Programación Lógica

Francisco López Fraguas **D423** **fraguas@sip.ucm.es**
Jesús Correas Fernández **D426** **jcorreas@fdi.ucm.es**

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

(elaborado parcialmente con material docente de R. Pinero,
M. Hermenegildo (UPM) y J. Sánchez)

Programación Lógica

1. Elementos del lenguaje: hechos, reglas, predicados, objetivos.
2. Ejecución de programas lógicos. Semántica de programas lógicos.
3. Arbol de resolución. *Backtracking*.
4. Términos: constantes, variables, estructuras.
5. Algoritmo de unificación.
6. Mecanismo de resolución.
7. Regla de búsqueda y Regla de cómputo
8. Tipos de datos y estructuras de datos.
9. Listas
10. Árboles y otros tipos de datos recursivos.
11. Programación recursiva.

Elementos del lenguaje: hechos y reglas

- Un programa lógico está formado por **cláusulas** que pueden ser de dos tipos:
  - **Hechos:** H $:-$ *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.
  ```

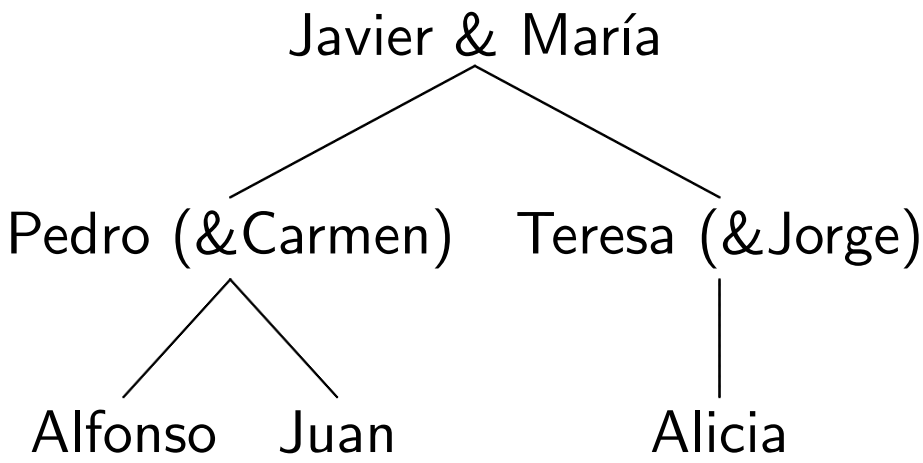
- La ejecución consiste en buscar la respuesta a ese objetivo.
- Ejemplos de objetivos:
  - ```
    ?- progenitor(pedro,arturo).
    ```
    ¿Pedro es progenitor de Arturo?
  - Los objetivos pueden contener variables:
    ```
    ?- progenitor(pedro,X).
    ```
    ¿De quién es Pedro progenitor?
    ```
    ?- 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 porporcionan 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

```
hombre(javier).
hombre(pedro).
hombre(jorge).
hombre(alfonso).
hombre(juan).
mujer(maria).
mujer(carmen).
mujer(teresa).
mujer(alicia).
```

```
progenitor(javier,pedro).
progenitor(javier,teresa).
progenitor(maria,pedro).
progenitor(maria,teresa).
progenitor(pedro,alfonso).
progenitor(pedro,juan).
progenitor(carmen,juan).
progenitor(carmen,alfonso).
progenitor(jorge,alicia).
progenitor(teresa,alicia).
```

- 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).

