

# Programación lógica

Paradigmas de la Programación  
FaMAF-UNC 2015  
capítulo 15 Mitchell  
basado en filminas de Vitaly Shmatikov

# algunas referencias

- <http://www.learnprolognow.org/>
- <http://cs.famaf.unc.edu.ar/wiki/doku.php?id=introalg:taller09>
- <http://www.swi-prolog.org/>

# programación lógica

- la primitiva básica en los lenguajes vistos hasta ahora es la función (método, procedimiento)

$F(x) = y$  – la función  $F$  toma  $x$  y devuelve  $y$

- en programación lógica, la primitiva básica es la **relación** (predicado)

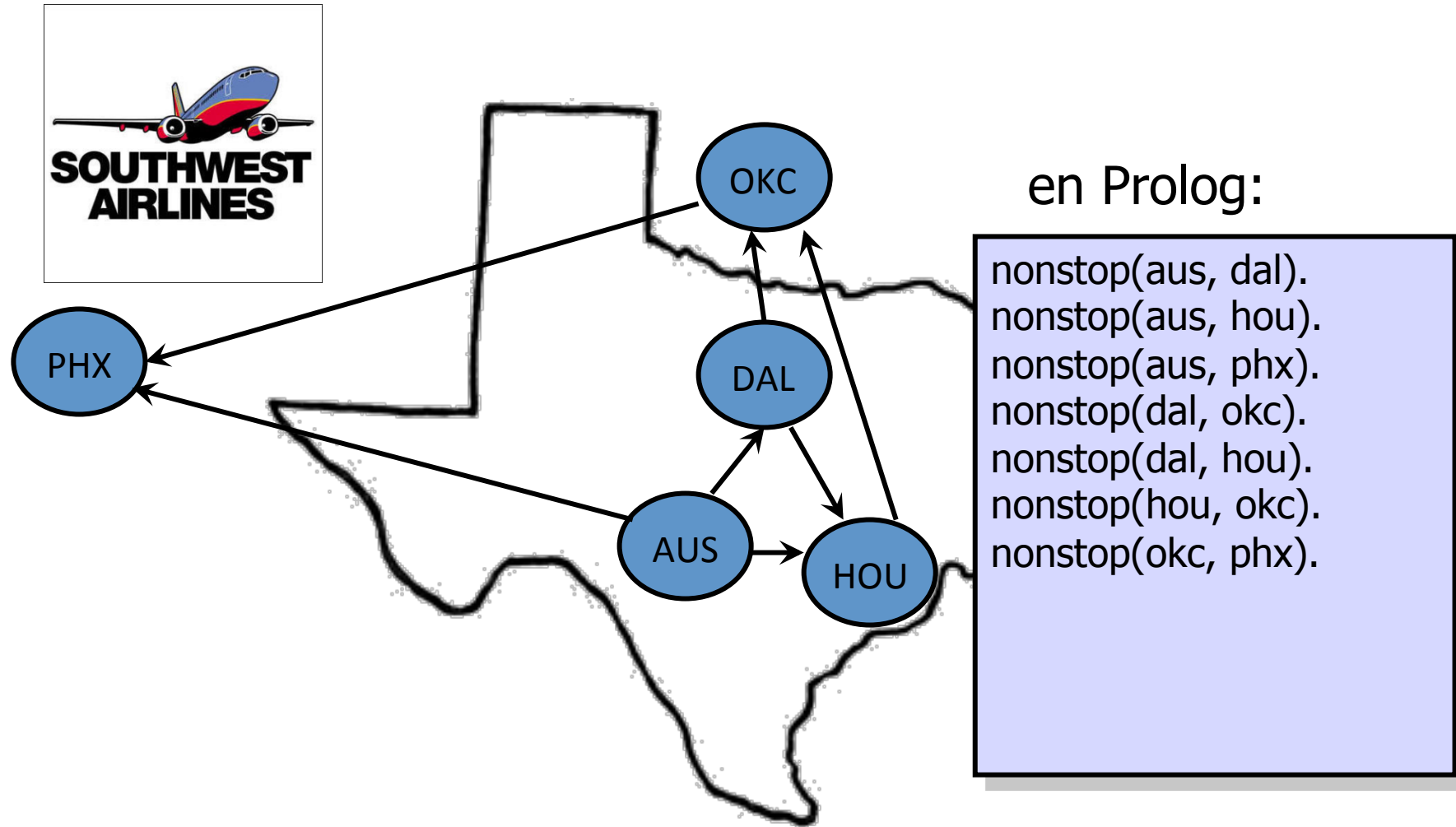
$R(x, y)$  – se da la relación  $R$  entre  $x$  e  $y$

# Prolog



- Acrónimo de **P**rogrammation en **l**ogic
  - Alain Colmerauer (1972)
- idea básica: el programa declara los objetivos de la computación, no la forma de obtenerlos
- aplicaciones en IA, bases de datos
  - originalmente desarrollado para procesamiento del lenguaje natural
  - razonamiento automático, probadores de teoremas
  - búsquedas en bases de datos
  - sistemas expertos

