

Elementos del lenguaje: hechos y reglas

- Un programa lógico está formado por **cláusulas** que pueden ser de dos tipos:
 - ▶ **Hechos:** $H : - \text{true}.$
Representan conocimiento que es *verdad* en nuestro programa.
(Notación: “: - true” se puede omitir, quedando: $H.$).
 - ▶ **Reglas:** $H : - B.$
Se leen de la siguiente forma: “ H es cierto **si** se cumple B ”.
 H se denomina *cabeza* y B se denomina *cuerpo*.
 - ▶ Ejemplo de relaciones familiares:

```
progenitor(pedro, juan) .  
hombre(pedro) .  
padre(X, Y) :- progenitor(X, Y), hombre(X) .
```
 - ▶ En el cuerpo de una regla pueden aparecer varios *literales* separados por comas. Las comas indican la *conjunción* de los literales del cuerpo de la regla.
- Se pueden incluir *variables lógicas*, que se identifican porque empiezan con una letra mayúscula o el símbolo de subrayado.

Elementos del lenguaje: predicados y objetivos

- En un programa lógico pueden existir varias cláusulas (hechos o reglas) para el mismo predicado:

```
hombre(pedro) .      progenitor(pedro,juan) .  
hombre(juan) .       progenitor(pedro,marta) .  
mujer(marta) .
```

- Un **predicado** queda definido por el conjunto de sus hechos y reglas.
 - ▶ Prolog permite utilizar el mismo nombre para dos predicados que difieren en su *aridad* (número de argumentos).
 - ▶ En el ejemplo anterior, `progenitor/2`, `hombre/1` y `mujer/1`.
 - ▶ Semántica declarativa de un predicado: las cláusulas (hechos y reglas) de un predicado forman **distintas alternativas** para que ese predicado sea cierto.
- Un **programa lógico** es un conjunto de definiciones de predicados.

Elementos del lenguaje: objetivos

- Para poder ejecutar un programa lógico, debemos formular una *consulta* al programa, denominada **objetivo**:

?- G.

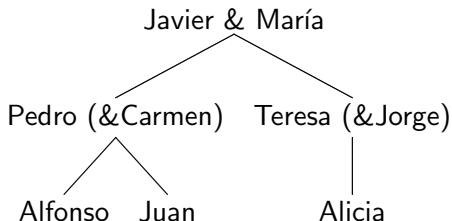
- La ejecución consiste en buscar la respuesta a ese objetivo.
- Ejemplos de objetivos:
 - ▶ ?- progenitor(pedro,arturo) .
¿Pedro es progenitor de Arturo?
 - ▶ Los objetivos pueden contener variables:
?- progenitor(pedro,X) .
¿De quién es Pedro progenitor?
?- padre(Y,marta) .
¿Quién es el padre de Marta?
 - ▶ Los objetivos pueden ser compuestos: son similares al cuerpo de una regla y se interpretan como la **conjunción** de sus literales.
?- padre(X,juan), padre(Y,X) .
¿Qué se pregunta en esta consulta?

Elementos del lenguaje: objetivos (cont.)

- La evaluación de un objetivo puede devolver dos resultados lógicos:
 - ▶ **Sí**: cuando el objetivo es consecuencia de las reglas del programa.
 - ▶ **No**: en caso contrario.
- Durante el **proceso de resolución** se encuentran además los **valores de las variables que satisfacen el objetivo** (que proporcionan una respuesta afirmativa al programa).

Ejemplo: Relaciones familiares

- Supongamos que tenemos las siguientes relaciones de parentesco



- ¿Cómo se representarían estas relaciones mediante hechos?

Ejemplo: Relaciones familiares

<code>hombre(javier).</code>	<code>progenitor(javier,pedro).</code>
<code>hombre(pedro).</code>	<code>progenitor(javier,teresa).</code>
<code>hombre(jorge).</code>	<code>progenitor(maria,pedro).</code>
<code>hombre(alfonso).</code>	<code>progenitor(maria,teresa).</code>
<code>hombre(juan).</code>	<code>progenitor(pedro,alfonso).</code>
<code>mujer(maria).</code>	<code>progenitor(pedro,juan).</code>
<code>mujer(carmen).</code>	<code>progenitor(carmen,juan).</code>
<code>mujer(teresa).</code>	<code>progenitor(carmen,alfonso).</code>
<code>mujer(alicia).</code>	<code>progenitor(jorge,alicia).</code>
	<code>progenitor(teresa,alicia).</code>

- Algunos predicados:

`padre(X,Y) :- progenitor(X,Y), hombre(X).`

`madre(X,Y) :- progenitor(X,Y), mujer(X).`

- El ámbito de las variables de un programa lógico es la cláusula donde aparecen: la variable `X` de `padre/2` es distinta de la variable `X` de `madre/2`.
- ¿Cómo se pueden representar las siguientes relaciones de parentesco: `hijo(X,Y)`, `abuelo(X,Y)`, `hermano(X,Y)`, `tio(X,Y)`, `descendiente(X,Y)`?

