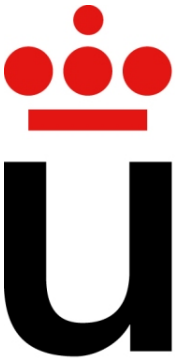


Universidad Rey Juan Carlos

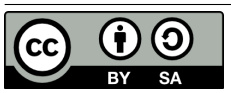
ESCUELA TÉCNICA SUPERIOR
DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA

Apuntes de Programación Lógica

HASTA PROLOG Y MÁS ALLÁ

Autor: Joaquín Arias



Índice general

Agradecimientos

Esta obra, contiene en formato de apuntes las transparencias de Arias, Joaquín (2024). Programación Lógica: hasta Prolog y más allá. BURJC, Madrid, las cuales están basada en transparencias de Ana Pradera (URJC'23) & ClipLab (UPM'24).

Universidad
Rey Juan Carlos

Cápitulo 0

Programación Lógica vs. Funcional

0.1. Programación Declarativa.

0.1.1. Contexto

- Qué es la programación declarativa?
  - Es un paradigma de programación diferente a la imperativa (**R) o a la orienta a objetos (Java).**

- **Los programas especifican las propiedades de los problemas a resolver.**
- **La ejecución de un programa consiste en "encontrar" las solucione(s).**

- Ejemplos de lenguajes de programación declarativos:
  - Algebraicos: Maude, SQL.
  - Lógicos: Prolog, ASP, Logica by Google.
  - Funcionales: Haskell, Scala by EPFL.

Asignación Destructiva vs. Recursión

- Ejemplo de implementación usando asignación destructiva.

```
# Sum list of numbers using R
sumaLista <- function(list) {
  sum <- 0
  for (n in list)
    sum <- sum + n
  return(sum) }
# Print the results of the sum, 10
print(sumaLista(list(1,2,3,4)))
```

- Ejemplo usando recursión.

```
-- Sum list of numbers using Haskell
sumaLista :: [Int] -> Int
sumaLista [] = 0
sumaLista (n : list) = n + (sumaLista list)


# Print the results of the sum, 10
main = print (sumaLista [1,2,3,4])
```

A continuación compararemos Haskell y Prolog

0.1.2. Programación Funcional

Programación Funcional: Introducción

- La programación funcional está basada en funciones matemáticas.
- Función: Una función es una regla de correspondencia entre dos conjuntos de modo que a cada elemento del primer conjunto le corresponde un y solo un elemento del segundo conjunto.
- Cualquier función computable puede expresarse y evaluarse con el Lambda Calculus.
- Church usó Lambda Calculus para resolver el Entscheidungsproblem (1936):

- No existe ningún algoritmo que determine si dos expresiones lambda arbitrarias son equivalentes.

Programación Funcional: Lambda Calculus

- Introducción al Lambda Calculus.
- Reglas de formación de las expresiones lambda (λ-expressions):
  - `x` es una λ-expression si `x` es una variable.
  - `(λx.t)` es una λ-expression (function) si `t` es una expresión y `x` una variable.
  - `(t s)` es una λ-expression (aplicación) si `t` y `s` son expresiones.

Evaluando λ-expressions

Función identidad aplicada a 3: `((λx.x) 3)` $\equiv$ `3`
Función suma aplicada a 2 y 3: `(((λx.λy.x+y) 2) 3)` $\equiv$ `((λy.2+y) 3)` $\equiv$ `(2+3)`
Función identidad aplicada a suma: `((λx.x) (λx.λy.x+y))` $\equiv$ `(λx.λy.x+y)`

Programación Funcional: Haskell

- Debe su nombre a Haskell Curry (1900-1982).
- Dada una función f de tipo $f : (X_1 \times X_2 \times \cdots \times X_n) \to Z$ decimos que su currificación es:
  - Una secuencia de funciones con un único argumento:

$$curry(f) : X_1 \to X_2 \to \cdots \to X_n \to Z.$$


```
sum :: Int -> Int -> Int                -- type declaration of sum
sum a b = a + b                         -- implementation with two arguments
curry_sum = \a -> \b -> a + b           -- currying version
successor :: Int -> Int                 -- Successor of a
successor = sum 1                       -- partial application
map :: (Int -> Int) -> [Int] -> [Int]   -- Take a function and a list
map _ [] = []
map f (n : list) = ((f n) : (map f list))

main = print (map successor [1,3,4])    -- print [2,4,5]
```

Programación Funcional: Booleanos en Lambda Calculus

- Primero implementamos las expresiones If-then-else, True and False:

- If-then-else: $\lambda x.\lambda y.\lambda z.x\ y\ z$
- true: $\lambda x.\lambda y.x$
- false: $\lambda x.\lambda y.y$

If-then-else True P Q $\equiv$
$\equiv (\lambda x.\lambda y.\lambda z.\ x\ y\ z)\ (\lambda x.\lambda y.x)\ P\ Q \equiv (\lambda x.\lambda y.x)\ P\ Q \equiv$ P

- Implementación usando Haskell:

```
1 if_then_else = \x -> \y -> \z -> x y z
2 true = \x -> \y -> x
3 false = \x -> \y -> y
4
5 k = if_then_else true 3 2          -- What is the value of k?
```

Programación Funcional: Booleanos en Lambda Calculus (cont.)

- Después, basado en estas expresiones definimos And, Or and Not:
  - And: $\lambda p.\lambda q.p\ q\ false \equiv \lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y)$
  - Or: $\lambda p.\lambda q.p\ true\ q \equiv \lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q$
  - Not: $\lambda p.p\ false\ true \equiv \lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x)$

And True False $\equiv (\lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y))\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv$
$(\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2)\ (\lambda x.\lambda y.y) \equiv (\lambda x_2.\lambda y_2.y_2) \equiv$ False
Or True False $\equiv (\lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q)\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv$
$(\lambda x_1.\lambda y_1.x_1)\ (\lambda x.\lambda y.x)\ (\lambda x_2.\lambda y_2.y_2) \equiv (\lambda x.\lambda y.x) \equiv$ True
Not True $\equiv$
$\equiv (\lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x))\ (\lambda x_1.\lambda y_1.x_1) \equiv \ldots \equiv (\lambda x.\lambda y.y) \equiv$ False

- Implementación usando Haskell (cont.):

```
5 my_and = \x -> \y -> x y false
6 my_or = \x -> \y -> x true y
7 my_not = \x -> x false true
8
9 k = if_then_else (my_and true false) 3 2          -- Value of k?
```

Programación Funcional: Booleanos en Lambda Calculus (cont.)

- Aunque podríamos considerar otras expresiones para And, Or and Not:
  - And$_2$: $\lambda p.\lambda q.p\ q\ p$
  - Or$_2$: $\lambda p.\lambda q.p\ p\ q$
  - Not$_2$: $\lambda p.\lambda x.\lambda y.p\ y\ x$ ¿Cuántos argumentos tiene?

Deberes

- Implementación (se requieren tipos) utilizando Haskell (cont.):

```haskell
{-# LANGUAGE Rank2Types #-}
type CB = forall a . a -> a -> a
my_and :: CB -> CB -> CB
my_and = \p -> \q -> p q p
my_or :: CB -> CB -> CB
my_or = \p -> \q -> p p q
my_not :: CB -> CB                    -- seems to have 1 argument
my_not = \p -> \x -> \y -> p y x

k = if_then_else (my_not false) 3 2   -- What is the value of k?
```

0.1.2.1. Características y Ventajas

Programación Funcional: Características y Ventajas

Características:

- Evaluación de funciones vs. ejecución de instrucciones (recursión vs. iteración).
- El valor de una función sólo depende de sus argumentos (siempre se obtiene el mismo valor, transparencia referencial).
- Las funciones son "ciudadanos de primera clase" (argumentos y/o valores)

Ventajas:

- Código más limpio, conciso y expresivo.
- Sin efectos secundarios, al ser el estado inmutable.
  - Adecuado para sistemas concurrentes/paralelos.
- Permite verificación formal y demostración automática.

Concatenar listas

```haskell
concatenar :: [a] -> [a] -> [a]                  % declaración de tipos
concatenar [] list = list                        % caso base
concatenar (x:xs) list = (x: (concatenar xs list))  % llamada recursiva

k = concatenar [1,2] [3,4]                       % ¿cuánto vale k?
```

0.1.3. Programación Lógica

Programación Lógica: Introducción

- La programación lógica esta basada en lógica de 1^{er} orden (LPO).
- Predicados: Un predicado es una afirmación sobre propiedades de un objeto y/o una relación entre dos o más objetos.
- Dado un conjunto de fórmulas inferimos nuevo conocimiento. P.ej.:

$$T[\forall x\ (\ Hombre(x) \rightarrow Mortal(x)\ ), Hombre(socrates)] \vdash Mortal(socrates)$$

1 Se reescribe como el siguientes conjunto de cláusulas:

$$\{\ Mortal(x) \vee \neg Hombre(x),\ Hombre(socrates),\ \neg Mortal(socrates)\ \}$$

donde el consecuente, $Mortal(socrates)$, están negado.

2 Si es insatisfacible, significa que hay consecuencia lógica.

3 Se resuelve aplicando método de Robinson con estrategia SLD.

0.1.3.1. Cláusulas Horn

Programación Lógica: Cláusulas de Horn

- Introducción a las cláusulas de Horn, definidas por Alfred Horn en 1951.
- Dada una cláusula (disyunción de literales) cualquiera $L_1 \vee L_2 \vee \cdots \vee L_n$, es una cláusula de Horn si tiene como máximo un literal positivo y esta reescrita como una implicación. Por ejemplo:

$\neg p \vee \neg q \vee \cdots \vee \neg t \vee u$	es una regla y se reescribe como	$p \wedge q \wedge \cdots \wedge t \rightarrow u$
u	es un hecho y se reescribe como	u
$\neg p \vee \neg q \vee \cdots \vee \neg t$	sin literal positivo, es una consulta	$p \wedge q \wedge \cdots \wedge t \rightarrow$

Aristóteles:	**Cláusulas:**	**Prolog:**
$Hombre(x) \rightarrow Mortal(x)$	$Mortal(x) \vee \neg Hombre(x)$	`mortal(X) :- hombre(X).`
$Hombre(socrates)$	$Hombre(socrates)$	`hombre(socrates).`
$Mortal(socrates)$	$\neg Mortal(socrates)$	`?- mortal(socrates).`

0.1.3.2. Prolog

Programación Lógica: Prolog.

- *Predicados*: Transforma $f : (X_1 \times X_2 \times \cdots \times X_n) \rightarrow Z$
  en una relación (n+1)-aria R y define el predicado r tal que:
  $r(x_1, x_2, \ldots, z_n, z) = true \longleftrightarrow (x_1, x_2, \ldots, z_n, z) \in R$.

SWI-Prolog

Ciao Prolog

Aristóteles

```
1 mortal(X) :- hombre(X).          % Todos los hombres son mortales.
2 hombre(socrates).                % Sócrates es un hombre.
3
4 ?- mortal(socrates).             % ¿Sócrates es mortal?
```

Contesta `yes` si la "pregunta" es consecuencia lógica (`no` en caso contrario).

Concatenar listas

```
1 concatenar([],Lista,Lista).
2 concatenar([X|Xs],Lista,[X|N_Lista]) :- concatenar(Xs,Lista,N_Lista).
3
4 ?- concatenar([1,2],[3,4],Lista).          % ¿Cuánto vale Lista?
```

... devuelve la(s) sustitución(es) que la hace(n) consistente: `Lista = [1,2,3,4] ?`

0.1.3.3. Hay algo más ...

Programación Lógica: Hay algo más ...

- Mientras las funciones devuelven un único resultado:
  `k = concatenar [1,2] [3,4]`
- Los predicados pueden "consultarse" de diferentes formas sin cambiar el programa:
  `?- concatenar([1,2], L, [1,2,3,4]).`
  devuelve:
  `L = [3,4] ?`

- Pero, hay algo más ...

  ```
  ?- concatenar(LA, LB, [1,2,3,4]).
  ```

  devuelve 5 respuestas:

  1. `LA = [], LB = [1,2,3,4] ?;`
  2. `LA = [1], LB = [2,3,4] ?;`
  3. `LA = [1,2], LB = [3,4] ?;`
  4. `LA = [1,2,3], LB = [4] ?;`
  5. `LA = [1,2,3,4], LB = [] ?`

Esto permite usar un único predicado para codificar/decodificar mensajes

```
char2morse('A','.-').   char2morse('B','-...').   char2morse('C','-.-.').
...

?- char2morse('B', Morse).                    % devuelve Morse = '-...'
?- char2morse(Char, '-.-.').                  % devuelve Char = 'C'
```

0.1.3.4. ... mucho más

Programación Lógica: ... mucho más (CLP)

- Constraint Logic Programming (CLP): Incorpora restricciones que nos permite expresar relaciones entre variables mediante ecuaciones:

$$\begin{cases} 3x+5y=2 \\ 5x+3y=-2 \end{cases}$$

```
sol(X,Y) :-
    3 * X + 5 * Y #= 2,
    5 * X + 3 * Y #= -2.
```

```
?- sol(X,Y).
   X = -1,
   Y = 1 ?
```

CLP(Q) nos permite definir la relación de una hipoteca como:

```
mg(P,T,_,_,B) :- T #= 0, B #= P.
mg(P,T,R,I,B) :- T #>= 1, NP #= P + P*I - R, NT #= T - 1, mg(NP,NT,R,I,B).
```

P=principal, T=time periods, R=repayment each period, I=interest rate, B=balance owing.

Podemos consultar de diferentes formas: ... y mucho más.

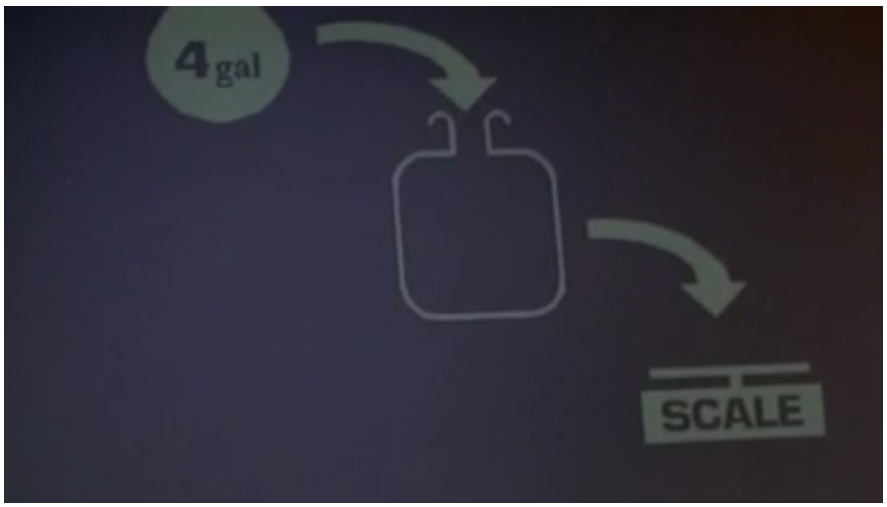
```
?- mg(1000,10,150,0.10,B).   ?- mg(P,10,150,0.10,0).    ?- mg(P,10,R,0.10,B).
  B = 203.13 ?                  P = 921.68 ?                 P = 6.14*R + 0.38*B ?
```

0.2. Algunos problemas interesantes.

0.2.1. El problemas de las garrafas

Ejemplo I: El problemas de las garrafas

- Escenario: tenemos dos garrafas, una de 5 y otra de 3 galones.
- Objetivo: Depositar 4 galones exactos para que la bomba no explote.



Link

0.2.2. El problema del puente y la linterna

Ejemplo II: El problema del puente y la linterna

- Escenario: Una familia tiene que cruzar un puente frágil y sin iluminar. La hija tarda 1 min, el padre 2 min, la madre embarazada 5 min y el abuelo 8 min.
- Objetivo: Que todos crucen (max 2 por viaje y usando la linterna), antes de que se caiga (en 15 min).

Link

0.2.3. Las Torres de Hanoi

Ejemplo III: Las Torres de Hanoi

- Escenario: Tenemos tres postes (A,B y C) y 4 discos distintos.
- Objetivo: mover los discos del poste A al poste C.
- Reglas:
  1. Se mueve un disco cada vez.
  2. Un disco no puede estar sobre uno más pequeño que él.
  3. Solo se puede mover el disco que se encuentre arriba en cada poste.

0.2.4. El problema del viajante

Ejemplo IV: El problema del viajante

- Formulado en 1930: Dada una lista de ciudades y las distancias entre cada par de ellas, ¿cuál es la ruta más corta posible que visita cada ciudad exactamente una vez y al finalizar regresa a la ciudad origen?
- El origen de este problema lo encontramos en el juego "Icosian" (planteado por Hamilton en 1857).

