

Joaquín Arias

Programación Lógica: hastra Prolog y más allá

Curso 2024-2025

<https://burjcdigital.urjc.es/handle/10115/41543>



Universidad
Rey Juan Carlos

Escuela Técnica Superior
Ingeniería Informática

Tema 0: Programación Lógica vs. Funcional

Asignatura: Programación Declarativa

Joaquín Arias

Universidad Rey Juan Carlos
Escuela Técnica Superior de Ingeniería Informática

Curso 2024-2025

Índice I

Tema 0.1: Programación Declarativa.

Contexto

Programación Funcional

Características y Ventajas

Programación Lógica

Cláusulas Horn

Prolog

Hay algo más ...

... mucho más



Copyright ©2024 Joaquín Arias.

Esta obra, basada en transparencias de Ana Pradera (URJC'23) & ClipLab (UPM'24), está bajo la licencia CC BY-SA 4.0, Creative Commons Atribución-CompartirIgual 4.0 Internacional. Cómo citar esta obra: Arias, Joaquín (2024). Programación Lógica: hasta Prolog y más allá. BURJC, Madrid.

Índice II

Tema 0.2: **Algunos problemas interesantes.**

El problemas de las garrafas

El problema del puente y la linterna

Las Torres de Hanoi

El problema del viajante

El problema de las n-reinas

Zebra puzzle (Einstein riddle)



Tema 0.1: Programación Declarativa.

Contexto

- Qué es la programación declarativa?
 - Es un paradigma de programación diferente a la imperativa (**R**) o a la orientada a objetos (**Java**).
 - Los programas especifican las propiedades de los problemas a resolver.
 - La ejecución de un programa consiste en “encontrar” las solución(es).

Contexto

- Qué es la programación declarativa?
 - Es un paradigma de programación diferente a la imperativa (**R**) o a la orientada a objetos (**Java**).
 - Los programas especifican las propiedades de los problemas a resolver.
 - La ejecución de un programa consiste en “encontrar” las soluciones.

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])
  
```

Contexto

- Qué es la programación declarativa?
 - Es un paradigma de programación diferente a la imperativa (**R**) o a la orientada a objetos (**Java**).
 - Los programas especifican las propiedades de los problemas a resolver.
 - La ejecución de un programa consiste en “encontrar” las solución(es).
- Ejemplos de lenguajes de programación declarativos:
 - Algebraicos: Maude, SQL.
 - Lógicos: Prolog, ASP, Logica by Google.
 - Funcionales: Haskell, Scala by EPFL.

Contexto

- Qué es la programación declarativa?
 - Es un paradigma de programación diferente a la imperativa (**R**) o a la orientada a objetos (**Java**).
 - Los programas especifican las propiedades de los problemas a resolver.
 - La ejecución de un programa consiste en “encontrar” las soluciones.
- Ejemplos de lenguajes de programación declarativos:
 - Algebraicos: Maude, SQL.
 - Lógicos: Prolog, ASP, Logica by Google.
 - Funcionales: Haskell, Scala by EPFL.

A continuación compararemos Haskell y Prolog



Programación Funcional: Introducción

Programación Funcional: Introducción

- La programación funcional está basada en funciones matemáticas.
- Función: Una función es una regla de correspondencia entre dos conjuntos de modo que a cada elemento del primer conjunto le corresponde un y solo un elemento del segundo conjunto.
- Cualquier función computable puede expresarse y evaluarse con el Lambda Calculus.
- Church usó Lambda Calculus para resolver el Entscheidungsproblem (1936):
 - No existe ningún algoritmo que determine si dos expresiones lambda arbitrarias son equivalentes.

Programación Funcional: Lambda Calculus

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

Programación Funcional: Lambda Calculus

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

Evaluando λ -expressions

Función identidad aplicada a 3: $((\lambda x.x) 3) \equiv 3$

Función suma aplicada a 2 y 3: $((\lambda x.\lambda y.x+y) 2) 3 \equiv ((\lambda y.2+y) 3) \equiv (2+3)$

Función identidad aplicada a suma: $((\lambda x.x) (\lambda x.\lambda y.x+y)) \equiv (\lambda x.\lambda y.x+y)$

Programación Funcional: Haskell

- Debe su nombre a Haskell Curry (1900-1982).
- Dada una función f de tipo $f : (X_1 \times X_2 \times \dots \times X_n) \rightarrow Z$ decimos que su currificación es:

- Una secuencia de funciones con un único argumento:

$$\text{curry}(f) : X_1 \rightarrow X_2 \rightarrow \dots \rightarrow X_n \rightarrow Z.$$



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]
```

Programación Funcional: Booleanos en Lambda Calculus

- Primero implementamos las expresiones If-then-else, True and False:
 - If-then-else: $\lambda x.\lambda y.\lambda z.x\ y\ z$
 - true: $\lambda x.\lambda y.x$
 - false: $\lambda x.\lambda y.y$

Programación Funcional: Booleanos en Lambda Calculus

- Primero implementamos las expresiones If-then-else, True and False:
 - If-then-else: $\lambda x.\lambda y.\lambda z.x\ y\ z$
 - true: $\lambda x.\lambda y.x$
 - false: $\lambda x.\lambda y.y$

If-then-else True P Q \equiv

Programación Funcional: Booleanos en Lambda Calculus

- Primero implementamos las expresiones If-then-else, True and False:
 - If-then-else: $\lambda x.\lambda y.\lambda z.x\ y\ z$
 - true: $\lambda x.\lambda y.x$
 - false: $\lambda x.\lambda y.y$

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

Programación Funcional: Booleanos en Lambda Calculus

- Primero implementamos las expresiones If-then-else, True and False:
 - If-then-else: $\lambda x.\lambda y.\lambda z.x\ y\ z$
 - true: $\lambda x.\lambda y.x$
 - false: $\lambda x.\lambda y.y$

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

Programación Funcional: Booleanos en Lambda Calculus

- Primero implementamos las expresiones If-then-else, True and False:
 - If-then-else: $\lambda x.\lambda y.\lambda z.x\ y\ z$
 - true: $\lambda x.\lambda y.x$
 - false: $\lambda x.\lambda y.y$

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

Programación Funcional: Booleanos en Lambda Calculus

- Primero implementamos las expresiones If-then-else, True and False:
 - If-then-else: $\lambda x.\lambda y.\lambda z.x\ y\ z$
 - true: $\lambda x.\lambda y.x$
 - false: $\lambda x.\lambda y.y$

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

- Implementación usando Haskell:

```

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

Programación Funcional: Booleanos en Lambda Calculus (cont.)

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

Programación Funcional: Booleanos en Lambda Calculus (cont.)

- Después, basado en estas expresiones definimos And, Or and Not:

- And: $\lambda p.\lambda q.p\ q\ \text{false} \equiv \lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y)$

- Or: $\lambda p.\lambda q.p\ \text{true}\ q \equiv \lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q$

- Not: $\lambda p.p\ \text{false}\ \text{true} \equiv \lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x)$

And True False \equiv

Programación Funcional: Booleanos en Lambda Calculus (cont.)

- Después, basado en estas expresiones definimos And, Or and Not:

- And: $\lambda p.\lambda q.p\ q\ \text{false} \equiv \lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y)$

- Or: $\lambda p.\lambda q.p\ \text{true}\ q \equiv \lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q$

- Not: $\lambda p.p\ \text{false}\ \text{true} \equiv \lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x)$

$$\text{And True False} \equiv (\lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y))\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv$$

Programación Funcional: Booleanos en Lambda Calculus (cont.)

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

And True False $\equiv (\lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y))\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv$
 $(\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2)\ (\lambda x.\lambda y.y) \equiv$

Programación Funcional: Booleanos en Lambda Calculus (cont.)

- Después, basado en estas expresiones definimos And, Or and Not:

- And: $\lambda p.\lambda q.p\ q\ \text{false} \equiv \lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y)$

- Or: $\lambda p.\lambda q.p\ \text{true}\ q \equiv \lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q$

- Not: $\lambda p.p\ \text{false}\ \text{true} \equiv \lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x)$

$$\begin{aligned} \text{And True False} &\equiv (\lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y))\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv \\ &(\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2)\ (\lambda x.\lambda y.y) \equiv (\lambda x_2.\lambda y_2.y_2) \equiv \end{aligned}$$

Programación Funcional: Booleanos en Lambda Calculus (cont.)

- Después, basado en estas expresiones definimos And, Or and Not:

- And: $\lambda p.\lambda q.p\ q\ \text{false} \equiv \lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y)$

- Or: $\lambda p.\lambda q.p\ \text{true}\ q \equiv \lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q$

- Not: $\lambda p.p\ \text{false}\ \text{true} \equiv \lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x)$

$$\begin{aligned} \text{And True False} &\equiv (\lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y))\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv \\ &(\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2)\ (\lambda x.\lambda y.y) \equiv (\lambda x_2.\lambda y_2.y_2) \equiv \text{False} \end{aligned}$$

Programación Funcional: Booleanos en Lambda Calculus (cont.)

- Después, basado en estas expresiones definimos And, Or and Not:

- And: $\lambda p.\lambda q.p\ q\ \text{false} \equiv \lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y)$

- Or: $\lambda p.\lambda q.p\ \text{true}\ q \equiv \lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q$

- Not: $\lambda p.p\ \text{false}\ \text{true} \equiv \lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x)$

$$\begin{aligned} \text{And True False} &\equiv (\lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y))\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv \\ &\quad (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2)\ (\lambda x.\lambda y.y) \equiv (\lambda x_2.\lambda y_2.y_2) \equiv \text{False} \end{aligned}$$

$$\text{Or True False} \equiv$$

Programación Funcional: Booleanos en Lambda Calculus (cont.)

- Después, basado en estas expresiones definimos And, Or and Not:

- And: $\lambda p.\lambda q.p\ q\ \text{false} \equiv \lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y)$

- Or: $\lambda p.\lambda q.p\ \text{true}\ q \equiv \lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q$

- Not: $\lambda p.p\ \text{false}\ \text{true} \equiv \lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x)$

$$\begin{aligned} \text{And True False} &\equiv (\lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y))\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv \\ &\quad (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2)\ (\lambda x.\lambda y.y) \equiv (\lambda x_2.\lambda y_2.y_2) \equiv \text{False} \end{aligned}$$

$$\text{Or True False} \equiv (\lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q)\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv$$

Programación Funcional: Booleanos en Lambda Calculus (cont.)

- Después, basado en estas expresiones definimos And, Or and Not:

- And: $\lambda p.\lambda q.p\ q\ \text{false} \equiv \lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y)$

- Or: $\lambda p.\lambda q.p\ \text{true}\ q \equiv \lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q$

- Not: $\lambda p.p\ \text{false}\ \text{true} \equiv \lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x)$

$$\begin{aligned} \text{And True False} &\equiv (\lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y))\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv \\ &\quad (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2)\ (\lambda x.\lambda y.y) \equiv (\lambda x_2.\lambda y_2.y_2) \equiv \text{False} \end{aligned}$$

$$\begin{aligned} \text{Or True False} &\equiv (\lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q)\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv \\ &\quad (\lambda x_1.\lambda y_1.x_1)\ (\lambda x.\lambda y.x)\ (\lambda x_2.\lambda y_2.y_2) \equiv \end{aligned}$$

Programación Funcional: Booleanos en Lambda Calculus (cont.)

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

And True False $\equiv (\lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y))\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv$
 $(\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2)\ (\lambda x.\lambda y.y) \equiv (\lambda x_2.\lambda y_2.y_2) \equiv \text{False}$

Or True False $\equiv (\lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q)\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv$
 $(\lambda x_1.\lambda y_1.x_1)\ (\lambda x.\lambda y.x)\ (\lambda x_2.\lambda y_2.y_2) \equiv (\lambda x.\lambda y.x) \equiv$

Programación Funcional: Booleanos en Lambda Calculus (cont.)

- Después, basado en estas expresiones definimos And, Or and Not:

- And: $\lambda p.\lambda q.p\ q\ \text{false} \equiv \lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y)$

- Or: $\lambda p.\lambda q.p\ \text{true}\ q \equiv \lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q$

- Not: $\lambda p.p\ \text{false}\ \text{true} \equiv \lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x)$

$$\begin{aligned} \text{And True False} &\equiv (\lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y))\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv \\ &(\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2)\ (\lambda x.\lambda y.y) \equiv (\lambda x_2.\lambda y_2.y_2) \equiv \text{False} \end{aligned}$$

$$\begin{aligned} \text{Or True False} &\equiv (\lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q)\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv \\ &(\lambda x_1.\lambda y_1.x_1)\ (\lambda x.\lambda y.x)\ (\lambda x_2.\lambda y_2.y_2) \equiv (\lambda x.\lambda y.x) \equiv \text{True} \end{aligned}$$

Programación Funcional: Booleanos en Lambda Calculus (cont.)

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

And True False $\equiv (\lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y))\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv$
 $(\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2)\ (\lambda x.\lambda y.y) \equiv (\lambda x_2.\lambda y_2.y_2) \equiv \text{False}$

Or True False $\equiv (\lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q)\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv$
 $(\lambda x_1.\lambda y_1.x_1)\ (\lambda x.\lambda y.x)\ (\lambda x_2.\lambda y_2.y_2) \equiv (\lambda x.\lambda y.x) \equiv \text{True}$

Not True \equiv

Programación Funcional: Booleanos en Lambda Calculus (cont.)

- Después, basado en estas expresiones definimos And, Or and Not:

- And: $\lambda p.\lambda q.p\ q\ \text{false} \equiv \lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y)$

- Or: $\lambda p.\lambda q.p\ \text{true}\ q \equiv \lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q$

- Not: $\lambda p.p\ \text{false}\ \text{true} \equiv \lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x)$

$$\begin{aligned} \text{And True False} &\equiv (\lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y))\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv \\ &(\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2)\ (\lambda x.\lambda y.y) \equiv (\lambda x_2.\lambda y_2.y_2) \equiv \text{False} \end{aligned}$$

$$\begin{aligned} \text{Or True False} &\equiv (\lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q)\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv \\ &(\lambda x_1.\lambda y_1.x_1)\ (\lambda x.\lambda y.x)\ (\lambda x_2.\lambda y_2.y_2) \equiv (\lambda x.\lambda y.x) \equiv \text{True} \end{aligned}$$

$$\text{Not True} \equiv (\lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x))\ (\lambda x_1.\lambda y_1.x_1) \equiv$$

Programación Funcional: Booleanos en Lambda Calculus (cont.)

- Después, basado en estas expresiones definimos And, Or and Not:

- And: $\lambda p.\lambda q.p\ q\ \text{false} \equiv \lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y)$

- Or: $\lambda p.\lambda q.p\ \text{true}\ q \equiv \lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q$

- Not: $\lambda p.p\ \text{false}\ \text{true} \equiv \lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x)$

$$\begin{aligned} \text{And True False} &\equiv (\lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y))\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv \\ &(\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2)\ (\lambda x.\lambda y.y) \equiv (\lambda x_2.\lambda y_2.y_2) \equiv \text{False} \end{aligned}$$

$$\begin{aligned} \text{Or True False} &\equiv (\lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q)\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv \\ &(\lambda x_1.\lambda y_1.x_1)\ (\lambda x.\lambda y.x)\ (\lambda x_2.\lambda y_2.y_2) \equiv (\lambda x.\lambda y.x) \equiv \text{True} \end{aligned}$$

$$\text{Not True} \equiv (\lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x))\ (\lambda x_1.\lambda y_1.x_1) \equiv \dots \equiv$$

Programación Funcional: Booleanos en Lambda Calculus (cont.)

- Después, basado en estas expresiones definimos And, Or and Not:

- And: $\lambda p.\lambda q.p\ q\ \text{false} \equiv \lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y)$

- Or: $\lambda p.\lambda q.p\ \text{true}\ q \equiv \lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q$

- Not: $\lambda p.p\ \text{false}\ \text{true} \equiv \lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x)$

$$\begin{aligned} \text{And True False} &\equiv (\lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y))\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv \\ &(\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2)\ (\lambda x.\lambda y.y) \equiv (\lambda x_2.\lambda y_2.y_2) \equiv \text{False} \end{aligned}$$

$$\begin{aligned} \text{Or True False} &\equiv (\lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q)\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv \\ &(\lambda x_1.\lambda y_1.x_1)\ (\lambda x.\lambda y.x)\ (\lambda x_2.\lambda y_2.y_2) \equiv (\lambda x.\lambda y.x) \equiv \text{True} \end{aligned}$$

$$\text{Not True} \equiv (\lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x))\ (\lambda x_1.\lambda y_1.x_1) \equiv \dots \equiv (\lambda x.\lambda y.y) \equiv$$

Programación Funcional: Booleanos en Lambda Calculus (cont.)

- Después, basado en estas expresiones definimos And, Or and Not:

- And: $\lambda p.\lambda q.p\ q\ \text{false} \equiv \lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y)$

- Or: $\lambda p.\lambda q.p\ \text{true}\ q \equiv \lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q$

- Not: $\lambda p.p\ \text{false}\ \text{true} \equiv \lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x)$

$$\begin{aligned} \text{And True False} &\equiv (\lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y))\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv \\ &(\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2)\ (\lambda x.\lambda y.y) \equiv (\lambda x_2.\lambda y_2.y_2) \equiv \text{False} \end{aligned}$$

$$\begin{aligned} \text{Or True False} &\equiv (\lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q)\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv \\ &(\lambda x_1.\lambda y_1.x_1)\ (\lambda x.\lambda y.x)\ (\lambda x_2.\lambda y_2.y_2) \equiv (\lambda x.\lambda y.x) \equiv \text{True} \end{aligned}$$

$$\text{Not True} \equiv (\lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x))\ (\lambda x_1.\lambda y_1.x_1) \equiv \dots \equiv (\lambda x.\lambda y.y) \equiv \text{False}$$

Programación Funcional: Booleanos en Lambda Calculus (cont.)

- Después, basado en estas expresiones definimos And, Or and Not:

- And: $\lambda p.\lambda q.p\ q\ \text{false} \equiv \lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y)$

- Or: $\lambda p.\lambda q.p\ \text{true}\ q \equiv \lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q$

- Not: $\lambda p.p\ \text{false}\ \text{true} \equiv \lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x)$

$$\begin{aligned} \text{And True False} &\equiv (\lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y))\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv \\ &(\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2)\ (\lambda x.\lambda y.y) \equiv (\lambda x_2.\lambda y_2.y_2) \equiv \text{False} \end{aligned}$$

$$\begin{aligned} \text{Or True False} &\equiv (\lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q)\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv \\ &(\lambda x_1.\lambda y_1.x_1)\ (\lambda x.\lambda y.x)\ (\lambda x_2.\lambda y_2.y_2) \equiv (\lambda x.\lambda y.x) \equiv \text{True} \end{aligned}$$

$$\text{Not True} \equiv (\lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x))\ (\lambda x_1.\lambda y_1.x_1) \equiv \dots \equiv (\lambda x.\lambda y.y) \equiv \text{False}$$

- Implementación usando Haskell (cont.):

```
5 my_and = \x -> \y -> x y false
```

```
6 my_or = \x -> \y -> x true y
```

```
7 my_not = \x -> x false true
```

```
8
```

```
9 k = if_then_else (my_and true false) 3 2           -- Value of k?
```

Programación Funcional: Booleanos en Lambda Calculus (cont.)

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

- And₂: $\lambda p.\lambda q.p\ q\ p$

- Or₂: $\lambda p.\lambda q.p\ p\ q$

- Not₂: $\lambda p.\lambda x.\lambda y.p\ y\ x$

¿Cuántos argumentos tiene?

Programación Funcional: Booleanos en Lambda Calculus (cont.)

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

- And₂: $\lambda p.\lambda q.p\ q\ p$

- Or₂: $\lambda p.\lambda q.p\ p\ q$

- Not₂: $\lambda p.\lambda x.\lambda y.p\ y\ x$

¿Cuántos argumentos tiene?

Deberes

Programación Funcional: Booleanos en Lambda Calculus (cont.)

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

- And₂: $\lambda p. \lambda q. p \ q \ p$

- Or₂: $\lambda p. \lambda q. p \ p \ q$

- Not₂: $\lambda p. \lambda x. \lambda y. p \ y \ x$

¿Cuántos argumentos tiene?

- 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           -- What is the value of k?
```


Programación Funcional: Características y Ventajas

Características:

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

Ventajas:

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

Programación Funcional: Características y Ventajas

Características:

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

Ventajas:

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

Concatenar listas

```

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: Introducción

Programación Lógica: Introducción

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

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

- 1 Se reescribe como el siguientes conjunto de cláusulas:

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

donde el consecuente, $\text{Mortal}(\text{socrates})$, están negado.

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

Programación Lógica: Cláusulas de Horn

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

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

Programación Lógica: Cláusulas de Horn

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

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

Aristóteles:

$Hombre(x) \rightarrow Mortal(x)$

$Hombre(socrates)$

$Mortal(socrates)$

Cláusulas:

$Mortal(x) \vee \neg Hombre(x)$

$Hombre(socrates)$

$\neg Mortal(socrates)$

Prolog:

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

`?- mortal(socrates).`

Programación Lógica: Prolog.

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



SWI-Prolog



Ciao Prolog

Programación Lógica: Prolog.

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



SWI-Prolog



Ciao Prolog

Aristóteles



```

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

?- mortal(socrates).

Contesta **yes** si la “pregunta” es consecuencia lógica (**no** en caso contrario).

Programación Lógica: Prolog.

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



SWI-Prolog



Ciao Prolog

Concatenar listas



```

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

```

?- mortal(socrates).

Contesta **yes** si la “pregunta” es consecuencia lógica (**no** en caso contrario).

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

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

Programación Lógica: Hay algo más ...

- Mientras las funciones devuelven un único resultado:

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

Programación Lógica: Hay algo más ...

- Mientras las funciones devuelven un único resultado:

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

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

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

devuelve:

`L = [3,4] ?`

Programación Lógica: Hay algo más ...

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

Programación Lógica: Hay algo más ...

- Mientras las funciones devuelven un único resultado:

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

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

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

devuelve:

```
L = [3,4] ?
```

- Pero, hay algo más ...