# ejemplo: base de datos lógica

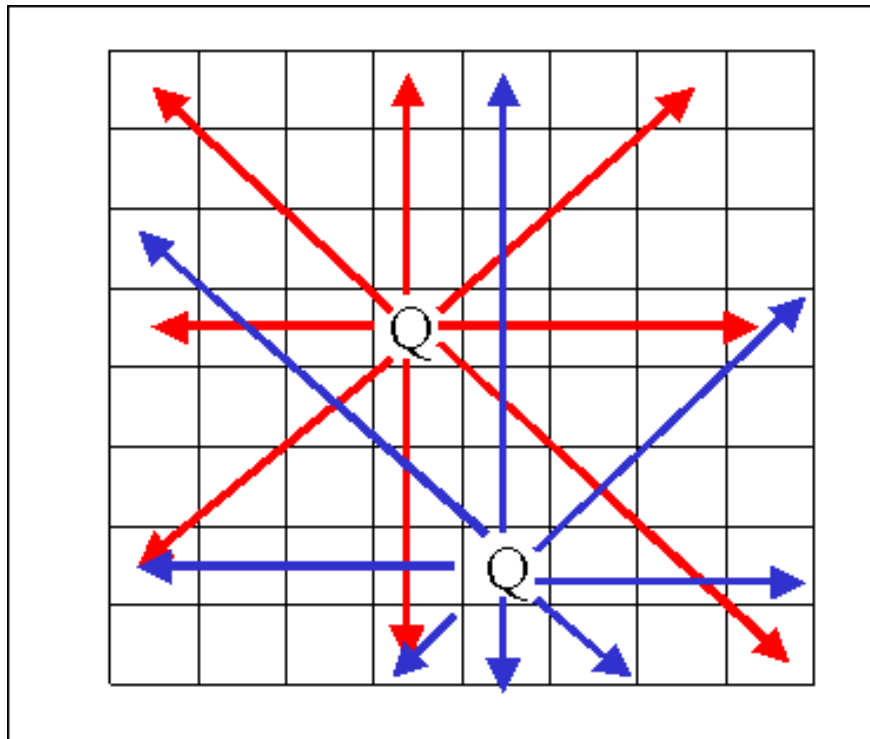


# consultas a una base de datos lógica

- a dónde podemos volar desde Austin?
- SQL
  - SELECT dest FROM nonstop WHERE source="aus";
- Prolog
  - ?- nonstop(aus, X).
  - más poderoso que SQL porque se puede usar recursión

# problema de N-Reinas

ubicar N reinas que no se estén atacando en un tablero de ajedrez (problema de búsqueda)



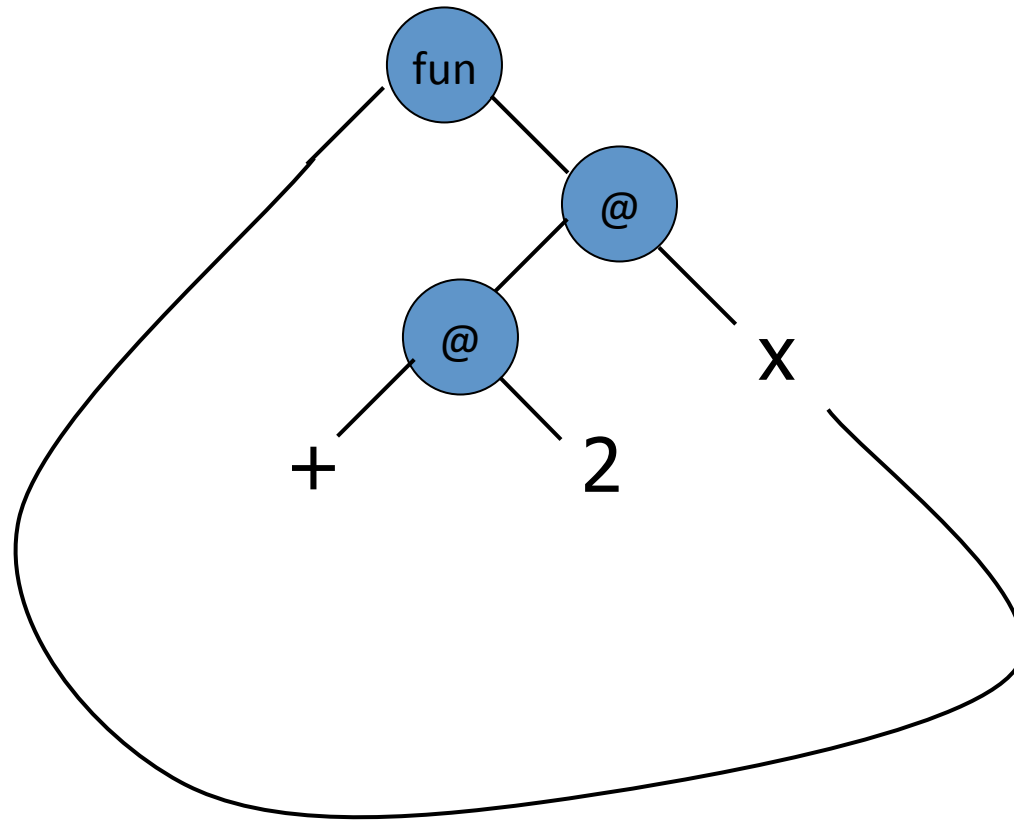
# N-Reinas en Prolog

```
diagsegura(_, _, []).
diagsegura(Columna, ColDist, [QR|QRs]) :-
    ColumnaHit1 is Columna + ColDist, QR \= ColumnaHit1,
    ColumnaHit2 is Columna - ColDist, QR \= ColumnaHit2,
    ColDist1 is ColDist + 1,
    diagsegura(Columna, ColDist1, QRs).
posicion_segura([_]).
posicion_segura([QR|QRs]) :-
    diagsegura(QR, 1, QRs),
    posicion_segura(QRs).
nreinas(N, Y) :-
    sequence(N, X), permute(X, Y), posicion_segura(Y).
```