Ejemplo: Relaciones familiares

```
hijo(X,Y) :- progenitor(Y,X), hombre(X).  
abuelo(X,Y) :- progenitor(Z,Y), padre(X,Z).  
hermano(X,Y) :- progenitor(Z,X), progenitor(Z,Y).  
tio(X,Y) :- progenitor(Z,Y), hermano(Z,X).
```

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

- Podemos probar esto en SWI-Prolog:

```
$ swipl  
% /home/jcorreas/.plrc compiled 0.01 sec, 11,372 bytes  
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 5.6.64)  
...  
For help, use ?- help(Topic). or ?- apropos(Word).  
  
?- [ejemplo02].  
% ejemplo02 compiled 0.00 sec, 3,916 bytes  
true.
```

Ejemplo: Relaciones familiares (cont.)

- Consultas sobre el programa de relaciones familiares:

```
hijo(juan,pedro).  
abuelo(javier,teresa).  
hijo(javier,X).  
hijo(X,pedro).  
descendiente(X,javier).  
hijo(pedro,X).  
hermano(pedro,X).  % ¿Pedro es hermano de Pedro?
```

- Modificamos el predicado hermano/2:

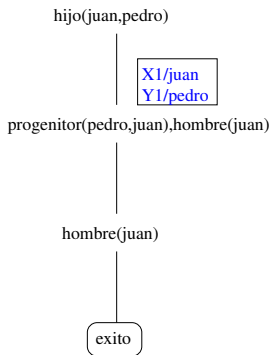
```
hermano(X,Y):-  
    progenitor(Z,X),  
    progenitor(Z,Y),  
    distinto(X,Y).  
  
distinto(X,Y):- X \= Y.  % \= está predefinido.
```

- Para obtener más de una solución: ;
- Para salir de SWI: halt.

Árboles de resolución (cont.)

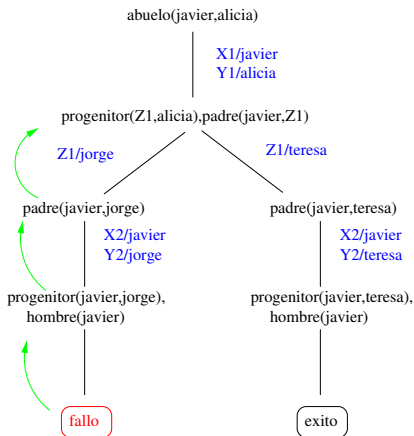
- Para resolver un objetivo sobre un programa lógico se utiliza el **mecanismo de resolución**.
- Aunque para definir correctamente el mecanismo de resolución es necesario tener en cuenta el concepto de unificación, vamos a ver cómo se aplica el mecanismo de resolución en los ejemplos anteriores.
- En el primer ejemplo:

- 1) Se aplica la regla
`hijo(X, Y) :-
 progenitor(Y, X), hombre(X) .`
- 2) Se generan nuevos objetivos. Se comprueba que existe el hecho `progenitor(pedro, juan) .`
- 3) Por último, se resuelve el último objetivo pendiente:
`hombre(juan) .`

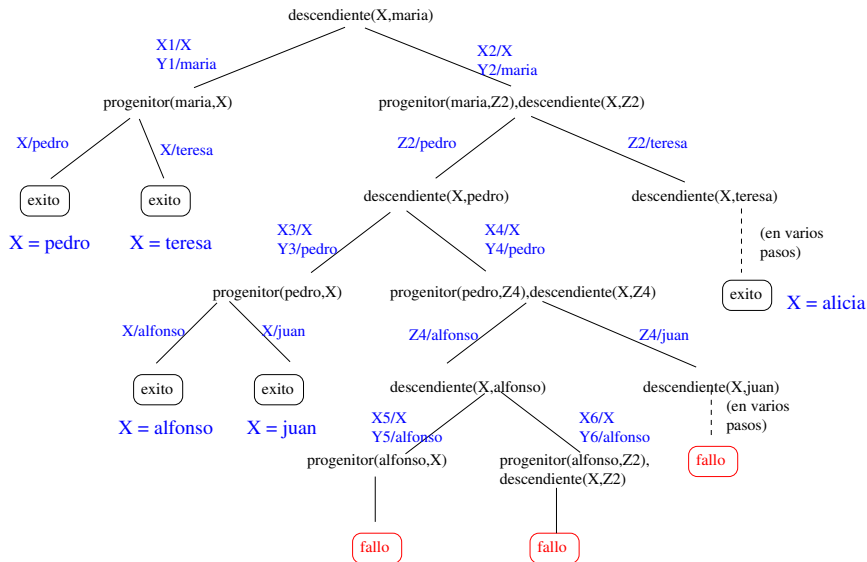


Árboles de resolución (cont.)