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

devuelve 5 respuestas:

Programación Lógica: Hay algo más ...

- Mientras las funciones devuelven un único resultado:

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

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

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

devuelve:

```
L = [3,4] ?
```

- Pero, hay algo más ...

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

devuelve 5 respuestas:

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

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

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

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

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

Programación Lógica: Hay algo más ...

- Mientras las funciones devuelven un único resultado:

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

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

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

devuelve:

```
L = [3,4] ?
```

- Pero, hay algo más ...

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

devuelve 5 respuestas:

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

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

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

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

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

Esto permite usar un único predicado para codificar/decodificar mensajes



```
1 char2morse('A', '-.-').    char2morse('B', '-...').    char2morse('C', '-.-.-').    ...
2
3 ?- char2morse('B', Morse).    % devuelve Morse = '-...'
```

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

Programación Lógica: ... mucho más (CLP)

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

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

Programación Lógica: ... mucho más (CLP)

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

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

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

Programación Lógica: ... mucho más (CLP)

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

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

1	<code>sol(X,Y) :-</code>	<code>?- sol(X,Y).</code>
2	<code>3 * X + 5 * Y #= 2,</code>	<code>X = -1,</code>
3	<code>5 * X + 3 * Y #= -2.</code>	<code>Y = 1 ?</code>

Programación Lógica: ... mucho más (CLP)

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

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

1	<code>sol(X,Y) :-</code>	<code>?- sol(X,Y).</code>
2	<code>3 * X + 5 * Y #= 2,</code>	<code>X = -1,</code>
3	<code>5 * X + 3 * Y #= -2.</code>	<code>Y = 1 ?</code>

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



```

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:

```

?- mg(1000,10,150,0.10,B).      ?- mg(P,10,150,0.10,0).
   B = 203.13 ?                  P = 921.68 ?
    
```

Programación Lógica: ... mucho más (CLP)

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

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

	1 sol(X,Y) :-	?- sol(X,Y).
	2 3 * X + 5 * Y #= 2,	X = -1,
	3 5 * X + 3 * Y #= -2.	Y = 1 ?

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



```

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:

```

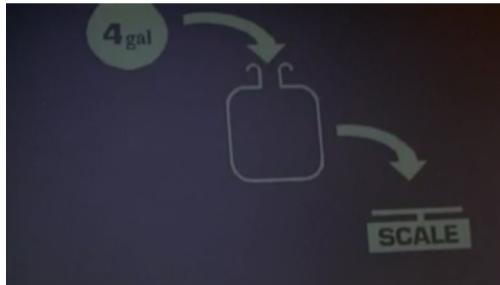
?- mg(1000,10,150,0.10,B).      B = 203.13 ?
?- mg(P,10,150,0.10,0).        P = 921.68 ?
?- mg(P,10,R,0.10,B).          P = 6.14*R + 0.38*B ?
    
```



Tema 0.2: **Algunos problemas interesantes.**

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 la bomba no explote.



[Link](#)

Ejemplo II: El problema del puente y la linterna

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



[Link](#)

Ejemplo III: Las Torres de Hanoi

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

Use Acrobat Reader

Ejemplo IV: El problema del viajante

- Formulado en 1930: Dada una lista de ciudades y las distancias entre cada par de ellas, ¿cuál es la ruta más corta posible que visita cada ciudad exactamente una vez y al finalizar regresa a la ciudad origen?

Ejemplo IV: El problema del viajante

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



Ejemplo IV: El problema del viajante

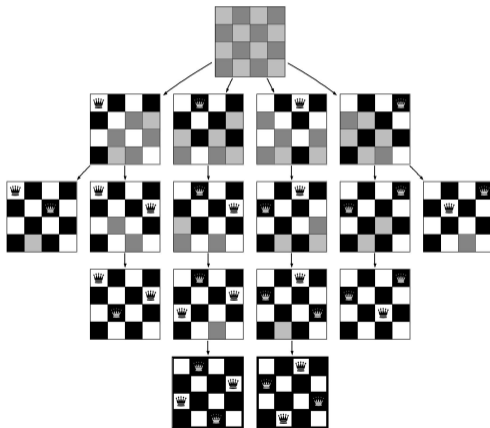
- Formulado en 1930: Dada una lista de ciudades y las distancias entre cada par de ellas, ¿cuál es la ruta más corta posible que visita cada ciudad exactamente una vez y al finalizar regresa a la ciudad origen?
- El origen de este problema lo encontramos en el juego “Icosian” (planteado por Hamilton en 1857).
- Su resolución está basada en la **búsqueda** de ciclos hamiltonianos: Dado un grafo, un ciclo hamiltoniano, es un ciclo que pasa una y solo una vez por todos los vértices del grafo.
- El problema del Ciclo Hamiltoniano (búsqueda) es NP-completo y el problema del Viajante (optimización) es NP-duro.
- **Aplicaciones:** Transporte y logística, fabricación circuitos impresos, planificación, etc.



Ejemplo V: El problema de las n-reinas

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

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



33grata@urjc.es

Ejemplo VI: Zebra puzzle (Einstein riddle)

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

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

- Modelización: $h(\text{Nationality}, \text{Pet}, \text{Cigarette}, \text{Drink}, \text{Color})$ representa cada casa.



Universidad
Rey Juan Carlos

Escuela Técnica Superior
Ingeniería Informática

Tema 1: El paradigma de la programación lógica.

Asignatura: Programación Declarativa

Joaquín Arias

Universidad Rey Juan Carlos
Escuela Técnica Superior de Ingeniería Informática

Curso 2024-2025

Índice I

Tema 1.1: Lógica de 1^{er} orden.

¿Por qué es importante?

Tipos de lógicas

Contexto histórico

Ejemplo del Entscheidungsproblem

Sintaxis, semántica y teoría interpretativa



Copyright ©2024 Joaquín Arias.

Esta obra, basada en transparencias de Ana Pradera (URJC'23) & ClipLab (UPM'24), está bajo la licencia CC BY-SA 4.0, Creative Commons Atribución-CompartirIgual 4.0 Internacional. Cómo citar esta obra: Arias, Joaquín (2024). Programación Lógica: hasta Prolog y más allá. BURJC, Madrid.

Índice II

Tema 1.2: **Método de resolución de Robinson.**

Método de Resolución de Robinson

Resolución con Unificador de Máxima Generalidad

Sustitución y algoritmo de unificación

Algoritmo de resolución con UMG

Estrategias de resolución

Estrategia de resolución SLD

Ejercicio

Tema 1.3: **Prolog.**

Algo de la historia de Prolog

Algunas aplicaciones de Prolog



Tema 1.1: Lógica de 1^{er} orden.

Lógica: ¿Por qué es importante?

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

Lógica: Tipos

- Debemos conocer al menos dos tipos de lógica

Lógica: Tipos

- Debemos conocer al menos dos tipos de lógica

Lógica proposicional

- “Sócrates es un hombre” h .
- “Sócrates es mortal” m .
- “Sócrates es un hombre y es mortal”

$$h \wedge m$$

Lógica de primer orden

- “ x es un hombre” $H(x)$.
- “ x es mortal” $M(x)$.
- “Todos los hombres son mortales”

$$\forall x. H(x) \rightarrow M(x)$$

Lógica: Tipos

- Debemos conocer al menos dos tipos de lógica

Lógica proposicional

- “Sócrates es un hombre” h .
- “Sócrates es mortal” m .
- “Sócrates es un hombre y es mortal”

$$h \wedge m$$

Lógica de primer orden

- “ x es un hombre” $H(x)$.
- “ x es mortal” $M(x)$.
- “Todos los hombres son mortales”

$$\forall x. H(x) \rightarrow M(x)$$

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

Lógica: Algo de historia

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

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

Lógica: Algo de historia

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

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

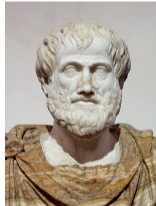
- ... tiempo después ...

Lógica: Algo de historia

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

Todos los hombres son mortales. Sócrates es 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

Lógica: Algo de historia (cont.)

Teorema de incompletitud (Gödel, 1931)

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

Lógica: Algo de historia (cont.)

Teorema de incompletitud (Gödel, 1931)

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

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

Lógica: Algo de historia (cont.)

Teorema de incompletitud (Gödel, 1931)

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

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

Segundo teorema de incompletitud (Gödel, 1931)

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

Lógica: Algo de historia (cont.)

Respuesta al *Entscheidungsproblem* (Church, 1936)

La lógica de de 1^{er} orden no posee un procedimiento de decisión (algoritmo) que permita demostrar que una fórmula cualquiera sea un teorema de dicha lógica. Esta lógica se considera, por tanto, **indecidable** [5].

Lógica: Algo de historia (cont.)

Respuesta al *Entscheidungsproblem* (Church, 1936)

La lógica de de 1^{er} orden no posee un procedimiento de decisión (algoritmo) que permita demostrar que una fórmula cualquiera sea un teorema de dicha lógica. Esta lógica se considera, por tanto, **indecidable** [5].

Teorema de la parada (Turing, 1936)

Dada una Máquina de Turing M y una palabra w , determinar si M terminará en un número finito de pasos cuando es ejecutada usando w como dato de entrada, **es indecible** [13].

Lógica: Algo de historia (cont.)

Respuesta al *Entscheidungsproblem* (Church, 1936)

La lógica de de 1^{er} orden no posee un procedimiento de decisión (algoritmo) que permita demostrar que una fórmula cualquiera sea un teorema de dicha lógica. Esta lógica se considera, por tanto, **indecidable** [5].

Teorema de la parada (Turing, 1936)

Dada una Máquina de Turing M y una palabra w , determinar si M terminará en un número finito de pasos cuando es ejecutada usando w como dato de entrada, **es indecible** [13].

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

Lógica: Algo de historia (cont.)

- **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: El juego de la vida (Conway, 1970)

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

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

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

PLAY¹

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

Lógica: Sintaxis, semántica y teoría interpretativa

- En este curso asumimos que el alumno a cursado “Lógica” en el grado.
... en caso contrario, hay cursos en abierto de la URJC.
- Trabajaremos con **cláusulas de Horn**, i.e., disyunción de literales con, como máximo, un literal positivo. P.ej. dada la consecuencia lógica:

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

se reescribe como el siguientes conjunto de cláusulas:

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

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



Tema 1.2: Método de resolución de Robinson.

Lógica: Método de resolución de Robinson, 1965

- Basado en:

Teorema de Herbrand

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

Lógica: Método de resolución de Robinson, 1965

- Basado en:

Teorema de Herbrand

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

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

$$\begin{array}{ccc} L \vee C1 & & \neg L \vee C2 \\ & \diagdown & / \\ & C1 \vee C2 & \end{array}$$

Lógica: Método de resolución de Robinson, 1965

- Basado en:

Teorema de Herbrand

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

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

$$\begin{array}{ccc} L \vee C1 & & \neg L \vee C2 \\ & \diagdown & / \\ & C1 \vee C2 & \end{array}$$

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

Lógica: Método de resolución de Robinson, 1965 (cont.)

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

Lógica: Método de resolución de Robinson, 1965 (cont.)

$C = \{I1: \neg p(a, f(b)), I2: \neg p(b, f(b)), I3: p(a, f(b)) \vee q(f(b)), I4: p(b, f(b)) \vee \neg q(f(b))\}$

resuelve I1 con I2: NO

resuelve I2 con I3: NO

resuelve I1 con I3: $q(f(b))$

resuelve I2 con I4: $\neg q(f(b))$

resuelve I1 con I4: NO

resuelve I3 con I4: $p(a, f(b)) \vee p(b, f(b))$

$R = \{I5: q(f(b)), I6: \neg q(f(b)), I7: p(a, f(b)) \vee p(b, f(b))\}$

En R no está \square , por tanto redefinimos $C = C \cup R$ y buscamos nuevos resolventes:

resuelve I1 con I5: NO

resuelve I2 con I5: NO

resuelve I1 con I6: NO

resuelve I2 con I6: NO

resuelve I1 con I7: $p(b, f(b))$

resuelve I2 con I7: $p(a, f(b))$

resuelve I3 con I5: NO

resuelve I4 con I5: $p(b, f(b))$

resuelve I3 con I6: $p(a, f(b))$

resuelve I4 con I6: NO

resuelve I3 con I7: NO

resuelve I4 con I7: NO

resuelve I5 con I6: \square

resuelve I5 con I7: NO

resuelve I6 con I7: NO

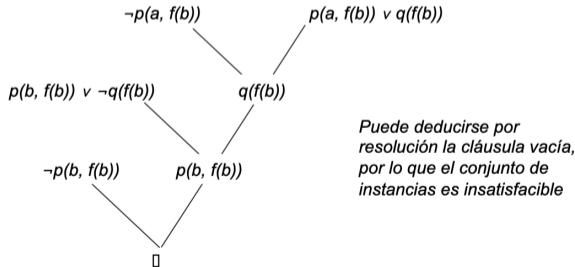
$R = \{p(b, f(b)), p(a, f(b)), \square\}$

R incluye a $\square \rightarrow C$ es insatisfacible

Lógica: Método de resolución de Robinson, 1965 (cont.)

- La aplicación de sucesivos pasos de resolución se representan 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 \square .

Conjunto de instancias básicas: $\{-p(a, f(b)), \neg p(b, f(b)), p(a, f(b)) \vee q(f(b)), p(b, f(b)) \vee \neg q(f(b))\}$





Resolución con UMG: Sustitución

Resolución con UMG: Sustitución

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

Resolución con UMG: Algoritmo de Unificación

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

Resolución con UMG: Algoritmo de Unificación: Ejemplos

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

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

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

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

Resolución con UMG: Definición

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

Resolución con UMG: Definición (cont.)

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

Teorema

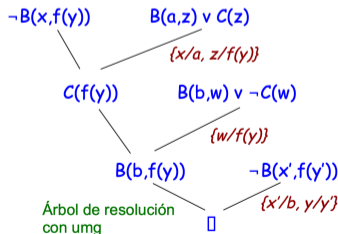
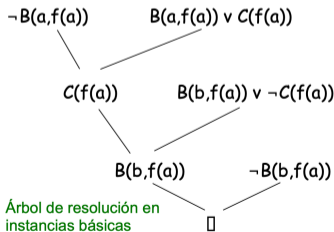
Un conjunto de cláusulas es insatisfacible sii se puede deducir \square a partir de él por resolución con umg.

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

Resolución con UMG: Ejemplo

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

Ejemplo: $C_1: \neg B(x, f(y))$ $C_2: B(a, z) \vee C(z)$ $C_3: B(b, w) \vee \neg C(w)$
 $I_1: \neg B(a, f(a))$ $I_1': \neg B(b, f(a))$ $I_2: B(a, f(a)) \vee C(f(a))$ $I_3: B(b, f(a)) \vee \neg C(f(a))$



Lógica: Estrategias de resolución (de Robinson)

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

Lógica: Estrategias de resolución (de Robinson) cont.

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

¿Qué significa ser correcta/completa?

Lógica: Estrategias de resolución (de Robinson) cont.

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

¿Qué significa ser correcta/completa?

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

Lógica: Estrategias de resolución (de Robinson) cont.

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

¿Qué significa ser correcta/completa?

- **Correcta:**
 $\square \rightarrow S$ insatisfacible
 \square se deduce sólo si el conjunto de cláusulas S es insatisfacible.
- **Completa:**
 S insatisfacible $\rightarrow \square$
 Si el conjunto de cláusulas S es insatisfacible, se deduce \square .

Lógica: Estrategia de resolución SLD.

- SLD¹: 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 $\{C_1, \dots, C_i, \dots, C_n\}$:
Existe una secuencia (*derivación*) $\langle C_i, C_{n+1}, \dots, [] \rangle$ tal que:
 - C_{n+1} es el resolvente de la cláusula objetivo C_i y una cláusula soporte.
 - C_k , con $k > n + 1$, es el resolvente de C_{k-1} con una cláusula soporte.
 - Cada paso de resolución es de la forma $L \vee C, \neg L \vee C' \rightarrow C \vee C'$.
...si y solo si el conjunto inicial es insatisfacible.

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).

Estrategia de resolución SLD: Ejercicio

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

$$\frac{\begin{array}{l} \forall ls \quad \text{Concatenar}([], ls, ls) \\ \forall x \forall xs \forall ls \forall ns \quad (\text{Concatenar}(xs, ls, ns) \rightarrow \text{Concatenar}([x|xs], ls, [x|ns])) \end{array}}{\exists la \exists lb \quad \text{Concatenar}(la, lb, [1, 2, 3, 4])}$$

2. Comprobar que es equivalente al programa Prolog:



```

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

```



Tema 1.3: Prolog.

Prolog: Algo de historia.

- **1960's:**
 - Robinson [11]: propone en 1965 una regla de inferencia a la que llama resolución, mediante la cual la demostración de un teorema puede ser llevada a cabo de manera automática.
 - Green [7]: diseña un probador de teoremas que extrae de la prueba, el valor de las variables para las cuales el teorema es válido.
- **1970's:**
 - Kowalski [10]: propone la resolución SLD (muy eficiente).
 - Alain Colmerauer [6]: crea **Prolog**, el primer lenguaje de programación lógica (implementado en Fortran).
 - Kowalski [9]: interpretación procedimental de la lógica:
Algorithm = logic + control
 - D.H.D. Warren [14]: desarrolla el primer compilador, DEC-10 Prolog (escrito casi por completo en Prolog).

Prolog: Algo de historia (cont.)

- **1980's and 1990's:**
 - Fifth Generation Project en Japón: Importante investigación sobre los paradigmas básicos y las técnicas de aplicación avanzadas.
 - Varias implementaciones comerciales de Prolog, libros y manuales, utilizando el estándar de facto, la familia Edinburgh Prolog.
 - En 1995 dio lugar a la norma ISO Prolog.
 - Programación Lógica con Restricciones (CLP) [8]: Extiende expresividad de Prolog –abrió nuevos campos de aplicación.
- **2000's:**
 - Muchas otras extensiones: orden superior, tipos/modos, concurrencia, distribución, objetos, sintaxis funcional, etc.
 - Compiladores altamente optimizadores, paralelismo automático, verificación y depuración automáticas, entornos avanzados.

Prolog: Algo de historia (cont.)

- **2010's:**
 - Variaciones: Datalog, Answer Set Programming (soporta negación mediante modelos estables), Minikanren, MiniZinc, SQL, Yedalog.
- **2020's:**
 - Verse [4]: Lenguaje lógico y funcional de Epic Games (Fortnite).
 - Logica language [12]: desarrollado en Google.
 - Janus [1]: Programación multiparadigma en Prolog y Python.
 - s(CASP) [3]: 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

²Ver detalles y los proyectos en <https://utd-hackreason-2021.devpost.com> y <https://utd-hackreason-2022.devpost.com>.

Prolog: Aplicaciones.

Áreas 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 procedimientos.
- El primer intérprete de Erlang fue desarrollado en Prolog por Joe Armstrong.
- El sistema de instalación y configuración de redes de Microsoft Windows NT.
- El Network Resource Manager (NRM) de Ericsson.
- “Sistema de reserva de vuelos que gestiona casi un tercio de los billetes de avión del mundo” (SICStus).
- La especificación de la máquina abstracta Java.

Referencias I

- [1] C. Andersen and T. Swift (2023). **The Janus System: A Bridge to New Prolog Applications**. In: *Prolog: The Next 50 Years*. Cham: Springer Nature Switzerland, pp. 93–104. DOI: [10.1007/978-3-031-35254-6_8](https://doi.org/10.1007/978-3-031-35254-6_8).
- [2] J. Arias (2022). **Lógica: desde Aristóteles hasta Prolog**. Madrid: Servicio de Publicaciones de la Universidad Rey Juan Carlos. ISBN: 978-84-09-38265-1.
- [3] J. Arias, M. Carro, E. Salazar, K. Marple, and G. Gupta (2018). **Constraint Answer Set Programming without Grounding**. In: *Theory and Practice of Logic Programming* 18.3-4, pp. 337–354. DOI: [10.1017/S1471068418000285](https://doi.org/10.1017/S1471068418000285).
- [4] L. Augustsson, J. Breitner, K. Claessen, R. Jhala, S. Peyton Jones, O. Shivers, G. L. Steele Jr, and T. Sweeney (2023). **The verse calculus: a core calculus for deterministic functional logic programming**. In: *Proceedings of the ACM on Programming Languages* 7.ICFP, pp. 417–447. DOI: [10.1145/3607845](https://doi.org/10.1145/3607845).

Referencias II

- [5] A. Church (1936). **A note on the Entscheidungsproblem**. In: *The journal of symbolic logic* 1.1, pp. 40–41. DOI: [10.2307/2269326](https://doi.org/10.2307/2269326).
- [6] A. Colmerauer and P. Roussel (1996). **The birth of Prolog**. In: *History of programming languages—II*, pp. 331–367. DOI: [10.1145/234286.1057820](https://doi.org/10.1145/234286.1057820).
- [7] C. Green (1969). **Application of theorem proving to problem solving**. In: *Proceedings of the 1st International Joint Conference on Artificial Intelligence*. IJCAI'69. Washington, DC, 219–239. URL: <https://shorturl.at/8rPHk>.
- [8] J. Jaffar and M. Maher (1994). **Constraint Logic Programming: A Survey**. In: *Journal of Logic Programming* 19/20, pp. 503–581. DOI: [10.1016/0743-1066\(94\)90033-7](https://doi.org/10.1016/0743-1066(94)90033-7).
- [9] R. Kowalski (1979). **Algorithm = logic + control**. In: *Communications of the ACM* 22.7, pp. 424–436. DOI: [10.1145/359131.359136](https://doi.org/10.1145/359131.359136).
- [10] R. Kowalski and D. Kuehner (1971). **Linear resolution with selection function**. In: *Artificial Intelligence* 2.3-4, pp. 227–260. DOI: [10.1016/0004-3702\(71\)90012-9](https://doi.org/10.1016/0004-3702(71)90012-9). URL: <https://shorturl.at/kW7Ts>.

