Programación Lógica - Parte 2

Paradigmas (de Lenguajes) de Programación

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

13 de junio de 2025

Nomenclatura para patrones de instanciación

Por convención se aclara mediante prefijos en los comentarios:

- `p(+A)` indica que `A` debe proveerse instanciado.
- `p(-A)` indica que `A` no debe estar instanciado.
- `p(?A)` indica que `A` puede o no proveerse instanciado.
- Existe un último caso en donde un argumento puede aparecer **semi instanciado** (es decir, contiene variables libres), por ejemplo: `[p,r,o,X,o,_]` unifica con `[p,r,o,l,o,g]` pero no con `[]` o `prolog`.

Nomenclatura para patrones de instanciación

Por convención se aclara mediante prefijos en los comentarios:

- `p(+A)` indica que `A` debe proveerse instanciado.
- `p(-A)` indica que `A` no debe estar instanciado.
- `p(?A)` indica que `A` puede o no proveerse instanciado.
- Existe un último caso en donde un argumento puede aparecer **semi instanciado** (es decir, contiene variables libres), por ejemplo: `[p,r,o,X,o,_]` unifica con `[p,r,o,l,o,g]` pero no con `[]` o `prolog`.

Predicados útiles

- `var(A)` tiene éxito si A es una variable libre.
- `nonvar(A)` tiene éxito si A no es una variable libre.
- `ground(A)` tiene éxito si A no contiene variables libres.

Ejercicio

Ejercicio: iésimo

- Implementar el predicado `iesimo(+I, +L, -X)`, donde `X` es el iésimo elemento de la lista `L`.

Ejercicio

Ejercicio: iésimo

- Implementar el predicado `iesimo(+I, +L, -X)`, donde `X` es el iésimo elemento de la lista `L`.
- ¿Es nuestra implementación reversible en `I`? Si no lo es, hacer una versión reversible.

Ejercicio

El predicado `desde`.

```
desde(X, X).
desde(X, Y) :- N is X+1, desde(N, Y).
```

Ejercicio

El predicado `desde`.

```
desde(X, X).
desde(X, Y) :- N is X+1, desde(N, Y).
```

Ejercicio: desde

- ¿Cómo deben instanciarse los parámetros para que el predicado funcione? (es decir, para que no se cuelgue ni produzca un error). ¿Por qué?

Ejercicio

El predicado `desde`.

```
desde(X, X).
desde(X, Y) :- N is X+1, desde(N, Y).
```

Ejercicio: desde

- ¿Cómo deben instanciarse los parámetros para que el predicado funcione? (es decir, para que no se cuelgue ni produzca un error). ¿Por qué?
- Implementar el predicado `desdeReversible(+X,?Y)` tal que si `Y` está instanciada, sea verdadero si `Y` es mayor o igual que `X`, y si no lo está, genere todos los `Y` de `X` en adelante.

Ejercicio

Definir el predicado `pmq(+X, -Y)` que genera todos los naturales pares menores o iguales a `X`.

Esquema general de Generate & Test

Una técnica que usaremos muy a menudo es:

1. Generar todas las posibles soluciones de un problema.
   (Léase, los *candidatos* a solución, según cierto criterio general)
2. Testear cada una de las soluciones generadas.
   (Hacer que fallen los candidatos que no cumplan cierto criterio particular)

La idea se basa fuertemente en el *orden* en que se procesan las reglas.

Esquema general de Generate & Test

Un predicado que usa el esquema G&T se define mediante otros dos:

```
pred(X1,...,Xn) :- generate(X1, ...,Xm), test(X1, ...,Xm).
```

Esta división de tareas implica que:

- `generate(...)` deberá **instanciar** ciertas variables.
- `test(...)` deberá **verificar** si los valores intanciados pertenecen a la solución, pudiendo para ello asumir que ya está instanciada.

Generate & Test

Ejercicio

- Definir el predicado `coprimos(-X, -Y)` que instancia en `X` e `Y` **todos** los pares de números coprimos.
- Tip: utilizar la función gcd del motor aritmético. `X is gcd(15, 8)` instancia `X=1`, por lo que 15 y 8 son coprimos.

