

Programación Declarativa: Lógica y Restricciones

El lenguaje de programación ISO-Prolog

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



POLITÉCNICA

Contenidos

- Predicados para Tipos
- Aritmética
- Acceso a Estructuras
- Predicados Meta-Lógicos
- Comparación de Términos
- Entrada/Salida
- Meta-Programación
- Modificación Dinámica
- *Parsing*

Predicados para Tipos

■ Términos Prolog:

- ❑ constante
 - átomo
 - número
 - entero
 - real
- ❑ variable
- ❑ estructura

■ Predicados para tipos: tienen éxito o fallan, pero no producen error

- ❑ integer(X)
- ❑ float(X)
- ❑ number(X)
- ❑ atom(X) → X es un término constante (aridad 0) no numérico
- ❑ atomic(X) → X es una constante (átomo o número)
- ❑ compound(X) → X es una estructura

Predicados para Tipos: Ejemplos

- ?- atom(vacio).
 - ☐ Yes
- ?- integer(-3).
 - ☐ Yes
- ?- compound([a,b | Xs]).
 - ☐ Yes
- ?- compound(-3.14).
 - ☐ No
- ?- integer([1]).
 - ☐ No
- ?- X = 2, integer(X).
 - ☐ Yes
- ?- compound([X]).
 - ☐ Yes

Aritmética (I)

■ Términos aritméticos

- Un número es un término aritmético
- Si f es un functor aritmético y X_1, \dots, X_n son términos aritméticos, entonces $f(X_1, \dots, X_n)$ es un término aritmético

■ Functores aritméticos

- $+$, $-$, $*$, $/$ (cociente), $//$ (cociente entero), mod (módulo), etc.

■ Una expresión aritmética sólo puede ser evaluada si no contiene variables libres. En otro caso aparece un error de evaluación

- $(3*X+Y)/Z \rightarrow$ correcta si cuando se evalúan X , Y , y Z son términos aritméticos, en otro caso se produce un error
- $a+3*X \rightarrow$ se produce un error ('a' no es un término aritmético)

Aritmética (II)

■ Predicados aritméticos

- $<$, $>$, $=<$, $>=$, $:=$ (igualdad aritmética), \neq (desigualdad aritmética), etc.
 - Ambos argumentos se evalúan y se comparan los resultados
- $Z \text{ is } X$
 - X , que debe ser un término aritmético, se evalúa y el resultado se unifica con Z

■ Ejemplos: supongamos que X vale 3 e Y 4, y que Z es una variable libre

- $Y < X+1$, $X \text{ is } Y+1$, $X := Y$. \rightarrow fallo
- $Y < a+1$, $X \text{ is } Z+1$, $X := f(a)$. \rightarrow error

Aritmética (III)

■ Ejemplos:

❑ ?- X is $4/2 + 3/7$.

▪ $X = 2.42857$

❑ ?- X is $2*4$, 2 is $X//3$.

▪ $X = 8$

❑ ?- X is 3, Y is $X+4$.

▪ $X = 3, Y = 7$

❑ ?- Y is $X+4$, X is 3.

▪ Error (porque X está libre)

❑ ?- $3+4$ is $3+4$.

▪ no

▪ La parte izquierda no unifica con 7 (resultado de evaluar la expresión)

❑ X is $X+1$

▪ Fracaso si X está instanciada en la llamada

▪ Error aritmético si X está libre

El orden de los literales es relevante en el uso de predicados evaluables

Aritmética: Ejemplo 1

- `plus(X,Y,Z) :- Z is X + Y`
 - Sólo funciona en modo (in, in, out). X e Y deben ser términos aritméticos
- Otra implementación de `plus/3`
 - `plus(X,Y,Z):- number(X),number(Y), Z is X + Y.` %in-in-out
 - `plus(X,Y,Z):- number(X),number(Z), Y is Z - X.` %in-out-in
 - `plus(X,Y,Z):- number(Y),number(Z), X is Z - Y.` %out-in-in
- Predicado 'suma' entre enteros para cubrir el caso en el que los sumandos puedan no estar instanciados pero el resultado sí

Aritmética: Ejemplo 2

■ Factorial usando aritmética de Peano

- ❑ `factorial(0,s(0)).`
- ❑ `factorial(s(N),F):- factorial(N,F1), times(s(N),F1,F).`

■ Factorial usando aritmética Prolog

- ❑ `factorial(0,1).`
- ❑ `factorial(N,F):-`
 - `N > 0,`
 - `N1 is N-1,`
 - `factorial(N1,F1),`
 - `F is F1*N.`

[factorial.pl](#)

Aritmética: Ejercicio 1

- Sucesión de Fibonacci: 0,1,1,2,3,5,8,13,21, ...
 - cada término, salvo los dos primeros, es la suma de los dos anteriores
- Definir el predicado `fibonacci/2` (`fibonacci(N,X)`) que se verifique si X es el N -ésimo término de la sucesión de Fibonacci.
 - `?- fibonacci(6,X).`
 - $X = 8$

Aritmética: Ejercicio 2

- Definir `lista_numeros/3` (`lista_numeros(N,M,L)`) que se verifica si `L` es la lista de los números entre `N` y `M`, ambos inclusive
 - ?- `lista_numeros(3,5,L)`.
 - `L = [3,4,5]`
 - ?- `lista_numeros(3,2,L)`.
 - `no`

[lista_numeros.pl](#)

Aritmética: Ejercicio 3

- Definir el predicado **mcd/3** ($\text{mcd}(X,Y,Z)$) que se verifique si Z es el máximo común divisor de X e Y
 - $?- \text{mcd}(10,15,X).$
 - $X=5$

Acceso a Estructuras

- Los meta-predicados de **inspección de estructuras** permiten:
 - Descomponer una estructura en sus componentes
 - Componer una estructura a partir de sus componentes
- Prolog proporciona 3 meta-predicados de inspección:
 - `functor/3`
 - `arg/3`
 - `=../2`

Acceso a Estructuras: functor/3

■ `functor(E, F, A)`: la estructura E tiene functor F y aridad A

- E es un término compuesto $f(X_1, \dots, X_n) \rightarrow F=f, A=n$
- F es el átomo f y A es el entero n $\rightarrow X=f(X_1, \dots, X_n)$

■ Ejemplos:

- ?- `functor(padre(juan,jose),padre,2)`.
 - `yes`
- ?- `functor(libro(autor, titulo), N, A)`.
 - `N = libro, A = 2`
- ?- `functor(X, libro, 2)`.
 - `X = libro(_G358, _G359)`
- ?- `functor(p, N, A)`.
 - `N = p, A = 0`
- ?- `functor(X, p, 0)`.
 - `X = p`