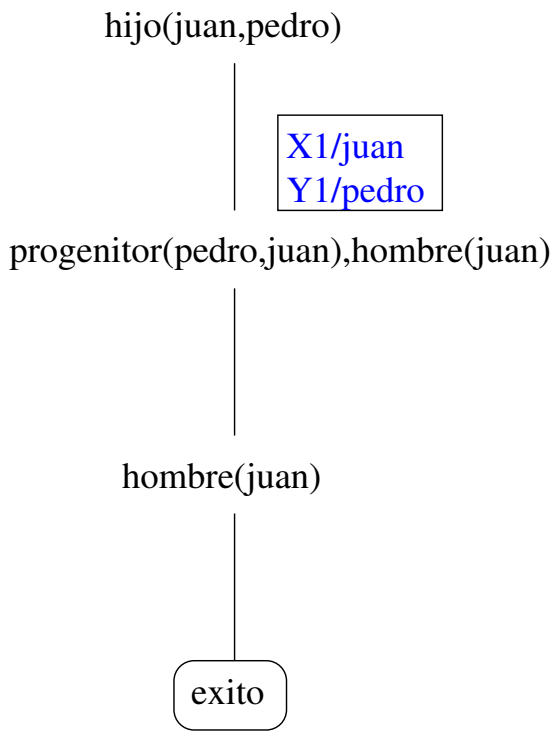
distinto(X,Y):- X \= Y.  % \= está predefinido.
```

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

Árboles de resolución (cont.)

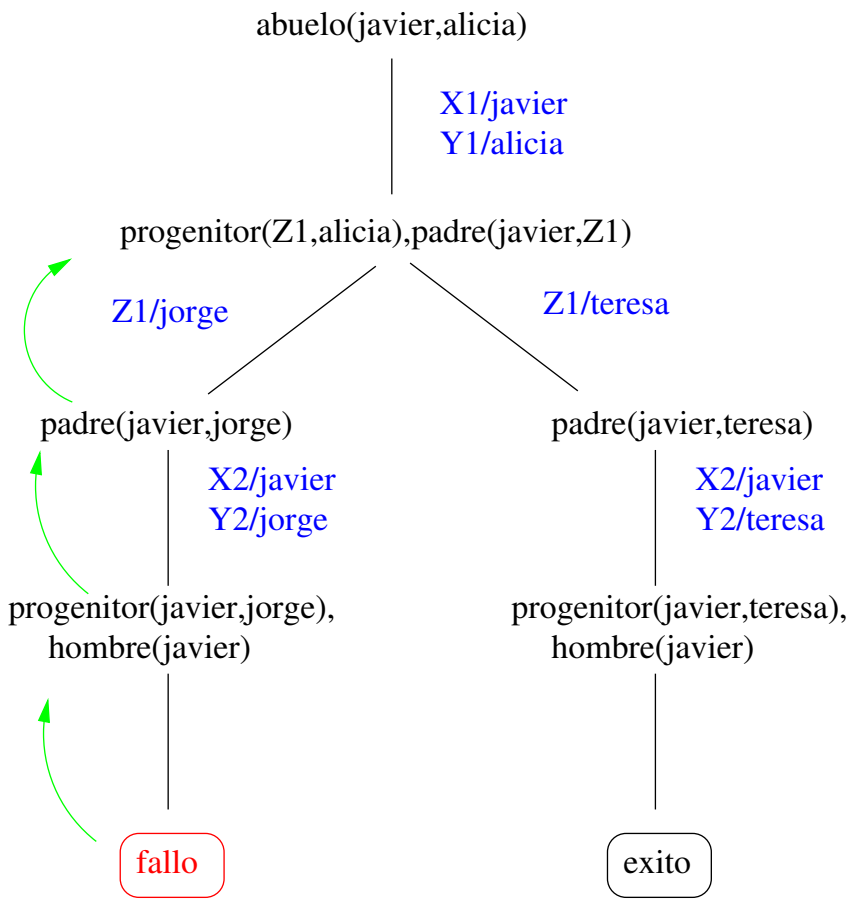
- Para resolver un objetivo sobre un programa lógico se utiliza el **mecanismo de resolución**.
- Aunque para definir correctamente el mecanismo de resolución es necesario tener en cuenta el concepto de unificación, vamos a ver cómo se aplica el mecanismo de resolución en los ejemplos anteriores.
- En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

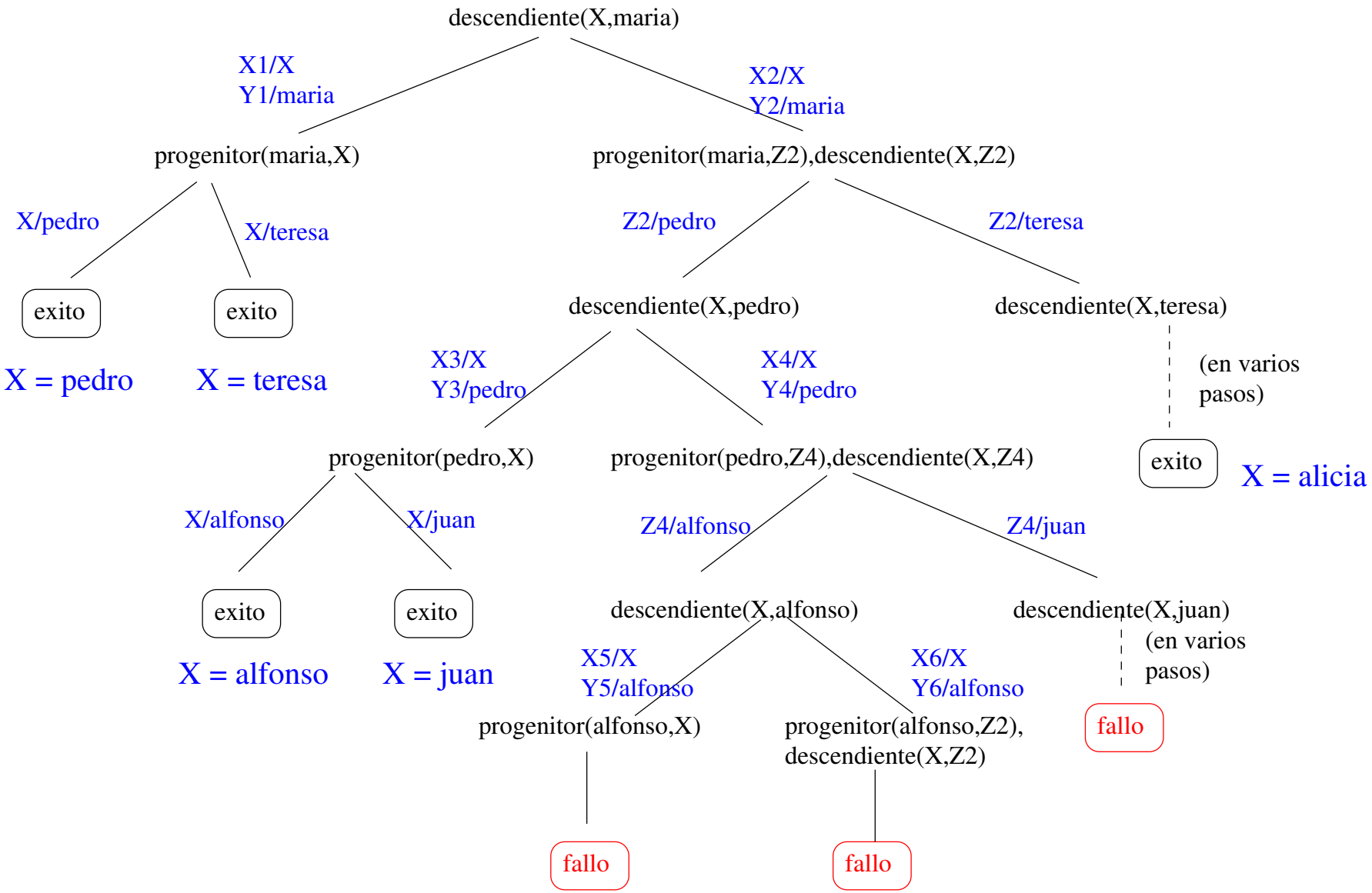


Árboles de resolución (cont.)

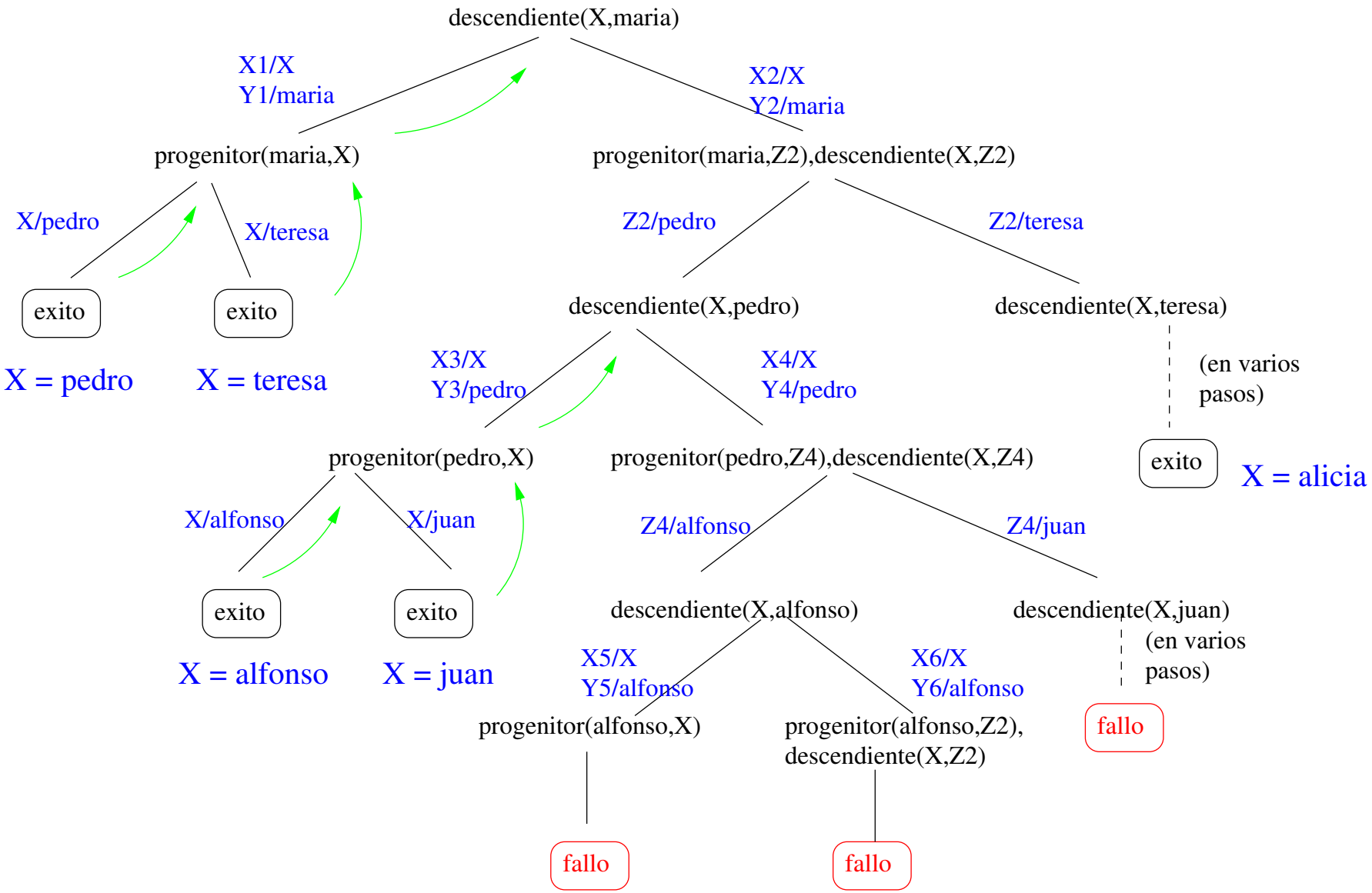
- Primero se aplica la regla
  `abuelo(X1,Y1) :-`
  `progenitor(Z,Y1), padre(X1,Z).`
- Se añaden los nuevos objetivos generados. Se pueden aplicar dos hechos diferentes para progenitor:
  `progenitor(jorge,alicia)` y
  `progenitor(teresa,alicia)` (*).
- Si se utiliza el primer hecho, se puede aplicar la regla
  `padre(X2,Y2) :-`
  `progenitor(X2,Y2), hombre(X2).`
  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.)

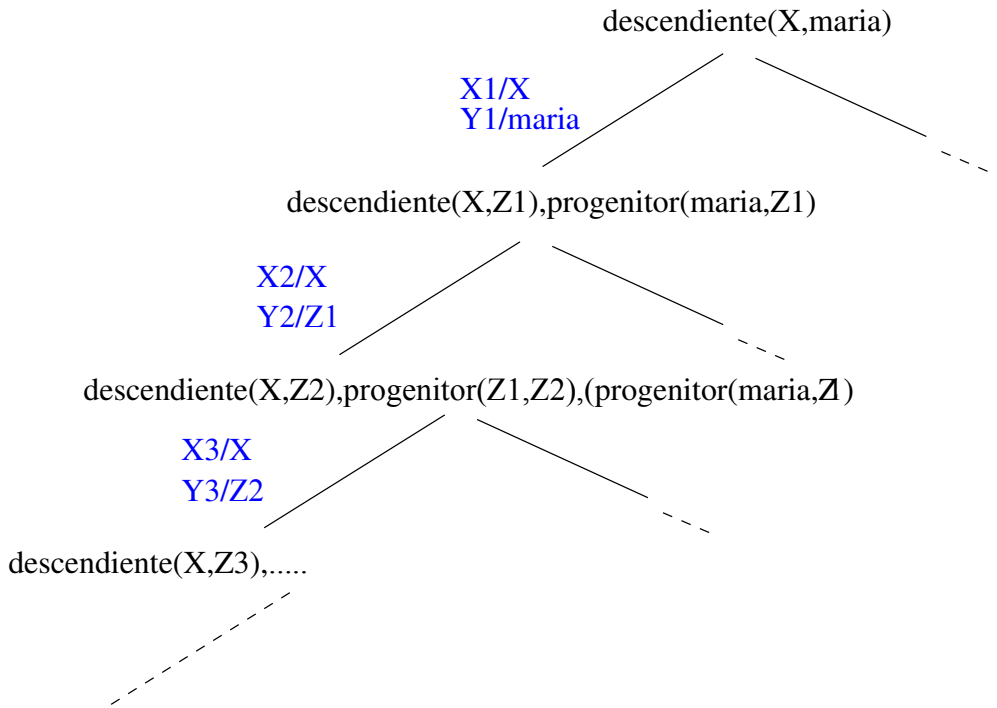


Árboles de resolución (cont.)

- 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 evaluan 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:

Término	*Tipo*	*Functor principal*
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 functores 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) $X_1, \ldots, X_n$ que escribimos en la forma
  $\{X_1/t_1, \ldots, X_n/t_n\}$ o bien $\{X_1 = t_1, \ldots, X_n = t_n\}$
  - 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 $X_1, \ldots, X_n$, definimos $X\theta = X$
- La **aplicación de una sustitución** $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$ a un término t, que escribimos como

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

  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** $S = S' \cup \{f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)\}$, **entonces** $S \leftarrow S' \cup \{t_1 = s_1, \ldots, t_n = s_n\}$.

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

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

OccurCheck **Si** $S = S' \cup \{X = t\}$ y $X \in var(t)$, **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: $\mathtt{A'} \ \mathtt{:-} \ \mathtt{B}_1\mathtt{,}\ldots\mathtt{,}\mathtt{B}_n$
      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 $B_1, \ldots, B_n$ a R // Regla de cómputo 2
      Aplicar la sustitución θ 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, functores 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:

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

- 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,[]))
[a,X,b] = [a,X|[b]] = [a|[X,b]] = [a|[X|[b]]] = [a,X,b|[]]
        = .(a,[X,b]) =.(a,.(X,.(b,[])))
[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]]          [1,2]   = [X,[Y|Z]]
[a,b|C]   = [X,Y]              [1,2,Z] = [X,Y]
[a,X,Y|U] = [X,Y|V]            [1,2]   = [X|[Y|Z]]
[1,2|[3]] = [X|[Y|Z]]          [[1,X]] = [Y|Z]
[1,X,Y|Z] = [X,Y|K]            [1,2]   = [1|2]
```