Referencias III

- [11] J. A. Robinson (1965). **A machine-oriented logic based on the resolution principle**. In: *Journal of the ACM (JACM)* 12.1, pp. 23–41. DOI: [10.1145/321250.321253](https://doi.org/10.1145/321250.321253).
- [12] E. Skvortsov, Y. Xia, and B. Ludäscher (2024). **Logica: Declarative Data Science for Mere Mortals**. In: *EDBT*, pp. 842–845. DOI: [10.48786/edbt.2024.84](https://doi.org/10.48786/edbt.2024.84).
- [13] A Turing (1936). **On computable numbers, with an application to the Entscheidungsproblem**. In: *Proceedings of the London Mathematical Society Series/2 (42)*, pp. 230–42. DOI: [10.1112/plms/s2-42.1.230](https://doi.org/10.1112/plms/s2-42.1.230). URL: <https://shorturl.at/4Zu1B>.
- [14] D. H. Warren (1977). **Implementing Prologcompiling predicate logic programs**. In: *Research Reports 39 and 40, Dpt. of Artificial Intelligence, Univ. of Edinburgh*. URL: <https://shorturl.at/2tSVh>.



Universidad
Rey Juan Carlos

Escuela Técnica Superior
Ingeniería Informática

Tema 2: Programación lógica pura (Turing completo).

Asignatura: Programación Declarativa

Joaquín Arias

Universidad Rey Juan Carlos
Escuela Técnica Superior de Ingeniería Informática

Curso 2024-2025

Índice I

Tema 2.1: Sintaxis y Semántica de Prolog.

Sintaxis

Semántica

Árbol de derivación

Estrategias de búsqueda



Copyright ©2024 Joaquín Arias.

Esta obra, basada en transparencias de Ana Pradera (URJC'23) & ClipLab (UPM'24), está bajo la licencia CC BY-SA 4.0, Creative Commons Atribución-CompartirIgual 4.0 Internacional. Cómo citar esta obra: Arias, Joaquín (2024). Programación Lógica: hasta Prolog y más allá. BURJC, Madrid.

Índice II

Tema 2.2: **Programas (y tipos) recursivos.**¹

Aritmética con enteros

Listas

Combinar aritmética y listas en Prolog

Árboles binarios

Expresiones Simbólicas

Reconocer polinomios.

Calcular la derivada.

Torres de Hanoi.

Satisfabilidad de fórmulas Booleanas

¹Capítulo 3 “Recursive Programming” de The Art of Prolog (Sterling and Shapiro, 1994).



Tema 2.1: Sintaxis y Semántica de Prolog.

Sintaxis: Alfabeto

- El alfabeto de Prolog se define con los siguientes tipos de símbolos:
 - Símbolos de constante (átomos): *Cadenas de letras, dígitos y guión bajo, que (i) empiezan por letra minúscula o (ii) están entre comillas simples.* `a`, `b`, `c`, ..., `'X'`, ...
 - Otros: Números (enteros y reales), lista vacía `[]`, `true`, `false`, etc.
 - Símbolos de variable: *Cadenas de letras, dígitos y guión bajo, que empiezan por (i) letra mayúscula o (ii) guión bajo.*² `X`, `Xs`, `_X`, `_`, ...
 - Símbolos de functor: `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.

²Las variables que empiezan con guión bajo se llaman variables anónimas (ver pág. 15).

Sintaxis: Expresiones

- Una expresión es cualquier concatenación finita de símbolos del alfabeto. Las expresiones relevantes en Prolog son:
 - Términos:
 - Un símbolo de constante es un término.
 - Un símbolo de variable es un término.
 - Si f es un símbolo de funtor n -aria y t_1, \dots, t_n son términos entonces $f(t_1, \dots, t_n)$ es un término.
 - Predicados:
 - Si p es un símbolo de predicado n -aria y t_1, \dots, t_n son términos entonces $p(t_1, \dots, t_n)$ es un predicado.
 - Cláusulas (reglas, hechos y consultas):
 - Las reglas son de la forma:
 $H :- B_1, B_2, \dots, B_n.$
donde H y B_i son predicados, $:-$ representa la implicación \leftarrow y $,$ conjunción \wedge . La cabeza de la cláusula es H y el resto es el cuerpo.
 - Los hechos son cláusulas sin cuerpo:
 $H.$ equivalente a $H :- \text{true}.$
 - Las consultas no tienen cabeza:
 $?- B_1, B_2, \dots, B_n.$

Sintaxis: Programas

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

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

```
?- mortal(X).  
  
X = socrates ?  
  
yes  
?-
```

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

Semántica: Formal, informal (y operacional)

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

- El modelo mínimo de Herbrand de P .

$$Mp = \{ A \in Bp \mid P \models A \}$$

- Least fixed point* de Tp (operador de consecuencia inmediata).

$$Tp \uparrow \omega = \bigcup_{i=0}^{\infty} Tp^i(\emptyset)$$

- Donde $Tp(I) = \{ A_0 \in Bp \mid A_0 \leftarrow A_1, \dots, A_n \in P \wedge A_i \in I \}$.

- Adicionalmente, su significado lo podemos ver como:

- Un conjunto de fórmulas a las que aplicamos deducción.

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

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

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

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

Semántica: Resolución SLD + búsqueda

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

$$\frac{\begin{array}{l} \forall ls \quad \text{Concatenar}([], ls, ls) \\ \forall x \forall xs \forall ls \forall ns \quad (\text{Concatenar}(xs, ls, ns) \rightarrow \text{Concatenar}([x|xs], ls, [x|ns])) \end{array}}{\exists la \exists lb \quad \text{Concatenar}(la, lb, [1, 2, 3, 4])}$$

2. Comprobar que es equivalente al resultado del programa:



```

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

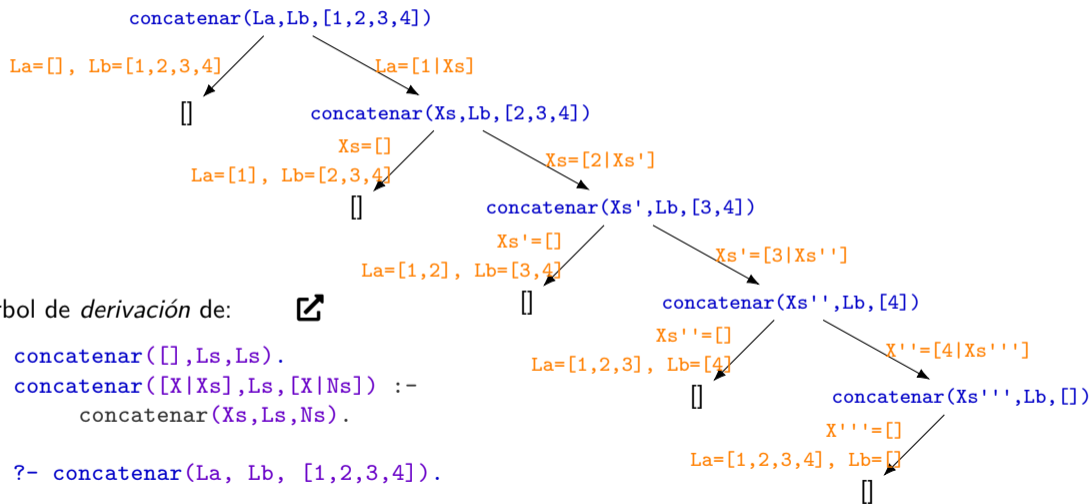
```

Semántica: Árbol de derivación

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

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

Semántica: Árbol de derivación (Ejemplo)



• Árbol de derivación de:



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

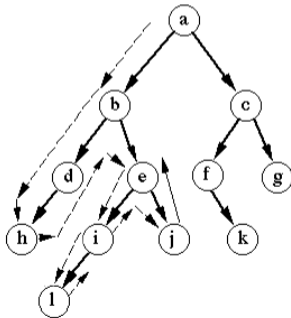
Semántica: Estrategias de búsqueda

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

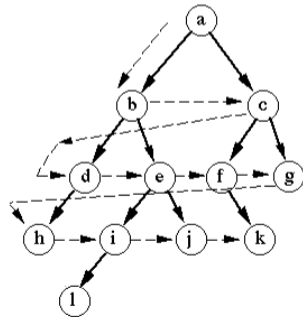
Semántica: Estrategias de búsqueda

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

En profundidad



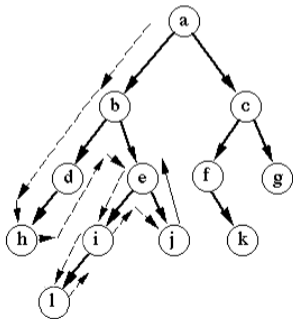
En Anchura



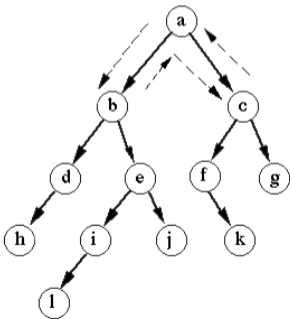
Semántica: Estrategias de búsqueda

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

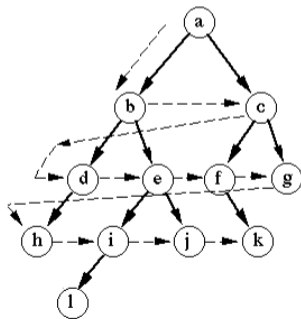
En profundidad



Profundización iterativa



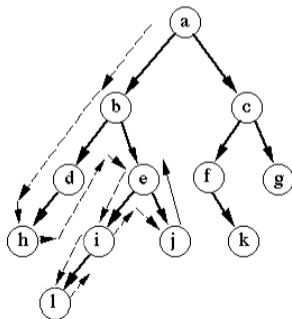
En Anchura



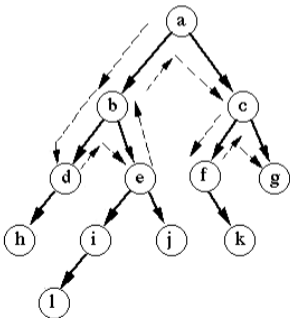
Semántica: Estrategias de búsqueda

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

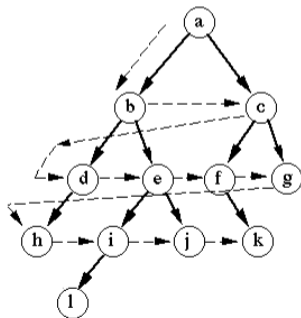
En profundidad



Profundización iterativa



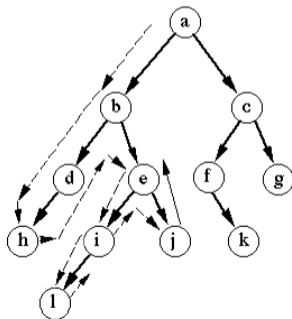
En Anchura



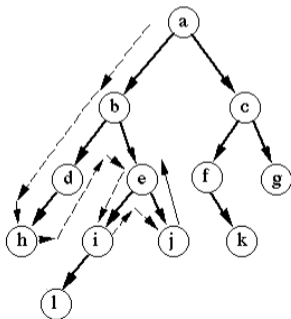
Semántica: Estrategias de búsqueda

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

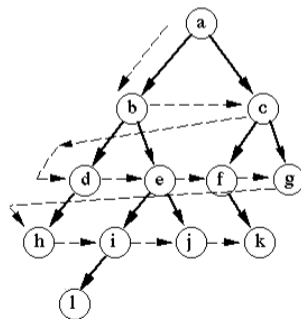
En profundidad



Profundización iterativa



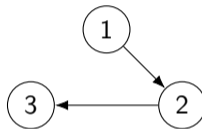
En Anchura



Semántica: Estrategias de búsqueda (Ejercicios)

1. Dado el siguiente programa:

```
1 edge(1, 2).  
2 edge(2, 3).  
3  
4 reach(X, Y) :- edge(X, Z), reach(Z, Y).  
5 reach(X, Y) :- edge(X, Y).  
6  
7 ?- reach(1, Y).
```

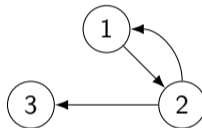


- Construir el árbol de derivación.
- Discutir las ventajas e inconvenientes de la búsqueda en profundidad vs. anchura.

Semántica: Estrategias de búsqueda (Ejercicios cont.)

2. Dado el siguiente programa:


```
1 edge(1, 2).  
2 edge(2, 1).  
3 edge(2, 3).  
4  
5 reach(X, Y) :- edge(X, Z), reach(Z, Y).  
6 reach(X, Y) :- edge(X, Y).  
7  
8 ?- reach(1, Y).
```



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

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

- Ciao Prolog [4] dispone de diferentes estrategias de búsqueda:

bf: Implementa búsqueda en anchura [2]. Ejemplo: 

```
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 [3]. 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].

NOTA: Uso de variables anónimas

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.

2. **En consultas**, cuando no estamos interesados en su valor:

- Si usamos solo el guión bajo, cada aparición de `_` es una variable diferente. P.ej., la consulta:

```
?- concatenar(_, [2], [1,2]), concatenar(_, Z, [1,2]).
```

genera tres soluciones `Z=[1,2]`, `Z=[2]`, y `Z=[]`.

- ...pero usando un mismo símbolo, p.ej., `_A`, la consulta:

```
?- concatenar(_A, [2], [1,2]), concatenar(_A, Z, [1,2]).
```

genera como única respuesta `Z=[2]`, porque `_A` se instancia a `[1]` pero al ser anónima no aparece en los resultados de la consulta.



Tema 2.2: Programas (y tipos) recursivos.³

³Capítulo 3 “Recursive Programming” de The Art of Prolog (Sterling and Shapiro, 1994).


Aritmética de Peano

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

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

⁴Los matemáticos (incluido Peano) suelen usar el 1 como primer número natural.

Aritmética de Peano: Implementación (y semántica) lógica

- Primero, veremos las operaciones como relaciones (no como funciones).
- Segundo, analizaremos la corrección y completitud de los programas.
- **Ejemplo 1:** `nat/1` define el tipo recursivo de los números naturales.
 - Los construimos con la constante `0` y el funtor `s/1`. 

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

Aritmética de Peano: Implementación (y semántica) lógica (cont.)

- **Demostración 1:**

(1) **Completitud:** Sea n un número natural. Demostramos que el objetivo $?- \text{nat}(n)$ es deducible a partir del programa mediante un árbol de derivación. O bien n es 0 o de las forma $s^i(0)$. El árbol de derivación para $\text{nat}(0)$ es trivial. El árbol de derivación para $\text{nat}(s(\dots s(0)\dots))$ contiene n derivaciones, usando la regla recursiva, hasta alcanzar el hecho $\text{nat}(0)$.⁵

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

⁵Ejercicio: dibujar el árbol de derivación.

Aritmética de Peano: Implementación (y semántica) lógica (cont.)

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

Aritmética de Peano: Ejercicios

- 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.
```
 2. 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.
```
 3. 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.
```
 4. Calculando el factorial.

```
1 % factorial(N,F): F equals N factorial.
```
 5. El mínimo de dos números.

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


Aritmética de Peano: Ejercicios (cont.)

- Implementa las siguientes operaciones:
 6. Calcular el residuo/módulo (resto de una división)
 - 1 `% mod(X,Y,Z): Z is the remainder of the integer`
 - 2 `% division of X by Y`
 - a) Implementar con: (i) `times/3` y (ii) recursivamente,
 - b) Comparar sus complejidades (nodos en el árbol de derivación).
 7. La función de Ackermann:⁶
 - 1 `% ackermann(X,Y,A): A is the value of Ackerman's`
 - 2 `% function for X and Y.`
 8. 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`

⁶Su definición está disponible en <https://shorturl.at/Sh6Ry>

Manipulación de listas en Prolog

- Un tipo recursivo mas complejo es una estructura binaria, la lista.
 - Se construye con la constante `[]`, para la lista vacía.
 - y con el functor 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]}`.

Manipulación de listas en Prolog: Ejercicios

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

```
1  % member(Element, List): Element is an element
2  %                               of the list List
```
2. Escribe la consulta correspondiente para:
 - Comprobar si **b** está en la lista **[a,b,c]**.
 - Encontrar los elementos de la lista **[a,b,c]**.
 - Encontrar una lista que contiene **a**.
3. Implementar operaciones para obtener los prefijos / sufijos de una lista.

```
1  % prefix(Prefix, List): Prefix is a prefix of List
2  % suffix(Suffix, List): Suffix is a suffix of List
```
4. Define concatenación de listas y redefine **prefix/2** y **suffix/2**.

```
1  % append(Xs,Ys,XsYs): XsYs is the result of concatenating
2  %                               the lists Xs and Ys
```

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

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

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

Manipulación de listas en Prolog: Recursión con acumulación

- Implementemos el predicado `reverse(List,Reverse)`, donde `Reverse` es una lista que contiene los elementos 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).

2. Como alternativa, definimos un predicado auxiliar `reverse_(Xs,Acc,Ys)`, donde `Ys` es el resultado de concatenar `Acc` a los elementos de `Xs` invertidos:

```
1 % a) Reverse-accumulate
2 reverse_(Xs, Ys) :-
3     reverse_(Xs, [], Ys).
4 reverse_([], Acc, Acc).
5 reverse_(Xs, Acc, Ys) :-
6     reverse_(Xs, [X|Acc], Ys).
```

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

Combinar enteros y listas en Prolog

- 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`.
 - a) Usando el programa `plus/3`.
 - b) Sin utilizar predicados auxiliares.

Árboles binarios en Prolog

- Otro importante tipo recursivo es el de los árboles binarios, que se representan por el funtor terciario `tree(Element, Left, Right)`, donde:
 - `Element` es el elemento en el nodo.
 - `Left` and `Right` son el sub-árbol de la izquierda y derecha respectivamente.
- El árbol vacío se representa por la constante `void`.
- Resuelve los siguientes ejercicios:
 1. Dibuja el árbol `tree(a,tree(b,void,void),tree(c,void,void))`.
 2. Implementa `binary_tree(Tree)` (Nota: es doblemente recursivo).
 3. Implementa la operación `tree_member(Element, Tree)`, que determina si un elemento está 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 en Prolog

- La manipulación de números, listas y árboles binarios no es nada que no se pueda hacer con otros lenguajes (p.ej., en programación funcional).
- En esta sección analizaremos cuatro programas recursivos donde explotaremos la capacidad para el razonamiento simbólico de Prolog.
 1. Reconocer polinomios en un termino **X**:
p.ej., $x^2 - 3x + 2$ es un polinomio en x .
 2. Calcular la derivada de una expresión con respecto de **X**:
p.ej., `derivate(X,X,s(0))` esta en la semántica del programa.
 3. Resolver las torres de Hanoi.
...en lugar de una relación definiremos un esquema.
 4. Determinar la satisfacibilidad de una fórmula Booleana.
...es decir `satisfiable(X)` se cumple cuando **X** es Verdadera, y `invalid(X)` cuando **X** es Falsa.

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

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

```

1 % polynomial(Expression,X): Expression 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 $x^2 - 3x + 2$ en `x`: `?- polynomial(x^s(s(0)) - s(s(s(0)))*x + s(s(0)), x)` tiene éxito.

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

- Resuelve los siguientes ejercicios:
 1. Da una lectura declarativa de las cláusulas de `polynomial/2`.
 2. Crea `my_polynomial/2` para en lugar de `+`, `-`, `*`, `/`, y `^` usar los funtores prefijos `p`, `s`, `m`, `d`, y `t`, el polinomio $x^2 - 3x + 2$ se representa como `p(s(t(x,s(s(0))),m(s(s(s(0))),x)),s(s(0)))`.
 3. Extiende `polynomial/2` y `my_polynomial/2`⁷ para que (también) acepte la negación, p.ej., $-x^2 - 3x + 2$.
 4. Modifica `polynomial/2` para que acepte términos con fracciones de polinomios, p.ej., $x^3 + \frac{x^2 - 3x}{x - 3} + 2$.

⁷La resta, `s/2`, se desambigua de sucesor, `s/1`, por la aridad. Para la negación usar `n/1`.

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

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

```
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).
```

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

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

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

Enunciado:

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

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

Solución:

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


```

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]`.

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

- Aún cuando queramos interpretar `hanoi/5` como una relación, es evidente su semántica “operacional”⁸ –dado `N` discos y tres postes `A`, `B`, y `C`, esperamos obtener en `Moves` la secuencia de movimientos...
- Igualmente las clausulas se entienden mejor leyéndolas de izquierda a derecha (de arriba a abajo), i.e., en un movimiento podemos mover un disco de `A` a `B` y para mover `s(N)` de `A` a `B` usando `C` como auxiliar:
 - Movemos `N` discos de `A` a `C` (ver línea 3).
 - Luego el disco en `A` se coloca en `B` (ver `[A - B]` en `append/3`).
 - Luego movemos los `N` discos de `C` a `B` (ver línea 4).
 - Finalmente concatenamos todos los movimientos en línea 5.
- (Ejercicio) Cambia la clausula base considerando que quedan `0` discos y comprueba si el programa funciona correctamente (explica porqué).

⁸Profundizaremos en la semántica operacional de Prolog en el Tema 3. Adicionalmente se muestra una implementación alternativa (más declarativa) basada en Answer Set Programming (ASP).

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

- Una fórmula Booleana es un término definido como:
 - Las constantes *true* y *false* son fórmulas Booleanas.
 - Si X y Y son fórmulas Booleanas también lo son:

$$X \wedge Y, X \vee Y, \text{ y } \neg X$$

donde \wedge y \vee son operadores binarios infijos para la conjunción y la disyunción, y \neg es el operador prefijo unario para la negación.

- Primero definiremos los operadores \wedge , \vee y \sim respectivamente:

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

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

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 functor) con precedencia `Precedence`.⁹

⁹Detalles sobre el uso de `op/3` disponibles en la documentación de `Ciao` y `Swi`.

Expresiones Simbólicas IV: satisfiable/1 & invalid/1 (Cont.)

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

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

- Puede aplicarse a fórmulas Booleanas con variables...
i.e., es mas potente de lo que parece ;-)
- (Ejercicio) Implementa `boolean/1`, constructor de fórmulas Booleanas.