Acceso a Estructuras: functor/3

■ En el uso (+,+,+) se comporta como un test

- ?- functor(t(X,a),t,2).
 - Yes
- ?- functor(2+3*5-1,'-',2).
 - Yes
- ?- functor(a,a,0).
 - Yes
- ?- functor([x,y],',',2).
 - Yes

Acceso a Estructuras: functor/3

- En el uso (+,-,-) se utiliza para obtener el functor principal de un término
 - ?- functor(punto(a,X),F,N).
 - F = punto
 - N = 2
 - ?- functor([A,f(X),Y],F,N).
 - F = '.'
 - N = 2
 - ?- functor([],F,N)
 - F = []
 - N = 0

Acceso a Estructuras: functor/3

- En el uso `(-,+,+)` se comporta como un generador único: se utiliza para generar una plantilla de estructura
 - ?- functor(T,punto,2).
 - `T = punto(_,_)`
 - ?- functor(T,'+',2).
 - `T = _ + _`
 - ?- functor(T,'+',4).
 - `T = '+'(_,_,_,_)`
 - ?- functor(T,a,0).
 - `T = a`

Acceso a Estructuras: arg/3

- **arg(P, E, C):** la estructura E tiene el componente C en la posición P (contando desde 1)
 - P es un entero positivo, E es un término compuesto → C unifica con el p-ésimo argumento de E
 - Los argumentos se numeran a partir de 1, de izquierda a derecha
 - Permite acceder a los argumentos de una estructura de forma compacta y en tiempo constante

- Ejemplos:
 - ?- _T=date(9,February,1947), arg(3,_T,X).
 - X = 1947
 - ?- arg(2, libro(autor, titulo), X).
 - X = titulo
 - ?- arg(N, libro(autor, titulo), autor).
 - N = 1

Acceso a Estructuras: arg/3

- En el uso (+,+,-) se utiliza para obtener el argumento i-ésimo de una estructura
 - ?- arg(3,arco(i,q2,q0),A).
 - $A = q0$
 - ?- arg(1,[a,b,c,d],A).
 - $A = a$
 - ?- arg(2,[a,b,c,d],A).
 - ?- arg(3,[a,b,c,d],A).

Acceso a Estructuras: arg/3

- En el uso (+,-,+) se utiliza para instanciar el argumento i-ésimo de una estructura
 - ?- arg(2,arco(i,Q,q0),q5).
 - $Q = q5$
 - ?- arg(1,[X,b,c,d],a).
 - $X = a$

Acceso a Estructuras: functor y arg. Ejemplo 1

■ `subTerm(X,Y)`: X es un subtérmino del término Y

❑ `subTerm(X,X).` Cualquier término es subtérmino de si mismo

❑ `subTerm(X,Y):-`
 `compound(Y),`
 `functor(Y,F,N),`
 `subTerm(N,X,Y).`

X es un subtérmino de un término compuesto Y si es subtérmino de uno de los argumentos
subTerm/3 comprueba iterativamente todos los argumentos

❑ `subTerm(N,X,Y):-` Decrementa el contador (número de argumentos) y llama recursivamente a subTerm
 `N>1, N1 is N-1, subTerm(N1,X,Y).`

❑ `subTerm(N,X,Y):-` Caso en el que X es un subtérmino del n-ésimo argumento de Y
 `arg(N,Y,A), subTerm(X,A).`

Acceso a Estructuras: functor y arg. Ejemplo 2

- `add_arrays(X,Y,Z)`: Z es el resultado de sumar las matrices X e Y

- `add_arrays(A1,A2,A3):-`

`functor(A1,array,N),
functor(A2,array,N),
functor(A3,array,N),
add_elements(N,A1,A2,A3).`

Se comprueba que los tres argumentos tienen el mismo nombre de functor y la misma aridad

- `add_elements(0,_A1,_A2,_A3).`

- `add_elements(I,A1,A2,A3):-`

`arg(I,A1,X1), %% I > 0
arg(I,A2,X2),
arg(I,A3,X3),
X3 is X1 + X2,
I1 is I - 1,
add_elements(I1,A1,A2,A3).`

Las matrices se recorren desde el final hasta el principio (para usar un solo índice, deteniéndose a 0), y se suman los elementos correspondientes de las matrices

Acceso a Estructuras: =../2

- $X =.. Y$ (se lee X univ Y)
 - X es cualquier término Prolog
 - Y es una lista cuya cabeza es el átomo del functor principal de X y cuyo resto está formado por los argumentos de X
 - Transforma un término estructurado en una lista:
 - $\text{padre}(\text{juan}, \text{jose}) =.. [\text{padre}, \text{juan}, \text{jose}]$
- Soporta los usos (+,+), (-,+) y (+,-)
- El uso (-,-) genera un error
 - ?- A =.. B.
 - instantiation error

Acceso a Estructuras: =../2

■ En el uso (+,+) se comporta como un test

□ ?- f(a, X, g(b,Y)) =.. [f, a, X , g(b,Y)].

■ Yes

□ ?- [a,b,c] =.. ['.', a, [b,c]].

■ Yes

□ ? 2+3*5-1 =.. ['- ',2+3*5,1].

■ Yes

□ ?- a =.. [a].

■ Yes

Acceso a Estructuras: =../2

- En el uso (+,-) se utiliza para descomponer un término en sus componentes
 - ?- punto(2,3) =.. Xs.
 - $Xs = [\text{punto}, 2, 3]$
 - ?- [A,f(X),Y] =.. Xs.
 - $Xs = ['.', A, [f(X), Y]] ;$
 - ?- sin(X)*cos(X) + 3.14 =.. Xs.
 - $Xs = [+ , \sin(X)*\cos(X), 3.14] ;$
 - ?- 6 =.. Xs.
 - $Xs = [6]$
 - ?- [] =.. Xs.
 - $Xs = [[]]$

Acceso a Estructuras: =../2

- En el uso (-,+), se comporta como un generador único. Se utiliza para componer un término a partir de sus componentes
 - ?- T =.. ['+',a+b+c,d].
 - T = a+b+c+d
 - ?- T =.. [arco,ej1,i,q3,q5].
 - T = arco(ej1, i, q3, q5)
 - ?- T =.. ['.', p,[a,k]].
 - T = [p, a, k]
 - ?- T =.. [fin].
 - T = fin