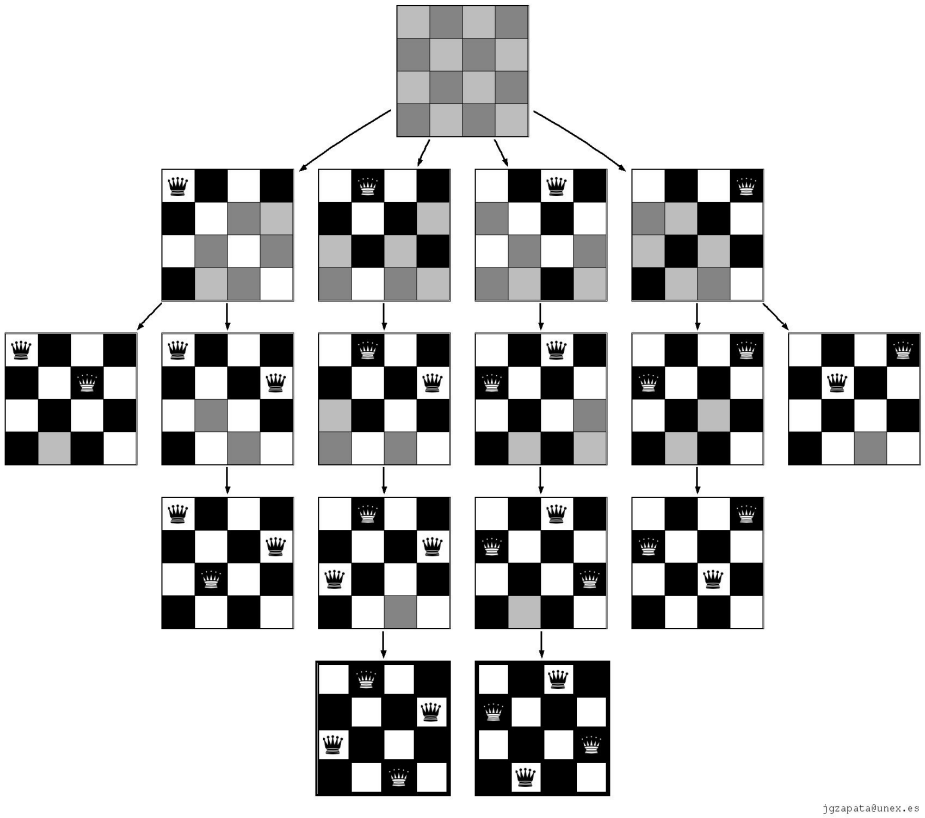
- Su resolución está basada en la **búsqueda** de ciclos hamiltonianos: Dado un grafo, un ciclo hamiltoniano, es un ciclo que pasa una y solo una vez por todos los vértices del grafo.
- El problema del Ciclo Hamiltoniano (búsqueda) es NP-completo y el problema del Viajante (optimización) es NP-duro.
- **Aplicaciones:** Transporte y logística, fabricación circuitos impresos, planificación, etc.

0.2.5. El problema de las n-reinas

Ejemplo V: El problema de las n-reinas

- Fue propuesto por el ajedrecista alemán Max Bezzel en 1848 para 8 reinas: El juego de las 8 reinas consiste en poner sobre un tablero de ajedrez (8x8) ocho reinas sin que estas se amenacen entre ellas.
- En 1972, Edsger Dijkstra usó este problema para describir el algoritmo de backtracking, "depth-first".

- Espacio de búsqueda (y soluciones) para $n = 4$:



0.2.6. Zebra puzzle (Einstein riddle)

Ejemplo VI: Zebra puzzle (Einstein riddle)

1. Hay cinco casas.
2. El inglés vive en la casa roja.
3. El español es el dueño del perro.
4. El café se bebe en la casa verde.
5. El ucraniano bebe té.
6. La casa verde está inmediatamente a la derecha de la casa de blanco.
7. El fumador de OldGold tiene caracoles.
8. En la casa amarilla se fuma Kools.
9. La leche se bebe en la casa del medio.
10. El noruego vive en la primera casa.
11. El hombre que fuma Chesterfields vive en la casa de al lado del hombre con el zorro.

12. El que fuma Kools vive en la casa contigua a la del caballo.
13. El fumador de Lucky Strike bebe zumo.
14. El japonés fuma Parliaments.
15. El noruego vive junto a la casa azul.

Pregunta, ¿quién bebe agua? ¿De quién es la cebra?

- Modelizacion: `h(Nationality, Pet, Cigarette, Drink, Color)` representa cada casa.

Universidad
Rey Juan Carlos

Cápitulo 1

El paradigma de la programación lógica

1.1. Lógica de 1^{er} orden.

1.1.1. ¿Por qué es importante?

Lógica: ¿Por qué es importante?

- La lógica formal es la ciencia que estudia las leyes de inferencia en los razonamientos. Repasar apuntes de Lógica (Arias 2022).
- Trata de resolver diversos problemas basándose en la formación del lenguaje y sus reglas básicas.
- Se aplica en multitud de áreas:
  - En matemáticas para demostrar teoremas
  - En ciencias de la computación para verificar si son o no correctos los programas
  - En las ciencias física y naturales, para sacar conclusiones de experimentos
  - En las ciencias sociales y en la vida cotidiana, para resolver una multitud de problemas.

1.1.2. Tipos de lógicas

Lógica: Tipos

- Debemos conocer al menos dos tipos de lógica

  Lógica proposicional
  - "Sócrates es un hombre" $h.$
  - "Sócrates es mortal" $m.$
  - "Sócrates es un hombre y es mortal" $h \wedge m$

  Lógica de primer orden
  - "x es un hombre" $H(x).$
  - "x es mortal" $M(x).$
  - "Todos los hombres son mortales" $\forall x.H(x) \rightarrow M(x)$

- Sin embargo existen otras lógicas:
  De 2^o orden, no monótona, epistémica, deóntica, temporal, lineal, etc.

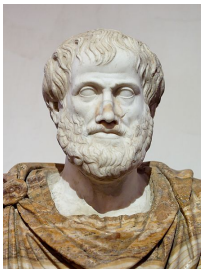
1.1.3. Contexto histótico

Lógica: Algo de historia

- **S. IV a.C.:** Aristóteles formaliza el razonamiento humano. Fundó la lógica clásica o lógica aristotélica. Por Ejemplo:

  Todos los hombres son mortales. Sócrates en un hombre. Luego Sócrates es mortal.

- ... tiempo después ...
- **1920:** Hilbert propone "la axiomatización de las matemáticas"...
- **...en 1931:** Gödel demuestra los teoremas de incompletitud.



Aristóteles

Hilbert

Gödel

Teorema de incompletitud (Gödel, 1931)

Ninguna teoría matemática formal capaz de describir los números naturales y la aritmética con suficiente expresividad, es a la vez consistente y completa.

- Si los axiomas de una teoría no se contradicen entre sí (consistencia), entonces existen enunciados que NO se pueden probar ni refutar a partir de ellos (completitud).

Segundo teorema de incompletitud (Gödel, 1931)

Caso particular del primero: Una de las sentencias indecidibles de una teoría es aquella que afirma la consistencia de la misma.

Respuesta al *Entscheidungsproblem* (Church, 1936)

La lógica de de 1^{er} orden no posee un procedimiento de decisión (algoritmo) que permita demostrar que una fórmula cualquiera sea un teorema de dicha lógica. Esta lógica se considera, por tanto, **indecidible** (Church 1936).

Teorema de la parada (Turing, 1936)

Dada una Máquina de Turing M y una palabra w, determinar si M terminará en un número finito de pasos cuando es ejecutada usando w como dato de entrada, **es indecidible** (Turing 1936).

- Lógica y Computación **son equivalentes**: Determinar el problema de parada se reduce a demostrar en LPO la fórmula que expresa la existencia de un output a partir de la aplicación de una serie de instrucciones.

- **1965:** Alan Robinson publica un **método de resolución** para lógica de primer orden. Sienta las bases de la deducción automática:
  - Verificación automática de programas: a partir de su especificación formal y utilizando demostradores automáticos de teoremas.
- **1972:** Alain Colmerauer crea **Prolog**, el primer lenguaje de programación lógica. Sienta las bases de la **inteligencia artificial**:
  - Permite inferir/deducir conocimiento a partir de una base de conocimientos y una serie de reglas (de inferencia).

1.1.3.1. Ejemplo del Entscheidungsproblem

Ejemplo del Entscheidungsproblem: El juego de la vida (Conway, 1970)

- Se define como una malla de células infinito donde:
  - Una célula muerta con 3 células vecinas vivas "nace".
  - Una célula viva con 2 o 3 vecinas vivas sigue viva.
  - ... en otro caso muere (por "soledad" o "superpoblación").
- Es una máquina de Turing $\rightarrow$ Dado un patrón inicial cualquiera, no podemos determinar si se estabilizará en un tiempo finito o no.

PLAY[1]

[1] Ver "Las Matemáticas tienen una Terrible Falla" en https://bit.ly/3sFyaxq.

1.1.4. Sintaxis, semántica y teoría interpretativa

Lógica: Sintaxis, semántica y teoría interpretativa

- En este curso asumimos que el alumno a cursado **"Lógica"** en el grado.

  ... en caso contrario, hay cursos en abierto de la URJC.
- Trabajaremos con **cláusulas de Horn**, i.e., disyunción de literales con, como máximo, un literal positivo. P.ej. dada la consecuencia lógica:

  $$T[\forall x\ (\ Hombre(x) \rightarrow Mortal(x)\ ), Hombre(socrates)] \vdash Mortal(socrates)$$

  se reescribe como el siguientes conjunto de cláusulas:

  $$\{\ Mortal(x) \vee \neg Hombre(x),\ Hombre(socrates),\ \neg Mortal(socrates)\ \}$$

  donde el consecuente, $Mortal(socrates)$, están negado. Si este conjunto es **insatisfacible**, significa que hay consecuencia lógica.

1.2. Método de resolución de Robinson.

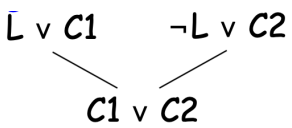
1.2.1. Método de Resolución de Robinson

Lógica: Método de resolución de Robinson, 1965

- Basado en:

Teorema de Herbrand
Un conjunto de cláusulas C es insatisfacible sii existe un conjunto finito de instancias básicas de cláusulas de C que es insatisfacible.

- **Idea general**: Plantear un método de obtención de nuevas instancias deducidas del conjunto original, de forma que si llega a deducirse un literal y su negación puede concluirse que el conjunto original es insatisfacible.
- Está basado en la **regla de resolución básica**: De dos instancias básicas $L \vee C1$ y $\neg L \vee C2$ (L es un literal) puede deducirse una nueva instancia básica $C1 \vee C2$, llamada **resolvente**:



- La aplicación sucesiva de la regla de resolución permite obtener una **contradicción** cuando el conjunto original es insatisfacible.

- Dado un conjunto C de instancias básicas:
  1. Generar el conjunto R de todos los resolventes que pueden obtenerse aplicando la regla de resolución entre instancias del conjunto C de todas las formas posibles.
  2. Si [] está incluida en R entonces terminar $\Rightarrow$ C es insatisfacible.
  3. Si $R \subseteq C$ significa que ya se han generado todos los resolventes posibles, entonces terminar $\Rightarrow$ C es satisfacible.
  4. Hacer $C = C \cup R$ y repetir desde paso 1.
- El método es **correcto**: Si deducimos [] entonces C es insatisfacible.
- El método es **completo**: Si C is insatisfacible, entonces con la aplicación de la regla de resolución deduciremos [].

C = {I1: ¬p(a, f(b)), I2: ¬p(b, f(b)), I3: p(a, f(b)) ∨ q(f(b)), I4: p(b, f(b)) ∨ ¬q(f(b))}

resuelve I1 con I2: NO
resuelve I1 con I3: q(f(b))
resuelve I1 con I4: NO
resuelve I2 con I3: NO
resuelve I2 con I4: ¬q(f(b))
resuelve I3 con I4: p(a, f(b)) ∨ p(b, f(b))

R = {I5: q(f(b)), I6: ¬q(f(b)), I7: p(a, f(b)) ∨ p(b, f(b))}

En R no está □ , por tanto redefinimos C = C ∪ R y buscamos nuevos resolventes:

resuelve I1 con I5: NO
resuelve I1 con I6: NO
resuelve I1 con I7: p(b, f(b))
resuelve I2 con I5: NO
resuelve I2 con I6: NO
resuelve I2 con I7: p(a, f(b))

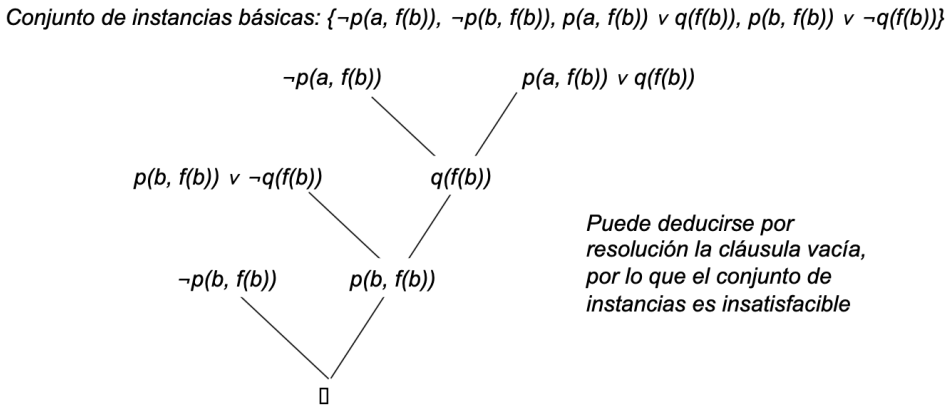
resuelve I3 con I5: NO
resuelve I3 con I6: p(a, f(b))
resuelve I3 con I7: NO
resuelve I4 con I5: p(b, f(b))
resuelve I4 con I6: NO
resuelve I4 con I7: NO

resuelve I5 con I6: □
resuelve I5 con I7: NO
resuelve I6 con I7: NO

R = {p(b, f(b)), p(a, f(b)), □ }

R incluye a □ ➔ C es insatisfacible

- La aplicación de sucesivos pasos de resolución se representar en forma de árbol (**árbol de resolución**):
  - Árbol binario invertido (cada dos nodos tienen un "hijo" común)
  - Cada nodo representa una instancia básica.
  - Sólo se representan los pasos relevantes para llegar a [].



1.2.2. Resolución con Unificador de Máxima Generalidad

1.2.2.1. Sustitución y algoritmo de unificación

Resolución con UMG: Sustitución

- Un **sustitución** $\alpha = \{x_1/t_1, \ldots, x_n/t_n\}$ es una función finita de un conjunto de variables de un lenguaje en el de términos. Donde x_i/t_i es una **ligadura**.
- Dada una fórmula F y una sustitución $\alpha = \{x_1/t_1, \ldots, x_n/t_n\}$, se denomina **aplicación de α a F ($F\alpha$)** a la fórmula obtenida reemplazando simultáneamente cada ocurrencia en F de x_i por t_i, para cada $x_i/t_i \in \alpha$.
- F' es instancia de F si existe una sustitución $\alpha \neq \emptyset$ tal que $F' = F\alpha$

Resolución con UMG: Algoritmo de Unificación

- Sean A y B dos átomos con el mismo símbolo de predicado:
  1. $\alpha = \lambda$
  2. Mientras $A\alpha \neq B\alpha$:
     a) Encontrar el símbolo más a la izquierda en $A\alpha$ tal que el símbolo correspondiente en $B\alpha$ sea diferente.
     b) Sean t_A y t_B los términos de $A\alpha$ y $B\alpha$ que empiezan con esos símbolos:
        - Si ni t_A ni t_B son variables o, si uno de ellos es una variable que aparece en el otro $\Rightarrow$ terminar con fallo (A y B no son unificables)
        - En otro caso, sea t_A una variable $\Rightarrow$ el nuevo α es el resultado de $\alpha\{t_A/t_B\}$
  3. Terminar, siendo α el umg de A y B

Resolución con UMG: Algoritmo de Unificación: Ejemplos

- Ejemplo 1: $A = P(x,x)$ y $B = P(f(a), f(b))$

α	Aα	Bα	(t_a, t_b)
λ	P(x,x)	P(f(a),f(b))	(x,f(a))
$\{x/f(a)\}$	P(f(a),f(a))	P(f(a),f(b))	(a,b)
FALLO	A y B NO son unificables		

- Ejemplo 2: $A = P(x, f(y))$ y $B = P(z,x)$

α	Aα	Bα	(t_a, t_b)
λ	P(x,f(y))	P(z,x)	(x,z)
$\{x/z\}$	P(z,f(y))	P(z,z)	(f(y),z)
$\{x/f(y), z/f(y)\}$	P(f(y),f(y))	P(f(y),f(y))	ÉXITO
A y B son unificables, su umg es $\{x/f(y), z/f(y)\}$			

1.2.2.2. Algoritmo de resolución con UMG

Resolución con UMG: Definición

- **Regla de resolución con umg**: Sean $L_1 \vee \cdots \vee L_n \vee C_1$ y $\neg L'_1 \vee \cdots \vee \neg L'_m \vee C_2$ dos cláusulas, donde todos los L_{ij} son literales con el mismo símbolo de predicado. Puede deducirse una nueva cláusula $(C_1\rho_1 \vee C_2\rho_2)\beta$, llamada resolvente, donde
  - ρ_1 y ρ_2 son renombrados cuyos dominios respectivos son todas las variables de cada cláusula y $Rango(\rho_1) \cap Rango(\rho_2) = \emptyset$
  - β es **umg** de $\{L_1\rho_1, \ldots, L_n\rho_1, \neg L'_1\rho_2, \ldots, \neg L'_m\rho_2\}$
- La regla de resolución con umg se apoya en una versión de la **regla de factorización** para LPO: Dada una cláusula $L_1 \vee \cdots \vee L_n \vee C$, siendo $L_1, \ldots, L_n$ literales con el mismo símbolo de predicado, puede deducirse una nueva cláusula $L \vee C\beta$ donde
  - β es unificador de $L_1, \ldots, L_n$
  - $L = L_1\beta = \cdots = L_n\beta$ El literal L se denomina

    factor de $L_1 \vee \cdots \vee L_n \vee C$
- La aplicación de la regla de resolución con UMG es **correcta**.
- La aplicación de la regla de resolución con UMG es **completa**.