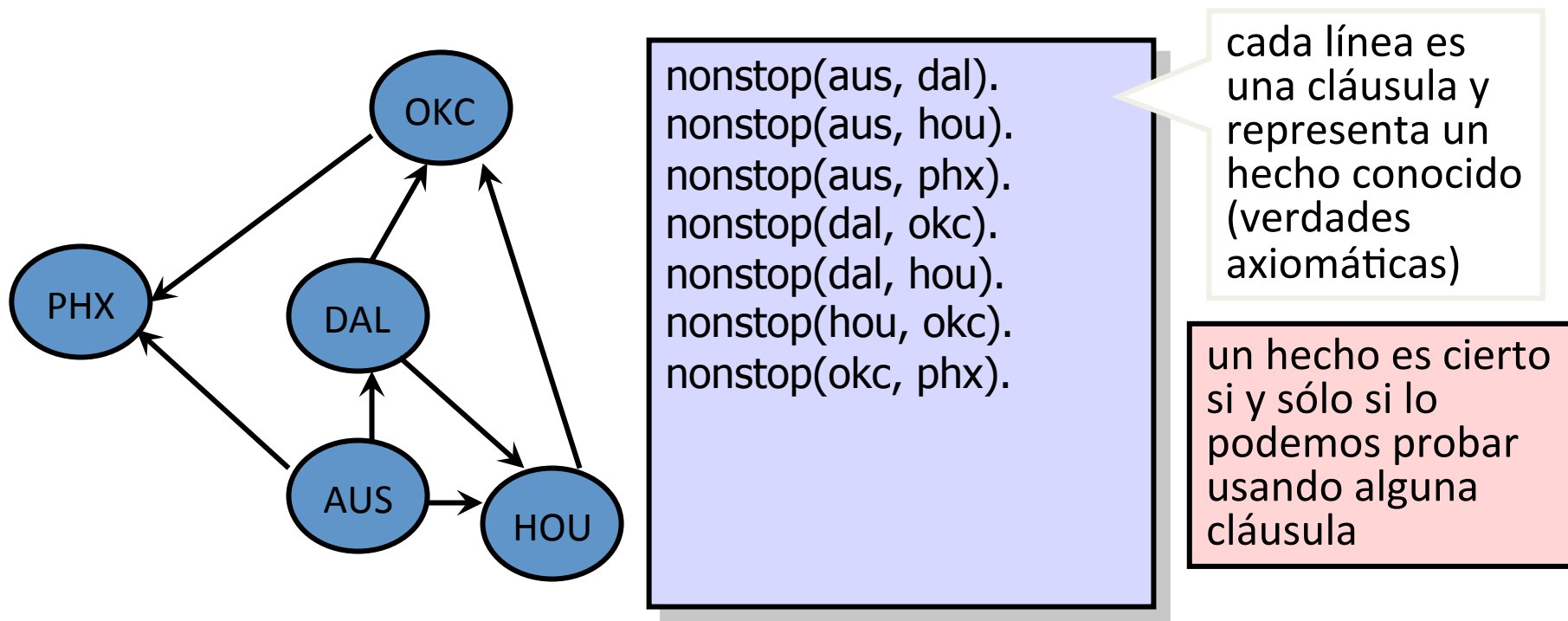


# inferencia de tipos en ML

- Given an ML term, find its type

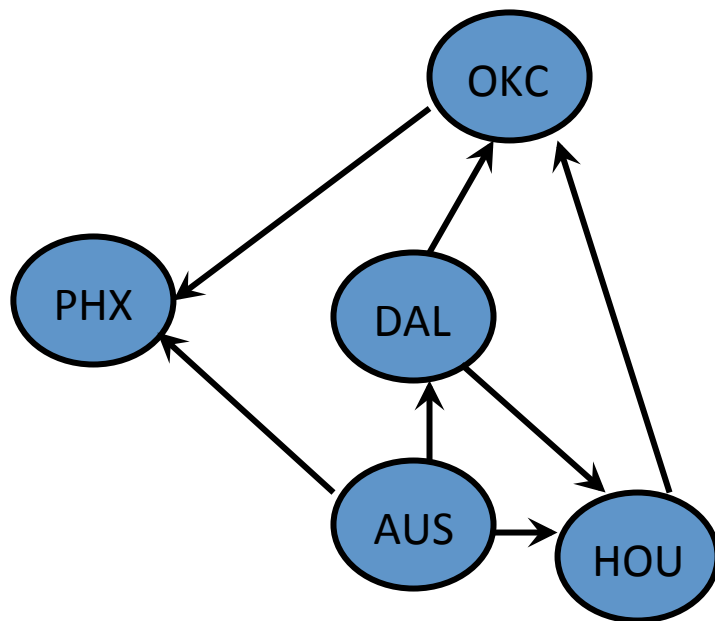


# planificación de vuelos



Relación: **nonstop(X, Y)** – hay un vuelo desde X hasta Y

# consultas



?- nonstop(aus, dal).

Yes

?- nonstop(dal, okc).

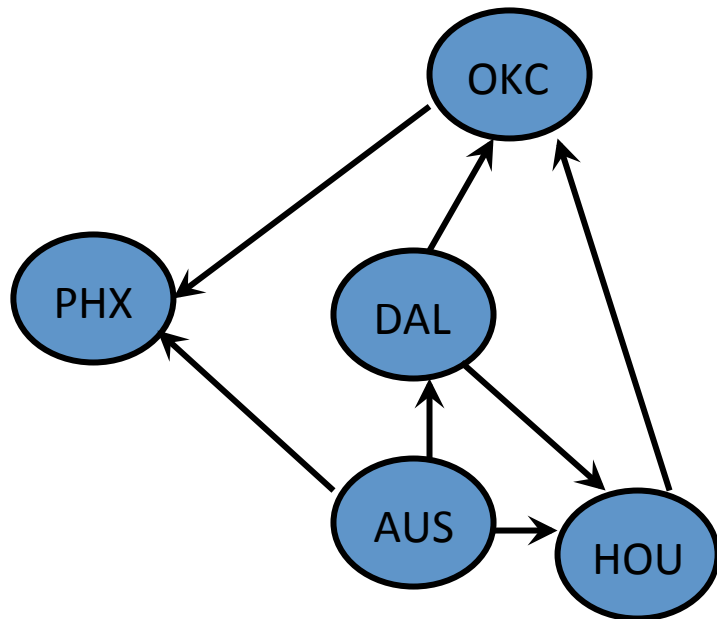
Yes

?- nonstop(aus, okc).

No

?-

# variables lógicas



hay algún X tal que  
nonstop(okc, X)?

?- nonstop(okc, X).