Acceso a Estructuras: =../2. Ejercicio

- Suponemos las siguientes figuras geométricas, donde los argumentos de las distintas figuras son números que indican sus dimensiones
 - `cuadrado(lado); rectangulo(anchura, altura); triangulo(lado1, lado2, lado3); circulo(radio)`
- Definir un predicado `escala/3` (`escala(+F,+K,?KF)`) que multiplique cada dimensión de la figura F por el factor K, obteniendo la figura escalada KF
- Ejemplo:
 - `?- escala(rectangulo(3,5), 2, R).`
 - `R = rectangulo(6,10)`
 - `?- escala(circulo(6.5),0.5,C).`
 - `C = circulo(3.25)`

Conversión entre Strings y Átomos

■ name(A,S)

- ❑ A es el átomo/número cuyo nombre es la lista de caracteres ASCII S
- ❑ ?- name(hello,S).
 - S = [104,101,108,108,111]
- ❑ ?- name(A,[104,101,108,108,111]).
 - A = hello
- ❑ ?- name(A,"hello").
 - A = hello

Predicados Meta-Lógicos (I)

- Se utilizan para examinar el estado de instanciación actual de un término:
 - `var(Term)`: es cierto si Term es una variable libre (no instanciada)
 - `?- var(X), X = f(a). % éxito`
 - `?- X = f(a), var(X). % fallo`
 - `nonvar(Term)`: es cierto si Term no es una variable libre
 - `?- X = f(Y), nonvar(X). % éxito`
 - `ground(Term)`: es cierto si Term no contiene variables libres (es un término básico)
 - `?- X = f(Y), ground(X). % fallo`

Predicados Meta-Lógicos (II)

- `is_list(Term)`: es cierto si Term está instanciado a una lista
 - Es decir, a la lista vacía [] ó a un término con funtor '.' y aridad 2, donde el segundo argumento es una lista
 - `?- is_list(a).` % fallo
 - `?- is_list(X).` % fallo
 - `?- is_list([a,b,c]).` % éxito

Predicados Meta-Lógicos: Ejemplo

- `length(Xs,N):-`
 `var(Xs), integer(N), length_num(N,Xs).` *% Xs es una variable libre*
 % modo out-in
- `length(Xs,N):-`
 `nonvar(Xs), length_list(Xs,N).` *Xs no es una variable libre*
 % modo in-out
- `length_num(0,[]).`
- `length_num(N,[_|Xs]):-`
 `N > 0, N1 is N - 1, length_num(N1,Xs).`
- `length_list([],0).`
- `length_list([X|Xs],N):-`
 `length_list(Xs,N1), N is N1 + 1.`

Note: esta definición no es necesaria; en realidad el predicado length ya es reversible (aunque menos eficiente que length_num(N,L), cuando L es una variable)

Comparación de Términos

- La igualdad de términos puede determinarse de diferentes formas
 - El operador “=” es la propia unificación. Esto es, se unifican las variables de los términos que se comparan
 - El operador “==” no unifica las variables de los términos que se comparan. Por tanto, una variable (no ligada) sólo será igual a sí misma
 - $T1 = T2$
 - Es cierto si $T1$ y $T2$ pueden unificarse
 - $T1 \neq T2$
 - Es cierto si $T1$ y $T2$ no pueden unificarse
 - $T1 == T2$
 - Es cierto si $T1$ y $T2$ son idénticos
 - $T1 \neq T2$
 - Es cierto si $T1$ y $T2$ no son idénticos

Comparación de Términos: Ejemplos

■ ?- a == a.

□ si

■ ?- a == X.

□ no

■ ?- X == Y.

□ no

■ ?- X == X.

□ X = _G2

■ f(X) == f(X).

□ si

■ ?- f(X) == f(Y).

□ no

Ordenación de Términos No Básicos

■ Orden alfabético/lexicográfico:

- $X @> Y, X @>= Y, X @< Y, X @<= Y$
- P. ej: $T1 @< T2$ se verifica si el término $T1$ es anterior que $T2$ en el orden de términos de Prolog
- Ejemplos:
 - $?- f(a) @> f(b).$ % fallo
 - $?- f(b) @> f(a).$ % éxito
 - $?- f(X) @> f(Y).$ % dependiente de la implementación
 - $?- X @< 3. \Rightarrow \text{Yes}$
 - $?- ab @< ac. \Rightarrow \text{Yes}$
 - $?- 21 @< 123. \Rightarrow \text{Yes}$
 - $?- 12 @< a. \Rightarrow \text{Yes}$
 - $?- g @< f(b). \Rightarrow \text{Yes}$
 - $?- f(b) @< f(a,b). \Rightarrow \text{Yes}$
 - $?- [a,1] @< [a,3]. \Rightarrow \text{Yes}$
 - $?- [a] @< [a,3]. \Rightarrow \text{Yes}$

Comparación de Términos No Básicos: Ejemplos

■ `subterm/2` con términos no básicos

- ❑ `subterm(Sub,Term):- Sub == Term.` *% Sub y Term son idénticos*
- ❑ `subterm(Sub,Term):-`
 `nonvar(Term),`
 `functor(Term,F,N),`
 `subterm(N,Sub,Term).` *% subterm/3 no varía con respecto a la definición vista anteriormente*

■ `insert/3` inserta un elemento en una lista ordenada

- ❑ `insert([], Item, [Item]).`
- ❑ `insert([H|T], Item, [H|T]):- H == Item.`
- ❑ `insert([H|T], Item, [Item, H|T]):- H @> Item.`
- ❑ `insert([H|T], Item, [H|NewT]) :- H @< Item, insert(T, Item, NewT).`

Entrada/Salida de Términos

■ Predicado `read(X)`:

- ❑ Lee por teclado un término, que se instanciará en la variable X
- ❑ El término debe ir seguido de “.” y un carácter no imprimible (espacio o intro)
- ❑ Se pueden introducir términos en minúsculas, o cadenas

■ Predicado `write(X)`:

- ❑ Siempre se satisface; nunca se intenta resatisfacer
- ❑ Si X está instanciada, se muestra en pantalla
- ❑ Si no, se muestra la variable interna (e.g., “_G244”)

■ Predicado `nl`:

- ❑ Provoca un salto de línea

■ Predicado `tab(X)`:

- ❑ Escribe X espacios en blanco

■ Predicado `display(X)`:

- ❑ Muestra X sin interpretar los funtores/operadores

Escritura de Términos: Ejemplo

- Escritura de una lista en una columna: [escribir_columna/1](#)
 - `escribir_columna([]).`
 - `escribir_columna([Cabeza|Cola]):-`
 `write(Cabeza),`
 `nl,`
 `escribir_columna(Cola).`

Escritura/Lectura de Términos: Ejemplo

■ Calcular y escribir el cubo de un número dado: [cubo/0](#)

□ cubo :-

```
write('Siguiente item: '),  
read(X),  
procesa(X).
```

□ procesa(stop) :- !.

