Técnicas Básicas de Programación Prolog

Ingeniería Informática
Ingeniería Técnica en Informática de Sistemas

Departamento de Lenguajes y
Ciencias de la Computación
Universidad de Málaga

Contenido

Programación Recursiva

Recursión y Prolog

Prolog carece de mecanismos iterativos

Prolog emplea la **recursión** para:

- representar información (estructuras recursivas)
- procesar la información (relaciones recursivas)

Ejemplo: la aritmética de Peano
objetos → naturales
relaciones → operaciones y relaciones aritméticas

Cuidado: más adelante emplearemos aritmética **extra-lógica**!

Los naturales de Peano

Definición: el conjunto **inductivo** de los naturales $\mathbb{N}$

1. $0 \in \mathbb{N}$ [base]
2. si $X \in \mathbb{N}$, entonces $\theta(X) \in \mathbb{N}$ [recursivo]

θ representa a la función **sucesor** θ: $\mathbb{N} \rightarrow \mathbb{N}$

$\mathbb{N} = \{0, \theta(0), \theta(\theta(0)), \theta(\theta(\theta(0))), \ldots\}$

base

recursivo

Cuidado: $\theta(\theta(\theta(0))) \neq 3$, θ es un **constructor**

Los naturales son recursivos: $\theta(X)$ "contiene" a X

Representación en Prolog (I)

Necesitamos términos Prolog para representar naturales

Seguimos la definición inductiva de $\mathbb{N}$:

1. $0 \in \mathbb{N}$ [base]
2. si $X \in \mathbb{N}$, entonces $\theta(X) \in \mathbb{N}$ [recursivo]

Caso base → constante `c`

Caso recursivo → estructura functor `s/1`

Un **natural bien formado** es un término Prolog generado por la gramática $G_{\mathbb{N}}$:

```
N ::= c          [base]
    | s(N)       [recursivo]
```


Representación en Prolog (y II)

Decimal	Peano	Prolog
0	0	`c`
1	$\theta(0)$	`s(c)`
2	$\theta(\theta(0))$	`s(s(c))`
3	$\theta(\theta(\theta(0)))$	`s(s(s(c)))`
...	...	...
n	$\theta(\ldots(\theta(0))\ldots)$	`s(...(s(c))...)`
	n veces	n veces

Ejercicio:

1. ¿qué representa el término `s(s(s(X)))`?

Pero Prolog no tiene tipos

Los siguientes términos **no** son naturales bien formados:

`s(s(f(s(c))))` `s(s(a))` `s((s(0)))` `s(2)`

Pero Prolog los acepta porque no podemos restringir la aplicación del functor `s/1`

Solución: introducir un predicado `es_natural(X)` que tenga éxito cuando `X` sea un natural bien formado.

Definición extensional de es_natural/1

```
es_natural(c).
es_natural(s(c)).
es_natural(s(s(c))).
es_natural(s(s(s(c)))).
…
```


Definición intensional de es_natural/1

Seguimos la definición inductiva de $\mathbb{N}$:

1. $0 \in \mathbb{N}$ [base]
2. si $X \in \mathbb{N}$, entonces $\theta(X) \in \mathbb{N}$ [recursivo]

```
es_natural(c).                    [base]
es_natural(s(X)) :- es_natural(X). [recursivo]
```

En general, para cada tipo a representar en Prolog daremos:

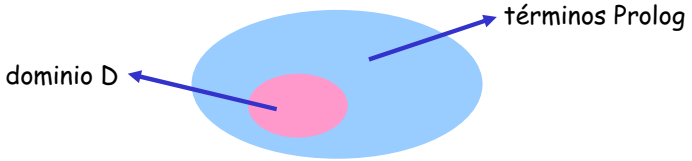
1. conjunto de valores (conjunto inductivo $\mathbb{N}$)
2. representación sintáctica (gramática $G_{\mathbb{N}}$)
3. definición de dominio (predicado `es_natural/1`)

Definición de dominio o tipo

`es_natural/1` es una **definición de dominio o tipo**

Se llama **dominio** a un conjunto de términos Prolog

Dado un dominio D, se llama **definición de dominio** o **tipo** a un predicado de aridad 1 que tiene éxito si y sólo si su argumento es un elemento del dominio D



La definición de dominio para D suele llamarse `es_d/1`

Comprobando naturales

```
:- es_natural(s(s(s(c)))).
Yes

:- es_natural(c).
Yes

:- es_natural(s(s(f(c)))).
No

:- es_natural(s(s(0)).
No
```

[en pizarra/SLD-Draw]

Generando naturales

```
:- es_natural(X).
X = c ;
X = s(c) ;
X = s(s(c)) ;
X = s(s(s(c))) ;
...

:- es_natural(s(s(Y))).
Y = c ;
Y = s(c) ;
...
```

[en pizarra/SLD-Draw]

Comprobando y generando naturales

El predicado `es_natural/1` funciona de dos maneras distintas:

- **pertenencia a** $\mathbb{N}$: comprueba que el argumento sea un natural bien formado (un elemento de $\mathbb{N}$)

```
:- es_natural(s(s(c))).
```

- **generación de** $\mathbb{N}$: genera todos los elementos del conjunto $\mathbb{N}$, partiendo del caso base, `c`

```
:- es_natural(X).
```

¿De qué depende el comportamiento de `es_natural/1`?

