

Escuela Técnica Superior de Ingeniería Informática

**Apuntes de Programación Lógica**

hasta Prolog y más allá

### Autor: Joaquín Arias

Copyright ⃝c 2024 Joaquín Arias [49](https://orcid.org/0000-0003-4148-311X).Este obra está bajo la licencia CC BY-SA 4.0, [Creative Commons Atribuciónn-CompartirIgual 4.0 Internacional](https://creativecommons.org/licenses/by-sa/4.0/deed.es). Como citar es- ta obra: Arias, Joaquín (2024). Apuntes de Programación Lógica: hasta Prolog y



más allá. Madrid.

# Índice general

**Índice general** [**i**](#_bookmark0)

1. Programación Lógica vs. Funcional [1](#_bookmark1)
   1. Programación Declarativa. [1](#_bookmark2)
   2. Programación Funcional [2](#_bookmark4)
   3. Programación Lógica [5](#_bookmark6)
   4. El problemas de las garrafas [9](#_bookmark11)
   5. El problema del puente y la linterna [9](#_bookmark12)
   6. Las Torres de Hanoi [10](#_bookmark13)
   7. El problema del viajante [10](#_bookmark14)
   8. El problema de las n-reinas [11](#_bookmark15)
   9. Zebra puzzle (Einstein riddle) [12](#_bookmark16)
2. El paradigma de la programación lógica [15](#_bookmark17)
   1. ¿Por qué es importante? [15](#_bookmark18)
   2. Tipos de lógicas [16](#_bookmark19)
   3. Contexto histótico [16](#_bookmark20)
   4. Sintaxis, semántica y teoría interpretativa [18](#_bookmark22)
   5. Método de Resolución de Robinson [19](#_bookmark23)
   6. Resolución con Unificador de Máxima Generalidad [21](#_bookmark24)
   7. Estrategias de resolución [23](#_bookmark27)
   8. Algo de la historia de Prolog [25](#_bookmark31)
   9. Algunas aplicaciones de Prolog [26](#_bookmark32)
3. Programación lógica pura (Turing completo) [27](#_bookmark34)
   1. Sintaxis [27](#_bookmark35)
   2. Semántica [29](#_bookmark37)
   3. Aritmética con enteros [33](#_bookmark41)
   4. Listas [35](#_bookmark44)
   5. Árboles binarios [38](#_bookmark47)
   6. Expresiones Simbólicas [38](#_bookmark48)
4. Programación lógica avanzada [43](#_bookmark56)
   1. Modelo computacional [44](#_bookmark57)
   2. Operador corte [46](#_bookmark58)
   3. Negación [47](#_bookmark59)
   4. Memorización [48](#_bookmark61)
   5. Aritmética [51](#_bookmark67)
   6. Inspeccionar estructuras, entrada/salida... [52](#_bookmark70)
   7. Algoritmos de ordenamiento [53](#_bookmark71)
   8. Meta-interprete [54](#_bookmark74)
   9. Trucos eficientes [55](#_bookmark75)
   10. Recolección de soluciones [56](#_bookmark76)
   11. Predicados de orden superior [57](#_bookmark78)
   12. Agregación de respuestas dinámica [59](#_bookmark79)
   13. Introducción [63](#_bookmark84)
   14. Sistema de restricciones [66](#_bookmark88)
   15. Resolutor de restricciones: Ejemplo: CLP(R) [68](#_bookmark91)
   16. Reduce el espacio de búsqueda [69](#_bookmark94)

Bibliografía [73](#_bookmark96)

## Agradecimientos

Esta obra, contiene en formato de apuntes las transparencias de Arias, Joaquín (2024). [Programación Lógica: hasta Prolog y más allá](https://burjcdigital.urjc.es/handle/10115/41543). Madrid, las cuales están basada en trasparencias de Ana Pradera (URJC’23) & ClipLab (UPM’24).



# Cápitulo 0

**Programación Lógica vs. Funcional**

* 1. Programación Declarativa. [**1**](#_bookmark2)
     1. Contexto [1](#_bookmark3)
  2. Programación Funcional [**2**](#_bookmark4)
     1. Características y Ventajas [5](#_bookmark5)
  3. Programación Lógica [**5**](#_bookmark6)
     1. Cláusulas Horn [6](#_bookmark7)
     2. Prolog [7](#_bookmark8)
     3. Hay algo más ... [7](#_bookmark9)
     4. ... mucho más [8](#_bookmark10)
  4. El problemas de las garrafas [**9**](#_bookmark11)
  5. El problema del puente y la linterna [**9**](#_bookmark12)
  6. Las Torres de Hanoi [**10**](#_bookmark13)
  7. El problema del viajante [**10**](#_bookmark14)
  8. El problema de las n-reinas [**11**](#_bookmark15)
  9. Zebra puzzle (Einstein riddle) [**12**](#_bookmark16)

## Programación Declarativa.

### Contexto

Qué es la programación declarativa?

* Es un paradigma de programación diferente a la imperativa (R) o a la orien- ta 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

1 # Sum list of numbers using R

2 sumaLista <- function(list) {

3 sum <- 0

4 for (n in list)

5 sum <- sum + n

6 return(sum) }

7 # Print the results of the sum, 10

8 print(sumaLista(list(1,2,3,4)))

1 -- Sum list of numbers using Haskell

2 sumaLista :: [Int] -> Int

3 sumaLista [] = 0

4 sumaLista (n : list) = n + (sumaLista list)

5

6

7 # Print the results of the sum, 10

8 main = print (sumaLista [1,2,3,4])

A continuación compararemos Haskell y Prolog

## Programación Funcional

[t]

 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 Cal- culus.

 Church usó Lambda Calculus para resolver el Entscheidungsproblem (1936):

* No existe ningún algoritmo que determine si dos expresiones lambda arbi- trarias son equivalentes.

[t, fragile]

 Introducción al Lambda Calculus.

 Reglas de formación de las expresiones lambda (*λ* -expressions):

* x es una *λ* -expression si x es una variable.
* (*λ* x.t) es una *λ* -expression (function) si t es una expresión y x una varia- ble.
* (t s) es una *λ* -expression (aplicación) si t y s son expresiones.

Función identidad aplicada a 3: ((*λ* x.x) 3) ≡ 3

Función suma aplicada a 2 y 3: (((*λ* x.*λ* y.x+y) 2) 3) ≡ ((*λ* y.2+y) 3) ≡ (2+3)

Función identidad aplicada a suma: ((*λ* x.x) (*λ* x.*λ* y.x+y)) ≡ (*λ* x.*λ* y.x+y)

[t, fragile]

 Debe su nombre a Haskell Curry (1900-1982).

[](https://paiza.io/es/languages/haskell) Dada una función *f* de tipo *f* : (*X*1 ×*X*2 ×···×*Xn*) → *Z*

decimos que su currificación es:

* Una secuencia de funciones con un único argu-

mento:

*curry*( *f* ) : *X*1 → *X*2 → · · · → *Xn* → *Z*.

[haskell.org](https://www.haskell.org/)

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]

[t, fragile]

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

* If-then-else: *λ* x.*λ* y.*λ* z.x y z
* true: *λ* x.*λ* y.x
* false: *λ* x.*λ* y.y

If-then-else True P Q ≡ (*λ* x.*λ* y.*λ* z. x y z) (*λ* x.*λ* y.x) P Q ≡ (*λ* x.*λ* y.x) P Q ≡

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?

[t, fragile]

|  |  |  |
| --- | --- | --- |
| Después, | basado en estas | expresiones definimos And, Or and Not: |
| * And: | *λ* p.*λ* q.p q false | ≡ *λ* p.*λ* q.p q (*λ* x.*λ* y.y) |
| * Or: | *λ* p.*λ* q.p true q | ≡ *λ* p.*λ* q.p (*λ* x.*λ* y.x) q |
| * Not: | *λ* p.p false true | ≡ *λ* p.p (*λ* x.*λ* y.y) (*λ* x.*λ* y.x) |

And True False ≡ (*λ* p.*λ* q.p q (*λ* x.*λ* y.y)) (*λ* x1.*λ* y1.x1) (*λ* x2.*λ* y2.y2) ≡ (*λ* x1.*λ* y1.x1) (*λ* x2.*λ* y2.y2) (*λ* x.*λ* y.y) ≡ (*λ* x2.*λ* y2.y2) ≡ False

Or True False ≡ (*λ* p.*λ* q.p (*λ* x.*λ* y.x) q) (*λ* x1.*λ* y1.x1) (*λ* x2.*λ* y2.y2) ≡ (*λ* x1.*λ* y1.x1) (*λ* x.*λ* y.x) (*λ* x2.*λ* y2.y2) ≡ (*λ* x.*λ* y.x) ≡ True

Not True ≡ (*λ* p.p (*λ* x.*λ* y.y) (*λ* x.*λ* y.x)) (*λ* x1.*λ* y1.x1) ≡ *. . .* ≡ (*λ* x.*λ* y.y) ≡ Fal-

se

 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?

[t, fragile]

 Aunque podríamos considerar otras expresiones para And, Or and Not:

* And2: *λ* p.*λ* q.p q p
* Or2: *λ* p.*λ* q.p p q
* Not2: *λ* p.*λ* x.*λ* y.p y x ¿Cuántos argumentos tiene?

2> Deberes

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

5 {-# LANGUAGE Rank2Types #-}

6 type CB = forall a . a -> a -> a

7 my\_and :: CB -> CB -> CB

8 my\_and = \p -> \q -> p q p

9 my\_or :: CB -> CB -> CB

10 my\_or = \p -> \q -> p p q

11 my\_not :: CB -> CB

-- seems to have 1 argument

12 my\_not = \p -> \x -> \y -> p y x

13

14 k = if\_then\_else (my\_not false) 3 2

### Características y Ventajas

[fragile]

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.

1 concatenar :: [a] -> [a] -> [a]

% declaración de tipos

2 concatenar [] list = list % caso base

3 concatenar (x:xs) list = (x: (concatenar xs list))

% llamada recursiva

4

5 k = concatenar [1,2] [3,4]

% ¿cuánto vale k?

## Programación Lógica

[fragile]

 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* [∀*x* ( *Hombre*(*x*) → *Mortal*(*x*) )*, Hombre*(*socrates*)] ⊢ *Mortal*(*socrates*)

1. Se reescribe como el siguientes conjunto de cláusulas:

{ *Mortal*(*x*) ∨¬*Hombre*(*x*)*, Hombre*(*socrates*)*,* ¬*Mortal*(*socrates*) }

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

1. Si es insatisfacible, significa que hay consecuencia lógica.
2. Se resuelve aplicando método de Robinson con estrategia SLD.

### Cláusulas Horn

[t, fragile]

Introducción a las cláusulas de Horn, definidas por Alfred Horn en 1951.

Dada una cláusula (disyunción de literales) cualquiera *L*1 ∨ *L*2 ∨··· ∨ *Ln*, es una cláusula de Horn si tiene como máximo un literal positivo y esta reescrita como una implicación. Por ejemplo:

|  |  |  |
| --- | --- | --- |
| ¬*p* ∨¬*q* ∨··· ∨¬*t* ∨ *u* | es una regla y se reescribe como | *p* ∧ *q* ∧··· ∧ *t* → *u* |
| *u* | es un hecho y se reescribe como | *u* |
| ¬*p* ∨¬*q* ∨··· ∨¬*t* | sin literal positivo, es una consulta | *p* ∧ *q* ∧··· ∧ *t* → |

Aristóteles:

*Hombre*(*x*) → *Mortal*(*x*) *Hombre*(*socrates*)

*Mortal*(*socrates*)

Cláusulas:

*Mortal*(*x*) ∨¬*Hombre*(*x*) *Hombre*(*socrates*)

¬*Mortal*(*socrates*)

Prolog:

mortal(X) :- hombre(X). hombre(socrates).

