Programación Declarativa:
Lógica y Restricciones

Conceptos Básicos de la Programación Lógica

Mari Carmen Suárez de Figueroa Baonza
mcsuarez@fi.upm.es

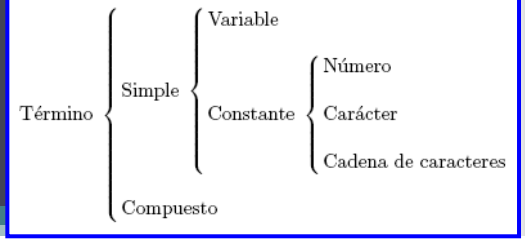
Recordatorio

- Vamos a usar la Lógica como sistema de especificación y como lenguaje de programación
- Hay que pensar de forma declarativa
- La idea esencial de la Programación Lógica es
  - Programa= lógica + control
    - 2 componentes independientes
    - **Lógica** (programador): hechos y reglas para representar conocimiento
    - **Control** (interprete): deducción lógica para dar respuestas (soluciones)
      - Se puede proporcionar de manera efectiva a través del ordenamiento de los literales en las cláusulas
      - Normalmente no hay que preocuparse del control gracias a la resolución

Contenidos

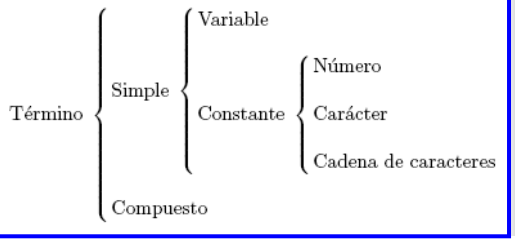
- Sintaxis: Datos
- Manipulación de Datos: Unificación
- Sintaxis: Código
- Semántica: Significado de Programas
- Ejecución de Programas Lógicos

Sintaxis: Términos (I)



- **Variables**: comienzan con mayúsculas (o "_"), pueden incluir "_" y dígitos
  - Ejemplos: X, Lm4u, Un_pequeño_edificio, _, _x, _22
- **Estructuras**: son términos compuestos formados por un *constructor* (el nombre de la estructura) seguido por un número fijo de *argumentos* entre paréntesis
  - *Argumentos*: pueden ser variables, constantes y estructuras
  - Ejemplos: fecha(lunes, Mes, 2014)
- *Constructores* (o *funtores*): comienzan con minúsculas, pueden incluir "_" y dígitos
  - También pueden incluir algunos caracteres especiales. Y entre comillas, cualquier caracter
  - Ejemplos: a, perro, un_gato, x22, 'Hungry man', [], *, >, 'Doesn''t matter'

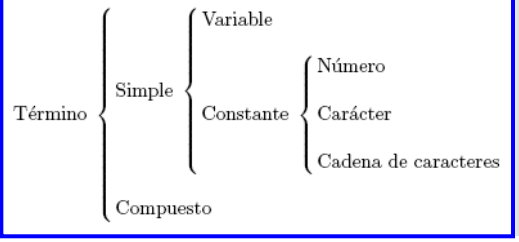
Sintaxis: Términos (II)



- **Constantes**: son estructuras sin argumentos (sólo el nombre) y también números
  - No numéricas: a, b, ana, estudiante
  - Numéricas: 0, 999, -77, 5.23, 0.23e-5, 0.23E-5
- **Aridad**: es el número de argumentos de una estructura

  Los constructores se representan de la siguiente manera: *nombre/aridad* (por ejemplo, fecha/3)
  - Una constante se puede ver como una estructura de aridad cero

Sintaxis: Términos (III)



- Variables, constantes, y estructuras son ***términos*** (son los términos de un lenguaje de primer orden)
- Los términos son las estructuras de datos de un programa lógico

Term	*Type*	*Constructor*
dad	constant	dad/0
time(min, sec)	structure	time/2
pair(Calvin, tiger(Hobbes))	structure	pair/2
Tee(Alf, rob)	illegal	—
A_good_time	variable	—

- Una variable es ***libre*** si no le ha sido asignado ningún valor
- Un término es ***cerrado*** (*ground*) si no contiene variables libres

Manipulación de Estructuras de Datos: Unificación

- La **unificación** es el único mecanismo disponible en programas lógicos para manipular estructuras de datos
- La **unificación** se usa para:
  - Pasar parámetros
  - Devolver valores
  - Acceder a partes de estructuras
  - Dar valor a variables
- La **unificación** es un procedimiento para resolver ecuaciones en estructuras de datos
  - Devuelve una solución mínima a la ecuación (o el sistema de ecuaciones)
  - Como muchos procedimientos de resolución de ecuaciones se basa en el aislamiento de las variables y en la posterior sustitución por sus valores

Unificación (I)

- La **unificación** de dos términos A y B está preguntando si se pueden hacer sintácticamente idénticos dando valores (mínimos) a sus variables
  - Es decir, la idea es encontrar una solución a la ecuación A = B
    - Si no es posible encontrarla, entonces la unificación falla
  - Es importante tener en cuenta que sólo se puede dar valor a las variables
  - Dos estructuras se pueden hacer idénticas sólo haciendo sus argumentos idénticos
- El proceso de **unificación** determina las condiciones y posibilidades de sustitución

Unificación (II)

- Un proceso de **unificación** se puede representar mediante un operador θ constituido por un conjunto de pares de términos
  - $\theta = \{ v1 \rightarrow t1, v2 \rightarrow t2, ..., vn \rightarrow tn\}$
    - donde el par $vi \rightarrow ti$, indica que la variable vi es sustituida por el término ti
    - Nota: Las siguientes notaciones equivalentes también son habituales
      - $\theta = \{ v1/t1, v2/t2, ..., vn/tn\}$
      - $\theta = \{ v1=t1, v2=t2, ..., vn=tn\}$
- El proceso de **unificación** implica la sustitución de una variable por otro término que puede ser variable, constante o funtor. En este último caso, el funtor no puede contener la variable sustituida
  - Es evidente que no siempre es posible unificar dos realizaciones de un mismo predicado

Algoritmo de Unificación

Sean A y B dos términos:

- 1. $\theta = \varnothing$, E = {A = B}
- 2. Mientras no E = $\varnothing$:
  - 2.1. Eliminar una ecuación T = S de E
  - 2.2. En caso de que T o S (o ambas) sean variables (distintas) [suponiendo la variable T]:
    - (*occur check*) Si T ocurre en el término S, entonces parada con fallo
    - Sustituir la variable T por el término S en todos los términos de θ
    - Sustituir la variable T por el término S en todos los términos de E
    - Añadir T = S a θ
  - 2.3. En caso de que T y S no sean variables:
    - Si sus nombres o aridades son diferentes, entonces parada con fallo
    - Obtener los argumentos {T1, . . . , Tn} de T y {S1, . . . , Sn} de S
    - Añadir {T1 = S1, . . . , Tn = Sn} a E
- 3. Parada con θ siendo el m.g.u de A y B

Algoritmo de Unificación: Ejemplos (I)

- Ejemplos: Unificar
  - A = p(X,X) y B = p(f(Z),f(W))
  - A = p(X,f(Y)) y B = p(Z,X)

- Unify: A = `p(X,X)` and B = `p(f(Z),f(W))`

θ	E	T	S
{}	{ p(X,X)=p(f(Z),f(W)) }	p(X,X)	p(f(Z),f(W))
{}	{ X=f(Z), X=f(W) }	X	f(Z)
{ X=f(Z) }	{ f(Z)=f(W) }	f(Z)	f(W)
{ X=f(Z) }	{ Z=W }	Z	W
{ X=f(W), Z=W }	{}		