Modo de un argumento

Un argumento o **parámetro actual** de un predicado `p/n` puede estar en dos modos:

Modo +: (entrada) el argumento no contiene variables libres, o las variables que contiene no resultan instanciadas en la ejecución de p/n

```
          +
:- es_natural(s(s(c))).
```

Modo -: (salida, entrada/salida) el argumento contiene variables libres que resultan instanciadas al ejecutar `p/n`

```
          -
:- es_natural(s(s(X))).
```


Uso de un predicado

Combinación de los modos + y - de los parámetros actuales de la invocación a un predicado

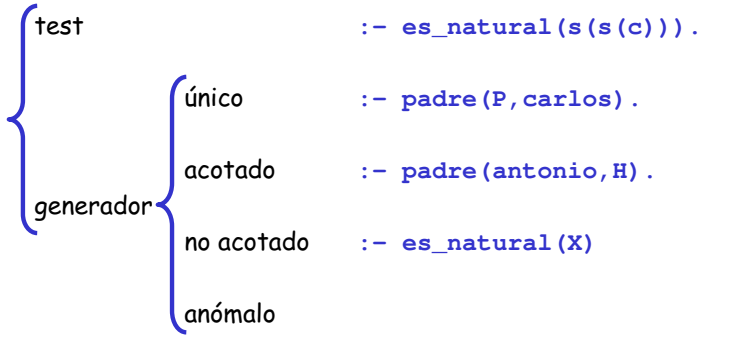
En general, un predicado de n argumentos, tendrá 2^n usos posibles:

```
:- es_natural(+).     :- padre(+,+).
:- es_natural(-).     :- padre(+,-).
                      :- padre(-,+).
                      :- padre(-,-).
```

No todos los usos serán útiles en la práctica (algunos no funcionarán)

Comportamiento de un predicado

Forma operacional en que se comporta un predicado para un **uso concreto**. Es una característica **cuantitativa**. Clasifica los usos de un predicado según el **número** de respuestas generadas:



Significado de un predicado

Cómputo particular llevado a cabo por un predicado para cada **uso concreto**. Es una característica **cualitativa**. Describe formalmente las respuestas computadas obtenidas

Para un **test**: describe los términos para los cuales tiene éxito

`es_natural(X)`, en uso +, tiene éxito si `X = s(…s(c)…)`

Para un **generador**: describe la secuencia de respuestas

`es_natural(X)`, en uso -, genera `X = c, s(c), s(s(c)),…`

Tabla de comportamiento de un predicado

`es_natural(X)`

Uso	Comportamiento	Significado
`(+)`	test	comprueba que `X` $\in \mathbb{N}$
`(-)`	generador no acotado	genera `X=c, s(c), s(s(c))..`

`padre(A,B)`

Uso	Comportamiento	Significado
`(+,+)`	test	comprueba que `A` es padre de `B`
`(+,-)`	generador acotado	genera en `B` los hijos de `A`
`(-,+)`	generador único	genera en `A` el padre de `B`
`(-,-)`	generador acotado	genera parejas de padres e hijos

La directiva mode

Podemos declarar los usos posibles de un predicado:

```
:- mode es_natural(+).
:- mode es_natural(-).
```

```
:- mode padre(+,+).
:- mode padre(+,-).
:- mode padre(-,+).
:- mode padre(-,-).
```

El comodín `?` (= `+/-`) permite abreviar las declaraciones:

```
:- mode es_natural(?).  :- mode padre(?,?).
```

Prolog comprueba los modos en tiempo de ejecución: sólo se pueden emplear los usos declarados.
SWI-Prolog **no comprueba** los modos, pero los emplea en la documentación (+ = entrada, - = salida, ? = entrada/salida)

Ejercicios

1. ¿Cómo se comporta el predicado `es_natural/1` si intercambiamos el orden del hecho y la regla?

```
es_natural(s(X)) :- es_natural(X).
es_natural(c).
```

Reconstruye la tabla de comportamiento y compara las semánticas declarativa y operacional.

2. Define los predicados `par/1` e `impar/1` utilizando recursión directa y mutua. Construye sus tablas de comportamiento y compáralas.

Operaciones como relaciones

Las operaciones aritméticas básicas se pueden representar por relaciones ternarias:

Z= X+Y → `suma(X,Y,Z)`
Z= X-Y → `resta(X,Y,Z)`
Z= X*Y → `producto(X,Y,Z)`
Z= X/Y → `cociente(X,Y,Z)`

- pasamos de un **estilo funcional** a un **estilo relacional**
- desaparece la distinción entre entrada y salida

La **recursión** jugará un papel fundamental en la definición **intensional** de las relaciones

La relación suma/3 (I)

`suma(X,Y,Z)` se satisface si y sólo si `X`, `Y` y `Z` son tres naturales tales que `Z= X+Y`

Aplicamos **recursión** al primer argumento:

```
suma(c,    ?, ?) :- …          [base]
suma(s(X), ?, ?) :- …          [recursivo]
```


La relación suma/3 (II)

`suma(X,Y,Z)` se satisface si y sólo si `X`, `Y` y `Z` son tres naturales tales que `Z= X+Y`

El segundo argumento es un natural arbitrario (no aplicamos recursión):

```
suma(c,    Y, ?) :- …          [base]
suma(s(X), Y, ?) :- …          [recursivo]
```

La relación suma/3 (III)

`suma(X,Y,Z)` se satisface si y sólo si `X`, `Y` y `Z` son tres naturales tales que `Z= X+Y`

