

Tema 5: Programación lógica de segundo orden

Grupo de Programación Declarativa

{José A. Alonso, Andrés Cordon y Miguel A. Gutiérrez}

Dpto. de Ciencias de la Computación e Inteligencia Artificial

UNIVERSIDAD DE SEVILLA

Modificación de la base de conocimiento

- Predicados assert y retract

- `assert(+Term)` inserta un hecho o una cláusula en la base de conocimientos. Term es insertado como última cláusula del predicado correspondiente.
- `retract(+Term)` elimina la primera cláusula de la base de conocimientos que unifica con Term

- Ejemplos

```
?- hace_frio.  
[WARNING: Undefined predicate: 'hace_frio/0']  
No  
?- assert(hace_frio).  
Yes  
?- hace_frio.  
Yes  
?- retract(hace_frio).  
Yes  
?- hace_frio.  
No
```

Modificación de la base de conocimiento

- El predicado listing

- listing(+Pred) lista las cláusulas en cuya cabeza aparece el predicado Pred

```
?- assert( (gana(X,Y) :- rápido(X), lento(Y))).
```

```
Yes
```

```
?- listing(gana).
```

```
gana(A, B) :- rápido(A), lento(B).
```

```
Yes
```

```
?- assert(rápido(juan)),assert(lento(jose)), assert(lento(luis)).
```

```
Yes
```

```
?- gana(X,Y).
```

```
X = juan   Y = jose ;
```

```
X = juan   Y = luis ;
```

```
No
```

```
?- retract(lento(X)).
```

```
X = jose ;
```

```
X = luis ;
```

```
No
```

```
?- gana(X,Y).
```

```
No
```

Modificación de la base de conocimiento

- Los predicados `asserta` y `assertz`
 - `asserta(+Term)` equivale a `assert/1`, pero `Term` es insertado como primera cláusula del predicado correspondiente
 - `assertz(+Term)` equivale a `assert/1`
 - Ejemplos

```
?- assert(p(a)), assertz(p(b)), asserta(p(c)).
```

```
Yes
```

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

```
X = c ;
```

```
X = a ;
```

```
X = b ;
```

```
No
```

```
?- listing(p).
```

```
p(c).
```

```
p(a).
```

```
p(b).
```

```
Yes
```

Modificación de la base de conocimiento

- Los predicados `retractall` y `abolish`
 - `retractall(+Head)` elimina de la base de conocimientos todas las cláusulas cuya cabeza unifica con `Head`
 - `abolish(+SimbPred/+Aridad)` elimina de la base de conocimientos todas las cláusulas que en su cabeza aparece el símbolo de predicado `SimbPred/Aridad`
 - Ejemplo

```
?- assert(p(a)), assert(p(b)).  
Yes  
?- retractall(p(_)).  
Yes  
?- p(a).  
No  
?- assert(p(a)), assert(p(b)).  
Yes  
?- abolish(p/1).  
Yes  
?- p(a).  
[WARNING: Undefined predicate: 'p/1']  
No
```

Modificación de la base de conocimiento

- Multiplicaciones

- crea_tabla añade los hechos producto(X,Y,Z) donde X e Y son números de 0 a 9 y Z es el producto de X e Y.

```
?- crea_tabla.
```

```
Yes
```

```
?- listing(producto).
```

```
producto(0,0,0).  producto(0,1,0).  ... producto(9,8,72).  producto(9,9,81).
```

```
Yes
```

```
crea_tabla :-
```

```
    L = [0,1,2,3,4,5,6,7,8,9],
```

```
    member(X,L),
```

```
    member(Y,L),
```

```
    Z is X*Y,
```

```
    assert(producto(X,Y,Z)),
```

```
    fail.
```

```
crea_tabla.
```

- Determinar las descomposiciones de 6 en producto de dos números.

```
?- producto(A,B,6).
```

```
A=1 B=6 ; A=2 B=3 ; A=3 B=2 ; A=6 B=1 ; No
```