X=phx ;

No

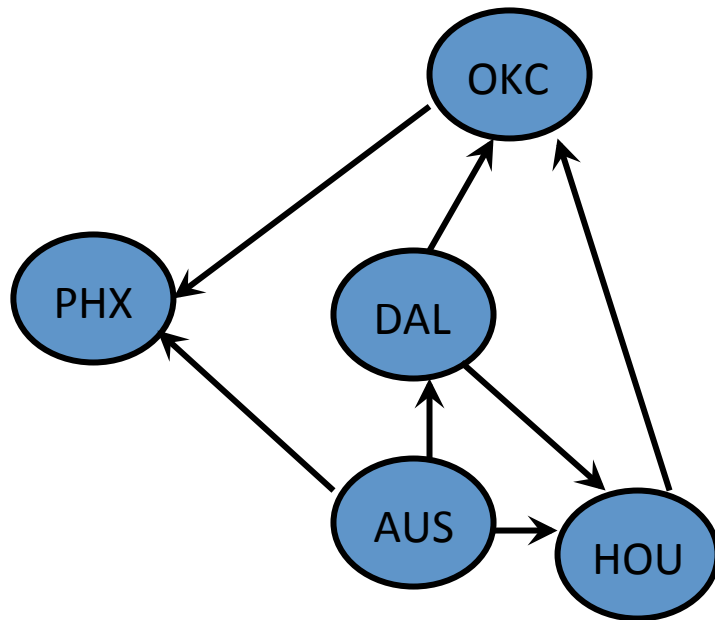
?- nonstop(Y, dal).

Y=aus ;

No

?-

# no-determinismo



?- nonstop(dal, X).

X=hou ;

X=okc ;

No

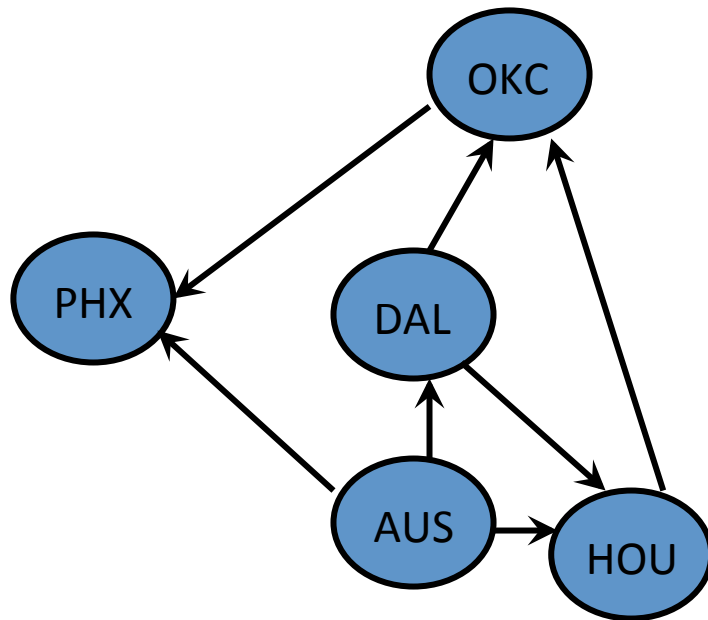
?- nonstop(phx, X).

No

?-

los predicados pueden devolver múltiples respuestas o ninguna

# conjunción lógica



?- nonstop(aus, X), nonstop(X, okc).  
X=dal ;  
X=hou ;  
No  
?-

combinar condiciones múltiples en una sola consulta

# predicados derivados

se pueden definir nuevos predicados con reglas:

**conclusión :- premisas.**

la conclusión es cierta si las premisas son ciertas

```
volar_via(Desde, Hacia, Via) :-  
    nonstop(Desde, Via),  
    nonstop(Via, Hacia).
```

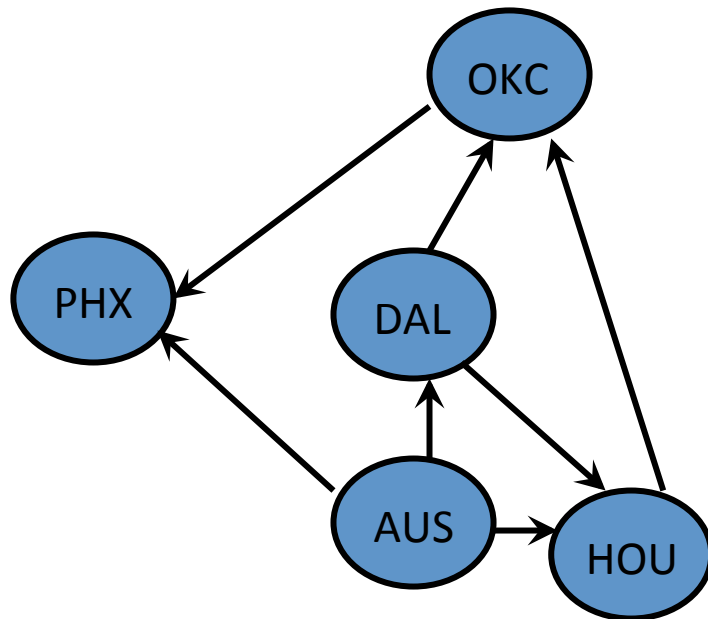
```
?- volar_via(aus, okc, Via).
```

```
Via=dal ;
```

```
Via=hou ;
```

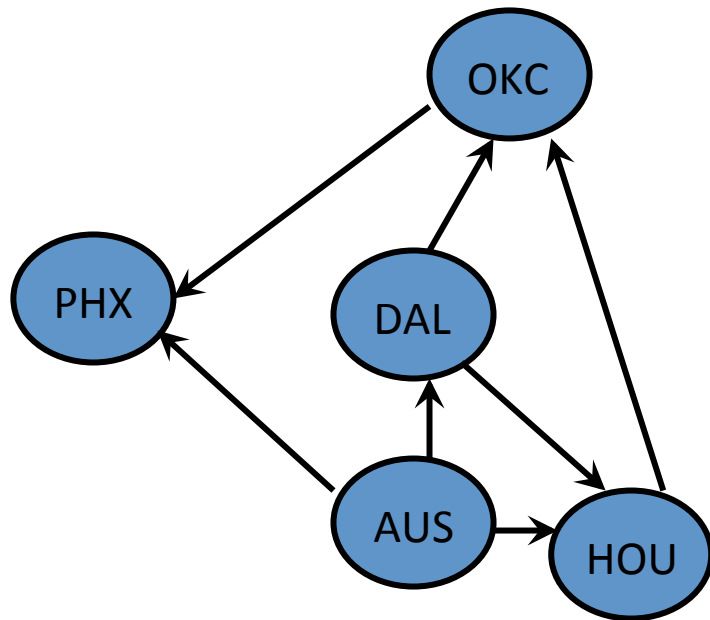
```
No
```

```
?-
```



# recursión

los predicados se pueden definir recursivamente



```
llegar(X, X).
```

```
llegar(X,Z) :-
```

```
    nonstop(X, Y), llegar(Y, Z).
```

```
?- llegar(X, phx).
```

```
X=aus ;
```

```
X=dal ;
```

```
...
```

```
?-
```