?- mortal(socrates).

### Prolog

[fragile,t]

 *Predicados*: Transforma *f* : (*X*1 × *X*2 ×··· × *Xn*) → *Z*

en una relación (n+1)-aria *R* y define el predicado *r* tal que:

[](https://swish.swi-prolog.org/)*r*(*x*1*, x*2*, . . . , zn, z*) = *true* ←→ (*x*1*, x*2*, . . . , zn, z*) ∈ *R*.

[](https://ciao-lang.org/playground)

[SWI-Prolog](https://www.swi-prolog.org/) [Ciao Prolog](https://www.ciao-lang.org/)

2>

1 mortal(X) :- hombre(X). % Todos los hombres son mortales.

2 hombre(socrates). % Sócrates es un hombre.

3

4 ?- mortal(socrates). % ¿Sócrates es mortal?

3>

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?

2->?- mortal(socrates).

Contesta yes si la “pregunta” es consecuencia lógica (no en caso contrario). 3>?- concatenar([1,2],[3,4],Lista).

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

### Hay algo más ...

[fragile]

 Mientras las funciones devuelven un único resultado:

k = concatenar [1,2] [3,4]

 Los predicados pueden “consultarse” de diferentes formas sin cambiar el pro- grama:

?- 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 = [] ?

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

2

3 ?- char2morse('B', Morse).

% devuelve Morse = '-...'

4 ?- char2morse(Char, '-.-.').

% devuelve Char = 'C'

### mucho más

[t,fragile]

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

3*x* + 5*y* = 2

(

5*x* + 3*y* = −2

1 sol(X,Y) :-

2 3 \* X + 5 \* Y #= 2,

3 5 \* X + 3 \* Y #= -2.

?- sol(X,Y).

X = -1, Y = 1 ?

1 mg(P,T,\_,\_,B) :- T #= 0, B #= P.

2 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: <3>... y mucho más.

2->

2->

5->

?- 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 ?

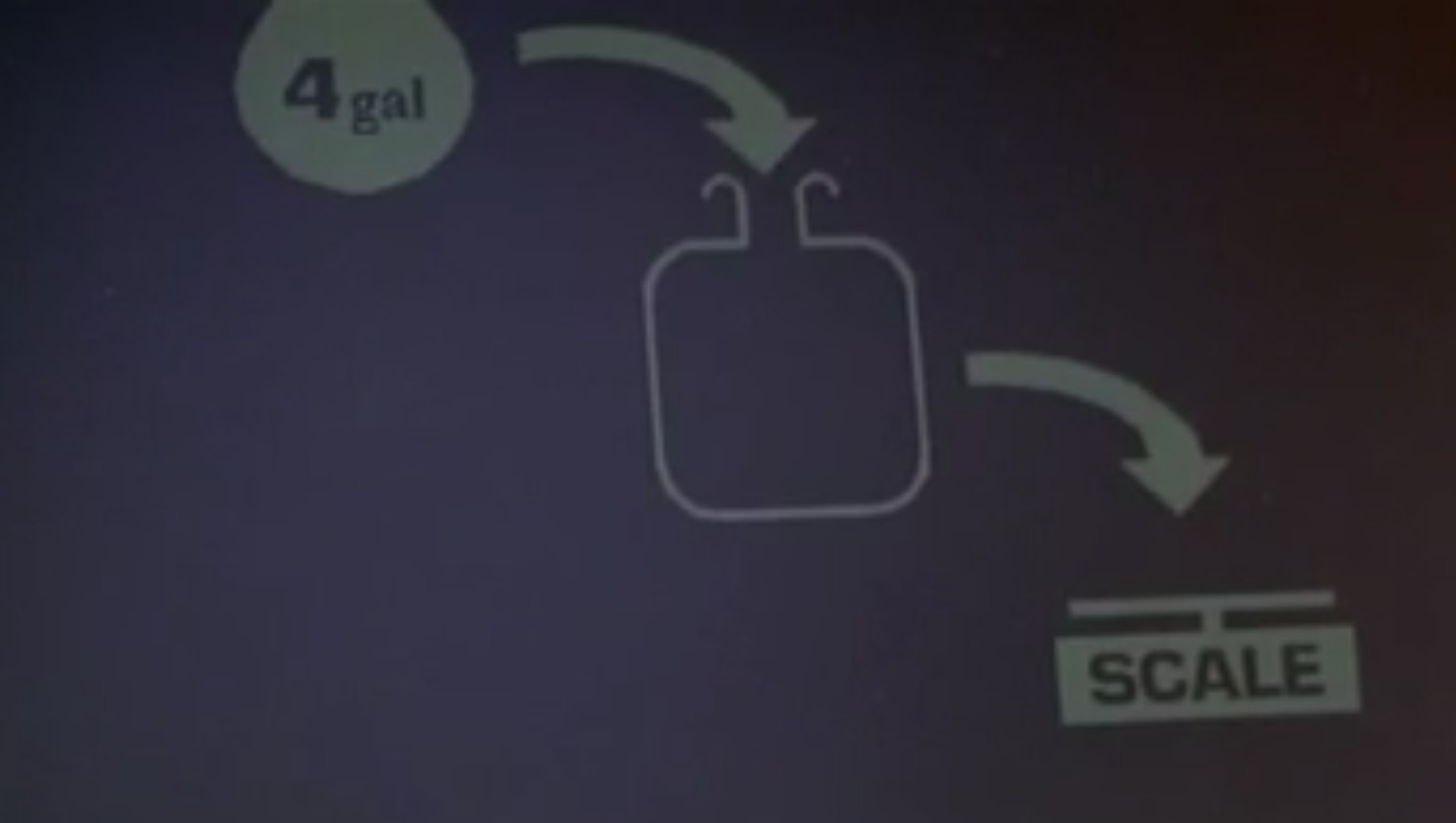
P = 6.14\*R + 0.38\*B ?

## 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.

[](tema0/garrafas.mp4)

[Link](https://youtu.be/cKmOsznvLBM)

## 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).

[](tema0/puente.mp4)

[Link](https://youtu.be/FQ7CNVZs4TM)

## Las Torres de Hanoi

[t]

 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.

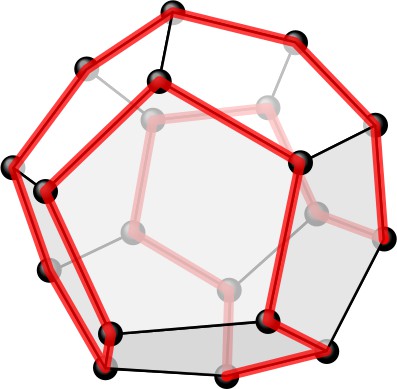
Use Acrobat Reader

## El problema del viajante

[t]

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 gra- fo, 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, planifica- ción, etc.

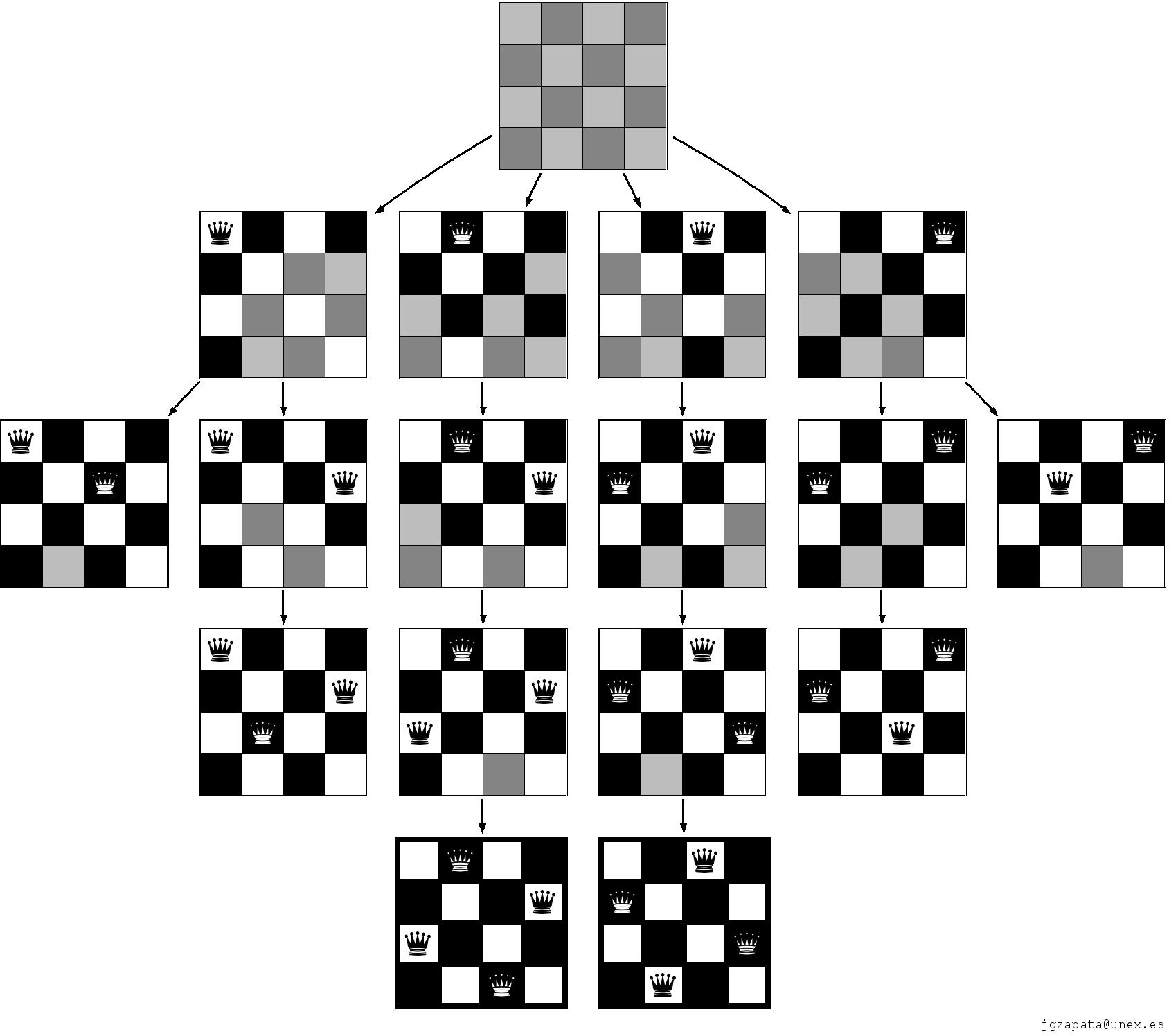
## El problema de las n-reinas

[t]

 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 back- tracking, "depth-first".

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



## Zebra puzzle (Einstein riddle)

[fragile,t]

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.



# Cápitulo 1

**El paradigma de la programación lógica**

* 1. ¿Por qué es importante? [**15**](#_bookmark18)
  2. Tipos de lógicas [**16**](#_bookmark19)
  3. Contexto histótico [**16**](#_bookmark20)
     1. Ejemplo del Entscheidungsproblem [18](#_bookmark21)
  4. Sintaxis, semántica y teoría interpretativa [**18**](#_bookmark22)
  5. Método de Resolución de Robinson [**19**](#_bookmark23)
  6. Resolución con Unificador de Máxima Generalidad [**21**](#_bookmark24)
     1. Sustitución y algoritmo de unificación [21](#_bookmark25)
     2. Algoritmo de resolución con UMG [22](#_bookmark26)
  7. Estrategias de resolución [**23**](#_bookmark27)
     1. Estrategia de resolución SLD [24](#_bookmark28)
     2. Ejercicio [24](#_bookmark29)
  8. Algo de la historia de Prolog [**25**](#_bookmark31)
  9. Algunas aplicaciones de Prolog [**26**](#_bookmark32)

## ¿Por qué es importante?

Lógica: ¿Por qué es importante?