□ procesa(N) :-

```
C is N*N*N,  
tab(3),  
write('El cubo de '),write(N),  
write(' is '), write(C), nl,  
cubo.
```

Entrada/Salida de Caracteres

■ Predicado `get_code(X)`:

- Si X no está instanciada, captura el primer carácter imprimible y lo instancia en X
- Si X está instanciada, intenta hacer equiparación con la entrada
 - `?- get_code(X).`
|: d
X= 100
 - `?- X=3, get_code(X).`
|: a
no

■ Predicado `put_code(X)`:

- Si X está instanciada a un código ASCII (entero positivo), entonces escribe el correspondiente caracter
 - `?- put_code (104).`
h

Entrada/Salida de Ficheros (I)

- Predicado `see(X)`:
 - Establece como canal de entrada el fichero X
 - Si X no esta instanciada, se produce un error
- Predicado `seeing(X)`:
 - Averigua el canal de entrada activo
- Predicado `seen`:
 - Cierra el fichero y restablece el teclado (*user*) como canal de entrada
- Nota: los ficheros se representan como átomos de Prolog, escribiéndolos entre comillas simples.
 - `'/home/usuario/fichero.txt'`
 - `'salida.txt'`

Entrada/Salida de Ficheros (II)

- Predicado **tell(X)**:
 - Establece como canal de salida el fichero X
 - Si X no esta instanciada, se produce un error
- Predicado **telling(X)**:
 - Averigua el canal de salida activo
- Predicado **told**:
 - Cierra el fichero y restablece el teclado como canal de salida

[input-output.pl](#) (write_list_to_file)

Entrada/Salida: Ejercicio

- Escribir un predicado ([barras/1](#)) que tenga el siguiente comportamiento:

- ?- barras([1,2,5,3,4]).

*

**

yes

[barras.pl](#)

Predicados de Orden Superior

- Su semántica viene dada en términos de un lenguaje de orden inferior
 - Los predicados de 2º orden (meta-lenguaje) “hablan sobre” símbolos de un lenguaje de 1º orden (lenguaje-objeto)
- Usos de predicados de orden superior:
 - Comprobación de tipos: `integer/1`, `atom/1`, `var/1`, `ground/1`
 - Construcción de fórmulas: `=../2`
 - Ejecución y control de la ejecución de objetivos: `call/1`
 - Recopilación de múltiples soluciones: `findall/3`, `setof/3`
- **No** son predicados de orden superior:
 - Aquellos que permiten añadir o eliminar cláusulas del conjunto soporte (programa) en tiempo de ejecución (programación dinámica): `assert/1`, `retract/1`, `instance/2`, etc.
 - El corte `!/0` que es un mecanismo de control del *backtracking* cuya sintaxis y ejecución es la de un predicado sin serlo realmente

Meta-Programación

- Los argumentos de los predicados “ordinarios” cumplen básicamente dos funciones:
 - Contienen datos sobre los que razonar
 - E.g. `member(2,[1,2,3])`
 - Contienen variables a instanciar
 - E.g. `plus(2,3,X)`
- Un **meta-predicado** tiene entre sus argumentos otros (meta) predicados cuya prueba es parte de la prueba del meta-predicado
 - `opcional(Goal):- call(Goal).`
 - `opcional(_Goal).`
 - `call/1` (predefinido) tiene éxito cuando su argumento lo tiene
 - Ej. `?- member(c,[a,b]).` *versus* `?- opcional(member(c,[a,b])).`

Meta-Programación: call/1

- El meta-predicado `call(X)` convierte el término `X` en un objetivo y llama a dicho objetivo
 - `X` debe estar instanciado a un término, sino se produce un error
- `call(X)` se cumple si se satisface `X` como objetivo
- Se usa habitualmente para
 - Meta-programación (intérpretes, *shells*)
 - Definir negación
 - Implementar orden superior
- Ejemplo:
 - `q(a).`
 - `p(X) :- call(X).`
 - `?- p(q(Y)).`
`Y = a`

Meta-Programación: call/1. Ejemplos

■ Ejemplo:

❑ `mipred(X) :- display(X), nl.`

Nota: este ejemplo carece de utilidad puesto que se puede ejecutar la llamada a `mipred/1` directamente

% Un predicado

❑ `ejemplo:- X = mipred(5), call(X).`

% Llamada de orden superior

■ Ejemplo: `call/1` resulta muy útil en combinación con otros predicados como `univ/2`

❑ `sujeto(12).`

❑ `sujeto(13).`

❑ `sujeto(78).`

❑ `aplicar(Predicado):-`

`sujeto(X), LLamada =.. [Predicado,X], call(LLamada), nl, fail.`

❑ `aplicar(_).`

Predicado fail/0

- El predicado `fail/0` es un predicado predefinido que siempre falla
 - Siempre produce fallo
 - Falla cuando se ejecuta (similar al objetivo $a=b$)
 - Es un objetivo que nunca se satisface
- Se utiliza para detectar prematuramente combinaciones de los argumentos que no llevan a solución
 - Evitando la ejecución de código que va a fallar
- Es útil cuando queremos detectar casos explícitos que invalidan un predicado

Corte y fallo

- Para evitar la aplicación de una regla, se puede forzar el fallo con una combinación de *cut* y *fail*
- Ejemplo: Comprobación de diferencia
 - `different(X,X) :- !, fail.`
 - `different(X,Y).`
- La combinación de *corte* y *fallo* (*cut-fail*) permite forzar el fracaso de un predicado
 - Especificando una respuesta negativa
 - Útil pero hay que usarlo con cuidado

Corte y fallo: Ejemplo

- Predicado `ground/1` para verificar que un término no contiene variables libres (es un término básico)
 - Falla tan pronto se encuentra una variable libre
 - `ground(Term):- var(Term), !, fail.`
 - `ground(Term):-`
 `nonvar(Term), functor(Term,F,N), ground(N,Term).`
 - `ground(0,T). % se han recorrido todos los subtérminos`
 - `ground(N,T):-`
 `N>0, arg(N,T,Arg), ground(Arg), N1 is N-1, ground(N1,T).`

Meta-Programación: negación como fallo (I)