# elementos de un programa Prolog

- los programas en Prolog tienen términos
  - variables, constantes, estructuras
- las variables empiezan en mayúscula

`Harry`

- las constantes son enteros o átomos

`24, zebra, 'Bob', '.'`

- las estructuras son predicados con argumentos

`n(zebra), habla(Y, Castellano)`

# cláusulas de Horn

- una cláusula de Horn tiene una cabeza  $h$  que es un predicado y un cuerpo, que es una lista de predicados  $p_1, p_2, \dots, p_n$ 
  - se escribe  $h \leftarrow p_1, p_2, \dots, p_n$
  - significa, “ $h$  es cierto si  $p_1, p_2, \dots$ , y  $p_n$  son ciertos simultáneamente”

ejemplo:

$\text{nieva}(C) \leftarrow \text{precipitación}(C), \text{hiela}(C)$

“nieva en la ciudad  $C$  si hay precipitación en  $C$  y hiela en  $C$ ”

# hechos, reglas y programas

- un hecho en Prolog es una cláusula de Horn sin parte derecha (o con parte derecha true)

`magoo (Harry) .`

- una regla Prolog es una cláusula de Horn con una parte derecha (`:-` es  $\leftarrow$ )

`term :- term1, term2, ... termn.`

– la parte izquierda se llama *cabeza*

- un programa Prolog es un conjunto de hechos y reglas

# cláusulas de Horn y predicados

- cualquier cláusula de Horn  $h \leftarrow p_1, p_2, \dots, p_n$  se puede escribir como un predicado  $p_1 \wedge p_2 \wedge \dots \wedge p_n \supset h$ , o, de forma equivalente  $\neg(p_1 \wedge p_2 \wedge \dots \wedge p_n) \vee h$
- no todo predicado se puede escribir como una cláusula de Horn
  - ejemplo:  $\text{literato}(x) \supset \text{lee}(x) \vee \text{escribe}(x)$

# listas

- una lista es una serie de términos separados por comas y entre corchetes
  - lista vacía: `[]`
  - elemento sin restricciones de `_`: `[_, x, y]`
  - también se puede escribir `[Cabeza | Cola]`

# añadir a una lista

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

```
append([Head | Tail], Y, [Head | Z]) :-  
    append(Tail, Y, Z).
```

el último parámetro va a contener el resultado de la función, pasamos como argumento la variable que va a contener el resultado

- esta definición dice:
  - añadir **X** a la lista vacía devuelve **X**
  - si añadimos **Y** a **Tail** para obtener **Z**, entonces **Y** se puede añadir a una lista un elemento más larga **[Head | Tail]** para obtener **[Head | Z]**

# estar en una lista (*existe*)

`member(X, [X | _]).`

`member(X, [_ | Y]) :- member(X, Y).`

- el predicado de la cabeza será cierto si:
  - `X` es la cabeza de la lista `[X | _]`
  - `X` no es la cabeza de la lista `[_ | Y]`, pero es un miembro de la cola `Y`
- la equivalencia se comprueba con *pattern matching*
- los elementos “sin restricciones” se marcan con `_`, y muestran elementos que no son importantes para la regla

# más funciones sobre listas

- X es un prefijo de Z si hay una lista Y que se puede añadir a X para hacer Z

– `prefix(X, Z) :- append(X, Y, Z).`

– `suffix(Y, Z) :- append(X, Y, Z).`

- encontrar todos los prefijos (o sufijos) de una lista:

`?- prefix(X, [my, dog, has, fleas]).`

`X = [];`

`X = [my];`

`X = [my, dog];`

`...`



# contestar consultas Prolog

- la computación en Prolog (contestar una consulta) es esencialmente buscar una prueba lógica
- dirigido por el objetivo, por backtracking, búsqueda en profundidad (vs. en anchura), con estrategia:  
si  $h$  es la cabeza de una cláusula de Horn

$h \leftarrow \text{términos}$

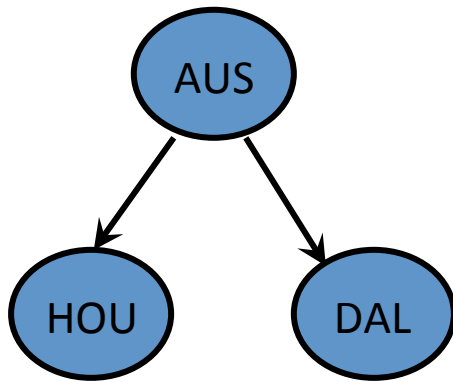
y hace pattern matching con uno de los términos de otra cláusula de Horn

$t \leftarrow t1, h, t2$

entonces ese término se puede reemplazar por los términos de  $h$ :

$t \leftarrow t1, \text{términos}, t2$

# ejemplo con planificación de vuelos



?- n(aus, hou).

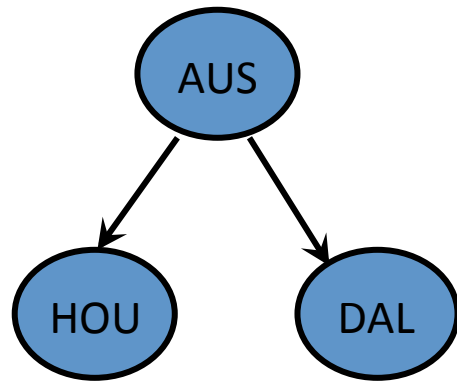
?- n(aus, dal).

?- r(X, X).