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 functores 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.)

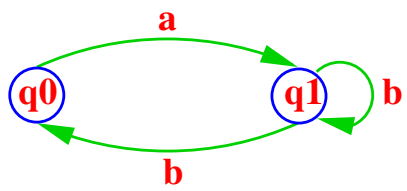
```
deriv(X,X,s(0)).
deriv(C,X,0) :- nat(C).
deriv(U+V,X,DU+DV) :- deriv(U,X,DU), deriv(V,X,DV).
deriv(U-V,X,DU-DV) :- deriv(U,X,DU), deriv(V,X,DV).
deriv(U*V,X,DU*V+U*DV) :- deriv(U,X,DU), deriv(V,X,DV).
deriv(U/V,X,(DU*V-U*DV)/V^s(s(0))) :- deriv(U,X,DU),
deriv(V,X,DV).
deriv(U^s(N),X,s(N)*U^N*DU) :- 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 $L = \{a^k b^k | k \geq 0\}$.

Programación Lógica

Aspectos propios del lenguaje Prolog

Francisco López Fraguas **D423** **fraguas@sip.ucm.es**
Jesús Correas Fernández **D426** **jcorreas@fdi.ucm.es**

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

(elaborado parcialmente con material docente de R. Pinero, M. Hermenegildo (UPM) y J. Sánchez)

1. **Predicados aritméticos en Prolog.**
2. Predicados metalógicos.
3. Control en Prolog: corte y negación.
4. Orden superior. Predicados de agregación.
5. Gestión de la base de reglas. Modificación dinámica del programa.
6. Precedencia de operadores (op/3). Creación de nuevos operadores.
7. Depurador de Prolog. El modelo "Byrd box".
8. Entrada/salida y manejo de ficheros.
9. Programación eficiente, listas en diferencias, DCGs.

Predicados aritméticos

- Hasta ahora hemos visto la notación de Peano para la aritmética (cero y sucesor), pero en programas reales esto es poco práctico.
- En Prolog los números son constantes del lenguaje.
- Además, se incorpora de forma estándar el predicado **is/2** para evaluar una expresión aritmética formada por números y operadores ($+, -, *, /, \hat{}, sin/1, cos/1, log/1$, etc.):

```
?- X is 2+3*5.
X = 17
```

- `is/2` requiere que los argumentos estén instanciados adecuadamente:
  - El primer argumento debe ser una variable o un término que represente un número.
  - El segundo argumento debe ser un término que represente una **expresión aritmética evaluable**
- Otros predicados aritméticos son: $</2$, $>/2$, $=</2$, $>=/2$, $=:=/2$, $=\backslash=/2$

Predicados aritméticos (cont.)

- `is/2` y los demás predicados aritméticos no son reversibles: `6 is X+X` no es correcto si `X` es una variable libre.
- Pero sí es correcto: `?- Y = 3*4+2, X is Y/2.`
- Ejercicios:
  - Diseña un predicado que calcule el factorial de un número.
  - Diseña un predicado que sume 1 a todos los elementos de una lista.
  - Diseña un predicado que, dadas dos listas de números de igual longitud, devuelva una lista resultado de sumar los elementos de las listas anteriores dos a dos.
  - Diseña un predicado que, dado un árbol binario de números enteros positivos, obtenga el valor máximo.

Programación Lógica

1. Predicados aritméticos en Prolog.
2. **Predicados metalógicos.**
3. Control en Prolog: corte y negación.
4. Orden superior. Predicados de agregación.
5. Gestión de la base de reglas. Modificación dinámica del programa.
6. Precedencia de operadores (op/3). Creación de nuevos operadores.
7. Depurador de Prolog. El modelo "Byrd box".
8. Entrada/salida y manejo de ficheros.
9. Programación eficiente, listas en diferencias, DCGs.

Predicados metalógicos. Comprobación de tipo

- Los predicados *metalógicos* son aquellos que no se pueden representar en lógica de primer orden.
- En Prolog existen predicados para determinar el tipo de datos de un término:
  - `integer(X)` tiene éxito si `X` está instanciado a un número entero.
  - `float(X)` tiene éxito si `X` está instanciado a un número en coma flotante.
  - `number(X)` tiene éxito si `X` está instanciado a un número.
  - `atomic(X)` tiene éxito si `X` está instanciado a una constante numérica o no numérica.
    Ejemplo: los objetivos `atomic(f(3))` y `atomic(3+4)` fallan.
  - `atom(X)` tiene éxito si `X` está instanciado a una constante no numérica (en el lenguaje Prolog las constantes se denominan *átomos*).
- No se pueden utilizar para **generar** constantes: si el argumento es una variable libre, fallan.

Predicados metalógicos. Comprobación de tipo (cont.)

- Por ejemplo, se pueden utilizar los predicados de comprobación de tipo para definir un predicado `suma/3` reversible:

```
suma(A,B,C):- number(A), number(B), C is A+B.
suma(A,B,C):- number(A), number(C), B is C-A.
suma(A,B,C):- number(B), number(C), A is C-B.
```

- Sin embargo, `suma(X,Y,10)` falla si `X` e `Y` son variables.
- **Ejercicios:**
  - Define el predicado `sumanum/2` que proporcione la suma de los elementos numéricos de una lista que puede tener cualquier tipo de términos en sus elementos.
    - ★ ¿Qué ocurre si pedimos todas las soluciones de `sumanum([3,a,4])`? (Más adelante veremos la forma de solucionarlo).
  - Define el predicado `listapos/2` que dada una lista de números proporcione la lista de los números positivos. ¿Qué ocurre si pedimos todas las soluciones?.
  - Define el predicado `particion(P,L,Men,May)` que, dada una lista de números `L` proporciona en `Men` la lista de elementos menores o iguales a `P` y en `May` los elementos mayores a `P`. Utilízalo para definir `quicksort/2`.

Predicados metalógicos. Inspección de estructuras

- Las estructuras que hemos manejado hasta ahora (por ejemplo, `arbol(a,void,void)`) se pueden manejar en Prolog de forma genérica.
- Existen tres predicados que permiten descomponer y construir estructuras dinámicamente: `functor/3`, `arg/3` y el operador `'=..'`.
- `functor(T,Fn,Ar)` tiene éxito si el término `T` tiene nombre de functor `Fn` y aridad `Ar`. Permite varios modos de uso:
  - Si `T` está instanciado a una estructura, unifica `Fn` con el nombre de functor y `Ar` con su aridad.
  - Si `Fn` y `Ar` están instanciados a una constante y un número natural, unifica `T` con ese nombre de functor y aridad.
- Ejemplos:

```
?- functor(arbol(A,B,C),Fn,Ar).
Fn = arbol,
Ar = 3.
```

```
?- functor(T,arbol,3).
T = arbol(_G236, _G237, _G238).
```

- ¿Cuál es el resultado del objetivo `functor([3,4],Fn,Ar)`?.

Predicados metalógicos. Inspección de estructuras (cont.)

- `arg(N,T,Arg)` tiene éxito si el término `T` tiene en el argumento `N` el término `Arg`. Modos de uso:
  - Si `N` está instanciado a un número natural y `T` a una estructura (con aridad `N` al menos), unifica `Arg` con el `N`-ésimo argumento de `T`.
  - (exclusivo de SWI) Si `T` está instanciado a una estructura, unifica en *backtracking* `N` y `Arg` con los distintos argumentos de `T`.
- Ejemplos:

```
?- arg(2,progenitor(juan,sara),V).
V = sara
?- arg(N,progenitor(juan,sara),V).
N = 1
V = juan ;
N = 2
V = sara
```

- ¿Qué devuelve el objetivo `arg(N,[a,b,c(6)],V)`?

Predicados metalógicos. Inspección de estructuras (cont.)

- `T =.. Lista` tiene éxito si `Lista` es una lista que contiene como primer elemento el nombre del functor del término `T` y como resto de los elementos los argumentos del término `T`.

$$\underbrace{<functor>\,(arg_1,\ldots,arg_n)}_{\text{Término}} \quad =.. \quad \underbrace{[<functor>,arg_1,\ldots,arg_n]}_{\text{Lista}}$$

- Modos de uso:
  - Si `T` está instanciado a una estructura, unifica `Lista` con el nombre del functor y los argumentos de `T`.
  - Si `Lista` está unificado a una lista (y el primer argumento es un átomo de Prolog: una constante no numérica), unifica `T` con el término tal que su functor es el primer elemento de `Lista` y sus argumentos son los demás elementos de `Lista`.
- Ejemplos:

```
?- progenitor(juan,sara) =.. L.
L = [progenitor, juan, sara]

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

```
?- T =.. [f, a, b(d)].
T = f(a, b(d))

?- [1,2] =.. L.
L = ['.', 1, [2]]
```

Predicados metalógicos. Inspección de estructuras (cont.)

- **Ejercicios:**
  - Define un predicado que, dado un término, obtenga la lista de los componentes atómicos (constantes, numéricas o no) del término. Por ejemplo, `term_atomic(arbol(1,arbol(a,void),void),L)` debe unificar `L` con la lista `[1,a,void,void]`.
    - ¿Qué ocurre con la consulta `term_atomic(arbol(X,Y,void),L)`?
  - Define un predicado que determine si un término es subtérmino de otro. Por ejemplo, `subterm(b,arbol([a,b,c,d],void,void))` tiene éxito, y `subterm(X,arbol([a,b,c,d],void,void))` debe devolver todos los subtérminos del segundo argumento.
    - ¿Qué ocurre con la consulta `subterm(d,arbol([a,b,c,d],X,void))`?

Predicados metalógicos. Estado de las variables

- Existen predicados metalógicos para consultar el estado de instanciación de las variables:
  - `var(X)` tiene éxito si `X` es una variable libre.
  - `nonvar(X)` tiene éxito si `X` **no** es una variable (pero puede contener variables).
  - `ground(X)` tiene éxito si `X` no contiene variables libres.
- También se pueden comparar términos entre sí:
  - `X == Y` si los términos `X` e `Y` son **idénticos** (con las mismas variables).
  - `X \== Y` si los términos `X` e `Y` **no** son idénticos.
  - `X @< Y`, `X @> Y` `X @=< Y` y `X @>= Y` permiten comparar términos (en orden alfabético): `f(a) @< f(b)` tiene éxito y `f(b) @> g(a)` falla (La comparación de variables depende del sistema).
- Ejercicios:
  - Modifica el predicado `term_atomic(Term,L)` para que no se produzca un error cuando `Term` contenga variables.
  - Modifica el predicado `subterm(Sub,Term)` para que no instancie variables de los argumentos.

1. Predicados aritméticos en Prolog.
2. Predicados metalógicos.
3. **Control en Prolog: corte y negación.**
4. Orden superior. Predicados de agregación.
5. Gestión de la base de reglas. Modificación dinámica del programa.
6. Precedencia de operadores (op/3). Creación de nuevos operadores.
7. Depurador de Prolog. El modelo "Byrd box".
8. Entrada/salida y manejo de ficheros.
9. Programación eficiente, listas en diferencias, DCGs.

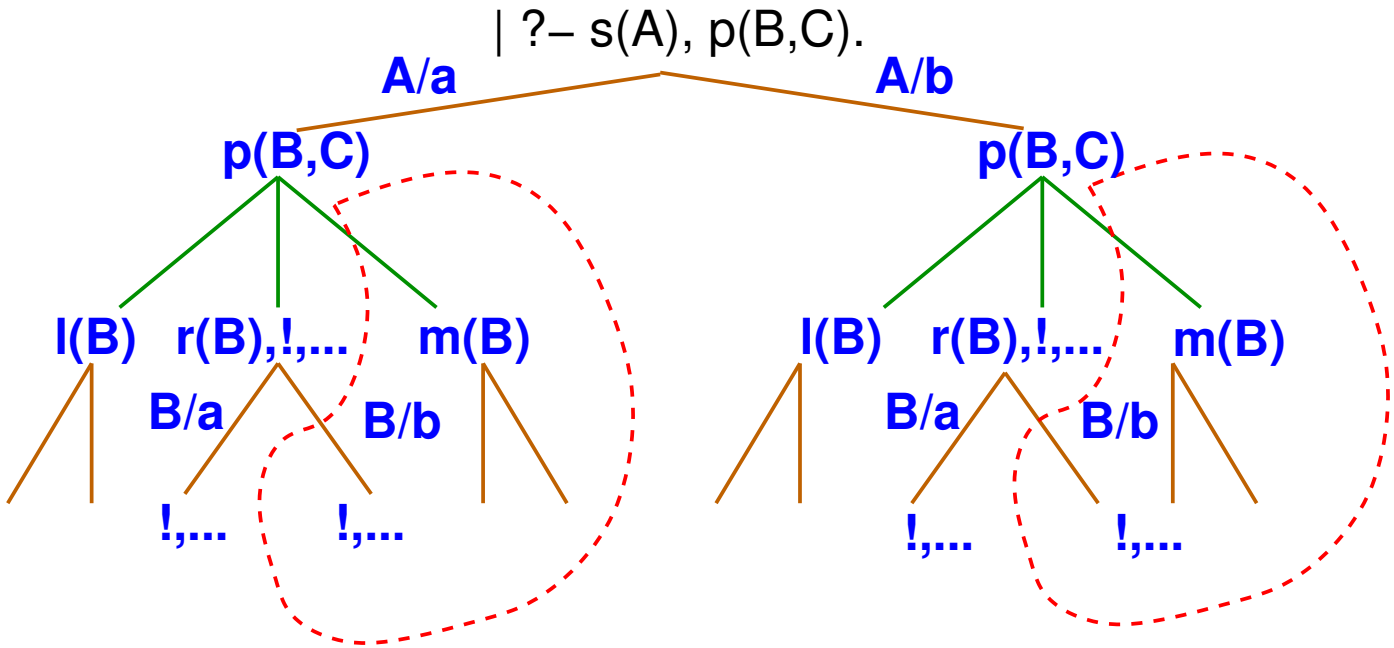
Control en Prolog: el corte (!)