- Unify: A = `p(X,f(Y))` and B = `p(Z,X)`

θ	E	T	S
{}	{ p(X,f(Y))=p(Z,X) }	p(X,f(Y))	p(Z,X)
{}	{ X=Z, f(Y)=X }	X	Z
{ X=Z }	{ f(Y)=Z }	f(Y)	Z
{ X=f(Y), Z=f(Y) }	{}		

Algoritmo de Unificación: Ejemplos (II)

- Ejemplos: Unificar
  - A = p(X,f(Y)) y B = p(a,g(b))
  - A = p(X,f(X)) y B = p(Z,Z)

- Unify: A = `p(X,f(Y))` and B = `p(a,g(b))`

θ	E	T	S
{}	{ p(X,f(Y))=p(a,g(b)) }	p(X,f(Y))	p(a,g(b))
{}	{ X=a, f(Y)=g(b) }	X	a
{ X=a }	{ f(Y)=g(b) }	f(Y)	g(b)
fail			

- Unify: A = `p(X,f(X))` and B = `p(Z,Z)`

θ	E	T	S
{}	{ p(X,f(X))=p(Z,Z) }	p(X,f(X))	p(Z,Z)
{}	{ X=Z, f(X)=Z }	X	Z
{ X=Z }	{ f(Z)=Z }	f(Z)	Z
fail			

Unificación: Ejemplos (I)

- Ejemplos: Definir las sustituciones necesarias para unificar el predicado p(X, f(Y), b) con cada uno de los siguientes predicados:
  - P1: p(g(Z), f(a), b) $\theta_1 = \{ X \rightarrow g(Z), Y \rightarrow a \}$
  - P2: p(X, f(a), b) $\theta_2 = \{ Y \rightarrow a \}$
  - P3: p(Z, f(U), b) $\theta_3 = \{ X \rightarrow Z, Y \rightarrow U \}$
  - P4: p(c, f(a), b) $\theta_4 = \{ X \rightarrow c, Y \rightarrow a \}$

Unificación: Ejemplos (II)

- Ejemplos:

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

  - (1) Estructuras con diferente nombre y/o aridad no se pueden unificar
  - (2) A una variable no se le puede dar como valor un término que contiene esa variable, porque crearía un término infinito
    - Esto se conoce como la verificación de ocurrencias (*occurs check*)

Sintaxis: Literales (I)

- **Literales**: se componen de un nombre de predicado (como un funtor) seguido de un número fijo de argumentos entre paréntesis
  - Ejemplo: llegar (pepe, fecha (lunes, Month, 2016))
  - Los argumentos son términos
  - El número de argumentos es la aridad del predicado
  - Los nombres completos de los predicados se denotan como nombre/aridad (por ejemplo, llegar/2)
- Literales y términos son sintácticamente identicos. Sólo se distinguen por su contexto
  - Ejemplo: Si perro(nombre(pluto), color(negro)) es un literal, entonces nombre(pluto) y color(negro) son términos
  - Ejemplo: Si color(perro(pluto,negro)) es un literal, entonces perro(pluto,negro) es un término

Sintaxis: Literales (II)

- Literales y términos son sintácticamente identicos. Sólo se distinguen por su contexto
  - Los **literales** se usan para definir procedimientos y llamadas a procedimientos
  - Los **términos** son estructuras de datos, y por tanto, argumentos de los literales

Sintaxis: Operadores

- Funtores y nombres de predicados se pueden definir como operadores prefijos, postfijos o infijos (*nota*: esto es sólo sintaxis)
  - **Prefijo**: el operador va delante de sus argumentos
  - **Infijo**: el operador se escribe entre los argumentos
  - **Postfijo**: el operador se escribe detrás de sus argumentos

  - Ejemplos:
    - - b es el término -(b), si -/1 se declara prefijo
    - a + b es el término +(a,b), si +/2 se declara infijo
    - juan padre maria es el término padre(juan,maria), si padre/2 se declara infijo
    - a b < es el término <(a,b), si </2 se declara postfijo

Sintaxis: Reglas y Hechos (I)

- **Reglas**: son implicaciones o inferencias lógicas que permiten deducir nuevo conocimiento
  - Las reglas permiten definir nuevas relaciones a partir de otras ya existentes
  - Son expresiones de la forma

$$p_0(t_1, t_2, \ldots, t_{n_0}) :- \\ p_1(t_1^1, t_2^1, \ldots, t_{n_1}^1), \\ \ldots \\ p_m(t_1^m, t_2^m, \ldots, t_{n_m}^m).$$

    - p0(...), …, pm(...) son literales
    - p0(...) se denomina **cabeza** (***head***) de la regla
    - Los pi a la derecha del símbolo ':-' se llaman **metas** (***goals***) y forman el **cuerpo** (***body***) de la regla
      - También se denominan **llamadas a procedimiento** (***procedure calls***)
    - Normalmente el símbolo ':-' se denomina **cuello** (***neck***) de la regla
  - Ejemplo:
    - mortal(X) :– humano(X).
      - X es mortal si X es humano

Sintaxis: Reglas y Hechos (II)

- **Hechos**: son declaraciones, cláusulas o proposiciones
  - Los hechos establecen una relación entre objetos y es la forma más sencilla de sentencia
  - Son expresiones de la forma
    - p(t1, t2, . . . , tn).
    - Es decir, una regla con el cuerpo vacío
  - Ejemplos:
    - humano (Sócrates). [Sócrates es humano]
    - ama (Juan, María). [Juan ama a María]

Sintaxis: Cláusulas (I)

- Reglas y hechos se denominan **cláusulas** (*clauses*)
  - Son cláusulas en lógica de primer orden
- Las **cláusulas** forman el código de un programa lógico
  - Ejemplo*:*

```
meal(soup, beef, coffee).
meal(First, Second, Third) :-
    appetizer(First),
    main_dish(Second),
    dessert(Third).
```

  - Nota:
    - :- es equivalente a ← (la implicación lógica, pero escrita al revés)
    - La coma es equivalente a la conjunción

Sintaxis: Cláusulas (II)

- Ejemplo*:*

```
meal(soup, beef, coffee).
meal(First, Second, Third) :-
    appetizer(First),
    main_dish(Second),
    dessert(Third).
```

  - La regla significa:
    - appetizer(First) $\wedge$ main_dish(Second) $\wedge$ dessert(Third) $\rightarrow$ meal(First, Second, Third)
  - Y por tanto, la claúsula de Horn es de la forma:
    - $\neg$appetizer(First) $\vee$ $\neg$main_dish(Second) $\vee$ $\neg$dessert(Third) $\vee$ meal(First, Second, Third)

Sintaxis: Predicados (I)

- **Predicados** (o definiciones de procedimiento): son conjuntos de cláusulas cuyas cabezas tienen el mismo nombre y aridad (el *nombre del predicado*)
  - Ejemplo:

```
pet(barry).                          animal(tim).
pet(X) :- animal(X), barks(X).       animal(spot).
pet(X) :- animal(X), meows(X).       animal(hobbes).
```

    - El predicado pet/1 tiene 3 cláusulas: 1 es un hecho y 2 son reglas
    - El predicado animal/1 tiene 3 cláusulas, que son hechos
  - Nota (*Alcance de las variables*): las variables X en las dos cláusulas anteriores son diferentes, a pesar de tener el mismo nombre
    - Las variables son locales a las cláusulas (y se rebautizan en el momento en el que la cláusula se utiliza (como sucede con las variables locales a un procedimiento en lenguajes convencionales))

Sintaxis: Predicados (II)

- **Predicados** (o definiciones de procedimiento): establecen relaciones
  - Cada predicado tiene un funtor (nombre/aridad) que lo identifica de manera unívoca
    - Ejemplos: "X es padre de Y" **padre(X,Y)** **padre/2**
      "la pila P está vacía" **esta_vacia(P)** **esta_vacia/1**
  - Los procedimientos pueden satisfacerse o fracasar
    - Ejemplos: padre(antonio,eva) éxito
      padre(eva,eva) fracaso
  - Los procedimientos no son funciones: no devuelven valores
    - Ejemplo: padre(antonio,padre(carlos,silvia)) error
      No expresa que 'antonio' es abuelo de 'silvia'
  - Las llamadas anidadas no se ejecutan: son argumentos (datos)

Sintaxis: Programas

- **Programas Lógicos**: son conjuntos de predicados
  - Ejemplo:

    padre(a,b).

    madre(b,c).

    abuelo(X,Z) ← progenitor(X,Y), progenitor(Y,Z).

    progenitor(X,Y) ← padre(X,Y).

    progenitor(X,Y) ← madre(X,Y).