Referencias I

- [1] I. Blázquez Ballesteros (2017). **Implementación de un algoritmo de búsqueda aleatoria en programación lógica**. ETSI Informáticos, TFG. Universidad Politécnica de Madrid. URL: <https://oa.upm.es/47203>.
- [2] D. Cabeza, M. Carro, and M. Hermenegildo (n.d.). **Breadth-first execution**. The Ciao System. URL: <https://shorturl.at/AWXWe>.
- [3] R. Haemmerlé, M. Carro, C. Vaucheret, and M. Hermenegildo (n.d.). **Iterative-deepening execution**. The Ciao System. URL: <https://shorturl.at/V57pU>.
- [4] M. V. Hermenegildo, F. Bueno, M. Carro, P. López-García, E. Mera, J. F. Morales, and G. Puebla (2012). **An overview of Ciao and its design philosophy**. In: *Theory and Practice of Logic Programming* 12.1-2, pp. 219–252. DOI: [10.1017/S1471068411000457](https://doi.org/10.1017/S1471068411000457). URL: <http://ciao-lang.org>.
- [5] L. Sterling and E. Y. Shapiro (1994). **The art of Prolog: advanced programming techniques**. URL: <https://shorturl.at/rN5Rt>.



Universidad
Rey Juan Carlos

Escuela Técnica Superior
Ingeniería Informática

Tema 3: Programación lógica avanzada.

Asignatura: Programación Declarativa

Joaquín Arias

Universidad Rey Juan Carlos
Escuela Técnica Superior de Ingeniería Informática

Curso 2024-2025

Índice I

Tema 3.1: Semántica operacional de Prolog.

Modelo computacional

Operador corte

Negación

Answer Set Programming

Memorización

Acumulador

Tabulación

Aritmética

Limitaciones y CLP

Inspeccionar estructuras, entrada/salida...

Compilar scripts ejecutables



Copyright ©2024 Joaquín Arias.

Esta obra, basada en transparencias de Ana Pradera (URJC'23) & ClipLab (UPM'24), está bajo la licencia CC BY-SA 4.0, Creative Commons Atribución-CompartirIgual 4.0 Internacional. Cómo citar esta obra: Arias, Joaquín (2024). Programación Lógica: hasta Prolog y más allá. BURJC, Madrid.

Índice II

Tema 3.2: Programación Procedimental.

Algoritmos de ordenamiento

Meta-interprete

Trucos eficientes

Tema 3.3: Programación Lógica de 2º Orden.

Recolección de soluciones

Predicados de orden superior

Agregación de respuestas dinámica

Motivación

Agregación de $?- \text{dist}(a, Y, D)$

Codificación de Agregados en Ciao

Ejemplos de uso de agregados

Índice III

Tema 3.4: Programación Lógica con Restricciones (CLP)

Introducción

- Programación con Restricciones
- Programación Lógica con Restricciones
- Ejemplo resolución de restricciones

Sistema de restricciones

- Definición Formal
- Dominios de restricciones

Resolutor de restricciones: Ejemplo: CLP(R)

- Ejemplo: CLP(R)
- Fibonacci usando CLP(R)

Reduce el espacio de búsqueda

- Árbol de derivación



Tema 3.1: Semántica operacional de Prolog.

Modelo computacional: SLD + búsqueda en profundidad

- Para que Prolog sea considerado un lenguaje de programación fijamos:
 - El orden en el que se seleccionan las cláusulas dentro del programa.
 - La estrategia de resolución de los objetivos en el resolvente.
- Las cláusulas 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).
```

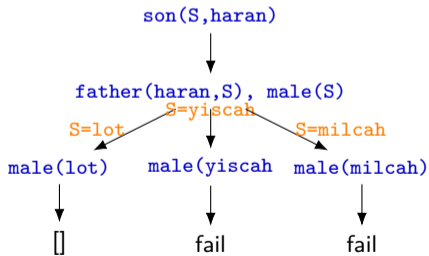
Modelo computacional: SLD + búsqueda en profundidad (Cont.)

- Supongamos la siguiente base de datos:

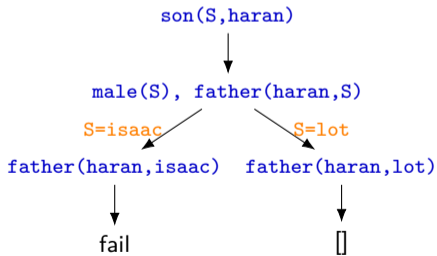
```

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:



Árbol de búsqueda versión b:



Modelo computacional: SLD + búsqueda en profundidad (Cont.)

- **Conclusión 1:**
 - El orden de las cláusulas en el programa.
 - Así como el orden de los objetivos en el cuerpo de las cláusulas.puede influir en:
 - El orden en el que se encuentran (o no) las respuestas.
 - La terminación (o no) de las consultas.
- De hecho, el modelo computacional de Prolog no es completo: no asegura encontrar soluciones:
 - Hay una rama infinita a la izq. de una rama con respuestas.
 - Como ya hemos explicado esto se puede resolver usando búsqueda en anchura (paquete `sr/bfall` [4]).
- **Conclusión 2:** Este modelo computacional nos permite definir una semántica operacional (procedimental).

Modelo computacional: Semántica operacional

- Como ya indicamos, además de las semánticas lógicas, podemos ver la ejecución en Prolog como un conjunto de procedimientos:
 - La consulta Q es un conjunto de objetivos Q_1, Q_2, \dots, Q_m .
 - Para probar Q_i, \dots, Q_m :
 - Encuentra una cláusula $H :- B_1, \dots, B_n$ tal que Q_i y H unifican (con un umg).
 - Bajo la substitución (del umg) de las variables en la cláusula, probar (recursivamente) $B_1, \dots, B_n, Q_{i+1}, \dots, Q_m$.
 - Si no queda nada por probar, la ejecución tiene éxito.
 - Si no hay más cláusulas que coincidan, la ejecución falla.

Operador corte

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

Ejemplo de corte verde:

```
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
```

Operador corte (Cont.)

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

```
1 % polynomial(Expression,X): Expression 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 por fallo: `!/0` + `fail/0` + `call/1`

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

$$\forall x (\text{flies}(x) \leftarrow \text{bird}(x) \wedge \neg \text{abnormal_bird}(x))$$

$$\forall x (\text{abnormal_bird}(x) \leftarrow \text{penguin}(x))$$

- (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 [5]).
 - El objetivo `\+ G` tiene éxito si `G` falla y viceversa.
 - Pero no soporta llamadas con variables ni recursión.
- La implementación de `\+` usa: el corte `!/0`, el predicado `fail/0`, que provoca el fallo, y `call/1`, que invoca el predicado que recibe:

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

```
2 \+ Goal.
```

Negación por fallo: Ejemplo y limitaciones

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

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

devuelve como única respuesta `X=sam` porque `tweety` es un pingüino.

- Limitación I: cambiando la posición del objetivo negado en línea 1:

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

la consulta falla, porque `call(abnormal_bird(X))` tiene éxito.

- Limitación II: Programas con negación no estratificada entra en bucle:

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

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

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

- **Answer Set Programming (ASP)**: Basada en la semántica de modelos estables [Gelfond and Lifschitz, 1988].

- El programa anterior:

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

Tiene dos modelos stables:

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

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

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

Memorización: Evita bucles y recomputaciones

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

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

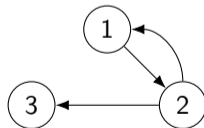
- En esta sección veremos algunas técnicas para evitar entrar en una derivación infinita y/o reducir el espacio de búsqueda:
 1. Memorización: recordar información de los estados visitados:
 - a) Añadiendo un argumento.
 - b) Usando técnicas avanzadas de búsqueda como tabulación.
 2. Uso de agregados para agrupar el conjunto de soluciones parciales durante la búsqueda [Arias and Carro, 2019] (ver pág. 38).

Memorización: 1a) Acumulador

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

- Dicha lista es el tercer argumento del predicado `reach_aux/3`.
- y con `\+ is_member(Z,Visitados)` nos aseguramos que `Z` no ha sido visitado.
- (Ejercicio) Dado el siguiente grafo y la consulta `?- reach(1,X)`, indica las respuestas esperadas:



Memorización: 1a) Acumulador (Ejercicio)

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

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

2. Dado el siguiente programa en Prolog

```
1 reach(X,Y) :-                               9 edge(1,2).
2   reach_aux(X,Y,[X]).                       10 edge(2,1).
3 reach_aux(X,Y,Visitados) :-                11 edge(2,3).
4   edge(X,Z),                                12
5   \+ is_member(Z,Visitados),                13 is_member(X,[X|_]).
6   reach_aux(Z,Y,[Z|Visitados]).            14 is_member(X,[_|T]) :-
7 reach_aux(X,Y,_):-                          15   is_member(X,T).
8   edge(X,Y).
```

modificarlo para que no permita ciclos, es decir, la consulta

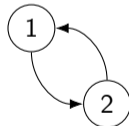
?- reach(X,X) falla y ?- reach(1,Y) solo devuelve Y=2 y Y=3.

Memorización: 1b) Tabulación

- La tabulación (**tabling** en inglés, ver Tamaki and Sato (1986) and Warren (1992)) es una estrategia de búsqueda en profundidad para Prolog que emula la búsqueda en anchura:
 - Memoriza las respuestas de la primera ocurrencia de un objetivo.
 - Suspende la evaluación de objetivos repetidos.
... los cuales re-arrancan para consumir dichas respuestas.
- Evaluaremos el siguiente programa usando tabulación:

```

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



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

Memorización: 1b) Tabulación: Evaluación de $reach(1, Y)$

$reach(1, Y)$ ①

Objetivo	Respuestas

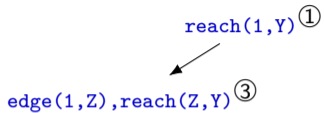
Memorización: 1b) Tabulación: Evaluación de $reach(1, Y)$

$reach(1, Y)$ ①

Objetivo	Respuestas
$reach(1, Y)$ ②	

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

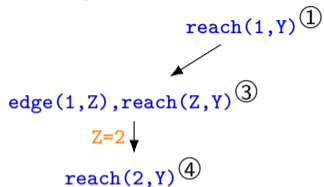
Memorización: 1b) Tabulación: Evaluación de $?- \text{reach}(1, Y)$



Objetivo	Respuestas
$\text{reach}(1, Y)$ ②	

② Guardamos $\text{reach}(1, Y)$ en la tabla.

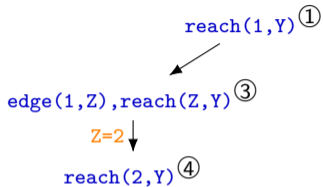
Memorización: 1b) Tabulación: Evaluación de $?- \text{reach}(1, Y)$



Objetivo	Respuestas
$\text{reach}(1, Y)$ ②	

- ② Guardamos $\text{reach}(1, Y)$ en la tabla.

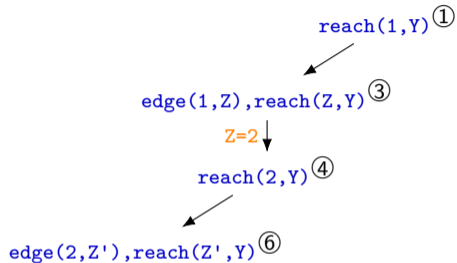
Memorización: 1b) Tabulación: Evaluación de $?- \text{reach}(1, Y)$



Objetivo	Respuestas
$\text{reach}(1, Y)$ (2)	
$\text{reach}(2, Y)$ (5)	

- (2) Guardamos $\text{reach}(1, Y)$ en la tabla.

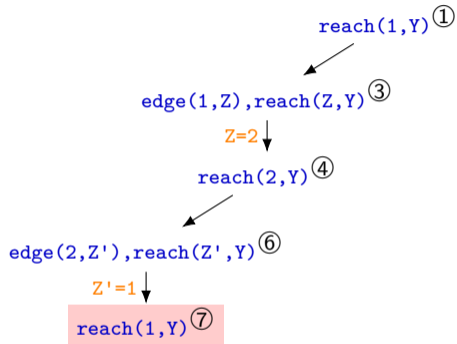
Memorización: 1b) Tabulación: Evaluación de $?- \text{reach}(1, Y)$



Objetivo	Respuestas
$\text{reach}(1, Y)$ ②	
$\text{reach}(2, Y)$ ⑤	

② Guardamos $\text{reach}(1, Y)$ en la tabla.

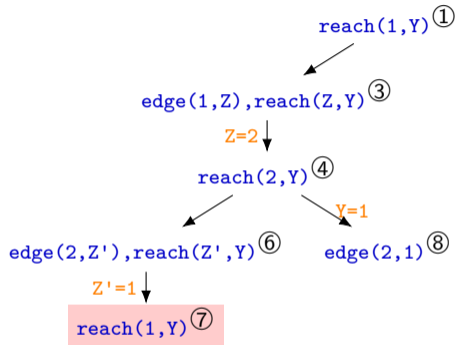
Memorización: 1b) Tabulación: Evaluación de $?- \text{reach}(1, Y)$



Objetivo	Respuestas
$\text{reach}(1, Y)$ ②	
$\text{reach}(2, Y)$ ⑤	

- ② Guardamos $\text{reach}(1, Y)$ en la tabla.
- ⑦ Suspendemos la evaluación de $\text{reach}(1, Y)$.

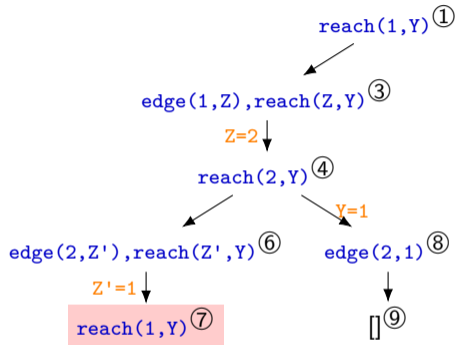
Memorización: 1b) Tabulación: Evaluación de $reach(1, Y)$



Objetivo	Respuestas
$reach(1, Y)$ ②	
$reach(2, Y)$ ⑤	

- ② Guardamos $reach(1, Y)$ en la tabla.
- ⑦ Suspendemos la evaluación de $reach(1, Y)$.

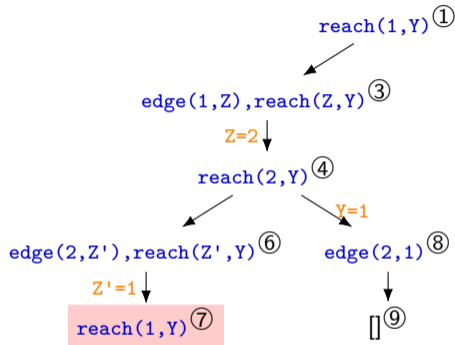
Memorización: 1b) Tabulación: Evaluación de $reach(1, Y)$



Objetivo	Respuestas
$reach(1, Y)$ ②	
$reach(2, Y)$ ⑤	$Y=1$ ⑨

- ② Guardamos $reach(1, Y)$ en la tabla.
- ⑦ Suspendemos la evaluación de $reach(1, Y)$.
- ⑨ Guardamos $Y=1$ para $reach(2, Y)$ y $reach(1, Y)$.

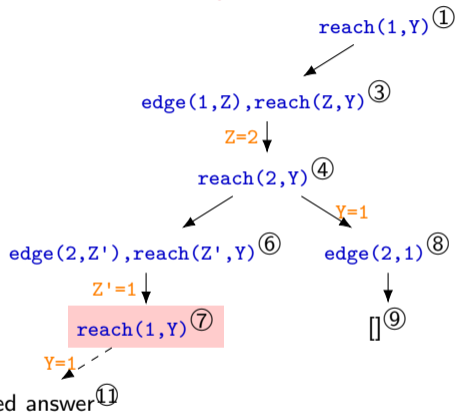
Memorización: 1b) Tabulación: Evaluación de $?- \text{reach}(1, Y)$



Objetivo	Respuestas
$\text{reach}(1, Y)$ ②	$Y=1$ ⑩
$\text{reach}(2, Y)$ ⑤	$Y=1$ ⑨

- ② Guardamos $\text{reach}(1, Y)$ en la tabla.
- ⑦ Suspendemos la evaluación de $\text{reach}(1, Y)$.
- ⑨ Guardamos $Y=1$ para $\text{reach}(2, Y)$ y $\text{reach}(1, Y)$.

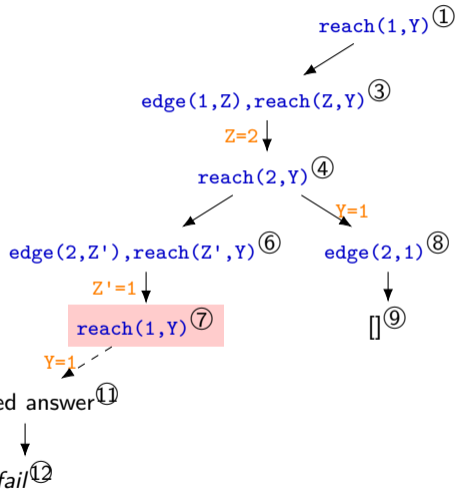
Memorización: 1b) Tabulación: Evaluación de $reach(1, Y)$



Objetivo	Respuestas
$reach(1, Y)$ (2)	$Y=1$ (10)
$reach(2, Y)$ (5)	$Y=1$ (9)

- (2) Guardamos $reach(1, Y)$ en la tabla.
- (7) Suspendemos la evaluación de $reach(1, Y)$.
- (9) Guardamos $Y=1$ para $reach(2, Y)$ y $reach(1, Y)$.
- (11) Consumimos $Y=1$, pero no genera nuevas respuestas.

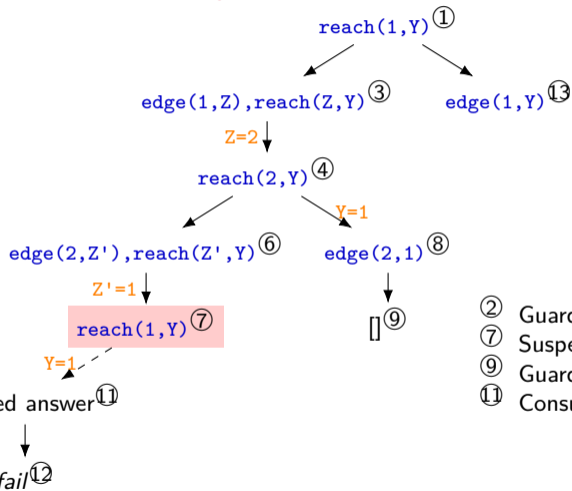
Memorización: 1b) Tabulación: Evaluación de $?- \text{reach}(1, Y)$



Objetivo	Respuestas
$\text{reach}(1, Y)$ (2)	$Y=1$ (10)
$\text{reach}(2, Y)$ (5)	$Y=1$ (9)

- (2) Guardamos $\text{reach}(1, Y)$ en la tabla.
- (7) Suspendemos la evaluación de $\text{reach}(1, Y)$.
- (9) Guardamos $Y=1$ para $\text{reach}(2, Y)$ y $\text{reach}(1, Y)$.
- (11) Consumimos $Y=1$, pero no genera nuevas respuestas.

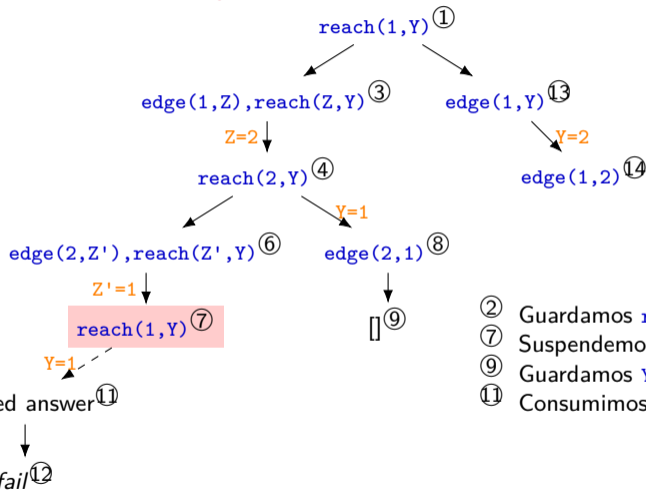
Memorización: 1b) Tabulación: Evaluación de $reach(1, Y)$



Objetivo	Respuestas
$reach(1, Y)$ (2)	$Y=1$ (10)
$reach(2, Y)$ (5)	$Y=1$ (9)

- (2) Guardamos $reach(1, Y)$ en la tabla.
- (7) Suspendemos la evaluación de $reach(1, Y)$.
- (9) Guardamos $Y=1$ para $reach(2, Y)$ y $reach(1, Y)$.
- (11) Consumimos $Y=1$, pero no genera nuevas respuestas.

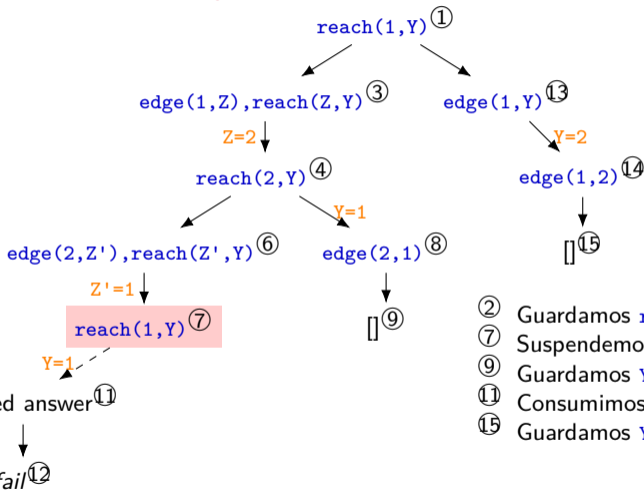
Memorización: 1b) Tabulación: Evaluación de $reach(1, Y)$



Objetivo	Respuestas
$reach(1, Y)$ (2)	$Y=1$ (10)
$reach(2, Y)$ (5)	$Y=1$ (9)

- (2) Guardamos $reach(1, Y)$ en la tabla.
- (7) Suspendemos la evaluación de $reach(1, Y)$.
- (9) Guardamos $Y=1$ para $reach(2, Y)$ y $reach(1, Y)$.
- (11) Consumimos $Y=1$, pero no genera nuevas respuestas.