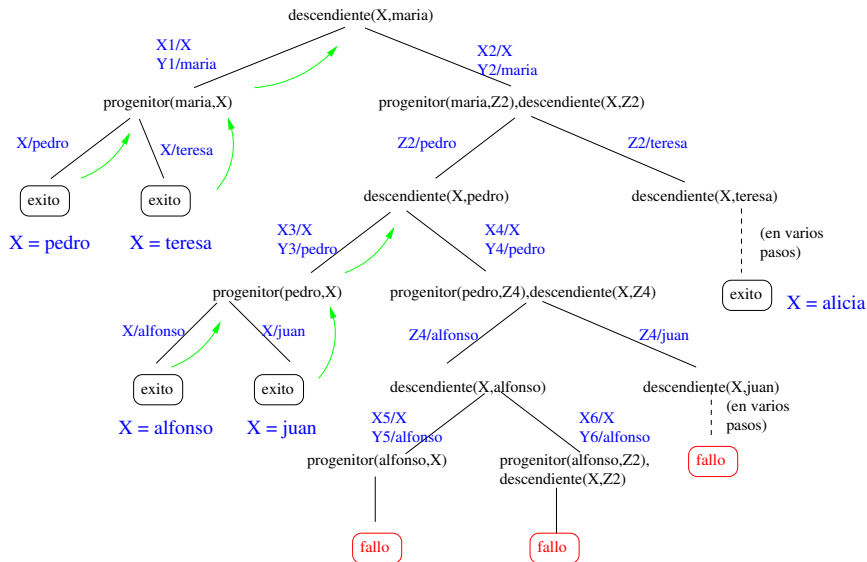
- Primero se aplica la regla
 $\text{abuelo}(X_1, Y_1) :-$
 $\text{progenitor}(Z, Y_1), \text{padre}(X_1, Z).$
- Se añaden los nuevos objetivos generados.
Se pueden aplicar dos hechos diferentes para `progenitor`:
`progenitor(jorge, alicia)` y
`progenitor(teresa, alicia) (*)`.
- Si se utiliza el primer hecho, se puede aplicar la regla
 $\text{padre}(X_2, Y_2) :-$
 $\text{progenitor}(X_2, Y_2), \text{hombre}(X_2).$
pero no es posible resolver uno de los nuevos objetivos generados
(`progenitor(javier, jorge)`).
- Por ello, debe **replantearse** (*backtracking*) la decisión tomada en (*).



Árboles de resolución (cont.)



Árboles de resolución (cont.)



Árboles de resolución (cont.)

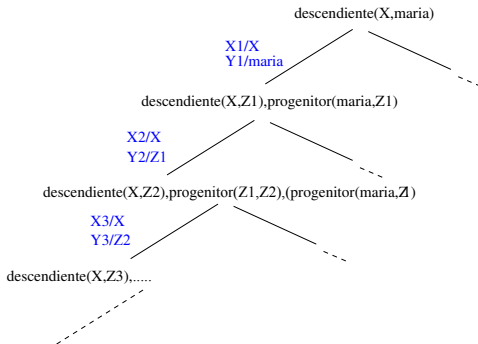
- Ahora veamos qué ocurriría si utilizáramos otra versión de descendiente/2:

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

```
descendiente(X,Y) :- progenitor(Y,X).
```

- Si siempre se utiliza la primera regla, y los objetivos siempre se evalúan de izquierda a derecha, se genera un árbol infinito.

- En Prolog debe tenerse esto en cuenta: puede existir solución en otra rama del árbol que no es alcanzable.



- En el espacio de búsqueda de un objetivo, Prolog realiza **búsqueda en profundidad y por la izquierda**, con *backtracking* en caso de fallo o de petición de más respuestas.

Términos: variables, constantes y estructuras

- **Variables:** Comienzan con una letra mayúscula (o “_”) y pueden incluir “_” y dígitos:

`X, Im4u, Var_2, _, _x, _22`

(La variable `_` se denomina variable anónima).

- **Constantes:** Comienzan con una letra minúscula y pueden incluir “_” y dígitos. También son constantes los números y algunos caracteres especiales. Entre comillas simples, cualquier cadena de caracteres:

`a, alicia, prog_logica, 23, 'Hungry man', []`

- **Estructuras:** están formadas por un nombre de estructura (**functor**) seguido por un número fijo de argumentos entre paréntesis:

`s(s(0))` `fecha(lunes, Mes, 2010)`

Los argumentos son a su vez términos:

`libro(titulo(don_quijote), autor(nombre(miguel),
apellido(de_cervantes)), Fecha_edicion)`

Términos: variables, constantes y estructuras (cont.)

- La **aridad** es el número de argumentos de una estructura. Una constante es una estructura con aridad cero.
- Ejemplos de términos:

<i>Término</i>	<i>Tipo</i>	<i>Functor principal</i>
pi	constante	pi/0
hora(min, sec)	estructura	hora/2
pair(Calvin, tiger(Hobbes))	estructura	pair/2
Tee(Alf, rob)	ilegal	—
A_good_time	variable	—

- Los funtores pueden definirse como **operadores** con notación *prefija*, *postfija* o *infija*:

'+' (a, b)	con notación infija es:	a + b
'-' (b)	con notación prefija es:	- b
'<' (a, b)	con notación infija es:	a < b
es_un(fluffy, gato)	con notación infija es:	fluffy es_un gato

Unificación

- La **unificación** es el mecanismo que se utiliza en Prolog para **pasar valores** y **devolver resultados** en los objetivos (y en el cuerpo de las reglas).
- También se utiliza para acceder a componentes de estructuras y dar valores a variables.
- Dos términos (o *literales*) A y B se dicen **unificables** si se pueden hacer **sintácticamente idénticos** dando valores a sus variables mediante una **sustitución**. Ejemplos:

A	B	θ	$A\theta$ y $B\theta$
dog	dog	\emptyset	dog
X	a	$\{X=a\}$	a
X	Y	$\{X=Y\}$	Y
$f(X, g(t))$	$f(m(h), g(M))$	$\{X=m(h), M=t\}$	$f(m(h), g(t))$
$f(X, g(t))$	$f(m(h), t(M))$	Imposible (1)	
$f(X, X)$	$f(Y, l(Y))$	Imposible (2)	

A continuación precisamos algo más estas ideas.