Significado Declarativo de Hechos y Reglas

- El **significado declarativo de las cláusulas** es el correspondiente al de la lógica de primer orden, de acuerdo con ciertas convenciones
  - **Hechos**: establecen cosas que son verdad
    - Un hecho "p." se puede ver como la regla "p ← true."
    - Ejemplo: el hecho "*animal(luna).*" se puede leer como "luna es un animal"
  - **Reglas**: establecen implicaciones que son verdad
    - p :- p1, …, pm. representa p1 ∧ … ∧ pm → p.
    - Por tanto, una regla como "p :- p1, …, pm." significa "si p1 y ... y pm son verdad, entonces p es verdad"
    - Ejemplo: la regla "*mascota(X) :- animal(X), maulla(X).*" se puede leer como "X es una mascota si es un animal y maulla"

Significado Declarativo de Predicados

- **Predicados**: las cláusulas de un mismo predicado proporcionan diferentes alternativas
  - p :- p1, ..., pn.
  - p :- q1, ..., qm.
  - etc.
  - Ejemplo: Las siguientes reglas expresan dos formas en las que X se puede considerar una mascota
    - pet(X) :- animal(X), barks(X).
    - pet(X) :- animal(X), meows(X).

Significado Declarativo de Programas

- **Programas**: son conjuntos de fórmulas lógicas (es decir, una teoría de primer orden)
  - Un programa es un conjunto de declaraciones que se asumen verdaderas
    - Por tanto, un conjunto de cláusulas de Horn
  - El significado declarativo de un programa es el conjunto de todos los hechos cerrados (*ground*) que se pueden deducir lógicamente de él

Consultas (*Queries*) (I)

- **Consultas** (*Queries*): son expresiones de la forma:

$$?- \; p_1(t_1^1, \ldots, t_{n_1}^1), \ldots, p_n(t_1^n, \ldots, t_{n_m}^n).$$

  Es decir, son cláusulas sin cabeza

  En este caso, ?- significa también ←

  - Los pi a la derecha de ?- se denominan **metas** (*goals* o *procedure calls*)
  - A veces, la consulta completa también se denomina meta u objetivo (complejo)

- Una **consulta** es una cláusula que queremos deducir
  - Ejemplo: ?- pet(X).

    Se puede ver como "true ← pet(X)", es decir, "¬pet(X)"

Consultas (*Queries*) (II)

- Una **consulta** representa una pregunta al programa
  - Ejemplo*:* ?- pet(spot).

    Pregunta si spot es una mascota
  - Ejemplo*:* ?- pet(X).

    Pregunta si hay (existe) algún X que sea una mascota
- Una **consulta** especifica el problema, la proposición a demostrar o el objetivo buscado
  - Ejemplo: Si sabemos que los humanos son mortales y que Sócrates es humano, entonces ¿podemos deducir que Sócrates es mortal?
    - mortal(X) :– humano(X). [Los humanos son mortales] *Regla*
    - humano(socrates). [Sócrates es humano] *Hecho*
    - ¿Sócrates es mortal? sería la *Consulta*
      - *?- mortal(socrates).*

Consultas (*Queries*) (III)

- Ejemplos: Dados los predicados factorial/2 y suma/3
  - ?- factorial(s(s(s(0))), Fact3).
  - ?- factorial(s(s(0)),s(s(0))).
  - ?- factorial(0,Z), factorial(s(s(0)),Z).
  - ?- suma(s(0), s(0), X).
  - ?- suma(X, Y, s(s(0))).
  - ?- suma(s(0), Y, s(s(s(0)))).

Consultas (*Queries*) (IV)

- **Consultas (Objetivos) sin variables**: cuestionan si ciertos objetos concretos satisfacen ciertas relaciones
  - La respuesta debe ser sí (éxito) o no (fracaso)
  - Ejemplos:
    - ¿es Antonio padre de Carlos? **?- padre(antonio,carlos).**
    - ¿es María antepasado de Silvia? **?- antepasado(maria,silvia).**
    - ¿son Antonio y María antepasados de Silvia?
      ?- antepasado(maria,silvia),antepasado(antonio,silvia).
- **Consultas (Objetivos) con variables**: cuestionan qué objetos satisfacen ciertas relaciones
  - El objetivo se puede interpretar como una ecuación y sus variables como incógnitas a determinar
  - Ejemplos:
    - ¿quién es el padre de Silvia? **?- padre(X,silvia).**
    - ¿quiénes son los descendientes de María? **?- antepasado(maria,X).**
    - ¿quiénes son hermanos? **?- hermanos(X,Y).**

¿Cómo se escribe un programa lógico?

- Para escribir un **programa lógico** debemos identificar:
  - qué objetos intervienen en el problema,
  - cuáles son las relaciones entre éstos, y
  - qué objetivos queremos alcanzar
- Una vez identificados los elementos anteriores debemos:
  - representar los objetos mediante términos
  - definir las relaciones mediante hechos y reglas
  - definir los objetivos mediante consultas

Ejemplo de Programa Lógico

```
:- module(_,_,['bf/af']).

nat(0) <- .
nat(s(X)) <- nat(X).

le(0,_X) <- .
le(s(X),s(Y)) <- le(X,Y).

add(0,Y,Y) <- nat(Y).
add(s(X),Y,s(Z)) <- add(X,Y,Z).

mult(0,Y,0) <- nat(Y).
mult(s(X),Y,Z) <- add(W,Y,Z), mult(X,Y,W).

nat_square(X,Y) <- nat(X), nat(Y), mult(X,X,Y)

output(X) <- nat(Y), le(Y,s(s(s(s(s(0))))), na
```

Hecho

Regla

Predicado (add/3)

Query	Answer
?- nat(s(0)).	yes
?- add(s(0),s(s(0)),X).	X = s(s(s(0)))
?- add(s(0),X,s(s(s(0)))).	X = s(s(0))
?- nat(X).	X = 0 ; X = s(0) ; X = s(s(0)) ; ...
?- add(X,Y,s(0)).	(X = 0 , Y=s(0)) ; (X = s(0) , Y = 0)
?- nat_square(s(s(0)), X).	X = s(s(s(s(0))))
?- nat_square(X,s(s(s(s(0))))).	X = s(s(0))
?- nat_square(X,Y).	(X = 0 , Y=0) ; (X = s(0) , Y=s(0)) ; (X = s(s(0)) , Y=s(s(s(s(0))))) ; ...
?- output(X).	X = 0 ; X = s(0) ; X = s(s(s(s(0)))) ; ...