El caso base es trivial (elemento neutro de la suma):

```
suma(c,    Y, Y).                  [base]
suma(s(X), Y, ?) :- …              [recursivo]
```


La relación suma/3 (IV)

`suma(X,Y,Z)` se satisface si y sólo si `X`, `Y` y `Z` son tres naturales tales que `Z= X+Y`

El caso recursivo se llama a sí mismo:

```
suma(c,    Y, Y).                  [base]
suma(s(X), Y, ?) :- suma(?,?,?).   [recursivo]
```


La relación suma/3 (V)

`suma(X,Y,Z)` se satisface si y sólo si `X`, `Y` y `Z` son tres naturales tales que `Z= X+Y`

En la llamada recursiva se **reduce** el problema:

```
suma(c,    Y, Y).                  [base]
suma(s(X), Y, ?) :- suma(X,Y,?).   [recursivo]
```

Reducimos `(X+1) + Y` a `X + Y`, un problema del mismo tipo pero "más pequeño"

La relación suma/3 (VI)

`suma(X,Y,Z)` se satisface si y sólo si `X`, `Y` y `Z` son tres naturales tales que `Z= X+Y`

En la llamada recursiva, **suponemos** que la solución es `Z`:

```
suma(c,    Y, Y).                  [base]
suma(s(X), Y, ?) :- suma(X,Y,Z).   [recursivo]
```

Inducción: lo suponemos para el caso n

La relación suma/3 (y VII)

`suma(X,Y,Z)` se satisface si y sólo si `X`, `Y` y `Z` son tres naturales tales que `Z= X+Y`

Partiendo de la solución de `X + Y`, **construimos** la solución de `(X+1) + Y`:

```
suma(c,    Y,Y).                    [base]
suma(s(X),Y,s(Z)) :- suma(X,Y,Z).   [recursivo]
```

Inducción: supuesto el caso n, demostramos el caso n+1

Pero Prolog no tiene tipos

`suma(X,Y,Z)` se satisface si y sólo si `X`, `Y` y `Z` son tres naturales tales que `Z= X+Y`

```
:- suma(c, a, a).
Yes

:- suma(s(s(s(c))), f(g(a)), Z).
Z = s(s(s(f(g(a))))) ;
No
```

¿Cómo podemos evitar este error?

Comprobación de tipos

`suma(X,Y,Z)` se satisface si y sólo si `X`, `Y` y `Z` son tres naturales tales que `Z= X+Y`

```
suma(c,Y,Y) :- es_natural(Y).        [base]
suma(s(X),Y,s(Z)) :- suma(X,Y,Z).    [recursivo]
```

Los errores anteriores son detectados:

```
:- suma(c,a,a).
No
```

Ejercicio: ¿Por qué no comprobamos los tipos del primer y tercer argumentos?

Usos de suma/3 (I)

Uso `(+,+,+)`, para comprobar sumas:

```
:- suma(s(s(c)), s(c), s(s(s(c)))).
Yes
```

Uso `(+,+,-)`, para sumar:

```
:- suma(s(s(s(c))), s(c), Z).
Z = s(s(s(s(c)))) ;
No
```

[en pizarra/SLD-Draw]

Usos de suma/3 (II)

Uso `(-,+,+)`, para restar:

```
:- suma(X, s(c), s(s(s(c)))).
X = s(s(c)) ;
No
```

Uso `(+,-,+)`, para restar:

```
:- suma(s(s(s(c))), Y, s(s(s(s(c))))).
Y = s(c) ;
No
```

[en pizarra/SLD-Draw]

Usos de suma/3 (III)

Uso `(-,-,+)`, para descomponer en sumandos:

```
:- suma(X, Y, s(s(c))).
X = c
Y = s(s(c)) ;
X = s(c)
Y = s(c) ;
X = s(s(c))
Y = c ;
...
```

[en pizarra/SLD-Draw]

Usos de suma/3 (IV)

Uso `(+,-,-)`, para generar a partir de un natural `X` dado:

```
:- suma(s(s(c)), Y, Z).
Y = c
Z = s(s(c)) ;
Y = s(c)
Z = s(s(s(c))) ;
Y = s(s(c))
Y = s(s(s(s(c)))) ;
...
```

[en pizarra/SLD-Draw]

Usos de suma/3 (y V)

Uso `(-,-,-)`, para nada (generador anómalo):

```
:- suma(X, Y, Z).
X = c
Y = c
Z = c ;
X = c
Y = s(c)
Z = s(c) ;
...
```

[en pizarra/SLD-Draw]

Tabla de comportamiento de suma/3

`suma(X,Y,Z)`

Uso	Comportamiento	Significado
(+,+,+)	test	comprueba que `Z=X+Y`
(+,+,-)	generador único	suma: `Z=X+Y`
(+,-,+)	generador único	resta: `Y= Z-X`
(-,+,+)	generador único	resta: `X= Z-Y`
(-,-,+)	generador acotado	{(`X`,`Y`) / `X` ∈ ℕ, `Y` ∈ ℕ, `Z=X+Y`}
(+,-,-)	generador no acotado	{(`Y`,`Z`) / `Y` ∈ ℕ, `Z=X+Y`}
(-,+,-)	generador no acotado	{(`X`,`Z`) / `X` ∈ ℕ, `Z=X+Y`}
(-,-,-)	generador anómalo	`X = c`, `Y` ∈ ℕ, `Z=X+Y`