- El predicado `!` (leído *corte*) proporciona control sobre el mecanismo de backtracking de Prolog:
  - Siempre tiene éxito,
  - pero tiene el **efecto lateral** de podar todas las elecciones alternativas en el nodo correspondiente en el árbol de búsqueda.
- Este predicado afecta al **comportamiento operacional** de Prolog.
  - es en cierto sentido un *predicado impuro* (ajeno a la lógica)
  - pero es un predicado muy útil... casi fundamental para el programador de Prolog

Control en Prolog: el corte (cont.)

```
s(a).          p(X,Y):- l(X), ...          r(a).
s(b).          p(X,Y):- r(X), !, ...       r(b).
               p(X,Y):- m(X), ...
```

- Supongamos que se realiza la siguiente consulta:



- La segunda alternativa de `r(X)` no se considera.
- La tercera cláusula de `p/2` no se considera.

Control en Prolog: el corte (cont.)

- Cuando se resuelve un objetivo p con una cláusula de la forma:

$$p :- \quad q_1, \ldots, q_n, !, r_1, \ldots, r_m$$

- Prolog intenta resolver los objetivos $q_1, \ldots, q_n$ normalmente (haciendo backtraking sobre cada uno de ellos si es necesario);
- pero una vez que se alcanza el corte ! se descartan automáticamente:
  - las alternativas que quedasen por explorar para $q_1, \ldots, q_n$
  - el resto de claúsulas para p que vengan a continuación de esta
- Sí que se puede hacer backtraking sobre $r_1, \ldots, r_m$

Control en Prolog: el corte (cont.)

- Por ejemplo, en el predicado `sumanum/2` que proporciona la suma de los elementos numéricos de una lista, se puede añadir un corte para evitar respuestas incorrectas:

```
sumanum([],0).
sumanum([X|Xs],N):- number(X),!, sumanum(Xs,N1), N is X+N1.
sumanum([_|Xs],N):- sumanum(Xs,N).
```

- Cuando se comprueba que el primer elemento de la lista es un número, se puede **descartar con seguridad** el caso en que no lo es (representado en la tercera cláusula de `sumanum/2`).
- El corte permite reducir el tamaño del árbol de búsqueda:

```
max(X,Y,X):- X>Y, !.
max(X,Y,Y):- X=<Y.
```

- Esto puede aumentar la eficiencia de algunos predicados.

Control en Prolog: el corte (cont.)

- Sin embargo, si se utiliza el corte para representar la semántica del programa se pueden producir soluciones incorrectas:

```
max(X,Y,X):- X>Y, !.
max(X,Y,Y).
```

- ¿Qué resultado proporciona la consulta `?- max(5,2,2).`?
- Otro ejemplo:

```
membercheck(X,[X|Xs]):- !.
membercheck(X,[_|Xs]):- membercheck(X,Xs).
```

- Sin embargo, este predicado no nos permite **reevaluar** el objetivo para distintos valores de `X`.
- `membercheck/2` sólo sirve para comprobar que un elemento dado está en la lista.
- Por ejemplo, `?- membercheck(X,[a,b,c]),membercheck(X,[b,c]).` no obtiene ninguna respuesta.

Control en Prolog: negación por fallo

- Utilizando el corte (y `fail/0`) es posible definir un predicado que tenga el comportamiento contrario a otro: que `p(X)` falle cuando `q(X)` tenga éxito y viceversa:

```
p(X):- q(X),!,fail.
p(_).
```

(`fail/0` es un predicado que nunca tiene éxito.)

- Esto se puede generalizar utilizando *orden superior* para definir la **negación**:

```
not(Goal):- call(Goal),!,fail.
not(_).
```

- `call/1` permite evaluar un objetivo que se le pasa como argumento.
- La negación en Prolog se debe entender como el **fallo finito** del objetivo.
- En Prolog, la negación está implementada con el operador `\+ Goal`.

Control en Prolog: negación por fallo (cont.)

- La negación de prolog es **negación por fallo**:
  - `not(P)` tiene éxito $\Leftrightarrow$ el árbol de resolución de P es finito y todos sus nodos son de fallo
  - `not(P)` tiene éxito $\Leftrightarrow$ P no es consecuencia lógica de las reglas del programa y esto es puede demostrar en tiempo finito
  - `not(P)` tiene éxito $\not\Leftrightarrow$ `not(P)` es consecuencia lógica de las reglas.
  - `not(P)` falla $\Leftrightarrow$ P tiene (algún) éxito en tiempo finito.
- La negación por fallo es solamente una aproximación a la negación lógica, y tiene restricciones de uso.
- En particular, la negación **nunca instancia variables** del objetivo negado. Por ejemplo:

```
estudiante(pepe).   estudiante(rufo).   casado(pepe).
estudiante_soltero(X):- estudiante(X),not(casado(X)).
```

- ¿Qué ocurre si se define así?

```
estudiante_soltero(X):- not(casado(X)),estudiante(X).
```

Control en Prolog: disyunción e *if-then-else*

- En Prolog, los objetivos del cuerpo de una cláusula se separan por comas, lo que se interpreta como su **conjunción**.
- Además, se puede utilizar el punto y coma para representar la **disyunción** de objetivos.
- Suele ser útil para evitar el uso de predicados auxiliares, y hacer más compactos los predicados con varias cláusulas con la misma cabeza.
- Por ejemplo:

```
membercheck(X,[Y|Xs]):- ( X = Y, ! ; membercheck(X,Xs)).
```

- Además, se utiliza para la definición de estructuras *if-then-else*

Control en Prolog: disyunción e *if-then-else* (cont.)

- Se puede definir la estructura condicional
  Si P(X) entonces Q(X) si no R(X) con el predicado siguiente:

```
siPentQsinoR(X):- P(X), !, Q(X).
siPentQsinoR(X):- R(X).
```

- En Prolog se puede utilizar la siguiente notación dentro del cuerpo de una cláusula para representar un *if-then-else*:

```
P(X) -> Q(X) ; R(X)
```

- Ejemplo:

```
sumanum([],0).
sumanum([X|Xs],N):-
        ( number(X) ->
          sumanum(Xs,N1),
          N is X+N1
        ;
          sumanum(Xs,N)
        ).
```

Programación Lógica

1. Predicados aritméticos en Prolog.
2. Predicados metalógicos.
3. Control en Prolog: corte y negación.
4. **Orden superior. Predicados de agregación.**
5. Gestión de la base de reglas. Modificación dinámica del programa.
6. Precedencia de operadores (op/3). Creación de nuevos operadores.
7. Depurador de Prolog. El modelo "Byrd box".
8. Entrada/salida y manejo de ficheros.
9. Programación eficiente, listas en diferencias, DCGs.

Orden superior. Predicados de agregación

- El predicado básico de orden superior es la *metallamada*, que en Prolog es `call/1`.
- `call(X)` evalúa el término `X` que recibe como argumento como si fuera un objetivo, y unifica las variables que pudiera contener apropiadamente.
- `X` **debe** estar instanciado a un término que represente un objetivo existente. Por ejemplo:

```
q(a).                    p(X) :- call(X).
?- p(q(Y)).
Y = a
```

- **Ejercicio**: define un predicado que, dado el nombre de un predicado de aridad 1 y una lista, tenga éxito si la evaluación de ese predicado para todos los elementos de la lista tiene éxito.
- En algunos sistemas como SWI, se puede utilizar orden superior simplemente utilizando una variable como objetivo. Por ejemplo:

```
?- X = append(A,B,[a,b]), X.
```

Orden superior. Predicados de agregación (cont.)

