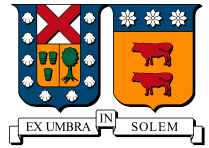
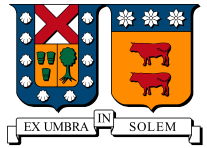


Programación Lógica y Lenguaje Prolog

Roberto Díaz
roberto.diazu@usm.cl

Universidad Técnica Federico Santa María
Departamento de Informática – Santiago, Chile

2021-2



Introducción a Prolog

Su nombre viene del francés *PROgrammation en LOGique* (Programación en Lógica) creado a comienzos de los ’70:

- Robert Kowalski (Edimburgo): lado teórico
- Maarten van Emden (Edimburgo): demostración práctica
- Alain Colmerauer (Marsella): Implementación

David Warren (Edimburgo), a mediados de los ’70 realizó la implementación eficiente del primer compilador de Prolog.

Prolog
Características

Basado en lógica simbólica y programación declarativa.

Produce estilo de programación orientado a metas. No se especifica cómo debe hacerse, sino qué debe lograrse (alto nivel).

El programador se concentra más en el conocimiento que en los algoritmos

- ¿Qué es conocido? (hechos y relaciones, y reglas)
- ¿Qué preguntar? (cómo resolverlo)

Pruebas Matemáticas
Demostración de Teoremas

Inteligencia Artificial
Sistemas Expertos

Consultas a bases de datos
Permite inferir relaciones no especificadas a priori

Prolog
Ejemplo

Hechos

```
progenitor(maria, pedro).
progenitor(juan, pedro).
progenitor(juan, carola).
progenitor(pedro, ana).
progenitor(pedro, paty).
progenitor(paty, aldo).
```

Consultas

```
?- progenitor(pedro, ana).
   true.
?- progenitor(ana, paty).
   false.
?- progenitor(X, carola).
   X = juan.
?- progenitor(pedro, X).
   X = ana ;
   X = paty .
```

Prolog
Ejemplo

Hechos

```
progenitor(maria, pedro).
progenitor(juan, pedro).
progenitor(juan, carola).
progenitor(pedro, ana).
progenitor(pedro, paty).
progenitor(paty, aldo).
```

Consultas

Abuelo de `aldo`

```
?- progenitor(X,Y),
    progenitor(Y, aldo).
   X = pedro,
   Y = paty ;
   false.
```

Prolog
Ejemplo

Hechos

```
progenitor(maria, pedro).
progenitor(juan, pedro).
progenitor(juan, carola).
progenitor(pedro, ana).
progenitor(pedro, paty).
progenitor(paty, aldo).
```

Consultas

Nietos de `juan`

```
?- progenitor(juan,X),
    progenitor(X,Y).
   X = pedro,
   Y = ana ;
   X = pedro,
   Y = paty ;
   false.
```

Hechos

```
progenitor(maria, pedro).
progenitor(juan, pedro).
progenitor(juan, carola).
progenitor(pedro, ana).
progenitor(pedro, paty).
progenitor(paty, aldo).
```

Consultas

Preguntar si `ana` y `paty` tienen progenitores en común

```
?- progenitor(X,ana),
    progenitor(X,paty).
   X = pedro.
```

Reglas en Prolog

La relación $a \Rightarrow b$ se expresa en Prolog como `b :- a.`

Una cláusula de este tipo se denomina **regla**.

La **cabeza** (parte izquierda de `:-`) es la conclusión de la proposición definida en el cuerpo (parte derecha de `:-`)

Resolución Simple

La relación `hijo` corresponde a:

$$\forall X, Y : (X \text{ es progenitor de } Y) \Rightarrow (Y \text{ es hijo de } X)$$

Se expresa en Prolog como:

```
hijo(Y,X) :- progenitor(X,Y)
```

Ejemplo: la meta siguiente es evaluada como:

La meta: `hijo(paty, pedro)`
Se convierte en submeta: `progenitor(pedro, paty)`
Se busca este hecho: `true`

Reglas
Ejemplo

Hechos

```
progenitor(maria, pedro).
progenitor(juan, pedro).
progenitor(juan, carola).
progenitor(pedro, ana).
progenitor(pedro, paty).
progenitor(paty, aldo).
masculino(aldo).
masculino(juan).
masculino(pedro).
femenino(ana).
femenino(carola).
femenino(maria).
femenino(paty).

hermana(X, Y) :-
        progenitor(Z, X),
        progenitor(Z, Y),
        femenino(X).
```

Consultas

Reglas
Ejemplo

Hechos

```
progenitor(maria, pedro).
progenitor(juan, pedro).
progenitor(juan, carola).
progenitor(pedro, ana).
progenitor(pedro, paty).
progenitor(paty, aldo).
masculino(aldo).
masculino(juan).
masculino(pedro).
femenino(ana).
femenino(carola).
femenino(maria).
femenino(paty).

hermana(X, Y) :-
        progenitor(Z, X),
        progenitor(Z, Y),
        femenino(X).
```

Consultas

```
?- hermana(ana,paty).
   true.

?- hermana(X,paty).
   X = ana ;
   X = paty ;
   false.
```