Chequeo de tipos vs. respuestas genéricas

Eliminando la comprobación de tipos, sacrificamos la corrección obteniendo a cambio **respuestas genéricas**:

```
suma(c,Y,Y).                          [base]
suma(s(X),Y,s(Z)) :- suma(X,Y,Z).     [recursivo]
```

Ejemplo:

```
?- suma(s(s(c)),A,B).
A = _G295
B = s(s(_G295)) ;
No
```

Ejercicio: construir la tabla de comportamiento de la versión de `suma/3` sin la comprobación de tipo en el caso base

Patrón de predicado recursivo

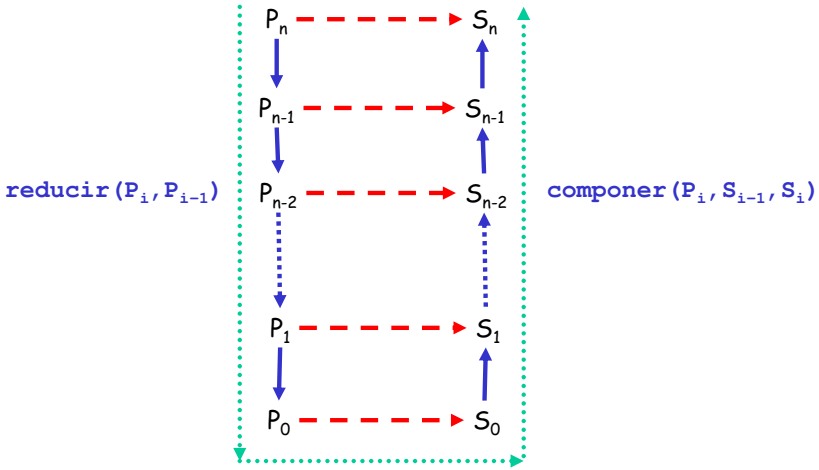
`resolver(P,S)` se satisface si `S` es solución del problema `P`

```
resolver(Caso_Base, Solución_Base):-
   Condiciones_Base.

resolver(Caso_recursivo, Solución) :-
   reducir(Caso_recursivo, Caso_menor),
   resolver(Caso_menor, Sol_menor),
   componer(Caso_recursivo,Sol_menor, Solución).
```

- puede haber varios casos base y recursivos
- la parte creativa es definir `reducir/2` y `componer/3`
- los casos base y recursivo no son necesariamente excluyentes

Recursión = Descenso + Ascenso



Ejercicios

Dados los naturales X, Y y Z, define las siguientes relaciones y construye sus tablas de comportamiento:

1. iguales(X,Y), X e Y son dos naturales iguales entre sí
2. menor(X,Y), X es menor que Y
3. mínimo(X,Y,Z), Z es el mínimo de X e Y
4. producto(X,Y,Z), Z es el producto de X por Y
5. exp(X,Y,Z), Z es X elevado a Y
6. mod(X,Y,Z), Z es el resto de dividir X por Y
7. mcd(X,Y,Z), Z es el máximo común divisor de X e Y
8. factorial(X,Y), Y es el factorial de X
9. fibonacci(X,Y), Y es el X-simo número de Fibonacci
10. entre(X,Y,Z), Z está entre X e Y

Recursión de cola y predicados iterativos

El coste de la iteración

Sea la iteración:

```
loop guarda(x,y)
      S1(x,z);
      S2(y,z);
      S3(y);
end loop;
```

Para N iteraciones, el coste es:

- **Tiempo**: *O*(N)
- **Espacio**: *O*(1)

Supuesto que las S_i son *O*(1) y no hay asignación dinámica

El coste de la recursión

```
procedure p(x,y,z)
begin
      if guarda(x,y) then
            S1(x,z);
            S2(y,z);
            S3(y);
            p(x,y,z);
      end if;
end;
```

Para N invocaciones, el coste es:

- **Tiempo**: *O*(N)
- **Espacio**: *O*(N) → un registro de activación por invocación

Recursión de cola

```
procedure p(x,y,z)
begin
      if guarda(x,y) then
            S1(x,z);
            S2(y,z);
            S3(y);
            p(x,y,z);
      end if;
end;
```

Un procedimiento es recursivo de cola cuando la llamada recursiva es la **última sentencia**

Optimización de la recursión de cola

```
procedure p(x,y,z)
begin
      if guarda(x,y) then
            S1(x,z);
            S2(y,z);
            S3(y);
            p(x,y,z);
      end if;
end;
```

no hay trabajo pendiente: **ejecución iterativa**

Para N invocaciones, el coste es:

- **Tiempo**: *O*(N)
- **Espacio**: *O*(1) → **reutiliza** el registro de activación

Recursión de cola en Prolog (I)

Cada lenguaje impone sus condiciones a la optimización (objetos, determinismo, etc).