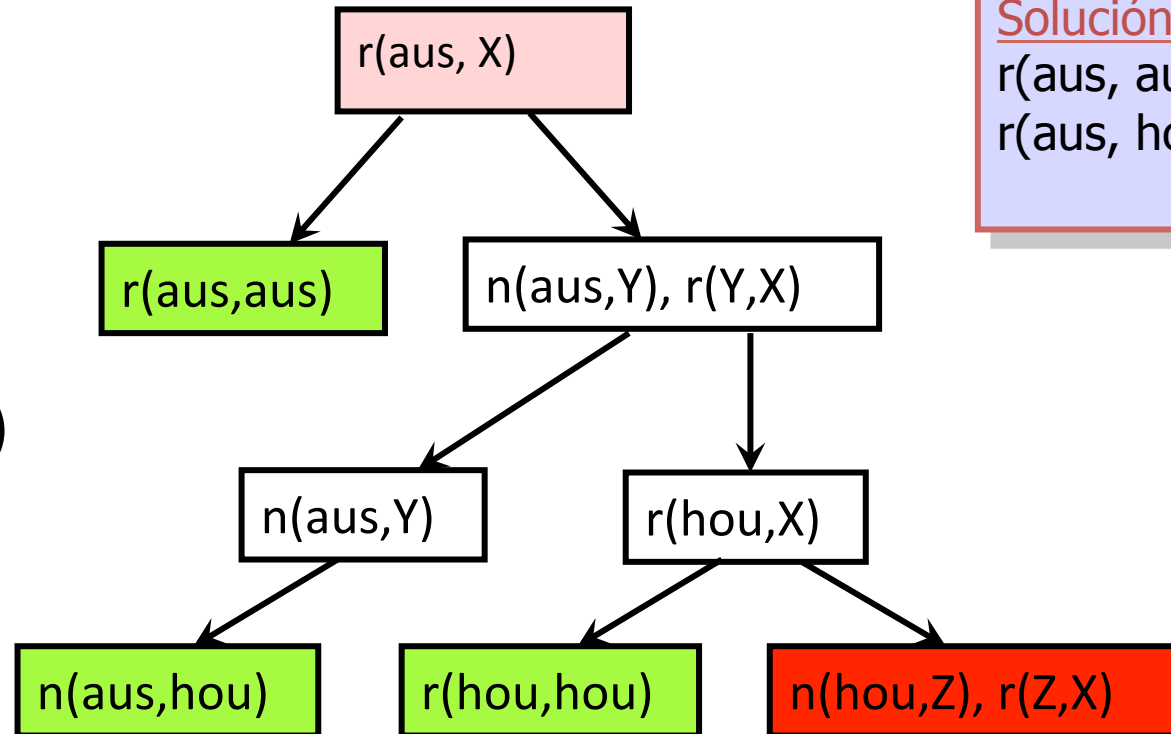
?- r(X, Z) :- n(X, Y), r(Y, Z).

?- r(aus, X)