Modificación dinámica de la BC

- Programa

```
:- dynamic r/2.  
relacionados(X,Y)    :- assert(r(X,Y)).  
no_relacionados(X,Y) :- retract(r(X,Y)).
```

- Sesión

```
?- r(X,Y).  
No  
?- relacionados(a,b).  
Yes  
?- r(X,Y).  
X=a Y=b ; No  
?- no_relacionados(a,b).  
Yes  
?- r(a,b).  
No  
?- relacionados(a,b), relacionados(c,d).  
Yes  
?- r(X,Y).  
X=a Y=b ; X=c Y=d ; No
```

Todas las soluciones

- El predicado `findall`

- `findall(T,0,L)` se verifica si `L` es la lista de las instancias del término `T` que verifican el objetivo `0`.

```
?- assert(clase(a,voc)), assert(clase(b,con)),
    assert(clase(e,voc)), assert(clase(c,con)).
?- findall(X,clase(X,voc),L).
X = _G331    L = [a, e]
?- findall(_X,clase(_X,voc),L).
L = [a, e]
?- findall(_X,clase(_X,_Clase),L).
L = [a, b, e, c]
?- findall(X,clase(X,vocal),L).
X = _G355    L = []
?- findall(X,(member(X,[c,b,c]),member(X,[c,b,a]))),L).
X = _G373    L = [c, b, c]
?- findall(X,(member(X,[c,b,c]),member(X,[1,2,3]))),L).
X = _G373    L = []
```

Todas las soluciones

- El predicado setof

- setof(T,O,L) se verifica si L es la lista ordenada sin repeticiones de las instancias del término T que verifican el objetivo O.

```
?- setof(X,clase(X,voc),L).
X = _G331    L = [a, e]
?- setof(X,clase(X,Clase),L).
X = _G343    Clase = voc    L = [a, e] ; X = _G343    Clase = con    L = [b, c] ; No
% L = {X: (existe Y)[clase(X,Y)]}
?- setof(X,Y^clase(X,Y),L).
X = _G379    Y = _G380    L = [a, b, c, e]
?- setof(_X,_Y^clase(_X,_Y),L).
L = [a, b, c, e]
?- setof(letra(_X),_Y^clase(_X,_Y),L).
L = [letra(a), letra(b), letra(c), letra(e)]
?- setof(X,clase(X,vocal),L).
No
?- setof(X,(member(X,[c,b,c]),member(X,[c,b,a]))),L).
X = _G361    L = [b, c]
?- setof(X,(member(X,[c,b,c]),member(X,[1,2,3]))),L).
No
```

Todas las soluciones

- El predicado bagof

- bagof(T,O,L) se verifica si L es el multiconjunto de las instancias del término T que verifican el objetivo O.

```
?- bagof(X, clase(X, voc), L).
```

```
X = _G331    L = [a, e]
```

```
?- bagof(X, clase(X, Clase), L).
```

```
X = _G343    Clase = voc    L = [a, e] ; X = _G343    Clase = con    L = [b, c] ; No
```

```
% L = {X: (existe Y)[clase(X,Y)]}
```

```
?- bagof(X, Y^clase(X,Y), L).
```

```
X = _G379    Y = _G380    L = [a, b, e, c]
```

```
?- bagof(_X, _Y^clase(_X, _Y), L).
```

```
L = [a, b, e, c]
```

```
?- bagof(letra(_X), _Y^clase(_X, _Y), L).
```

```
L = [letra(a), letra(b), letra(e), letra(c)]
```

```
?- bagof(X, clase(X, vocal), L).
```

```
No
```

```
?- bagof(X, (member(X, [c,b,c]), member(X, [c,b,a]))), L).
```

```
X = _G361    L = [c, b, c]
```

```
?- bagof(X, (member(X, [c,b,c]), member(X, [1,2,3]))), L).
```

```
No
```