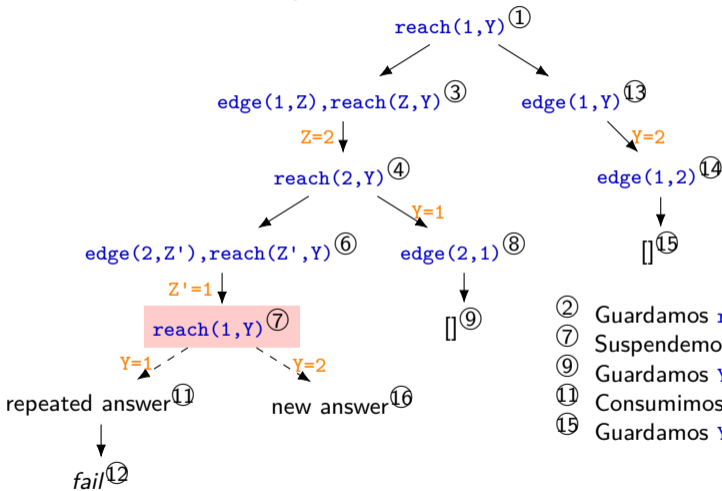
Memorización: 1b) Tabulación: Evaluación de $reach(1, Y)$



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

- (2) Guardamos $reach(1, Y)$ en la tabla.
- (7) Suspendemos la evaluación de $reach(1, Y)$.
- (9) Guardamos $Y=1$ para $reach(2, Y)$ y $reach(1, Y)$.
- (11) Consumimos $Y=1$, pero no genera nuevas respuestas.
- (15) Guardamos $Y=2$ y rearrancamos.

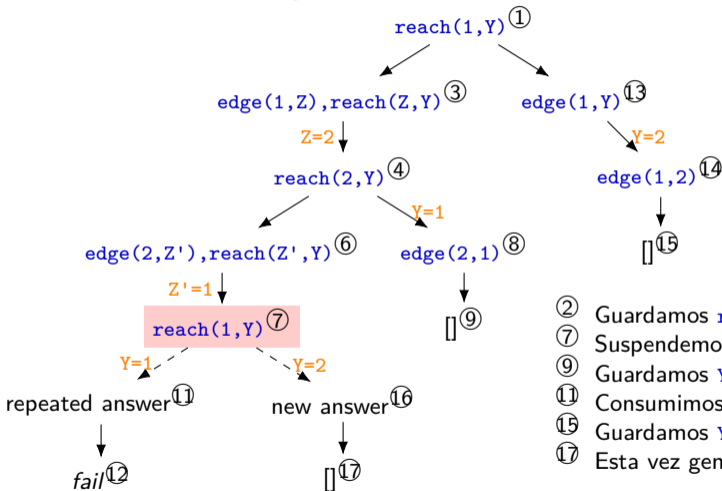
Memorización: 1b) Tabulación: Evaluación de $reach(1, Y)$



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

- (2) Guardamos $reach(1, Y)$ en la tabla.
- (7) Suspendemos la evaluación de $reach(1, Y)$.
- (9) Guardamos $Y=1$ para $reach(2, Y)$ y $reach(1, Y)$.
- (11) Consumimos $Y=1$, pero no genera nuevas respuestas.
- (15) Guardamos $Y=2$ y rearrancamos.

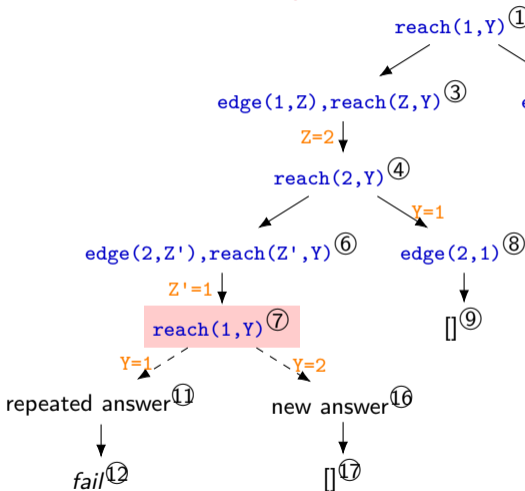
Memorización: 1b) Tabulación: Evaluación de $\text{reach}(1, Y)$



Objetivo	Respuestas
$\text{reach}(1, Y)$ (2)	Y=1 (10) Y=2 (15)
$\text{reach}(2, Y)$ (5)	Y=1 (9) Y=2 (17)

- (2) Guardamos $\text{reach}(1, Y)$ en la tabla.
- (7) Suspendemos la evaluación de $\text{reach}(1, Y)$.
- (9) Guardamos Y=1 para $\text{reach}(2, Y)$ y $\text{reach}(1, Y)$.
- (11) Consumimos Y=1, pero no genera nuevas respuestas.
- (15) Guardamos Y=2 y rearrancamos.
- (17) Esta vez genera nueva respuesta para $\text{reach}(1, Y)$.

Memorización: 1b) Tabulación: Evaluación de $reach(1, Y)$



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

- (2) Guardamos $reach(1, Y)$ en la tabla.
- (7) Suspendemos la evaluación de $reach(1, Y)$.
- (9) Guardamos Y=1 para $reach(2, Y)$ y $reach(1, Y)$.
- (11) 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.

Memorización: 1b) Tabulación: Ventajas

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

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

³La declaración `use_package` no es necesaria en Swi (online/local) pero si en Ciao (solo funciona en local).

Aritmética: Usando funciones del sistema

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

⁴Detalles sobre los operadores aritméticos disponibles en la documentación de `Ciao` y `Swi`.

Aritmética: Limitaciones con las funciones del sistema

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

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

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

- **Solución:** Afortunadamente existe una extensión de Prolog llamada restricciones (ver Tema 3.4, pág. 47) 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, etc

- Comparación de términos:

- Unifican: $A = B$, se cumple si A y B unifican.
- No unifican: $A \neq B$, se cumple si A no unifica con B .
- Idénticos: $A == B$, se cumple si son idénticos.
- No idénticos: $A \neq B$, se cumple si no son idénticos.

$$\begin{aligned} p(X, f(Y)) &= p(a, Z) \\ p(X, X) &\neq p(f(a), f(b)) \\ p(X) &== p(X) \\ p(X) &\neq p(Y) \end{aligned}$$

- 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.
- Functor y aridad: $\text{functor}(T, F, A)$ se cumple si F es el functor del término T y A es su aridad.
- N-esimo argumento: $\text{arg}(N, T, A)$ se cumple si A es el argumento del término T que ocupa el lugar N .
- Constante a ASCII: $\text{name}(A, L)$ se cumple si L es la lista de códigos ASCII de la constante A .

$$p(a, Y) =.. [p, a, Y].$$

$$\text{functor}(p(a, Y), p, 2)$$

$$\text{arg}(1, p(a, Y), a)$$

$$\text{name}('Hello', [72, 101, 108, 108, 111])$$

Inspeccionar estructuras, entrada/salida, etc (Cont.)

- Comprobación sobre tipos de términos.

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

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

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

- Manipulación de ficheros (lectura/escritura).

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

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

Inspeccionar estructuras, entrada/salida, etc (Cont.)

- Para imprimir en el top-level (o en ficheros) tenemos:
 - `display(Term)`: imprime `Term` por el stream actual.⁵
 - `nl`: imprime un salto de línea.
 - `format(Format,Arguments)`: imprime (simulando el `printf` de `stdio.h`) la lista de `Arguments` según el formato `Format`.
P.ej., `?- A=world, display('Hello '), display(A), nl` y `?- A=world, format('Hello ~p\n',[A])`, son equivalentes, ambas imprimen `Hello world`.
- Para leer del top-level (o de ficheros) hay predicados que permiten leer términos y cláusulas directamente. Ver documentación en `Ciao` y `Swi`.
 - ...también hay predicados para leer caracteres (`get_char(X)`) o incluso bytes (`get_byte(X)`), p.ej., el char `A` es el byte `65`.

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

Compilar scripts ejecutables main/1

- El siguiente código permite ejecutar `append/3` desde la línea de comandos:

```
1 :- use_module(library(actmod/actmod_dist)).
2 main([A1,B1,C1]) :-
3     format("Answer for append(~p,~p,~p)\n",[A1,B1,C1]),
4     atom_to_term(A1,A), atom_to_term(B1,B), atom_to_term(C1,C),
5     append(A,B,C),
6     format("X=~p\nY=~p\nZ=~p ?;\n\n",[A,B,C]),
7     fail.    % Force backtrack to generate all the answers
8 main(_) :- format("no\n",[]).    % Execution finish fine
```

para compilarlo con Ciao invocaremos `ciaoc -o append my_append.pl`, donde `append` es el ejecutable y `my_append.pl` el archivo con el código.⁶

- El ejecutable `append` puede invocarse con `append [1,2] [3,4] C`, con la llamada `append A B [1,2,3,4]` y con muchas otras...

⁶Más detalle sobre la compilación de scripts de Prolog en la documentación de `Ciao` y `Swi`.



Tema 3.2: Programación Procedimental.

Algoritmos de ordenamiento I: Especificación lógica

- Un procedimiento naive para ordenar una lista `Xs` consiste en encontrar una permutación `Ys` que este ordenada:
 - 1 `sort(Xs,Ys) :- permutation(Xs,Ys), ordered(Ys).`
 - Este predicado explota la semántica operacional de Prolog y es un ejemplo del paradigma **genera y prueba**.⁷
 - 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`.

⁷Por otro lado la programación lógica con restricciones (pág. 47) representa el paradigma **restringe y genera**.

Algoritmos de ordenamiento II: Divide y vencerás

- En este ejemplo implementaremos `quicksort/2` usando la estrategia de divide y vencerás...
 Ys es una versión ordenada de $[X|Xs]$ si $Littles$ y $Bigs$ son el resultado de particionar Xs según X ; Ls y Bs son el resultado de ordenar recursivamente $Littles$ y $Bigs$; e Ys es el resultado de concatenar $[X|Bs]$ a Ls .

es decir la regla recursiva sería:

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

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

Meta-interprete en Prolog, para Prolog

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

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

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

Trucos eficientes

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

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

Trucos eficientes II

4. Evita programas no-deterministas:

- Eliminando puntos de elección:

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

- Implementando programas deterministas de manera determinista:

```
1 plus_a(X,Y,N) :- var(X), var(Y), between(0,N,X), between(0,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 %
```

5. Ordenar los predicados para podar la búsqueda (más detalles en pág. 47).

Trucos eficientes III

6. Los interpretes de Prolog indexan por el primer argumento:

- Al compilar crean una tabla mirando la cabeza de las cláusulas de cada predicado.
- Durante la ejecución solo considera cláusulas compatibles con el primer argumento.

```

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).      %
    
```

7. 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_iter(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). %
    
```



Tema 3.3: Programación Lógica de 2º Orden.

Lógica de 2º Orden I: Recolección de soluciones

- El no-determinismo de Prolog provoca que un objetivo pueda generar varias respuestas y necesitemos agregarlas para su manipulación.
- Hay tres predicados predefinidos (básicos) para recolectar en una lista resultados de un objetivo (o conjunto de objetivos).⁸

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.

⁸Consulta la documentación de `Ciao` y `Swi` para más detalles.

Lógica de 2º Orden I: Recolección de soluciones (Cont.)

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

```

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

```

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

```

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

```

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

```

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

```

Lógica de 2º Orden I: Recolección de soluciones (Cont.)

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

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

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

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

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

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

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

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

Lógica de 2º Orden II: Predicados de orden superior

- Predicados de orden superior reciben como argumento otro predicado.
 - Implementados usando `call/1`, ya mencionado en pág. 11.
- Los más conocidos son `map/3`, `filter/3`, `foldl/4`, y `foldr/4`.
- (Ejemplo) Implementación de `map/3`:

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

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

Lógica de 2º Orden: Agregación dinámica

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

Evaluación incremental usando tabulación [Arias and Carro, 2019]

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

Lógica de 2º Orden III: Agregados, motivación

- Distancia entre nodos en un grafo.

1 `edge(a,b,2).`

2 `edge(a,b,4).`

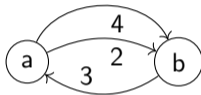
3 `edge(b,a,3).`

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.`

Modelo **infinito** para un grafo cíclico.



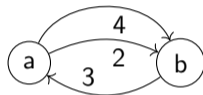
Lógica de 2º Orden III: Agregados, motivación

- Distancia entre nodos en un grafo.

```

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.
    
