Tema 4: **Introducción a la Programación Lógica**

Asignatura: Lógica

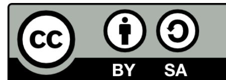
Joaquín Arias

Universidad Rey Juan Carlos
Escuela Técnica Superior de Ingeniería Informática

Curso 2022-2023

Índice I

Tema 4.1: **Estrategia de resolución SLD.**
Estrategia de resolución SLD.
Ejercicio
Árbol de derivación
Ejemplo



Copyright ©2022 Joaquín Arias. Esta obra, basada en trasparencias de Antonio González Pardo (URJC'20) & Pepa Hernández (UPM'11), está bajo la licencia CC BY-SA 4.0, Creative Commons Atribución-CompartirIgual 4.0 Internacional. Cómo citar esta obra: Arias, Joaquín (2022). *Lógica: desde Aristóteles hasta Prolog*. Madrid: Servicio de Publicaciones de la Universidad Rey Juan Carlos.
ISBN:978-84-09-38265-1

Índice II

Tema 4.2: **Prolog & la Programación Declarativa.**
- **Programación Declarativa**
- **Programación Funcional**
  - Cálculo Lambda
  - Haskell
  - Booleanos en cálculo lambda
  - Características y Ventajas
- **Programación Lógica**
  - Cláusulas Horn
  - Prolog
  - Hay algo más ...
  - ... mucho más
- **UTD HackReason**

Tema 4.1: **Estrategia de resolución SLD.**

Estrategia de resolución SLD.[1]

Estrategia de resolución SLD.[1]

- Se trata de la estrategia usada por el lenguaje de programación Prolog.
- Es un caso particular de la resolución general para cláusulas de Horn:
  - Las cláusulas objetivo no tiene literal afirmado.
  - Las cláusulas soporte tienen un literal afirmado (el primero).
- Dado un conjunto inicial de cláusulas de Horn $\{C_1, \ldots, C_i, \ldots, C_n\}$:
  - Existe una secuencia (*derivación*) $< C_i, C_{n+1}, \ldots, [] >$ tal que:
  - C_{n+1} es el resolvente de la cláusula objetivo C_i y una cláusula soporte.
  - C_k, con $k > n+1$, es el resolvente de C_{k-1} con una cláusula soporte.
  - Cada paso de resolución es de la forma $L \vee C, \neg L \vee C' \rightarrow C \vee C'$.
- Si y solo si el conjunto inicial es insatisfacible.

[1] SLD significa resolución **L**ineal con funciónn de **S**elección para cláusulas **D**efinidas.

Estrategia de resolución SLD: Ejercicio

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

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

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]).
```

Árbol de derivación SLD

Árbol de derivación SLD

- 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 son el cuerpo (resolvente) de las cláusulas cuya cabeza unifica con la consulta.
  3. Si el resolvente es [] la consulta tiene éxito:
     - 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.
     - Se crea un hijo por cada cláusula que unifica con dicho literal.
     - Se añade el cuerpo de la clausula al resolvente.
     - Se vuelve al punto 3.

Árbol de derivación SLD: Ejemplo

```
concatenar(La,Lb,[1,2,3,4])
├─ La=[], Lb=[1,2,3,4] → []
└─ La=[1|Xs] → concatenar(Xs,Lb,[2,3,4])
   ├─ Xs=[]  La=[1], Lb=[2,3,4] → []
   └─ Xs=[2|Xs'] → concatenar(Xs',Lb,[3,4])
      ├─ Xs'=[]  La=[1,2], Lb=[3,4] → []
      └─ Xs'=[3|Xs''] → concatenar(Xs'',Lb,[4])
         ├─ Xs''=[]  La=[1,2,3], Lb=[4] → []
         └─ X''=[4|Xs'''] → concatenar(Xs''',Lb,[])
            └─ X'''=[]  La=[1,2,3,4], Lb=[] → []
```

- Árbol de *derivación* de:

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

Tema 4.2: **Prolog & la Programación Declarativa.**

Programación Declarativa

Programación Declarativa

- ¿Que es la programación declarativa?
  - Paradigma de programación diferente a la imperativa (R) o la orientada a objetos (Java).
  - Los programas especifican las propiedades del problema a resolver.
  - La ejecución del programa consiste en "encontrar" la(s) solución(es).

Programación Declarativa

- ¿Que es la programación declarativa?
  - Paradigma de programación diferente a la imperativa (R) o la orientada a objetos (Java).
  - Los programas especifican las propiedades del problema a resolver.
  - La ejecución del programa consiste en "encontrar" la(s) solución(es).

Asignación Destructiva