 La lógica formal es la ciencia que estudia las leyes de inferencia en los razona- mientos. Repasar apuntes de Lógica ([Arias 2022](#_bookmark99)).

 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 pro- gramas
* 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.

## Tipos de lógicas

[t]

 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 mor-

tal”

*h* ∧ *m*

#### Lógica de primer orden

 “*x* es un hombre” *H*(*x*).

 “*x* es mortal” *M*(*x*).  “Todos los hombres son morta-

les”

∀*x.H*(*x*) → *M*(*x*)

 Sin embargo existen otras lógicas:

De 2*o* orden, no monótona, epistémica, deóntica, temporal, lineal, etc.

## Contexto histótico

[t]

**S. IV a.C.:** Aristóteles formaliza el razonamiento humano. Fundó la lógica clá- sica 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

[t]

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

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

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

[t]

La lógica de de 1*er* orden no posee un procedimiento de decisión (al- goritmo) que permita demostrar que una fórmula cualquiera sea un teorema de dicha lógica. Esta lógica se considera, por tanto, indeci- dible ([Church 1936](#_bookmark106)).

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](#_bookmark120)).

 Lógica y Computación son equivalentes: Determinar el proble- ma 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.

[t]

 **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).

### Ejemplo del Entscheidungsproblem

[t]

 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 → Dado un patrón inicial cualquiera, no podemos determinar si se estabilizará en un tiempo finito o no.

[PLAY1](https://youtu.be/u1-yOFuLTaU)

Ver “Las Matemáticas tienen una Terrible Falla” en <https://bit.ly/3sFyaxq>.

## Sintaxis, semántica y teoría interpretativa

[t]

 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 má-

ximo, un literal positivo. P.ej. dada la consecuencia lógica:

*T* [∀*x* ( *Hombre*(*x*) → *Mortal*(*x*) )*, Hombre*(*socrates*)] ⊢ *Mortal*(*socrates*)

se reescribe como el siguientes conjunto de cláusulas:

{ *Mortal*(*x*) ∨¬*Hombre*(*x*)*, Hombre*(*socrates*)*,* ¬*Mortal*(*socrates*) }

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

## Método de Resolución de Robinson

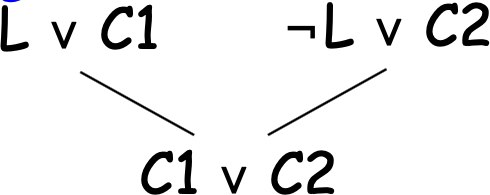
[t]

 Basado en:

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* ∨ *C*1 y ¬*L* ∨*C*2 (*L* es un literal) puede deducirse una nueva instancia básica *C*1 ∨*C*2, llamada resolvente:



[t]

[t]

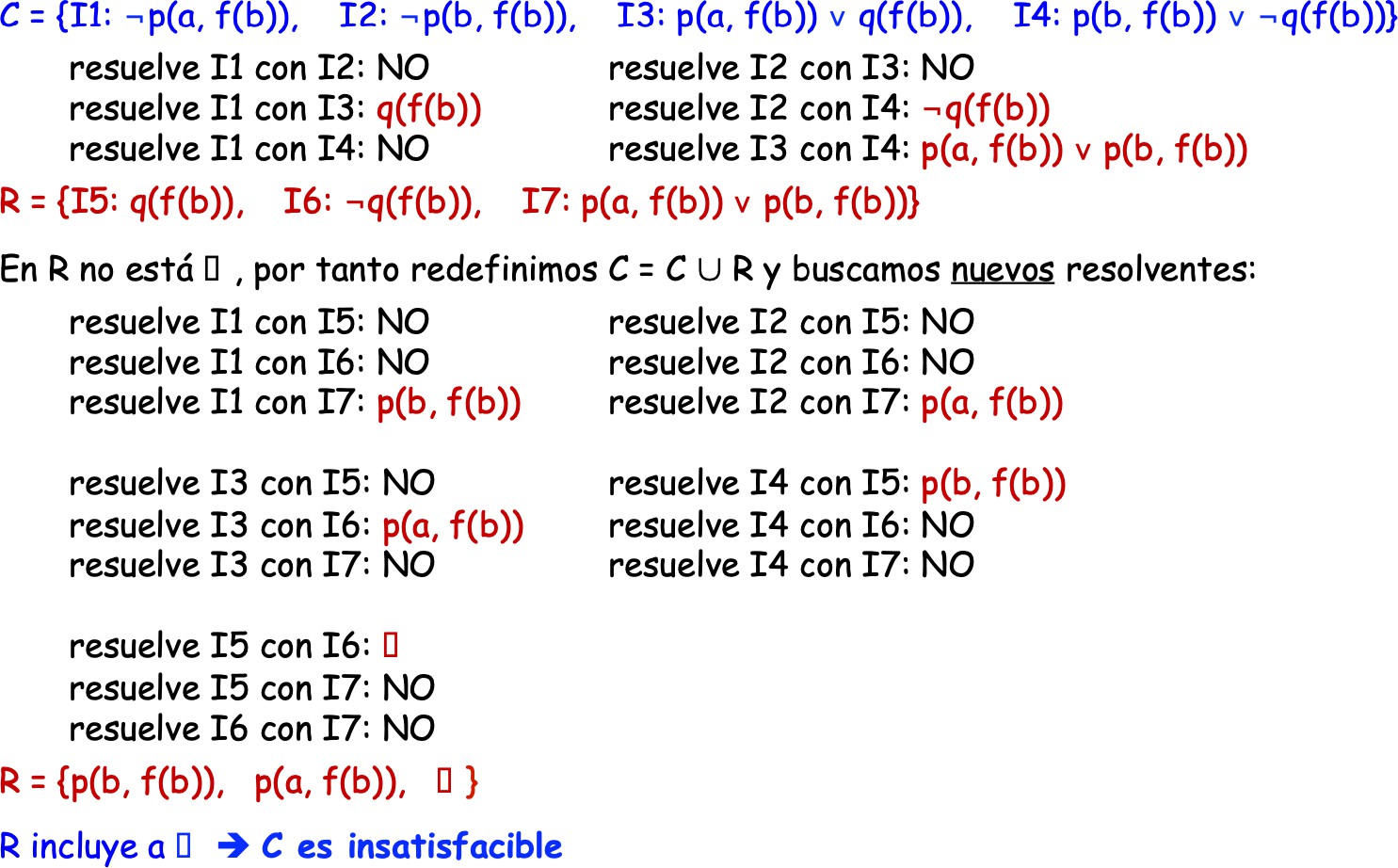
La aplicación sucesiva de la regla de resolución permite obtener una contradic- ción cuando el conjunto original es insatisfacible.

Dado un conjunto C de instancias básicas:

1. Generar el conjunto *R* de todos los resolventes que pueden obtenerse apli- cando la regla de resolución entre instancias del conjunto *C* de todas las formas posibles.
2. Si [] está incluida en *R* entonces terminar ⇒ *C* es insatisfacible.
3. Si *R* ⊆ *C* significa que ya se han generado todos los resolventes posibles, entonces terminar ⇒ *C* es satisfacible.
4. Hacer *C* = *C* ∪ *R* y repetir desde paso 1.

El método es correcto: Si deducimos [] entonces *C* es insatisfacible.

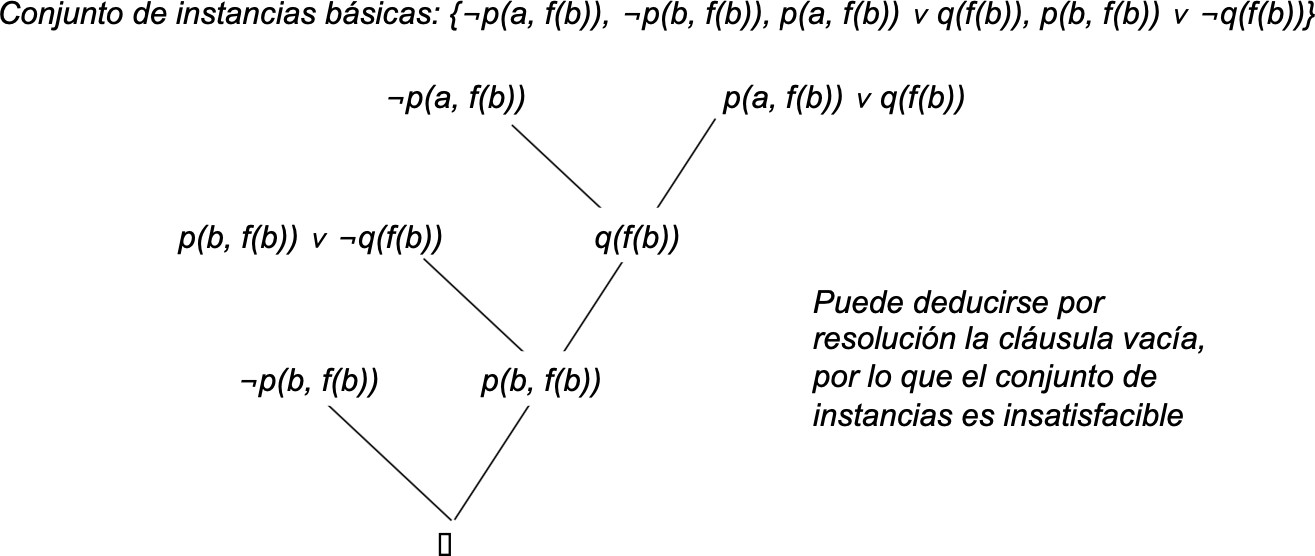
El método es completo: Si *C* is insatisfacible, entonces con la aplicación de la regla de resolución deduciremos [].



[t]

 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 [].



## Resolución con Unificador de Máxima Generali- dad

### Sustitución y algoritmo de unificación

[t]

[t]

[t]

 Un sustitución *α* = {*x*1*/t*1*, . . . , xn/tn*} es una función finita de un conjunto de variables de un lenguaje en el de términos. Donde *xi/ti* es una ligadura.

 Dada una fórmula *F* y una sustitución *α* = {*x*1*/t*1*, . . . , xn/tn*}, se denomina apli- cación de *α* a *F* (*Fα*) a la fórmula obtenida reemplazando simultáneamente cada ocurrencia en *F* de *xi* por *ti*, para cada *xi/ti* ∈ *α*.