En el caso de Prolog:

regla recursiva de cola: una regla recursiva **simple**, donde la llamada recursiva aparece como **última** condición del cuerpo

predicado recursivo de cola: todas sus reglas recursivas son recursivas de cola

predicado iterativo: predicado recursivo de cola cuya ejecución se puede optimizar (es iterativa)

Recursión de cola en Prolog (II)

Un predicado `p/n` es **iterativo** si y sólo si:

1) en cada regla recursiva, los predicados anteriores a la llamada recursiva no dejan alternativas por explorar

solución única

```
p(T1) :- r(T1,X), s(X), p(X).
```

2) las definiciones de `p/n` son excluyentes:

casos excluyentes

```
p(T1) :- ...
p(T2) :- ...
p(T3).
```

Es decir, la ejecución de `p/n` es **determinista**

Recursión de cola y Prolog (y III)

Prolog comprueba en **tiempo de ejecución** si la llamada recursiva es optimizable

Prolog aplica esta optimización a la **última llamada** de un procedimiento, sea recursiva o no

Para ser eficaz, esta optimización debe combinarse con otras técnicas:

- recolección de basura
- indexación de cláusulas

Ejemplo 1

El predicado `p/0`:

```
p :- q(X), p.

q(a).
q(b).
```

es recursivo de cola, pero no iterativo

Ejemplo 2

El predicado `p/0`:

```
p :- q, p.
p :- r, p.

q.
r.
```

es recursivo de cola, pero no iterativo

Ejemplo 3

El predicado `p/0`:

```
p :- q(a), p.

q(a).
q(b).
```

es recursivo de cola e iterativo (recursión infinita sin desborbamiento de pila)

Ejemplo 4

El predicado `p/0`:

```
p(X) :- q(X), p(X).

q(a).
q(b).
```

es recursivo de cola e iterativo (recursión infinita sin desborbamiento de pila)

¿Cómo definir predicados recursivos de cola?

La recursión de cola suele corresponder a bucles con acumuladores:

```
Acum:= valor_inicial;
loop guarda (...)
      ...
      actualizar(Acum);
      ...
end loop;
```

Al salir del bucle la variable `Acum` contiene la solución

Ejemplo: sumar los elementos de un vector, etc

Recursión de cola y acumuladores

Podemos traducir el anterior bucle a una recursión de cola:

```
procedure rec_de_cola(Problema, Acum)
begin
      if guarda (…) then
            reducir(Problema, Problema_Menor);
            actualizar(Acum);
            rec_de_cola(Problema_Menor, Acum);
      end if;
end;
```

¿Cómo traducir este procedimiento a Prolog?

Primer intento

```
rec_de_cola(Caso_Base, Acum) :-
      condiciones_base.

rec_de_cola(Caso_Recursivo, Acum) :-
      reducir(Caso_Recursivo, Caso_Menor),
      actualizar(Acum),
      rec_de_cola(Caso_Menor, Acum).
```

¿Por qué no funciona?

Segundo intento

```
rec_de_cola(Caso_Base, Acum) :-
      condiciones_base.

rec_de_cola(Caso_Recursivo, Acum) :-
      reducir(Caso_Recursivo, Caso_Menor),
      actualizar(Acum, NAcum),
      rec_de_cola(Caso_Menor, NAcum).
```

¿Por qué no funciona?

Patrón de predicado recursivo de cola

```
rec_de_cola(Caso_Base, Sol, Sol) :-
      condiciones_base.

rec_de_cola(Caso_Recursivo, Acum, Sol) :-
      reducir(Caso_Recursivo, Caso_Menor),
      actualizar(Acum, NAcum),
      rec_de_cola(Caso_Menor, NAcum, Sol).
```

Los pasos creativos son `reducir/2` y `actualizar/2`

Interfaz de recursión de cola

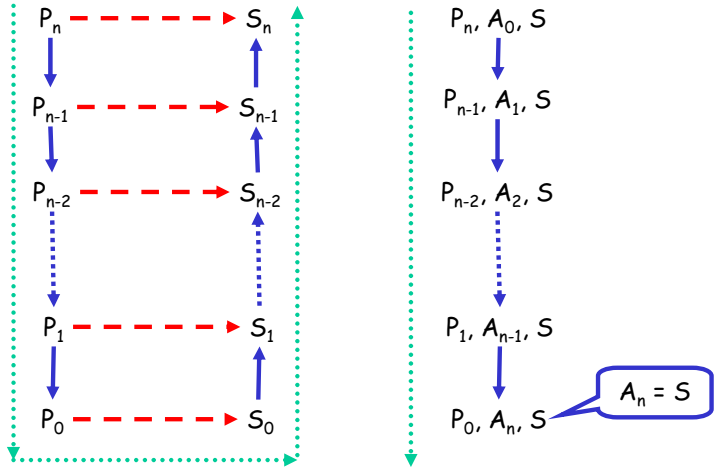
Debemos ocultar al usuario el empleo de la recursión de cola:

```
resolver(Problema, Solución):-
      rec_de_cola(Problema, Sol_Base, Solución).

rec_de_cola(Caso_Base, Sol, Sol) :-
      condiciones_base.

rec_de_cola(Caso_Recursivo, Acum, Sol) :-
      reducir(Caso_Recursivo, Caso_Menor),
      actualizar(Acum, NAcum),
      rec_de_cola(Caso_Menor, NAcum, Sol).
```


Recursión y recursión de cola comparadas



Ejemplo: el factorial (I)

Aplicando **mecánicamente** el patrón obtenemos:

```
factorial(X,Y) :-
      fact_cola(X,s(c),Y).

fact_cola(c,Sol,Sol).

fact_cola(s(X),Acum,Y) :-
      reducir(?,?),
      actualizar(Acum,NAcum),
      fact_cola(?,NAcum,Y).
```

`reducir/2` es trivial

Ejemplo: el factorial (II)

```
factorial(X,Y) :-
      fact_cola(X,s(c),Y).

fact_cola(c,Sol,Sol).

fact_cola(s(X),Acum,Y) :-
      actualizar(Acum,NAcum),
      fact_cola(X,NAcum,Y).
```

¿En qué consiste actualizar?