```
1  def sumalista(lista):
2    sum = 0
3    for elem in lista:
4          sum += elem
5    return sum
6
7  print(sumalista([1,2,3,4]))
```

Recursión

```
1  sumaLista :: [Int] -> Int
2  sumaLista [] = 0
3  sumaLista (n:list) = n + (sumaLista list)
4
5
6
7  main = print (sumaLista [1,2,3,4])
```

Programación Declarativa (cont.)

Ejemplos de programación declarativa:

- Lenguajes funcionales: Haskell, Scala by EPFL.
- Lenguajes lógicos: Prolog, ASP, Logica by Google.
- Lenguajes algebraicos: Maude, SQL.
- etc...

Programación Funcional

Programación Funcional

- La programación funcional esta basada en funciones matemáticas.
- Función: Una función es una regla de correspondencia entre dos conjuntos de tal manera que a cada elemento del primer conjunto le corresponde uno y sólo un elemento del segundo conjunto.
- Cualquier función computable puede expresarse y evaluarse con el cálculo lambda.
- El cálculo lambda fue usado por Church para resolver el Entscheidungsproblem (1936):
  - No hay un algoritmo que determine si dos expresiones lambda arbitraria son equivalentes.

Programación Funcional: Cálculo Lambda

- Introducción al cálculo lambda.
- Reglas de formación de las expresiones lambda (λ-expresiones):
  - `x` es una λ-expresión si `x` es una variable.
  - `(`λ`x.t)` es una λ-expresión (función) si `t` una expresión y `x` una variable.
  - `(t s)` es una λ-expresión (aplicación) si `t` y `s` son expresiones.

Programación Funcional: Cálculo Lambda

- Introducción al cálculo lambda.
- Reglas de formación de las expresiones lambda (λ-expresiones):
  - `x` es una λ-expresión si `x` es una variable.
  - $(\lambda$`x.t`$)$ es una λ-expresión (función) si `t` una expresión y `x` una variable.
  - `(t s)` es una λ-expresión (aplicación) si `t` y `s` son expresiones.

Evaluando λ-expresiones

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

Programación Funcional: Haskell

en honor a Haskell Curry (1900-1982).

haskell.org

- *Currificación:* Transforma $f : (X_1 \times X_2 \times \cdots \times X_n) \to Z$ en una secuencia de funciones con un único argumento: $curry(f) : X_1 \to X_2 \to \cdots \to X_n \to Z$.

Ejemplos de funciones en Haskell

```
1  suma :: Int -> Int -> Int                        % suma a y b
2  suma a b = a + b                                 % version con dos argumentos
3  suma' = \a -> \b -> a + b                        % version currificada
4  sucesor :: Int -> Int                            % sucesor de a
5  sucesor = suma 1                                 % aplicación parcial
6  aplica :: (Int -> Int) -> [Int] -> [Int]         % aplica una función a una lista
7  aplica _ [] = []                                 % caso base
8  aplica f (n : list) = ((f n) : (aplica f list))  % caso recursivo
9
10 main = print (aplica sucesor [1,3,4])            % imprime [2,4,5]
```

Programación Funcional: Booleanos en cálculo lambda

- Definición de los booleanos en λ:
  - true: $\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{x}$
  - false: $\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{y}$
  - If-then-else: $\lambda \mathtt{x}.\lambda \mathtt{y}.\lambda \mathtt{z}.\mathtt{x\ y\ z}$

Programación Funcional: Booleanos en cálculo lambda

- Definición de los booleanos en λ:
  - true: $\lambda \texttt{x}.\lambda \texttt{y}.\texttt{x}$
  - false: $\lambda \texttt{x}.\lambda \texttt{y}.\texttt{y}$
  - If-then-else: $\lambda \texttt{x}.\lambda \texttt{y}.\lambda \texttt{z}.\texttt{x y z}$

Evaluación

If-then-else True P Q $\equiv$

Programación Funcional: Booleanos en cálculo lambda

- Definición de los booleanos en λ:
  - true: $\lambda \texttt{x}.\lambda \texttt{y}.\texttt{x}$
  - false: $\lambda \texttt{x}.\lambda \texttt{y}.\texttt{y}$
  - If-then-else: $\lambda \texttt{x}.\lambda \texttt{y}.\lambda \texttt{z}.\texttt{x y z}$

Evaluación

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

Programación Funcional: Booleanos en cálculo lambda

- Definición de los booleanos en λ:
  - true: $\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{x}$
  - false: $\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{y}$
  - If-then-else: $\lambda \mathtt{x}.\lambda \mathtt{y}.\lambda \mathtt{z}.\mathtt{x\ y\ z}$

Evaluación

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

- Implementación usando Haskell:

```
1  true = \x -> \y -> x
2  false = \x -> \y -> y
3  if_then_else = \x -> \y -> \z -> x y z
4
5  k = if_then_else true 3 2                % ¿cuánto vale k?
```

Programación Funcional: Booleanos en cálculo lambda (cont.)

- .. ahora, basadas en estas expresiones definimos And, Or y Not:
  - And: λp.λq.p q false $\equiv$ λp.λq.p q (λx.λy.y)
  - Or: λp.λq.p true q $\equiv$ λp.λq.p (λx.λy.x) q
  - Not: λp.p false true $\equiv$ λp.p (λx.λy.y) (λx.λy.x)

Programación Funcional: Booleanos en cálculo lambda (cont.)

- .. ahora, basadas en estas expresiones definimos And, Or y Not:
  - And: $\lambda p.\lambda q.p\ q\ \texttt{false} \equiv \lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y)$
  - Or: $\lambda p.\lambda q.p\ \texttt{true}\ q \equiv \lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q$
  - Not: $\lambda p.p\ \texttt{false}\ \texttt{true} \equiv \lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x)$

Evaluación

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 cálculo lambda (cont.)

- .. ahora, basadas en estas expresiones definimos And, Or y Not:
  - And: $\lambda p.\lambda q.p\ q\ \texttt{false} \equiv \lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y)$
  - Or: $\lambda p.\lambda q.p\ \texttt{true}\ q \equiv \lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q$
  - Not: $\lambda p.p\ \texttt{false}\ \texttt{true} \equiv \lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x)$

Evaluación

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

Programación Funcional: Booleanos en cálculo lambda (cont.)

- .. ahora, basadas en estas expresiones definimos And, Or y Not:
  - And: $\lambda p.\lambda q.p\ q\ \mathtt{false} \equiv \lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y)$
  - Or: $\lambda p.\lambda q.p\ \mathtt{true}\ q \equiv \lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q$
  - Not: $\lambda p.p\ \mathtt{false}\ \mathtt{true} \equiv \lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x)$

Evaluación

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 cálculo lambda (cont.)

- .. ahora, basadas en estas expresiones definimos And, Or y Not:
  - And: $\lambda p.\lambda q.p\ q\ \texttt{false} \equiv \lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y)$
  - Or: $\lambda p.\lambda q.p\ \texttt{true}\ q \equiv \lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q$
  - Not: $\lambda p.p\ \texttt{false}\ \texttt{true} \equiv \lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x)$

Evaluación

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

Programación Funcional: Booleanos en cálculo lambda (cont.)

- .. ahora, basadas en estas expresiones definimos And, Or y Not:
  - And: $\lambda p.\lambda q.p\ q\ \texttt{false} \equiv \lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y)$
  - Or: $\lambda p.\lambda q.p\ \texttt{true}\ q \equiv \lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q$
  - Not: $\lambda p.p\ \texttt{false}\ \texttt{true} \equiv \lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x)$

Evaluación

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 cálculo lambda (cont.)

- .. ahora, basadas en estas expresiones definimos And, Or y Not:
  - And: λp.λq.p q false $\equiv$ λp.λq.p q (λx.λy.y)
  - Or: λp.λq.p true q $\equiv$ λp.λq.p (λx.λy.x) q
  - Not: λp.p false true $\equiv$ λp.p (λx.λy.y) (λx.λy.x)

Evaluación

Not True $\equiv$ (λp.p (λx.λy.y) (λx.λy.x)) ($\lambda x_1.\lambda y_1.x_1$) $\equiv$ ... $\equiv$ (λx.λy.y) $\equiv$ False

Programación Funcional: Booleanos en cálculo lambda (cont.)

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

Evaluación

Not True $\equiv (\lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x))\ (\lambda x_1.\lambda y_1.x_1) \equiv \ \ldots \equiv (\lambda x.\lambda y.y) \equiv$ False

- Implementación usando Haskell (cont.):