Ejecución (I)

- **Ejecución**: dado un programa (lógico) y una consulta (*query*), ejecutar el programa (lógico) consiste en encontrar una respuesta a la consulta
  - La **ejecución** de un programa lógico con un objetivo dado consiste en
    - determinar si el objetivo es deducible del programa,
    - y en caso de que lo sea, determinar los valores de las variables del objetivo que dan una respuesta al mismo

Ejecución (II)

- Ejemplo: Dado el siguiente programa lógico y la consulta *"?- pet(X)."*
  - animal(tim).
  - animal(spot).
  - animal(hobbes).
  - pet(X) :- animal(X), barks(X).
  - pet(X) :- animal(X), meows(X).

  meows(tim).

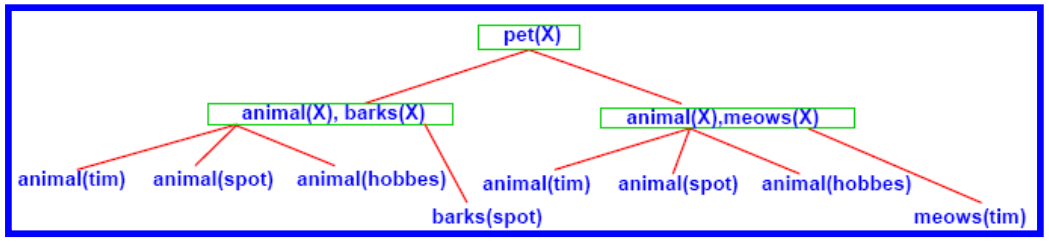
  barks(spot).

  roars(hobbes).

  el sistema intenta encontrar una "solución" para X que haga cierto pet(X)
  - Esto se puede hacer de diferentes modos:
    - Ver el programa como un conjunto de fórmulas y aplicar la deducción
    - Ver el programa como un conjunto de cláusulas y aplicar resolución SLD
    - Ver el programa como un conjunto de definiciones de procedimientos y ejecutar las llamadas a procedimientos correspondientes a las consultas

El Árbol de Búsqueda (I)

- Una *consulta* + un *programa lógico* especifican un **árbol de búsqueda**
- Ejemplo: La consulta *"?- pet(X)."* con el programa lógico dado genera el siguiente árbol de búsqueda



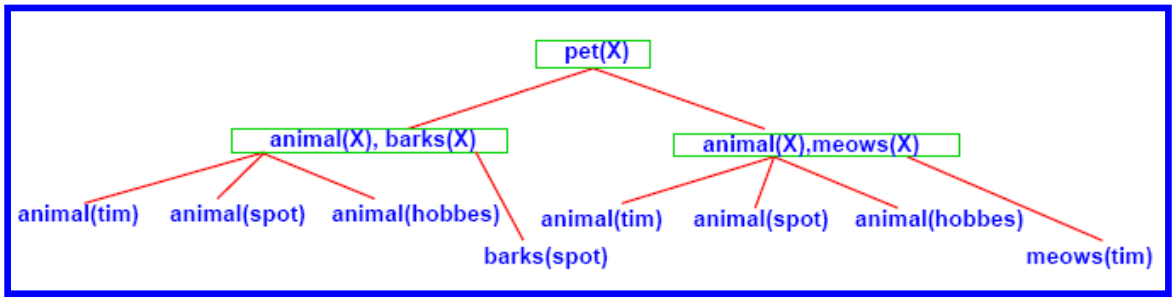
***Nota**: La ejecución siempre finaliza en las hojas del árbol (los hechos)*

```
animal(tim).
animal(spot).
animal(hobbes).
barks(spot).
meows(tim).
roars(hobbes).

pet(X) :- animal(X), barks(X).
pet(X) :- animal(X), meows(X).
```

- Consultas diferentes implican árboles de búsqueda diferentes
- Las estrategias de ejecución definen como se explora el árbol de búsqueda durante la ejecución

El Árbol de Búsqueda (II)

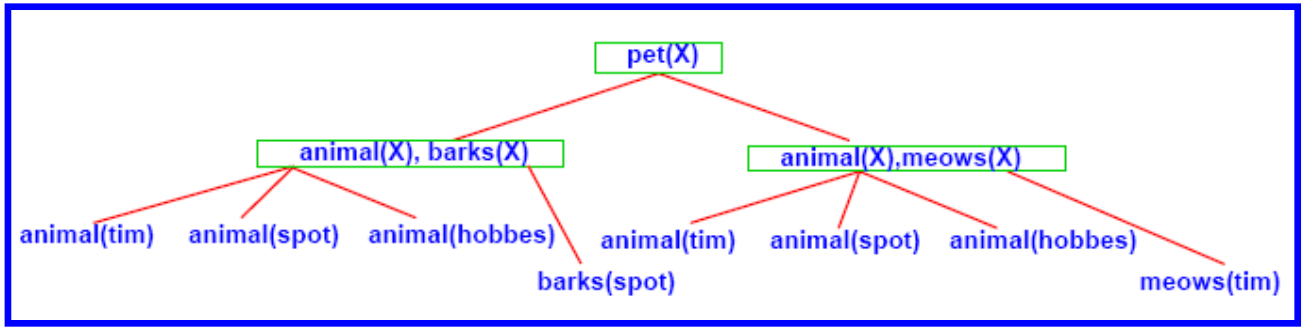
- Los **nodos** del árbol están etiquetados con cláusulas objetivo
  - El **nodo raíz** está etiquetado con la consulta (*query*)
- Cada nodo padre genera tantos **descendientes** como cláusulas del programa lógico coinciden con el primer literal del nodo padre
  - En ocasiones, cada arco se etiqueta con la cláusula del programa lógico empleada



animal(tim).
animal(spot).
animal(hobbes).
barks(spot).
meows(tim).
roars(hobbes).

pet(X) :- animal(X), barks(X).
pet(X) :- animal(X), meows(X).

Explorando el Árbol de Búsqueda



- Explorar el árbol de arriba a abajo (*top-down*) → "llamada" ("*call*")
- Explorar el árbol de abajo a arriba (bottom-up) → "deducir" ("*deduce*")
- Explorar las metas (objetivos; *goals*) de izquierda a derecha o de derecha a izquierda
- Explorar las ramas (*branches*) de izquierda a derecha o de derecha a izquierda
- Explorar las metas (objetivos; *goals*) todas al mismo tiempo
- Explorar las ramas (*branches*) todas al mismo tiempo
- Etc.

Ejemplo: Árboles de Búsqueda

- Dado el siguiente programa lógico

  padre(a,b).

  madre(b,c).

  abuelo(X,Z) ← progenitor(X,Y), progenitor(Y,Z).

  progenitor(X,Y) ← padre(X,Y).

  progenitor(X,Y) ← madre(X,Y).

  ¿Quién es abuelo de ‘c’?

  ¿Quién es nieto de ‘a’?