```

Modelo **infinito** para un grafo cíclico.



- La distancia más corta entre nodos.

Modelo **finito** también en grafos cíclicos.

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

La implementación usando findall/3 no termina

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

Lógica de 2º Orden III: Agregación de ?- $\text{dist}(a, Y, D)$

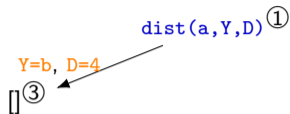
$\text{dist}(a, Y, D)$ ①

Objetivo	Respuestas

Lógica de 2º Orden III: Agregación de ?- $\text{dist}(a, Y, D)$

$\text{dist}(a, Y, D)$ ①

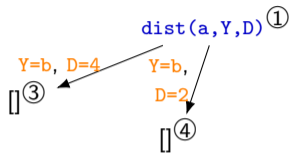
Objetivo	Respuestas
$\text{dist}(a, Y, D)$ ②	

Lógica de 2º Orden III: Agregación de $\text{?- dist}(a, Y, D)$ 

Objetivo	Respuestas
$\text{dist}(a, Y, D)$ ②	$Y=b, D=4$ ③

③ Guardamos $Y=b, D=4$ en la tabla.

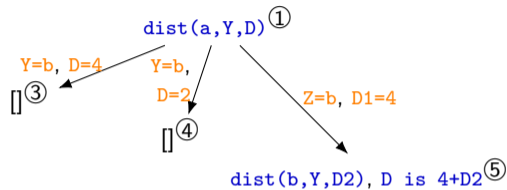
Lógica de 2º Orden III: Agregación de ?- $\text{dist}(a, Y, D)$



Objetivo	Respuestas
$\text{dist}(a, Y, D)$ ②	$Y=b, D=2$ ④

- ③ Guardamos $Y=b, D=4$ en la tabla.
- ④ Sustituimos la respuesta porque $\min(4, 2) = 2$.

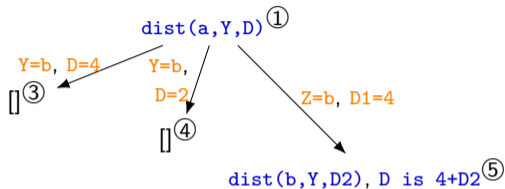
Lógica de 2º Orden III: Agregación de ?- $\text{dist}(a, Y, D)$



Objetivo	Respuestas
$\text{dist}(a, Y, D)$ ②	Y=b, D=2 ④

- ③ Guardamos $Y=b, D=4$ en la tabla.
- ④ Sustituimos la respuesta porque $\min(4, 2) = 2$.

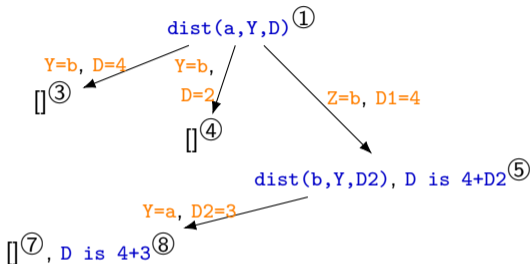
Lógica de 2º Orden III: Agregación de ?- dist(a, Y, D)



Objetivo	Respuestas
$\text{dist}(a, Y, D)$ ②	Y=b, D=2 ④
$\text{dist}(b, Y, D)$ ⑥	

- ③ Guardamos Y=b, D=4 en la tabla.
- ④ Sustituimos la respuesta porque $\min(4, 2) = 2$.

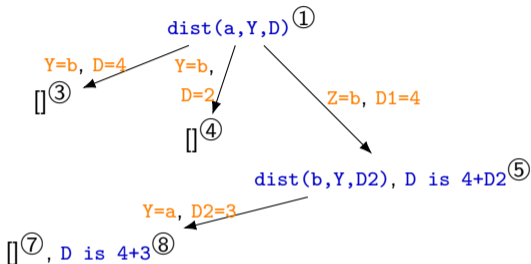
Lógica de 2º Orden III: Agregación de ?- dist(a, Y, D)



Objetivo	Respuestas
dist(a, Y, D) ②	Y=b, D=2 ④
dist(b, Y, D) ⑥	Y=a, D=3 ⑦

- ③ Guardamos $Y=b, D=4$ en la tabla.
- ④ Sustituimos la respuesta porque $\min(4, 2) = 2$.
- ⑦ Guardamos la respuesta de $\text{dist}(b, Y, D)$.

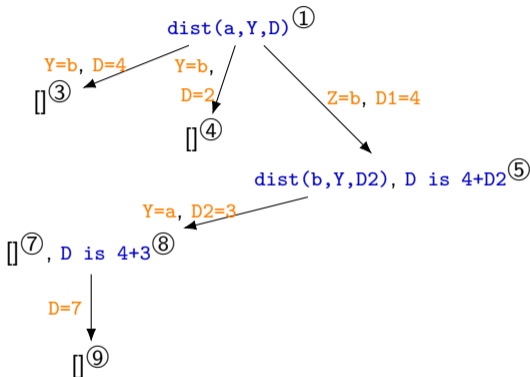
Lógica de 2º Orden III: Agregación de ?- dist(a, Y, D)



Objetivo	Respuestas
$\text{dist}(a, Y, D)$ ②	$Y=b, D=2$ ④
$\text{dist}(b, Y, D)$ ⑥	$Y=a, D=3$ ⑦

- ③ Guardamos $Y=b, D=4$ en la tabla.
- ④ Sustituimos la respuesta porque $\min(4, 2) = 2$.
- ⑦ Guardamos la respuesta de $\text{dist}(b, Y, D)$.

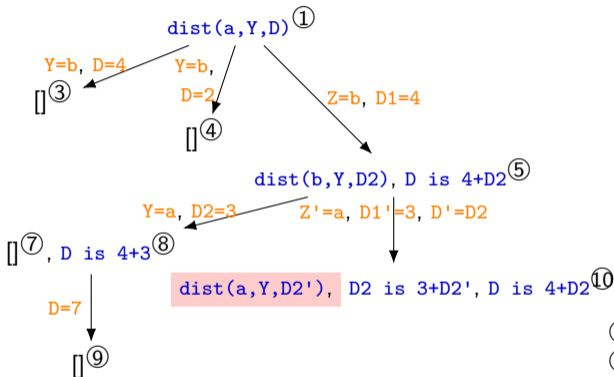
Lógica de 2º Orden III: Agregación de ?- dist(a, Y, D)



Objetivo	Respuestas
$\text{dist}(a, Y, D)$ ②	$Y=b, D=2$ ④ $Y=a, D=7$ ⑨
$\text{dist}(b, Y, D)$ ⑥	$Y=a, D=3$ ⑦

- ③ Guardamos $Y=b, D=4$ en la tabla.
- ④ Sustituimos la respuesta porque $\min(4, 2) = 2$.
- ⑦ Guardamos la respuesta de $\text{dist}(b, Y, D)$.

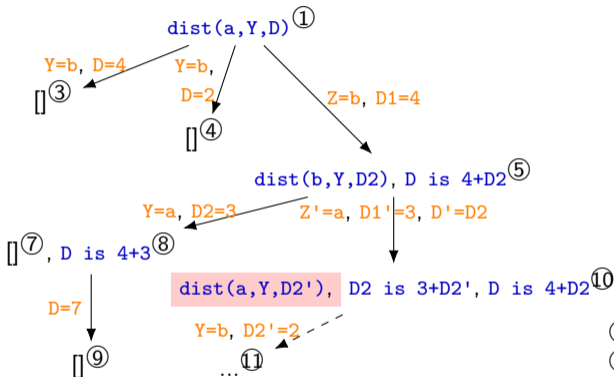
Lógica de 2º Orden III: Agregación de ?- dist(a, Y, D)



Objetivo	Respuestas
$\text{dist}(a, Y, D)$ ②	$Y=b, D=2$ ④ $Y=a, D=7$ ⑨
$\text{dist}(b, Y, D)$ ⑥	$Y=a, D=3$ ⑦

- ③ Guardamos $Y=b, D=4$ en la tabla.
- ④ Sustituimos la respuesta porque $\min(4, 2) = 2$.
- ⑦ Guardamos la respuesta de $\text{dist}(b, Y, D)$.

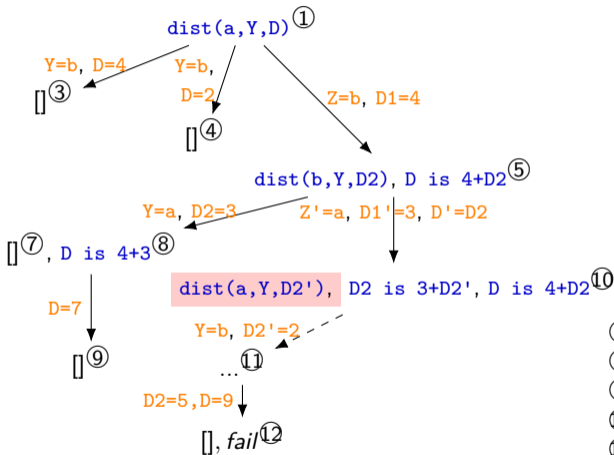
Lógica de 2º Orden III: Agregación de ?- dist(a, Y, D)



Objetivo	Respuestas
$\text{dist}(a, Y, D)$ ②	$Y=b, D=2$ ④ $Y=a, D=7$ ⑨
$\text{dist}(b, Y, D)$ ⑥	$Y=a, D=3$ ⑦

- ③ Guardamos $Y=b, D=4$ en la tabla.
- ④ Sustituimos la respuesta porque $\min(4, 2) = 2$.
- ⑦ Guardamos la respuesta de $\text{dist}(b, Y, D)$.
- ⑪ Consumimos respuestas de $\text{dist}(a, Y, D)$.

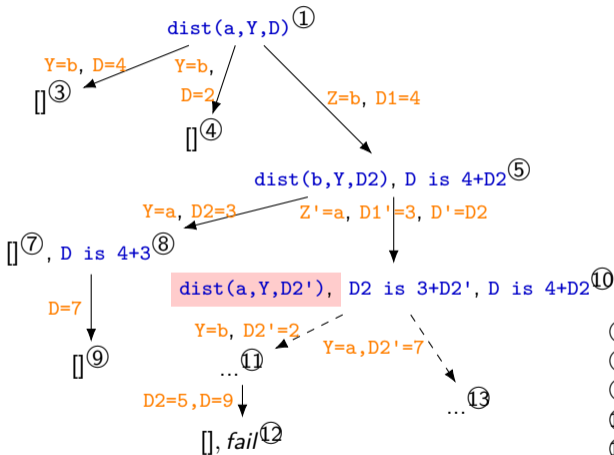
Lógica de 2º Orden III: Agregación de ?- dist(a, Y, D)



Objetivo	Respuestas
$\text{dist}(a, Y, D)$ ②	$Y=b, D=2$ ④ $Y=a, D=7$ ⑨
$\text{dist}(b, Y, D)$ ⑥	$Y=a, D=3$ ⑦ $Y=b, D=5$ ⑫

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

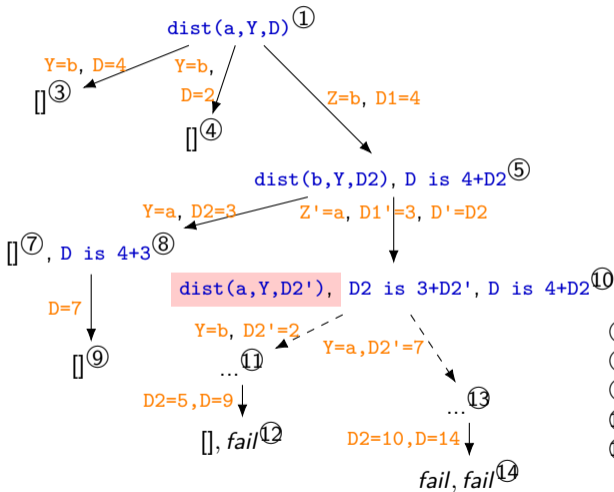
Lógica de 2º Orden III: Agregación de ?- dist(a, Y, D)



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

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

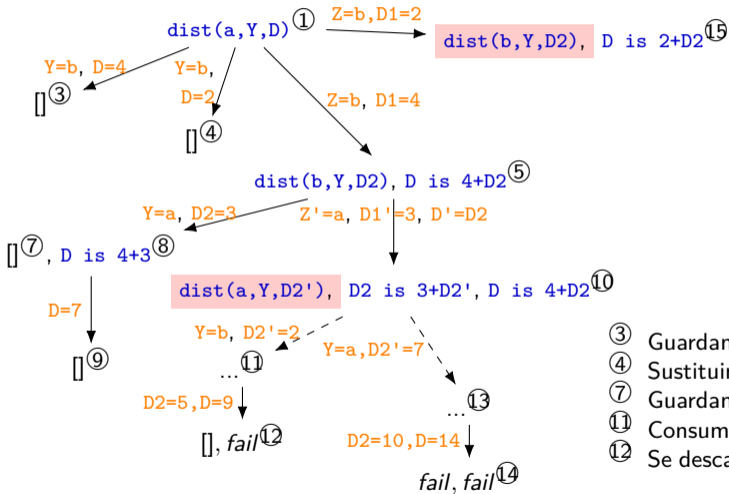
Lógica de 2º Orden III: Agregación de ?- dist(a, Y, D)



Objetivo	Respuestas
$\text{dist}(a, Y, D)$ ②	$Y=b, D=2$ ④ $Y=a, D=7$ ⑨
$\text{dist}(b, Y, D)$ ⑥	$Y=a, D=3$ ⑦ $Y=b, D=5$ ⑫

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

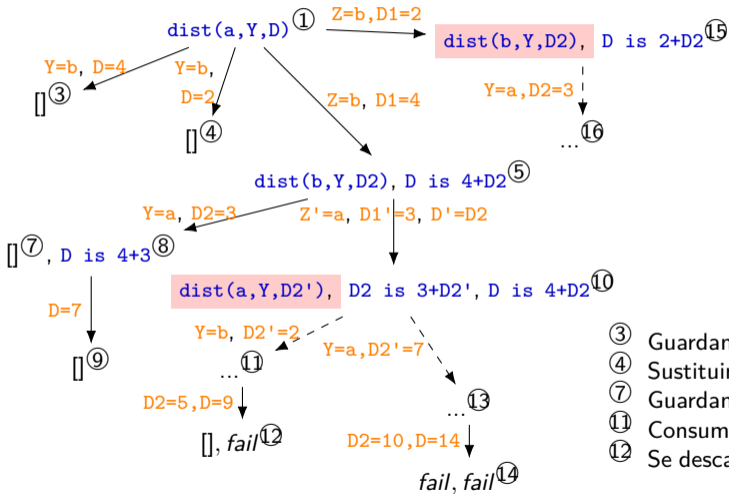
Lógica de 2º Orden III: Agregación de ?- dist(a, Y, D)



Objetivo	Respuestas
$\text{dist}(a, Y, D)$ ②	$Y=b, D=2$ ④ $Y=a, D=7$ ⑨
$\text{dist}(b, Y, D)$ ⑥	$Y=a, D=3$ ⑦ $Y=b, D=5$ ⑫

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

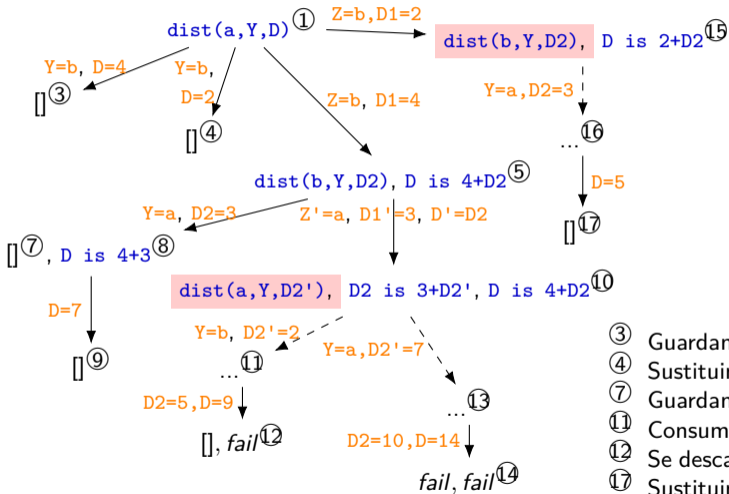
Lógica de 2º Orden III: Agregación de ?- dist(a, Y, D)



Objetivo	Respuestas
$\text{dist}(a, Y, D)$ ②	$Y=b, D=2$ ④ $Y=a, D=7$ ⑨
$\text{dist}(b, Y, D)$ ⑥	$Y=a, D=3$ ⑦ $Y=b, D=5$ ⑫

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

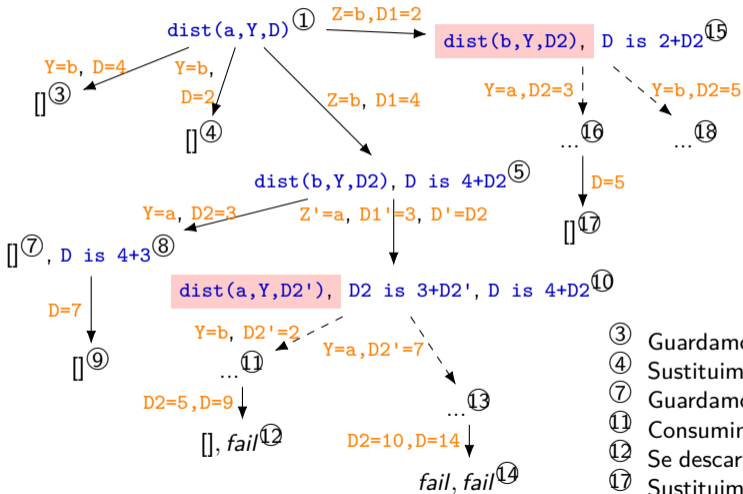
Lógica de 2º Orden III: Agregación de ?- $\text{dist}(a, Y, D)$



Objetivo	Respuestas
$\text{dist}(a, Y, D)$ ②	$Y=b, D=2$ ④ $Y=a, D=5$ ⑬
$\text{dist}(b, Y, D)$ ⑥	$Y=a, D=3$ ⑦ $Y=b, D=5$ ⑫

- ③ Guardamos $Y=b, D=4$ en la tabla.
- ④ Sustituimos la respuesta porque $\min(4, 2) = 2$.
- ⑦ Guardamos la respuesta de $\text{dist}(b, Y, D)$.
- ⑪ Consumimos respuestas de $\text{dist}(a, Y, D)$.
- ⑫ Se descartan respuestas mayores a la existente.
- ⑬ Sustituimos la respuesta porque $\min(7, 5) = 5$.

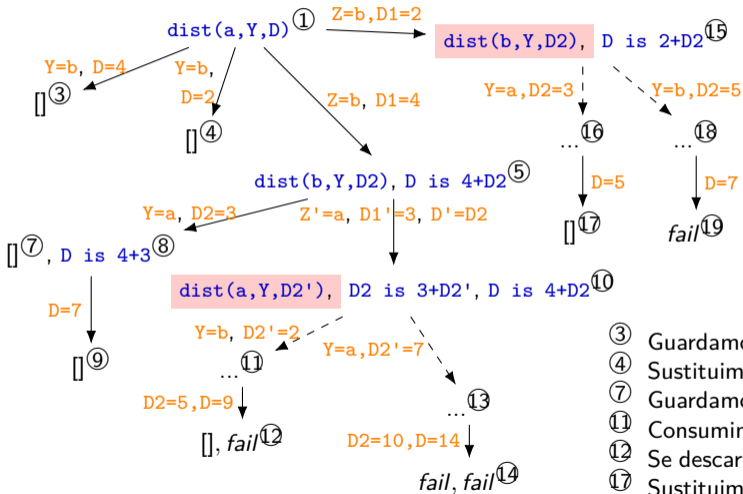
Lógica de 2º Orden III: Agregación de ?- $\text{dist}(a, Y, D)$



Objetivo	Respuestas
$\text{dist}(a, Y, D)$ ②	$Y=b, D=2$ ④ $Y=a, D=5$ ⑰
$\text{dist}(b, Y, D)$ ⑥	$Y=a, D=3$ ⑦ $Y=b, D=5$ ⑱

- ③ Guardamos $Y=b, D=4$ en la tabla.
- ④ Sustituimos la respuesta porque $\min(4, 2) = 2$.
- ⑦ Guardamos la respuesta de $\text{dist}(b, Y, D)$.
- ⑱ Consumimos respuestas de $\text{dist}(a, Y, D)$.
- ⑫ Se descartan respuestas mayores a la existente.
- ⑰ Sustituimos la respuesta porque $\min(7, 5) = 5$.

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



Objetivo	Respuestas
$\text{dist}(a, Y, D)$ ②	$Y=b, D=2$ ④ $Y=a, D=5$ ⑦
$\text{dist}(b, Y, D)$ ⑥	$Y=a, D=3$ ⑦ $Y=b, D=5$ ⑫

- ③ Guardamos $Y=b, D=4$ en la tabla.
- ④ Sustituimos la respuesta porque $\min(4, 2) = 2$.
- ⑦ Guardamos la respuesta de $\text{dist}(b, Y, D)$.
- ⑪ Consumimos respuestas de $\text{dist}(a, Y, D)$.
- ⑫ Se descartan respuestas mayores a la existente.
- ⑬ Sustituimos la respuesta porque $\min(7, 5) = 5$.

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

Ejemplo de codificación de *entailment-based* agregados.

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

Lógica de 2º Orden III: Codificación de Agregados en Ciao (cont.)

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

Aggregate	Code for entailment checking and join
least upper bound	<pre>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).</pre>
widest enclosing interval	<pre>interval(A1-A2,B1-B2,C1-C2) :- (A1=<B1 -> C1=A1; C1=B1), (A2>=B2 -> C2=A2; C2=B2).</pre>
set	<pre>set(A,B,C) :- ord_union(A,B,C).</pre>

Lógica de 2º Orden III: Codificación de Agregados en Ciao (cont.)


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

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

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

1. El problema de los **Juegos**.


ICLP 2015 LP/CP contest [ALP, 2015].

- Tienes que jugar **n** juegos al menos una vez. Algunos son mas divertidos que otros.
 - Tienes que administrar tus **Tokens** para obtener la mayor **Felicidad** de los juegos: 
- `reach(JuegoA, JuegoB, Tokens, Felicidad):-`

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

1. El problema de los **Juegos**.


ICLP 2015 LP/CP contest [ALP, 2015].

- Tienes que jugar **n** juegos al menos una vez. Algunos son mas divertidos que otros.
- Tienes que administrar tus **Tokens** para obtener la mayor **Felicidad** de los juegos: 
`reach(JuegoA,JuegoB,Tokens,Felicidad):-`
- Maximiza **Tokens** y **Felicidad** con el agregado `>=`. `:- agg_entail reach(_,_,>=,>=)`
 - **No** evalúa estados peores que otros ya evaluados.
 - Reduce el espacio de búsqueda!

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

1. El problema de los **Juegos**.

ICLP 2015 LP/CP contest [ALP, 2015].

- Tienes que jugar n juegos al menos una vez. Algunos son mas divertidos que otros.
 - Tienes que administrar tus **Tokens** para obtener la mayor **Felicidad** de los juegos: 
- ```
reach(JuegoA,JuegoB,Tokens,Felicidad):-
```
- Maximiza **Tokens** y **Felicidad** con el agregado  $\geq$ . `:- 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.

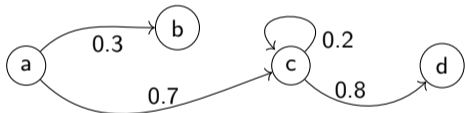
|              | Prolog   | Tabling | Agregados    |
|--------------|----------|---------|--------------|
| game_data_01 | 8062.49  | 14.66   | <b>2.89</b>  |
| game_data_02 | > 5 min. | 37.59   | <b>4.87</b>  |
| game_data_03 | > 5 min. | 1071.26 | <b>19.61</b> |
| game_data_04 | > 5 min. | 4883.00 | <b>23.21</b> |

## Agregados: Ejemplos (cont.)

### 2. Camino Aleatorio (Random walk):

Probabilidad  $P$  de alcanzar un nodo  $N$  desde  $a$ , considerando caminos aleatorios de  $a$  a  $N$ :

- Es la suma (add) de las probabilidades de transición de **todos** los caminos.
- Sin embargo, ciclos en un grafo pueden recorrerse un número ilimitado de veces...  
...descartemos caminos cuya contribución sea inferior a cierto umbral( $thr/1$ ).



$$P(b) = 0.3$$

$$P_0(d) = 0.7 * 0.8 = 0.56$$

$$P_1(d) = 0.7 * 0.2 * 0.8 = 0.112$$

$$P_2(d) = 0.7 * 0.2 * 0.2 * 0.8 = 0.0224$$

...

$$P_n(d) = 0.7 * 0.2^n * 0.8 = \dots$$

$$P(d) = 0.7$$

## Agregados: Ejemplos (cont.)

### 2. Camino Aleatorio (Random walk):

Probabilidad  $P$  de alcanzar un nodo  $N$  desde  $a$ , considerando caminos aleatorios de  $a$  a  $N$ :

- Es la suma (`add`) de las probabilidades de transición de **todos** los caminos.
- Sin embargo, ciclos en un grafo pueden recorrerse un número ilimitado de veces...  
 ...descartemos caminos cuya contribución sea inferior a cierto umbral(`thr/1`).

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



```

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



## Tema 3.4: Programación Lógica con Restricciones (CLP)

## Restricciones: Introducción

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



# Programación con Restricciones

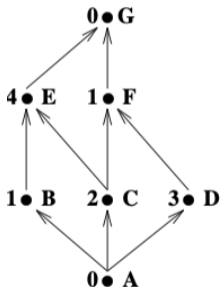
- Es necesario integrarlas en un lenguaje de programación que ofrezca:
  - Estructuras de datos y abstracción de datos.
  - Algoritmos ad-hoc, cuando se desee y/o sea ventajoso.
  - Modularidad.
- Aporta potencia computacional: el lenguaje anfitrión se enriquece.
- Permite establecer restricciones durante 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

- 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**.

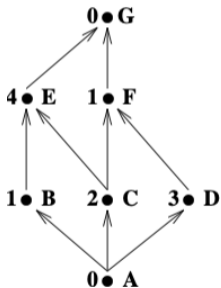
## Ejemplo Resolución de Restricciones

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



## Ejemplo Resolución de Restricciones

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



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

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

$$a \leq b, c, d$$

$$b + 1 \leq e$$

$$c + 2 \leq e$$

$$c + 2 \leq f$$

$$d + 3 \leq f$$

$$\#\#\#\#$$

$$f + 1 \leq g$$

## 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.

## Ejemplo Resolución de Restricciones (cont.)

- Hemos usado un dominio finito:
  - Cada variable tiene como dominio asociado los naturales del 1 al 10.
  - Hay inequaciones que relacionan las variables.
- Una posible estrategia de resolución es:
  - Evaluar las ecuaciones una a una.
  - Actualizar el dominio de las variables.
  - Termina cuando no hay mas actualizaciones.

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

## 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.

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

## 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.

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

$$c + 2 \leq e$$



## Ejercicio

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

# Sistema de Restricciones

- Las restricciones son condiciones que debe cumplir una solución:
  - $X + Y = 20$
  - $X \wedge Y$  es verdadero
  - El tercer campo de la estructura de datos es mayor que el segundo.
  - El asesino no conocía al mayordomo.
- CLP es Prolog extendido con la capacidad de calcular algún tipo de restricciones (que el sistema resuelve durante la búsqueda).
- Características (adicionales) de un sistema CLP:
  - Dominio (reales, racionales, enteros, booleanos, estructuras, etc).
  - Expresiones que se pueden construir ( $+$ ,  $*$ ,  $\wedge$ ,  $\vee$ ).
  - Restricciones permitidas: (in)ecuaciones ( $=$ ,  $\neq$ ,  $\leq$ ,  $\geq$ ,  $<$ ,  $>$ ).
  - Algoritmos de resolución de restricciones: simplex, gauss, etc.
- Soluciones: asignaciones a variables y/o restricciones entre variables.

## Sistema de Restricciones: Definición Formal

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

$\Sigma$  es el conjunto de símbolos de predicado y función, junto con su aridad.

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

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

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

## Sistema de Restricciones: Definición Formal

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

$\Sigma$  es el conjunto de símbolos de predicado y función, junto con su aridad.

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

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

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

- Y se cumple que:

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

Ver Jaffar and Maher (1994).

## Sistema de Restricciones: Dominios

$\mathbb{R}$  **Aritmética sobre Reales:**

$\Sigma = \{0, 1, +, *, =, <, \leq\}$   $\mathcal{D} =$  Reales interpretando  $\Sigma$  normalmente.

P.ej.:  $x^2 + 2xy < \frac{y}{x} \wedge x > 0$

$\mathbb{R}_{Lin}$  **Aritmética Lineal:**

$\Sigma = \{0, 1, +, =, <, \leq\}$   $\mathcal{D} =$  Reales interpretando  $\Sigma$  normalmente.

P.ej.:  $3x - y < 3$

$\mathbb{R}_{LinEq}$  **Ecuaciones lineales:**

$\Sigma = \{0, 1, +, =\}$   $\mathcal{D} =$  Reales interpretando  $\Sigma$  normalmente.

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

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

## Sistema de Restricciones: Dominios

$\mathbb{R}$  **Aritmética sobre Reales:**

$\Sigma = \{0, 1, +, *, =, <, \leq\}$      $\mathcal{D} =$  Reales interpretando  $\Sigma$  normalmente.

P.ej.:  $x^2 + 2xy < \frac{y}{x} \wedge x > 0$      $\equiv$      $xxx + xxy + xxy < y \wedge 0 < x$

$\mathbb{R}_{Lin}$  **Aritmética Lineal:**

$\Sigma = \{0, 1, +, =, <, \leq\}$      $\mathcal{D} =$  Reales interpretando  $\Sigma$  normalmente.

P.ej.:  $3x - y < 3$

$\mathbb{R}_{LinEq}$  **Ecuaciones lineales:**

$\Sigma = \{0, 1, +, =\}$      $\mathcal{D} =$  Reales interpretando  $\Sigma$  normalmente.

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

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

## Sistema de Restricciones: Dominios

$\mathbb{R}$  **Aritmética sobre Reales:**

$\Sigma = \{0, 1, +, *, =, <, \leq\}$      $\mathcal{D} =$  Reales interpretando  $\Sigma$  normalmente.

P.ej.:  $x^2 + 2xy < \frac{y}{x} \wedge x > 0$      $\equiv$      $xxx + xxy + xxy < y \wedge 0 < x$

$\mathbb{R}_{Lin}$  **Aritmética Lineal:**

$\Sigma = \{0, 1, +, =, <, \leq\}$      $\mathcal{D} =$  Reales interpretando  $\Sigma$  normalmente.

P.ej.:  $3x - y < 3$      $\equiv$      $x + x + x < 1 + 1 + 1$

$\mathbb{R}_{LinEq}$  **Ecuaciones lineales:**

$\Sigma = \{0, 1, +, =\}$      $\mathcal{D} =$  Reales interpretando  $\Sigma$  normalmente.

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

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

## Sistema de Restricciones: Dominios

$\mathbb{R}$  **Aritmética sobre Reales:**

$\Sigma = \{0, 1, +, *, =, <, \leq\}$      $\mathcal{D} =$  Reales interpretando  $\Sigma$  normalmente.

P.ej.:  $x^2 + 2xy < \frac{y}{x} \wedge x > 0$      $\equiv$      $xxx + xxy + xxy < y \wedge 0 < x$

$\mathbb{R}_{Lin}$  **Aritmética Lineal:**

$\Sigma = \{0, 1, +, =, <, \leq\}$      $\mathcal{D} =$  Reales interpretando  $\Sigma$  normalmente.