Ejemplo: el factorial (y III)

```
factorial(X,Y) :-
      fact_cola(X,s(c),Y).

fact_cola(c,Sol,Sol).

fact_cola(s(X),Acum,Y) :-
      producto(s(X),Acum,NAcum),
      fact_cola(X,NAcum,Y).
```


Anticipando el trabajo pendiente

```
fact(c,s(c)).
fact(s(X),Y) :-
      fact(X,T),
      producto(s(X),T,Y).

fact_cola(c,Sol,Sol).
fact_cola(s(X),Acum,Y) :-
      producto(s(X),Acum,NAcum),
      fact_cola(X,NAcum,Y).
```

La actualización del acumulador corresponde a anticipar el trabajo que deja pendiente la versión recursiva

La transformación no siempre es trivial

Ejercicios

Define versiones recursivas de cola de los siguientes predicados y comprueba si los predicados son iterativos:

1. producto(X,Y,Z), Z es el producto de X por Y
2. exp(X,Y,Z), Z es X elevado a Y
3. fibonacci(X,Y), Y es el X-simo número de Fibonacci

El paradigma generar/comprobar

El paradigma generar/comprobar

`resolver(P,S)` - `S` es solución del problema `P`

```
                  +        -
resolver(Problema, Solución) :-
                      +        -
        generar(Problema, Candidato),
                      +        +
        comprobar(Problema, Candidato),
            -          +
        Solución = Candidato.
```

- apropiado para problemas sin solución algorítmica
- método de prueba y error
- basado en generadores y retroceso

Principios de diseño

La clave está en diseñar un buen generador:

- evitar generadores anómalos y no acotados
- emplear generadores acotados
- **completitud**: las soluciones son un subconjunto de los candidatos
- **eficiencia**: generar tan pocos candidatos como sea posible (sin perder completitud)
- **descomponer** generar y comprobar en operaciones más simples para **entrelazar** las fases de generación y comprobación
- empezar por los generadores/comprobadores **más restrictivos**
- aprovechar **determinismo**

Ejemplo: descomposición en pares

Problema:

Dado un natural `N`, encontrar 2 naturales `X` e `Y` tales que:

1. `N= X+Y`
2. `X` e `Y` son pares

Escribiremos una solución empleando generar/comprobar:

```
en_pares(N,X,Y) :- ...
```


Primera solución

```
          + - -
en_pares(N,X,Y) :-
                -
     es_par(X),          % generar
                -
     es_par(Y),          % generar
          + + +
     suma(X,Y,N).        % comprobar
```


Segunda solución

```
          + - -
en_pares(N,X,Y) :-
                - +
     menor_ig(X,N),      % generar
                - +
     menor_ig(Y,N),      % generar
          + + +
     suma(X,Y,N),        % comprobar
             +
     es_par(X),          % comprobar
             +
     es_par(Y).          % comprobar
```


Tercera solución

```
          + - -
en_pares(N,X,Y) :-
                  - +
     menor_ig(X,N),      % generar
             +
     es_par(X),          % comprobar
          + - +
     suma(X,Y,N),        % generar
             +
     es_par(Y).          % comprobar
```

Cuarta solución

```
          + - -
en_pares(N,X,Y) :-
            - - +
      suma(X,Y,N),      % generar
            +
      es_par(X),        % comprobar
            +
      es_par(Y).        % comprobar
```


Quinta solución

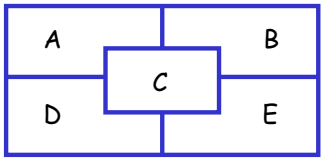
```
          + - -
en_pares(N,X,Y) :-
            +
      es_par(N),        % comprobar
            - - +
      suma(X,Y,N),      % generar
            +
      es_par(X).        % comprobar
```

Conclusiones:

- misma semántica declarativa, distinta operacional
- distinta eficiencia
- el **papel** de un predicado **depende de su uso**

Ejercicios

1. Dado el mapa de la figura, encontrar un coloreado que asigne colores **distintos** a regiones **fronterizas**, **minimizando** el número de colores




```
colorear(A,B,C,D,E) :- …
```

2. Calcular el cociente y el resto de la división entera X/Y:

```
coc_rest(X,Y,Cociente,Resto) :- ...
```


Relaciones y Bases de Datos Relacionales

Relaciones binarias

Definición: Dado un dominio $\mathcal{D}$, $\mathcal{R}$ es una **relación binaria** sobre $\mathcal{D}$ sii

$$\mathcal{R} \subseteq \mathcal{D} \times \mathcal{D}$$

Propiedades:

- reflexiva: $\forall\ x \in \mathcal{D}.\ (x, x) \in \mathcal{R}$
- simétrica: $\forall\ x, y.\ (x, y) \in \mathcal{R} \Rightarrow (y, x) \in \mathcal{R}$
- transitiva: $\forall\ x, y, z.\ (x, y) \in \mathcal{R} \wedge (y, z) \in \mathcal{R} \Rightarrow (x, z) \in \mathcal{R}$

Cierre o clausura de una relación binaria

Definición: Dada una relación $\mathcal{R}$, llamamos **cierre** o **clausura** de $\mathcal{R}$ a una relación $\mathcal{R}'$ tal que $\mathcal{R} \subseteq \mathcal{R}'$