 *F*′ es instancia de *F* si existe una sustitución *α* ̸= 0/ tal que *F*′ = *Fα*

 Sean A y B dos átomos con el mismo símbolo de predicado:

1. *α* = *λ*
2. Mientras A*α* ̸= B*α*:
   1. Encontrar el símbolo más a la izquierda en A*α* tal que el símbolo corres- pondiente en B*α* sea diferente.
   2. Sean *tA* y *tB* los términos de A*α* y B*α* que empiezan con esos símbolos:
      * Si ni *tA* ni *tB* son variables o, si uno de ellos es una variable que aparece en el otro ⇒ terminar con fallo (*A* y *B* no son unificables)
      * En otro caso, sea *tA* una variable ⇒ el nuevo *α* es el resultado de

*α*{*tA/tB*}

1. Terminar, siendo *α* el umg de *A* y *B*

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

|  |  |  |  |
| --- | --- | --- | --- |
| *α* | A*α* | B*α* | (*ta,tb*) |
| *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* | A*α* | B*α* | (*ta,tb*) |
| *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*)}

### Algoritmo de resolución con UMG

[t]

 Regla de resolución con umg: Sean *L*1 ∨··· ∨ *Ln* ∨*C*1 y ¬*L*′ ∨··· ∨¬*L*′ ∨*C*2 dos

1 *m*

cláusulas, donde todos los *Li j* son literales con el mismo símbolo de predicado. Puede deducirse una nueva cláusula (*C*1*ρ*1 ∨*C*2*ρ*2)*β* , llamada resolvente, donde

* + - * *ρ*1 y *ρ*2 son renombrados cuyos dominios respectivos son todas las varia- bles de cada cláusula y *Rango*(*ρ*1) ∩ *Rango*(*ρ*2) = 0/
      * *β* es umg de {*L*1*ρ*1*, . . . , Lnρ*1*,* ¬*L*′ *ρ*2*, . . . ,* ¬*L*′ *ρ*2}

1 *m*

La regla de resolución con umg se apoya en una versión de la regla de factori- zación para LPO: Dada una cláusula *L*1 ∨··· ∨ *Ln* ∨*C*, siendo *L*1*, . . . , Ln* literales con el mismo símbolo de predicado, puede deducirse una nueva cláusula *L* ∨*Cβ*

donde

* + - * *β* es unificador de *L*1*, . . . , Ln*
      * *L* = *L*1*β* = · · · = *Lnβ* El literal *L* se denomina

factor de *L*1 ∨··· ∨ *Ln* ∨*C*

[t]

 La aplicación de la regla de resolución con UMG es correcta.  La aplicación de la regla de resolución con UMG es completa.

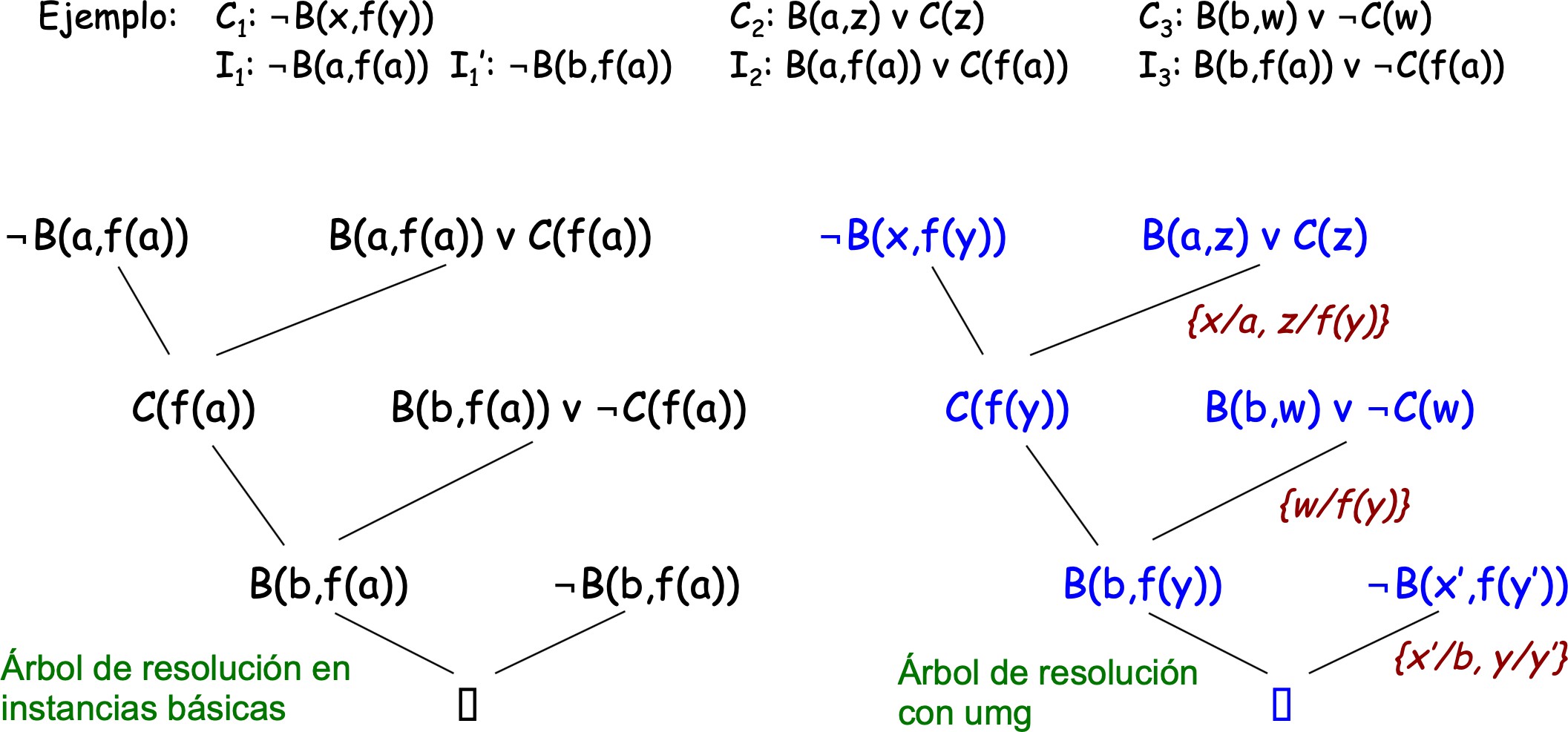
Un conjunto de cláusulas es insatisfacible sii se puede deducir [] a partir de él por resolución con umg.

 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.

[t]

 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.



## Estrategias de resolución

[t]

 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 cláu- sulas en el conjunto.
* Estrategias de refinamiento: con el objetivo de limitar la generación de cláusulas.

|  |  |  |  |
| --- | --- | --- | --- |
| [t] |  | | |
|  | Estrategia | Correcta | Completa |
|  | Saturación | Si | Si |
|  | Lineal | Si | Si |
|  | Input | Si | No, (caso general) |
|  | Dirigida | Si | Si, si el conjunto so- porte es satisfacible |
|  | Ordenada | Si | No |

¿Qué significa ser correcta/completa?

* Correcta:

[] → S insatisfacible

[] se deduce sólo si el conjunto de cláusulas S es insatisfacible.

* Completa:

S insatisfacible → []

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

### Estrategia de resolución SLD

[t]

 SLD[1](#_bookmark30): 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*, . . . ,Ci, . . . ,Cn*}:

Existe una secuencia (*derivación*) *<Ci,Cn*+1*, . . . ,* [] *>* tal que:

* + - * *Cn*+1 es el resolvente de la cláusula objetivo *Ci* y una cláusula soporte.
      * *Ck*, con *k > n* + 1, es el resolvente de *Ck*−1 con una cláusula soporte.
      * Cada paso de resolución es de la forma *L* ∨*C,* ¬*L* ∨*C*′ → *C* ∨*C*′.

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

### Ejercicio

[fragile,t]

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

∀*ls Concatenar*([ ]*, ls, ls*)

∀*x* ∀*xs* ∀*ls* ∀*ns* ( *Concatenar*(*xs, ls, ns*) → *Concatenar*([*x*|*xs*]*, ls,* [*x*|*ns*]) )

∃*la* ∃*lb Concatenar*(*la, lb,* [1*,* 2*,* 3*,* 4])

1. Comprobar que es equivalente al programa Prolog: [rt](https://bit.ly/3cfvOAl)

1 concatenar([],Ls,Ls).

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

3

1SLD significa resolución **L**ineal con funciónn de **S**elección para cláusulas **D**efinidas (Selecting a literal, using a Linear strategy, restricted to Denfinite clauses).

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

## Algo de la historia de Prolog

[t]

[t]

[t]

#### 1960’s:

* Robinson ([Robinson 1965](#_bookmark117)): propone en 1965 una regla de inferencia a la que llama resolución, mediante la cual la demostración de un teorema pue- de ser llevada a cabo de manera automática.
* Green ([Green 1969](#_bookmark110)): diseña un probador de teoremas que extrae de la prue- ba, el valor de las variables para las cuales el teorema es válido.

#### 1970’s:

* Kowalski ([Kowalski y Kuehner 1971](#_bookmark116)): propone la resolución SLD (muy eficiente).
* Alain Colmerauer ([Colmerauer y Roussel 1996](#_bookmark108)): crea Prolog, el primer lenguaje de programación lógica (implementado en Fortran).
* Kowalski ([Kowalski 1979](#_bookmark115)): interpretación procedimental de la lógica: Algorithm = logic + control
* D.H.D. Warren ([Warren 1977](#_bookmark122)): 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 pa- radigmas básicos y las técnicas de aplicación avanzadas.
* Varias implementaciones comerciales de Prolog, libros y manuales, utili- zando 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](#_bookmark114)): Ex- tiende expresividad de Prolog –abrió nuevos campos de aplicación.

#### 2000’s:

* Muchas otras extensiones: orden superior, tipos/modos, concurrencia, dis- tribución, objetos, sintaxis funcional, etc.
* Compiladores altamente optimizadores, paralelismo automático, verifica- ción y depuración automáticas, entornos avanzados.

#### 2010’s:

* Variaciones: Datalog, Answer Set Programming (soporta negación median- te modelos estables), Minikanren, MiniZinc, SQL, Yedalog.

#### 2020’s:

* Verse ([Augustsson et al. 2023](#_bookmark102)): Lenguaje lógico y funcional de Epic Games (Fortnite).
* Logica language ([Skvortsov, Xia y Ludäscher 2024](#_bookmark118)): desarrollado en Goo- gle.
* Janus ([Andersen y Swift 2023](#_bookmark97)): Programación multiparadigma en Prolog y Python.
* s(CASP) ([Arias et al. 2018](#_bookmark101)): 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.[2](#_bookmark33)

## Algunas aplicaciones de Prolog

[t]

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.

 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 procedi- mientos.

 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.

2Ver detalles y los proyectos en [https://utd-hackreason-2021.devpost.com](https://utd-hackreason-2021.devpost.com/) y [https:](https://utd-hackreason-2022.devpost.com/)

[//utd-hackreason-2022.devpost.com](https://utd-hackreason-2022.devpost.com/).



# Cápitulo 2

**Programación lógica pura (Turing completo)**

* 1. Sintaxis [**27**](#_bookmark35)
  2. Semántica [**29**](#_bookmark37)
     1. Árbol de derivación [30](#_bookmark38)
     2. Estrategias de búsqueda [31](#_bookmark39)
  3. Aritmética con enteros [**33**](#_bookmark41)
  4. Listas [**35**](#_bookmark44)
     1. Combinar aritmética y listas en Prolog [37](#_bookmark46)
  5. Árboles binarios [**38**](#_bookmark47)
  6. Expresiones Simbólicas [**38**](#_bookmark48)
     1. Reconocer polinomios. [39](#_bookmark49)
     2. Calcular la derivada. [39](#_bookmark50)
     3. Torres de Hanoi. [40](#_bookmark52)
     4. Satisfabilidad de fórmulas Booleanas [41](#_bookmark53)

## Sintaxis

[t,fragile]

 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 empie- zan por (i) letra mayúscula o (ii) guión bajo.*[1](#_bookmark36) 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.

[t,fragile]

 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 ’,’ con- junción ’∧’. 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.