```
6  my_and = \x -> \y -> x y false
7  my_or = \x -> \y -> x true y
8  my_not = \x -> x false true
9
10 k = if_then_else (my_and true false) 3 2   % ¿cuánto vale k?
```

Programación Funcional: Booleanos en cálculo lambda (cont.)

- **Alternativa** para And, Or y Not:
  - And: λp.λq.p q p
  - Or: λp.λq.p p q
  - Not: λp.λx.λy.p y x

Programación Funcional: Booleanos en cálculo lambda (cont.)

- **Alternativa** para And, Or y Not:
  - And: λp.λq.p q p
  - Or: λp.λq.p p q
  - Not: λp.λx.λy.p y x

¿Cuantos argumentos tiene?

Programación Funcional: Booleanos en cálculo lambda (cont.)

- **Alternativa** para And, Or y Not:
  - And: $\lambda p.\lambda q.p\ q\ p$
  - Or: $\lambda p.\lambda q.p\ p\ q$
  - Not: $\lambda p.\lambda x.\lambda y.p\ y\ x$

¿Cuantos argumentos tiene?

Evaluación

Sin hacer :-), deberes para casa

Programación Funcional: Booleanos en cálculo lambda (cont.)

- **Alternativa** para And, Or y Not:
  - And: $\lambda p.\lambda q.p\ q\ p$
  - Or: $\lambda p.\lambda q.p\ p\ q$
  - Not: $\lambda p.\lambda x.\lambda y.p\ y\ x$

¿Cuantos argumentos tiene?

- Implementación usando Haskell:

```
6  {-# LANGUAGE Rank2Types #-}
7  type CB = forall a . a -> a -> a        % Ojo, requiere tipos
8
9  my_and :: CB -> CB -> CB
10 my_and = \p -> \q -> p q p
11 my_or :: CB -> CB -> CB
12 my_or = \p -> \q -> p p q
13 my_not :: CB -> CB                    % tiene 1 argumento !!!
14 my_not = \p -> \x -> \y -> p y x
```

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

Programación Lógica

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

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

1 Se reescribe como el siguientes conjunto de cláusulas:

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

donde el consecuente, $Mortal(socrates)$, están negado.
2 Si es insatisfacible, significa que hay consecuencia lógica.
3 Se resuelve aplicando método de Robinson con estrategia SLD.

Programación Lógica: Cláusulas de Horn

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

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

Programación Lógica: Cláusulas de Horn

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

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

Aristóteles:

$$\frac{\begin{array}{l} Hombre(x) \rightarrow Mortal(x) \\ Hombre(socrates) \end{array}}{Mortal(socrates)}$$

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

SWI-Prolog

Ciao Prolog

Programación Lógica: Prolog.

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

SWI-Prolog

Ciao Prolog

Aristóteles

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

`?- mortal(socrates).`
Contesta `yes` si la “pregunta” es consecuencia lógica (`no` en caso contrario).

Programación Lógica: Prolog.

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

SWI-Prolog

Ciao Prolog

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:

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

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

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

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

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

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.
```

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

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.
```

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

“Una cosa más...”

“One More Thing...” is a reference to a practice that started in 1999, where Steve Jobs would leave (often quite big) announcements to the end of a presentation.

UTD HackReason 2021, 2022, . . . : January 14-15 (World Logic Day)

UTD HackReason 2021, 2022, . . . : January 14-15 (World Logic Day)

Ask to participate in the next edition