- Otros predicados definidos en Prolog que implementan orden superior se utilizan para **recolectar las respuestas de objetivos**. Son los **predicados de agregación**
- Existen tres predicados de agregación: `findall/3`, `setof/3` y `bagof/3`.
- `findall/3` permite obtener **todas las respuestas de un objetivo** en una lista.
- `findall(Term, Objetivo, Lista)` proporciona en `Lista` todas las instancias de `Term` que satisfacen la evaluación de `Objetivo`.
- Por cada una de las respuestas de `Objetivo`, `findall` unifica el término `Term` con las variables del objetivo y lo añade a `Lista`.
- Las variables libres de `Objetivo` que no aparecen en `Term` se suponen cuantificadas existencialmente.

Orden superior. Predicados de agregación (cont.)

- Por ejemplo, el siguiente objetivo proporciona los resultados:

```
?- subterm(X,f(a,g(b),3)).
X = f(a, g(b), 3) ;
X = 3 ;
X = g(b) ;
X = b ;
X = a ;
false.
```

- Utilizando `findall/3`, se obtiene:

```
?- findall(X,subterm(X,f(a,g(b),3)),L).
L = [f(a, g(b), 3), 3, g(b), b, a].
```

- El término del primer argumento puede contener varias variables:

```
?- findall(par(X,Y),append(X,Y,[a,b,c]),L).
L = [par([],[a,b,c]), par([a],[b,c]), par([a,b],[c]),
par([a,b,c],[])].
```

- También se pueden utilizar objetivos compuestos:

```
?- findall(X,(member(X,[1,a,3,c]),member(X,[3,4,c,b])),L).
```

Orden superior. Predicados de agregación (cont.)

- `setof(Term,Objetivo,Lista)` proporciona en `Lista` la lista **ordenada y sin repeticiones** de todas las instancias de `Term` que satisfacen `Objetivo`.
- Si `Objetivo` contiene variables distintas de las que aparecen en `Term`, devuelve en *backtracking* tantos resultados como posibles instanciaciones existen para dichas variables.
- Las variables de `Objetivo` que no aparecen en `Term` se pueden cuantificar existencialmente mediante el operador ^.
- Ejemplos:

```
?- setof((X,Y),descendiente(X,Y),L).
L = [(alfonso, carmen), (alfonso, javier), (alfonso,
maria), (alfonso, pedro), (alicia, javier)...].
```

Orden superior. Predicados de agregación (cont.)

- Más ejemplos:

```
?- setof(X,descendiente(X,Y),L).
Y = carmen,
L = [alfonso, juan] ;
Y = javier,
L = [alfonso, alicia, juan, pedro, teresa]

?- setof(X,Y^descendiente(X,Y),L).
L = [alfonso, alicia, juan, pedro, teresa].
```

- `bagof(Term,Objetivo,Lista)` funciona de forma similar a findall, pero por defecto ninguna variable de `Objetivo` que no aparece en `Term` está cuantificada existencialmente:

```
p(a,c). p(b,e). p(a,b). p(b,d). p(a,b).
?- bagof(Y,p(X,Y),L).
L = [c,b,b],
X = a ? ;
L = [e,d],
X = b ?
```

Programación Lógica

1. Predicados aritméticos en Prolog.
2. Predicados metalógicos.
3. Control en Prolog: corte y negación.
4. Orden superior. Predicados de agregación.
5. **Gestión de la base de reglas. Modificación dinámica del programa.**
6. Precedencia de operadores (op/3). Creación de nuevos operadores.
7. Depurador de Prolog. El modelo "Byrd box".
8. Entrada/salida y manejo de ficheros.
9. Programación eficiente, listas en diferencias, DCGs.

Modificación dinámica del programa

- Prolog está relacionado con la inteligencia artificial entre otros motivos por la capacidad de los programas Prolog de modificarse a sí mismos
  - Si un programa es un conjunto de reglas que representan conocimiento, un programa puede **aprender** modificándose a sí mismo...
- Aunque la complejidad de muchas de estas ideas es mayor de la que se pensó en su momento, estas técnicas siguen siendo interesantes y útiles.
- En cualquier caso, la modificación dinámica de un programa **debe hacerse con cuidado**, pues los programas resultantes pueden ser muy difíciles de mantener.
- Pero en determinados casos es muy conveniente.

Modificación dinámica del programa

- Un programa Prolog es un conjunto de reglas (hechos, cláusulas). Estas reglas están almacenadas en la memoria de forma similar a los intérpretes que hemos visto en semanas anteriores.
- Existen predicados para modificar el conjunto de reglas del programa:
  - `asserta(C)` introduce la cláusula `C` **al principio** del conjunto de reglas del predicado correspondiente a `C`.
  - `assertz(C)` introduce la cláusula `C` **al final** del conjunto de reglas del predicado correspondiente a `C`.
  - `retract(C)` elimina **la primera cláusula** del predicado que unifica con `C`, unificando las variables de `C`. En *backtracking*, elimina las siguientes cláusulas del predicado.
  - `retractall(C)` elimina **todas las cláusulas** del predicado correspondiente a `C`.
- En cualquiera de estos predicados, `C` puede ser un **hecho** (nombre de predicado seguido de sus argumentos) o una **cláusula** (utilizando el operador `':-'`).
- Existen otros predicados (`recorda/1`, `abolish/1`,...).

Modificación dinámica del programa (cont.)

- **Ejemplos**:

?- asserta(p(a,b)).

```
true.
```

?- assertz(p(a,c)).

```
true.
```

?- asserta(p(A,C)).

```
true.
```

?- listing(p/2).

```
:- dynamic p/2.

p(_, _).
p(a, b).
p(a, c).

true.
```

?- retractall(p(a,b)).

```
true.
```

?- asserta(p(a,b)).

```
true.
```

?- assertz(q(r,s)).

```
true.
```

?- asserta(p(X,Y):-q(X,Y)).

```
true.
```

?- listing(p/2).

```
:- dynamic p/2.

p(A, B) :- q(A, B).
p(a, b).

true.
```

?- p(X,Y).

```
X = r, Y = s ;
X = a, Y = b.
```

Modificación dinámica del programa (cont.)

- Las cláusulas añadidas al programa tienen efecto en la **siguiente** ejecución del predicado que está siendo modificado:
- **Ejemplo**:

```
?- assertz((p(A):- assertz(p(A)),fail)).
true.
?- p(a).
false.
?- listing(p).
:- dynamic p/1.
p(A) :-
    assertz(p(A)),
    fail.
p(a).
true.
```

- **Ejercicio**: ¿Qué ocurre si evaluamos el siguiente objetivo?

```
?- p(X),p(b).
```

Modificar dinámicamente el programa puede hacerlo inmantenible.

Modificación dinámica del programa (cont.)

- Uno de los usos tradicionales de este tipo de técnicas es para **almacenar conocimiento** que ya ha sido calculado anteriormente.
  Por ejemplo, el predicado siguiente:

```
tabla(L) :-
    member(X,L), member(Y,L), V is X*Y,
    assertz(mult(X,Y,V)), fail.
```

- Si evaluamos el siguiente objetivo:

```
?- tabla([1,2,3,4,5,6,7,8,9]).
```

  ¿Cuál es el resultado?

Modificación dinámica del programa (cont.)

- Uno de los usos tradicionales de este tipo de técnicas es para **almacenar conocimiento** que ya ha sido calculado anteriormente. Por ejemplo, el predicado siguiente:

```
tabla(L) :-
    member(X,L), member(Y,L), V is X*Y,
    assertz(mult(X,Y,V)), fail.
```

- Si evaluamos el siguiente objetivo:

```
?- tabla([1,2,3,4,5,6,7,8,9]).
```

¿Cuál es el resultado?

- El objetivo falla, pero **como efecto lateral** obtenemos una colección de hechos *mult(X,Y,V)* que contienen la **tabla de multiplicar**.
- Esta técnica para recorrer las soluciones de un objetivo se denomina **bucle de fallo**.

Modificación dinámica del programa (cont.)