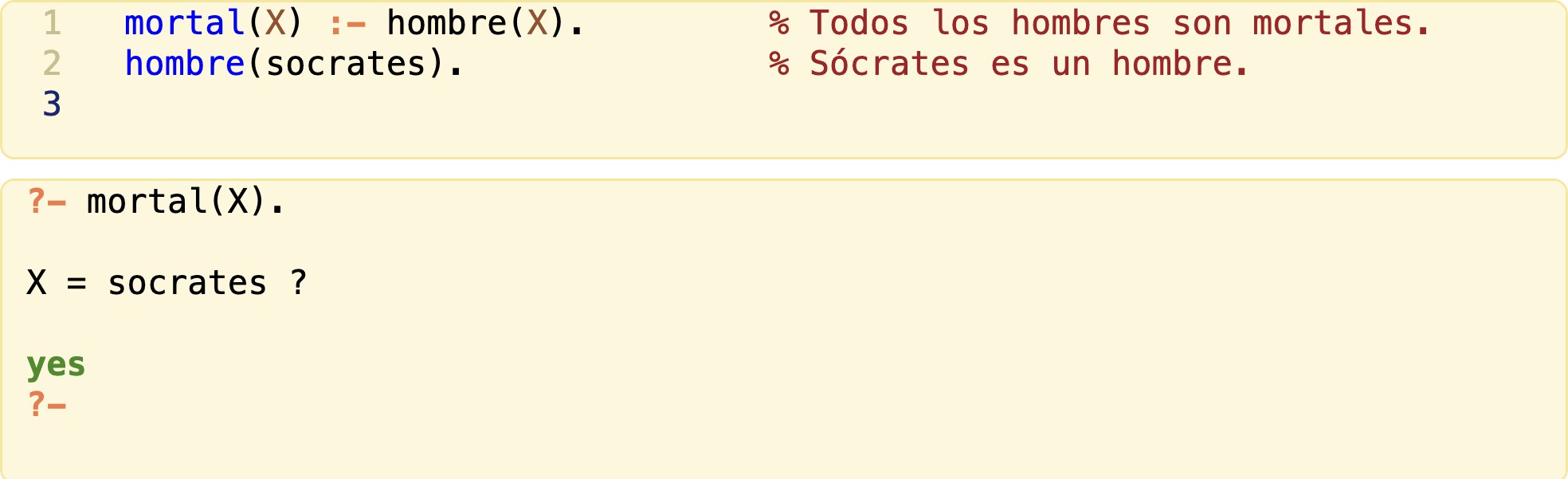
[t,fragile]

 En Prolog, un programa es:

* Un conjunto de clausulas (reglas y hechos).

1Las variables que empiezan con guión bajo se llaman variables anónimas (ver pág. [2.2.2](#_bookmark40)).

* Se pueden incluir comentarios en línea con el símbolo %.  El programa se ’ejecuta’:
* Invocando la consulta en el top-level.



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

## Semántica

[t]

Programas de cláusulas de Horn (definite programs) tiene dos semánticas equi- valentes:

* El modelo mínimo de Herbrand de *P*. *Mp* = { *A* ∈ *Bp* | *P* ⊨ *A* }
* *Least fixed point* de *T p* (operador de consecuencia inmediata). *T p* ↑ *ω* =

L

∞

*T pi*(0/ )

*i*=0

* + Donde *T p*(*I*) = {*A*0 ∈ *Bp* | *A*0 ← *A*1*, . . . , An* ∈ *P* ∧ *Ai* ∈ *I*}.

Adicionalmente, su significado lo podemos ver como:

1. Un conjunto de fórmulas a las que aplicamos deducción.

*T* [ ∀*x* ( *Hombre*(*x*) → *Mortal*(*x*) )*, Hombre*(*socrates*) ] ⊢ ∃*x Mortal*(*x*)

1. Un conjunto de cláusulas a las que aplicamos resolución SLD.

{ *Mortal*(*x*) ∨¬*Hombre*(*x*)*, Hombre*(*socrates*)*,* ¬*Mortal*(*x*) }

1. Un conjunto de procedimientos ejecutados a partir de la consulta siguiendo una semántica operacional dada (ver tema 3).

[t, fragile]

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

∀*ls Concatenar*([ ]*, ls, ls*)

∀*x* ∀*xs* ∀*ls* ∀*ns* ( *Concatenar*(*xs, ls, ns*) → *Concatenar*([*x*|*xs*]*, ls,* [*x*|*ns*]) )

∃*la* ∃*lb Concatenar*(*la, lb,* [1*,* 2*,* 3*,* 4])

1. Comprobar que es equivalente al resultado del programa: [rt](https://bit.ly/3cfvOAl)

1 concatenar([],Ls,Ls).

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

3

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

### Árbol de derivación

[fragile,t]

 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 ca- beza 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).

* + - 1. En caso contrario, se “resuelve” un literal del resolvente.
         1. Se crea un hijo por cada cláusula que unifica con dicho literal.
         2. Se añade el cuerpo de la clausula al resolvente.
         3. Se vuelve al punto 3.

[fragile,t]

concatenar(La,Lb,[1,2,3,4])

La=[], Lb=[1,2,3,4] La=[1|Xs]

[] concatenar(Xs,Lb,[2,3,4])

Xs=[]

La=[1], Lb=[2,3,4]

Xs=[2|Xs']

[] concatenar(Xs',Lb,[3,4])

Árbol de *derivación* de:

Xs'=[]

La=[1,2], Lb=[3,4]

Xs'=[3|Xs'']

concatenar(Xs'',Lb,[4])

[rt](https://bit.ly/3cfvOAl) []

Xs''=[]

X''=[4|Xs''']

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]).

La=[1,2,3], Lb=[4]

[] concatenar(Xs''',Lb,[])

X'''=[]

La=[1,2,3,4], Lb=[]

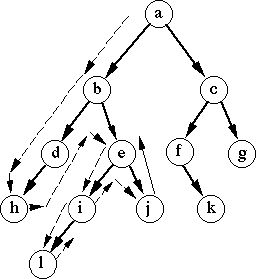
[]

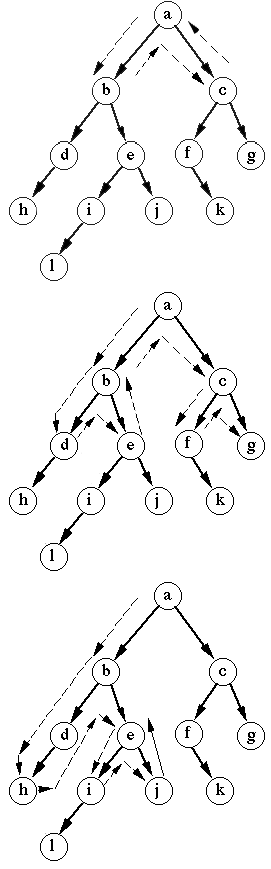
### Estrategias de búsqueda

[fragile,t]

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

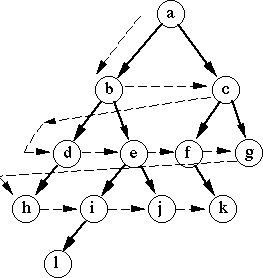
#### En profundidad Profundización itera-

**tiva**



[fragile,t]

#### En Anchura



* + - 1. Dado el siguiente programa: [rt](https://bit.ly/3QRy3IP)

1 edge(1, 2).

2 edge(2, 3).



1

3

2

3

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

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

6

7 ?- reach(1,Y).

* + - * 1. Construir el árbol de derivación.
        2. Discutir las ventajas e inconvenientes de la búsqueda en profundidad vs. anchura.

[fragile,t]

* + - 1. Dado el siguiente programa: [rt](https://bit.ly/3KzWqc1)

1 edge(1, 2).

2 edge(2, 1).

3 edge(2, 3).



1

3

2

4

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

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

7

8 ?- reach(1, Y).

* + - * 1. Construir el árbol de derivación.
        2. Discutir las ventajas e inconvenientes de la búsqueda en profundidad vs. anchura.

[fragile,t]

 Ciao Prolog ([Hermenegildo et al. 2012](#_bookmark112)) dispone de diferentes estrategias de bús- queda:

***bf* :** Implementa búsqueda en anchura ([Cabeza, Carro y Hermenegildo](#_bookmark105) ). Ejem- plo:

1 :- use\_package(sr/bfall).

2 ...

3 bf\_reach(X, Y) :- edge(X, Z), bf\_reach(Z, Y).

4 bf\_reach(X, Y) :- edge(X, Y).

***id*:** Implementa profundidad iterativa ([Haemmerlé et al.](#_bookmark111) ). Ejemplo:

1 :- use\_package(id).

2 :- iterative(id\_reach/2,0,(\_(X,Y) :- Y is X + 5),10).

3 ...

4 id\_reach(X, Y) :- edge(X, Z), id\_reach(Z, Y).

5 id\_reach(X, Y) :- edge(X, Y).

***rndsearch*:** Este bundle externo implementa un algoritmo de búsqueda aleatoria.

TFG de ([Blázquez Ballesteros 2017](#_bookmark104)).

[t,fragile,label=anonimas]

1. **En predicados**, cuando no son necesarias (evita avisos). P.ej., en:

1 node(X) :- edge(X, \_).

2 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.

1. **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.

## Aritmética con enteros

[t,fragile]

 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](#_bookmark42)
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.

1. El principio de inducción matemática, su formulación en lógica de segundo orden es:

[t,fragile]

∀*φ* *φ* (1) ∧∀*x*(*φ* (*x*) → *φ* (*x*′)) → ∀*x φ* (*x*)

 Primero, veremos las operaciones como relaciones (no como funciones).

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

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.

1 % nat(X): X is a natural number.

2 nat(0).

3 nat(s(X)) :- nat(X).

* ?- nat(X) genera recursivamente: 0, s(0), s(s(0)), *. . .*
* Donde *sn*(0) denota *n* (aplicar *n* veces el funtor sucesor al 0).

 **Proposición 1:** el programa nat/1 es correcto y completo con respecto al con- junto de consultas ?- nat(s*i*(0)) para *i* ≥ 0.

[t,fragile]

#### 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 de- rivación. O bien n es O o de las forma si(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](#_bookmark43)

1. **Corrección:** *Supongamos que el nat(X) es deducible a partir del progra- ma 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*, en-*

*tonces 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'=sk(0) para algún k* ≥ 0*.*

[t,fragile]

 **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.

1 % plus(X,Y,Z): X, Y, and Z are natural numbers

2 % such that Z is the sum of X and Y

3 plus(0,X,X) :- nat(X).

4 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.

3Ejercicio: dibujar el árbol de derivación.

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

[t,fragile]

 Implementa y demostrar si son correctos y completos:

1. Menor o igual.

1 % leq(X,Y): X, and Y are natural numbers

2 % such that X is less than or equal to Y.

1. Multiplicación como suma repetida.

1 % times(X,Y,Z): X, Y, and Z are natural numbers

2 % such that Z is the product of X and Y.

1. Exponenciación como multiplicación repetida.

1 % exp(N,X,Y): N, X, and Y are natural numbers

2 % such that Y equals X raised to the power N.

1. Calculando el factorial.

1 % factorial(N,F): F equals N factorial.

1. El mínimo de dos números.

1 % minimum(N1,N2,Min): The minimum of N1 and N2 is Min.

[t,fragile]

 Implementa las siguientes operaciones:

1. Calcular el residuo/módulo (resto de una division)

1 % mod(X,Y,Z): Z is the remainder of the integer

2 % division of X by Y

* 1. Implementar con: (i) times/3 y (ii) recursivamente,
  2. Comparar sus complejidades (nodos en el árbol de derivación).