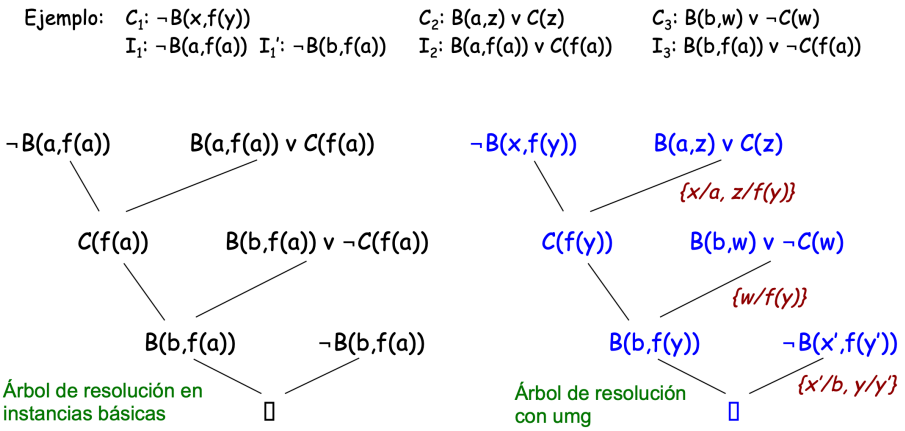
Teorema

Un conjunto de cláusulas es insatisfacible sii se puede deducir [] a partir de él por resolución con umg.

- Por tanto, el método general de insatisfacibilidad se puede reducir a la búsqueda de [] a partir del conjunto de cláusulas, en lugar de tener que generar conjuntos de instancias básicas.

Resolución con UMG: Ejemplo

- Pueden construirse árboles de resolución en los que los resolventes de cada dos cláusulas se obtienen en un paso de resolución con umg.
  - Por cada paso de resolución en instancias básicas puede definirse un paso de resolución con umg.



1.2.3. Estrategias de resolución

Lógica: Estrategias de resolución (de Robinson)

- Distintas estrategias de resolución tienen sus ventajas e incovenientes.
  - P.ej., la aplicación del procedimiento de saturación, sin limitaciones, genera normalmente muchas cláusulas irrelevantes y redundantes.
- Para hacer el proceso de resolución computacionalmente eficiente es necesario aplicar criterios selectivos de forma sistemática que simplifiquen el proceso.
  - Estrategias de **simplificación**: con el objetivo de reducir el número de cláusulas en el conjunto.
  - Estrategias de **refinamiento**: con el objetivo de limitar la generación de cláusulas.

Estrategia	Correcta	Completa
Saturación	Si	Si
Lineal	Si	Si
Input	Si	No, (caso general)
Dirigida	Si	Si, si el conjunto soporte es satisfacible
Ordenada	Si	No

¿Qué significa ser correcta/completa?

- **Correcta**:

  $$[] \rightarrow \text{S insatisfacible}$$

  [] se deduce sólo si el conjunto de cláusulas S es insatisfacible.
- **Completa**:

  $$\text{S insatisfacible} \rightarrow []$$

  Si el conjunto de cláusulas S es insatisfacible, se deduce [].

1.2.3.1. Estrategia de resolución SLD

Lógica: Estrategia de resolución SLD.

- SLD[2]: La estrategia usada por el lenguaje de programación Prolog.
- Caso particular de la resolución general para cláusulas de Horn, donde:
  - Las cláusulas objetivo no tiene literal afirmado.
  - Las cláusulas soporte tienen un literal afirmado (el primero).
- Dado un conjunto inicial de cláusulas de Horn $\{C_1, \ldots, C_i, \ldots, C_n\}$:

  Existe una secuencia (*derivación*) $< C_i, C_{n+1}, \ldots, [] >$ tal que:
  - C_{n+1} es el resolvente de la cláusula objetivo C_i y una cláusula soporte.
  - C_k, con $k > n+1$, es el resolvente de C_{k-1} con una cláusula soporte.
  - Cada paso de resolución es de la forma $L \vee C, \neg L \vee C' \rightarrow C \vee C'$.

  ...si y solo si el conjunto inicial es insatisfacible.

[2] SLD significa resolución **L**ineal con funciónn de **S**elección para cláusulas **D**efinidas (Selecting a literal, using a Linear strategy, restricted to Denfinite clauses).

1.2.3.2. Ejercicio

Estrategia de resolución SLD: Ejercicio

1. Obtener la forma clausular y resolver usando la estrategia SLD:

$$\begin{array}{rl} \forall ls & Concatenar([\,],ls,ls) \\ \forall x\,\forall xs\,\forall ls\,\forall ns & (\,Concatenar(xs,ls,ns) \rightarrow Concatenar([x|xs],ls,[x|ns])\,) \\ \hline \exists la\,\exists lb & Concatenar(la,lb,[1,2,3,4]) \end{array}$$

2. Comprobar que es equivalente al programa Prolog:

```
concatenar([],Ls,Ls).
concatenar([X|Xs],Ls,[X|Ns]) :- concatenar(Xs,Ls,Ns).

?- concatenar(La, Lb, [1,2,3,4]).
```

1.3. Prolog.

1.3.1. Algo de la historia de Prolog

Prolog: Algo de historia.

- **1960's:**
  - Robinson (Robinson 1965): propone en 1965 una regla de inferencia a la que llama resolución, mediante la cual la demostración de un teorema puede ser llevada a cabo de manera automática.
  - Green (Green 1969): diseña un probador de teoremas que extrae de la prueba, el valor de las variables para las cuales el teorema es válido.
- **1970's:**
  - Kowalski (Kowalski y Kuehner 1971): propone la resolución SLD (muy eficiente).
  - Alain Colmerauer (Colmerauer y Roussel 1996): crea **Prolog**, el primer lenguaje de programación lógica (implementado en Fortran).
  - Kowalski (Kowalski 1979): interpretación procedimental de la lógica:
    Algorithm = logic + control
  - D.H.D. Warren (Warren 1977): desarrolla el primer compilador, DEC-10 Prolog (escrito casi por completo en Prolog).
- **1980's and 1990's:**

  - Fifth Generation Project en Japón: Importante investigación sobre los paradigmas básicos y las técnicas de aplicación avanzadas.
  - Varias implementaciones comerciales de Prolog, libros y manuales, utilizando el estándar de facto, la familia Edinburgh Prolog.
    - En 1995 dio lugar a la norma ISO Prolog.
  - Programación Lógica con Restricciones (CLP) (Jaffar y Maher 1994): Extiende expresividad de Prolog –abrió nuevos campos de aplicación.
- **2000's:**
  - Muchas otras extensiones: orden superior, tipos/modos, concurrencia, distribución, objetos, sintaxis funcional, etc.
  - Compiladores altamente optimizadores, paralelismo automático, verificación y depuración automáticas, entornos avanzados.
- **2010's:**
  - Variaciones: Datalog, Answer Set Programming (soporta negación mediante modelos estables), Minikanren, MiniZinc, SQL, Yedalog.
- **2020's:**
  - Verse (Augustsson et al. 2023): Lenguaje lógico y funcional de Epic Games (Fortnite).
  - Logica language (Skvortsov, Xia y Ludäscher 2024): desarrollado en Google.
  - Janus (Andersen y Swift 2023): Programación multiparadigma en Prolog y Python.
  - s(CASP) (Arias et al. 2018): desarrollado por mi en colaboración con la University of Texas at Dallas e IMDEA Software Institute:
    - HackReason: Varios hackathon sobre s(CASP) organizado por la UT Dallas AI Society en Enero 14-15, 2021 y 2022.[3]

1.3.2. Algunas aplicaciones de Prolog

Prolog: Aplicaciones.

Areas de aplicación:

- Procesamiento del lenguaje natural.
- Problemas de programación y optimización.
- Muchos problemas relacionados con la IA, programación de (Multi) agentes.
- Integración de datos heterogéneos.

[3] Ver detalles y los proyectos en `https://utd-hackreason-2021.devpost.com` y `https://utd-hackreason-2022.devpost.com`.

- Analizadores y verificadores de programas.

Aplicaciones concretas:

- El sistema IBM Watson (2011) tiene partes importantes escritas en Prolog.
- Clarissa, una interfaz de usuario de voz por la NASA para navegar ISS procedimientos.
- El primer intérprete de Erlang fue desarrollado en Prolog por Joe Armstrong.
- El sistema de instalación y configuración de redes de Microsoft Windows NT.
- El Network Resource Manager (NRM) de Ericsson.
- "Sistema de reserva de vuelos que gestiona casi un tercio de los billetes de avión del mundo" (SICStus).
- La especificación de la máquina abstracta Java.

Universidad
Rey Juan Carlos

Cápitulo 2

Programación lógica pura (Turing completo)

2.1. Sintaxis y Semántica de Prolog.

2.1.1. Sintaxis

Sintaxis: Alfabeto

- El alfabeto de Prolog se define con los siguientes tipos de símbolos:
  - Símbolos de constante (átomos): *Cadenas de letras, dígitos y guión bajo, que (i) empiezan por letra minúscula o (ii) están entre comillas simples.* `a, b, c, ..., 'X', ...`
    - Otros: Números (enteros y reales), lista vacía `[]`, `true`, `false`, etc.
  - Símbolos de variable: *Cadenas de letras, dígitos y guión bajo, que empiezan por (i) letra mayúscula o (ii) guión bajo.*[1] `X, Xs, _X, _ , ...`
  - Símbolos de funtor: `f(_), g(_,_), ...` o indicando aridad `f/1, g/2`.
  - Símbolos de predicado: `r, p(_), q(_,_), ...` o `r/0, p/1, q/2`.
- En Prolog puede haber símbolos comunes entre los conjuntos de constantes, funciones y predicados (se desambiguan por posición) pero si hay predicados con el mismo nombre y distinta aridad Prolog genera un aviso.

Sintaxis: Expresiones

- Una expresión es cualquier concatenación finita de símbolos del alfabeto. Las expresiones relevantes en Prolog son:

- Términos:
  - Un símbolo de constante es un término.
  - Un símbolo de variable es un término.
  - Si `f` es un símbolo de funtor n-aria y `t1,..., tn` son términos entonces `f(t1,..., tn)` es un término.
- Predicados:
  - Si `p` es un símbolo de predicado n-aria y `t1,..., tn` son términos entonces `p(t1,..., tn)` es un predicado.

- Cláusulas (reglas, hechos y consultas):
  - Las reglas son de la forma:

    ```
    H :- B1, B2,..., Bn.
    ```

    donde `H` y `Bi` son predicados, '`:-`' representa la implicación '←' y '`,`' conjunción '∧'. La cabeza de la clausula es `H` y el resto es el cuerpo.
  - Los hechos son clausulas sin cuerpo:

    ```
    H.
    ```

    equivalente a `H :- true.`
  - Las consultas no tienen cabeza:

    ```
    ?- B1, B2,..., Bn.
    ```

Sintaxis: Programas

[1] Las variables que empiezan con guión bajo se llaman variables anónimas (ver pág. 2.1.2.2).

- En Prolog, un programa es:
  - Un conjunto de clausulas (reglas y hechos).
  - Se pueden incluir comentarios en línea con el símbolo `%`.
- El programa se 'ejecuta':
  - Invocando la consulta en el top-level.

```
1   mortal(X) :- hombre(X).          % Todos los hombres son mortales.
2   hombre(socrates).                % Sócrates es un hombre.
3
```

```
?- mortal(X).

X = socrates ?

yes
?-
```

  - Compilando y ejecutando un programa con el predicado `main/1`.

2.1.2. Semántica

Semántica: Formal, informal (y operacional)

- Programas de cláusulas de Horn (definite programs) tiene dos semánticas equivalentes:
  - El modelo mínimo de Herbrand de P. $Mp = \{ A \in Bp \mid P \vDash A \}$
  - *Least fixed point* de Tp (operador de consecuencia inmediata). $Tp \uparrow \omega = \bigcup_{i=0}^{\infty} Tp^i(\emptyset)$
    - Donde $Tp(I) = \{A_0 \in Bp \mid A_0 \leftarrow A_1, \ldots, A_n \in P \wedge A_i \in I\}$.

- Adicionalmente, su significado lo podemos ver como:
  1. Un conjunto de fórmulas a las que aplicamos deducción.
     $T[\, \forall x \, (\, Hombre(x) \rightarrow Mortal(x) \,),\, Hombre(socrates) \,] \vdash \exists x \, Mortal(x)$
  2. Un conjunto de cláusulas a las que aplicamos resolución SLD.
     $\{\, Mortal(x) \vee \neg Hombre(x),\, Hombre(socrates),\, \neg Mortal(x) \,\}$
  3. Un conjunto de procedimientos ejecutados a partir de la consulta siguiendo una semántica operacional dada (ver tema 3).

Semántica: Resolución SLD + búsqueda

1. Obtener la forma clausular y resolver usando la estrategia SLD:

$$\begin{array}{rl} \forall ls & Concatenar([\,], ls, ls) \\ \forall x\, \forall xs\, \forall ls\, \forall ns & (\, Concatenar(xs, ls, ns) \rightarrow Concatenar([x|xs], ls, [x|ns])\,) \\ \hline \exists la\, \exists lb & Concatenar(la, lb, [1,2,3,4]) \end{array}$$

2. Comprobar que es equivalente al resultado del programa:

```
concatenar([],Ls,Ls).
concatenar([X|Xs],Ls,[X|Ns]) :- concatenar(Xs,Ls,Ns).

?- concatenar(La, Lb, [1,2,3,4]).
```

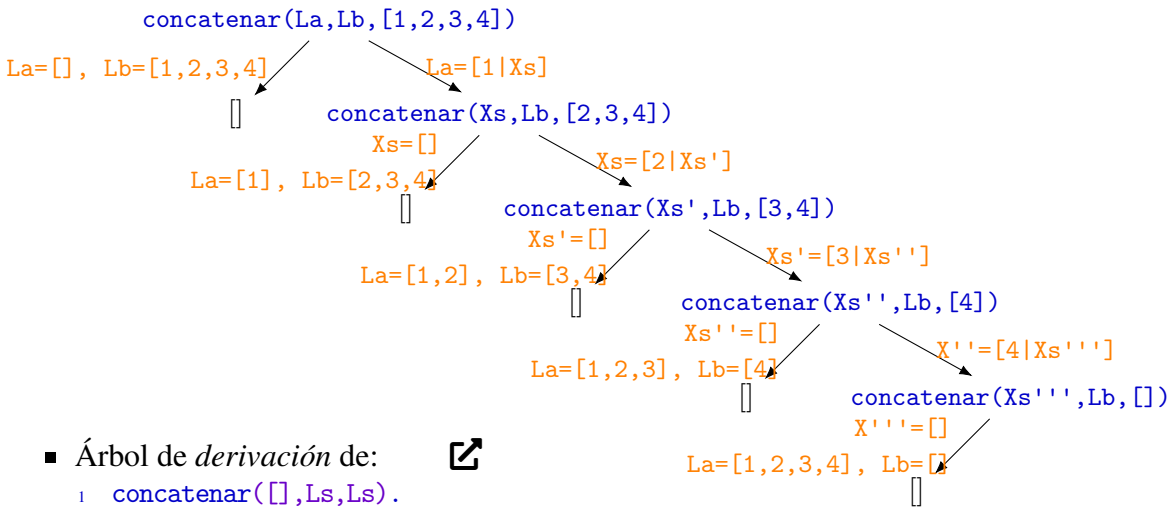
2.1.2.1. Árbol de derivación

Semántica: Árbol de derivación

- La estrategia de resolución SLD, desde el punto de vista de la evaluación de un programa Prolog, es un árbol de derivación:
  1. La cabeza es la consulta Q.
  2. Los nodos hijos de Q son el cuerpo (resolvente) de las cláusulas cuya cabeza unifica con la consulta Q.
  3. Si el resolvente es [] la consulta tiene éxito:

     Nota: si la consulta tiene variables, la solución es la composición de las sustitución calculadas (en las aristas).
  4. En caso contrario, se "resuelve" un literal del resolvente.
     a) Se crea un hijo por cada cláusula que unifica con dicho literal.
     b) Se añade el cuerpo de la clausula al resolvente.
     c) Se vuelve al punto 3.