Unificación (cont.)

- Una **sustitución** θ es una asignación de términos a variables (distintas) X_1, \dots, X_n que escribimos en la forma $\{X_1/t_1, \dots, X_n/t_n\}$ o bien $\{X_1 = t_1, \dots, X_n = t_n\}$
 - ▶ A cada X_i/t_i le llamamos *ligadura*. Escribimos $\theta(X_i) = t_i$ o $X_i\theta = t_i$
 - ▶ Para todas las variables que no sean X_1, \dots, X_n , definimos $X\theta = X$
- La **aplicación de una sustitución** $\theta = \{X_1/t_1, \dots, X_n/t_n\}$ a un término t , que escribimos como

$$\theta(t) \quad \text{o bien} \quad t\theta \quad \text{o bien} \quad t[X_1/t_1, \dots, X_n/t_n]$$

es el resultado de reemplazar simultáneamente en t todas las apariciones de cada X_i por t_i . Las sustituciones se aplican también a literales, cláusulas, ...

- La **composición** de dos sustituciones θ, θ' (que notamos por $\theta\theta'$) queda definida por: $t(\theta\theta') = (t\theta)\theta'$. Escribimos $t\theta\theta'$ a secas.
- θ es **idempotente** si $\theta\theta = \theta$.
- θ es **más general** que θ' si $\theta\theta'' = \theta'$, para cierta θ'' .

Unificación (cont.)

- Un **unificador** de dos términos t_1, t_2 es una sustitución θ tal que $t_1\theta = t_2\theta$. *Se aplica también a literales, secuencias de términos,...*
- t_1, t_2 son **unificables** si tienen algún unificador.
- Dos términos pueden tener diferentes unificadores. Por ejemplo, si A es $f(X, g(T))$ y B es $f(m(H), g(M))$, tenemos:

θ	$A\theta$ y $B\theta$
$\{X=m(a), H=a, M=b, T=b\}$	$f(m(a), g(b))$
$\{X=m(H), M=f(A), T=f(A)\}$	$f(m(H), g(f(A)))$
$\{X=m(H), T=M\}$	$f(m(H), g(M))$

- Si t_1, t_2 son unificables, existe un unificador θ que cumple:
 - es más general que cualquier otro
 - es idempotente
 - no involucra a variables que no estén en t_1, t_2

Este θ es único salvo cambio de orden en ligaduras $X = Y$. Se denomina **unificador de máxima generalidad (umg)**.

- El *algoritmo de unificación* encuentra el umg.

Algoritmo de unificación (variante de Martelli-Montanari)

Plantea la unificación de A y B como un problema de resolución de ecuaciones.

- **Entrada:** Términos A y B.
- **Salida:** Conjunto S de ligaduras de variables que representa un umg de A y B, o **FALLO**

1. Inicialmente $S = \{A = B\}$
2. Mientras sea posible, aplicar alguna de las siguientes reglas:

Trivial Si $S = S' \cup \{X = X\}$, entonces $S \leftarrow S'$.

Descomp. Si $S = S' \cup \{f(t_1, \dots, t_n) = f(s_1, \dots, s_n)\}$,
entonces $S \leftarrow S' \cup \{t_1 = s_1, \dots, t_n = s_n\}$.

Conflicto Si $S = S' \cup \{f(t_1, \dots, t_n) = g(s_1, \dots, s_m)\}$ y $(f \neq g \text{ o } n \neq m)$,
entonces **FALLO**.

Ligadura Si $S = S' \cup \{X = t\}$ y $X \notin \text{var}(t)$ y $X \in \text{var}(S')$ y $X \neq t$,
entonces $S \leftarrow S'[X/t] \cup \{X = t\}$,

OccurCheck Si $S = S' \cup \{X = t\}$ y $X \in \text{var}(t)$, entonces **FALLO**.

Reorden Si $S = S' \cup \{t = X\}$ y t no es una variable,
entonces $S \leftarrow S' \cup \{X = t\}$.

Algoritmo de unificación – Ejemplos

- Proporciona el umg de: $A = p(X, X)$ y $B = p(f(Z), f(W))$

Algoritmo de unificación – Ejemplos

- Proporciona el umg de: $A = p(X, X)$ y $B = p(f(Z), f(W))$

ecuación	S	regla
-	$\{ p(X, X) = p(f(Z), f(W)) \}$	
$p(X, X) = p(f(Z), f(W))$	$\{ X = f(Z), X = f(W) \}$	descomposición
$X = f(Z)$	$\{ f(Z) = f(W), X = f(Z) \}$	ligadura
$f(Z) = f(W)$	$\{ X = f(Z), Z = W \}$	descomposición
$Z = W$	$\{ X = f(W), Z = W \}$	ligadura

Algoritmo de unificación – Ejemplos

- Proporciona el umg de: $A = p(X, X)$ y $B = p(f(Z), f(W))$

ecuación	S	regla
-	$\{ p(X, X) = p(f(Z), f(W)) \}$	
$p(X, X) = p(f(Z), f(W))$	$\{ X = f(Z), X = f(W) \}$	descomposición
$X = f(Z)$	$\{ f(Z) = f(W), X = f(Z) \}$	ligadura
$f(Z) = f(W)$	$\{ X = f(Z), Z = W \}$	descomposición
$Z = W$	$\{ X = f(W), Z = W \}$	ligadura