Reglas
Ejemplo

Hechos

```
progenitor(maria, pedro).
progenitor(juan, pedro).
progenitor(juan, carola).
progenitor(pedro, ana).
progenitor(pedro, paty).
progenitor(paty, aldo).
masculino(aldo).
masculino(juan).
masculino(pedro).
femenino(ana).
femenino(carola).
femenino(maria).
femenino(paty).

hermana(X, Y) :-
        progenitor(Z, X),
        progenitor(Z, Y),
        femenino(X), X \== Y.
```

Consultas

```
?- hermana(ana,paty).
   true.

?- hermana(X,paty).
   X = ana ;
   false.
```

Reglas Recursivas

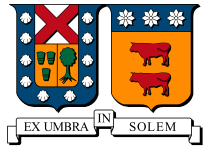
Hechos

```
progenitor(maria, pedro).
progenitor(juan, pedro).
progenitor(juan, carola).
progenitor(pedro, ana).
progenitor(pedro, paty).
progenitor(paty, aldo).


antepasado(X,Z) :-
        progenitor(X,Z).
antepasado(X,Z) :-
        progenitor(X,Y),
        antepasado(Y,Z).
```

Consultas

Descendientes de maria.

Reglas Recursivas



Hechos

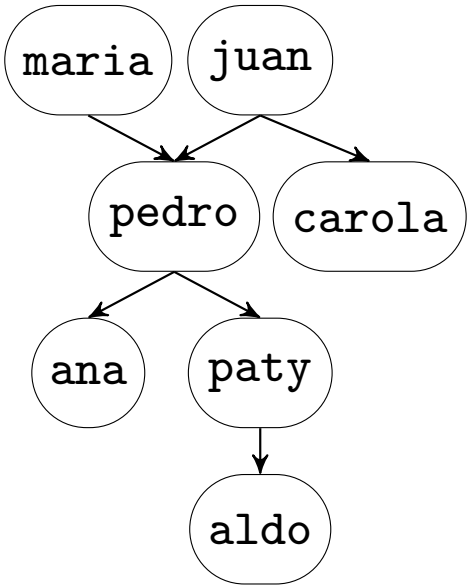
```
progenitor(maria, pedro).
progenitor(juan, pedro).
progenitor(juan, carola).
progenitor(pedro, ana).
progenitor(pedro, paty).
progenitor(paty, aldo).


antepasado(X,Z) :-
        progenitor(X,Z).
antepasado(X,Z) :-
        progenitor(X,Y),
        antepasado(Y,Z).
```