Semántica: Árbol de derivación (Ejemplo)



- Árbol de *derivación* de:

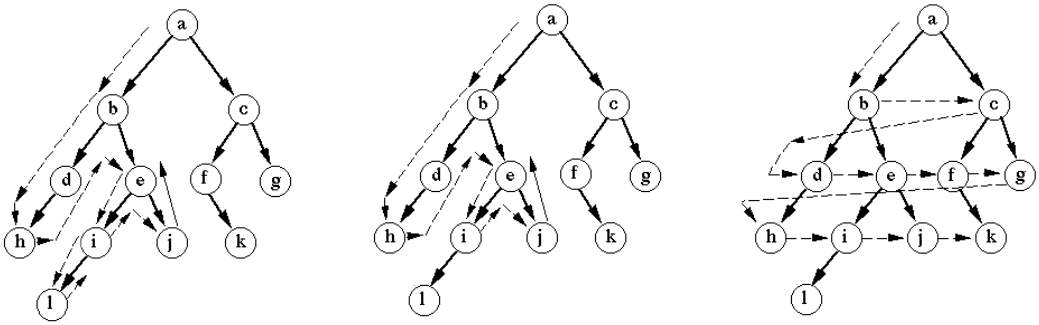
```
1 concatenar([],Ls,Ls).
2 concatenar([X|Xs],Ls,[X|Ns]) :-
3       concatenar(Xs,Ls,Ns).
4
5 ?- concatenar(La, Lb, [1,2,3,4]).
```

2.1.2.2. Estrategias de búsqueda

Semántica: Estrategias de búsqueda

- ... por lo tanto, ejecutar un programa Prolog es un **problema de búsqueda en el árbol de derivación. Y existen varias estrategias:**

En profundidad **Profundización iterativa** **En Anchura**



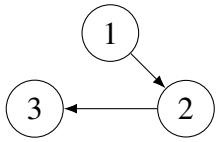
Semántica: Estrategias de búsqueda (Ejercicios)

1. Dado el siguiente programa:

```
edge(1, 2).
edge(2, 3).

reach(X, Y) :- edge(X, Z), reach(Z, Y).
reach(X, Y) :- edge(X, Y).

?- reach(1,Y).
```



a) Construir el árbol de derivación.

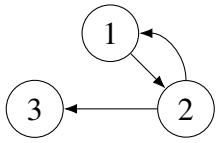
b) Discutir las ventajas e inconvenientes de la búsqueda en profundidad vs. anchura.

2. Dado el siguiente programa:

```
edge(1, 2).
edge(2, 1).
edge(2, 3).

reach(X, Y) :- edge(X, Z), reach(Z, Y).
reach(X, Y) :- edge(X, Y).

?- reach(1, Y).
```



a) Construir el árbol de derivación.

b) Discutir las ventajas e inconvenientes de la búsqueda en profundidad vs. anchura.

Semántica: Estrategias de búsqueda (Paquetes disponibles en Ciao Prolog)

- Ciao Prolog (Hermenegildo et al. 2012) dispone de diferentes estrategias de búsqueda:

  ***bf*:** Implementa búsqueda en anchura (Cabeza, Carro y Hermenegildo). Ejemplo:

  ```
  :- use_package(sr/bfall).
  ...
  bf_reach(X, Y) :- edge(X, Z), bf_reach(Z, Y).
  bf_reach(X, Y) :- edge(X, Y).
  ```

  ***id*:** Implementa profundidad iterativa (Haemmerlé et al.). Ejemplo:

  ```
  :- use_package(id).
  :- iterative(id_reach/2,0,(_(X,Y) :- Y is X + 5),10).
  ...
  id_reach(X, Y) :- edge(X, Z), id_reach(Z, Y).
  id_reach(X, Y) :- edge(X, Y).
  ```

  ***rndsearch*:** Este bundle externo implementa un algoritmo de búsqueda aleatoria. TFG de (Blázquez Ballesteros 2017).

NOTA: Uso de variables anónimas

1. **En predicados**, cuando no son necesarias (evita avisos). P.ej., en:
   ```
   node(X) :- edge(X, _).
   node(Y) :- edge(_, Y).     % node(X) :- edge(_, X).
   ```
   donde definimos los nodos de un grafo a partir del predicado `edge/2`, y solo uno de los argumentos es necesario.
2. **En consultas**, cuando no estamos interesados en su valor:
   - Si usamos solo el guión bajo, cada aparición de `_` es una variable diferente. P.ej., la consulta:
     ```
     ?- concatenar(_,[2],[1,2]), concatenar(_,Z,[1,2]).
     ```
     genera tres soluciones `Z=[1,2]`, `Z=[2]`, y `Z=[]`.
   - ...pero usando un mismo símbolo, p.ej., `_A`, la consulta:
     ```
     ?- concatenar(_A,[2],[1,2]), concatenar(_A,Z,[1,2]).
     ```
     genera como única respuesta `Z=[2]`, porque `_A` se instancia a `[1]` pero al ser anónima no aparece en los resultados de la consulta.

2.2. Programas (y tipos) recursivos.

2.2.1. Aritmética con enteros

Aritmética de Peano

- Los números naturales son el tipo recursivo más simple.
- En 1889, Peano publicó sus cinco axiomas para la aritmética:
  1. El 0 es un número natural.[2]
  2. Todo número natural n tiene un sucesor $s(n)$.
  3. El 0 no es el sucesor de ningún número natural.
  4. Si hay dos números naturales n y m con el mismo sucesor, entonces n y m son el mismo número natural.
  5. El principio de inducción matemática, su formulación en lógica de segundo orden es:

$$\forall \phi \left(\left(\phi(1) \wedge \forall x (\phi(x) \rightarrow \phi(x')) \right) \rightarrow \forall x \, \phi(x) \right)$$

Aritmética de Peano: Implementación (y semántica) lógica

[2] Los matemáticos (incluido Peano) suelen usar el 1 como primer número natural.

- Primero, veremos las operaciones como relaciones (no como funciones).
- Segundo, analizaremos la corrección y completitud de los programas.

- **Ejemplo 1:** `nat/1` define el tipo recursivo de los números naturales.
  - Los construimos con la constante `0` y el funtor `s/1`.
  - `?- nat(X)` genera recursivamente: `0`, `s(0)`, `s(s(0))`, ...
  - Donde $s^n(0)$ denota n (aplicar n veces el funtor sucesor al 0).

```
% nat(X): X is a natural number.
nat(0).
nat(s(X)) :- nat(X).
```

- **Proposición 1:** el programa `nat/1` es correcto y completo con respecto al conjunto de consultas `?- nat(s^i(0))` para $i \geq 0$.

- **Demostración 1:**

  (1) Completitud: *Sea `n` un número natural. Demostramos que el objetivo `?- nat(n)` es deducible a partir del programa mediante un árbol de derivación. O bien `n` es `0` o de las forma `s^i(0)`. El árbol de derivación para `nat(0)` es trivial. El árbol de derivación para `nat(s(...s(0)...))` contiene `n` derivaciones, usando la regla recursiva, hasta alcanzar el hecho `nat(0)`.*[3]

  (2) Corrección: *Supongamos que el `nat(X)` es deducible a partir del programa en `n` deducciones. Demostramos que `nat(X)` es parte de la semántica del programa por inducción en n. Si $n = 0$, entonces la consulta debe haber sido probada usando la cláusula base, lo que implica que `X=0`. Si $n > 0$, entonces la consulta debe ser de la forma `nat(s(X'))`, ya que es deducible a partir del programa, y además, `nat(X')` es deducible en $n-1$ deducciones. Por la hipótesis de inducción, `X'` es parte de la semántica del programa, i.e., `X'=s^k(0)` para algún $k \geq 0$.*

- **Ejemplo 2**: el predicado `plus/3` define la suma (de manera recursiva) como una relación.
  - La semántica del programa es el conjunto de hechos `plus(X,Y,Z)` donde `X`, `Y`, y `Z` son números naturales y `X+Y=Z`.

```
% plus(X,Y,Z): X, Y, and Z are natural numbers
%               such that Z is the sum of X and Y
plus(0,X,X) :- nat(X).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).
```

- **Proposición 2**: El programa `plus/3` constituyen una axiomatización correcta y completa de la suma con respecto a la semántica de `plus/3`.

- **Demostración 2**: (Ejercicio) realizar la demostración (siguiendo esquema usado en `nat/1`).

[3] Ejercicio: dibujar el árbol de derivación.

Aritmética de Peano: Ejercicios

- Implementa y demostrar si son correctos y completos:
  1. Menor o igual.
     ```
     % leq(X,Y): X, and Y are natural numbers
     %           such that X is less than or equal to Y.
     ```
  2. Multiplicación como suma repetida.
     ```
     % times(X,Y,Z): X, Y, and Z are natural numbers
     %               such that Z is the product of X and Y.
     ```
  3. Exponenciación como multiplicación repetida.
     ```
     % exp(N,X,Y): N, X, and Y are natural numbers
     %             such that Y equals X raised to the power N.
     ```
  4. Calculando el factorial.
     ```
     % factorial(N,F): F equals N factorial.
     ```
  5. El mínimo de dos números.
     ```
     % minimum(N1,N2,Min): The minimum of N1 and N2 is Min.
     ```
- Implementa las siguientes operaciones:
  6. Calcular el residuo/módulo (resto de una division)
     ```
     % mod(X,Y,Z): Z is the remainder of the integer
     %             division of X by Y
     ```
     a) Implementar con: (i) `times/3` y (ii) recursivamente,
     b) Comparar sus complejidades (nodos en el árbol de derivación).
  7. La función de Ackermann:[4]
     ```
     % ackermann(X,Y,A): A is the value of Ackerman's
     %                    function for X and Y.
     ```
  8. El algoritmo de Euclides para calcular el máximo común divisor.
     ```
     % gcd(X,Y,Z): Z is the greatest common divisor of
     %             the natural numbers X and Y
     ```

2.2.2. Listas

Manipulación de listas en Prolog

- Un tipo recursivo mas complejo es una estructura binaria, la lista.
  - Se construye con la constante `[]`, para la lista vacía.
  - y con el funtor binario `./2`, de modo que:
    ```
    % list(List): List is a list
    list([]).
    ```

[4] Su definición está disponible en https://shorturl.at/Sh6Ry

```
3 list(.(_,Ls)) :- list(Ls).
```

donde la variable anónima `_` es un elemento cualquiera (*head*) y `Ls` es una lista (*tail*).

- `?- list(X)` genera recursivamente: `[]`, `[_]`, `[_,_]`, `[_,_,_]`, ...

- Syntactic Sugar: en lugar de `.(_,Ls)` usaremos `[_|Ls]`.
  - Es decir, la lista `[1,2,3]` se construye como `[1|[2|[3|[]]]]`, pero también `[1|[2,3]]`, `[1,2|[3]]`, `[1,2,3|[]]`, etc.
  - Nota: el mgu de `[1,2,3]` y `[X|Xs]` es `{X=1, Xs=[2,3]}`.

Manipulación de listas en Prolog: Ejercicios

1. Implementa la mas básica operación con listas, determinar si un elemento particular esta en una lista:
```
1 % member(Element, List): Element is an element
2 %                        of the list List
```
2. Escribe la consulta correspondiente para:
   - Comprobar si `b` está en la lista `[a,b,c]`.
   - Encontrar los elementos de la lista `[a,b,c]`.
   - Encontrar una lista que contiene `a`.
3. Implementar operaciones para obtener los prefijos / sufijos de una lista.
```
1 % prefix(Prefix, List): Prefix is a prefix of List
2 % suffix(Suffix, List): Suffix is a suffix of List
```
4. Define concatenación de listas y redefine `prefix/2` y `suffix/2`.
```
1 % append(Xs,Ys,XsYs): XsYs is the result of concatenating
2 %                      the lists Xs and Ys
```

Manipulación de listas en Prolog: Analizando versiones de sublist/2

- El predicado `sublist(Sub,List)` determina si `Sub` es una sublista de `List`, de modo que las sublistas tiene los elementos consecutivos: `[b,c]` es una sublista de `[a,b,c,d]`, mientras que `[a,c]` no lo es.
- (Ejercicio) Explica la lógica las siguientes implementaciones de `sublist/2`.

```
1  % a)
2  sublist(Xs,Ys) :-  prefix(Ps,Ys), suffix(Xs,Ps).
3  % b)
4  sublist(Xs,Ys) :-  prefix(Xs,Ss), suffix(Ss,Ys).
5  % c)
6  sublist(Xs,Ys) :-  prefix(Xs,Ys).
7  sublist(Xs,[Y|Ys]) :- sublist(Xs,Ys).
8  % d)
9  sublist(Xs,AsXsBs) :-  append(As,XsBs,AsXsBs), append(Xs,Bs,XsBs).
10 % e)
11 sublist(Xs,AsXsBs) :-  append(AsXs,XsBs,AsBsXs), append(As,Xs,AsXs).
```

Manipulación de listas en Prolog: Recursión con acumulación

- Implementemos el predicado `reverse(List,Reverse`, donde `Reverse` es una lista que contiene los elemento en `List` en orden inverso a como aparecen en `List`. I.e., `?- reverse([1,2,3],[3,2,1])` tiene éxito.

1. Usando `append/3` tenemos:
```
% a) Naive reverse
reverse([],[]).
reverse([X|Xs], Zs) :-
      reverse(Xs,Ys),
      append(Ys,[X],Zs).
```
pero con complejidad cuadrática (ver árbol de derivación de la consulta anterior).

2. Como alternativa, definimos un predicado auxiliar `reverse_(Xs,Acc,Ys)`, donde `Ys` es el resultado de concatenar `Acc` a los elementos de `Xs` invertidos:
```
% a) Reverse-accumulate
reverse(Xs, Ys) :-
      reverse_(Xs, [], Ys).
reverse_([], Acc, Acc).
reverse_([X|Xs], Acc, Ys) :-
      reverse_(Xs, [X|Acc], Ys).
```

- (Ejercicio) Calcula y justifica la complejidad de esta versión usando la consulta anterior.

2.2.2.1. Combinar aritmética y listas en Prolog

Combinar enteros y listas en Prolog

- Dada la siguiente implementación de `length/2`:
```
% length(Xs, N): The list Xs has N elements
length([], 0).
length([_|Xs], s(N)) :- length(Xs, N).
```
expresa una relación entre números y listas, usando la estructura recursiva de ambos tipos, y donde, `length([1,2], s(s(0)))`, que indica que `[a,b]` tiene dos elementos, está en la semántica del programa.
- Resuelve los siguientes ejercicios:
  1. Considera tres usos alternativos del programa `length/2`.
  2. Define la relación `sum(ListOfIntegers, Sum)`, que se cumple si `Sum` es la suma de los elementos en `ListOfIntegers`.

a) Usando el programa `plus/3`.

b) Sin utilizar predicados auxiliares.

2.2.3. Árboles binarios

Árboles binarios en Prolog

- Otro importante tipo recursivo es el de los árboles binarios, que se representan por el funtor terciario `tree(Element, Left, Right)`, donde:
  - `Element` es el elemento en el nodo.
  - `Left` and `Right` son el sub-árbol de la izquierda y derecha respectivamente.
- El árbol vacío se representa por la constante `void`.
- Resuelve los siguientes ejercicios:
  1. Dibuja el árbol `tree(a,tree(b,void,void),tree(c,void,void))`.
  2. Implementa `binary_tree(Tree)` (Nota: es doblemente recursivo).
  3. Implementa la operación `tree_member(Element, Tree)`, que determina si un elemento esta en el árbol.
  4. Implementa `isotree(Tree1, Tree2)` que se cumple cuando `Tree1` y `Tree2` son isomorfos, i.e., uno puede obtenerse a partir del otro reordenando las ramas de sus sub-árboles.

2.2.4. Expresiones Simbólicas

Expresiones Simbólicas en Prolog

- La manipulación de números, listas y árboles binarios no es nada que no se pueda hacer con otros lenguajes (p.ej., en programación funcional).
- En esta sección analizaremos cuatro programas recursivos donde explotaremos la capacidad para el razonamiento simbólico de Prolog.
  1. Reconocer polinomios en un termino `X`:

     p.ej., $x^2 - 3x + 2$ es un polinomio en x.
  2. Calcular la derivada de una expresión con respecto de `X`:

     p.ej., `derivate(X,X,s(0))` esta en la semántica del programa.
  3. Resolver las torres de Hanoi.

     ...en lugar de una relación definiremos un esquema.
  4. Determinar la satisfacibilidad de una fórmula Booleana.

     ...es decir `satisfiable(X)` se cumple cuando `X` es Verdadera, y `invalid(X)` cuando `X` es Falsa.