- Una versión modificada del predicado de cálculo de fibonacci:

```
:- dynamic tab_fib/2.
tab_fib(0,1).
tab_fib(1,1).

fibTabulado(N,F):- tab_fib(N,F), !.
fibTabulado(N,F):-
    N1 is N-1, N2 is N-2,
    fibTabulado(N1,F1), fibTabulado(N2,F2),
    F is F1+F2, assert(tab_fib(N,F)).
```

- La declaración `:- dynamic tab_fib/2.` indica que el predicado `tab_fib` de aridad 2 es **dinámico**: puede cambiar durante la ejecución (por medio de assert/retract).
- El predicado `fibTabulado(N,F)` calcula el valor del N-ésimo término de la serie utilizando tabulación o caching o memoization, es decir, utilizando la tabla `tab_fib`.

Modificación dinámica del programa (cont.)

- El problema de estas técnicas es que su uso produce **efectos laterales** y **globales**, que no desaparecen tras la resolución ni el *backtracking*.
- Los programas pueden no tener un sentido declarativo independiente de la ejecución.
- Se puede perder la "transparencia referencial": idénticas llamadas al mismo predicado pueden proporcionar resultados diferentes.
- Conclusión: **debe hacerse un uso muy controlado** de ellos, habitualmente en forma de hechos:
  - Para almacenar conocimiento obtenido previamente.
  - Para almacenar propiedades globales.
- **Ejercicio:** Diseña un predicado `contador(X)` que en sucesivas llamadas unifique el argumento con números naturales distintos comenzando en 1.
- **Ejercicio:** Diseña utilizando `assert` y `retract` un predicado `copy_term(X,Y)` que a partir de un término `X` proporcione otro término igual `Y` pero con todas las variables renombradas.

Programación Lógica

1. Predicados aritméticos en Prolog.
2. Predicados metalógicos.
3. Control en Prolog: corte y negación.
4. Orden superior. Predicados de agregación.
5. Gestión de la base de reglas. Modificación dinámica del programa.
6. **Precedencia de operadores (op/3). Creación de nuevos operadores.**
7. Depurador de Prolog. El modelo "Byrd box".
8. Entrada/salida y manejo de ficheros.
9. Programación eficiente, listas en diferencias, DCGs.

Precedencia de operadores

- Anteriormente hemos visto que en Prolog existen operadores predefinidos: `is/2`, `^/2`, los operadores aritméticos, etc.
- Si no existieran, el objetivo `X is 3*4+Y/2` se debería escribir `is(X,+(*(3,4),/(Y,2)))`.
- Los operadores tienen tres características fundamentales:
  - La **posición del operador**: prefija (por ejemplo, `-3`), infija (`2 + 3`), o postfija.
  - La **precedencia**: `2 + 3 * 4` se lee como `2 + (3 * 4)`.
  - La **asociatividad**: define cómo se interpretan expresiones como `8 - 5 - 2` (asociativo *por la derecha*: `8 - (5 - 2)`).
- Las reglas de precedencia se pueden ignorar utilizando paréntesis.

Precedencia de operadores

- En prolog se pueden utilizar nuevos operadores mediante la directiva `op/3`:

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

- **Nombre** es una constante Prolog.
- **Precedencia** es un número entre 0 y 1200 (los números más bajos representan **mayor** precedencia).
- **PosAsoc** es una constante de las siguientes:

`xf, yf`	posición postfija
`fx, fy`	posición prefija
`xfx, xfy, yfx, yfy`	posición infija

  - `f` representa la posición del operador,
  - en `x` solo pueden aparecer otros operadores con *menor* precedencia que `f`.
  - en `y` pueden aparecer operadores con precedencia *menor o igual* a `f` (para la asociatividad)

Precedencia de operadores (cont.)

- Algunos de los operadores estándar de SWI-Prolog:

Prec	PosAsoc	Operadores
1200	xfx	`-->, :-`
1200	fx	`:-, ?-`
1150	fx	`dynamic, discontiguous`
1100	xfy	`;, |`
1050	xfy	`->`
1000	xfy	`,`
900	fy	`\+`
700	xfx	`<, =, =.., =@=, =:=, =<, ==, =\=, >, >=, @<, @=<, @>, @>=, \=, \==, is`
600	xfy	`:`
500	yfx	`+, -, /\, \/, xor`
500	fx	`?`
400	yfx	`*, /, //, rdiv, <<, >>, mod, rem`
200	xfx	`**`
200	xfy	`^`
200	fy	`+, -, \`

- `':-'` y la coma también son operadores. ¿Qué efecto tiene esto sobre el if-then-else (`A -> B ; C`)?

Creación de nuevos operadores

- Por ejemplo, se podrían definir operadores para cambiar la sintaxis de if-then-else:

```
:- op(1000,fx,if).
:- op(1050,xfy,then).
:- op(1100,xfy,else).

else(if Condicion then Then,Else):-
     call(Condicion) -> call(Then) ; call(Else).
```

- Con esta definición se pueden definir predicados como:

```
P(X,Y) :- ..., (if X > 4 then Y = 10 else Y = 20).
```

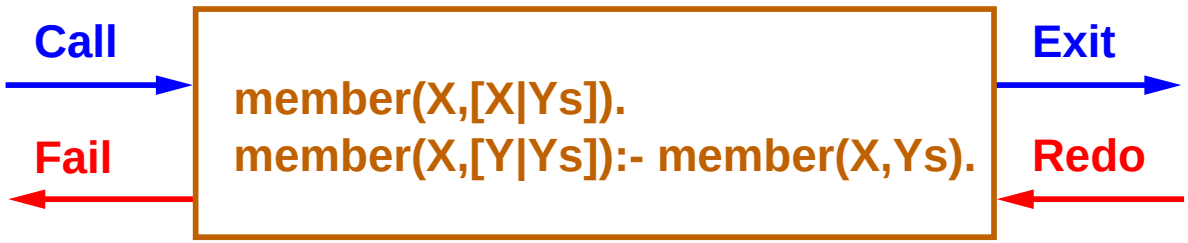
- **Ejercicio:** Define un operador infijo `nand/2` y el predicado correspondiente que realice la operación lógica con el mismo nombre sobre objetivos Prolog, y que permita utilizar objetivos compuestos separados por comas en sus operandos sin necesidad de paréntesis, pero no disyunciones (que utilizan ';').

Programación Lógica

1. Predicados aritméticos en Prolog.
2. Predicados metalógicos.
3. Control en Prolog: corte y negación.
4. Orden superior. Predicados de agregación.
5. Gestión de la base de reglas. Modificación dinámica del programa.
6. Precedencia de operadores (op/3). Creación de nuevos operadores.
7. **Depurador de Prolog. El modelo "Byrd box".**
8. Entrada/salida y manejo de ficheros.
9. Programación eficiente, listas en diferencias, DCGs.

El depurador de Prolog. el modelo "Byrd Box"

- Todos los sistemas Prolog tienen un depurador de línea de comandos, y en algunos casos un depurador de código fuente.
- El modelo de ejecución de Prolog es más complejo que en otros lenguajes (por ejemplo, imperativos) debido al *backtracking*.
  - En un lenguaje imperativo, la llamada a un procedimiento se puede ver como una caja negra que recibe una entrada y produce una salida.
  - En Prolog la llamada a un predicado se puede ver como una caja negra con **dos entradas y dos salidas**:



  - por ejemplo, podemos observar el comportamiento de `esta/2`.

El depurador de Prolog. el modelo "Byrd Box" (cont.)

Ejemplo:

```
esta(X,[X|_]).      esta(X,[_|Xs]):- esta(X,Xs).
```

```
[debug] ?- trace.
Unknown message: query(yes)
[trace] ?- esta(X,[a,b,c]).
   Call: (7) esta(_G310, [a, b, c]) ?
   Exit: (7) esta(a, [a, b, c]) ?
X = a ;
   Redo: (7) esta(_G310, [a, b, c]) ?
   Call: (8) esta(_G310, [b, c]) ?
   Exit: (8) esta(b, [b, c]) ?
   Exit: (7) esta(b, [a, b, c]) ?
X = b ;
   Redo: (8) esta(_G310, [b, c]) ?
   Call: (9) esta(_G310, [c]) ?
   Exit: (9) esta(c, [c]) ?
   Exit: (8) esta(c, [b, c]) ?
   Exit: (7) esta(c, [a, b, c]) ?
X = c ;
   Redo: (9) esta(_G310, [c]) ?
   Call: (10) esta(_G310, []) ?
   Fail: (10) esta(_G310, []) ?
false.
```

El depurador de Prolog. el modelo "Byrd Box" (cont.)

