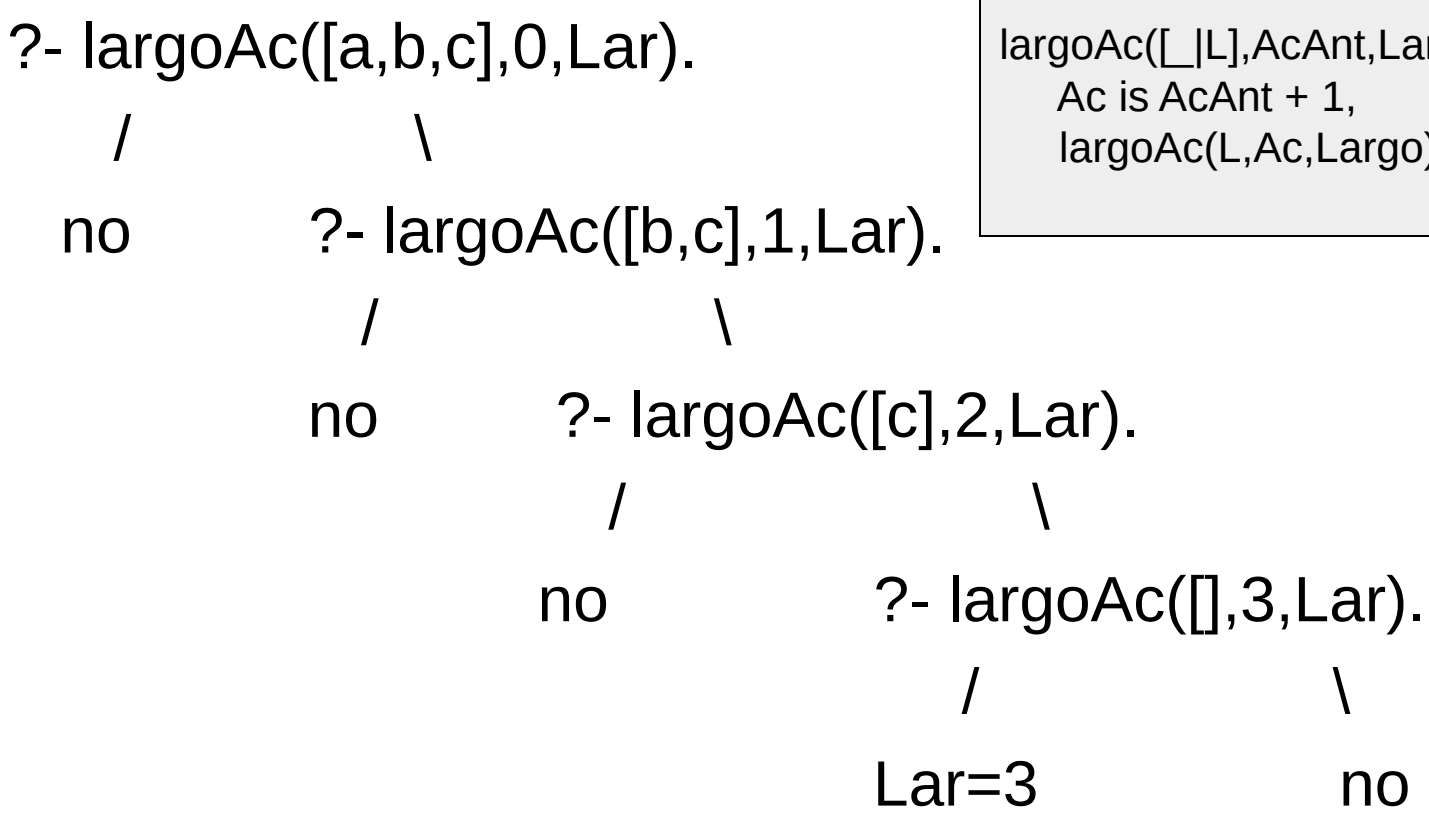
Arbol de búsqueda para largo/2




```
largo([],0).

largo([_|L],N):-
    largo(L,X),
    N is X + 1.
```