$\mathcal{R}'$ se obtiene añadiendo tuplas a $\mathcal{R}$

Podemos cerrar $\mathcal{R}$ añadiendo el número **mínimo** de tuplas tal que $\mathcal{R}'$ satisfaga cierta propiedad (reflexiva, simétrica,…)

Ejemplo: cierre reflexivo

$$\mathcal{R} \rightarrow \mathcal{R}' = \mathcal{R} \cup \Delta$$

Δ son las tuplas que faltan a $\mathcal{R}$ para ser reflexiva

Relaciones binarias en Prolog

Dados:

- un dominio $\mathcal{D}$ definido mediante una definición de dominio:

```
es_d(X) :- …
```

- una relación binaria $\mathcal{R}$ sobre $\mathcal{D}$, definida por el predicado:

```
r(X,Y) :- …
```

definiremos diferentes cierres de $\mathcal{R}$ en Prolog

Ejemplo

```
hombre(juan).          mujer(elena).
hombre(pedro).         mujer(maria).
hombre(salvador).      mujer(eva).
hombre(miguel).        mujer(silvia).

persona(X) :- hombre(X).    % dominino D
persona(Y) :- mujer(Y).

misma_edad(juan,pedro).     % relación R
misma_edad(salvador,elena).
misma_edad(pedro,maria).
misma_edad(salvador,eva).
misma_edad(silvia,miguel).
```

Cierre reflexivo

Definición:

$$\mathcal{R} \rightarrow \mathcal{R}' = \mathcal{R} \cup \{ (x,x) / x \in \mathcal{D} \}$$

```
r_reflex(X,Y) :- r(X,Y).
r_reflex(X,X) :- es_d(X).
```

Ejemplo:

```
misma_edad_reflex(X,Y) :- misma_edad(X,Y).
misma_edad_reflex(X,X) :- persona(X).
```


Cierre simétrico

Definición:

$$\mathcal{R} \rightarrow \mathcal{R}' = \mathcal{R} \cup \{ (y, x) / (x, y) \in \mathcal{R} \}$$

```
r_sim(X,Y) :- r(X,Y).
r_sim(Y,X) :- r(X,Y).
```

Ejemplo:

```
misma_edad_sim(X,Y) :- misma_edad(X,Y).
misma_edad_sim(Y,X) :- misma_edad(X,Y).
```


Cierre transitivo

Definición:

$$\mathcal{R} \rightarrow \mathcal{R}' = \mathcal{R} \cup \{ (x, y) / (x, t) \in \mathcal{R} \wedge (t,y) \in \mathcal{R}' \}$$

(?)

```
r_trans(X,Y) :- r(X,Y).
r_trans(X,Y) :- r(X,T), r_trans(T,Y).
```

Ejemplo:

```
misma_edad_trans(X,Y) :- misma_edad(X,Y).
misma_edad_trans(X,Y) :-
      misma_edad(X,T),
      misma_edad_trans(T,Y).
```


Relación de preorden (orden parcial)

Definición: un preorden es una relación reflexiva y transitiva

```
r_preorden(X,X) :- es_d(X).
r_preorden(X,Y) :- r(X,T), r_preorden(T,Y).
```

¿Por qué eliminamos uno de los casos base?

Ejemplo:

```
misma_edad_pre(X,X) :- persona(X).
misma_edad_pre(X,Y) :-
      misma_edad(X,T),
      misma_edad_pre(T,Y).
```

Relación de equivalencia

Definición: una equivalencia es una relación reflexiva, simétrica y transitiva

```
r_eq(X,X) :- es_d(X).
r_eq(X,Y) :- r_sim(X,T), r_eq(T,Y).
```

¿Por qué lo definimos a través de la relación `r_sim/2`?

Ejemplo:

```
misma_edad_eq(X,X) :- persona(X).
misma_edad_eq(X,Y) :-
        misma_edad_sim(X,T),
        misma_edad_pre(T,Y).
```


Aplicación: búsqueda en grafos

Dado un dominio de nodos (`es_nodo/1`), un **digrafo** se puede representar por una relación binaria `arco/2`:

```
arco(a,b).  arco(b,c).  arco(c,d).  arco(e,f).
arco(a,c).  arco(b,e).  arco(c,f).
arco(a,d).              arco(c,g).
```

Un **camino** en un digrafo es un cierre reflexivo y transitivo:

```
hay_camino(X,X) :- es_nodo(X).
hay_camino(X,Y) :- arco(X,Z), hay_camino(Z,Y).
```

Ejercicio: ¿Cómo afecta el orden de la cláusulas? ¿Y si el grafo tiene ciclos? ¿Y si el grafo no es dirigido?