Otro ejemplo: `p(A,B,C,X):- esta(X,A),esta(X,B),esta(X,C).`

```
?- trace.
Unknown message: query(yes)
[trace] ?- p([0,2,4],[4,2],[8,2],X).
   Call: (7) p([0, 2, 4], [4, 2], [8, 2], _G325) ?
   Call: (8) esta(_G325, [0, 2, 4]) ?
   Exit: (8) esta(0, [0, 2, 4]) ?
```

En este punto se ha obtenido un resultado para la primera llamada a `esta/2`.

```
   Call: (8) esta(0, [4, 2]) ? s
```
(se omite la traza de esta llamada)
```
   Fail: (8) esta(0, [4, 2]) ?
```

La segunda llamada falla, pues 0 no está en la lista `[4,2]`. Se realiza *backtracking sobre el primer objetivo*:

```
   Redo: (8) esta(_G325, [0, 2, 4]) ? s
```
(se omite la traza de esta llamada)
```
   Exit: (8) esta(2, [0, 2, 4]) ?
```

Se vuelve a evaluar el segundo objetivo

```
   Call: (8) esta(2, [4, 2]) ? s
```
(se omite la traza de esta llamada)
```
   Exit: (8) esta(2, [4, 2]) ?
```

El segundo objetivo tiene éxito, se pasa a evaluar el tercer objetivo:

```
   Call: (8) esta(2, [8, 2]) ? s
```
(se omite la traza de esta llamada)
```
   Exit: (8) esta(2, [8, 2]) ?
   Exit: (7) p([0, 2, 4], [4, 2], [8, 2], 2) ?
X = 2 ;
   Redo: (8) esta(2, [8, 2]) ? ...
```

El depurador de Prolog. el modelo "Byrd Box" (cont.)

- Las opciones más comunes de depuración son las siguientes:

`h`	Ayuda — muestra esta lista, posiblemente con más opciones
`c`	Ejecuta paso a paso, avanzando la ejecución al siguiente call/exit/redo/fail
`intro`	lo mismo que `c`
`s`	Omite la traza de la ejecución del objetivo actual
`l`	Omite la traza hasta el siguiente "spypoint"
`f`	Hace fallar el objetivo actual, forzando el *backtracking*
`a`	Aborta la ejecución en curso
`r`	Vuelve a evaluar el objetivo actual (normalmente después de un fallo o una terminación con éxito)
`b`	Break — invoca un *top-level* recursivo (que se finaliza con ctrl+d)

El depurador de Prolog. el modelo "Byrd Box" (cont.)

- Los ejemplos y opciones anteriores se consideran para el modo de depuración paso a paso (`trace`).
- A diferencia del modo `trace`, el modo `debug` permite parar solamente en los *spypoints*.
- Se puede rastrear el funcionamiento de un predicado con `spy`: El objetivo `spy(esta/2)` coloca un *spypoint* en el predicado `esta/2`, de forma que se parará la ejecución en cuanto se invoque este predicado.
- Se pueden eliminar los *spypoints* con `nospy/1` y `nospyall/0`.
- Los modos de depuración se pueden activar y desactivar utilizando `trace/0, notrace/0, debug/0, nodebug/0`.

1. Predicados aritméticos en Prolog.
2. Predicados metalógicos.
3. Control en Prolog: corte y negación.
4. Orden superior. Predicados de agregación.
5. Gestión de la base de reglas. Modificación dinámica del programa.
6. Precedencia de operadores (op/3). Creación de nuevos operadores.
7. Depurador de Prolog. El modelo "Byrd box".
8. **Entrada/salida y manejo de ficheros.**
9. Programación eficiente, listas en diferencias, DCGs.

Entrada/salida y manejo de ficheros

- Como todos los lenguajes de programación, Prolog tiene diversos predicados para realizar operaciones de entrada/salida y de manejo de ficheros.
- Algunos predicados básicos de entrada son:
  - `get_code(N)` obtiene el siguiente carácter de la entrada estándar y unifica su código ASCII con `N`.
  - `read(Term)` lee **un término Prolog** de la entrada estándar (terminado con un punto + intro) y lo unifica con `Term`.
    - ★ Este predicado realiza el análisis sintáctico de lo leído por la entrada estándar para poder unificar el término Prolog con `Term`.
    - ★ Este predicado **no lee términos en backtracking**: hay que llamarlo de nuevo para leer el siguiente término.
- **Ejercicio:** Diseña un predicado que lea un término Prolog de la forma `dato(X)` y lo añada a la definición del predicado `dato/1` en el programa.
- **Ejercicio:** Diseña un predicado que lea **términos Prolog** de la forma `dato(X)` y los añada a la definición del predicado `dato/1` en el programa. Este predicado debe terminar cuando lea la constante `fin`.

Entrada/salida y manejo de ficheros (cont.)

- Algunos predicados básicos de salida son los siguientes:
  - `display(Term)` escribe el término `Term` en la salida estándar, ignorando las definiciones de operadores. Por ejemplo: `display('a b' + 'c d')` escribe `+(a b, c d)`.
  - `write(Term)` escribe `Term` en la salida estándar utilizando las definiciones de operadores. Ejemplo: `write('a b' + 'c d')` escribe `a b+c d)`.
  - `displayq(Term)` es similar a `display/1`, pero escribe `Term` añadiendo comillas simples si es necesario para posteriormente otro programa Prolog pueda parsear `Term`. Ejemplo: `displayq('a b' + 'c d')` escribe `+('a b', 'c d')`.
  - `writeq(Term)` es similar a `write/1`, pero añade comillas si es necesario. Ej.: `writeq('a b' + 'c d')` escribe `'a b'+'c d'`.
  - `nl/0` escribe un salto de línea.
- **Ejercicio**: Diseña un predicado `listar/1` que reciba un nombre de predicado **que solo contiene hechos** y su aridad (ej. `dato/1`) y muestre por pantalla todos los hechos de este predicado. Puedes utilizar un *bucle de fallo*.

Manejo de ficheros

- Los predicados clásicos de entrada/salida permiten modificar la entrada/salida estándar para leer/escribir sobre un fichero en lugar del terminal.
- **Entrada:**
  - `see(Archivo)` hace que `Archivo` sea la entrada estándar. Desde ese momento los predicados de lectura leen de este archivo.
  - `seeing(Archivo)` unifica `Archivo` con la entrada estándar que está siendo utilizada (`user` por defecto).
  - `seen/0` cierra la entrada estándar y deja la que estuviera anteriormente.
- **Salida:**
  - `tell(Archivo)` hace que `Archivo` sea la salida estándar. Desde este momento los predicados de escritura escriben en este archivo.
  - `telling(Archivo)` unifica `Archivo` con la salida estándar que está siendo utilizada (`user` por defecto).
  - `told/0` cierra la entrada estándar y deja la que estuviera anteriormente.

Manejo de ficheros (cont.)

- También se pueden abrir ficheros como en otros lenguajes de programación, obteniendo un identificador del *stream* para utilizarlo en los predicados de lectura/escritura:
  - `open(+F,+Modo,?S)` abre el fichero con nombre `F` y unifica `S` con el *stream* asociado. `Modo` puede ser `read`, `write` o `append`.
  - `close(+S)` cierra el fichero con *stream* `S`.
  - Las operaciones de lectura/escritura se pueden realizar con los predicados anteriores, añadiendo el *stream* como primer argumento: `read(S,Term)`, `write(S,Term)`, `nl(S)`, etc.
- **Ejercicio**: Diseña un predicado `imprime_tabla(L,F)` que escriba en el fichero `F` la tabla de multiplicar de los números que están en la lista `L`. Puedes utilizar un *bucle de fallo*.
- Existen otros muchos predicados para entrada/salida sobre ficheros (y sobre *sockets*), documentados en el manual del sistema Prolog (`http://www.swi-prolog.org/pldoc/index.html`).

Programación Lógica

1. Predicados aritméticos en Prolog.
2. Predicados metalógicos.
3. Control en Prolog: corte y negación.
4. Orden superior. Predicados de agregación.
5. Gestión de la base de reglas. Modificación dinámica del programa.
6. Precedencia de operadores (op/3). Creación de nuevos operadores.
7. Depurador de Prolog. El modelo "Byrd box".
8. Entrada/salida y manejo de ficheros.
9. **Programación eficiente, listas en diferencias, DCGs.**