- Proporciona el umg de: $A = p(X, f(Y))$ y $B = p(Z, X)$

Algoritmo de unificación – Ejemplos

- Proporciona el umg de: $A = p(X, X)$ y $B = p(f(Z), f(W))$

ecuación	S	regla
-	$\{ p(X, X) = p(f(Z), f(W)) \}$	
$p(X, X) = p(f(Z), f(W))$	$\{ X = f(Z), X = f(W) \}$	descomposición
$X = f(Z)$	$\{ f(Z) = f(W), X = f(Z) \}$	ligadura
$f(Z) = f(W)$	$\{ X = f(Z), Z = W \}$	descomposición
$Z = W$	$\{ X = f(W), Z = W \}$	ligadura

- Proporciona el umg de: $A = p(X, f(Y))$ y $B = p(Z, X)$

ecuación	S	regla
-	$\{ p(X, f(Y)) = p(Z, X) \}$	
$p(X, f(Y)) = p(Z, X)$	$\{ X = Z, f(Y) = X \}$	descomposición
$X = Z$	$\{ f(Y) = Z, X = Z \}$	ligadura
$f(Y) = Z$	$\{ X = Z, Z = f(Y) \}$	reorden
$Z = f(Y)$	$\{ X = f(Y), Z = f(Y) \}$	ligadura

Algoritmo de unificación – Ejemplos (cont.)

- Proporciona el umg de: $A = p(X, f(Y))$ y $B = p(a, g(b))$

Algoritmo de unificación – Ejemplos (cont.)

- Proporciona el umg de: $A = p(X, f(Y))$ y $B = p(a, g(b))$

ecuación	S	regla
-	$\{ p(X, f(Y)) = p(a, g(b)) \}$	
$p(X, f(Y)) = p(a, g(b))$	$\{ X=a, f(Y)=g(b) \}$	descomposición
$f(Y)=g(b)$	FALLO	conflicto

Algoritmo de unificación – Ejemplos (cont.)

- Proporciona el umg de: $A = p(X, f(Y))$ y $B = p(a, g(b))$

ecuación	S	regla
-	$\{ p(X, f(Y)) = p(a, g(b)) \}$	
$p(X, f(Y)) = p(a, g(b))$	$\{ X=a, f(Y)=g(b) \}$	descomposición
$f(Y)=g(b)$	FALLO	conflicto

- Proporciona el umg de: $A = p(X, f(X))$ and $B = p(Z, Z)$

Algoritmo de unificación – Ejemplos (cont.)

- Proporciona el umg de: $A = p(X, f(Y))$ y $B = p(a, g(b))$

ecuación	S	regla
-	$\{ p(X, f(Y)) = p(a, g(b)) \}$	
$p(X, f(Y)) = p(a, g(b))$	$\{ X=a, f(Y)=g(b) \}$	descomposición
$f(Y)=g(b)$	FALLO	conflicto

- Proporciona el umg de: $A = p(X, f(X))$ and $B = p(Z, Z)$

ecuación	S	regla
-	$\{ p(X, f(X)) = p(Z, Z) \}$	
$p(X, f(X)) = p(Z, Z)$	$\{ X=Z, f(X)=Z \}$	descomposición
$X=Z$	$\{ f(Z)=Z, X=Z \}$	ligadura
$f(Z)=Z$	$\{ X=Z, Z=f(Z) \}$	reorden
$Z=f(Z)$	FALLO	Occur Check

- El predicado Prolog '='/2 permite unificar dos términos. Se define como el hecho '=' (X, X) y puede utilizar con notación infija:

?- $f(A, B) = f(g(B), f(A))$.
 $A = g(f(**))$,
 $B = f(g(**))$. (en SWI)

El mecanismo de resolución

- **Entrada:** Un programa lógico P y un objetivo Q
- **Salida:** sustitución respuesta θ , si $Q\theta$ deducible de P , *fallo* si no existe tal θ
- **(Seudo)-Algoritmo:**

$R \leftarrow Q$ // R es la secuencia de objetivos pendientes de evaluar.

$\theta \leftarrow \epsilon$ // θ es la sustitución respuesta acumulada, inicialmente la identidad.

mientras $R \neq \emptyset$ **hacer**

 Seleccionar A literal de R // **Regla de cómputo 1**

 Elegir una cláusula (renombrada) de P : $A' :- B_1, \dots, B_n$

 tal que A y A' unifican con unificador θ_1 // **regla de búsqueda**

si no existe ninguna cláusula que unifique con A **entonces**

 // **backtracking**

Fallo: replantear la elección de cláusula del literal anterior, restaurando θ

si no

 Eliminar A de R

 Añadir B_1, \dots, B_n a R // **Regla de cómputo 2**

 Aplicar la sustitución θ_1 a R

$\theta \leftarrow \theta\theta_1$

fin si

fin mientras

devolver θ

Regla de búsqueda y regla de cómputo

- En Prolog, la **regla de cómputo 1** selecciona los literales de izquierda a derecha, y la **regla de cómputo 2** añade los literales del cuerpo de una cláusula en el orden en el que aparecen en ésta, y al principio de la lista de objetivos pendientes.
- Por su parte, la **regla de búsqueda** se aplica en el orden en el que aparecen las cláusulas en el programa.
- El *backtracking* se realiza de forma implícita: si se produce un fallo, se vuelve al último punto en el que se ha aplicado la regla de búsqueda.
- Del mismo modo, cuando se ha tenido éxito y se necesitan más respuestas, se continúa el algoritmo como si se hubiera producido un fallo (se fuerza el *backtracking*).