Todas las soluciones

- Operaciones conjuntistas

- `setof0(T,0,L)` es como `setof` salvo en el caso en que ninguna instancia de `T` verifique `0`, en cuyo caso `L` es la lista vacía. Por ejemplo,

```
?- setof0(X,(member(X,[c,a,b]),member(X,[c,b,d])),L).  
X = _G404    L = [b, c]  
?- setof0(X,(member(X,[c,a,b]),member(X,[e,f])),L).  
X = _G392    L = []
```

```
setof0(X,0,L) :- setof(X,0,L), !.  
setof0(_,_,[]).
```

- `intersección(S,T,U)` se verifica si `U` es la intersección de `S` y `T`. Por ejemplo,

```
?- intersección([1,4,2],[2,3,4],U).  
U = [2,4]
```

```
intersección(S,T,U) :-  
    setof0(X, (member(X,S), member(X,T)), U).
```

Todas las soluciones

- `diferencia(S,T,U)` se verifica si `U` es la diferencia de los conjuntos de `S` y `T`. Por ejemplo,

```
?- diferencia([5,1,2],[2,3,4],U).  
U = [1,5]
```

```
diferencia(S,T,U) :-  
    setof(X, (member(X,S), not(member(X,T))), U).
```

- `unión(S,T,U)` se verifica si `U` es la unión de `S` y `T`. Por ejemplo,

```
?- unión([1,2,4],[2,3,4],U).  
U = [1,2,3,4]
```

```
unión(S,T,U) :-  
    setof(X, (member(X,S); member(X,T)), U).
```

Todas las soluciones

- `partes(X,L)` se verifica si `L` es el conjunto de las partes de `X`. Por ejemplo,

```
?- partes([a,b,c],L).  
L = [[], [a], [a, b], [a, b, c], [a, c], [b], [b, c], [c]]
```

```
partes(X,L) :-  
    setof(Y,subconjunto(Y,X),L).
```

- `subconjunto(-L1,+L2)` se verifica si `L1` es un subconjunto de `L2`. Por ejemplo,

```
?- subconjunto(L,[a,b]).  
L = [a, b] ;  
L = [a] ;  
L = [b] ;  
L = [] ;  
No  
subconjunto([],[]).  
subconjunto([X|L1],[X|L2]) :-  
    subconjunto(L1,L2).  
subconjunto(L1,[_|L2]) :-  
    subconjunto(L1,L2).
```

Procesamiento de términos

- Transformación entre términos y listas

- $?T =.. ?L$ se verifica si L es una lista cuyo primer elemento es el functor del término T y los restantes elementos de L son los argumentos de T . Por ejemplo,

```
?- padre(juan, luis) =.. L.  
L = [padre, juan, luis]  
?- T =.. [padre, juan, luis].  
T = padre(juan, luis)
```

- $alarga(+F1, +N, -F2)$ se verifica si $F1$ y $F2$ son figuras geométricas del mismo tipo y el tamaño de la $F1$ es el de la $F2$ multiplicado por N , donde las figuras geométricas se representan como términos en los que el functor indica el tipo de figura y los argumentos su tamaño; por ejemplo,

```
?- alarga(triángulo(3, 4, 5), 2, F).  
F = triángulo(6, 8, 10)  
  
?- alarga(cuadrado(3), 2, F).  
F = cuadrado(6)
```

Procesamiento de términos

- Definición

```
alarga(Figura1,Factor,Figura2) :-  
    Figura1 =.. [Tipo|Argumentos1],  
    multiplica_lista(Argumentos1,Factor,Argumentos2),  
    Figura2 =.. [Tipo|Argumentos2].
```

```
multiplica_lista([],_,[]).  
multiplica_lista([X1|L1],F,[X2|L2]) :-  
    X2 is X1*F,  
    multiplica_lista(L1,F,L2).
```