Aplicación: autómatas finitos

Dada una relación de transición $\delta : \mathcal{Q} \rightarrow \mathcal{A} \rightarrow \mathcal{Q}$

```
delta(q0, a, q1).
delta(q0, b, q3).
...
```

La aceptación de una cadena se basa en un cierre transitivo:

```
acepta(Qi,'$') :- es_final(Qi).
acepta(Qi,cadena(A,Resto)) :-
        delta(Qi,A,Qj),
        acepta(Qj,Resto).
```


Bases de datos relacionales

dominio: conjunto finito de elementos atómicos

relación o tabla: subconjunto de un producto cartesiano de dominios

$$\mathcal{R} \subseteq \mathcal{D}_1 \times \mathcal{D}_2 \times \ldots \mathcal{D}_{n-1} \times \mathcal{D}_n$$

tupla: elemento de una relación

atributo: identificador $\mathcal{A}_i$ de la dimensión i de una relación

esquema de una relación: tupla de sus atributos

$$\mathcal{R}\,(\mathcal{A}_1,\ \mathcal{A}_2,\ \ldots,\ \mathcal{A}_{n-1},\ \mathcal{A}_n)$$

base de datos relacional: conjunto de relaciones

vista: relación creada a partir de otras relaciones y vistas, aplicando el álgebra relacional

Ejemplo

dominios: Varón = { javier, eduardo, jacobo, felipe, juan }
Mujer = { sonia, elisa, maribel, mónica, alicia }
Persona = Varón ∪ Mujer

relación: padre(Padre, Hijo) ⊆ Varón × Persona

Padre	**Hijo**
eduardo	maribel
javier	eduardo
javier	felipe
felipe	mónica
…	…

Lógica y bases de datos relacionales (I)

Las cláusulas de Horn y la **negación** pueden expresar dominios, relaciones y vistas

dominios: definiciones de dominios

```
varon(javier).          mujer(sonia).
varon(eduardo).         mujer(elisa).
varon(jacobo).          mujer(maribel).
varon(felipe).          mujer(mónica).
varon(juan).            mujer(alicia).

persona(X) :- varon(X).
persona(X) :- mujer(X).
```


Lógica y bases de datos relacionales (II)

relaciones: un hecho por cada tupla

```
padre(eduardo, maribel).
padre(javier, eduardo).
padre(javier, felipe).
padre(felipe, mónica).
….
```

inconvenientes:

- consistencia padre(maribel, jacobo)
  se puede resolver comprobando los dominios

- actualizaciones
  difícil de modelar (problema del marco)

Lógica y bases de datos relacionales (y III)

vistas: reglas y consultas

Ejemplo:

```
:- padre(X,_Y), padre(_Y, Z).    % abuelo(X,Z)

hermanos(X,Y) :-                 % hermanos(X,Y)
          progenitor(P,X),
          progenitor(P,Y),
          X\=Y.
```

Las **vistas** se expresan en un **lenguaje de consultas** (ej. SQL) que se traduce a expresiones del **álgebra relacional**

Basta expresar en Prolog las operaciones del álgebra relacional

Unión

Dadas $\mathcal{R}_1$ → `r1(X1,…Xn)` y $\mathcal{R}_2$ → `r2(X1,…Xn)`

$\mathcal{R}_1 \cup \mathcal{R}_2 = \{ (X_1, \ldots, X_n) / (X_1, \ldots, X_n) \in \mathcal{R}_1 \vee (X_1, \ldots, X_n) \in \mathcal{R}_2 \}$

```
r1ur2(X1,…,Xn) :- r1(X1,…,Xn).
r1ur2(X1,…,Xn) :- r2(X1,…Xn).
```

Ejemplo:

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


Diferencia

Dadas $\mathcal{R}_1$ → `r1(X1,…Xn)` y $\mathcal{R}_2$ → `r2(X1,…Xn)`

$\mathcal{R}_1 \setminus \mathcal{R}_2 = \{ (X_1, \ldots, X_n) / (X_1, \ldots, X_n) \in \mathcal{R}_1 \wedge (X_1, \ldots, X_n) \notin \mathcal{R}_2 \}$

```
r1_r2(X1,…,Xn) :-
        r1(X1,…,Xn),
        \+ r2(X1,…Xn).
```

Ejemplo:

```
padre(X,Y) :-
        progenitor(X,Y),
        \+ madre(X,Y).
```


Producto cartesiano

Dadas $\mathcal{R}_1$ → `r1(X1,…Xn)` y $\mathcal{R}_2$ → `r2(Y1,…,Ym)`

$\mathcal{R}_1 \times \mathcal{R}_2 = \{ (X_1, \ldots, X_n, Y_1, \ldots, Y_m) / (X_1, \ldots, X_n) \in \mathcal{R}_1 \wedge (Y_1, \ldots, Y_m) \in \mathcal{R}_2 \}$

```
r1yr2(X1,…,Xn,Y1,…,Ym) :-
        r1(X1,…,Xn),
        r2(Y1,…Ym).
```

Ejemplo:

```
pareja(X,Y) :-
        persona(X),
        persona(Y).
```


Proyección

Dada $\mathcal{R}$ → `r1(X1,…Xn)`

$\Pi_i \mathcal{R} = \{ X_i / (X_1, \ldots, X_n) \in \mathcal{R} \}$

```
r_i(Xi):-
        r(X1,…,Xi,…,Xn).
```

Ejemplo:

```
es_padre(X) :-
        padre(X,_). % variable anónima
```

Selección

Dadas $\mathcal{R}$ → `r1(X1,…Xn)` y $f : \mathcal{D}_1 \times \ldots \times \mathcal{D}_n \rightarrow$ Bool

$$\sigma_f \mathcal{R} = \{(X_1, \ldots, X_n) / (X_1, \ldots, X_n) \in \mathcal{R} \wedge f(X_1, \ldots, X_n)\}$$

```
r_f(X1,…,Xn):-
        r(X1,…, Xn),
        f(X1,…,Xn).   % función booleana
```

Ejemplo:

```
mayor_de_edad(X) :-
        edad(X,N),
        N >= 18.      % extra-lógica
```