2.2.4.1. Reconocer polinomios.

Expresiones Simbólicas I: `polynomial(Expression,X)`

- Para implementar este programa aprovecharemos la existencia de la definicion (infija) de los operadores binarios: `+`, `-`, `*`, `/`, y `^`:

```
% polynomial(Expresion,X): Expresion is a polynomial in X.
polynomial(X, X).
polynomial(Term, X) :- nat(Term).
polynomial(Term1 + Term2, X) :- polynomial(Term1, X), polynomial(Term2, X).
polynomial(Term1 - Term2, X) :- polynomial(Term1, X), polynomial(Term2, X).
polynomial(Term1 * Term2, X) :- polynomial(Term1, X), polynomial(Term2, X).
polynomial(Term1 / Term2, X) :- polynomial(Term1, X), nat(Term2).
polynomial(Term^N) :- polynomial(Term, X), nat(N).
```

- De modo que la consulta correspondiente al polinomio $x^2 - 3x + 2$ en x: `?- polynomial(x^s(s(0)) - s(s(s(0)))*x + s(s(0)), x)` tiene éxito.

Expresiones Simbólicas I: `polynomial(Expression,X)` **(Ejercicios)**

- Resuelve los siguientes ejercicios:
  1. Da una lectura declarativa de las clausulas de `polynomial/2`.
  2. Crea `my_polynomial/2` para en lugar de `+`, `-`, `*`, `/`, y `^` usar los funtores prefijos `p`, `s`, `m`, `d`, y `t`, el polinomio $x^2 - 3x + 2$ se representa como `p(s(t(x,s(s(0))),m(s(s(s(0))),x)),s(s(0)))`.
  3. Extiende `polynomial/2` y `my_polynomial/2`[5] para que (también) acepte la negación, p.ej., $-x^2 - 3x + 2$.
  4. Modifica `polynomial/2` para que acepte términos con fracciones de polinomios, p.ej., $x^3 + \dfrac{x^2 - 3x}{x - 3} + 2$.

2.2.4.2. Calcular la derivada.

Expresiones Simbólicas II: `derivative(Exp, X, Dif)`

- En este caso, el programa lógico para calcular la derivada no es más que un conjunto de reglas de derivada, escritas con la sintaxis correcta.

```
% derivative(Exp, X, Dif): Dif is the derivative of Exp with respect to X.
derivative(X, X, s(0)).
```

[5] La resta, `s/2`, se desambigua de sucesor, `s/1`, por la aridad. Para la negación usar `n/1`.

```
derivative(N, _, 0) :- nat(N).
derivative(X^s(N), X, s(N)*X^N).
derivative(sin(X), X, cos(X)).
derivative(cos(X), X, -sin(X)).
derivative(e^X, X, e^X).
derivative(log(X), X, s(0)/X).
derivative(F+G, X, DF+DG) :- derivative(F,X,DF), derivative(G,X,DG).
derivative(F-G, X, DF-DG) :- derivative(F,X,DF), derivative(G,X,DG).
derivative(F*G, X, F*DG+DF*G) :- derivative(F,X,DF), derivative(G,X,DG).
derivative(F/G,X,(G*DF-F*DG)/(G*G)) :- derivative(F,X,DF), derivative(G,X,DG).
```

- Como no hemos especificado como simplificar expresiones, la derivada de la expresión $3x+2$ es $3*1+0*x+0$ (que se simplificaría a 3).
  - Es decir, la consulta `?- derivative(s(s(s(0)))*x+s(s(0)),x,D)` genera la respuesta `D = s(s(s(0)))*s(0)+0*x+0`.
- En `derivative(Exp, X, Dif)` están definidas la derivada del producto y el cociente), sin embargo, no está la regla de la cadena.
- Resuelve los siguientes ejercicios:
  1. Define la regla de la cadena para la derivada de x^N y $sin(X)$.
  2. Si en lugar de `sin(X)` usamos el funtor `unary_term(sin,X)`:
     a) Define una única regla de la cadena.
     b) Adapta el resto de reglas a la nueva representación.

2.2.4.3. Torres de Hanoi.

Expresiones Simbólicas III: `hanoi(N,A,B,C,Moves)`

Enunciado:

En algún lugar escondido en los alrededores de Hanoi (un oscuro pueblo del Lejano Oriente cuando se contó la leyenda por primera vez) hay un monasterio. Los monjes están realizando una tarea que Dios les asignó cuando se creó el mundo: *"mover una torre de 64 discos dorados de una clavija a otra con la ayuda de una clavija auxiliar (con sólo dos reglas: (i) sólo se puede mover un disco a la vez, y (ii) nunca se puede colocar un disco más grande encima de otro más pequeño)"*.

En el momento en que completen la tarea, el mundo se derrumbará en el polvo.

Solución:

- Se sabe que para n discos la solución óptima requiere 2^n-1 movimientos.
- En el programa `hanoi(N,A,B,C,Moves)`: `Moves` es una secuencia de movimientos para `N` discos y 3 postes (`A`, `B` y `C`):

```
hanoi(s(0), A,B,C, [A - B]).
```

```
2 hanoi(s(N), A,B,C, Moves) :-
3     hanoi(N, A,C,B, Ms1),
4     hanoi(N, C,B,A, Ms2),
5     append(Ms1, [A - B|Ms2], Moves).
```

- El resultado de la consulta con $n = 3$, `?- hanoi(s(s(s(0))),a,b,c,Moves)`, es `Moves=[a-b,a-c,b-c,a-b,c-a,c-b,a-b]`.

- Aún cuando queramos interpretar `hanoi/5` como una relación, es evidente su semántica "operacional"[6] –dado `N` discos y tres postes `A`, `B`, y `C`, esperamos obtener en `Moves` la secuencia de movimientos...
- Igualmente las clausulas se entienden mejor leyéndolas de izquierda a derecha (de arriba a abajo), i.e., en un movimiento podemos mover un disco de `A` a `B` y para mover `s(N)` de `A` a `B` usando `C` como auxiliar:
  - Movemos `N` discos de `A` a `C` (ver línea 3).
  - Luego el disco en `A` se coloca en `B` (ver `[A - B]` en `append/3`).
  - Luego movemos los `N` discos de `C` a `B` (ver línea 4).
  - Finalmente concatenamos todos los movimientos en línea 5.
- (Ejercicio) Cambia la clausula base considerando que quedan `0` discos y comprueba si el programa funciona correctamente (explica porqué).

2.2.4.4. Satisfabilidad de fórmulas Booleanas

Expresiones Simbólicas IV: `satisfiable/1 & invalid/1`

- Una fórmula Booleana es un término definido como:
  - Las constantes *true* y *false* son fórmulas Booleanas.
  - Si X y Y son fórmulas Booleanas también lo son:
    $$X \wedge Y, X \vee Y, \text{ y } \neg X$$
    donde $\wedge$ y $\vee$ son operadores binarios infijos para la conjunción y la disyunción, y $\neg$ es el operador prefijo unario para la negación.
- Primero definiremos los operadores `/\`, `\/` y `~` respectivamente:

```
1 :- op(200, fy, ~).
2 :- op(400, xfy,[/\, \/]).
```

  Donde `op(Precedence, Type, Name)` declara que `Name` (puede ser una lista de nombre) es un operador de tipo `Type` (la `f` indica la posición del funtor) con precedencia `Precedence`.[7]

[6] Profundizaremos en la semántica operacional de Prolog en el Tema 3. Adicionalmente se muestra una implementación alternativa (más declarativa) basada en Answer Set Programming (ASP).

[7] Detalles sobre el uso de `op/3` disponibles en la documentación de Ciao y Swi.

- El programa para determinar si una fórmula Booleana es verdadera o falsa requiere de dos predicados mutuamente recursivos:

```
1  satisfiable(true).
2  satisfiable(X /\ Y) :- satisfiable(X), satisfiable(Y).
3  satisfiable(X \/ Y) :- satisfiable(X).
4  satisfiable(X \/ Y) :- satisfiable(Y).
5  satisfiable(~X) :- invalid(X).
6  invalid(false).
7  invalid(X \/ Y) :- invalid(X), invalid(Y).
8  invalid(X /\ Y) :- invalid(X).
9  invalid(X /\ Y) :- invalid(Y).
10 invalid(~X) :- satisfiable(X).
```

- Puede aplicarse a fórmulas Booleanas con variables...

  i.e., es mas potente de lo que parece ;-)

- (Ejercicio) Implementa `boolean/1`, constructor de fórmulas Booleanas.

Universidad
Rey Juan Carlos

Cápitulo 3

Programación lógica avanzada

3.1. Semántica operacional de Prolog.

3.1.1. Modelo computacional

Modelo computacional: SLD + busqueda en profundidad

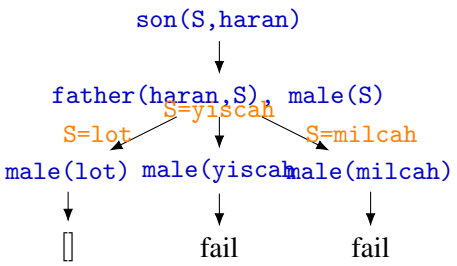
- Para que Prolog sea considerado un lenguaje de programación fijamos:
  - El orden en el que se seleccionan las cláusulas dentro del programa.
  - La estrategia de resolución de los objetivos en el resolvente.
- Las clausulas se evalúan de arriba a abajo (top-down) y los objetivos de izquierda a derecha (añadiéndolos a la izq. del resolvente).
- Este modelo computacional se corresponde con la búsqueda en profundidad (con backtracking) del árbol de derivación SLD.
- Por lo tanto la consulta `?-son(S,haran)` generan distintos árboles de derivación dependiendo del orden de los literales en el predicado `son/2`.

```
% Version a
son(S,P) :- father(P,S), male(S).
% Version b
son(S,P) :- male(S), father(P,S).
```

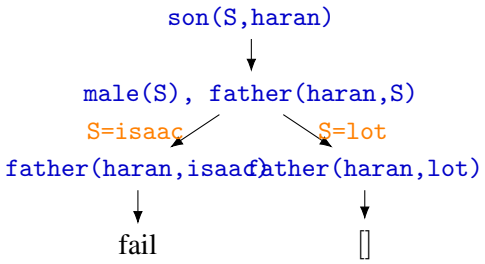
- Supongamos la siguiente base de datos:

```
father(abraham, isaac).        male(isaac).
father(haran, lot).            male(lot).
father(haran, milcah).         female(milcah).
father(haran, yiscah).         female(yiscah).
```

Árbol de búsqueda versión a:



Árbol de búsqueda versión b:



- **Conclusión 1**:
  - El orden de las cláusulas en el programa.
  - Asi como el orden de los objetivos en el cuerpo de las cláusulas.

  puede influir en:
  - El orden en el que se encuentran (o no) las respuestas.
  - La terminación (o no) de las consultas.
- De hecho, el modelo computacional de Prolog no es completo: no asegura encontrar soluciones:
  - Hay una rama infinita a la izq. de una rama con respuestas.
  - Como ya hemos explicado esto se puede resolver usando busqueda en anchura (paquete `sr/bfall` (Cabeza, Carro y Hermenegildo)).
- **Conclusión 2**: Este modelo computacional nos permite definir una semántica operacional (procedimental).

Modelo computacional: Semántica operacional

- Como ya indicamos, además de las semánticas lógicas, podemos ver la ejecución en Prolog como un conjunto de procedimientos:
  - La consulta `Q` es un conjunto de objetivos $\texttt{Q}_1\texttt{,}\ \texttt{Q}_2\texttt{,}\ \ldots\texttt{,}\ \texttt{Q}_m$.
  - Para probar $\texttt{Q}_i\texttt{,}\ \ldots\texttt{,}\ \texttt{Q}_m$:
    - Encuentra una cláusula $\texttt{H :- B}_1\texttt{,}\ \ldots\texttt{,}\ \texttt{B}_n$ tal que $\texttt{Q}_i$ y `H` unifican (con un umg).
    - Bajo la substitución (del umg) de las variables en la cláusula, probar (recursivamente) $\texttt{B}_1\texttt{,}\ \ldots\texttt{,}\ \texttt{B}_n\texttt{,}\ \texttt{Q}_{i+1}\texttt{,}\ \ldots\texttt{,}\ \texttt{Q}_m$.

- Si no queda nada por probar, la ejecución tiene éxito.
- Si no hay más cláusulas que coincidan, la ejecución falla.

3.1.2. Operador corte

Operador corte

- Basándonos en la semántica operacional de Prolog definimos un operador, llamado corte, `!/0` que permite cortar la búsqueda.
  - Este corte permite reducir el tamaño del árbol de búsqueda.
  - Sin embargo, hay que distinguir entre cortes verdes y rojos.
- Los cortes verdes: no afectan al sentido declarativo del programa, podan ramas inútiles, redundantes o infinitas.
- Los cortes rojos: Evitan soluciones erróneas podando ramas que conducen a éxitos no deseados.

Ejemplo de corte verde:

```
% Avoid redundancies
is_father(P) :- father(P,S), !.

member(X, [X|_]) :- !.
member(X, [_|L]) :- member(X,L)
```

Ejemplo de corte rojo:

```
% Prune search of ?- minimum(s(0),s(s(0)),X)
minimum(X,Y,X) :- leq(X,Y), !.
minimum(X,Y,Y).
% However, ?- minimum(s(0),s(s(0)),s(s(0)))
%                       succeeds incorrectly
```

- Otra de las ventajas del corte es su capacidad para "eliminar" puntos de elección (evitando backtracking). P.ej., la siguiente versión de `polynomial(Expression,X)` es más eficiente que la del Tema 2:
  - Observa como se introduce el corte en los hechos.

```
% polynomial(Expresion,X): Expresion is a polynomial in X.
polynomial(X, X) :- !.
polynomial(Term, X) :- nat(Term), !.
polynomial(Term1 + Term2, X) :- !, polynomial(Term1, X), polynomial(Term2, X).
polynomial(Term1 - Term2, X) :- !, polynomial(Term1, X), polynomial(Term2, X).
polynomial(Term1 * Term2, X) :- !, polynomial(Term1, X), polynomial(Term2, X).
polynomial(Term1 / Term2, X) :- !, polynomial(Term1, X), nat(Term2).
```

```
8 polynomial(Term^N) :- !, polynomial(Term, X), nat(N).
```

3.1.3. Negación

Negación por fallo: !/0 + fail/0 + call/1

- Una de las principales limitaciones de la semántica de Prolog (basada en las cláusulas de Horn) es la ausencia de negación. P.ej.:

  $\forall x\ (\ flies(x) \leftarrow bird(x) \wedge \neg abnormal\_bird(x)\ )$
  $\forall x\ (\ abnormal\_bird(x) \leftarrow penguin(x)\ )$

- (Ejercicio) Explica porque la primera no es una cláusula de Horn.
- Para resolver esta situación Prolog dispone del operador `\+` que implementa la Negación por Fallo (semántica SLDNF (Clark 1978)).
  - El objetivo `\+ G` tiene éxito si `G` falla y viceversa.
  - Pero no soporta llamadas con variables ni recursion.
- La implementación de `\+` usa: el corte `!/0`, el predicado `fail/0`, que provoca el fallo, y `call/1`, que invoca el predicado que recibe:

```
1 \+ Goal :- call(Goal), !, fail.
2 \+ Goal.
```

Negación por fallo: Ejemplo y limitaciones

- Ejemplo: la consulta `?- flies(X)` para el siguiente programa:

```
1 flies(X) :- bird(X), \+ abnormal_bird(X).
2 abnormal_bird(X) :- penguin(X).
3 bird(tweety).            penguin(tweety).
4 bird(sam).
```

  devuelve como única respuesta `X=sam` porque `tweety` es un pingüino.
- Limitación I: cambiando la posición del objetivo negado en línea 1:

```
1 flies(X) :- \+ abnormal_bird(X), bird(X).
```

  la consulta falla, porque `call(abnormal_bird(X))` tiene éxito.
- Limitación II: Programas con negación no estratificada entra en bucle:

```
1 flies(tweety) :- not penguin(tweety).
2 penguin(tweety) :- not flies(tweety).
```

  además hay dos modelos (en uno vuela y en otro es un pingüino).

3.1.3.1. Answer Set Programming

ASP & s(CASP): Alternativas a la Negación por fallo

- **Answer Set Programming (ASP)**: Basada en la semántica de modelos estables (Gelfond y Lifschitz 1988).
  - El programa anterior:

```
1 flies(tweety) :- not penguin(tweety).
2 penguin(tweety) :- not flies(tweety).
```

    Tiene dos modelos stables:

    - { flies(tweety) }
    - { penguin(tweety) }

- Implementaciones disponibles:
  clingo: requiere realizar un grounding de las variables del programa.[1]
  s(CASP): evaluación *Goal-directed* de ASP, implementado en Prolog.[2]

3.1.4. Memorización

Memorización: Evita bucles y recomputaciones

- Aún cuando el conjunto de soluciones es finito aplicar SLD puede generar una derivación infinita o respuestas redundantes:

```
1 edge(1, 2).        edge(2, 1).        edge(2, 3).
2
3 reach(X, Y) :- edge(X, Z), reach(Z, Y).
4 reach(X, Y) :- edge(X, Y).
```

- En esta sección veremos algunas técnicas para evitar entrar en una derivación infinita y/o reducir el espacio de búsqueda:
  1. Memorización: recordar información de los estados visitados:
     a) Añadiendo un argumento.
     b) Usando técnicas avanzadas de búsqueda como tabulación.
  2. Uso de agregados para agrupar el conjunto de soluciones parciales durante la búsqueda (Arias y Carro 2019) (ver pág. 3.3).