- Los procedimientos functor y arg

- functor(T,F,A) se verifica si F es el functor del término T y A es su aridad.
- arg(N,T,A) se verifica si A es el argumento del término T que ocupa el lugar N.

```
?- functor(g(b,c,d),F,A).           => F = g      A = 3  
?- functor(T,g,2).                  => T = g(_G237,_G238)  
?- arg(2,g(b,c,d),X).               => X = c  
?- functor(T,g,3),arg(1,T,b),arg(2,T,c). => T = g(b, c, _G405)
```

Transformaciones entre átomos y listas

- La relación name

- name(A,L) se verifica si L es la lista de códigos ASCII de los caracteres del átomo A.
Por ejemplo,

```
?- name(bandera,L).  
L = [98, 97, 110, 100, 101, 114, 97]  
?- name(A,[98, 97, 110, 100, 101, 114, 97]).  
A = bandera
```

- concatena_átomos(A1,A2,A3) se verifica si A3 es la concatenación de los átomos A1 y A2. Por ejemplo,

```
?- concatena_átomos(pi,ojo,X).  
X = piojo
```

```
concatena_átomos(A1,A2,A3) :-  
    name(A1,L1),  
    name(A2,L2),  
    append(L1,L2,L3),  
    name(A3,L3).
```

Procedimientos aplicativos

- El procedimiento apply

- `apply(T,L)` se verifica si es demostrable `T` después de aumentar el número de sus argumentos con los elementos de `L`; por ejemplo,

```
?- plus(2,3,X).           => X = 5
?- apply(plus,[2,3,X]).   => X = 5
?- apply(plus(2),[3,X]).  => X = 5
?- apply(plus(2,3),[X]).  => X = 5
?- apply(append([1,2]),[X,[1,2,3,4,5]]). => X = [3, 4, 5]
```

- Definición de apply

```
n_apply(Término,Lista) :-
    Término =.. [Pred|Arg1],
    append(Arg1,Lista,Arg2),
    Átomo =.. [Pred|Arg2],
    Átomo.
```

Procedimientos aplicativos

- El procedimiento `maplist`

- `maplist(P,L1,L2)` se verifica si se cumple el predicado `P` sobre los sucesivos pares de elementos de las listas `L1` y `L2`; por ejemplo,

```
?- succ(2,X).           => 3
?- succ(X,3).           => 2
?- maplist(succ,[2,4],[3,5]). => Yes
?- maplist(succ,[0,4],[3,5]). => No
?- maplist(succ,[2,4],Y).  => Y = [3, 5]
?- maplist(succ,X,[3,5]).  => X = [2, 4]
```

- Definición de `maplist`

```
n_maplist(_,[],[]).
n_maplist(R,[X1|L1],[X2|L2]) :-
    apply(R,[X1,X2]),
    n_maplist(R,L1,L2).
```

Predicados sobre tipos de término

- `var(T)` se verifica si `T` es una variable.
`atom(T)` se verifica si `T` es un átomo.
`number(T)` se verifica si `T` es un número.
`compound(T)` se verifica si `T` es un término compuesto.
`atomic(T)` se verifica si `T` es una variable, átomo, cadena o número).

<code>?- var(X1).</code>	<code>=></code>	<code>Yes</code>
<code>?- X1=a, var(X1).</code>	<code>=></code>	<code>No</code>
<code>?- atom(átomo).</code>	<code>=></code>	<code>Yes</code>
<code>?- atom(3).</code>	<code>=></code>	<code>No</code>
<code>?- atom(f(a,b)).</code>	<code>=></code>	<code>No</code>
<code>?- number(123).</code>	<code>=></code>	<code>Yes</code>
<code>?- number(-25.14).</code>	<code>=></code>	<code>Yes</code>
<code>?- number(1+3).</code>	<code>=></code>	<code>No</code>
<code>?- compound(f(X,a)).</code>	<code>=></code>	<code>Yes</code>
<code>?- compound([1,2]).</code>	<code>=></code>	<code>Yes</code>
<code>?- compound(abc).</code>	<code>=></code>	<code>No</code>
<code>?- atomic(átomo).</code>	<code>=></code>	<code>Yes</code>
<code>?- atomic(123).</code>	<code>=></code>	<code>Yes</code>
<code>?- atomic(X).</code>	<code>=></code>	<code>No</code>
<code>?- atomic(f(1,2)).</code>	<code>=></code>	<code>No</code>