1. La función de Ackermann:[4](#_bookmark45)

1 % ackermann(X,Y,A): A is the value of Ackerman's

2 % function for X and Y.

1. El algoritmo de Euclides para calcular el máximo común divisor.

1 % gcd(X,Y,Z): Z is the greatest common divisor of

2 % the natural numbers X and Y

## Listas

[t,fragile]

 Un tipo recursivo mas complejo es una estructura binaria, la lista.

4Su definición está disponible en <https://shorturl.at/Sh6Ry>

* Se construye con la constante [], para la lista vacía.
* y con el funtor binario ./2, de modo que:

1 % list(List): List is a list

2 list([]).

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]}. [t,fragile]

1. Implementa la mas básica operación con listas, determinar si un elemento parti- cular esta en una lista:

1 % member(Element, List): Element is an element

2 % of the list List

1. 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.

1. 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

1. 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

[t,fragile]

 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).

[t,fragile]

 Implementemos el predicado reverse(List,Reverse, donde Reverse es una lis- ta 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:

1 % a) Naive reverse

2 reverse([],[]).

3 reverse([X|Xs], Zs) :-

4 reverse(Xs,Ys),

5 append(Ys,[X],Zs).

pero con complejidad cuadrática (ver árbol de derivación de la consulta anterior).

1. Como alternativa, definimos un predicado auxiliar reverse\_(Xs,Acc,Ys), donde

Ys es el resultado de concatenar Acc a los elementos de Xs invertidos:

1 % a) Reverse-accumulate

2 reverse(Xs, Ys) :-

3 reverse\_(Xs, [], Ys).

4 reverse\_([], Acc, Acc).

5 reverse\_([X|Xs], Acc, Ys) :-

6 reverse\_(Xs, [X|Acc], Ys).

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

### Combinar aritmética y listas en Prolog

[t,fragile]

 Dada la siguiente implementación de length/2:

1 % length(Xs, N): The list Xs has N elements

2 length([], 0).

3 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.
         1. Usando el programa plus/3.
         2. Sin utilizar predicados auxiliares.

## Árboles binarios

[t,fragile]

 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.

## Expresiones Simbólicas

[t,fragile]

 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 − 3*x* + 2 es un polinomio en *x*.

1. Calcular la derivada de una expresión con respecto de X:

p.ej., derivate(X,X,s(0)) esta en la semántica del programa.

1. Resolver las torres de Hanoi.

...en lugar de una relación definiremos un esquema.

1. Determinar la satisfacibilidad de una fórmula Booleana.

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

invalid(X) cuando X es Falsa.

### Reconocer polinomios.

[t,fragile]

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

1 % polynomial(Expresion,X): Expresion is a polynomial in X.

2 polynomial(X, X).

3 polynomial(Term, X) :- nat(Term).

4 polynomial(Term1 + Term2, X) :- polynomial(Term1, X), polynomial(Term2, X).

5 polynomial(Term1 - Term2, X) :- polynomial(Term1, X), polynomial(Term2, X).

6 polynomial(Term1 \* Term2, X) :- polynomial(Term1, X), polynomial(Term2, X).

7 polynomial(Term1 / Term2, X) :- polynomial(Term1, X), nat(Term2).

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

 De modo que la consulta correspondiente al polinomio *x*2 − 3*x* + 2 en *x*:

?- polynomial(x^s(s(0)) - s(s(s(0)))\*x + s(s(0)), x) tiene éxito.

[t,fragile]

 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 funto- res prefijos p, s, m, d, y t, el polinomio *x*2 − 3*x* + 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](#_bookmark51) para que (también) acepte la negación, p.ej., −*x*2 − 3*x* + 2.
      4. Modifica polynomial/2 para que acepte términos con fracciones de poli-

nomios, p.ej., *x*3 +

*x*2 − 3*x*

*x* − 3 + 2.

### Calcular la derivada.

[t,fragile]

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

 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.

1 % derivative(Exp, X, Dif): Dif is the derivative of Exp with respect to X.

2 derivative(X, X, s(0)).

3 derivative(N, \_, 0) :- nat(N).

4 derivative(X^s(N), X, s(N)\*X^N).

5 derivative(sin(X), X, cos(X)).

6 derivative(cos(X), X, -sin(X)).

7 derivative(e^X, X, e^X).

8 derivative(log(X), X, s(0)/X).

9 derivative(F+G, X, DF+DG) :- derivative(F,X,DF), derivative(G,X,DG).

10 derivative(F-G, X, DF-DG) :- derivative(F,X,DF), derivative(G,X,DG).

11 derivative(F\*G, X, F\*DG+DF\*G) :- derivative(F,X,DF), derivative(G,X,DG).

12 derivative(F/G,X,(G\*DF-F\*DG)/(G\*G)) :- derivative(F,X,DF), derivative(G,X,DG).

[t,fragile]

 Como no hemos especificado como simplificar expresiones, la derivada de la expresión 3*x* + 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 co- ciente), sin embargo, no está la regla de la cadena.

 Resuelve los siguientes ejercicios:

* + - 1. Define la regla de la cadena para la derivada de *xN* y *sin*(*X* ).
      2. Si en lugar de sin(X) usamos el funtor unary\_term(sin,X):
         1. Define una única regla de la cadena.
         2. Adapta el resto de reglas a la nueva representación.

### Torres de Hanoi.

[t,fragile]

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):

1 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]. [t,fragile]

 Aún cuando queramos interpretar hanoi/5 como una relación, es evidente su se- mántica “operacional”[6](#_bookmark54) –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 com- prueba si el programa funciona correctamente (explica porqué).

### Satisfabilidad de fórmulas Booleanas

[t,fragile]

 Una fórmula Booleana es un término definido como:

* Las constantes *true* y *f alse* son fórmulas Booleanas.
* Si *X* y *Y* son fórmulas Booleanas también lo son:

*X* ∧*Y* , *X* ∨*Y* , y ¬*X*

donde ∧ y ∨ son operadores binarios infijos para la conjunción y la disyun- ción, y ¬ es el operador prefijo unario para la negación.

 Primero definiremos los operadores /\, \/ y ~ respectivamente:

1 :- op(200, fy, ~).

2 :- op(400, xfy,[/\, \/]).

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

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](#_bookmark55)

[t,fragile]

 El programa para determinar si una fórmula Booleana es verdadera o falsa re- quiere 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.

7Detalles sobre el uso de op/3 disponibles en la documentación de [Ciao](https://ciao-lang.org/ciao/build/doc/ciao.html/operators.html) y [Swi](https://www.swi-prolog.org/pldoc/man?predicate=op/3).



# Cápitulo 3

**Programación lógica avanzada**

* 1. Modelo computacional [**44**](#_bookmark57)
  2. Operador corte [**46**](#_bookmark58)
  3. Negación [**47**](#_bookmark59)
     1. Answer Set Programming [48](#_bookmark60)
  4. Memorización [**48**](#_bookmark61)
     1. Acumulador [48](#_bookmark62)
     2. Tabulación [49](#_bookmark65)
  5. Aritmética [**51**](#_bookmark67)
     1. Limitaciones y CLP [51](#_bookmark68)
  6. Inspeccionar estructuras, entrada/salida... [**52**](#_bookmark70)
  7. Algoritmos de ordenamiento [**53**](#_bookmark71)
  8. Meta-interprete [**54**](#_bookmark74)
  9. Trucos eficientes [**55**](#_bookmark75)
  10. Recolección de soluciones [**56**](#_bookmark76)
  11. Predicados de orden superior [**57**](#_bookmark78)
  12. Agregación de respuestas dinámica [**59**](#_bookmark79)
      1. Motivación [59](#_bookmark80)
      2. Agregación de ?- dist(a, Y, D) [59](#_bookmark81)
      3. Codificación de Agregados en Ciao [60](#_bookmark82)
      4. Ejemplos de uso de agregados [61](#_bookmark83)
  13. Introducción [**63**](#_bookmark84)
      1. Programación con Restricciones [63](#_bookmark85)
      2. Programación Lógica con Restricciones [64](#_bookmark86)
      3. Ejemplo resolución de restricciones [64](#_bookmark87)
  14. Sistema de restricciones [**66**](#_bookmark88)
      1. Definición Formal [66](#_bookmark89)
      2. Dominios de restricciones [67](#_bookmark90)
  15. Resolutor de restricciones: Ejemplo: CLP(R) [**68**](#_bookmark91)
      1. Ejemplo: CLP(R) [68](#_bookmark92)
      2. Fibonacci usando CLP(R) [69](#_bookmark93)
  16. Reduce el espacio de búsqueda [**69**](#_bookmark94)
      1. Árbol de derivación [70](#_bookmark95)

## Modelo computacional

[t,fragile]

 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.

1 % Version a

2 son(S,P) :- father(P,S), male(S).

3 % Version b

4 son(S,P) :- male(S), father(P,S).

[t,fragile]

 Supongamos la siguiente base de datos:

1 father(abraham, isaac). male(isaac).

2 father(haran, lot). male(lot).

3 father(haran, milcah). female(milcah).

4 father(haran, yiscah). female(yiscah).

Árbol de búsqueda versión a:

son(S,haran)

father(hSa=ryains,cSa)h, male(S)

S=milcah

S=lot

male(lot) male(yiscahmale(milcah)

[] fail fail

son(S,haran)

Árbol de búsqueda versión b: [t,fragile]

#### Conclusión 1:

male(S), father(haran,S)

S=isaac S=lot father(haran,isaacf)ather(haran,lot)

fail []

* + 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 en- contrar soluciones:

* + Hay una rama infinita a la izq. de una rama con respuestas.
  + Como ya hemos explicado esto se puede resolver usando busqueda en an- chura (paquete sr/bfall ([Cabeza, Carro y Hermenegildo](#_bookmark105) )).

 **Conclusión 2**: Este modelo computacional nos permite definir una semántica operacional (procedimental).

[t,fragile]

 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 Q1, Q2, *. . .*, Q*m*.
  + Para probar Q*i*, *. . .*, Q*m*:
    - Encuentra una cláusula H :- B1, *. . .*, B*n* tal que Q*i* y H unifican (con un umg).
    - Bajo la substitución (del umg) de las variables en la cláusula, probar

(recursivamente) B1, *. . .*, B*n*, Q*i*+1, *. . .*, Q*m*.

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

## Operador corte

[t,fragile]

 Basándonos en la semántica operacional de Prolog definimos un operador, lla- mado 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:

1 % Avoid redundancies

2 is\_father(P) :- father(P,S), !.

3

4 member(X, [X|\_]) :- !.

5 member(X, [\_|L]) :- member(X,L)

Ejemplo de corte rojo:

1 % Prune search of ?- minimum(s(0),s(s(0)),X)

2 minimum(X,Y,X) :- leq(X,Y), !.

3 minimum(X,Y,Y).

4 % However, ?- minimum(s(0),s(s(0)),s(s(0)))

5 % succeeds incorrectly

[t,fragile]

 Otra de las ventajas del corte es su capacidad para “eliminar” pun- tos 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.

1 % polynomial(Expresion,X): Expresion is a polynomial in X.

2 polynomial(X, X) :- !.

3 polynomial(Term, X) :- nat(Term), !.

4 polynomial(Term1 + Term2, X) :- !, polynomial(Term1, X), polynomial(Term2, X).

5 polynomial(Term1 - Term2, X) :- !, polynomial(Term1, X), polynomial(Term2, X).

6 polynomial(Term1 \* Term2, X) :- !, polynomial(Term1, X), polynomial(Term2, X).