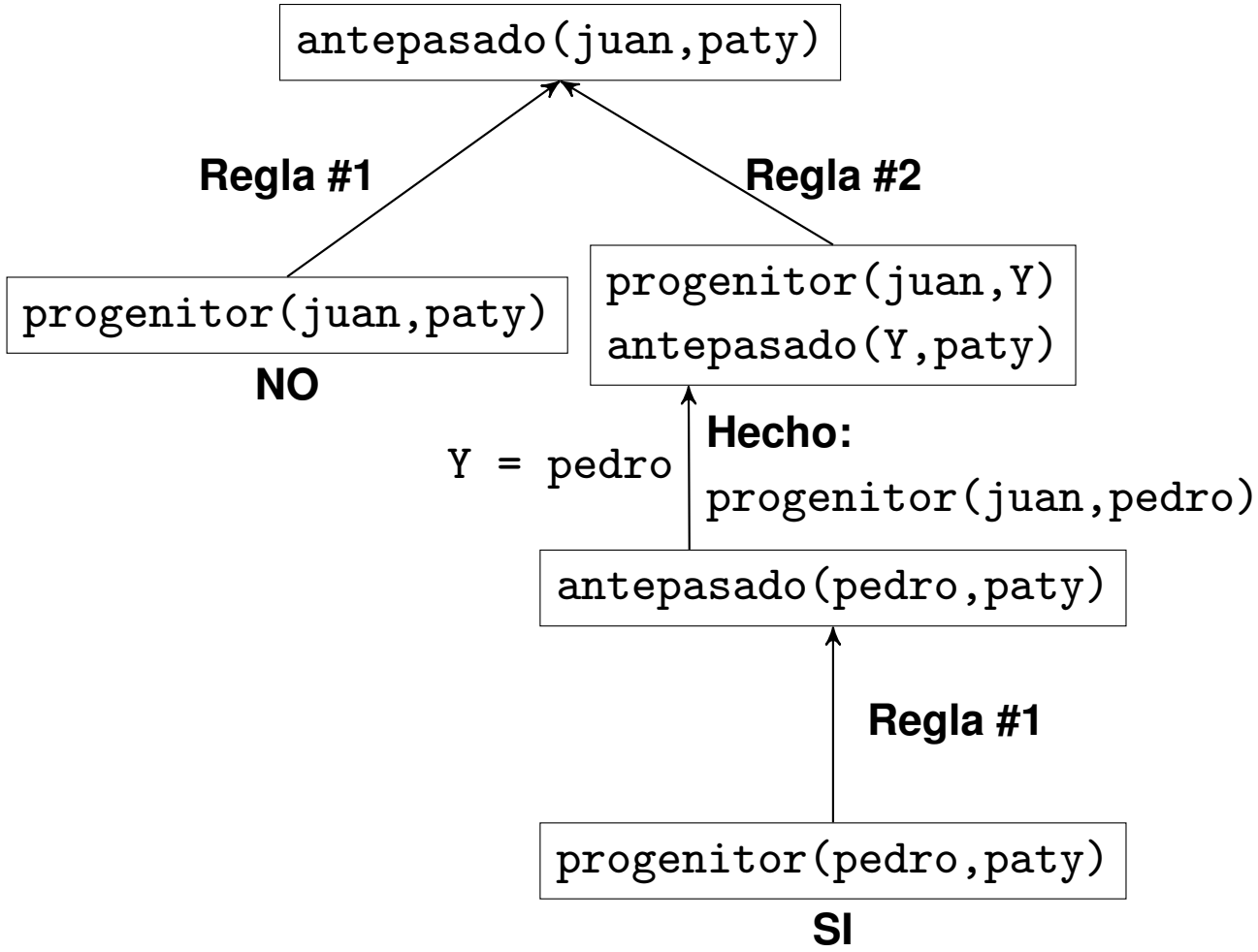
Consultas

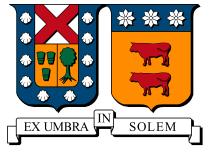
Descendientes de maria.

```
?- antepasado(maria,X).
   X = pedro ;
   X = ana ;
   X = paty ;
   X = aldo.
```

Resolución de Consulta







Tipos de Datos en Prolog

Términos

En Prolog los datos son representados por términos, los que se pueden clasificar como:

- Términos
  - Términos simples
    - Constantes: Átomos y Números
    - Variables
  - Términos compuestos

Se reconoce el tipo de un dato por su forma sintáctica. No se requiere de declaración de tipos.

Ejemplo:

- Variables comienzan con letra en mayúsculas (e.g. `X`)
- Átomos comienzan con una letra en minúscula (e.g. `pedro`)

Formación de Variables y Átomos

Strings de los siguientes carácteres:

- Letras mayúsculas `A...Z`
- Letras minúsculas `a...z`
- Dígitos `0...9`
- Caracteres especiales: `+ - * / < > = : . & _ ~`

Átomos

Los átomos pueden escribirse como simples strings

1. Strings de letras, dígitos y *underscore* (_), comenzando con minúscula.
   - `pedro`
   - `nil`
   - `x_25`
   - `algo_especial`
2. Strings de caracteres especiales
   - `<--->`
   - `===>`
   - `...`
3. Strings con citación simple
   - `'Juan'`
   - `'San Francisco'`

Números

Prolog tiene una representación para los números. Sin embargo, dado que Prolog es principalmente un lenguaje de computación simbólica, los números no son su fuerte (el entero es lo que más se usa).

- Enteros
  - `1`
  - `3213`
  - `0`
  - `-323`
- Reales
  - `3.14`
  - `-0.0234`
  - `100.2`

Variables

Una variable de Prolog puede representar cualquier cosa: un número, un nombre, una estructura, etc.

Se escriben como un string de letras, dígitos y *underscore*, comenzando con mayúscula o *underscore*.

- `X`
- `Resultado`
- `_X1`
- `_12`

Si una variable aparece una solo vez en una cláusula, se puede usar variables anónima con *underscore*.

```
?- progenitor(juan,_).
   true            %no se imprime variable
```

Ámbito de variable es una cláusula.

Términos compuestos

Son términos que tienen varias componentes. Son tratadas como un único término.

Se construyen usando un *functor*:

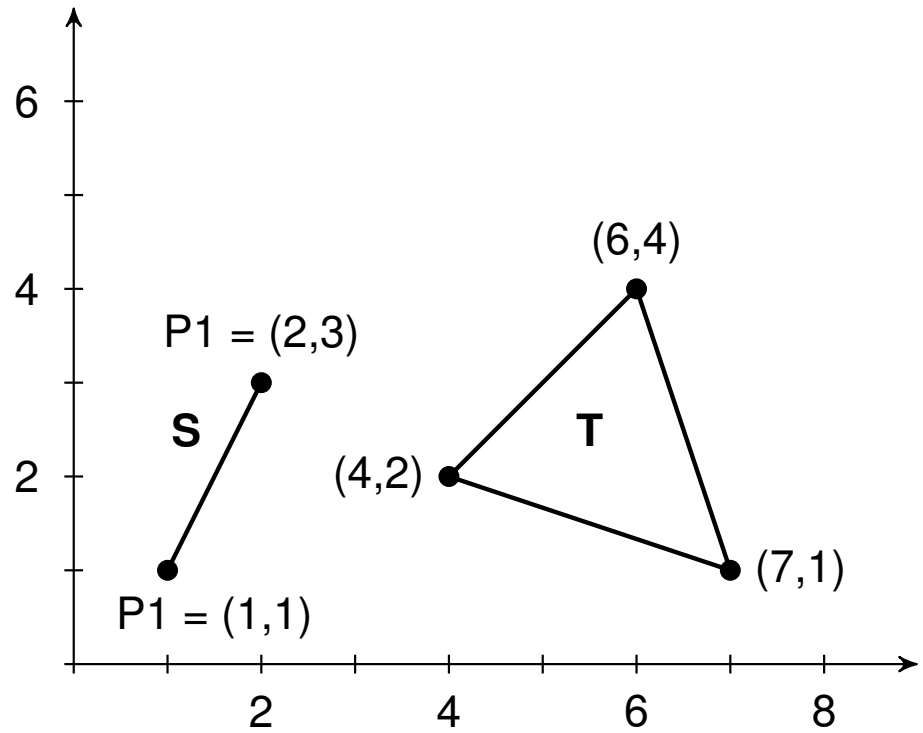
```
fecha(22, mayo, 2000)
```

Componentes pueden ser constantes, variables o estructuras.

```
fecha(Dia, mayo, 2000)
```

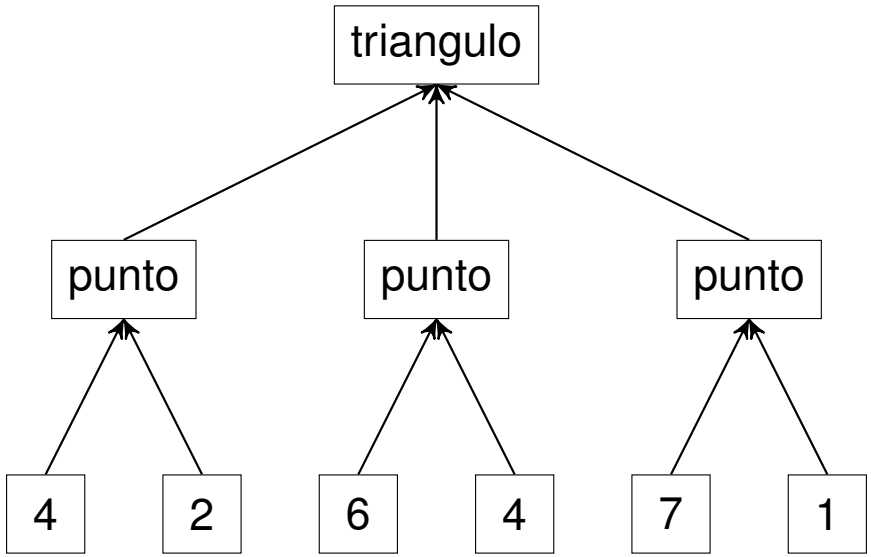
Ejemplo
Figuras Geométricas

```
P1 = punto(1,1),
P2 = punto(2,3),
S = segmento(P1,P2),
T = triangulo(punto(4,2), punto(6,4), punto(7,1)).
```



Representación de Árbol de Estructuras

```
T = triangulo(punto(4,2),punto(6,4),punto(7,1)).
```



Calce (*Matching*)

La operación más importante sobre términos es el **calce**, que corresponde a la **unificación** en el cálculo de predicados.

Dos términos calzan si:

- Son idénticos
- Las variables en ambos términos pueden ser instanciados, sustituyendo variables, tal que los términos se hacen idénticos

Calce
Ejemplo

Calzar `fecha(D,M,2000)` y `fecha(D1,mayo,A1)`, entonces:

- `D` se instancia a `D1`
- `M` se instancia a `mayo`
- `A1` se instancia a 2000

Que como salida de Prolog se escribe:

- `D = D1`
- `M = mayo`
- `A1 = 2000`

Al intentar calzar `fecha(D, M, 2000) = fecha(D1, julio, 1956)` no existe respuesta (salida es `false`).

Se tiene:

- No es posible encontrar un calce (se dice que el proceso de calce ha fracasado).
- En caso contrario, se dice que el proceso ha sido exitoso.

Calce
Ejemplo

```
?- fecha(D, M, 2000) = fecha(D1, mayo, A1).
D = D1,
M = mayo,
A1 = 2000.

?- fecha(D, M, 2000) = fecha(D1, julio, 1956).
false
```

Dos términos S y T calzan, si:

- Si S y T son constantes, entonces S y T calzan si ambos son el mismo objeto.
- Si S es una variable y T cualquier cosa, entonces calzan y S se instancia como T. Viceversa, si T es variable, entonces T se instancia como S.
- Si S y T son estructuras, entonces calzan sólo si:
  - S y T tienen el mismo *functor*, y
  - Todas sus correspondientes componentes calzan. Instanciaciones resultantes son determinadas por proceso de calce de componentes.