Ejecución de Programas: Interacción con el Sistema (I)

- Cada sistema implementa una **estrategia** concreta para ejecutar los programas lógicos
  - Todos los sistemas Prolog implementan la misma estrategia
- La estrategia está orientada a explorar todo el árbol y a devolver las soluciones una a una
- Ejemplo: ("?-" es el símbolo del sistema)
  - ?- pet(X).
    X = spot ?
    yes
    ?-

    ?- pet(X).
    X = spot ? ;
    X = tim ? ;
    no
    ?-

Ejecución de Programas: Interacción con el Sistema (II)

- Los sistemas Prolog también permiten crear ejecutables que comienzan con una consulta predefinida dada
  - Habitualmente *main/0* y/o *main/n*
- Algunos sistemas permiten introducir consultas en el texto del programa, comenzando con :- (es decir, una regla sin cabeza)
  - Estas se ejecutan al cargar el archivo (o iniciar el ejecutable)

Significado Operacional de Programas (I)

- Un **programa lógico** es operacionalmente un conjunto de definiciones de procedimientos (los predicados)
- Una **consulta** "?- p." es una llamada a un procedimiento
- Una **definición de procedimiento con una cláusula**

  p :- p1, ..., pm. significa:

  - "Para ejecutar una llamada a p tienes que llamar a p1 y ... y a pm"
    - En principio, el orden en que se llama a p1, ..., pm no importa, pero en la práctica los sistemas fijan un determinado orden

Significado Operacional de Programas (II)

- Una **definición de procedimiento con varias cláusulas** (definiciones)

  p :- p1, ..., pn.

  p :- q1, ..., qm.

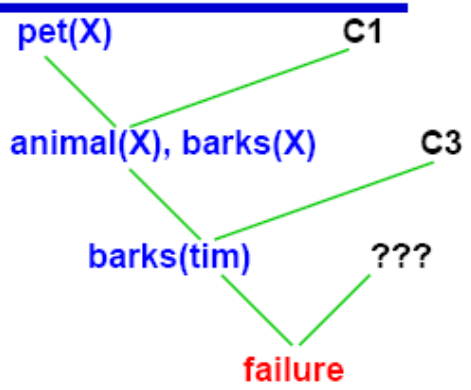
  etc. significa:

  - "Para ejecutar una llamada a p, llame p1 y ... y pn, o, alternativamente, a q1 y ... y qm, o etc. . . "
    - Es como tener varias definiciones de procedimientos alternativos
    - Significa que pueden existir varios caminos posibles para una solución y que dichos caminos deben ser explorados
    - El sistema generalmente se detiene cuando encuentra la primera solución. El usuario puede pedir más soluciones
    - En principio, el orden en el que se exploran estos caminos no importa (si se cumplen ciertas condiciones), pero para un sistema dado, este orden normalmente se fija

Ejecución de Programas: Caminos Alternativos

Running Programs: Alternative Execution Paths

```
C1:  pet(X) :-
         animal(X), barks(X).
C2:  pet(X) :-
         animal(X), meows(X).
C3:  animal(tim).        C6:  barks(spot).
C4:  animal(spot).       C7:  meows(tim).
C5:  animal(hobbes).     C8:  roars(hobbes).
```



- `?- pet(X).` (top-down, left-to-right)

Q	R	Clause	θ
pet(X)	pet(X)	C_1*	{ $X=X_1$ }
pet(X_1)	animal(X_1), barks(X_1)	C_3*	{ X_1=tim }
pet(tim)	barks(tim)	???	*failure*

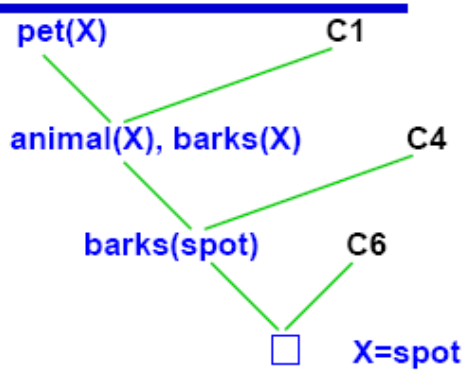
* means *choice-point*, i.e., other clauses applicable.

- But solutions exist in other paths!

Ejecución de Programas: Diferentes Ramas

Running Programs: Different Branches

```
C1:  pet(X) :-
         animal(X), barks(X).
C2:  pet(X) :-
         animal(X), meows(X).
C3:  animal(tim).        C6:  barks(spot).
C4:  animal(spot).       C7:  meows(tim).
C5:  animal(hobbes).     C8:  roars(hobbes).
```



- `?- pet(X).` (top-down, left-to-right, different branch)

Q	R	Clause	θ
pet(X)	pet(X)	C_1*	{ X=X_1 }
pet(X_1)	animal(X_1), barks(X_1)	C_4*	{ X_1=spot }
pet(spot)	barks(spot)	C_6	{}
pet(spot)	—	—	—

Backtracking (I)

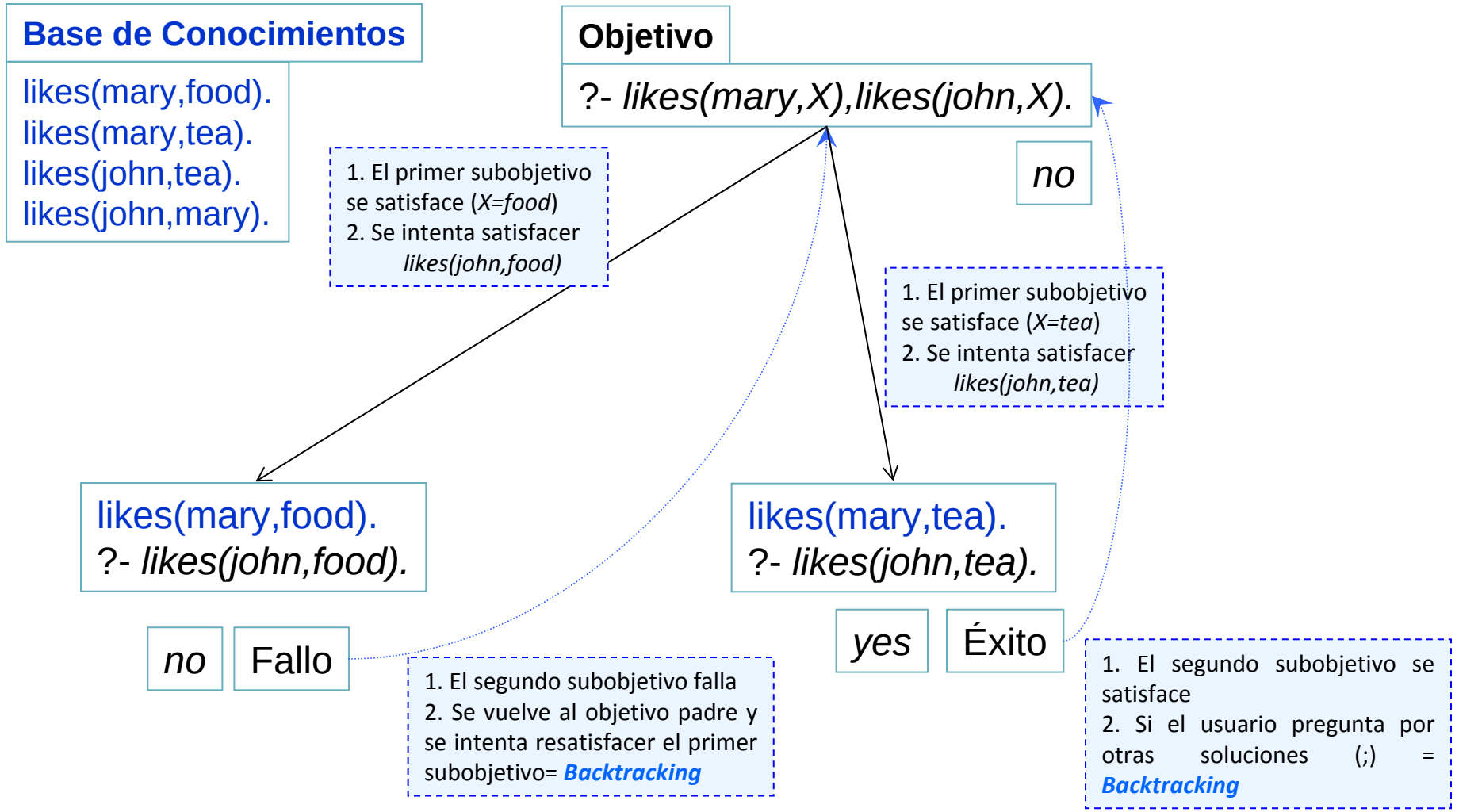
- Un hecho puede hacer que un objetivo se cumpla inmediatamente
- Una regla sólo puede reducir la tarea (el objetivo o meta buscada) a la de satisfacer una conjunción de subobjetivos
- Si no se puede satisfacer un objetivo, se iniciará un proceso de ***backtracking***
  - Este proceso consiste en intentar satisfacer los objetivos buscando una forma alternativa de hacerlo
- El mecanismo de ***backtracking*** permite explorar los diferentes caminos de ejecución hasta que se encuentre una solución