Arbol de búsqueda para largoAc

```
largoAc([ ],Ac,Ac).

largoAc([_|L],AcAnt,Largo):-
    Ac is AcAnt + 1,
    largoAc(L,Ac,Largo).
```



Más predefinidos: comparación

- Comparación entre números
- Realizan cálculos si hay expresiones artiméticas.
- Ya vimos uno de estos predicados

=:=

Comparación de números

Artmética	Prolog
$x < y$	X < Y
$x \leq y$	X =< Y
$x = y$	X =:= Y
$x \neq y$	X =\= Y
$x \geq y$	X >= Y
$x > y$	X > Y

Comparación de números

Artmética	Prolog
$x < y$	X < Y
$x \leq y$	X =< Y
$x = y$	X =:= Y
$x \neq y$	X =\= Y
$x \geq y$	X >= Y
$x > y$	X > Y

Nota: En Prolog	?- X=4.	?- X \= 4.
= es unifican	Yes	No
\= es no unifican		

Operadores de comparación

- Significado obvio.
- Evalúan ambos argumentos (lado izquierdo y lado derecho.
- En este caso sí hay restricción de tipo

```
?- 2 < 4+1.
yes

?- 4+3 > 5+5.
no
```

Operadores de comparación

- Significado obvio.
- Evalúan ambos argumentos (lado izquierdo y lado derecho.
- En este caso sí ha restricción de tipo

```
?- 4 = 4.
yes

?- 2+2 = 4.
no

?- 2+2 =:= 4.
yes
```

Predicado maxLista

- Definiremos un predicado con 2 argumentos:
  - El 1er argumento es una lista de números
  - El 2do argumentos es el elemento máximo de la lista

- Documentación mínima de un predicado, se describe la propiedad que el predicado hace verdadera:

% maxLista(Lista,Max) ← Max es el máximo de la lista de números Lista.

% a comienzo de línea – indica que la línea es comentario

Predicado maxLista

Definiremos un predicado con 2 argumentos:

- El 1er argumento es una lista de números
- El 2do argumentos es el elemento máximo de la lista

Idea básica

- Usar un acumulador
- El acumulador registra el mayor valor encontrado hasta el momento
- Si se encuentra un valor mayor, se actualiza el acumulador

Definition of accMax/3

```
% accMax(Lista,Actual,Max) ← Max es el máximo de la lista de números Lista,
% y Actual es el máximo de los elementos ya considerados.
accMax([H|T],A,Max):-
    H > A,
    accMax(T,H,Max).

accMax([H|T],A,Max):-
    H =< A,
    accMax(T,A,Max).

accMax([],A,A).
```

```
?- accMax([1,0,5,4],0,Max).
Max=5
yes
```

maxLista, *wrapper* de accMax

```
accMax([H|T],A,Max):-
    H > A,
    accMax(T,H,Max).

accMax([H|T],A,Max):-
    H =< A,
    accMax(T,A,Max).

accMax([],A,A).

maxLista([H|T],Max):-
    accMax(T,H,Max).
```

```
?- maxLista([1,0,5,4], Max).
Max=5
yes

?- maxLista([-3, -1, -5, -4],
    Max).
Max= -1
yes

?-
```

maxLista, *wrapper* de accMax

```
accMax([H|T],A,Max):-
    H > A,
    accMax(T,H,Max).

accMax([H|T],A,Max):-
    H =< A,
    accMax(T,A,Max).

accMax([],A,A).

maxLista([H|T],Max):-
    accMax(T,H,Max).
```

```
?- maxLista([1,0,5,4], 5).
yes

?- maxLista([-3, -1, -5, -4], 2).
no

?- maxLista(X,4).
ERROR: >/2: Arguments are
    not sufficiently instantiated
  Exception: (8)
    accMax(_G331, _G330,
    4) ?
```