Calce de Estructuras
Ejemplo

```
?- triangulo(punto(1, 1), A, punto(2, 3)) = triangulo(X,
    punto(4, Y), punto(2, Z)).
  A = punto(4,Y),
  X = punto(1,1),
  Z = 3.
```

Calce de Estructuras
Ejemplo

Hechos:

```
vertical(seg(punto(X, Y),
    punto(X, Y1))).
horizontal(seg(punto(X, Y),
    punto(X1, Y))).
```

Consultas:

```
?- vertical(seg(punto(1,1),
    punto(1,2))).
  yes
?- vertical(seg(punto(1,1),
    punto(2,Y))).
  no
?- horizontal(seg(punto(1,1),
    punto(2,Y))).
  Y = 1
?- vertical(seg(punto(2,3), Y)
    ).
  Y = punto(2,_G561)
?- vertical(S), horizontal(S).
  S = seg(punto(_G576,_G577),
      punto(_G576,_G577))
```

Declarativo vs Procedural

La cláusula: `P :- Q, R.`
Se **interpreta declarativamente** como:

- `P` es verdadero si `Q` y `R` lo son.
- o, de `Q` y `R` se deriva `P`.

En cambio, una **interpretación procedural** sería:

- Para resolver `P`, primero se debe resolver `Q` y luego `R`.
- Para satisfacer a `P`, primero se debe satisfacer `Q` y luego `R`.

Declarativo

Una meta `G` es verdadera (satisface o se deriva lógicamente de un programa), si y solo si:

- Existe en el programa una cláusula `C`, y
- existe una cláusula `I`, instancia de `C`, tal que:
  - La cabeza de `I` es idéntica a `G`, y
  - todas las metas en el cuerpo de `I` son verdaderas.

Disjunción en Cláusulas

La cláusula: `P :- Q; R.` se interpreta como:

```
P :- Q.
P :- R.
```

La cláusula: `P :- Q, R; S, T, U.`

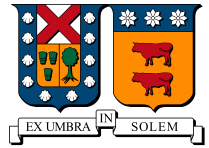
```
P :- Q, R.
P :- S, T, U.
```

Significado Procedural

Especifica *cómo* responder a una pregunta.

Para obtener la respuesta es necesario satisfacer una lista de metas.

Las metas pueden ser satisfechas si a través de la instanciación de sus variables se permite que del programa se deriven las metas.



Listas y Operadores

Listas

Una lista en Prolog se escribe con sus elementos entre corchetes

```
[perro, gato, raton, loro]
```

Sin embargo esto es sólo un *sabor sintáctico*, pues Prolog lo traduce a una forma de estructura.

Si existe una estrucutra del tipo `.(Cabeza, Cola)`, entonces la lista anterior (más legible) equivale a:

```
.(perro, .(gato, .(raton, .(loro, [] ))))
```

Representación de Listas

Una lista define un árbol binario, similar a las listas propias de Scheme.

Prolog permite una notación similar a los pares:

- `L = [a | Cola]`, donde `a` es la cabeza (cualquier tipo) y `Cola` es el resto de la lista (debe ser una lista) .
- La lista vacía se expresa como `[]`.

Ejemplo:

```
?- L2 = [a | [b | []]].
  L2 = [a, b].
```

Listas
Operadores

- Membresía del objeto `X` en la lista `L`:

```
member(X, L)
```

- Concatenación de listas `L1` y `L2` en `L3`:

```
append(L1, L2, L3)
```

- Borrar un elemento `X` en una lista `L`:

```
delete(L, X, L1)
```

Listas
Ejemplo de Operadores

```
?- member(X, [a, b]).
  X = a ;
  X = b.
```

```
?- append([a], [b], L).
  L = [a,b].
```

```
?- delete([a, b, c, b, d], b, L).
  L = [a,c,d].
```

Listas
Ejemplos de Uso - Sublista

Se puede definir un operador para sublista con la siguiente regla:

```
sublist(S,L) :- append(L1, L2, L), append(S, L3, L2).
```

Ejemplo de uso:

```
?- sublist([b,X], [a,b,c,d])
  X = c.
```

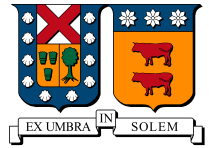
Recursión sobre listas

El cálculo del largo de una lista se puede implementar así:

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

Ejemplo de uso:

```
?- largo([a,b,c,d,e],L).
  L = 5.
```



Operadores y Aritmética

Notación de Operadores

Las operaciones en Prolog se expresan normalmente como *functores*.

Se permite también especificar operadores especiales con notación prefija, infija, post-fija con su relación de precedencia mediante directivas al traductor Prolog.

Este mecanismo permite mejorar la lectura de programas (sabor sintáctico), con mecanismo similar a la sobrecarga de operadores en C++.

Operadores
Ejemplo

Operadores como *functores*.

```
?- X = +(*(2, 3), *(4, 5)).
   X = 2*3+4*5

?- X is +(*(2, 3), *(4, 5)).
   X = 26.
```

Operadores en notación infija

```
?- X = 2*3 + 4*5.
   X = 2*3+4*5.

?- X is 2*3 + 4*5.
   X = 26.
```

Existen reglas de asociatividad y precedencia

```
?- 2+3+4 = +(+(2,3),4).
   true.
?- 2+3+4 = +(2,+(3,4)).
   false.
```

Operadores
Equivalencia

El operador = en Prolog es usado para el calce, pero no lleva a cabo una asignación numérica ni una evaluación aritmética.

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

?- 2+1 = 1+2.
   false.

?- 3 = 1+2.
   false.