Backtracking (Prolog) (II)

- ***Backtracking*** es la forma en la que la estrategia de ejecución de Prolog explora diferentes ramas del árbol de búsqueda
  - Se trata de una especie de "ejecución hacia atrás" ("*backwards execution*")
- Algoritmo (Esquemático).

  "Explorar la última rama pendiente" significa:
  - Tomar el último literal ejecutado con éxito
  - Tomar la cláusula contra la que fue ejecutado
  - Tomar el unificador del literal y la cabeza de la cláusula
  - Deshacer las unificaciones
  - Realizar la ejecución hacia delante de nuevo

Backtracking (III): Ejemplo



Ejecución de Programs: Ejecución Completa (Todas las Soluciones) (I)

Running Programs: Complete Execution (All Solutions)

```
C1: pet(X) :- animal(X), barks(X).
C2: pet(X) :- animal(X), meows(X).
C3: animal(tim).
C4: animal(spot).
C5: animal(hobbes).
C6: barks(spot).
C7: meows(tim).
C8: roars(hobbes).
```

- `?- pet(X).` (top-down, left-to-right)

Q	R	Clause	θ	Choice-points		
pet(X)	pet(X)	C_1*	{ $X=X_1$ }			*
pet(X_1)	animal(X_1), barks(X_1)	C_3*	{ X_1=tim }		*	
pet(tim)	barks(tim)	???	*failure*			
	deep backtracking				*	
pet(X_1)	animal(X_1), barks(X_1)	C_4*	{ X_1=spot }		*	
pet(spot)	barks(spot)	C_6	{}			
pet(spot)	—	—	—			
;	triggers backtracking				*	
continues...						

Ejecución de Programs: Ejecución Completa (Todas las Soluciones) (II)

Running Programs: Complete Execution (All Solutions)

```
C1:  pet(X) :- animal(X), barks(X).
C2:  pet(X) :- animal(X), meows(X).
C3:  animal(tim).
C4:  animal(spot).
C5:  animal(hobbes).
C6:  barks(spot).
C7:  meows(tim).
C8:  roars(hobbes).
```

- `?- pet(X).` (continued)

Q	R	Clause	θ	Choice-points		
pet(X_1)	animal(X_1), barks(X_1)	C_5	{ X_1=hobbes }			
pet(hobbes)	barks(hobbes)	???	*failure*			
	deep backtracking					*
pet(X)	pet(X)	C_2	{ X=X_2 }			
pet(X_2)	animal(X_2), meows(X_2)	C_3*	{ X_2=tim }		*	
pet(tim)	meows(tim)	C_7	{}			
pet(tim)	—	—	—			
;	triggers backtracking				*	
continues...						

Ejecución de Programs: Ejecución Completa (Todas las Soluciones) (III)

Running Programs: Complete Execution (All Solutions)

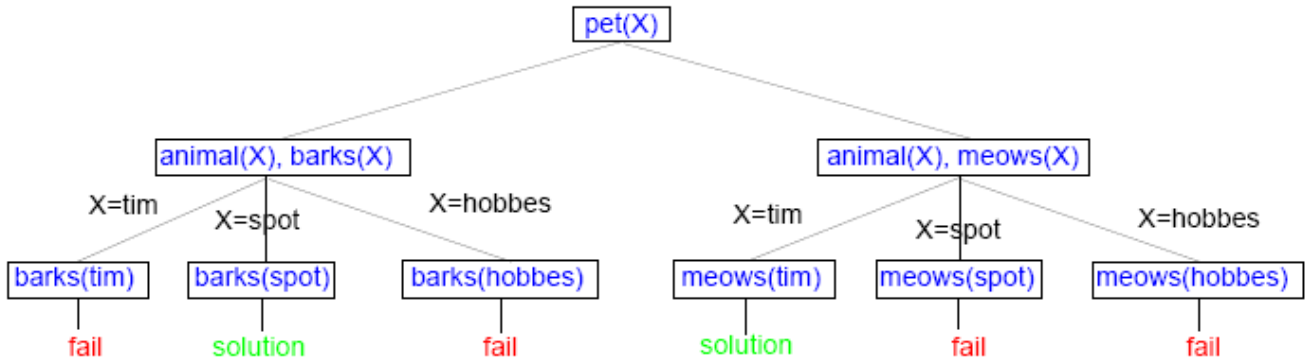
```
C1:  pet(X) :- animal(X), barks(X).
C2:  pet(X) :- animal(X), meows(X).
C3:  animal(tim).
C4:  animal(spot).
C5:  animal(hobbes).
C6:  barks(spot).
C7:  meows(tim).
C8:  roars(hobbes).
```

- `?- pet(X).` (continued)

Q	R	Clause	θ	Choice-points		
$pet(X_2)$	$animal(X_2), meows(X_2)$	C_4*	$\{ X_2$=spot $\}$		*	
pet(spot)	meows(spot)	???	*failure*			
	deep backtracking				*	
$pet(X_2)$	$animal(X_2), meows(X_2)$	C_5	$\{ X_2$=hobbes $\}$			
pet(hobbes)	meows(hobbes)	???	*failure*			
	deep backtracking					
failure						

Árbol de Búsqueda: Resumen (I)

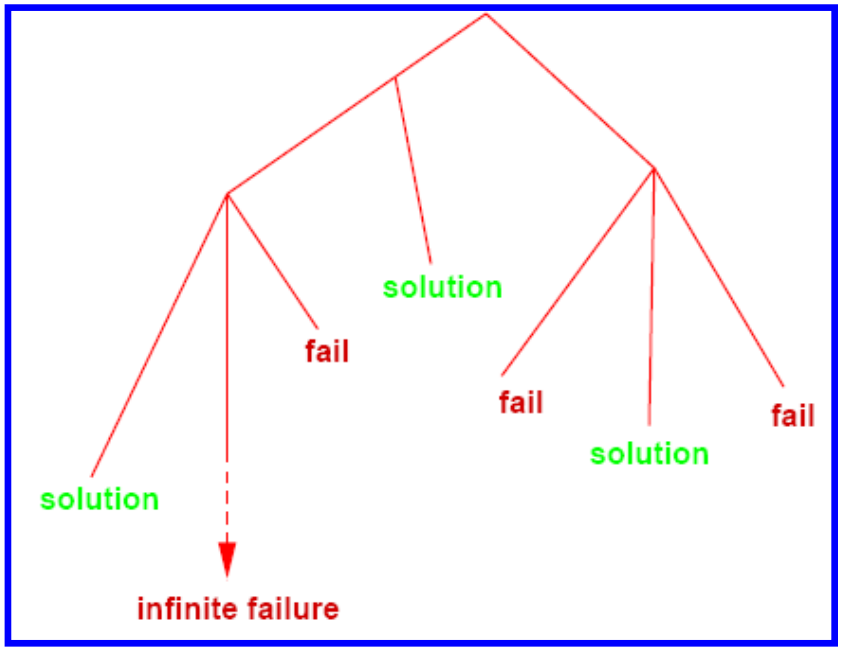
- Diferentes estrategias de ejecución exploran el árbol de búsqueda de manera diferente
- Una estrategia es completa si garantiza que encuentra todas las soluciones existentes




```
pet(X) :- animal(X), barks(X).
pet(X) :- animal(X), meows(X).

animal(tim).
animal(spot).
animal(hobbes).

barks(spot).

meows(tim).
```