- La negación en Prolog consiste el predicado predefinido de segundo orden `\+ /1`
 - Recibe como argumento un objetivo
 - Si dicho objetivo tiene éxito la negación falla y viceversa
 - Ejemplo: `\+ (X > 5)` es equivalente a $X \leq 5$
- `not/1` usa el `corte` y el predicado `fail`
 - `not(Goal) :- call(Goal), !, fail.`
 - `not(Goal).`
- La terminación de `not(Goal)` depende de la terminación de `Goal`
 - `not(Goal)` termina si se encuentra éxito para `Goal` antes de una rama infinita
 - `not(Goal)` tiene éxito cuando `Goal` no puede ser probado

Meta-Programación: negación como fallo (II)

- Funciona de manera adecuada para objetivos básicos (no contienen variables libres)
 - Es responsabilidad del programador asegurar esta condición
 - Nunca instancia variables
- Es útil pero hay que saber utilizarlo:
 - `unmarried_student(X):- not(married(X)), student(X).`
 - `student(joe).` `?- unmarried_student(joe).`
 - `married(john).` `?- unmarried_student(X).`
- Prolog asume que aquellos objetivos que no tienen solución (fallan) son falsos
 - Cualquier cosa que no puede probarse con las reglas y los hechos de la base de conocimiento se considera falsa

Meta-Programación: negación como fallo. Ejemplo

■ Definición de conjuntos disjuntos:

- ❑ `overlap(S1,S2):-` % S1 y S2 se solapan si comparten algún elemento
 `member(X,S1),member(X,S2).`
- ❑ `disjoint(S1,S2):-`
 `\+overlap(S1,S2).`

- ❑ `?- disjoint([a,b,c],[2,c,4]).`
 - `no`
- ❑ `?- disjoint([a,b],[1,2,3,4]).`
 - `yes`
- ❑ `?- disjoint([a,c],X).`
 - `no`

Meta-Programación: negación como fallo. Ejemplo

- `aprobado(X) :- not(suspenso(X)), matriculado(X).`
- `matriculado(juan).`
- `matriculado(luis).`
- `suspenso(juan).`

■ Consultas

- `?- aprobado(luis).`
 - Yes
- `?- aprobado(X).`
 - No

Meta-Programación: negación como fallo. Ejemplo

- `aprobado2(X) :- matriculado2(X), not(suspenso2(X)).`
- `matriculado2(juan).`
- `matriculado2(luis).`
- `suspenso2(juan).`

¿Qué pasa si cambiamos el orden de los predicados?
?- aprobado2 (X).

Ejercicio

- Definir el predicado **borra/3** (`borra(L1,X,L2)`) que se verifica si L2 es la lista obtenida eliminando los elementos de L1 unificables simultáneamente con X
 - Solucion 1: definición con not
 - Solucion 2: definición con corte
 - ?- `borra([a,b,a,c],a,L)`.
 - `L = [b, c] ;`
 - No
 - ?- `borra([a,Y,a,c],a,L)`.
 - `Y = a`
 - `L = [c] ;`
 - No
 - ?- `borra([a,Y,a,c],X,L)`.
 - `Y = a`
 - `X = a`
 - `L = [c] ;`
 - No

Ejercicio

■ Definición con not

- `borra_1([],_,[]).`
- `borra_1([X|L1],Y,L2) :-`
 `X=Y,`
 `borra_1(L1,Y,L2).`
- `borra_1([X|L1],Y,[X|L2]) :-`
 `not(X=Y),`
 `borra_1(L1,Y,L2).`

Ejercicio

■ Definición con corte

- ❑ `borra_2([],_,[]).`
- ❑ `borra_2([X|L1],Y,L2) :-`
 `X=Y, !,`
 `borra_2(L1,Y,L2).`
- ❑ `borra_2([X|L1],Y,[X|L2]) :-`
 `% not(X=Y),`
 `borra_2(L1,Y,L2).`

Meta-predicados de control

- Los meta-predicados de control se encargan de imponer control sobre sus argumentos:

```
just_once(Goal):- call(Goal), !.
```

- En lugar de que ese control deba imponerse en la definición de cada predicado a controlar:

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

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

```
?- member_check(2,[1,2,3,2,4]).      ?- just_once(member(2,[1,2,3,2,4])).
```

- Ventajas de los meta-predicados de control:
 - ❑ Control explícito y su definición centralizada en el meta-predicado
 - ❑ Los predicados a controlar mantienen múltiples usos y mayor claridad

Meta-predicados de control más frecuentes

❑ Negación como fallo:

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

❑ Prueba determinista:

```
just_once(Goal):- call(Goal), !.
```

❑ Prueba condicional:

```
ifthenelse(If,Then,_):-  
    call(If),  
    call(Then).
```

```
ifthenelse(If,_Then,Else):-  
    \+ If,  
    call(Else).
```

```
If -> Then ; _Else:-  
    call(If),  
    !,  
    call(Then).
```

```
If -> _Then ; Else:-  
    call(Else).
```

❑ Bucle:

```
free_while(Cond,Do):-  
    call(Cond),  
    call(Do),  
    fail.
```

```
free_while(_,_).
```

```
while(Cond,Do):-  
    call(Cond),  
    ( Do -> fail ; (!, fail) ).
```

```
while(_,_).
```

Meta-Programación: findall/3

- El meta-predicado `findall/3` (`findall(Term, Goal, ListResults)`) se verifica si `ListResults` es el conjunto de todas las instancias del término `Term` que verifican el objetivo `Goal`
 - `ListResults` es `[]` si no hay instancias de `Term`
 - El número de soluciones debería ser finita (y enumerable en un tiempo finito)
- Ejemplos:
 - `?- findall(X,(member(X,[d,4,a,3,d,4,2,3]),number(X)),L).`
 - `L = [4, 3, 4, 2, 3]`
 - `?- findall(X,(member(X,[d,4,a,3,d,4,2,3]),compound(X)),L).`
 - `L = []`

Meta-Programación: setof/3

- El meta-predicado **setof/3** (setof(Term, Goal, ListResults)) se verifica si ListResults es la lista ordenada sin repeticiones de las instancias del término Term que verifican el objetivo Goal
 - El predicado falla si no hay instancias de Term
 - El conjunto debe ser finito (y enumerable en tiempo finito)
- Ejemplos:
 - ?- setof(X,(member(X,[d,4,a,3,d,4,2,3]),number(X)),L).
 - L = [2, 3, 4]
 - ?- setof(X,member(X,[d,4,a,3,d,4,2,3]),L).
 - L = [2, 3, 4, a, d]
 - ?- setof(X,(member(X,[d,4,a,3,d,4,2,3]),compound(X)),L).
 - No