[1] Playground online de clingo disponible en https://potassco.org/clingo/run/
[2] Playground online de s(CASP) disponible en https://ciao-lang.org/playground/scasp.html

3.1.4.1. Acumulador

Memorización: 1a) Acumulador

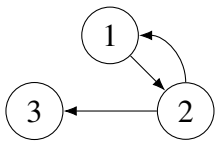
- Para no entrar en bucles guardaremos los vértices visitados en una lista:

```
reach(X,Y) :-
      reach_aux(X,Y,[X]).
reach_aux(X,Y,Visitados) :-
      edge(X,Z),
      \+ is_member(Z,Visitados),
      reach_aux(Z,Y,[Z|Visitados]).
reach_aux(X,Y,_) :-
      edge(X,Y).

is_member(X,[X|_]).
is_member(X,[_|T]) :-
      is_member(X,T).
```

- Dicha lista es el tercer argumento del predicado `reach_aux/3`.
- y con `\+ is_member(Z,Visitados)` nos aseguramos que `Z` no ha sido visitado.

- (Ejercicio) Dado el siguiente grafo y la consulta `?- reach(1,X)`, indica las respuestas esperadas:



Memorización: 1a) Acumulador (Ejercicio)

1. Explicar porque la siguiente regla no representa una cláusula de Horn.

```
reach_aux(X,Y,Vs) :- edge(X,Z), \+ is_member(Z,Vs), reach_aux(Z,Y,[Z|Vs]).
```

2. Dado el siguiente programa en Prolog

```
reach(X,Y) :-
   reach_aux(X,Y,[X]).
reach_aux(X,Y,Visitados) :-
   edge(X,Z),
   \+ is_member(Z,Visitados),
   reach_aux(Z,Y,[Z|Visitados]).
reach_aux(X,Y,_) :-
   edge(X,Y).
```

```
edge(1,2).
edge(2,1).
edge(2,3).

is_member(X,[X|_]).
is_member(X,[_|T]) :-
   is_member(X,T).
```

   modificarlo para que no permita ciclos, es decir, la consulta `?- reach(X,X)` falla y `?- reach(1,Y)` solo devuelve `Y=2` y `Y=3`.

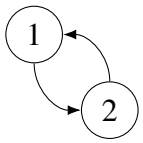
3.1.4.2. Tabulación

Memorización: 1b) Tabulación

- La tabulación (**tabling** en inglés, ver Tamaki y Sato (1986) y Warren (1992)) es una estrategia de búsqueda en profundidad para Prolog que emula la búsqueda en anchura:
  - Memoriza las respuestas de la primera ocurrencia de un objetivo.
  - Suspende la evaluación de objetivos repetidos.

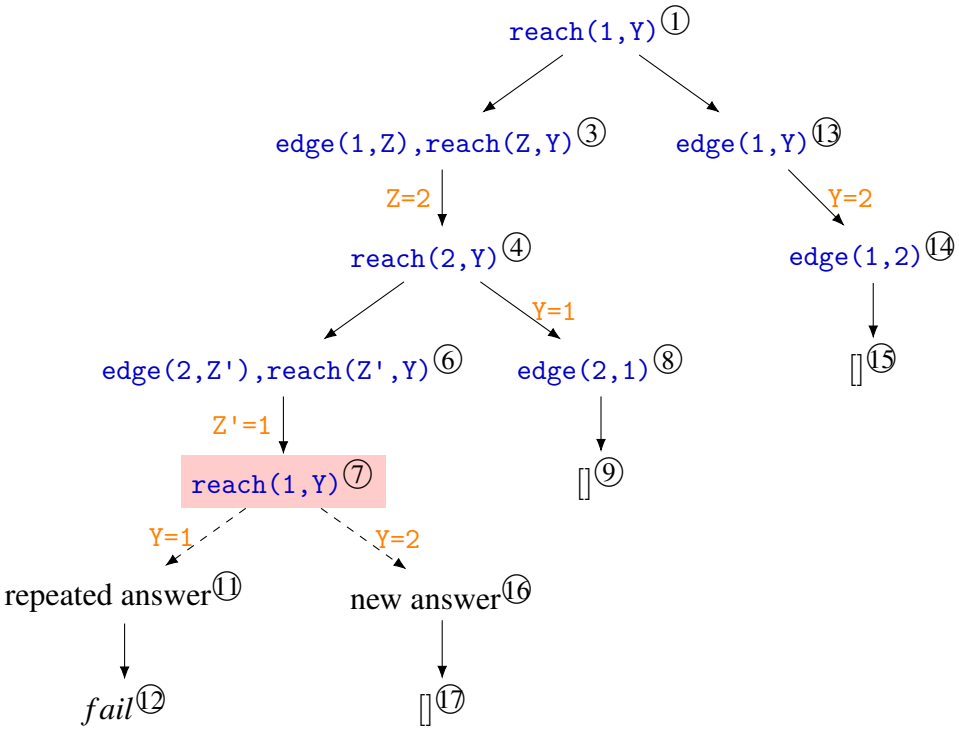
    ... los cuales re-arrancan para consumir dichas respuestas.
- Evaluaremos el siguiente programa usando tabulación:

```
1  :- use_package(library(tabling)).
2  :- table reach/2.
3
4  edge(1,2).
5  edge(2,1).
6  reach(X,Y) :-
7      edge(X,Z),
8      reach(Z,Y).
9  reach(X,Y) :-
10     edge(X,Y).
```



- Con `:- table` (línea 2) definimos los predicados a tabular.

Memorización: 1b) Tabulación: Evaluación de `?- reach(1 ,Y)`



② Guardamos `reach(1,Y)` en la tabla.
⑦ Suspendemos la evaluación de `reach(1,Y)`.
⑨ Guardamos `Y=1` para `reach(2,Y)` y `reach(1,Y)`.
⑪ Consume `Y=1`, pero no genera nuevas respuestas.
⑮ Guardamos `Y=2` y rearrancamos.
⑰ Esta vez genera nueva respuesta para `reach(1,Y)`.
⑱ No hay mas ramas ni respuestas. FIN.

Objetivo	Respuestas
`reach(1,Y)`②	`Y=1`⑩
	`Y=2`⑮
`reach(2,Y)`⑤	`Y=1`⑨
	`Y=2`⑰

Memorización: 1b) Tabulación: Ventajas

- Además de mejorar terminación, la memorización evita re-computar estados ya visitados, reduciendo el tiempo de ejecución.
- Ejemplo: La sucesión de Fibonacci tiene aplicaciones en ciencias de la computación, teoría de juegos, análisis bursátil, etc.
- Dado `fib(N,F)`, que verifica que `F` es el `N`-esimo término de la sucesión de Fibonacci.
- ...al tabular `fib/2` pasamos de complejidad O(2^n) a complejidad lineal O(n).
- Implementación de `fib(N,F)` con tabulación.[3]

```
1 :- use_package(library(tabling)).
2 :- table fib/2.
3
4 fib(0, 0).
5 fib(s(0), s(0)).
6 fib(s(s(N)), F) :-
7       fib(s(N), F1),
8       fib(N, F2),
9       plus(F1,F2,F).
```

- (Ejercicio) Analiza y explica la mejora en complejidad.

3.1.5. Aritmética

Aritmética: Usando funciones del sistema

- Una vez hemos decidido explotar la semántica operacional de Prolog, podemos delegar ciertas tareas en funciones del sistema:
  1. El predicado infijo `X is Exp`, donde `Exp` es una expresión con operadores como `+, -, *, mod, abs, log, sqrt, ...` [4] que se evalúa y el resultado

[3] La declaración `use_package` no es necesaria en Swi (online/local) pero si en Ciao (solo funciona en local).
[4] Detalles sobre los operadores aritméticos disponibles en la documentación de Ciao y Swi.

se unifica con `X` (que puede ser una variable):

- `?- 8 is 3+5` tiene éxito porque 3+5 se evalúa a 8.
- `?- 3+5 is 3+5` falla porque el 3+5 de la izq. no unifica con 8.
- `?- X is 3+5` devuelve `X=8` (el uso previsto para `is/2`).

2. Los operadores de comparación, `>`, `<`, `>=`, `=<`, `=:=`, y `=\=`, requieren que ambas expresiones se puedan evaluar (p.ej., `?- 3-2 < 6`), en caso contrario genera error de instanciación (p.ej., `?- N < 1`).

3.1.5.1. Limitaciones y CLP

Aritmética: Limitaciones con las funciones del sistema

- Ahora podemos escribir los predicados aritméticos usando las funciones del sistema. P.ej., `plus/3` seria:

```
1   plus(X,Y,Z) :- Z is X + Y.
```

sin embargo, ahora la consulta `?- plus(3,X,8)` genera un error de instanciación, en lugar del esperado `X=5`.

- **Solución:** Afortunadamente existe una extensión de Prolog llamada restricciones (ver Tema 3.4, pág. 3.4.1) que permite definir ecuaciones aritméticas. P.ej., usando `clp(Q)` en Ciao tendríamos:

```
1  :- use_package(clpq).
2  plus(X,Y,Z) :- Z .=. X + Y.
```

donde la consulta `?- plus(3,X,8)` si devuelve la respuesta esperada `X=5`.

3.1.6. Inspeccionar estructuras, entrada/salida...

Inspeccionar estructuras, entrada/salida, etc

- Comparación de términos:
  - Unifican: `A = B`, se cumple si `A` y `B` unifican. `p(X,f(Y)) = p(a,Z)`
  - No unifican: `A \= B`, se cumple si `A` no unifica con `B`. `p(X,X) \= p(f(a),f(b))`
  - Idénticos: `A == B`, se cumple si son idénticos. `p(X) == p(X)`
  - No idénticos: `A \== B`, se cumple si no son idénticos. `p(X) \== p(Y)`
- Transformación de términos
  - Término a lista: `T =.. L`, se cumple si el primer elemento

de `L` es el functor de `T` y el resto `L` son sus argumentos. `p(a,Y) =.. [p,a,Y]`.

- Funtor y aridad: `functor(T,F,A)` se cumple si `F` es el functor del término `T` y `A` es su aridad. `functor(p(a,Y),p,2)`
- N-esimo argumento: `arg(N,T,A)` se cumple si `A` es el argumento del término `T` que ocupa el lugar `N`. `arg(1,p(a,Y),a`
- Constante a ASCII: name(A,L) se cumple si `L` es la lista de códigos ASCII de la constante `A`. `name('Hello',[72,101,108,108,111])`

- Comprobación sobre tipos de términos.
  - `var(T)` se verifica si `T` es una variable. `var(X)`
  - `atom(T)` se verifica si `T` es un átomo. `atom('Hello')`
  - `number(T)` se verifica si `T` es un número. `number(1)`
  - `compound(T)` se verifica si `T` es un término compuesto. `compound(p(a,Y))`

  P.ej.: tratemos de hacer `plus/3` tan declarativo como podamos usando `is/2`.

```
1 plus(X,Y,Z) :- number(X), number(Y), Z is X + Y.
2 plus(X,Y,Z) :- var(X), X is Z - Y.
3 plus(X,Y,Z) :- var(Y), Y is Z - X.
```

- Manipulación de ficheros (lectura/escritura).
  - Prolog proporciona predicados básicos para el manejo de archivos y streams, para realizar entradas/salidas en ellos (ver documentación correspondiente en Ciao y Swi)

    P.ej.: `current_output(StreamA)`, `open(File,Mode,StreamB)`, `set_output(StreamB)`, "escribes en el archivo", `close(StreamB)`, `set_output(StreamA)`.

- Para imprimir en el top-level (o en ficheros) tenemos:
  - `display(Term)`: imprime `Term` por el stream actual.[5]
  - `nl`: imprime un salto de línea.
  - `format(Format,Arguments)`: imprime (simulando el `printf` de `stdio.h`) la lista de `Arguments` según el formato `Format`.

    P.ej., `?- A=world, display('Hello '), display(A), nl` y `?- A=world, format('Hello ~p\n',[A])`, son equivalentes, ambas imprimen `Hello world`.

- Para leer del top-level (o de ficheros) hay predicados que permiten leer términos y cláusulas directamente. Ver documentación en Ciao y Swi.
  - ...también hay predicados para leer caracteres (`get_char(X)`) o incluso by-

[5] Esta implementado usando una versión con un argumento extra, i.e., `current_output(S), display(S,Term)`.

tes (`get_byte(X)`), p.ej., el char `A` es el byte `65`.

3.2. Programación Procedimental.

3.2.1. Algoritmos de ordenamiento

Algoritmos de ordenamiento I: Especificación lógica

- Un procedimiento naive para ordenar una lista `Xs` consiste en encontrar una permutación `Ys` que este ordenada:

```
1 sort(Xs,Ys) :- permutation(Xs,Ys), ordered(Ys).
```

  - Este predicado explota la semántica operacional de Prolog y es un ejemplo del paradigma **genera y prueba**.[6]
  - Comprobar si una lista está ordenada es trivial:

```
1 ordered([]).
2 ordered([X]).
3 ordered([X,Y|Xs] ) :- X =< Y, ordered(Xs).
```

- (Ejercicio) Implementar `permutation`, un predicado que dada una lista `Xs` genera todas las permutaciones posibles.
  - Pista: Define `select(X, HasX, OneLessX)`, donde la lista `OneLessX` es el resultado de elimina una ocurrencia de `X` de la lista `HasX`.

Algoritmos de ordenamiento II: Divide y vencerás

- En este ejemplo implementaremos `quicksort/2` usando la estrategia de divide y vencerás...

  > `Ys` es una versión ordenada de `[X|Xs]` si `Litttles` y `Bigs` son el resultado de particionar `Xs` según `X`; `Ls` y `Bs` son el resultado de ordenar recursivamente `Littles` y `Bigs`; e `Ys` es el resultado de concatenar `[X|Bs]` a `Ls`.

  es decir la regla recursiva sería:

```
1 quicksort([X|Xs], Ys) :-
2       partition(Xs, X, Littles, Bigs),
3       quicksort(Littles, Ls),
4       quicksort(Bigs, Bs),
5       append(Ls, [X|Bs], Ys).
```

- (Ejercicio) Define la cláusula base de `quicksort/2` y el predicado `partition/4`, según la especificación esperada.

[6] Por otro lado la programación lógica con restricciones (pág. 3.4.1) representa el paradigma **restringe y genera**.

3.2.2. Meta-interprete

Meta-interprete en Prolog, para Prolog

- Un meta-interprete para un lenguaje, es un interprete escrito en dicho lenguaje. Para ello definiremos el predicado `solve/1`, tal que:
  - Las reglas las definiremos como `rule(H,[B|Bs])`.
  - Los hechos serán de la forma `rule(H,[])`.
  - Una consulta `[Q|Qs]` se invocarán con `?- solve([Q|Qs])`.

```
1 solve([]).
2 solve([X|Xs]) :- rule(X,[]), solve(Xs).
3 solve([X|Xs]) :- rule(X,[B|Bs]), append([B|Bs],Xs,Xss), solve(Xss).
```

- (Ejercicio) Escribe un programa usando `rule/2` y comprueba si `solve/1` lo resuelve correctamente.
- (Ejercicio) Extiende el meta-interprete para que soporte el operador aritmético `is/2` (Pista, delega en Prolog la evaluación de `is/2`).

3.2.3. Trucos eficientes

Trucos eficientes

- Eficacia se refiere a (i) tiempo o (ii) memoria.
- Implica usar (i) buenos algoritmos y (ii) estructuras de datos adecuadas.

1. **Las listas** ocupan mas memoria (útiles si no se conoce el número de elementos) y acceso a los datos es mas lento (mantener ordenadas).
   - Si la cantidad de elementos es conocida usar **funtores**.
2. Prolog permite construir **estructuras de datos avanzadas** con facilidad: árboles ordenados, estructuras anidadas, etc.
3. Usas la unificación para manipular los datos:

```
1 three_elements_a(L) :- length(L,N), N=3.
2 three_elements_b([_,_,_]).   % Better
```

4. Evita programas no-deterministas:
   - Eliminando puntos de elección:

```
1 member(X, [X|_]) :- !.
2 member(X, [_|Xs]) :- member(X,Xs).
```

   - Implementando programas deterministas de manera determinista:

```
plus_a(X,Y,N) :- var(X), var(Y), between(0,N,X), between(0,N,Y), N is X + Y.
plus_b(X,Y,N) :- var(X), var(Y), between(0,N,X), Y is N - X.
% Better %
```

- Listas ordenadas + unificación:

```
equal_sets_a(S1, S2) :- \+ (member(X,S1), \+ member(X,S2) ),
                        \+ (member(X,S2), \+ member(X,S1) ).
equal_sets_a(S1, S2) :- sort(S1,Sort), sort(S2,Sort).
% Better %
```

5. Ordenar los predicados para podar la búsqueda (más detalles en pág. 3.4.1).

6. Los interpretes de Prolog indexan por el primer argumento:
   - Al compilar crean una tabla mirando la cabeza de las cláusulas de cada predicado.
   - Durante la ejecución solo considera cláusulas compatibles con el primer argumento.

```
greater_a(_, []).
greater_a(X, [Y|Ys]) :- X > Y, greater_b(X,Ys).
greater_b([], _).                                   % Better %
greater_b([Y|Ys], X) :- X > Y, greater_b(X,Ys).     %
```

7. Transformar recursión en iteración (recursión por cola).
   - Cuando la llamada recursiva es el último objetivo y no hay alternativas.

```
sum_a([], 0).
sum_a([N|Ns], Sum) :- sum_a(Ns, Ss), Sum is N + S1.
sum_b(L, Out) :- sum_inter(L, 0, Out).
% Better %
sum_iter([], In, In).
%
sum_iter([N|Ns], In, Out) :- Mid is N + In, sum_iter(Ns, Mid, Out).
%
```

3.2.4. Recolección de soluciones

Lógica de 2^o Orden I: Recolección de soluciones

- El no-determinismo de Prolog provoca que un objetivo pueda generar varias respuestas y necesitemos agregarlas para su manipulación.
- Hay tres predicados predefinidos (básicos) para recolectar en una lista resultados de un objetivo (o conjunto de objetivos).[7]

[7] Consulta la documentación de Ciao y Swi para más detalles.