El procedimiento de búsqueda de Prolog se suele recordar como ...

Búsqueda en profundidad y por la izquierda, con backtracking cronológico en caso de fallo.

Tipos de datos y estructuras de datos

- Hemos visto anteriormente los tipos de términos que se pueden utilizar en un programa lógico.
- Prolog no es un lenguaje tipado, pero en cualquier caso las variables lógicas pueden ligarse a términos de un tipo dado.
- Existen algunos tipos de datos predefinidos en Prolog (átomos, números, estructuras) y otros para los que se define una notación especial (listas, funtores con notación infija).
- Veremos algunos tipos de datos utilizados frecuentemente: listas y árboles, y cómo pueden utilizarse.

Listas

- Una lista es una sucesión ordenada y finita de objetos:

$$[X_1, X_2, \dots, X_n]$$

- En Prolog las listas se definen mediante una estructura con dos argumentos:
 - ▶ El primer argumento corresponde al **primer elemento** de la lista (cabeza).
 - ▶ El segundo argumento es el **resto** de la lista.
- La lista vacía se representa mediante la constante `[]`.
- Tradicionalmente, el functor de las listas en Prolog es el punto.
- Por ejemplo, la lista formada por las constantes uno, dos y tres se representa mediante el término `.(uno, .(dos, .(tres, [])))`.

Listas (cont.)

- **Notación alternativa:** Las listas también se pueden representar con otra notación:
 - . (A, B) se representa mediante $[A|B]$.
- Esta notación permite representar más de un elemento que aparece al principio de una lista: $[A, B|C]$ representa $.(A, .(B, C))$ (C es el resto de la lista excepto los dos primeros elementos).
- Ejemplos:

. (a, [])	$[a []]$	$[a]$
. (a, . (b, []))	$[a [b []]]$	$[a,b]$
. (a, . (b, . (c, [])))	$[a [b [c []]]]$	$[a,b,c]$
. (a, X)	$[a X]$	$[a X]$
. (a, . (b, X))	$[a [b X]]$	$[a,b X]$

- ▶ $[a,b]$ y $[a|X]$ unifican con $\{X = [b]\}$.
- ▶ $[a]$ y $[a|X]$ unifican con $\{X = []\}$.
- ▶ $[a]$ y $[a,b|X]$ no unifican.
- ▶ $[]$ y $[X]$ no unifican.

Listas (cont.)

- Más ejemplos:

$$[a, b] = [a, b | []] = [a | [b]] = .(a, .(b, []))$$

$$\begin{aligned} [a, X, b] &= [a, X | [b]] = [a | [X, b]] = [a | [X | [b]]] = [a, X, b | []] \\ &= .(a, [X, b]) = .(a, .(X, .(b, []))) \end{aligned}$$

$$[a, X, b | Y] = [a, X | [b | Y]] = [a | [X, b | Y]] = .(a, .(X, .(b, Y)))$$

$$[a, X, b, Y] = .(a, .(X, .(b, .(Y, []))))$$

- Ejercicios: Determina el resultado de la unificación de los siguientes pares de términos:

$$[a, b, c] = [X | [Y | Z]] \quad X=a, Y=b, Z=c \quad [1, 2] = [X, [Y | Z]] \quad !!$$

$$[a, b | C] = [X, Y] \quad X=a, Y=b, C=[] \quad [1, 2, Z] = [X, Y] \quad !!$$

$$[a, X, Y | U] = [X, Y | V] \quad X=a, Y=a, V=[a | U] \quad [1, 2] = [X | [Y | Z]] \quad X=1, Y=2, Z=[]$$

$$[1, 2 | [3]] = [X | [Y | Z]] \quad X=1, Y=2, Z=[3] \quad [[1, X]] = [Y | Z] \quad Y=[1, X], Z=[]$$

$$[1, X, Y | Z] = [X, Y | K] \quad X=1, Y=1, K=[3 | Z] \quad [1, 2] = [1 | 2] \quad !!$$

Predicados sobre listas

- Comprobación del tipo lista:

`% list(X) <-> X es una lista.`

Predicados sobre listas

- Comprobación del tipo lista:

`% list(X) <-> X es una lista.`

`list([]).`

`list([-|Y]) :- list(Y).`

- Para saber si un elemento está en una lista (member/2):

`% member(X,L) <-> X es un elemento de L.`

Predicados sobre listas

- Comprobación del tipo lista:

```
% list(X) <-> X es una lista.  
list([]).  
list([_|Y]) :- list(Y).
```

- Para saber si un elemento está en una lista (member/2):

```
% member(X,L) <-> X es un elemento de L.  
member(X, [X|_]).  
member(X, [_|Xs]) :- member(X,Xs).  
member/2 se puede utilizar de varias formas:
```

- ▶ para saber si un elemento está en una lista:
?- member(a, [b,c,a]).
- ▶ para enumerar los elementos de una lista:
?- member(X, [b,c,a]).
- ▶ para encontrar una lista en la que aparece un elemento:
?- member(a, L).