Meta-Programación: bagof/3

- **bagof/3** es igual que **setof/3**, pero devuelve una lista no ordenada y con duplicados (según el orden del *backtracking*)
 - El predicado falla si no hay instancias de Term
 - El conjunto debe ser finito (y enumerable en tiempo finito)
- Ejemplos:
 - ?- bagof(X,(member(X,[d,4,a,3,d,4,2,3]),number(X)),L).
 - L = [4,3,4,2,3]
 - ?- bagof(X,member(X,[d,4,a,3,d,4,2,3]),L).
 - L = [d,4,a,3,d,4,2,3]
 - ?- bagof(X,(member(X,[d,4,a,3,d,4,2,3]),compound(X)),L).
 - No

Ejercicio 1

- Se denomina **factor** de un número natural N , a otro número también natural que es divisor de N , pero diferente de N
 - los factores de 28 son 1, 2, 4, 7 y 14
- Definir el predicado **factores/2** (`factores(+N,-L)`) que se verifique si L es la lista ordenada de los factores del número N
 - `?- factores(42,L).`
 - `L= [1,2,3,6,7,14,21]`

Ejercicio 2

- Los números naturales se pueden clasificar en tres tipos:
 - N es de tipo a si N es mayor que la suma de sus factores
 - N es de tipo b si N es igual que la suma de sus factores
 - N es de tipo c si N es menor que la suma de sus factores
- Definir el predicado **tipoNatural/2** (`tipoNatural(+N,-T)`) que se verifique si T es el tipo del número N
 - `tipoNatural(10,T).`
 - `T=a`
 - `tipoNatural(28,T).`
 - `T=b`
 - `tipoNatural(12,T).`
 - `T=c`

Ejercicio 3

- Definir el predicado `soloConsonantes/2` (`soloConsonantes(+P,-Q)`) que se verifica si Q es la palabra que se obtiene al eliminar todas las vocales de la palabra P
 - ?- `soloConsonantes(segoviano,P)`.
 - P=sgvn

Ejercicio 4

- Definir el predicado `traduceDigitos/2` (`traduceDigitos(+L1,-L2)`) que se verifica si L2 es la lista de palabras correspondientes a los dígitos de la lista L1
 - ?- `traduceDigitos([1,2],L)`.
 - `L = [uno,dos]`
 - Solución 1: usando recursividad
 - Solución 2: usando metapredicados
- Nota: usar el predicado auxiliar `nombreDigito/2` (`nombreDigito(D,N)`) que se verifica si N es el nombre del dígito D

Modificación Dinámica (I)

- La base de conocimientos en Prolog se puede modificar en tiempo de ejecución (mientras se ejecuta el programa)
 - Muy potente
- Esto permite
 - Añadir conocimiento adquirido durante la ejecución
 - Suprimir reglas que se hacen innecesarias durante la ejecución
 - Simular algunas técnicas de programación imperativa no disponibles directamente en Prolog
 - Algunos trucos de programación
- A veces, esto es muy útil, pero a menudo un error
 - Código difícil de leer, difícil de entender, difícil de depurar
 - Típicamente, lento

Modificación Dinámica (II)

- La modificación dinámica debe utilizarse esporádicamente, con cuidado y a nivel local
 - Para ello existen predicados que añaden o eliminan cláusulas de la base de conocimientos
- La afirmación y la retracción pueden justificarse lógicamente en algunos casos:
 - Aserción de cláusulas que lógicamente se derivan del programa (lemas)
 - Retracción de las cláusulas que son lógicamente redundante
- El comportamiento y/o los requisitos pueden diferir entre las implementaciones de Prolog
 - Por lo general, el predicado debe declararse dinámico
 - `:- dynamic predicado/n.`

Modificación Dinámica: Añadir Conocimiento (I)

- **assert/1 (assert(Clausula))**: añade la cláusula a la base de conocimientos al final de todas las cláusulas del predicado
 - Clausula debe estar instanciada a una cláusula Prolog
- Ejemplo:
 - padre(pepe, juan). Hecho en la base de conocimientos.
 - ?- padre(pepe, X).
X = juan
yes
 - ?- assert(padre(pepe, javi)), padre(pepe, X).
X = juan (;)
X = javi
yes

Modificación Dinámica: Añadir Conocimiento (II)

- `assert/1 (assert(Clausula))`: (continuación)
 - Si se introduce una regla, hay que encerrarla entre paréntesis para que los distintos operadores que posea no se confundan
 - Ejemplo:
 - `assert((hijo(X, Y):- padre(Y, X)))`.
- `asserta(Clausula)`: como `assert`, pero coloca la cláusula en primer lugar
- No admiten *backtracking*

Modificación Dinámica: Eliminar Conocimiento (I)

- **retract(Clausula)**: elimina de la base de conocimientos la primera cláusula unificable con Clausula, que no debe ser una variable
 - ❑ Por *backtracking* puede eliminar otras cláusulas
 - ❑ Si no hay cláusulas, falla

- Ejemplo:

- ❑ padre(pepe, juan).
- ❑ padre(pepe, javi).
- ❑ ?- retract(padre(pepe, X)), padre(pepe, X).

X = javi

yes

Modificación Dinámica: Eliminar Conocimiento (II)

- **abolish(Predicado/Aridad)**: elimina todas las cláusulas del predicado Predicado con aridad Aridad
 - ❑ Predicado debe estar instanciado
- Ejemplo:
 - ❑ padre(pepe, juan).
 - ❑ padre(pepe, javi).
 - ❑ ?- abolish(padre/2).
yes
 - ❑ ?- padre(X,Y).
existence error

Modificación Dinámica: Ejemplo 1

- `relate_numbers(X, Y):- assert(related(X, Y)).`
- `unrelate_numbers(X, Y):- retract(related(X, Y)).`
- `?- related(1, 2).`
 - ☐ no
- `?- relate_numbers(1, 2).`
 - ☐ yes
- `?- related(1, 2).`
 - ☐ yes
- `?- unrelate_numbers(1, 2).`
 - ☐ yes
- `?- related(1, 2).`
 - ☐ No

Modificación Dinámica: Ejemplo 2

■ Números de Fibonacci

```
fibonacci(0,0).  
fibonacci(1,1).  
fibonacci(N,X):-  
    N>1,  
    N1 is N-1,  
    fibonacci(N1,X1),  
    N2 is N-2,  
    fibonacci(N2,X2),  
  
    X is X1+X2.
```

```
lfib(N,F):- lemma_fib(N,F),!.  
lfib(N,F):-  
    N>1,  
    N1 is N-1,  
    N2 is N1-1,  
    lfib(N1,F1),  
    lfib(N2,F2),  
    F is F1+F2,  
    assert(lemma_fib(N,F)).  
  
:-dynamic lemma_fib/2.  
  
lemma_fib(0,0).  
lemma_fib(1,1).
```