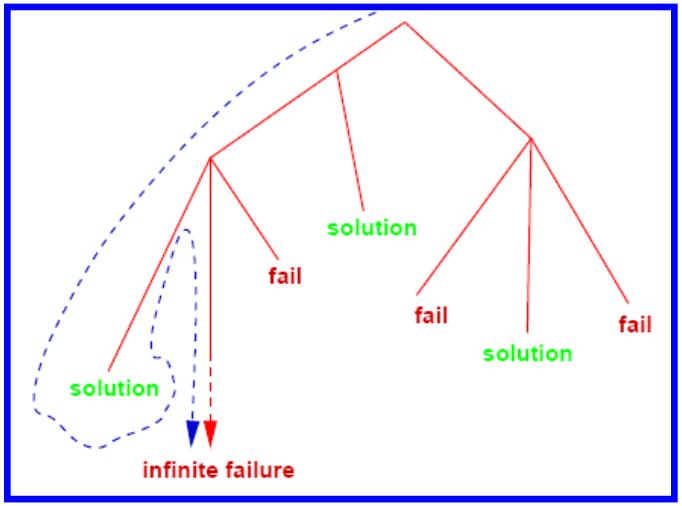
Árbol de Búsqueda: Resumen (II)

- Todas las soluciones tienen una profundidad finita en el árbol
- Los fallos pueden tener una profundidad finita o, en algunos casos, ser una rama infinita
- Existen dos **tipos de búsqueda**:
  - Búsqueda en Profundidad
  - Búsqueda en Anchura



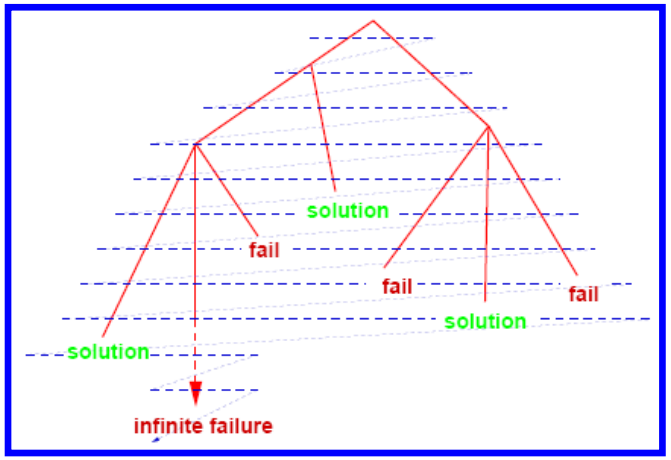
Búsqueda en Profundidad

- La búsqueda en **profundidad** explora un descendiente y no pasa a explorar otro hasta que todos sus caminos terminan (con éxito o con fallo)
  - Ante un nodo terminal de fallo, debe "dar marcha atrás" y retomar la deducción a partir de otro descendiente ("vuelta atrás" o *backtracking*)
  - Problema con caminos infinitos: no hay marcha atrás
    - Es una búsqueda incompleta: se puede caer en una rama infinita antes de encontrar todas las soluciones
  - Ventaja: sólo es necesario acceder a la rama que está siendo explorada
    - Es una búsqueda muy eficiente: se puede implementar con una pila de llamadas (muy similar a una lenguaje de programación tradicional)



Búsqueda en Anchura

- La búsqueda en **anchura** explora todos los descendientes en paralelo, extendiendo cada uno un paso hasta que se alcanza éxito en algún paso o no quedan nodos que prologar y se finaliza con fallo
  - No hay necesidad de “dar marcha atrás” ni problema con caminos infinitos
  - Encuentra todas las soluciones antes de caer en una rama infinita
  - Es una búsqueda costosa en términos de tiempo y memoria
    - Es necesario tener disponible en memoria el árbol completo



Ejemplo: Árbol de Búsqueda

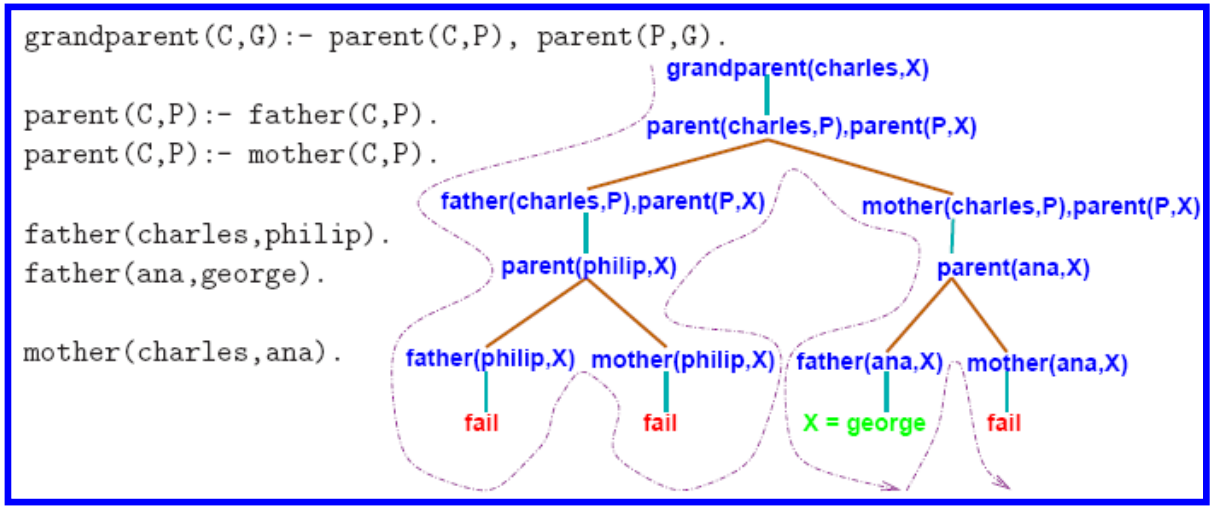
- Consideremos el siguiente programa lógico:

```
amigo(juan,pepe) <- amigo(pepe,juan).
amigo(pepe,juan) <- amigo(juan,pepe).
amigo(juan,pepe) <- rico(pepe).
rico(pepe) <- .
```

Y el objetivo: `?- amigo(juan,pepe).`

Mecanismo de Ejecución de Prolog

- Siempre se ejecutan literales en el cuerpo de las cláusulas de **izquierda a derecha**
- En cualquier punto de elección, se toma la primera cláusula que unifica
  - Es decir, la rama sin explorar más a la izquierda
- En caso de fallo, se da marcha atrás (*backtracking*) a la siguiente cláusula no explorada del último punto de elección