P.ej.:  $3x - y < 3$      $\equiv$      $x + x + x < 1 + 1 + 1$

$\mathbb{R}_{LinEq}$  **Ecuaciones lineales:**

$\Sigma = \{0, 1, +, =\}$      $\mathcal{D} =$  Reales interpretando  $\Sigma$  normalmente.

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

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



## Sistema de Restricciones: Dominios (cont.)

FT **Árboles finitos o dominio de Herbrand:**<sup>1</sup>

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

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

P.ej.:  $g(h(Z), Y) = g(Y, h(a))$

*String* **Ecuaciones sobre strings:**

$\Sigma = \{a, b, \dots, \lambda, \cdot, =\}$      $\mathcal{D} =$  string donde:

- $\cdot$  se interpreta como concatenación de strings.

P.ej.:  $X.A.X = X.A$

<sup>1</sup>Prolog puede verse como una sistema de restricciones sobre términos de Herbrand con un único símbolo de restricción  $=$ .

## Sistema de Restricciones: Dominios (cont.)

FT **Árboles finitos o dominio de Herbrand:**<sup>1</sup>

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

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

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

*String* **Ecuaciones sobre strings:**

$\Sigma = \{a, b, \dots, \lambda, \cdot, =\}$      $\mathcal{D} =$  string donde:

- $\cdot$  se interpreta como concatenación de strings.

P.ej.:  $X.A.X = X.A$

<sup>1</sup>Prolog puede verse como una sistema de restricciones sobre términos de Herbrand con un único símbolo de restricción  $=$ .

## Sistema de Restricciones: Dominios (cont.)

FT **Árboles finitos o dominio de Herbrand:**<sup>1</sup>

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

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

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

*String* **Ecuaciones sobre strings:**

$\Sigma = \{a, b, \dots, \lambda, \cdot, =\}$      $\mathcal{D} =$  string donde:

- $\cdot$  se interpreta como concatenación de strings.

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

<sup>1</sup>Prolog 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** (*Bool*).
2. Escribe una restricción para uno de los siguientes dominio de restricciones y resuélvela:
  - $\mathbb{R}$ .
  - $\mathbb{Q}_{Lin}$ .
  - $\mathbb{R}_{LinEq}$ .
  - $\mathbb{Q}$ .
  - *FT*.
  - *String*.
  - *Bool*.

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

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

## Resolutor de restricciones: Fibonacci usando CLP(R)

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

```
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).
```

## Resolutor de restricciones: Fibonacci usando CLP(R)

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

```
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.

## CLP: Reduce el espacio de búsqueda: Prolog vs. CLP

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



## CLP: Reduce el espacio de búsqueda: Prolog vs. CLP

- 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:  
 $X = 14, Y = 15, Z = 16$  ?

## CLP: Reduce el espacio de búsqueda: Prolog vs. CLP

- Encontrar tres números consecutivos de la relación `p/1`.
- De “genera y prueba” con Prolog. 
- ...a “restringe y genera” con CLP. 

```

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 ?`

```

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: Reduce el espacio de búsqueda: Prolog vs. CLP

- 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).

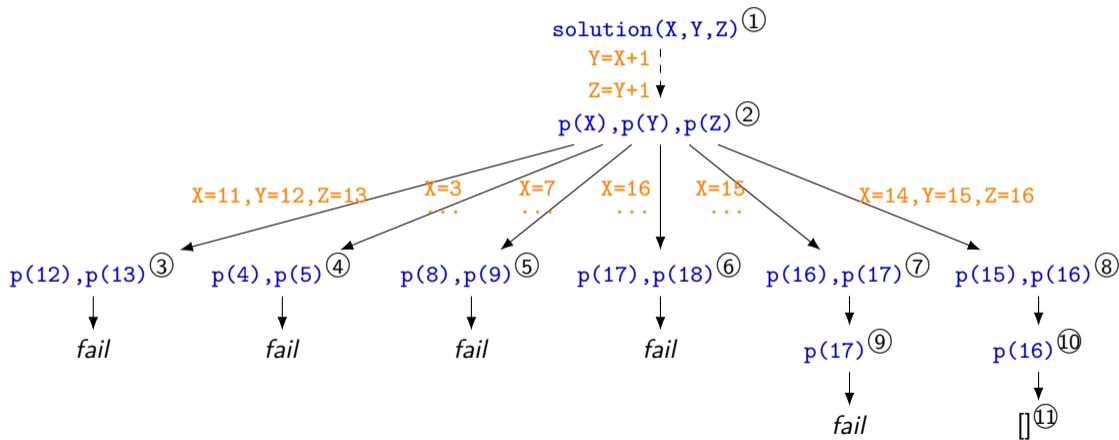
```
- ...a “restringe y genera” con CLP. 

```


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).

```
- La consulta `?- solution(X,Y,Z)` requiere **458 pasos** para encontrar la solución:  
`X = 14, Y = 15, Z = 16 ?`
- CLP(R) solo necesita 11 pasos!

# CLP: Reduce el espacio de búsqueda: Árbol de derivación



## CLP: Reduce el espacio de búsqueda: Ejercicios


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).
```

2. 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).
```

# Referencias I

- [1] ALP (2015). **ICLP 2015 LP/CP contest**. URL: <https://www.cs.nmsu.edu/ALP/2015/09/report-2015-lpcp-programming-contest/>.
- [2] J. Arias and M. Carro (2019). **Incremental Evaluation of Lattice-Based Aggregates in Logic Programming Using Modular TCLP**. In: *21st Int'l. Symposium on Practical Aspects of Declarative Languages*. Vol. 11372. LNCS. Springer, pp. 98–114. DOI: [10.1007/978-3-030-05998-9\\_7](https://doi.org/10.1007/978-3-030-05998-9_7).
- [3] M. Bauer (2019). **A Comparison of Six Constraint Solvers for Variability Analysis**. Department of Informatics and Mathematics, Master's Thesis. University of Passau. URL: <https://www.se.cs.uni-saarland.de/theses/MartinBauerMA.pdf>.
- [4] D. Cabeza, M. Carro, and M. Hermenegildo (n.d.). **Breadth-first execution**. The Ciao System. URL: <https://shorturl.at/AWXWe>.
- [5] K. L. Clark (1978). **Negation as Failure**. In: *Logic and Data Bases*. Ed. by H. Gallaire and J. Minker. Springer, pp. 293–322. DOI: [10.1007/978-1-4684-3384-5\\_11](https://doi.org/10.1007/978-1-4684-3384-5_11).

## Referencias II

- [6] M. Gelfond and V. Lifschitz (1988). **The Stable Model Semantics for Logic Programming**. In: *5th International Conference on Logic Programming*, pp. 1070–1080. DOI: [10.2307/2275201](https://doi.org/10.2307/2275201).
- [7] C. Holzbaaur (1995). **OFAI CLP(Q,R) Manual, Edition 1.3.3**. Tech. rep. TR-95-09. Vienna: Austrian Research Institute for Artificial Intelligence.
- [8] J. Jaffar and M. Maher (1994). **Constraint Logic Programming: A Survey**. In: *Journal of Logic Programming* 19/20, pp. 503–581. DOI: [10.1016/0743-1066\(94\)90033-7](https://doi.org/10.1016/0743-1066(94)90033-7).
- [9] H. Tamaki and M. Sato (1986). **OLD Resolution with Tabulation**. In: *Third International Conference on Logic Programming*. London: Lecture Notes in Computer Science, Springer-Verlag, pp. 84–98.
- [10] D. S. Warren (1992). **Memoing for Logic Programs**. In: *Communications of the ACM* 35.3, pp. 93–111.



Universidad  
Rey Juan Carlos

Escuela Técnica Superior  
Ingeniería Informática

# Tema 4: s(CASP).

## Asignatura: Programación Declarativa

Joaquín Arias

Universidad Rey Juan Carlos  
Escuela Técnica Superior de Ingeniería Informática

Curso 2024-2025



# Índice I

## Tema 4.1: Introduction, Applications and Background.

### Introduction

Background

Why s(CASP)

### Applications

## Tema 4.2: Syntax and Operational Semantics

### Getting Started

Foundations

### Installation

## DEMO



Copyright ©2024 Joaquín Arias.

Esta obra, basada en transparencias de Ana Pradera (URJC'23) & ClipLab (UPM'24), está bajo la licencia CC BY-SA 4.0, Creative Commons Atribución-CompartirIgual 4.0 Internacional. Cómo citar esta obra: Arias, Joaquín (2024). Programación Lógica: hasta Prolog y más allá. BURJC, Madrid.



# Tema 4.1: Introduction, Applications and Background.

# Introduction

---

<sup>1</sup>Under Ciao Prolog is available at <http://ciao-lang.org/>.

<sup>2</sup>Under SWI Prolog is available at <https://www.swi-prolog.org/>.

# Introduction

- s(CASP), by Arias et al. (2018), is a goal directed execution of Constraint Answer Set Programming based on:
  - Galliwasp, by Marple and Gupta (2012), for propositional logic.
  - s(ASP) by Marple et al. (2017a), for first order logic..
- It avoids the grounding phase (required by traditional ASP systems).
  - It can constraint variables that, as in CLP, are kept during execution and in answer sets (which are partial models).
- Additionally it provides human-understandable justifications.
- s(CASP) is implemented in Prolog (Ciao<sup>1</sup> / SWI<sup>2</sup> Prolog).

---

<sup>1</sup>Under Ciao Prolog is available at <http://ciao-lang.org/>.

<sup>2</sup>Under SWI Prolog is available at <https://www.swi-prolog.org/>.

## Background: Logic Programming

- Logic programming is based on First Order Logic.
  - A program is a set of formulas and the goal is to infer new knowledge.
- Aristotle (4th century BC) formalized human reasoning using Logic:

All men are mortal. Socrates is a man. Therefore, Socrates is mortal.

## Background: Logic Programming

- Logic programming is based on First Order Logic.
  - A program is a set of formulas and the goal is to infer new knowledge.
- Aristotle (4th century BC) formalized human reasoning using Logic:

All men are mortal. Socrates is a man. Therefore, Socrates is mortal.

Logic

vs.

Implementation using Prolog

$T[ \text{Man}(x) \rightarrow \text{Mortal}(x),$   
 $\text{Man}(\text{socrates}) ]$   
 $\models \text{Mortal}(\text{socrates})$

```
mortal(X) :- man(X).
man(socrates).
?- mortal(socrates).
```

## Background: Logic Programming (cont.)

### Concatenate two lists

```
1 concatenate([], Ys, Ys).
2 concatenate([X|Xs], Ys, [X|Zs]) :- concatenate(Xs, Ys, Zs).
3
4 ?- concatenate([1,2], [3,4], Zs). % What is the value of Zs?
```

## Background: Logic Programming (cont.)

### Concatenate two lists

```
1 concatenate([], Ys, Ys).
2 concatenate([X|Xs], Ys, [X|Zs]) :- concatenate(Xs, Ys, Zs).
3
4 ?- concatenate([1,2], [3,4], Zs). % What is the value of Zs?
```

Functions are evaluated to a single result, but predicates can be “consulted” in different ways:

- For the query

```
?- concatenate([1,2], Ys, [1,2,3,4])
```

we obtain

1. `Ys = [3,4]`



## Background: Logic Programming (cont.)

### Concatenate two lists

```
1 concatenate([], Ys, Ys).
2 concatenate([X|Xs], Ys, [X|Zs]) :- concatenate(Xs, Ys, Zs).
3
4 ?- concatenate([1,2], [3,4], Zs). % What is the value of Zs?
```

Functions are evaluated to a single result, but predicates can be “consulted” in different ways:

- For the query  
`?- concatenate([1,2], Ys, [1,2,3,4])`  
we obtain
  1. `Ys = [3,4]`
- ... and for:  
`?- concatenate(ListA, ListB, [1,2,3,4])`  
we obtain

## Background: Logic Programming (cont.)

### Concatenate two lists

```

1 concatenate([], Ys, Ys).
2 concatenate([X|Xs], Ys, [X|Zs]) :- concatenate(Xs, Ys, Zs).
3
4 ?- concatenate([1,2], [3,4], Zs). % What is the value of Zs?

```

Functions are evaluated to a single result, but predicates can be “consulted” in different ways:

- For the query

```
?- concatenate([1,2], Ys, [1,2,3,4])
```

we obtain

1. Ys = [3,4]

- ... and for:

```
?- concatenate(ListA, ListB, [1,2,3,4])
```

we obtain 5 answers:

1. ListA = [], ListB = [1,2,3,4]
2. ListA = [1], ListB = [2,3,4]
3. ListA = [1,2], ListB = [3,4]
4. ListA = [1,2,3], ListB = [4]
5. ListA = [1,2,3,4], ListB = []

## Background: Logic Programming + Constraints (...much more)

- Constraints allow us to express relationships between variables by means of equations
- E.g., we can define the relationship of a mortgage as:

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

$\text{mg}(P, T, \_, \_, B) :- T = 0, B = P.$

$\text{mg}(P, T, R, I, B) :- T >= 1, NP = P + P*I - R, NT = T - 1, \text{mg}(NP, NT, R, I, B).$

Likewise, we can ask in different ways:

## Background: Logic Programming + Constraints (...much more)

- Constraints allow us to express relationships between variables by means of equations
- E.g., we can define the relationship of a mortgage as:

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

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

$mg(P, T, R, I, B) :- T >= 1, NP = P + P*I - R, NT = T - 1, mg(NP, NT, R, I, B).$

Likewise, we can ask in different ways:

$?- mg(1000, 10, 150, 0.10, B).$        $?- mg(P, 10, 150, 0.10, 0).$   
 B = 203.13 ?                              P = 921.68 ?

## Background: Logic Programming + Constraints (...much more)

- Constraints allow us to express relationships between variables by means of equations
- E.g., we can define the relationship of a mortgage as:

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

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

$mg(P, T, R, I, B) :- T >= 1, NP = P + P*I - R, NT = T - 1, mg(NP, NT, R, I, B).$

Likewise, we can ask in different ways:

?-  $mg(1000, 10, 150, 0.10, B).$   
B = 203.13 ?

?-  $mg(P, 10, 150, 0.10, 0).$   
P = 921.68 ?

?-  $mg(P, 10, R, 0.10, B).$   
P = 6.14\*R + 0.38\*B ?

## Background: Logic Programming + Negation (+ Constraints)

- **ASP**: Answer Set Programming is based on the stable model semantics [16], supporting non-stratified negation:

```
% Even loop
```

```
p :- not q.
```

```
q :- not p.
```

Models: {p}, {q}

```
% Odd loop
```

```
p :- not q.
```

```
q :- p.
```

Models: no models

## Background: Logic Programming + Negation (+ Constraints)

- **ASP**: Answer Set Programming is based on the stable model semantics [16], supporting non-stratified negation:

```
% Even loop
```

```
p :- not q.
```

```
q :- not p.
```

Models: {p}, {q}

```
% Odd loop
```

```
p :- not q.
```

```
q :- p.
```

Models: no models

- **s(CASP)**: It is a top-down, goal-directed interpreter.
  - Can provide justifications (in natural language) [Arias et al., 2020].
    - They can be manipulated (to hide sensitive information) using the directive `#show` and the flag `--short`.
  - It solves negated atoms (`not p`) against the dual rules of the program (the negation of the rules present in the program).

## Why s(CASP)?

## COMMONSENCE REASONING

- Humans employ commonsense reasoning to explain things:  
E.g., we convert sensory input to knowledge, over which we reason.
- “Thinking Fast and Slow” (2011): System 1 vs. System 2. Explainability and interpretability in AI need to automate the reflective thinking system (System 2).



## Why s(CASP)?

## COMMONSENCE REASONING

- Humans employ commonsense reasoning to explain things:  
E.g., we convert sensory input to knowledge, over which we reason.
- “Thinking Fast and Slow” (2011): System 1 vs. System 2. Explainability and interpretability in AI need to automate the reflective thinking system (System 2).

Commonsense reasoning can be approximated with answer set programming

## Why s(CASP)?

## COMMONSENCE REASONING

- Humans employ commonsense reasoning to explain things:  
E.g., we convert sensory input to knowledge, over which we reason.
- “Thinking Fast and Slow” (2011): System 1 vs. System 2. Explainability and interpretability in AI need to automate the reflective thinking system (System 2).

Commonsense reasoning can be approximated with answer set programming

- **Default Rules:** express what is true in a majority of cases but not always.  
E.g., “by default birds fly, but there are exceptional ones that do not”.
- ```
flies(X) :- bird(X), not ab0(X).              ab0(X) :- penguin(X).
```

Why s(CASP)?

COMMONSENCE REASONING

- Humans employ commonsense reasoning to explain things:
 E.g., we convert sensory input to knowledge, over which we reason.
- “Thinking Fast and Slow” (2011): System 1 vs. System 2. Explainability and interpretability in AI need to automate the reflective thinking system (System 2).

Commonsense reasoning can be approximated with answer set programming

- **Default Rules:** express what is true in a majority of cases but not always.
 E.g., “by default birds fly, but there are exceptional ones that do not”.
`flies(X) :- bird(X), not ab0(X). ab0(X) :- penguin(X).`
- **Integrity Constraints:** express impossibility conditions.
`false :- sit(X), stand(X).` E.g., A person cannot sit and stand at the same time.

Why s(CASP)?

COMMONSENCE REASONING

- Humans employ commonsense reasoning to explain things:
E.g., we convert sensory input to knowledge, over which we reason.
- “Thinking Fast and Slow” (2011): System 1 vs. System 2. Explainability and interpretability in AI need to automate the reflective thinking system (System 2).

Commonsense reasoning can be approximated with answer set programming

- **Default Rules:** express what is true in a majority of cases but not always.
E.g., “by default birds fly, but there are exceptional ones that do not”.
`flies(X) :- bird(X), not ab0(X).` `ab0(X) :- penguin(X).`
- **Integrity Constraints:** express impossibility conditions.
`false :- sit(X), stand(X).` E.g., A person cannot sit and stand at the same time.
- **Assumption based reasoning:** if we don't know something, we will assume it holds (or does not hold) and continue (abductive reasoning). E.g., Game of clue.

Why s(CASP)?

EXPLAINABILITY

- Provide a human-understandable justifications for AI advice:
 - ML-based AI systems cannot explain their decisions.
 - But: ethical & legal concerns (e.g., AI Act by EU).
- Answer Set Programming used to develop intelligent applications, e.g., medical advisor systems.
 - An ASP program is a set of rules encoding knowledge.
 - **Explainability**: justify decision in terms of the program rules.

However:

- Current ASP explanation frameworks [12; 23; 24]:
 - Only support grounded programs,
 - ... or do not justify negated literals,
 - ... and, do not support constraints and/or dense domains.

Applications of s(CASP)

- Event Calculus Reasoner. [Arias et al., 2019]
 - Model real-world avionics systems. [Hall et al., 2021]
 - Used for cyber-defense (network of device). [Moyle et al., 2023]
- Explainable Artificial Intelligence (XAI). [Arias et al., 2020]
 - Medical advice system. [Chen et al., 2016]
 - Inductive Logic Programming. [Shakerin and Gupta, 2019]
- Natural language understanding systems. [Basu et al., 2021a]
 - CASPR: chatbot for “Alexa Grand Challenge 4”. [Basu et al., 2021b]
- Coding rule 34 of the Singapore Bar + **Blawx**. [Morris, 2021]
- Online playground for **Swi Prolog**. [Wielemaker et al., 2021]
 - ...and for **Ciao Prolog**. [García-Pradales et al., 2022]

Applications of s(CASP) cont.

- Administrative and judicial discretion reasoner. [Arias et al., 2021]
 - Extended version of s(LAW) using variables. [Arias et al., 2023a]
 - Towards value-awareness in administrative processes. [Arias et al., 2024]
 - A forgetting technique for XAI. [Fidilio-Allende and Arias, 2024]
- Spatial reasoner for Building Information Modelling. [Arias et al., 2022]
- Teaching Logic (Programming) + s(CASP). [Arias, 2023]
 - Deduccionnatural.pl: base case. [Arias et al., 2023b]

... more at [GDE'21](#), [GDE'22](#), [GDE'23](#), (and [GDE'24³](#)) Workshops.

³Workshop proposed within the ICLP 2024 in Dallas (USA).



Tema 4.2: **Syntax and Operational Semantics**

Getting Started: Syntax of s(CASP)

- Goal directed execution of CASP programs **without** grounding.
 - The execution starts with a query. `?- T#>5, T#<8, cross(T).`
 - Returns (partial) stable models. Only literals supporting the query.
 - For each successful top-down derivation, on backtracking returns:
 - A justification tree. Explanation for observations.
 - Bindings/constraints in the model. `{T#>5, T#<7/3, cross(T), train(7.3)}`.

Getting Started: Syntax of s(CASP)

- Goal directed execution of CASP programs **without** grounding.
 - The execution starts with a query. `?- T#>5, T#<8, cross(T).`
 - Returns (partial) stable models. Only literals supporting the query.
 - For each successful top-down derivation, on backtracking returns:
 - A justification tree. Explanation for observations.
 - Bindings/constraints in the model. `{T#>5, T#<7/3, cross(T), train(7.3)}`.
- Provides a constructive and sound *default* negation:
 - Make deductions in the absence of positive information. `cross(T):- not train(T).`
 - Dual rules (synthesized by **s(CASP)** compiler) infer conditions for goal to fail.