# planificación de vuelos: búsqueda de prueba

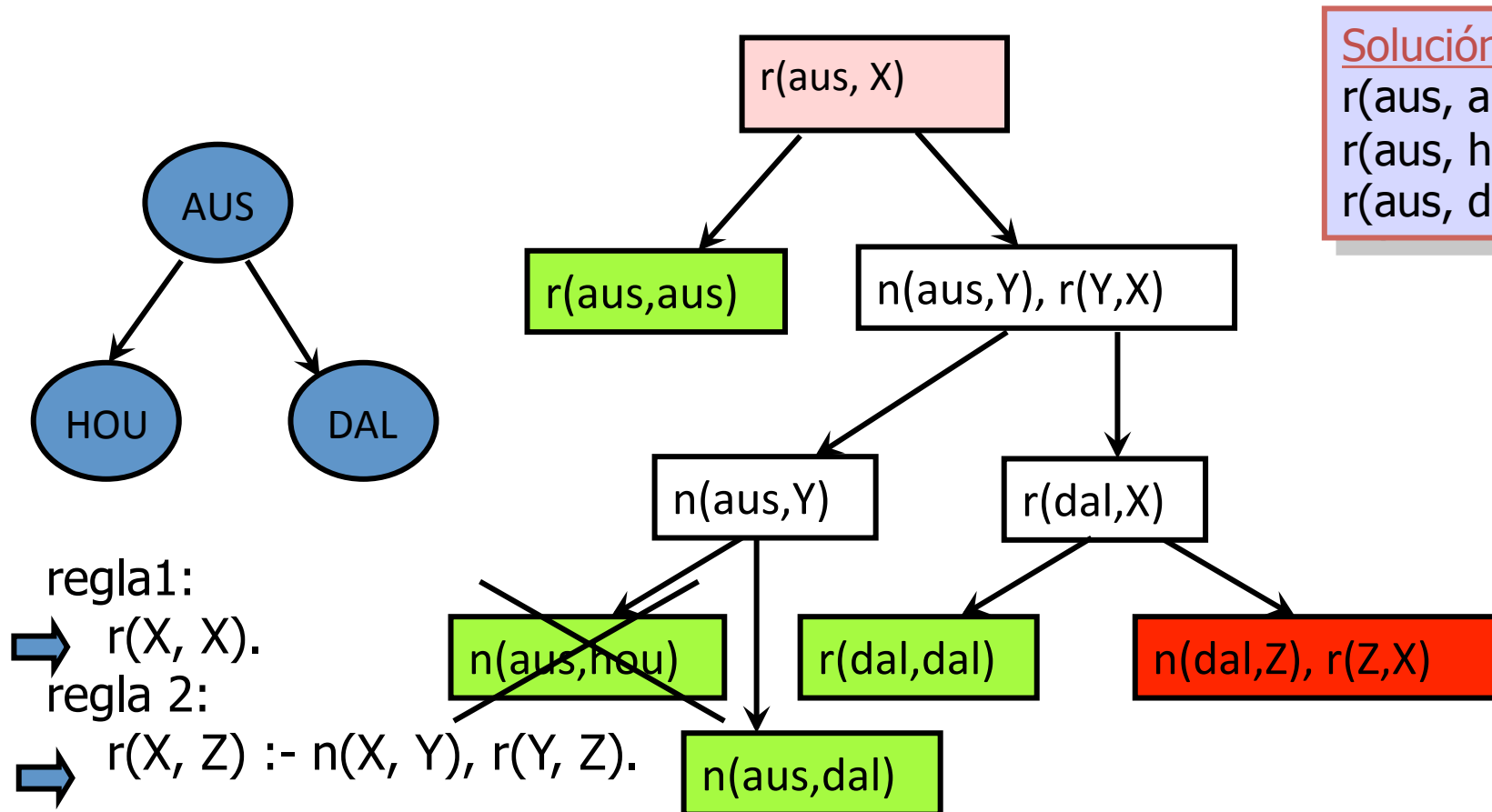


regla1:  
➡  $r(X, X).$   
regla 2:  
➡  $r(X, Z) :- n(X, Y), r(Y, Z).$



Solución  
 $r(aus, aus)$   
 $r(aus, hou)$

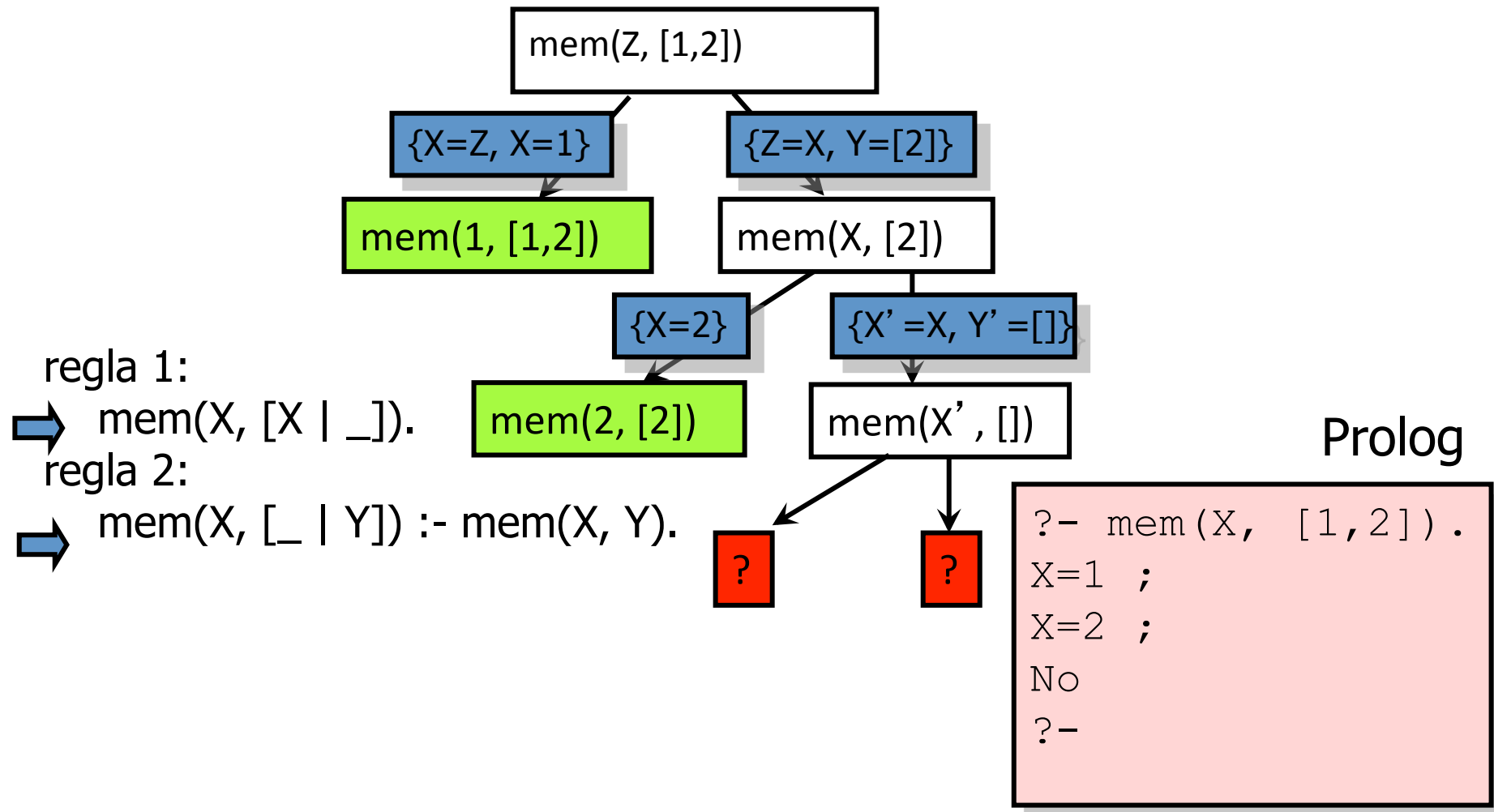
# Flight Planning: Backtracking



# unificación

- dos términos son unificables si hay una sustitución de variables que hace que puedan llegar a ser el mismo
  - por ejemplo,  $f(X)$  y  $f(3)$  se unifican con  $[X=3]$ 
    - $f(f(Y))$  y  $f(X)$  se unifican con  $[X=f(Y)]$
    - y  $g(X,Y)$  y  $f(3)$ ?
- la asignación de valores a las variables durante la resolución se llama instanciación
- la unificación es un proceso de pattern-matching que determina qué instanciaciones se pueden hacer a las variables durante una serie de resoluciones

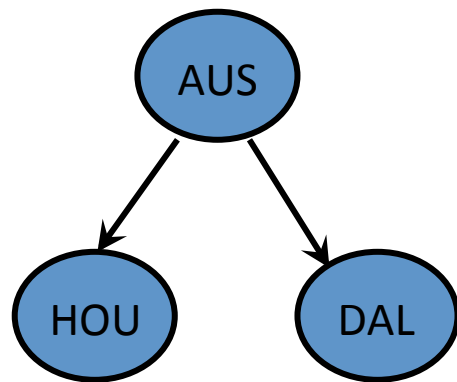
# ejemplo: está en la lista



# completitud


el procedimiento de búsqueda de Prolog devuelve cosas que son todas ciertas, pero no se puede probar todo lo que es cierto (es incompleto)