7 polynomial(Term1 / Term2, X) :- !, polynomial(Term1, X), nat(Term2).

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

## Negación

[t,fragile,label=neg]

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

∀*x* ( *f lies*(*x*) ← *bird*(*x*) ∧¬*abnormal*\_*bird*(*x*) )

∀*x* ( *abnormal*\_*bird*(*x*) ← *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](#_bookmark107))).

* 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.

[t,fragile]

 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).

### Answer Set Programming

[t,fragile]

 Answer Set Programming (ASP): Basada en la semántica de modelos estables ([Gelfond y Lifschitz 1988](#_bookmark109)).

* + - * 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](#_bookmark63)

**s(CASP):** evaluación *Goal-directed* de ASP, implementado en Prolog.[2](#_bookmark64)

## Memorización

[t,fragile]

 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:
   1. Añadiendo un argumento.
   2. 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](#_bookmark100)) (ver pág. [3.12](#_bookmark79)).

### Acumulador

[fragile,t]

1Playground online de clingo disponible en <https://potassco.org/clingo/run/>

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

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

1 reach(X,Y) :-

2 reach\_aux(X,Y,[X]).

3 reach\_aux(X,Y,Visitados) :-

4 edge(X,Z),

5 \+ is\_member(Z,Visitados),

6 reach\_aux(Z,Y,[Z|Visitados]).

7 reach\_aux(X,Y,\_) :-

8 edge(X,Y).

9

10 is\_member(X,[X|\_]).

11 is\_member(X,[\_|T]) :-

12 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 res- puestas esperadas:



1

3

2

[fragile,t]

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

1 reach\_aux(X,Y,Vs) :- edge(X,Z), \+ is\_member(Z,Vs), reach\_aux(Z,Y,[Z|Vs]).

* + - 1. Dado el siguiente programa en Prolog

1 reach(X,Y) :-

2 reach\_aux(X,Y,[X]).

3 reach\_aux(X,Y,Visitados) :-

4 edge(X,Z),

5 \+ is\_member(Z,Visitados),

6 reach\_aux(Z,Y,[Z|Visitados]).

7 reach\_aux(X,Y,\_) :-

8 edge(X,Y).

9 edge(1,2).

10 edge(2,1).

11 edge(2,3).

12

13 is\_member(X,[X|\_]).

14 is\_member(X,[\_|T]) :-

15 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.

### Tabulación

[fragile,t]

 La tabulación (tabling en inglés, ver Tamaki y Sato ([1986](#_bookmark119)) y Warren ([1992](#_bookmark121))) 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(tabli6ngr)e)a.ch(X,Y) :-



1

2

2 :- table reach/2.

3

7 edge(X,Z),

8 reach(Z,Y).



10

4 edge(1,2).

5 edge(2,1).

reach(1,Y)9**1**



reach(X,Y) :-

edge(X,Y).

edge(1,Z),reach(Z,Y)**3** edge(1,Y)**13** Z=2 



Y=2

* Con :r-eatcahb(l2,eY()l**4**ínea 2) definimedogsel(o1s,2p)re**1**d**4**icados a tabular.



Y=1

[efrdaggei(le2,,tZ]'),reach(Z',Y)**6** edge(2,1)**8** Z'=1 

[]**15**

reach(1,Y)**7**



Y=2

[]**9**

**2** Guardamos reach(1,Y) en la tabla.

**7** Suspendemos la evaluación de reach(1,Y).

repeated answer**11**



Y=1

*f ail***12**

[fragile,t]

new answer**16**

[]**17**

**9** Guardamos Y=1 para reach(2,Y) y

reach(1,Y).

**11** Consumimos 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.

 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 compu- tación, teoría de juegos, análisis bursátil, etc.

 Dado fib(N,F), que verifica que F es el N-esimo término de la sucesión de Fibo- nacci.

 ...al tabular fib/2 pasamos de complejidad O(2*n*) a complejidad lineal O(n).

 Implementación de fib(N,F) con tabulación.[3](#_bookmark66)

1 :- use\_package(library(tabling)).

2 :- table fib/2.

3La declaración use\_package no es necesaria en Swi (online/local) pero si en Ciao (solo funciona en local).

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.

## Aritmética

[fragile,t]

 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](#_bookmark69) que se evalúa y el resultado 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).

### Limitaciones y CLP

[fragile,t]

 Ahora podemos escribir los predicados aritméticos usando las funciones del sis- tema. 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.

4Detalles sobre los operadores aritméticos disponibles en la documentación de [Ciao](https://ciao-lang.org/ciao/build/doc/ciao.html/arithmetic.html) y [Swi](https://www.swi-prolog.org/pldoc/man?section=arith).

 **Solución:** Afortunadamente existe una extensión de Prolog llamada restriccio- nes (ver Tema 3.4, pág. [3.13](#_bookmark84)) 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.

## Inspeccionar estructuras, entrada/salida...

[fragile,t]

 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])

[fragile,t]

 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 corres- pondiente en [Ciao](https://ciao-lang.org/ciao/build/doc/ciao.html/streams.html) y [Swi](https://www.swi-prolog.org/pldoc/man?section=IO))

P.ej.: current\_output(StreamA), open(File,Mode,StreamB), set\_output(StreamB), “escribes en el archivo”, close(StreamB), set\_output(StreamA).

[fragile,t]

 Para imprimir en el top-level (o en ficheros) tenemos:

* display(Term): imprime Term por el stream actual.[5](#_bookmark72)
* 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](https://ciao-lang.org/ciao/build/doc/ciao.html/read.html) y [Swi](https://www.swi-prolog.org/pldoc/man?section=termrw).

* ...también hay predicados para leer caracteres (get\_char(X)) o incluso by- tes (get\_byte(X)), p.ej., el char A es el byte 65.

## Algoritmos de ordenamiento

[t,fragile]

 Un procedimiento naive para ordenar una lista Xs consiste en encontrar una per- mutación 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](#_bookmark73)

5Esta implementado usando una versión con un argumento extra, i.e.,

current\_output(S), display(S,Term).

6Por otro lado la programación lógica con restricciones (pág. [3.13](#_bookmark84)) representa el paradigma **restringe y genera**.

* 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.

[t,fragile]

 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 re- sultado 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.

## Meta-interprete

[t,fragile]

 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 re- suelve correctamente.

 (Ejercicio) Extiende el meta-interprete para que soporte el operador aritmético

is/2 (Pista, delega en Prolog la evaluación de is/2).

## Trucos eficientes

[t,fragile]

 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**.

1. Prolog permite construir **estructuras de datos avanzadas** con facilidad: árboles ordenados, estructuras anidadas, etc.
2. Usas la unificación para manipular los datos:

1 three\_elements\_a(L) :- length(L,N), N=3.

2 three\_elements\_b([\_,\_,\_]). % Better

1. Evita programas no-deterministas [t,fragile]
2. 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:

1 plus\_a(X,Y,N) :- var(X), var(Y), between(0,N,X), between(O,N,Y), N is X + Y.

2 plus\_b(X,Y,N) :- var(X), var(Y), between(0,N,X), Y is N - X.

% Better %

 Listas ordenadas + unificación:

1 equal\_sets\_a(S1, S2) :- \+ (member(X,S1), \+ member(X,S2) ),

2 \+ (member(X,S2), \+ member(X,S1) ).

3 equal\_sets\_a(S1, S2) :- sort(S1,Sort), sort(S2,Sort).

% Better %

1. Ordenar los predicados para podar la búsqueda (más detalles en pág. [3.13](#_bookmark84)). [t,fragile]
2. 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.

1 greater\_a(\_, []).

2 greater\_a(X, [Y|Ys]) :- X > Y, greater\_b(X,Ys).

3 greater\_b([], \_). % Better %

4 greater\_b([Y|Ys], X) :- X > Y, greater\_b(X,Ys). %

1. Transformar recursión en iteración (recursión por cola).

Cuando la llamada recursiva es el último objetivo y no hay alternativas.

1 sum\_a([], 0).

2 sum\_a([N|Ns], Sum) :- sum\_a(Ns, Ss), Sum is N + S1.

3 sum\_b(L, Out) :- sum\_inter(L, 0, Out).

% Better %

4 sum\_iter([], In, In).

%

5 sum\_iter([N|Ns], In, Out) :- Mid is N + In, sum\_iter(Ns, Mid, Out).

%

## Recolección de soluciones

[t,fragile]

 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](#_bookmark77)

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.

[t,fragile]

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

7Consulta la documentación de [Ciao](https://ciao-lang.org/ciao/build/doc/ciao.html/aggregates.html) y [Swi](https://www.swi-prolog.org/pldoc/man?section=allsolutions) para más detalles.

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] ?;

[t,fragile]

1. 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] ?;

1. 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] ?;

## Predicados de orden superior

[t,fragile]

 Predicados de orden superior reciben como argumento otro predicado.

* Implementados usando call/1, ya mencionado en pág. [3.3](#_bookmark59).  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. ¿Por- que solo tiene dos argumentos?

## Agregación de respuestas dinámica

[fragile,t,label=agregados]

 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. set(*x* | *p*(*X* )) = {a,b,c}
* *suma* de una lista de números. 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*.
* Cada nueva respuesta se agrega con el anterior agregado y se guarda.
* Incrementa eficiencia y terminación, reduciendo memoria necesaria.

### Motivación

[fragile,t]

 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) }

* Solución: *minimizar* el argumento D en dist(X,Y,D).
  + Usando la directiva :- agg\_entail dist(\_,\_,=<).
    1. **Agregación de** ?- dist(a, Y, D)

[fragile,t]

Z=b,D1=2

Y=b, D=4

dist(a,Y,D)**1** Y=b,

dist(b,Y,D2), D is 2+D2**15**

[]**3**

D=2

[]**4**

Z=b, D1=4

Y=a,D2=3

...**16**

Y=b,D2=5

...**18**

dist(b,Y,D2), D is 4+D2**5**

D=5

D=7

Y=a, D2=3

[]**7**, D is 4+3**8**

Z'=a, D1'=3, D'=D2

[]**17**

*f ail***19**

D=7

[]**9**

dist(a,Y,D2'), D2 is 3+D2', D is 4+D2**10** Y=b, D2'=2 Y=a,D2'=7

D2=5,D=9 ...**13**

...**11**

**3** Guardamos Y=b, D=4 en la tabla.

**4** Sustituimos la respuesta porque

*min*(4*,* 2) = 2.

[]*, f ail***12**

D2=10,D=14 

*f ail, f ail***14**

**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.

### Codificación de Agregados en Ciao

[fragile,t]

Ejemplo de codificación de *entailment-based* agregados.

Aggregate Code for entailment checking

minimum among num- =<

bers

maximum among num- >=

bers

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).

[fragile,t]

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 inter- val

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).

[fragile,t]

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 add | last(\_,\_)  add(\_,\_) is A + B. | :-  :- | fail. fail. | last(\_,B,B).  add(A,B,C) :- C |
| multiplication | mult(\_,\_) is A \* B. | :- | fail. | mult(A,B,C) :- C |

### Ejemplos de uso de agregados

[fragile,t]

* + - 1. El problema de los Juegos. ICLP 2015 LP/CP contest ([ALP 2015](#_bookmark98)).
         * 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** |

[fragile,t]

* + - 1. **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

P0(d) = 0.7 \* 0.8 = 0.56

P1(d) = 0.7 \* 0.2 \* 0.8 = 0.112

P2(d) = 0.7 \* 0.2 \* 0.2 \* 0.8 = 0.0224



P(d) = 0.7

...