Modificación Dinámica: Ejercicio

- Programa que permita a un usuario preguntar (vía teclado) por la **capital de un determinado país**
 - Si el país está en la base de conocimientos, entonces se devuelve el nombre de su capital
 - Si el país no está en la base de conocimientos, entonces se solicita el nombre de la capital y se introduce este hecho en la base de conocimientos
 - Si el usuario teclea "stop.", entonces se graba la nueva base de conocimientos y se sale del programa

Modificación Dinámica: clause/2 (I)

■ *clause(Head, Body):*

- ❑ Busca una cláusula cuya cabeza es *Head* y cuyo cuerpo es *Body*
- ❑ La cláusula *Head:-Body* existe en el programa actual
 - *Head* es un término que no es una variable libre
- ❑ El predicado correspondiente debe ser dinámico

■ Ejemplo:

- ❑ $p(X) :- q(X).$
- ❑ $p(b) :- r(b), s(b).$
- ❑ $q(a).$
- ❑ $?- \text{clause}(p(a), \text{Cuerpo}).$
 - $\text{Cuerpo} = q(a)$
- ❑ $?- \text{clause}(p(b), \text{Cuerpo}).$
 - $\text{Cuerpo} = q(b);$
 - $\text{Cuerpo} = r(b), s(b)$
- ❑ $?- \text{clause}(q(a), \text{Cuerpo}).$
 - $\text{Cuerpo} = \text{true}$

Modificación Dinámica: clause/2 (II)

■ Ejemplo: Meta-intérprete simple (“vanilla”)

□ `solve(true).` La meta vacía es cierta.
La meta vacía está resuelta.

□ `solve((A,B)) :- solve(A), solve(B).` La meta conjuntiva (A, B) es cierta si A es cierta y B es cierta.
Para resolver la meta (A, B) resolver primero A y después B.

□ `solve(A) :- clause(A,B), solve(B).` La meta A es cierta si existe una cláusula A:-B y B es cierta.
Para resolver la meta A, seleccionar una cláusula cuya cabeza unifique con A y resolver el cuerpo.

■ Este código se puede mejorar para realizar diferentes tareas: *tracing*, *debugging*, proporcionar explicaciones (sistemas expertos), etc.

EjemploClause.pl

Modificación Dinámica: clause/2 (III)

- Ejemplo: Definir un metaintérprete que cuente la cantidad de hechos visitados a lo largo de la resolución de una cierta consulta
 - `solve(true, 1).`
 - `solve((A, B), CHAB) :-`
 `solve(A, CHA), solve(B, CHB),`
 `CHAB is CHA + CHB.`
 - `solve(A, CH) :-`
 `clause(A, B), solve(B, CH).`

Análisis Sintáctico (*Parsing*) en Prolog

- Supongamos que necesitamos definir un predicado que sea capaz de aceptar frases sencillas como:
 - Tu hermano es el hijo de tus padres
 - Tu abuelo es el padre de tus padres
 - Mi primo es el hijo de mis tios

Análisis Sintáctico en Prolog

■ Estas frases se ajustan a la siguiente gramática BNF:

- ❑ `<frase> ::= <sn> <sv>`
- ❑ `<sn> ::= <posesivo> <nombre> | <determinante> <nombre>`
- ❑ `<sv> ::= <verbo> <atributo>`
- ❑ `<atributo> ::= <sn> <cn>`
- ❑ `<posesivo> ::= tu | mi | mis | tus`
- ❑ `<nombre> ::= hermano | abuelo | primo | padres | tios | hijo |
padre`
- ❑ `<verbo> ::= es`
- ❑ `<determinante> ::= el`
- ❑ `<cn> ::= <prep> <sn>`
- ❑ `<prep> ::= de`

Análisis Sintáctico en Prolog

- Escribir un predicado `frase/1` que
 - acepte una frase si se adecúa a las reglas de la gramática, o
 - la rechace en caso contrario
- La frase se representa como una lista de palabras
 - `?- frase([mi,primo,es,el,hijo,de,mis,tios]).`
- Una manera de implementar estas reglas gramaticales es emplear la estrategia de “`generar y testear`”
 - `frase(L):-`
 - `append(SN,SV,L),` Genera posibles valores para SN y SV, dividiendo la lista original L
 - `sn(SN),` Comprueban si cada sublista es gramaticalmente correcta. Si no, por *backtracking* se
 - `sv(SV).` genera otra posible división

[Ejemplo Analisis Sintactico-Listas.pl](#)

Nota: Los predicados `sn/1` y `sv/1` son similares a `frase/1`, y llaman a otros predicados que tratan con unidades más pequeñas de una sentencia

Parsing (usando *append* y listas): Ejemplo

- La frase, en inglés, se representa como una lista de caracteres

- myphrase(X) :-

- append(A,T1,X), article(A), append(SP,T2,T1), spaces(SP),
 - append(N,T3,T2), noun(N), append(SPN,V,T3), spaces(SPN),
 - verb(V).

- article([a]).

- article([t,h,e]).

- spaces([' ']).

- spaces([' ' | Y]) :- spaces(Y).

- noun([c,a,r]).

- noun([p,l,a,n,e]).

- verb([f,l,i,e,s]).

- verb([d,r,i,v,e,s]).

- ?- myphrase([t,h,e,' ',p,l,a,n,e,' ',f,l,i,e,s]).

Análisis Sintáctico en Prolog

- La estrategia “generar y testear” es ineficaz
- Una estrategia más eficiente consiste en
 - evitar la etapa de generación
 - pasar la lista completa a los predicados que implementan las reglas gramaticales
 - Estos predicados identifican los correspondientes elementos gramaticales procesando secuencialmente los elementos de la lista de izquierda a derecha, devolviendo el resto de la lista
- Para ello podemos emplear **listas diferencia**

Listas Diferencia (*Difference Lists*)

- Son estructuras de datos incompletas
- Ejemplo: Supongamos la lista [1, 2, 3]. Podríamos representar dicha lista como la diferencia de los siguientes pares de listas:
 - [1, 2, 3, 5, 8] y [5, 8]
 - [1, 2, 3, 6, 7, 8, 9] y [6, 7, 8, 9]
 - [1,2,3] y [].
 - Cada uno de estos son casos del par de dos listas incompletas [1,2,3 | X] y X.
 - El par se llama **lista diferencia**
- Una **lista diferencia** se representa mediante A-B
 - A es una lista abierta que acaba en B
 - B es una variable libre