1. `bagof(Template, Goal, Bag)`.

   Unifica `Bag` con las alternativas de `Template`. Si `Goal` tiene variables libres además de la que comparte con `Template`, `bagof/3` retrocederá sobre las alternativas de estas variables libres, unificando `Bag` con las correspondientes alternativas de `Template`. La construcción `Var^Goal` indica a `bagof/3` que no debe enlazar `Var` en `Goal`.

   Nota: `bagof/3` falla si `Goal` no tiene soluciones.

- (Ejemplo) Uso de `bagof/3` dado el programa:

```
1 foo(a,b,c).       foo(a,b,d).       foo(b,c,f).
2 foo(b,c,e).       foo(c,c,f).
```

  La consulta `?- bagof(C, foo(A,B,C), Cs)` retrocede sobre `A` y `B`:

```
1 A = a, B = b, Cs = [c,d] ?;
2 A = b, B = c, Cs = [f,e] ?;
3 A = c, B = c, Cs = [f] ?;
```

  ...mientras que `?- bagof(C, A^foo(A,B,C), Cs)` solo sobre `B`:

```
1 B = b, Cs = [c,d] ?;
2 B = c, Cs = [f,e,f] ?;
```

2. `setof(Template, Goal, Set)`.

   Equivale a `bagof/3`, pero ordena `Set` utilizando `sort/2` para obtener una lista ordenada de alternativas sin duplicados.

   La consulta que `?- setof(C, A^foo(A,B,C), Cs)` genera:

```
1 B = b, Cs = [c,d] ?;
2 B = c, Cs = [e,f] ?;
```

3. `findall(Template, Goal, List)`.

   Crea una lista de las instancias que `Template` obtiene sucesivamente en el backtracking sobre `Goal` y unifica el resultado con `List`. Tiene éxito con una lista vacía si `Goal` no tiene soluciones. Es equivalente a `bagof/3` con todas las variables libres cuantificadas existencialmente pero no falla cuando `Goal` no tiene soluciones.

   La consulta que `?- findall(C, foo(A,B,C), Cs)` genera:

```
1 Cs = [c,d,f,e,f] ?;
```

3.2.5. Predicados de orden superior

Lógica de 2^o Orden II: Predicados de orden superior

- Predicados de orden superior reciben como argumento otro predicado.

    - Implementados usando `call/1`, ya mencionado en pág. 3.1.3.
- Los más conocidos son `map/3`, `filter/3`, `foldl/4`, y `foldr/4`.
- (Ejemplo) Implementación de `map/3`:

```
1 map(_,[],[]).
2 map(P,[X|Xs],[Y|Ys]) :- Goal =.. [P,X,Y], call(Goal), map(P,Xs,Ys).
```

1. (Ejercicio) Implementa recursivamente `filter/3`, `foldl/4`, y `foldr/4`. Compara sus complejidades y plantea implementaciones alternativas.
2. (Ejercicio) Implementa `maplist/2`, una versión de `map/3` para $n-1$ listas. ¿Porque solo tiene dos argumentos?

3.3. Agregación de respuestas dinámica

Lógica de 2^o Orden: Agregación dinámica

- Una función de agregación calcula un único resultado a partir de elementos de datos separados:
  - *mínimo entre números*. $\min(\{2,4,7\}) = 2$
  - *conjunto de respuestas* a una consulta. $\text{set}(x \mid p(X)) = \{a,b,c\}$
  - *suma* de una lista de números. $\text{add}(\{\{1,2,5,2\}\}) = 10$
- Existen diferentes estrategias de evaluación, donde la eficacia y la terminación son un reto.
  - Naïve: Recoge los elementos (`findall/3`) y calcula el agregado.
  - Incremental: calcular (algunos) agregados *sobre la marcha*.

Evaluación incremetal usando tabulación ( )

- Cada nueva respuesta se agrega con el anterior agregado y se guarda.
- Incrementa eficiencia y terminación, reduciendo memoria necesaria.

3.3.1. Motivación

Lógica de 2^o Orden III: Agregados, motivación

- Distancia entre nodos en un grafo. Modelo **infinito** para un grafo cíclico.

```
1 edge(a,b,2).
2 edge(a,b,4).
3 edge(b,a,3).
4
5 dist(X,Y,D) :- edge(X,Y,D).
6 dist(X,Y,D) :- edge(X,Z,D1), dist(Z,Y,D2), D is D1 + D2.
```

- La distancia más corta entre nodos. Modelo **finito** también en grafos cíclicos.

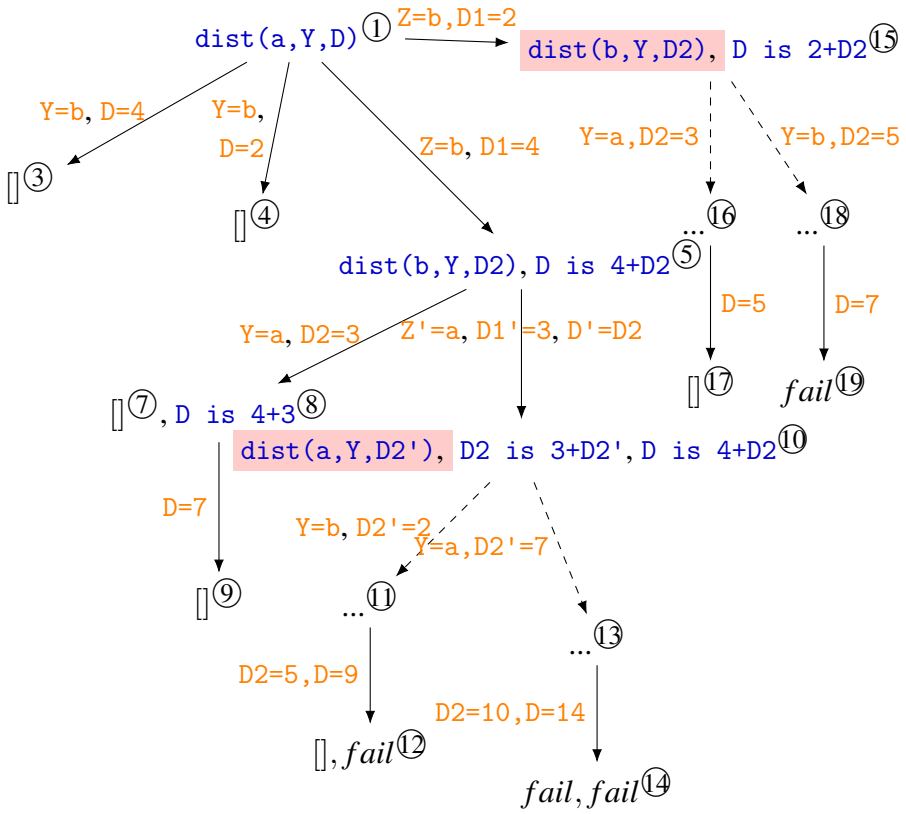
```
{ dist(a,a,5), dist(b,b,5), dist(a,b,2), dist(b,a,3) }
```

La implementación usando `findall/3` no termina

- Solución: *minimizar* el argumento `D` en `dist(X,Y,D)`.
  - Usando la directiva `:- agg_entail dist(_,_,=<).`

3.3.2. Agregación de `?- dist(a, Y, D)`

Lógica de 2^o Orden III: Agregación de `?- dist(a, Y, D)`



③ Guardamos `Y=b, D=4` en la tabla.
④ Sustituimos la respuesta porque $min(4,2) = 2$.
⑦ Guardamos la respuesta de `dist(b,Y,D)`.
⑪ Consumimos respuestas de `dist(a,Y,D)`.
⑫ Se descartan respuestas mayores a la existente.
⑰ Sustituimos la respuesta porque $min(7,5) = 5$.

Objetivo	Respuestas
`dist(a,Y,D)` ②	`Y=b, D=4` ③ `Y=b, D=2` ④ `Y=a, D=7` ⑨ `Y=a, D=5` ⑰
`dist(b,Y,D)` ⑥	`Y=a, D=3` ⑦ `Y=b, D=5` ⑫

3.3.3. Codificación de Agregados en Ciao

Lógica de 2^o Orden III: Codificación de Agregados en Ciao

Ejemplo de codificación de **entailment-based** agregados.

Aggregate	Code for entailment checking
minimum among numbers	`=<`
maximum among numbers	`>=`
enclosing interval	`interval(A1-A2,B1-B2) :- A1 =< B1, A2 >= B2.`
containing set	`set(A,B) :- ord_subset(B,A).`
index / variant	`variant`
answer subsumption	`sub(A,B) :- instance(B,A).`
Pareto-frontier(Op)	`frontier(Op,As,Bs) :- maplist(Op,As,Bs).`

n Pareto-frontier(Ops)	`n_frontier([],[],[]).`
	`n_frontier([Op|Ops],[A|As],[B|Bs]) :-`
	`  Op(A,B),`
	`  n_frontier(Ops,As,Bs).`

Ejemplo de codificación de **join-based** agregados.

Aggregate	Code for entailment checking and join

least upper bound	`lub(A,B) :- lub(A,B,A).`
	`lub(a,b,c).  lub(a,c,c).  lub(a,d,d).`
	`lub(b,a,c).  lub(b,c,c).  lub(b,d,d).`
	`lub(c,d,d).  lub(X,X,X).`

widest enclosing interval	`interval(A1-A2,B1-B2,C1-C2) :-`
	`  (A1=<B1 -> C1=A1; C1=B1),`
	`  (A2>=B2 -> C2=A2; C2=B2).`

set	`set(A,B,C) :- ord_union(A,B,C).`

Ejemplo de codificación de **non-lattice** agregados.

Aggregate	Code for entailment checking and join
first or nt	`first(_,_) :- true.`
all solutions	`all(_,_) :- fail.`
threshold(Epsilon)	`threshold(Epsilon,A,B) :- A < Epsilon * B.`
last	`last(_,_) :- fail.` `last(_,B,B).`
add	`add(_,_) :- fail.` `add(A,B,C) :- C is A + B.`
multiplication	`mult(_,_) :- fail.` `mult(A,B,C) :- C is A * B.`

3.3.4. Ejemplos de uso de agregados

Lógica de 2^o Orden III: Ejemplos de uso de agregados

1. El problema de los `Juegos`. ICLP 2015 LP/CP contest (ALP 2015).
   - Tienes que jugar `n` juegos al menos una vez. Algunos son mas divertidos que otros.
   - Tienes que administrar tus `Tokens` para obtener la mayor `Felicidad` de los juegos:

     `reach(JuegoA,JuegoB,Tokens,Felicidad):- ....`
   - Maximiza `Tokens` y `Felicidad` con el agregado `>=`.

     `:- agg_entail reach(_,_,>=,>=)`
     - **No** evalúa estados peores que otros ya evaluados.
     - Reduce el espacio de búsqueda!

   Run time (ms) comparison for *Games* with different scenarios.

	Prolog	**Tabling**	**Agregados**
game_data_01	8062.49	14.66	**2.89**
game_data_02	$>$ 5 min.	37.59	**4.87**
game_data_03	$>$ 5 min.	1071.26	**19.61**
game_data_04	$>$ 5 min.	4883.00	**23.21**

2. **Camino Aleatorio** (Random walk):
   Probabilidad `P` de alcanzar un nodo `N` desde `a`, considerando caminos aleatorios de `a` a `N`:
   - Es la suma (`add`) de las probabilidades de transición de **todos** los caminos.
   - Sin embargo, ciclos en un grafo pueden recorrerse un número ilimitado de

veces...

...descartemos caminos cuya contribución sea inferior a cierto umbral(`thr/1`).

$$P(b) = 0.3$$

$$\left.\begin{array}{ll} P_0(d) = 0.7 * 0.8 & = 0.56 \\ P_1(d) = 0.7 * 0.2 * 0.8 & = 0.112 \\ P_2(d) = 0.7 * 0.2 * 0.2 * 0.8 & = 0.0224 \\ \quad \ldots & \\ P_n(d) = 0.7 * 0.2^n * 0.8 & = \ldots \end{array}\right\} \quad P(d) = 0.7$$

Codificación usando los non-lattice agregados `add` **y** `thr/1`

```
1  :- use_package(tclp_aggregates).
2  :- agg_join reach(_,add).
3  :- agg_entail path(_,_,thr(0.001)).
4
5  add(_,_) :- fails.
6  add(A,B,New) :- New is A + B.
7  thr(Epsilon,A,B) :- A < Epsilon * B.
8  reach(N,P) :- path(a,N,P).
9
10 path(X,Y,P) :- edge(X,Y,P).
11 path(X,Y,P) :- edge(X,Z,P1),
12                path(Z,Y,P2),
13                P is P1 * P2.
```

`?- reach(d,P)` returns `P=0.699776`. Es una buena aprox. de `0.70`.

3.4. Programación Lógica con Restricciones (CLP)

3.4.1. Introducción

- Permite modelizar problemas desde alto nivel:
  - Las restricciones son como (des)ecuaciones sobre elementos arbitrarios.
  - Las primitivas del sistema de restricciones se utilizan para codificar las condiciones del problema.
- Pero:
  - Falta de modularidad.
  - La creación de restricciones dinámicas no es fácil: deben definirse estáticamente.
  - Probablemente el sistema de restricciones no es lo suficientemente potente como para reflejar todo el problema.
  - O las soluciones no se dan en el formato deseado.
- SOLVERS en Bauer (2019): Z3, Microsoft Solver Foundation, Choco, JaCoP, Google's Operations Research Tools and OptiMathSAT.

3.4.1.1. Programación con Restricciones

- Es necesario integrarlas en un lenguaje de programación que ofrezca:
  - Estructuras de datos y abstracción de datos.
  - Algoritmos ad-hoc, cuando se desee y/o sea ventajoso.
  - Modularidad.
- Aporta potencia computacional: el lenguaje anfitrión se enriquece.
- Permite establecer restricciones durante la ejecución del programa.
- Ofrece la posibilidad de añadir control:
  - Flujo de datos.
  - Ejecución del programa.
  - Resolución de restricciones incremental.
- Comunicación externa.

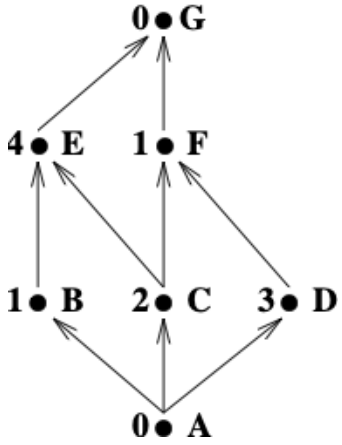
3.4.1.2. Programación Lógica con Restricciones

- Ventajas de su integración con Prolog:
- Tiene una lectura declarativa:
  - Transparencia referencial.

  - Las pequeñas unidades de código tienen un significado propio y aislado (como las ecuaciones matemáticas).
- Las variables lógicas (matemáticas) permiten:
  - Paso de parámetros bidireccional (más que la coincidencia de patrones).
  - Asignación única.
  - No se necesita una gestión explícita de la memoria.
  - Estructuras de datos fáciles de construir.
- Ofrece no-determinismo gracias a su procedimiento de **búsqueda**.

3.4.1.3. Ejemplo resolución de restricciones

- Supongamos la siguiente red de precedencia y las longitudes de las tareas:



- Si todo el trabajo debe estar terminado en 10 unidades de tiempo o menos, una **posible modelización** es:

$$a,b,c,d,e,g \in \{1,\ldots,10\}$$

$$a \leq b,c,d$$
$$b+1 \leq e$$
$$c+2 \leq e$$
$$c+2 \leq f$$
$$d+3 \leq f$$
$$\#\#\#\#$$
$$f+1 \leq g$$

- Hemos usado un dominio finito:
  - Cada variable tiene como dominio asociado los naturales del 1 al 10.
  - Hay inecuaciones que relacionan las variables.
- Una posible estrategia de resolución es:

- Evaluar las ecuaciones una a una.
- Actualizar el dominio de las variables.
- Termina cuando no hay mas actualizaciones.