?- X = 1+2.
   X = 1+2.
```

Operadores
Equivalencia

El operador `is` compara un número con una expresión. Retorna verdadero si es que el número a su izquierda es equivalente a la expresión a su derecha. Debe ser usado normalmente con una variable a su izquierda.

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

Operadores
Equivalencia

El operador `is` compara un número con una expresión. Retorna verdadero si es que el número a su izquierda es equivalente a la expresión a su derecha. Debe ser usado normalmente con una variable a su izquierda.

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

De hecho, puede fallar en ciertos casos como cuando los tipos numericos difieren.

```
?- 1 is sin(pi/2).
  false.
```

Operadores
Equivalencia

Para probar igualdad se debe usar el operador =:= que prueba si ambas expresiones se evalúan al mismo número.

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

?- 1 =:= sin(pi/2).
   true.
```

Para probar desigualdad existe el operador =\=

```
?- 1+2 =\= 2+1.
   false.

?- 1+2 =\= 10+1.
   true.
```

Operadores
Definición

Se permite definición de operadores prefijos, infijos y postfijos

A cada operador se le puede definir el nivel de precedencia mediante un valor (p.e. entre 1-1200 en una implementación)

Nombre del operador debe ser un átomo

Sintaxis:

```
:- op(Precedencia, Tipo, Nombre).
```

Ejemplo: Operador binario infijo `gusta`

```
:- op(600, xfx, gusta).
```

Operadores
Tipos

Dependiendo de la notación se tiene que los tipos pueden ser:

- **Operador Infijo**: `xfx`, `xfy`, `yfx`
- **Operador Prefijo**: `fx`, `fy`
- **Operador Postfijo**: `xf`, `yf`

La notación se interpreta como.

- `f` corresponde al nombre del operador
- `x` e `y` representan los argumentos
- `x` representa operando con precedencia estrictamente menor que el operador
- `y` representa operando cuya precedencia es menor o igual que el operador

Esta definición define la asociatividad de los operadores (`y` domina sobre `x`)

Operadores
Conjunto predefinido

```
:- op(1200, xfx, :-)
:- op(1200, fx,  [:-, ?-])
:- op(1100, xfy, ;)
:- op(1000, xfy, ,)
:- op(990,  xfx, :=)
:- op(700,  xfx, [<, =, =<, ==, =\=, >, >=, \==, is, =:=])
:- op(500,  yfx, [+, -, /\, \/, xor])
:- op(400,  yfx, [*, /, //, div, rdiv, <<, >>, mod, rem])
:- op(200,  xfx, **)
:- op(200,  xfy, ^)
:- op(200,  fy,  [+, -, \])
:- op(100,  yfx, .)
```

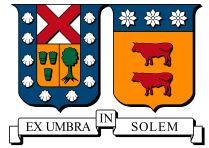
Definición de Operadores
Ejemplo

Hechos

```
mas_grande(perro,hormiga).
mas_grande(elefante,hormiga).
mas_grande(elefante,perro).
mas_grande(luna, hormiga).
mas_grande(luna, perro).
mas_grande(luna, elefante).

:- op(600,xfx,mas_grande).
```

Consultas

```
?- mas_grande(perro, hormiga).
  true.

?- perro mas_grande hormiga.
  true.

?- luna mas_grande X.
  X = hormiga ;
  X = perro ;
  X = elefante.
```



Control de Backtracking

Control de Backtracking
Razones

Prolog realiza `backtracking` automático si falla la satisfacción de una cláusula.

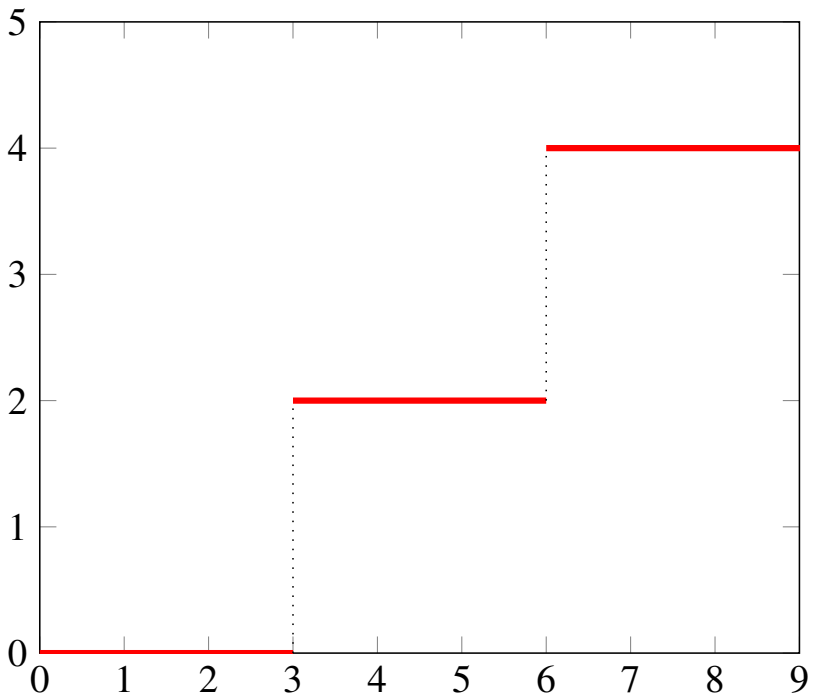
Sin embargo, en algunos casos el `backtracking` automático es ineficiente.