- Ejercicios: define prefijo/2, sufijo/2, sublista/2.

Predicados sobre listas (cont.)

- Concatenación de listas: En muchos casos se necesita un predicado que concatene dos listas:

`% concatenar(X,Y,Z) <-> Z es la concatenación de X e Y`

- Por ejemplo: `?- concatenar([a,b,c],[d,e],Z).`

`Z = [a, b, c, d, e].`

- ¿Cuál sería el caso base para definir este predicado?

Predicados sobre listas (cont.)

- Concatenación de listas: En muchos casos se necesita un predicado que concatene dos listas:

`% concatenar(X,Y,Z) <-> Z es la concatenación de X e Y`

- Por ejemplo: `?- concatenar([a,b,c],[d,e],Z).`

`Z = [a, b, c, d, e].`

- ¿Cuál sería el caso base para definir este predicado?
- Podemos definir un caso trivial: `concatenar([],X,X).`

Predicados sobre listas (cont.)

- Concatenación de listas: En muchos casos se necesita un predicado que concatene dos listas:

`% concatenar(X,Y,Z) <-> Z es la concatenación de X e Y`

- Por ejemplo: `?- concatenar([a,b,c],[d,e],Z).`

`Z = [a, b, c, d, e].`

- ¿Cuál sería el caso base para definir este predicado?
- Podemos definir un caso trivial: `concatenar([],X,X).`
- Si la primera lista tiene un solo elemento: `concatenar([a],X,[a|X]).`

Predicados sobre listas (cont.)

- Concatenación de listas: En muchos casos se necesita un predicado que concatene dos listas:

`% concatenar(X,Y,Z) <-> Z es la concatenación de X e Y`

- Por ejemplo: `?- concatenar([a,b,c],[d,e],Z).`

`Z = [a, b, c, d, e].`

- ¿Cuál sería el caso base para definir este predicado?
- Podemos definir un caso trivial: `concatenar([],X,X).`
- Si la primera lista tiene un solo elemento: `concatenar([a],X,[a|X]).`
- Si la primera lista tiene dos elementos:
`concatenar([a,b],X,[a,b|X]).`
- ¿Cómo se puede generalizar?

Predicados sobre listas (cont.)

- Definición de concatenar/3:

```
concatenar([], X, X).
```

```
concatenar([X|Xs], Ys, [X|Zs]) :- concatenar(Xs, Ys, Zs).
```

- En las librerías de Prolog, este predicado se denomina `append/3`.
- Posibles usos de concatenar/3:
 - ▶ Para concatenar dos listas:

```
?- concatenar([a,b], [c,d], X).
```
 - ▶ Para obtener la diferencia de dos listas:

```
?- concatenar(X, [c,d], [a,b,c,d]).
```
 - ▶ Para dividir una lista en dos:

```
?- concatenar(X, Y, [a,b,c,d]).
```

Predicados sobre listas (cont.)

- Invertir una lista:

`% invertir(X,Y) <-> Y es la lista inversa de X`

- Por ejemplo: `?- invertir([a,b,c],Z).`

`Z = [c, b, a].`

- ¿Cómo se podría definir este predicado?

Predicados sobre listas (cont.)

- Invertir una lista:

`% invertir(X,Y) <-> Y es la lista inversa de X`

- Por ejemplo: `?- invertir([a,b,c],Z).`

`Z = [c, b, a].`

- ¿Cómo se podría definir este predicado?

`invertir([], []).`

`invertir([X|Xs],Ys):-invertir(Xs,Zs),concatenar(Zs,[X],Ys).`

- ¿Qué complejidad tiene este predicado? ¿Podría ser más eficiente?

Predicados sobre listas (cont.)

- Invertir una lista:

`% invertir(X,Y) <-> Y es la lista inversa de X`

- Por ejemplo: `?- invertir([a,b,c],Z).`

`Z = [c, b, a].`

- ¿Cómo se podría definir este predicado?

`invertir([],[]).`

`invertir([X|Xs],Ys):-invertir(Xs,Zs),concatenar(Zs,[X],Ys).`

- ¿Qué complejidad tiene este predicado? ¿Podría ser más eficiente?

`invertir2([],Xs,Xs).`

`invertir2([X|Xs],Ys,Zs):-invertir2(Xs,[X|Ys],Zs).`

- Ejercicio: define predicados para eliminar un elemento de una lista (la primera o todas las apariciones):

`eliminar_uno(Xs,Elem,Zs), eliminar_todos(Xs,Elem,Zs)`

(`Zs` es la lista `Xs` sin una/todas las apariciones del elemento `Elem`).

Arboles binarios

- No existe una sintaxis específica para árboles en Prolog: se pueden utilizar estructuras. Por ejemplo:

```
arbol(Elemento,Izquierdo,Derecho)
```

- Un árbol vacío se puede representar con una constante, por ejemplo: `void`.
- ¿Cómo se puede verificar si un término es un árbol binario?

Arboles binarios

- No existe una sintaxis específica para árboles en Prolog: se pueden utilizar estructuras. Por ejemplo:

```
arbol(Elemento,Izquierdo,Derecho)
```

- Un árbol vacío se puede representar con una constante, por ejemplo: `void`.
- ¿Cómo se puede verificar si un término es un árbol binario?

```
arbol_binario(void).
```

```
arbol_binario(arbol(_,I,D)):-
```

```
    arbol_binario(I),
```

```
    arbol_binario(D).
```

Arboles binarios (cont.)