Getting Started: Syntax of s(CASP)

- Goal directed execution of CASP programs **without** grounding.
 - The execution starts with a query. `?- T#>5, T#<8, cross(T).`
 - Returns (partial) stable models. `Only literals supporting the query.`
 - For each successful top-down derivation, on backtracking returns:
 - A justification tree. `Explanation for observations.`
 - Bindings/constraints in the model. `{T#>5, T#<7/3, cross(T), train(7.3)}.`
- Provides a constructive and sound *default* negation:
 - Make deductions in the absence of positive information. `cross(T):- not train(T).`
 - Dual rules (synthesized by *s(CASP)* compiler) infer conditions for goal to fail.
- Provides support for *classical* negation.
 - Represent explicit (negative) knowledge. `cross(T):- -train(T).`
 - Allows rules with negated heads. `-train(T):- not barrier(up,T).`
 - Global constraints ensure consistency. `:- train(T), -train(T).`

Getting Started: s(CASP) Bindings

- s(CASP) can return models with the bindings for negated calls:

- Using the disequality constraint solver...

```
% Program                                % Model
p(a).                                     { not p(X | {X \= a} ) }
% Query                                   % Bindings
?- not p(X)                               X \= a
```

- and a solver for linear constraints on rationals (and reals).

```
% Program                                % Model
p(X) :- X #> 0.                           { not p(X | {X #=< 0} ) }
% Query                                   % Bindings
?- not p(X).                               X #=< 0
```

Getting Started: Uninterpreted Functions

- ASP grounded program with lists, as `[f(a)|Rest]`, can be infinite.

```
% Program                                % Query
member(X, [X|Xs]).                        ?- list(A), not member(B, A).
member(X, [_|Xs]):- member(X, Xs).
list([1,2,3,4,5]).
```

Getting Started: Uninterpreted Functions

- ASP grounded program with lists, as `[f(a)|Rest]`, can be infinite.

```

% Program                                % Query
member(X, [X|Xs]).                       ?- list(A), not member(B, A).
member(X, [_|Xs]):- member(X, Xs).
list([1,2,3,4,5]).

% Model
{ list([1,2,3,4,5]),
  not member(B | {B ≠ 1,B ≠ 2,B ≠ 3,B ≠ 4,B ≠ 5}, [1,2,3,4,5]),
  not member(B | {B ≠ 1,B ≠ 2,B ≠ 3,B ≠ 4,B ≠ 5}, [2,3,4,5]),
  not member(B | {B ≠ 1,B ≠ 2,B ≠ 3,B ≠ 4,B ≠ 5}, [3,4,5]),
  not member(B | {B ≠ 1,B ≠ 2,B ≠ 3,B ≠ 4,B ≠ 5}, [4,5]),
  not member(B | {B ≠ 1,B ≠ 2,B ≠ 3,B ≠ 4,B ≠ 5}, [5]),
  not member(B | {B ≠ 1,B ≠ 2,B ≠ 3,B ≠ 4,B ≠ 5}, []) }

% Bindings
A = [1,2,3,4,5], B ≠ 1, B ≠ 2, B ≠ 3, B ≠ 4, B ≠ 5

```

Getting Started: (partial) Stable Models

- Given the even loop with predicates...

```
opera(saturday) :- not home(saturday).  
home(saturday) :- not opera(saturday).  
dinner(sunday).
```

```
?- opera(saturday) returns {opera(saturday), not home(saturday)}  
?- home(saturday) returns {home(saturday), not opera(saturday)}  
?- dinner(sunday) returns {dinner(sunday)}  
?- opera(saturday), home(saturday) returns no models.
```

- Additionally, s(CASP) supports **default** and **classical** negation:
 - `not opera(saturday)`: no evidence (we *cannot prove* it).
 - `-opera(saturday)`: explicit evidence (there is a *proof*).

Getting Started: Ophthalmologist diagnosis [Schulz and Toni, 2016]

- Justifying recommended treatments increases trust.

```
% Treatments
```

```
intraocularLens :-  
    correctiveLens,  
    not glasses,  
    not contactLens.
```

```
laserSurgery :-  
    shortSighted,  
    not tightOnMoney,  
    not correctiveLens.
```

```
glasses :-  
    correctiveLens,  
    not caresPracticality,  
    not contactLens.
```

```
contactLens :-  
    correctiveLens,  
    not afraidToTouchEyes,  
    not longSighted,  
    not glasses.
```

```
% Auxiliar
```

```
correctiveLens :-  
    shortSighted,  
    not laserSurgery.
```

```
tightOnMoney :-  
    student,  
    not richParents.
```

```
caresPracticality :-  
    likesSports.
```

```
% The patient (Peter)
```

```
shortSighted.  
afraidToTouchEyes.  
student.  
likesSports.
```

```
% Query
```

```
?- intraocularLens.
```


Getting Started: Ophthalmologist diagnosis [Schulz and Toni, 2016]

- Justifying recommended treatments increases trust.

```
% Justification for first answer
```

```
intraocularLens :-
    correctiveLens :-
        shortSighted,
        not laserSurgery :-
            tightOnMoney :-
                student,
                not richParents.
    not glasses :-
        caresPracticality :-
            likesSports.
    not contactLens :-
        afraidToTouchEyes.
```

```
% Treatments
```

```
intraocularLens :-
    correctiveLens,
    not glasses,
    not contactLens.
laserSurgery :-
    shortSighted,
    not tightOnMoney,
    not correctiveLens.
glasses :-
    correctiveLens,
    not caresPracticality,
    not contactLens.
contactLens :-
    correctiveLens,
    not afraidToTouchEyes,
    not longSighted,
    not glasses.
```

```
% Auxiliar
```

```
correctiveLens :-
    shortSighted,
    not laserSurgery.
tightOnMoney :-
    student,
    not richParents.
caresPracticality :-
    likesSports.
```

```
% The patient (Peter)
```

```
shortSighted.
afraidToTouchEyes.
student.
likesSports.
```

```
% Query
```

```
?- intraocularLens.
```

Getting Started: Ophthalmologist diagnosis [Schulz and Toni, 2016]

- Justifying recommended treatments increases trust.

```
% Justification for second answer
```

```
intraocularLens :-
    correctiveLens :-
        shortSighted,
        not laserSurgery :-
            chs(correctiveLens).
    not glasses :-
        caresPracticality :-
            likesSports.
    not contactLens :-
        afraidToTouchEyes.
```

```
% Treatments
```

```
intraocularLens :-
    correctiveLens,
    not glasses,
    not contactLens.
laserSurgery :-
    shortSighted,
    not tightOnMoney,
    not correctiveLens.
glasses :-
    correctiveLens,
    not caresPracticality,
    not contactLens.
contactLens :-
    correctiveLens,
    not afraidToTouchEyes,
    not longSighted,
    not glasses.
```

```
% Auxiliar
```

```
correctiveLens :-
    shortSighted,
    not laserSurgery.
tightOnMoney :-
    student,
    not richParents.
caresPracticality :-
    likesSports.
```

```
% The patient (Peter)
```

```
shortSighted.
afraidToTouchEyes.
student.
likesSports.
```

```
% Query
```

```
?- intraocularLens.
```

Foundations of Constraint ASP without Grounding

ASP + Constraints - Grounding = s(CASP)

- s(CASP) evaluates ASP programs with constraints:
 - Follows a top-down execution strategy based on s(ASP) [Marple et al., 2017b].
 - Constructive negation, `not p(X)`, is resolved against the dual of `p(X)`.

```

p(0) :- s.
p(X) :- q(X,Y).

not p(X) :- not p1(X), not p2(X).
not p1(X) :- X\=0.
not p1(X) :- X=0, not s.
not p2(X) :- c_forall(Y, not q(X,Y)).

```

- A new `clp(≠)` solver handles (partially) the negation of the unification.
- A new `c_forall/2` predicate computes the universal quantifier.
- To ensure consistency the *extended* compiler synthesizes NMR-check.

$$p(X) :- q(X,Y), \text{not } p(X). \quad \forall x(\text{chk}_i(x) \longleftrightarrow \forall y(\neg q(x,y) \vee p(x)))$$
- Facilitates the integration of different constraint domains, e.g., `clp(Q)` and `clp(R)`.

Foundations of Constraint ASP without Grounding (cont.)

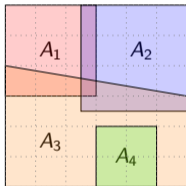
Internals: Universal quantifier

We execute `c_forall(X,goal(X))` to determine if `goal(X)` is true
...for all possible values of `X` in its constraint domain.

Foundations of Constraint ASP without Grounding (cont.)

Internals: Universal quantifier

We execute `c_forall(X,goal(X))` to determine if `goal(X)` is true
...for all possible values of `X` in its constraint domain.



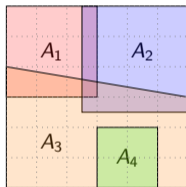
answers

`goal(X){T}`

Foundations of Constraint ASP without Grounding (cont.)

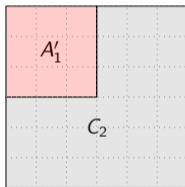
Internals: Universal quantifier

We execute `c_forall(X,goal(X))` to determine if `goal(X)` is true
 ...for all possible values of `X` in its constraint domain.



answers

`goal(X){T}`



(a)

$$A'_1 = A_1 \wedge T$$

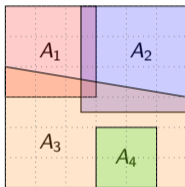
$$C_2 = T \wedge \neg A'_1$$

`goal(X){C2}`

Foundations of Constraint ASP without Grounding (cont.)

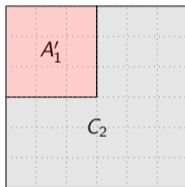
Internals: Universal quantifier

We execute `c_forall(X,goal(X))` to determine if `goal(X)` is true
 ...for all possible values of `X` in its constraint domain.



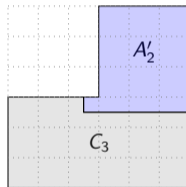
answers

`goal(X){T}`



(a)

$A'_1 = A_1 \wedge T$
 $C_2 = T \wedge \neg A'_1$
`goal(X){C2}`



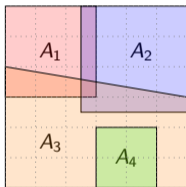
(b)

$A'_2 = A_2 \wedge C_2$
 $C_3 = C_2 \wedge \neg A'_2$
`goal(X){C3}`

Foundations of Constraint ASP without Grounding (cont.)

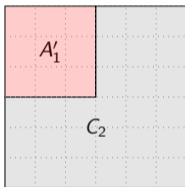
Internals: Universal quantifier

We execute `c_forall(X,goal(X))` to determine if `goal(X)` is true
 ...for all possible values of `X` in its constraint domain.



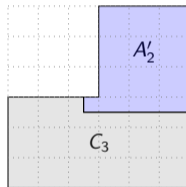
answers

`goal(X){T}`



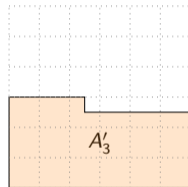
(a)

$A'_1 = A_1 \wedge \top$
 $C_2 = \top \wedge \neg A'_1$
`goal(X){C2}`



(b)

$A'_2 = A_2 \wedge C_2$
 $C_3 = C_2 \wedge \neg A'_2$
`goal(X){C3}`



(c)

$A'_3 = A_3 \wedge C_3$
 $C_4 = C_3 \wedge \neg A'_3 = \emptyset$
 End

The *C-forall* evaluation succeeds.

Foundations of Constraint ASP without Grounding (cont.)

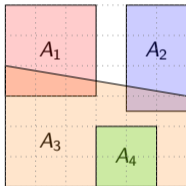
Internals: Universal quantifier (cont.)

If the set of answers of `goal(X)` is constraint-compact the algorithm finishes.
...this happens also when the answers do not cover the domain of `X`.

Foundations of Constraint ASP without Grounding (cont.)

Internals: Universal quantifier (cont.)

If the set of answers of `goal(X)` is constraint-compact the algorithm finishes.
 ...this happens also when the answers do not cover the domain of `X`.



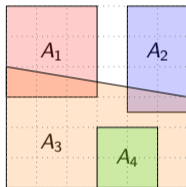
answers

`goal(X){T}`

Foundations of Constraint ASP without Grounding (cont.)

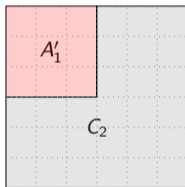
Internals: Universal quantifier (cont.)

If the set of answers of $\text{goal}(X)$ is constraint-compact the algorithm finishes.
 ...this happens also when the answers do not cover the domain of X .



answers

$\text{goal}(X)\{\top\}$



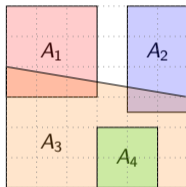
(a)

$A'_1 = A_1 \wedge \top$
 $C_2 = \top \wedge \neg A'_1$
 $\text{goal}(X)\{C_2\}$

Foundations of Constraint ASP without Grounding (cont.)

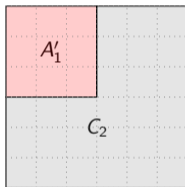
Internals: Universal quantifier (cont.)

If the set of answers of $\text{goal}(X)$ is constraint-compact the algorithm finishes.
 ...this happens also when the answers do not cover the domain of X .



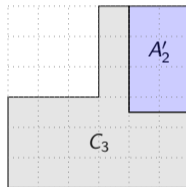
answers

$\text{goal}(X)\{\top\}$



(a)

$A'_1 = A_1 \wedge \top$
 $C_2 = \top \wedge \neg A'_1$
 $\text{goal}(X)\{C_2\}$



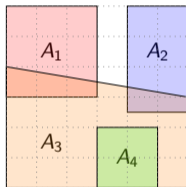
(b)

$A'_2 = A_2 \wedge C_2$
 $C_3 = \top \wedge \neg A'_2$
 $\text{goal}(X)\{C_3\}$

Foundations of Constraint ASP without Grounding (cont.)

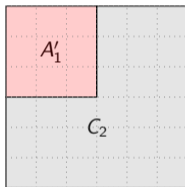
Internals: Universal quantifier (cont.)

If the set of answers of $\text{goal}(X)$ is constraint-compact the algorithm finishes.
 ...this happens also when the answers do not cover the domain of X .



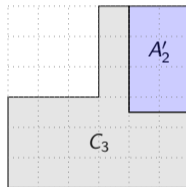
answers

$\text{goal}(X)\{\top\}$



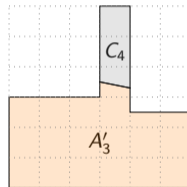
(a)

$A'_1 = A_1 \wedge \top$
 $C_2 = \top \wedge \neg A'_1$
 $\text{goal}(X)\{C_2\}$



(b)

$A'_2 = A_2 \wedge C_2$
 $C_3 = \top \wedge \neg A'_2$
 $\text{goal}(X)\{C_3\}$



(c)

$A'_3 = A_3 \wedge C_3$
 $C_4 = \top \wedge \neg A'_3$
 $\text{goal}(X)\{C_4\}$ fails

The *C-forall* evaluation fails.

Installation under Ciao and SWI

- The source code of s(CASP) is available at:
 - <https://gitlab.software.imdea.org/ciao-lang/sCASP> for Ciao.
 - <https://github.com/JanWielemaker/sCASP> for SWI-Prolog.
- As a **standalone** application: the installation creates `scasp`:

```
scasp --help_all
```

- **Playgrounds Online:** SWISH=<https://swish.swi-prolog.org/> and by Ciao=<https://ciao-lang.org/playground/scasp.html>.





DEMO

Bibliography I

- [1] J. Arias, Z. Chen, M. Carro, and G. Gupta (2019). **Modeling and Reasoning in Event Calculus Using Goal-Directed Constraint Answer Set Programming**. In: *Pre-Proc. of the 29th Int'l. Symposium on Logic-based Program Synthesis and Transformation*.
- [2] J. Arias, M. Moreno-Rebato, J. A. Rodríguez-García, and S. Ossowski (2023a). **Automated legal reasoning with discretion to act using s(LAW)**. In: *Artificial Intelligence and Law*, pp. 1–24. DOI: [10.1007/s10506-023-09376-5](https://doi.org/10.1007/s10506-023-09376-5).
- [3] J. Arias (2023). **Teach the importance of logic (programming) in Computer Science and why it is important**. In: *2023 International Symposium on Computers in Education (SIIE)*. IEEE, pp. 1–6.
- [4] J. Arias, M. Carro, Z. Chen, and G. Gupta (2020). **Justifications for Goal-Directed Constraint Answer Set Programming**. In: *Proceedings 36th International Conference on Logic Programming (Technical Communications)*. Vol. 325. EPTCS. Open Publishing Association, pp. 59–72. DOI: [10.4204/EPTCS.325.12](https://doi.org/10.4204/EPTCS.325.12).

Bibliography II

- [5] J. Arias, M. Carro, E. Salazar, K. Marple, and G. Gupta (2018). **Constraint Answer Set Programming without Grounding**. In: *Theory and Practice of Logic Programming* 18.3-4, pp. 337–354. DOI: [10.1017/S1471068418000285](https://doi.org/10.1017/S1471068418000285).
- [6] J. Arias, M. Moreno-Rebato, J. A. Rodríguez-García, and S. Ossowski (2021). **Modeling Administrative Discretion Using Goal-Directed Answer Set Programming**. In: *Advances in Artificial Intelligence, CAEPIA 20/21*. Vol. 12882. LNCS. Springer, pp. 258–267. DOI: [10.1007/978-3-030-85713-4_25](https://doi.org/10.1007/978-3-030-85713-4_25).
- [7] J. Arias, M. Moreno-Rebato, J. A. Rodríguez-García, and S. Ossowski (2024). **Towards value-awareness in administrative processes: an approach based on constraint answer set programming**. In: *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*, pp. 770–778.
- [8] J. Arias, I. Ramírez, and A. Gallinari (2023b). **DeducciónNatural. pl: herramienta escrita en Prolog para el aprendizaje de la asignatura de Lógica**. In: *Asociación de Enseñantes Universitarios de la Informática (AENUUI)*.

Bibliography III

- [9] J. Arias, S. Törmä, M. Carro, and G. Gupta (2022). **Building Information Modeling Using Constraint Logic Programming**. In: *Theory and Practice of Logic Programming* 22.5, pp. 723–738. DOI: [10.1017/S1471068422000138](https://doi.org/10.1017/S1471068422000138).
- [10] K. Basu, S. Varanasi, F. Shakerin, J. Arias, and G. Gupta (2021a). **Knowledge-driven Natural Language Understanding of English Text and its Applications**. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 14, pp. 12554–12563. URL: <https://ojs.aaai.org/index.php/AAAI/article/view/17488>.
- [11] K. Basu, H. Wang, N. Dominguez, X. Li, F. Li, S. C. Varanasi, and G. Gupta (2021b). **CASPR: A Commonsense Reasoning-based Conversational Socialbot**. In: *4th Proceedings of Alexa Prize (Alexa Prize 2021)*.
- [12] P. Cabalar, J. Fandinno, and M. Fink (2014). **Causal Graph Justifications of Logic Programs**. In: *Theory and Practice of Logic Programming* 14.4-5, pp. 603–618. DOI: [10.1017/S1471068414000234](https://doi.org/10.1017/S1471068414000234).

Bibliography IV

- [13] Z. Chen, K. Marple, E. Salazar, G. Gupta, and L. Tamil (2016). **A Physician Advisory System for Chronic Heart Failure Management Based on Knowledge Patterns**. In: *Theory and Practice of Logic Programming* 16.5-6, pp. 604–618. DOI: [10.1017/S1471068416000429](https://doi.org/10.1017/S1471068416000429).
- [14] L. Fidilio-Allende and J. Arias (2024). **f_{CASP} : A forgetting technique for XAI based on goal-directed constraint ASP models**. In: *Actas de las XXIII Jornadas de Programación y Lenguajes (PROLE 2024)*. Sistedes. URL: <https://hdl.handle.net/11705/PROLE/2024/13>.
- [15] G. García-Pradales, J. F. Morales, M. V. Hermenegildo, J. Arias, and M. Carro (2022). **Ans (CASP) In-Browser Playground based on Ciao Prolog**. In: *ICLP Workshops*.
- [16] M. Gelfond and V. Lifschitz (1988). **The Stable Model Semantics for Logic Programming**. In: *5th International Conference on Logic Programming*, pp. 1070–1080. DOI: [10.2307/2275201](https://doi.org/10.2307/2275201).

Bibliography V

- [17] B. Hall, S. C. Varanasi, J. Fiedor, J. Arias, K. Basu, F. Li, D. Bhatt, K. Driscoll, E. Salazar, and G. Gupta (2021). **Knowledge-Assisted Reasoning of Model-Augmented System Requirements with Event Calculus and Goal-Directed Answer Set Programming**. In: *Proc. 8th Workshop on Horn Clause Verification and Synthesis*.
- [18] K. Marple and G. Gupta (2012). **Galliwasp: A goal-directed answer set solver**. In: *International Symposium on Logic-Based Program Synthesis and Transformation*. Springer, pp. 122–136.
- [19] K. Marple, E. Salazar, Z. Chen, and G. Gupta (2017a). **The s(ASP) Predicate Answer Set Programming System**. In: *The Association for Logic Programming Newsletter*.
- [20] K. Marple, E. Salazar, and G. Gupta (2017b). **Computing Stable Models of Normal Logic Programs Without Grounding**. In: *arXiv 1709.00501*. arXiv: 1709.00501. URL: <http://arxiv.org/abs/1709.00501>.
- [21] J. Morris (2021). **Constraint answer set programming as a tool to improve legislative drafting: a rules as code experiment**. In: *ICAIL*. ACM, pp. 262–263.

Bibliography VI

- [22] S. Moyle, N. Allott, and J. Manslow (2023). **Modelling Cyber Defenses using s(CASP)**. In: *ICLP'23 Workshop on Goal-directed Execution of Answer Set Programs*.
- [23] E. Pontelli, T. C. Son, and O. El-Khatib (2009). **Justifications for Logic Programs under Answer Set Semantics**. In: *Theory and Practice of Logic Programming* 9.1, pp. 1–56. DOI: [10.1017/S1471068408003633](https://doi.org/10.1017/S1471068408003633).
- [24] C. Schulz and F. Toni (2016). **Justifying Answer Sets Using Argumentation**. In: *Theory and Practice of Logic Programming* 16.1, pp. 59–110. DOI: [10.1017/S1471068414000702](https://doi.org/10.1017/S1471068414000702).
- [25] F. Shakerin and G. Gupta (2019). **Induction of Non-Monotonic Logic Programs to Explain Boosted Tree Models Using LIME**. In: *AAAI 2019*, pp. 3052–3059. DOI: [10.1609/aaai.v33i01.33013052](https://doi.org/10.1609/aaai.v33i01.33013052).
- [26] J. Wielemaker, J. Arias, and G. Gupta (2021). **s(CASP) for SWI-Prolog**. In: *Proceedings of the 37th ICLP 2021 Workshops*. Vol. 2970. CEUR-WS.org. URL: <http://ceur-ws.org/Vol-2970/gdeinvited4.pdf>.