El programador puede controlar o prevenir el `backtracking` usando `cut`.

Ejemplo Función

Suponga las siguientes tres reglas para una función doble escalón:

- **Regla 1**: $(X < 3) \Rightarrow Y = 0$
- **Regla 2**: $(X \geq 3 \wedge X < 6) \Rightarrow Y = 2$
- **Regla 3**: $(X \geq 6) \Rightarrow Y = 4$



Ejemplo Función

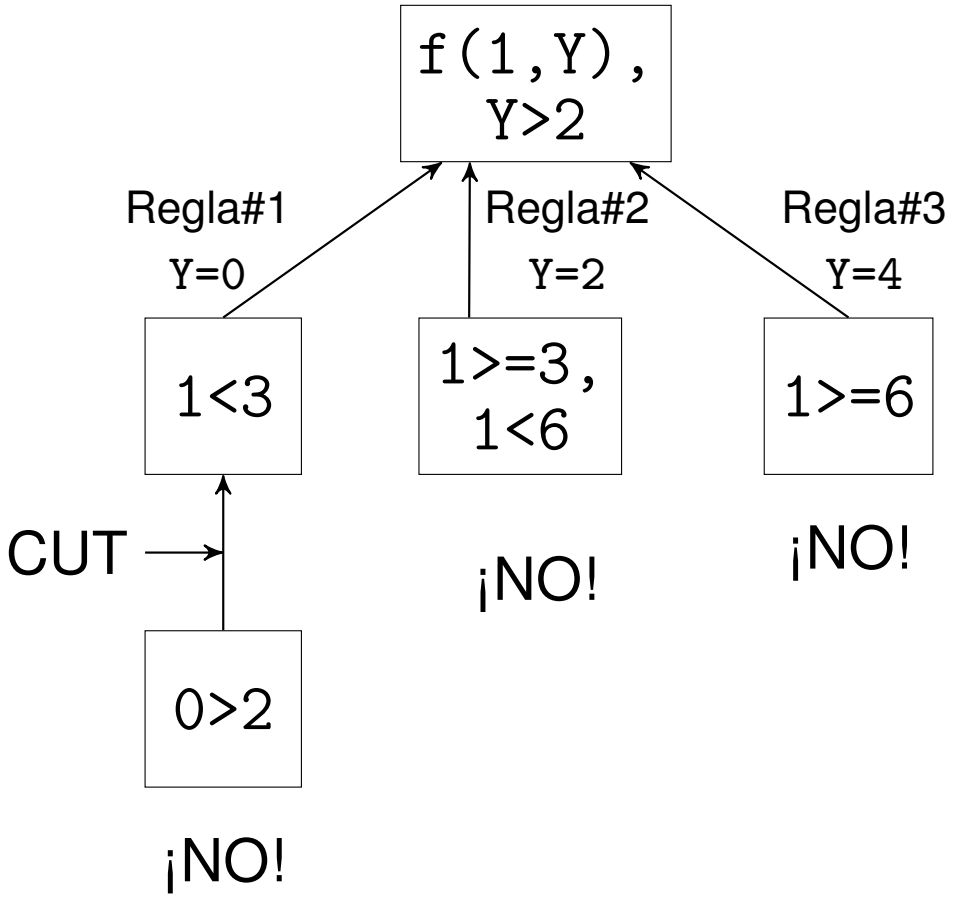
Hechos

```
% regla #1
f(X, 0) :- X < 3.
% regla #2
f(X, 2) :- X >= 3, X < 6.
% regla #3
f(X, 4) :- X >= 6.
```

Consultas

```
-? f(1, Y), Y > 2.
  false.
```

Evaluación de la meta



Ejemplo Función
Usando Cut

Hechos

```
% regla #1
f(X, 0) :- X < 3, !.
% regla #2
f(X, 2) :- X >= 3, X < 6, !.
% regla #3
f(X, 4) :- X >= 6, !.
```

Consultas

```
-? f(1, Y), Y > 2.
  false.
```

Observaciones

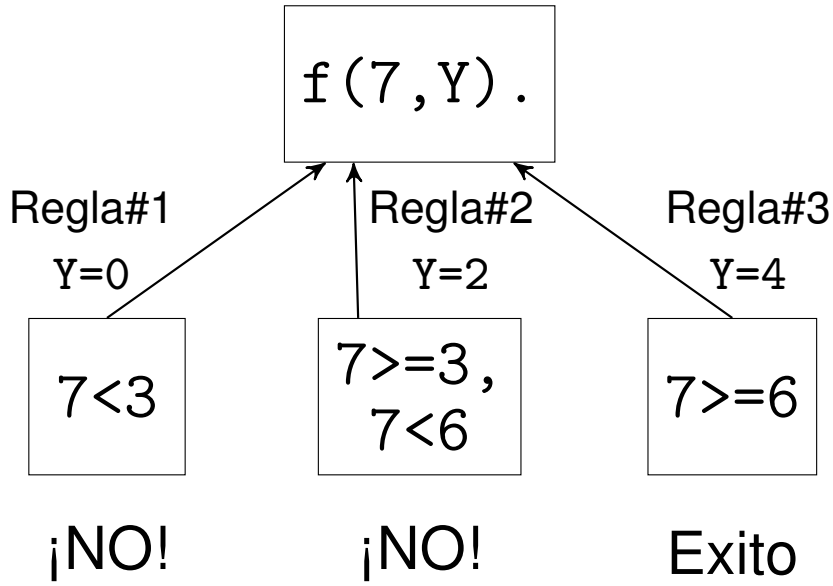
Ambas definiciones arrojan el mismo resultado.