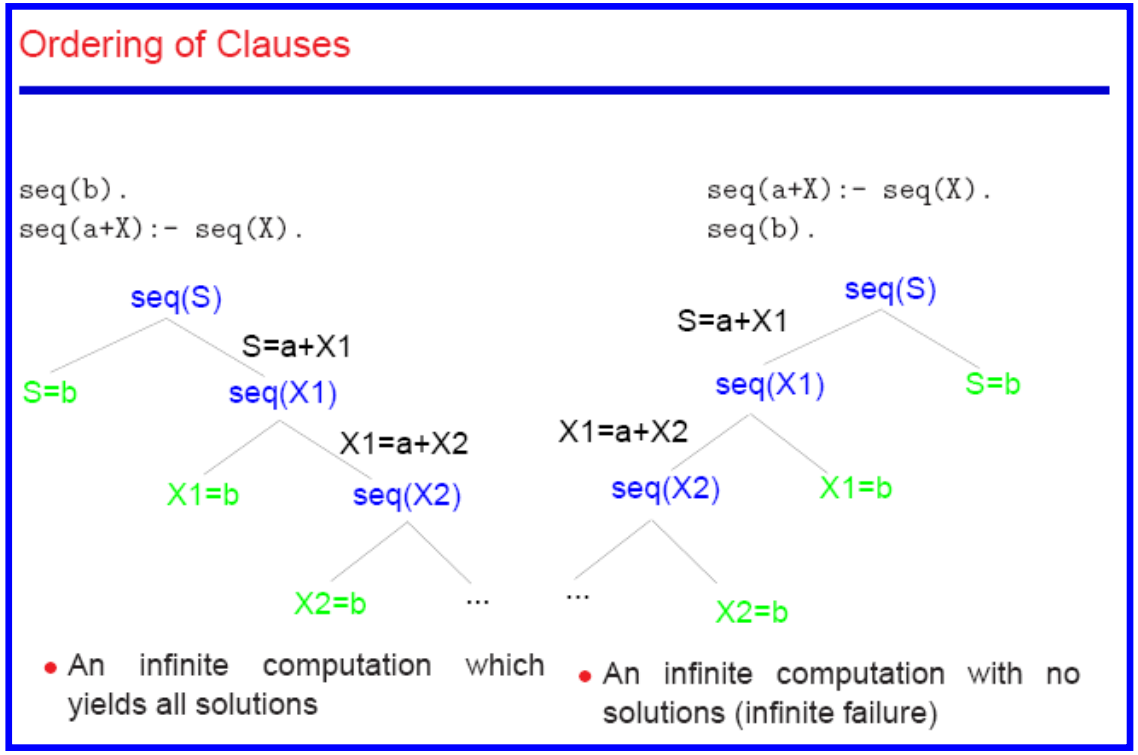


Orden de Cláusulas y Objetivos (I)

- Puesto que la estrategia de ejecución de Prolog es fija, el **orden** en el que el programador escribe cláusulas y objetivos es importante
- El **orden de las cláusulas** determina el orden en el que se exploran los caminos alternativos (del árbol). Por tanto:
  - El orden en que se encuentran las soluciones
  - El orden en que se producen los fallos (y se dispara el retroceso)
  - El orden en que se producen los fallos infinitos

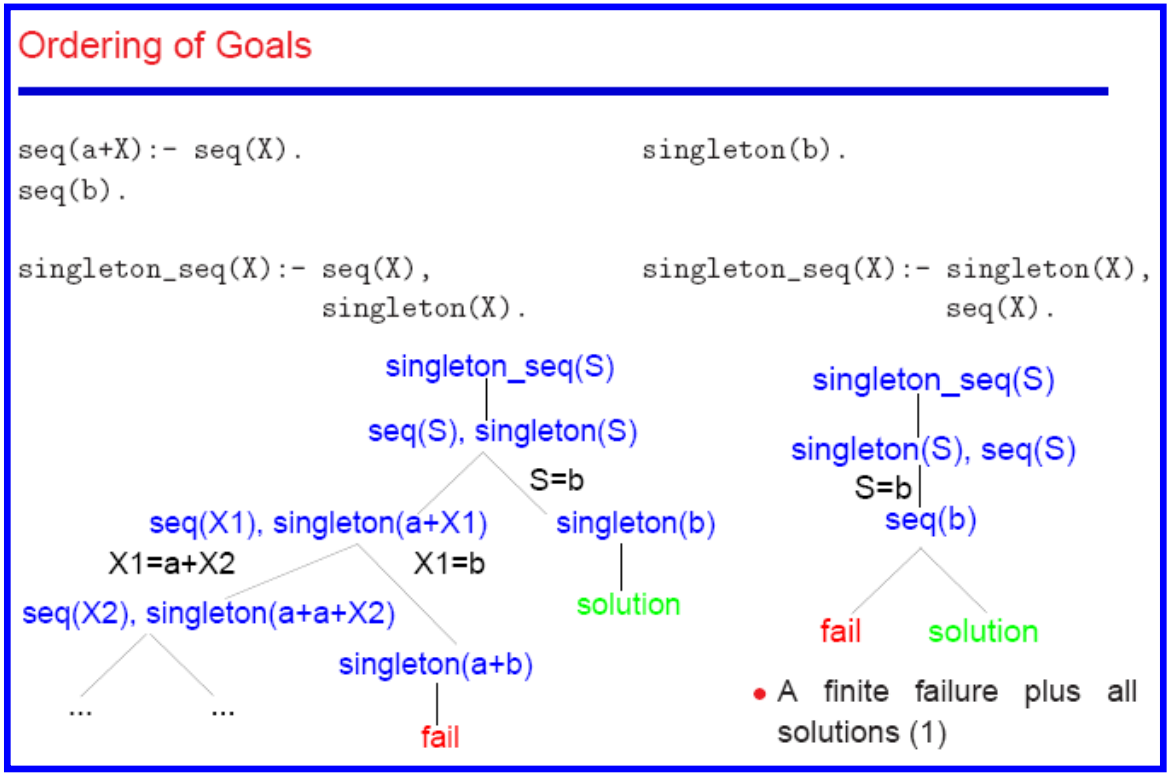
Orden de Cláusulas y Objetivos (II)

- El **orden de las cláusulas** determina el orden en el que se exploran los caminos alternativos (del árbol)



Orden de Cláusulas y Objetivos (III)

- El **orden de los objetivos** determina el orden en que se lleva a cabo la unificación. Por lo tanto:
  - La selección de cláusulas durante la ejecución
    - Es decir, el orden en que se exploran los caminos alternativos



Orden de Cláusulas y Objetivos (IV)

- El orden en que se producen los fallos afecta al tamaño de la computación (eficiencia)
- El orden en que se producen los fallos infinitos afecta a la exhaustividad (terminación)

Estrategias de Ejecución

- **Reglas de búsqueda**: indican cómo se seleccionan las cláusulas/ramas en el árbol de búsqueda
- **Reglas de cálculo**: indican cómo se seleccionan los objetivos en los cuadros del árbol de búsqueda
- **Estrategia de ejecución de Prolog**
  - Regla de búsqueda: de arriba hacia abajo (tal y como se escribe)
  - Regla de cálculo: de izquierda a derecha (tal y como se escribe)
  - Prolog sigue una estrategia *top-down*, *left-to-right* (búsqueda en profundidad)

Resumen

- Un **programa lógico** declara la información conocida en forma de reglas (consecuencias) y hechos
- La **ejecución** de un programa lógico equivale a deducir nueva información
- Diferentes **estrategias de ejecución** tienen diferentes consecuencias en el cálculo de programas
- Prolog es un lenguaje de programación lógica que utiliza una estrategia particular de ejecución
  - Además, incluye predicados predefinidos (que aportan un nivel añadido a la lógica en la que se basa principalmente)

Programación Declarativa:
Lógica y Restricciones

Conceptos Básicos de la Programación Lógica

Mari Carmen Suárez de Figueroa Baonza
mcsuarez@fi.upm.es