- ¿Como podemos saber si un elemento está en un arbol?

Arboles binarios (cont.)

- ¿Como podemos saber si un elemento está en un arbol?

```
member_arbol(X, arbol(X, _, _)) .
```

```
member_arbol(X, arbol(_, I, _)) :- member_arbol(X, I) .
```

```
member_arbol(X, arbol(_, _, D)) :- member_arbol(X, D) .
```

- ¿Cómo se puede proporcionar el recorrido en preorden de un árbol binario?

Arboles binarios (cont.)

- ¿Como podemos saber si un elemento está en un arbol?

```
member_arbol(X, arbol(X, _, _)) .  
member_arbol(X, arbol(_, I, _)) :- member_arbol(X, I) .  
member_arbol(X, arbol(_, _, D)) :- member_arbol(X, D) .
```

- ¿Cómo se puede proporcionar el recorrido en preorden de un árbol binario?

```
pre_orden(void, []) .  
pre_orden(arbol(X, I, D), Orden) :-  
    pre_orden(I, OrdenI) ,  
    pre_orden(D, OrdenD) ,  
    append([X|OrdenI], OrdenD, Orden) .
```

- Ejercicio: define los predicados para recorrer un árbol binario en inorden y en postorden.
- Ejercicio: define un predicado que calcule el número de nodos de un árbol binario (con aritmética de Peano).

Estructuras recursivas: expresiones simbólicas

- Una expresión aritmética como $2*x+3*x^2$ también es para Prolog un término, donde los operadores se corresponden con funtores en notación infija. Por ejemplo:

```
?- display(2*x+3*x^2).  
+(* (2, x), * (3, ^ (x, 2)))  
true.
```

`display/1` muestra por pantalla en notación prefija el argumento que recibe.

- La evaluación de expresiones aritméticas se realiza mediante predicados específicos (`is/2`, `</2`, `>/2`) que veremos más adelante.
- Se puede hacer un (mini) derivador simbólico: `deriv(Exp, Var, Deriv)`

Estructuras recursivas: expresiones simbólicas (cont.)

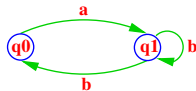
```
deriv(X,X,s(0)).  
deriv(C,X,0) :- nat(C).  
deriv(U+V,X,DU+DV) :- deriv(U,X,DU), deriv(V,X,DV).  
deriv(U-V,X,DU-DV) :- deriv(U,X,DU), deriv(V,X,DV).  
deriv(U*V,X,DU*V+U*DV) :- deriv(U,X,DU), deriv(V,X,DV).  
deriv(U/V,X,(DU*V-U*DV)/V^s(s(0))) :- deriv(U,X,DU),  
deriv(V,X,DV).  
deriv(U^s(N),X,s(N)*U^N*DU) :- deriv(U,X,DU), nat(N).  
deriv(log(U),X,DU/U) :- deriv(U,X,DU).  
...
```

- La expresión resultante se podría simplificar.
- Consultas:

```
deriv(s(s(s(0)))*x+s(s(0)),x,Y).  
deriv(s(s(s(0)))*x+s(s(0)),x,0*x+s(s(s(0)))*s(0)+0).  
deriv(E,x,0*x+s(s(s(0)))*s(0)+0).
```

Programación recursiva: autómatas

- Reconocimiento de la secuencia de caracteres aceptada por un *autómata finito no determinista* (**q0** es estado *inicial* y *final*):



- Podemos representar cada transición con un hecho:

```
delta(q0,a,q1).
```

```
delta(q1,b,q0).
```

```
delta(q1,b,q1).
```

- Los estados iniciales y finales también los representamos con hechos:

```
inicial(q0).
```

```
final(q0).
```

- El programa que determina si una secuencia es aceptada es:

```
aceptar(S):- inicial(Q), aceptar_desde(S,Q).
```

```
aceptar_desde([],Q):- final(Q).
```

```
aceptar_desde([X|Xs],Q):-
```

```
    delta(Q,X,NQ), aceptar_desde(Xs,NQ).
```

Programación recursiva: autómatas (cont.)

- ¿Se podría definir de modo similar un autómata con pila?

Programación recursiva: autómatas (cont.)

- ¿Se podría definir de modo similar un autómata con pila?

```
aceptarP(S) :- inicialP(Q), aceptarP_desde(S,Q,[]).
```

```
aceptarP_desde([],Q,[]) :- finalP(Q).
```

```
aceptarP_desde([X|Xs],Q,S) :-
```

```
    delta(Q,X,S,NQ,NS), aceptarP_desde(Xs,NQ,NS).
```

```
inicialP(q0).
```

```
finalP(q1)
```

```
delta(q0,X,Xs,q0,[X|Xs]).
```

```
delta(q0,X,Xs,q1,[X|Xs]).
```

```
delta(q0,X,Xs,q1,Xs).
```

```
delta(q1,X,[X|Xs],q1,Xs).
```

- ¿Qué secuencia reconoce este autómata?
- **Ejercicio: define un programa que reconozca el lenguaje**
 $L = \{a^k b^k | k \geq 0\}.$