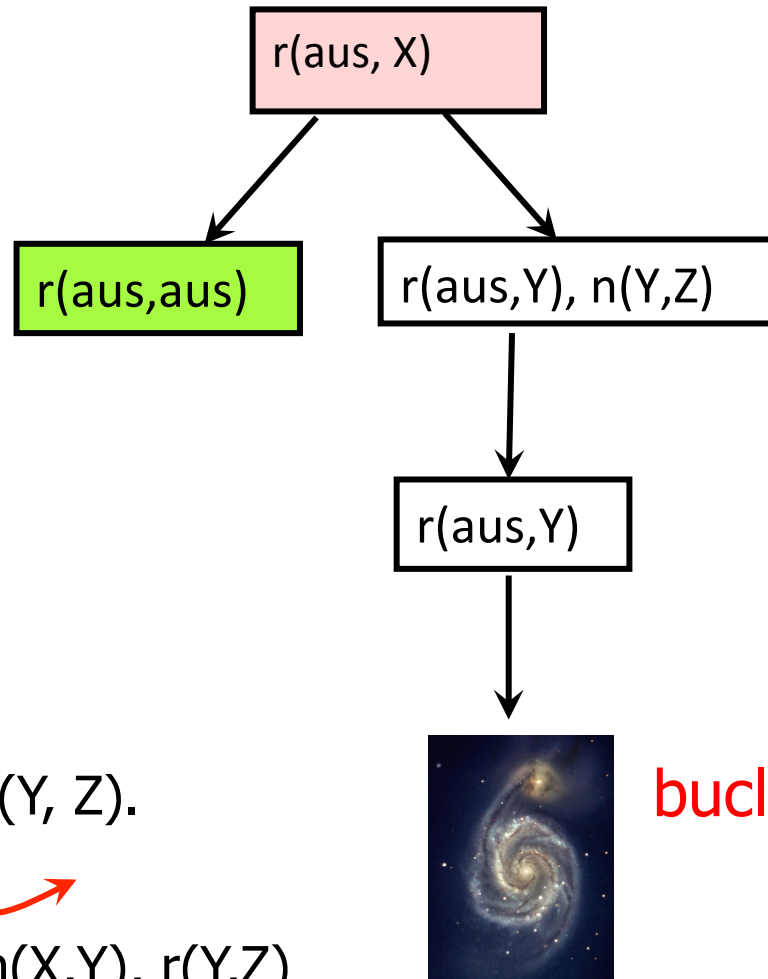
# planificación de vuelos: un cambio



Rule 1:  
 $r(X, X).$

Rule 2:  
 $r(X, Z) :- r(X, Y), n(Y, Z).$

  
en lugar de  $n(X, Y), r(Y, Z)$



Solution  
 $r(aus, aus)$

bucle infinito



# el operador “Is”

- **is** instancia una variable temporal, comparable a una variable local en lenguajes tipo Algol

ejemplo:

```
?- factorial(0, 1).  
?- factorial(N, Result) :-  
    N > 0,  
    M is N - 1,  
    factorial(M, SubRes),  
    Result is N * SubRes.
```

# traza

- la traza sirve para que el programador pueda ver cómo funciona una búsqueda de prueba

ejemplo

```
?- factorial(0, 1).
```

```
?- factorial(N, Result) :-
```

```
    N > 0, M is N - 1,
```

```
    factorial(M, SubRes), Result  
is N * SubRes.
```

```
?- trace(factorial/2).
```

- el argumento de “trace” incluye la aridad de la función

```
?- factorial(4, X).
```

# traza de factorial

- ?- factorial(4, X).
- Call: ( 7) factorial(4, \_G173)
- Call: ( 8) factorial(3, \_L131)
- Call: ( 9) factorial(2, \_L144)
- Call: (10) factorial(1, \_L157)
- Call: (11) factorial(0, \_L170)
- Exit: (11) factorial(0, 1)
- Exit: (10) factorial(1, 1)
- Exit: ( 9) factorial(2, 2)
- Exit: ( 8) factorial(3, 6)
- Exit: ( 7) factorial(4, 24)
- X = 24

estas son  
variables  
temporales



estos son los  
niveles en el  
árbol de búsqueda

# el cut

- cuando se inserta en la parte derecha de la regla, el operador cut **!** fuerza a que no se revisiten los subobjetivos si la parte derecha encuentra un resultado una vez

- ejemplo:

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

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

devuelve  
una sola  
respuesta

# el cut

- cuando se inserta en la parte derecha de la regla, el operador cut **!** fuerza a que no se revisiten los subobjetivos si la parte derecha encuentra un resultado una vez

- ejemplo:

devuelve  
una sola  
respuesta

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

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

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

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

# cuts rojos vs. cuts verdes

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

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

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

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

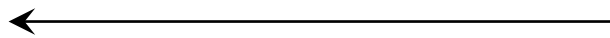
`max(X, Y, Z) :- X =< Y, !, Y = Z.`

`max(X, Y, X).`

# Tracing Bubble Sort

- ?- bsort([5,2,3,1], Ans).
- Call: ( 7) bsort([5, 2, 3, 1], \_G221)
- Call: ( 8) bsort([2, 5, 3, 1], \_G221)
- ...
- Call: ( 12) bsort([1, 2, 3, 5], \_G221)
- Redo: ( 12) bsort([1, 2, 3, 5], \_G221)
- ...
- Exit: ( 7) bsort([5, 2, 3, 1], [1, 2, 3, 5])
- Ans = [1, 2, 3, 5] ;
- No

Without the cut, this  
would have given some  
wrong answers



# negación

- el operador `not` se implementa como fallo del objetivo

`not(G) :- G, !, fail`

- “fail” es un objetivo especial que siempre falla

## ejemplo

```
factorial(N, 1) :- N < 1.
```

```
factorial(N, Result) :- not(N < 1), M is N - 1,  
                        factorial(M, P),  
                        Result is N * P.
```