P*n*(d) = 0.7 \* 0.2*n*

\* 0.8 = ... 

1 :- use\_package(tclp\_aggregates). 8 reach(N,P) :- path(a,N,P).

2 :- agg\_join reach(\_,add). 9

3 :- agg\_entail path(\_,\_,thr(0.0011)0).path(X,Y,P) :- edge(X,Y,P).

4

5 add(\_,\_) :- fails.

6 add(A,B,New) :- New is A + B.

11 path(X,Y,P) :- edge(X,Z,P1),

12 path(Z,Y,P2),

13 P is P1 \* P2.

7 thr(Epsilon,A,B) :- A < Epsilon \* B.

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

## Introducción

[label=clp]

 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 estáti- camente.
* 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](#_bookmark103)): Z3, Microsoft Solver Foundation, Choco, JaCoP, Google’s Operations Research Tools and OptiMathSAT.

### Programación con Restricciones

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.

### Programación Lógica con Restricciones

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:

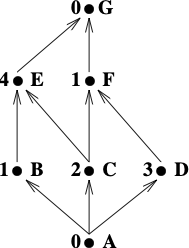
* Paso de parámetros bidireccional (más que la coincidencia de patrones).
* Asignación única.
* No se necesita una gestión explícita de la memoria.
* Estructuras de datos fáciles de construir.

 Ofrece no-determinismo gracias a su procedimiento de búsqueda.

### Ejemplo resolución de restricciones

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* ∈ {1*,...,* 10}

*a* ≤ *b, c, d*

*b* + 1 ≤ *e*

*c* + 2 ≤ *e*

*c* + 2 ≤ *f*

*d* + 3 ≤ *f*

####

*f* + 1 ≤ *g*

Ejemplo Resolución de Restricciones (cont.)

 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.

6

*c* +72 ≤ *e*0*.,*7 8

9

10

3*.,*9

0*.,*5

*c* + 2 ≤ *e* 0*.,*4

0*.,*6

11 0*.,*4

*Sol.* 0*.,*4 0*.,*5 0*.,*4 0*.,*6 2*.,*6 3*.,*9 6*.,*10

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *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 |  |  | *<* 3 *>* 1*.,*10 |  |  |
| 2 |  |  | 0*.,*8 |  | 2*.,*10 |  |  |
| 3 |  |  |  |  |  | 2*.,*10 |  |
| 4 |  |  |  | 0*.,*7 |  | 3*.,*10 |  |
| 5 |  |  |  |  | *<* 4 *>* 2*.,*6 |  | 6*.,*10 |

Ejercicio

* + - 1. Completa la modelización del ejemplo anterior con la inecuación que falta (in- dicada 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?

## Sistema de restricciones

Sistema de Restricciones

 Las restricción son condiciones que debe cumplir una solución:

* *X* +*Y* = 20
* *X* ∧*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 (+, ∗, ∧, ∨).
* Restricciones permitidas: (in)ecuaciones (=, ̸=, ≤, ≥, *<*, *>*).
* Algoritmos de resolución de restricciones: simplex, gauss, etc.

 Soluciones: asignaciones a variables y/o restricciones entre variables.

### Definición Formal

Sistema de Restricciones: Definición Formal

 Un *esquema de programación lógica de restricciones*, CLP(X ), sobre un domi- nio de restricciones (D, L) se define instanciando el parámetro X , que representa la 4-tupla (Σ, D, L, T ) donde:

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

D es una Σ-estructura: la estructuras sobre la que se evalúa.

L es una clase de Σ-fórmulas: el conjunto de restricciones que se pueden escribir.

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

 Y se cumple que:

* L está construido sobre un lenguaje de primer orden.

• =∈ Σ y = es identidad en D.

* Hay restricciones idénticamente falsas e idénticamente verdaderas en L.
* L es cerrado en cuanto a renombrado de variables, conjunción y cuantifi- cación existencial.

Ver Jaffar y Maher ([1994](#_bookmark114)).

### Dominios de restricciones

Sistema de Restricciones: Dominios

R Aritmética sobre Reales:

Σ = {0*,* 1*,* +*,* ∗*,* =*, <,* ≤} D = Reales interpretando Σ normalmente. P.ej.: *x*2 + 2*xy < y* ∧ *x >* 0

*x*

≡ *xxx* + *xxy* + *xxy < y* ∧ 0 *< x*

R*Lin* Aritmética Lineal:

Σ = {0*,* 1*,* +*,* =*, <,* ≤} D = Reales interpretando Σ normalmente. P.ej.: 3*x* − *y <* 3

≡ *x* + *x* + *x <* 1 + 1 + 1

R*LinEq* Ecuaciones lineales:

Σ = {0*,* 1*,* +*,* =} D = Reales interpretando Σ normalmente. P.ej.: 3*x* + *y* = 5 ∧ *y* = 2*x*

≡ *x* = 1 ∧ *y* = 1 + 1

Equivalentes dominios pueden definirse sobre los Racionales (Q).

Sistema de Restricciones: Dominios (cont.)

FT Árboles finitos o dominio de Herbrand:1

Σ = {*a, b, f /n, g/m,...,* =} D = Árboles finitos donde:

* Cada *f* ∈ Σ 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*))

≡ *Z* = *a* ∧*Y* = *h*(*a*)

S*tring* Ecuaciones sobre strings:

Σ = {*a, b, . . . , λ,.,* =} D = string donde:

* *.* se interpreta como concatenación de strings. P.ej.: *X.A.X* = *X.A*

≡ *X* = *λ*

1Prolog puede verse como una sistema de restricciones sobre términos de Herbrand con un único símbolo de restricción =.

Sistema de Restricciones: Ejercicio

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

R.

Q*Lin*. R*LinEq*. Q.

FT.

S*tring*.

B*ool*.

## Resolutor de restricciones: Ejemplo: CLP(R)

### Ejemplo: CLP(R)

[fragile,t]

 CLP(R) por Holzbaur ([1995](#_bookmark113)): lenguaje basado en Prolog + resolución de restric- ciones sobre los reales, 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.

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

 Versión en Prolog

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

 Versión en CLP(R)

1 :- use\_package(clpr).

2 suma(X,Y,Z) :- Z .=. X + Y, Z .>. 0.

### Fibonacci usando CLP(R)

[fragile,t]

 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 com- porta como la versión con la aritmética de Peanno:

1 :- use\_package(clpr).

2 fib(0,0).

3 fib(1,1).

4 fib(N,F) :-

5 N .>. 1, F1 .>=. 0, F2 .>=. 0,

6 N1 .=. N - 1, N2 .=. N - 2,

7 F .=. F1 + F2,

8 fib(N1 ,F1), fib(N2 ,F2).

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

## Reduce el espacio de búsqueda

[fragile]

 Encontrar tres números consecutivos de la relación p/1.

De “genera y prueba” con Prolog.

1 p(11). p(3). p(7).

2 p(16). p(15). p(14).

3

4 test(X, Y, Z) :-

5 Y is X + 1,

6 Z is Y + 1.

7 solution(X, Y, Z) :-

8 p(X), p(Y), p(Z),

9 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.

1 :- use\_package(clpr).

2 p(11). p(3). p(7).

3 p(16). p(15). p(14).

4

5 test(X, Y, Z) :-

6 Y .=. X + 1,

7 Z .=. Y + 1.

8 solution(X, Y, Z) :-

9 test(X, Y, Z),

10 p(X), p(Y), p(Z).

 CLP(R) solo necesita 11 pasos!

### Árbol de derivación

[fragile]

solution(X,Y,Z)**1** Y=X+1

Z=Y+1 p(X),p(Y),p(Z)**2**

X=11,Y=12,Z=13 X=3

...

X=7

...

X=16

...

X=15

...

X=14,Y=15,Z=16

p(12),p(13)**3**

p(4),p(5)**4**

p(8),p(9)**5**

p(17),p(18)**6** p(16),p(17)**7** p(15),p(16)**8**

*f ail f ail f ail f ail* p(17)**9** p(16)**10**

*f ail* []**11**

[fragile]

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

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

2

3 test(X, Y, Z) :- Y is X + 1, Z is Y + 1.

4 solution(X, Y, Z) :- p(X), p(Y), p(Z), test(X, Y, Z).

* + - 1. Dado el programa anterior, indica la principal diferencia con respecto a la versión con CLP.

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

2

3 test(X, Y, Z) :- Y .=. X + 1, Z .=. Y + 1.

4 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 /](https://www.cs.nmsu.edu/ALP/2015/09/report-2015-lpcp-programming-contest/) [2015/09/report-2015-lpcp-programming-contest/](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](https://doi.org/10.1007/978-3-031-35254-6_8).

Arias, Joaquín (2022). **Lógica: desde Aristóteles hasta Prolog**. Servicio de Publica-

ciones 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. Sym- posium on Practical Aspects of Declarative Languages*. Vol. 11372. LNCS. Sprin- ger, págs. 98-114. DOI: [10.1007/978-3-030-05998-9\_7](https://doi.org/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 /](https://doi.org/10.1017/S1471068418000285) [S1471068418000285](https://doi.org/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 calcu- lus: a core calculus for deterministic functional logic programming**. En: *Pro- ceedings of the ACM on Programming Languages* 7(ICFP), págs. 417-447. DOI: [10.1145/3607845](https://doi.org/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.](https://www.se.cs.uni-saarland.de/theses/MartinBauerMA.pdf) [pdf](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 Politéc- nica de Madrid. URL: <https://oa.upm.es/47203>.

Cabeza, Daniel, Carro, Manuel y Hermenegildo, Manuel (s.f.). **Breadth-first execu-**

**tion**. 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](https://doi.org/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-](https://doi.org/10.1007/978-1-4684-3384-5\_11) [5\\_11](https://doi.org/10.1007/978-1-4684-3384-5\_11).

BIBLIOGRAFÍA

Colmerauer, Alain y Roussel, Philippe (1996). **The birth of Prolog**. En: *History of* *programming languages—II*, págs. 331-367. DOI: [10.1145/234286.1057820](https://doi.org/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](https://doi.org/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/](https://shorturl.at/V57pU) [V57pU](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](https://doi.org/10.1017/S1471068411000457). URL: [http://ciao-lang.org](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: *Jour- nal of Logic Programming* 19/20, págs. 503-581. DOI: [10.1016/0743- 1066(94)](https://doi.org/10.1016/0743-1066(94)90033-7) [90033-7](https://doi.org/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](https://doi.org/10.1145/359131.359136).

Kowalski, Robert y Kuehner, Donald (1971). **Linear resolution with selection fun-**

**ction**. En: *Artificial Intelligence* 2(3-4), págs. 227-260. DOI: [10 . 1016 / 0004 -](https://doi.org/10.1016/0004-3702(71)90012-9) [3702(71)90012-9](https://doi.org/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 /](https://doi.org/10.1145/321250.321253) [321250.321253](https://doi.org/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.](https://doi.org/10.48786/edbt.2024.84) [84](https://doi.org/10.48786/edbt.2024.84).

Tamaki, H. y Sato, M. (1986). **OLD Resolution with Tabulation**. En: *Third Inter- national 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 Entschei- dungs problem**. En: *Proceedings of the London Mathematical Society Series/2 (42)*, págs. 230-42. DOI: [10.1112/plms/s2-42.1.230](https://doi.org/10.1112/plms/s2-42.1.230). URL: [https://shorturl.](https://shorturl.at/4ZulB) [at/4ZulB](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 pro- grams**. En: *Research Reports 39 and 40, Dpt. of Artificial Intelligence, Univ. of Edinburgh*. URL: <https://shorturl.at/2tSVh>.