Sin embargo, la segunda versión con *cut* es **más eficiente** dado que reconoce antes que la evaluación ha fallado.

Se puede decir que el *cut* ha modificado el **significado procedural** del programa y que en este caso no ha sido modificado el **significado declarativo**.

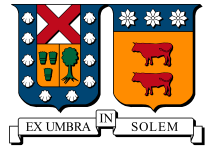
Evaluación de la meta

```
-? f(7, Y).
   Y=4.
```



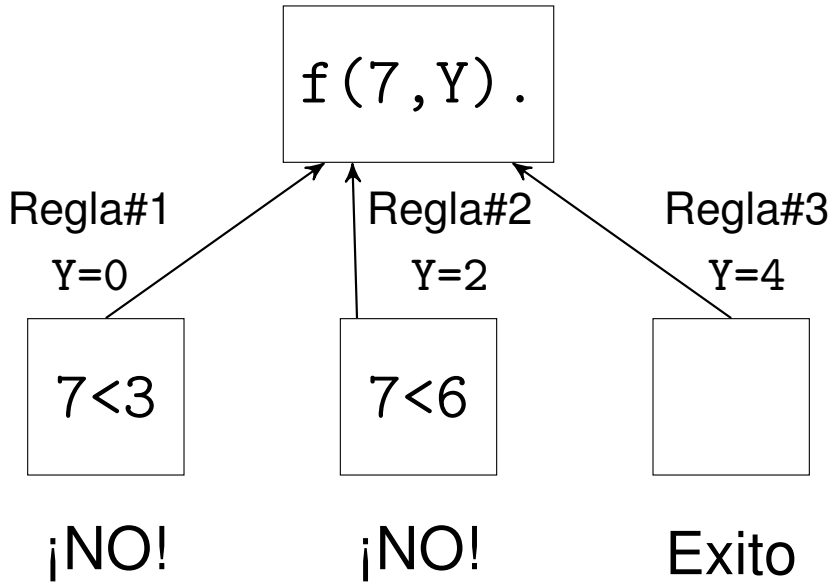
No se alcanza el CUT en ninguna evaluación.

Optimización evaluación de función usando Cut



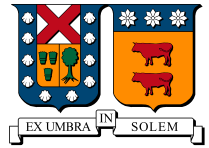

```
f(X, 0) :- X < 3, !. % regla #1

f(X, 2) :- X < 6, !. % regla #2

f(X, 4) . % regla #3


-? f(7, Y).
Y=4
```

Evaluación de la meta



Se reduce el número de evaluaciones.

Observación



Esta optimización sólo funciona con Cut.
Si se sacaran los Cuts y se evalúa `f(1, Y)` produciría varios resultados.
En este caso se dice que afecta el significado declarativo.

Negación como falla

A María le gustan los animales...

```
gusta(maria,X) :- animal(X).
```

...pero no le gustan las serpientes

```
gusta(maria,X) :- serpiente(X), !, fail.
gusta(maria,X) :- animal(X).
```

Negación
Operador

El procedimiento interno `not` de Prolog se comporta como:

```
not(P) :- P, !, fail;
          true.
```

Una forma equivalente es con el operador prefijo \+:

```
\+(P) :- P, !, fail;
         true.
```

Negación
Aplicación

Hay dos formas de escribir la regla sobre los gustos de Maria:

```
gusta(maria,X) :- \+ serpiente(X), animal(X).
```

```
gusta(maria,X) :- animal(X), \+ serpiente(X).
```

Estas dos formas aunque parecen iguales, proceduralmente no lo son. Por ejemplo si se hace la consulta `gusta(maria,X)`:

- La primera falla apenas encuentra X tal que serpiente(X) es verdadero, por lo que retorna falso.
- La segunda continua a pesar de encontrar X tal que serpiente(X) es verdadero, por lo que retorna los gustos de maria.

Uso de Cut y Negación

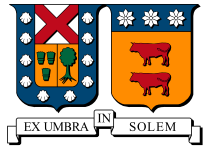
Ventajas:

- Se puede aumentar la eficiencia.
- Se pueden expresar reglas que son mutuamente excluyentes.

Desventajas:

- Se pierde correspondencia entre significado declarativo y procedural.
- Cambio del orden de las cláusulas puede afectar significado declarativo.

Referencias



- Capítulo XVI de [Sebesta, 2011]