Funcionamiento de maxLista

```
?- maxLista([1,0,5,4], 5).
Yes

?- maxLista([1,0,5,4], Max).
Max=5
Yes

?- maxLista(X,4).
ERROR: >/2: Arguments are not sufficiently instantiated
   Exception: (8) accMax(_G331, _G330, 4) ?
```

Funcionamiento de maxLista

```
?- maxLista([1,0,5,4], 5).
Yes

?- maxLista([1,0,5,4], Max).
Max=5
Yes

?- maxLista(X,4).
ERROR: >/2: Arguments are not sufficiently instantiated
   Exception: (8) accMax(_G331, _G330, 4) ?
```

El primer argumento debe estar instanciado.

El segundo puede estar o no instanciado.

Se documenta: maxLista(+Lista,?Max) (Es documentación!!)

Patrones de instanciación

- Son las formas posibles de instanciar los argumentos de los predicados de modo de obtener un funcionamiento correcto.
- Los patrones de instanciación se describen en la documentación.
- Cada argumento está precedido por uno de los símbolos {+,-,?}
- Ej. maxLista(+Lista,?Max)

Patrones de instanciación

- Son las formas posibles de instanciar los argumentos de los predicados en invocación, de modo de obtener un funcionamiento correcto.
- Los patrones de instanciación se describen en la documentación.
- Cada argumento está precedido por uno de los símbolos {+,-,?}
- Ej. maxLista(+Lista,?Max)

+	El argumento debe estar instanciado
-	El argumento no debe estar instanciado
?	El argumento puede o no estar instanciado

Patrones de instanciación

- Son las formas posibles de instanciar los argumentos de los predicados en invocación, de modo de obtener un funcionamiento correcto.
- Los patrones de instanciación se describen en la documentación.
- Cada argumento está precedido por uno de los símbolos {+,-,?}
- Ej. maxLista(+Lista,?Max)

+	El argumento debe estar instanciado
-	El argumento no debe estar instanciado
?	El argumento puede o no estar instanciado

Puede haber más de un patrón para el mismo predicado.

Resumen

- Hemos mostrado como hacer **aritmética** en Prolog
- Hemos mostrado la diferencia entre predicados ***tail-recursive*** y predicados que no son *tail-recursive*
- Introducimos la técnica de programación de **acumuladores**
- Introducimos la idea de usar predicados ***wrapper***
- Documentación: introducimos **patrones de instanciación** de los argumentos

Ejercicios

5.1 l.p.n. Qué responde Prolog ante:

?- X=3*4.

?- X is 3*4.

?- 4 is X.

?- X = +(3,2).

?- X is +(3,2).

?- 3+2 =:= 3+2.

?- 3+2 = +(3,2).

Etc.

Ejercicios

5.2 l.p.n.

a. Defina un predicado Prolog masUno(X,Y)

masUno(X,Y) ← Y = X+1

b. Documente la instanciación de argumentos válida.

5.3 l.p.n.

a. Defina un predicado Prolog masUnoLista(L1,L2)

masUnoLista(L1,L2) ← cada elemento de la lista L2 es el elemento correspondiente de la lista L1 más uno.

b. Documente la instanciación de argumentos válida.

Ejercicios

Defina un predicado **suma(X,Y,Z)** usando predefinidos Prolog, tal que:

suma(?X,?Y,+Z) ← X y Y son dos naturales que sumados dan Z

P.ej., suma(X,Y,3) da los siguientes resultados:

X=0, Y=3

X=1, Y=2

X=2, Y=1

X=3, Y=0

Próxima

Más listas!

- Definición de append/3, predicado para concatenar listas
- Predicados definibles en base a append/3
- Reverso de una lista, modo *naif* usando append/3, modo más eficiente usando acumuladores

Más patrones de instanciación!