Paso	*a*	*b*	*c*	*d*	*e*	*f*	*g*
0	1.,10	1.,10	1.,10	1.,10	1.,10	1.,10	1.,10
1		0.,9			1.,10		
2			0.,8		2.,10		
3						2.,10	
4				0.,7		3.,10	
5					2.,6		6.,10
6						3.,9	
7	0.,7						
8		0.,5					
9			0.,4				
10				0.,6			
11	0.,4						
Sol.	0.,4	0.,5	0.,4	0.,6	2.,6	3.,9	6.,10

1. Completa la modelización del ejemplo anterior con la inecuación que falta (indicada con ####).
2. Indica una "instanciación" concreta de las variables de modo que todo el trabajo este terminado en 8 unidades de tiempo. NOTA: a esta operación se le llama "labeling constraints".
3. Explica que significa que el dominio de la tarea e es 2.,6.
4. Responder a las siguientes preguntas:
   - ¿Es posible acabar todo el trabajo en 6 unidades de tiempo?
   - ¿y en 5 unidades de tiempo?

3.4.2. Sistema de restricciones

- Las restricción son condiciones que debe cumplir una solución:
  - $X + Y = 20$
  - $X \wedge Y$ es verdadero
  - El tercer campo de la estructura de datos es mayor que el segundo.
  - El asesino no conocía al mayordomo.

- CLP es Prolog extendido con la capacidad de calcular algún tipo de restricciones (que el sistema resuelve durante la búsqueda).
- Características (adicionales) de un sistema CLP:
  - Dominio (reales, racionales, enteros, booleanos, estructuras, etc).
  - Expresiones que se pueden construir ($+$, $*$, $\wedge$, $\vee$).
  - Restricciones permitidas: (in)ecuaciones ($=$, $\neq$, $\leq$, $\geq$, $<$, $>$).
  - Algoritmos de resolución de restricciones: simplex, gauss, etc.
- Soluciones: asignaciones a variables y/o restricciones entre variables.

3.4.2.1. Definición Formal

- Un *esquema de programación lógica de restricciones*, CLP($\mathcal{X}$), sobre un dominio de restricciones ($\mathcal{D}$, $\mathcal{L}$) se define instanciando el parámetro $\mathcal{X}$, que representa la 4-tupla (Σ, $\mathcal{D}$, $\mathcal{L}$, $\mathcal{T}$) donde:

Σ es el conjunto de símbolos de predicado y función, junto con su aridad.

$\mathcal{D}$ es una Σ-estructura: la estructuras sobre la que se evalúa.

$\mathcal{L}$ es una clase de Σ-fórmulas: el conjunto de restricciones que se pueden escribir.

$\mathcal{T}$ es una Σ-teoría de primer orden: una axiomatización de las propiedades de $\mathcal{D}$. Determina qué restricciones se cumplen y qué restricciones no se cumplen.

- Y se cumple que:
  - $\mathcal{L}$ está construido sobre un lenguaje de primer orden.
  - $=\in \Sigma$ y $=$ es identidad en $\mathcal{D}$.
  - Hay restricciones idénticamente falsas e idénticamente verdaderas en $\mathcal{L}$.
  - $\mathcal{L}$ es cerrado en cuanto a renombrado de variables, conjunción y cuantificación existencial.

Ver Jaffar y Maher (1994).

3.4.2.2. Dominios de restricciones

$\mathbb{R}$ **Aritmética sobre Reales**:
$\Sigma = \{0, 1, +, *, =, <, \leq\}$ $\mathcal{D}$ = Reales interpretando Σ normalmente.
P.ej.: $x^2 + 2xy < \frac{y}{x} \wedge x > 0$ $\equiv$ $xxx + xxy + xxy < y \wedge 0 < x$

$\mathbb{R}_{Lin}$ **Aritmética Lineal**:
$\Sigma = \{0, 1, +, =, <, \leq\}$ $\mathcal{D}$ = Reales interpretando Σ normalmente.
P.ej.: $3x - y < 3$ $\equiv$ $x + x + x < 1 + 1 + 1$

$\mathbb{R}_{LinEq}$ **Ecuaciones lineales**:
$\Sigma = \{0, 1, +, =\}$ $\mathcal{D}$ = Reales interpretando Σ normalmente.

P.ej.: $3x+y=5 \wedge y=2x \quad \equiv \quad x=1 \wedge y=1+1$

Equivalentes dominios pueden definirse sobre los Racionales ($\mathbb{Q}$).

$\mathbb{FT}$ **Árboles finitos o dominio de Herbrand**:[8]

$\Sigma = \{a, b, f/n, g/m, \ldots, =\}$ $\mathcal{D}$ = Árboles finitos donde:

- Cada $f \in \Sigma$ con aridad n es un árbol con la raíz etiquetada f y cuyos n subárboles son los argumentos del functor f (las constantes son hojas).
- $=$ Igualdad sintáctica de árboles.

P.ej.: $g(h(Z), Y) = g(Y, h(a)) \quad \equiv \quad Z = a \wedge Y = h(a)$

$\mathbb{S}tring$ **Ecuaciones sobre strings**:

$\Sigma = \{a, b, \ldots, \lambda, ., =\}$ $\mathcal{D}$ = string donde:

- . se interpreta como concatenación de strings.

P.ej.: $X.A.X = X.A \quad \equiv \quad X = \lambda$

1. Define el dominio de restricciones correspondiente a las **Restricciones Booleanas** ($\mathbb{B}ool$).
2. Escribe una restricción para uno de los siguientes dominio de restricciones y resuélvela:

- $\mathbb{R}$.
- $\mathbb{Q}_{Lin}$.
- $\mathbb{R}_{LinEq}$.
- $\mathbb{Q}$.
- $\mathbb{FT}$.
- $\mathbb{S}tring$.
- $\mathbb{B}ool$.

3.4.3. Resolutor de restricciones: Ejemplo: CLP(R)

3.4.3.1. Ejemplo: CLP(R)

- CLP(R) por Holzbaur (1995): lenguaje basado en Prolog + resolución de restricciones sobre los reales, $\mathbb{R}_{Lin}$.
  - Misma estrategia de ejecución que Prolog, SLD.
  - Permite ecuaciones y ecuaciones lineales sobre los reales.
  - Las restricciones lineales se resuelven.
  - Las restricciones no lineales son pasivas, se retrasan:
    - $X * Y = 7$ se convierte en **lineal** cuando se instancia X.
    - $X * X + 2 * X + 1 = 0$ se convierte en una **comprobación**...
- Soportado en Prolog junto con las primitivas aritméticas `is/2`, `>/2` etc.

[8] Prolog puede verse como una sistema de restricciones sobre términos de Herbrand con un único símbolo de restricción $=$.

- En Ciao, importar el paquete `clpr` y utilizar `.=.`, `.>.`, etc:

- Versión en Prolog

```
suma(X,Y,Z) :- Z is X + Y, Z > 0.
```

- Versión en CLP(R)

```
:- use_package(clpr).
suma(X,Y,Z) :- Z .=. X + Y, Z .>. 0.
```

3.4.3.2. Fibonacci usando CLP(R)

- Vimos en el tema 2 una versión de Fibonacci que solo se podía usar si el primer argumento estaba instanciado a un número.
- Gracias a CLP(R) podemos escribir una versión con restricciones que se comporta como la versión con la aritmética de Peanno:

```
:- use_package(clpr).
fib(0,0).
fib(1,1).
fib(N,F) :-
    N .>. 1, F1 .>=. 0, F2 .>=. 0,
    N1 .=. N - 1, N2 .=. N - 2,
    F .=. F1 + F2,
    fib(N1 ,F1), fib(N2 ,F2).
```

...i.e, `?- fib(6,F)`, `?- fib(N,8)` y `?- fib(P,P)` son consultas válidas.

3.4.4. Reduce el espacio de búsqueda

- Encontrar tres números consecutivos de la relación `p/1`.

- De "genera y prueba" con Prolog.

```
p(11).   p(3).    p(7).
p(16).   p(15).   p(14).

test(X, Y, Z) :-
   Y is X + 1,
   Z is Y + 1.
solution(X, Y, Z) :-
   p(X), p(Y), p(Z),
   test(X, Y, Z).
```

- La consulta `?- solution(X,Y,Z)` requiere **458 pasos** para encontrar la solución:

  `X = 14, Y = 15, Z = 16 ?`

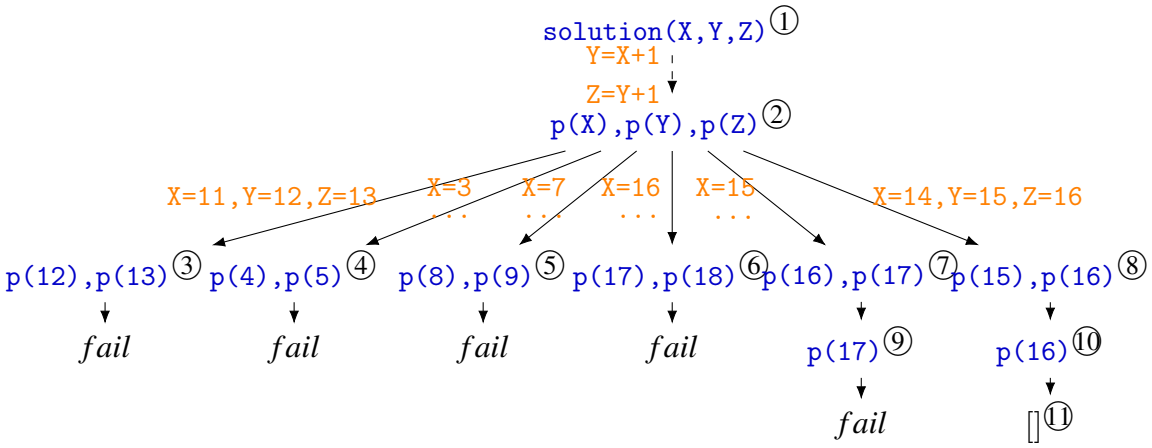
- ...a "restringe y genera" con CLP.

```
:- use_package(clpr).
p(11).   p(3).    p(7).
p(16).   p(15).   p(14).

test(X, Y, Z) :-
   Y .=. X + 1,
   Z .=. Y + 1.
solution(X, Y, Z) :-
   test(X, Y, Z),
   p(X), p(Y), p(Z).
```

- CLP(R) solo necesita 11 pasos!

3.4.4.1. Árbol de derivación



1. Calcula (sin dibujar) el número de nodos del árbol de derivación del siguiente programa con Prolog:

```
p(11).   p(3).    p(7).   p(16).   p(15).   p(14).

test(X, Y, Z) :- Y is X + 1, Z is Y + 1.
solution(X, Y, Z) :- p(X), p(Y), p(Z), test(X, Y, Z).
```

2. Dado el programa anterior, indica la principal diferencia con respecto a la versión con CLP.

```
p(11).   p(3).    p(7).   p(16).   p(15).   p(14).

test(X, Y, Z) :- Y .=. X + 1, Z .=. Y + 1.
solution(X, Y, Z) :- test(X, Y, Z), p(X), p(Y), p(Z).
```

Bibliografía

ALP (2015). **ICLP 2015 LP/CP contest**. URL: https://www.cs.nmsu.edu/ALP/2015/09/report-2015-lpcp-programming-contest/.

Andersen, Carl y Swift, Theresa (2023). **The Janus System: A Bridge to New Prolog Applications**. En: *Prolog: The Next 50 Years*. Springer Nature Switzerland: Cham, págs. 93-104. DOI: 10.1007/978-3-031-35254-6_8.

Arias, Joaquín (2022). **Lógica: desde Aristóteles hasta Prolog**. Servicio de Publicaciones de la Universidad Rey Juan Carlos: Madrid. ISBN: 978-84-09-38265-1.

Arias, Joaquín y Carro, Manuel (2019). **Incremental Evaluation of Lattice-Based Aggregates in Logic Programming Using Modular TCLP**. En: *21st Int'l. Symposium on Practical Aspects of Declarative Languages*. Vol. 11372. LNCS. Springer, págs. 98-114. DOI: 10.1007/978-3-030-05998-9_7.

Arias, Joaquín, Carro, Manuel, Salazar, Elmer, Marple, Kyle y Gupta, Gopal (2018). **Constraint Answer Set Programming without Grounding**. En: *Theory and Practice of Logic Programming* 18(3-4), págs. 337-354. DOI: 10.1017/S1471068418000285.

Augustsson, Lennart, Breitner, Joachim, Claessen, Koen, Jhala, Ranjit, Peyton Jones, Simon, Shivers, Olin, Steele Jr, Guy L y Sweeney, Tim (2023). **The verse calculus: a core calculus for deterministic functional logic programming**. En: *Proceedings of the ACM on Programming Languages* 7(ICFP), págs. 417-447. DOI: 10.1145/3607845.

Bauer, Martin (2019). **A Comparison of Six Constraint Solvers for Variability Analysis**. Department of Informatics and Mathematics, Master's Thesis. University of Passau. URL: https://www.se.cs.uni-saarland.de/theses/MartinBauerMA.pdf.

Blázquez Ballesteros, Inés (2017). **Implementación de un algoritmo de búsqueda aleatoria en programación lógica**. ETSI Informáticos, TFG. Universidad Politécnica de Madrid. URL: https://oa.upm.es/47203.

Cabeza, Daniel, Carro, Manuel y Hermenegildo, Manuel (s.f.). **Breadth-first execution**. The Ciao System. URL: https://shorturl.at/AWXWe.

Church, Alonzo (1936). **A note on the Entscheidungsproblem**. En: *The journal of symbolic logic* 1(1), págs. 40-41. DOI: 10.2307/2269326.

Clark, Keith L. (1978). **Negation as Failure**. En: *Logic and Data Bases*. Ed. por H. Gallaire y J. Minker. Springer, págs. 293-322. DOI: 10.1007/978-1-4684-3384-5_11.

Colmerauer, Alain y Roussel, Philippe (1996). **The birth of Prolog**. En: *History of programming languages—II*, págs. 331-367. DOI: 10.1145/234286.1057820.

Gelfond, Michael y Lifschitz, Vladimir (1988). **The Stable Model Semantics for Logic Programming**. En: *5th International Conference on Logic Programming*, págs. 1070-1080. DOI: 10.2307/2275201.

Green, Cordell (1969). **Application of theorem proving to problem solving**. En: *Proceedings of the 1st International Joint Conference on Artificial Intelligence*. IJCAI'69. Washington, DC, 219–239. URL: https://shorturl.at/8rPHk.

Haemmerlé, Rémy, Carro, Manuel, Vaucheret, Claudio y Hermenegildo, Manuel (s.f.). **Iterative-deepening execution**. The Ciao System. URL: https://shorturl.at/V57pU.

Hermenegildo, Manuel V, Bueno, Francisco, Carro, Manuel, López-García, Pedro, Mera, Edison, Morales, José F y Puebla, Germán (2012). **An overview of Ciao and its design philosophy**. En: *Theory and Practice of Logic Programming* 12(1-2), págs. 219-252. DOI: 10.1017/S1471068411000457. URL: http://ciao-lang.org.

Holzbaur, C. (1995). **OFAI CLP(Q,R) Manual, Edition 1.3.3**. Inf. téc. TR-95-09. Vienna: Austrian Research Institute for Artificial Intelligence.

Jaffar, J. y Maher, M.J. (1994). **Constraint Logic Programming: A Survey**. En: *Journal of Logic Programming* 19/20, págs. 503-581. DOI: 10.1016/0743-1066(94)90033-7.

Kowalski, Robert (1979). **Algorithm = logic + control**. En: *Communications of the ACM* 22(7), págs. 424-436. DOI: 10.1145/359131.359136.

Kowalski, Robert y Kuehner, Donald (1971). **Linear resolution with selection function**. En: *Artificial Intelligence* 2(3-4), págs. 227-260. DOI: 10.1016/0004-3702(71)90012-9. URL: https://shorturl.at/kW7Ts.

Robinson, John Alan (1965). **A machine-oriented logic based on the resolution principle**. En: *Journal of the ACM (JACM)* 12(1), págs. 23-41. DOI: 10.1145/321250.321253.

Skvortsov, Evgeny, Xia, Yilin y Ludäscher, Bertram (2024). **Logica: Declarative Data Science for Mere Mortals.** En: *EDBT*, págs. 842-845. DOI: 10.48786/edbt.2024.84.

Tamaki, H. y Sato, M. (1986). **OLD Resolution with Tabulation**. En: *Third International Conference on Logic Programming*. Lecture Notes in Computer Science, Springer-Verlag: London, págs. 84-98.

Turing, A (1936). **On computable numbers, with an application to the Entscheidungs problem**. En: *Proceedings of the London Mathematical Society Series/2 (42)*, págs. 230-42. DOI: 10.1112/plms/s2-42.1.230. URL: https://shorturl.at/4ZulB.

Warren, D. S. (1992). **Memoing for Logic Programs**. En: *Communications of the ACM* 35(3), págs. 93-111.

Warren, David HD (1977). **Implementing Prologcompiling predicate logic programs**. En: *Research Reports 39 and 40, Dpt. of Artificial Intelligence, Univ. of Edinburgh*. URL: https://shorturl.at/2tSVh.