Predicados sobre tipos de término

- Suma segura

- Definir `suma_segura(X,Y,Z)` que se verifique si `X` e `Y` son enteros y `Z` es la suma de `X` e `Y`. Por ejemplo,

```
?- suma_segura(2,3,X).
```

```
X = 5
```

```
Yes
```

```
?- suma_segura(7,a,X).
```

```
No
```

```
?- X is 7 + a.
```

```
[WARNING: Arithmetic: 'a' is not a function]
```

```
suma_segura(X,Y,Z) :-
```

```
    number(X),
```

```
    number(Y),
```

```
    Z is X+Y.
```

Comparación y ordenación de términos

- Comparación de términos
 - $T1 = T2$ se verifica si $T1$ y $T2$ son unificables.
 - $T1 == T2$ se verifica si $T1$ y $T2$ son idénticos.

```
?- f(X) = f(Y).
```

```
X = _G164
```

```
Y = _G164
```

```
Yes
```

```
?- f(X) == f(Y).
```

```
No
```

```
?- f(X) == f(X).
```

```
X = _G170
```

```
Yes
```

Comparación y ordenación de términos

- Definir el predicado `cuenta(A,L,N)` que se verifique si `N` es el número de ocurrencias del átomo `A` en la lista `L`. Por ejemplo,

```
?- cuenta(a,[a,b,a,a],N).
```

```
N = 3
```

```
?- cuenta(a,[a,b,X,Y],N).
```

```
N = 1
```

```
cuenta(_,[],0).
```

```
cuenta(A,[B|L],N) :-
```

```
    A == B, !,
```

```
    cuenta(A,L,M),
```

```
    N is M+1.
```

```
cuenta(A,[B|L],N) :-
```

```
    % A \== B,
```

```
    cuenta(A,L,N).
```

Comparación y ordenación de términos

- Ordenación de términos

- $T1 @< T2$ se verifica si el término $T1$ es anterior que $T2$ en el orden de términos de Prolog

?- X @< 3.	=> Yes
?- ab @< ac.	=> Yes
?- 21 @< 123.	=> Yes
?- 12 @< a.	=> Yes
?- g @< f(b).	=> Yes
?- f(b) @< f(a,b).	=> Yes
?- [a,1] @< [a,3].	=> Yes
?- [a] @< [a,3].	=> Yes

- Ordenación con sort

- $\text{sort}(+L1, -L2)$ se verifica si $L2$ es la lista obtenida ordenando de manera creciente los distintos elementos de $L1$ y eliminando las repeticiones.

```
?- sort([c4,2,a5,2,c3,a5,2,a5],L).  
L = [2, a5, c3, c4]
```

Bibliografía

- Alonso, J.A. y Borrego, J. *Deducción automática (Vol. 1: Construcción lógica de sistemas lógicos)* (Ed. Kronos, 2002)
(www.cs.us.es/~jalonso/libros/da1-02.pdf)
 - Cap. 2: “Introducción a la programación lógica con Prolog”
- Bratko, I. *Prolog Programming for Artificial Intelligence (3 ed.)* (Addison–Wesley, 2001)
 - Cap. 7: “More Built–in Procedures”
- Van Le, T. *Techniques of Prolog Programming* (John Wiley, 1993)
 - Cap. 6: “Advanced programming techniques and data structures”
- Clocksin, W.F. y Mellish, C.S. *Programming in Prolog (Fourth Edition)* (Springer Verlag, 1994)
 - Cap. 6: “Built–in Predicates”