Listas Diferencia (*Difference Lists*)

- Ejemplo: La lista [1,2,3] se representa usando listas diferencia como
 - $[1, 2, 3 \mid X] - X$
 - $([1, 2, 3 \mid X], X)$ (lista abierta, referencia al resto de la lista)
- Permiten mantener un puntero al final de la lista
- Permiten concatenación en tiempo constante
 - `append_dl (X-Y, Y-Z, X-Z).`
- Permiten manipular listas de forma más eficiente definiendo “patrones de listas”

Listas Diferencia (*Difference Lists*)

- Predicado para transformar una lista diferencia en una lista “normal”
 - `dl_to_list([] - _, []) :- !.`
 - `dl_to_list([X|Y] - Z, [X|W]) :- dl_to_list(Y - Z, W).`
- Predicado para transformar una lista “normal” en una lista diferencia
 - `list_to_dl([], X - X).`
 - `list_to_dl([X|W], [X|Y] - Z) :- list_to_dl(W, Y - Z).`

Gramática Definida por Cláusulas

- Prolog incorpora la posibilidad de definir gramáticas mediante una sintaxis especial que oculta la presencia de las listas diferencia
- Esta sintaxis se denomina **Gramática Definida por Cláusulas** (*Definite Clause Grammar* - DCG)
 - Extensión sintáctica de la sintaxis ordinaria de Prolog
 - Se utiliza para la creación de gramáticas formales en forma abreviada
 - Simplifica y hace más legibles los analizadores sintácticos
- Notación:
 - `no_terminal --> cuerpo`
 - no terminales: átomos de Prolog
 - cuerpo: terminales y no terminales separados por “,”
 - cadenas de terminales: listas de átomos de Prolog

Gramática Definida por Cláusulas

■ Ejemplos:

- ❑ $S \rightarrow [a], [b], S.$
- ❑ $S \rightarrow [c].$
- ❑ $Se \rightarrow sn, sv.$
- ❑ $sn \rightarrow det, nom.$
- ❑ $sv \rightarrow verbo, sn.$
- ❑ $det \rightarrow [el].$
- ❑ $nom \rightarrow [Pepe]$
- ❑ $nom \rightarrow [gato].$
- ❑ $verb \rightarrow [come].$

■ Ejemplo:

- ❑ $oracion(S0, S):- \text{ sintagma_nominal}(S0, S1), \text{ sintagma_verbal}(S1, S).$
- ❑ $oracion \rightarrow \text{ sintagma_nominal}, \text{ sintagma_verbal}. \quad \% \text{ Sintaxis DCG}$

Gramática Definida por Cláusulas

- Las cláusulas gramaticales definidas con DCG se analizan y traducen a cláusulas Prolog que usan listas diferencias
- Ejemplo:
 - `sentence --> nounphrase, verbphrase.` % usando DCG
 - `sentence (S1, S2) :- nounphrase (S1, S3) , verbphrase (S3, S2).` % traducido
 - Nota: Para analizar una frase, hemos de invocar `sentence/2`
 - `?- sentence ([dog, chases, cat],R).`
- El vocabulario (los símbolos terminales) en DCG se representa con listas simples
 - `noun --> [dog].`
 - `verb --> [chases].`
- Estas listas se traducen a listas diferencia en Prolog
 - `noun([dog|X], X).`
 - `verb([chases|X], X).`

Gramática Definida por Cláusulas

- La gramática definida presenta algunas deficiencias como la falta de concordancia entre el número del posesivo y el nombre
- Por ejemplo, “mi padres” resultaría aceptable como sintagma nominal (sn):
 - ?- sn([mi,padres], R).
 - R = []
 - Yes
- Por ello una frase como la siguiente sería aceptable:
 - ?- frase([mi,abuelo,es,el,padres,de,mis,padre],R).
 - R = []
 - Yes

Gramática Definida por Cláusulas: Uso de Variables

- Para resolver este problema se pueden emplear argumentos en las cláusulas de la gramática
 - `posesivo(sing) --> [mi].`
 - `posesivo(plur) --> [mis].`
 - `nombre(plur) --> [padres].`
 - `nombre(sing) --> [padre].`
- Con esta solución la siguiente frase no es válida
 - `?- frase([mi,primo,es,el,hijo,de,mi,tios],R).`
 - No

Gramática Definida por Cláusulas: Uso de Variables

- Ejemplo: Uso de variables para devolver un análisis sintáctico (y eventualmente morfológico) de la frase

[EjemploAnálisisSintactico-DCGs-Analysis.pl](#)

Gramática Definida por Cláusulas: Acciones

- Es posible incluir cláusulas de Prolog en la definición de las cláusulas gramaticales.
- Estas cláusulas deben encerrarse entre llaves { }
- Ejemplo:
 - ❑ %% ?- myphrase(NChars,"the plane flies",[]).
 - ❑ %% ?- phrase(myphrase(N),"the plane flies").
 - ❑ :- use_package(dcg).
 - ❑ myphrase(N) --> article(AC), spaces(S1), noun(NC), spaces(S2),
verb(VC), { N is AC + S1 + NC + S2 + VC }.
 - ❑ article(3) --> "the". spaces(1) --> " ".
 - ❑ article(1) --> "a". spaces(N) --> " ", spaces(N1), {N is N1+1}.
 - ❑ noun(5) --> "plane". verb(5) --> "flies".
 - ❑ noun(3) --> "car". verb(6) --> "drives".

DCGs: Ejemplo

■ Gramática para reconocer expresiones aritméticas.

- `expr --> term.`
- `expr --> term, [+], expr.`
- `expr --> term, [-], expr.`
- `term --> num.`
- `term --> num, [*], term.`
- `term --> num, [/], term.`
- `num --> [D], { number(D) }.`
- `parse(E) :- expr(E,[]).`

DCGs: Ejercicio

- Escribir una gramática DCG para analizar contactos telefónicos
 - ❑ `texto_contacto('Garcia, Manolo 992168010')`.
 - ❑ `texto_contacto('Pedro 634873429')`.
 - ❑ `texto_contacto('Perea, Jose Manuel 634987450')`.
 - ❑ `texto_contacto('Lopez Carmona, Carmen 594270328')`.
 - ❑ `texto_contacto('Castro Moreno, Luis Miguel 340769600')`.
 - ❑ `texto_contacto('Martinez de la Rosa, Pedro 917349238')`.

[gramatica_contactos.pl](#)

Programación Declarativa: Lógica y Restricciones

El lenguaje de programación ISO-Prolog

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



POLITÉCNICA