Generate & Test

Ejercicio

- Definir el predicado `coprimos(-X, -Y)` que instancia en `X` e `Y` **todos** los pares de números coprimos.
- Tip: utilizar la función gcd del motor aritmético. `X is gcd(15, 8)` instancia `X=1`, por lo que 15 y 8 son coprimos.
- ¿Es reversible en `X` e `Y`? Justificar.

Soluciones repetidas

Algunos hechos sobre la tragedia de Edipo

```
progenitorx(yocasta,edipo).
progenitorx(yocasta,antígona).
progenitorx(edipo,antígona).

abuela(X,Y) :- progenitorx(X,Z),
progenitorx(Z,Y).

pariente(X,Y) :- progenitorx(X,Y).
pariente(X,Y) :- abuela(X,Y)
```

Soluciones repetidas

Algunos hechos sobre la tragedia de Edipo

```
progenitorx(yocasta,edipo).
progenitorx(yocasta,antígona).
progenitorx(edipo,antígona).

abuela(X,Y) :- progenitorx(X,Z),
progenitorx(Z,Y).

pariente(X,Y) :- progenitorx(X,Y).
pariente(X,Y) :- abuela(X,Y)
```

Consulta

```
?- pariente(yocasta,X).
```

Soluciones repetidas

Algunos hechos sobre la tragedia de Edipo

```
progenitorx(yocasta,edipo).
progenitorx(yocasta,antígona).
progenitorx(edipo,antígona).

abuela(X,Y) :- progenitorx(X,Z),
progenitorx(Z,Y).

pariente(X,Y) :- progenitorx(X,Y).
pariente(X,Y) :- abuela(X,Y)
```

Consulta

```
?- pariente(yocasta,X).
```

Resultados

```
X = edipo;
```

Soluciones repetidas

Algunos hechos sobre la tragedia de Edipo

```
progenitorx(yocasta,edipo).
progenitorx(yocasta,antígona).
progenitorx(edipo,antígona).

abuela(X,Y) :- progenitorx(X,Z),
progenitorx(Z,Y).

pariente(X,Y) :- progenitorx(X,Y).
pariente(X,Y) :- abuela(X,Y)
```

Consulta

```
?- pariente(yocasta,X).
```

Resultados

```
X = edipo;

X = antígona;
```

Soluciones repetidas

Algunos hechos sobre la tragedia de Edipo

```
progenitorx(yocasta,edipo).
progenitorx(yocasta,antígona).
progenitorx(edipo,antígona).

abuela(X,Y) :- progenitorx(X,Z),
progenitorx(Z,Y).

pariente(X,Y) :- progenitorx(X,Y).
pariente(X,Y) :- abuela(X,Y)
```

Consulta

```
?- pariente(yocasta,X).
```

Resultados

```
X = edipo;

X = antígona;

X = antígona;
```

Soluciones repetidas

Algunos hechos sobre la tragedia de Edipo

```
progenitorx(yocasta,edipo).
progenitorx(yocasta,antígona).
progenitorx(edipo,antígona).

abuela(X,Y) :- progenitorx(X,Z),
progenitorx(Z,Y).

pariente(X,Y) :- progenitorx(X,Y).
pariente(X,Y) :- abuela(X,Y)
```

Consulta

```
?- pariente(yocasta,X).
```

Resultados

```
X = edipo;

X = antígona;

X = antígona;

false.
```

- ¿Razonable o erróneo?

Soluciones repetidas

Algunos hechos sobre la tragedia de Edipo

```
progenitorx(yocasta,edipo).
progenitorx(yocasta,antígona).
progenitorx(edipo,antígona).

abuela(X,Y) :- progenitorx(X,Z),
progenitorx(Z,Y).

pariente(X,Y) :- progenitorx(X,Y).
pariente(X,Y) :- abuela(X,Y)
```

Consulta

```
?- pariente(yocasta,X).
```

Resultados

```
X = edipo;

X = antígona;

X = antígona;

false.
```

- ¿Razonable o erróneo?
- ¿Cómo hacer para evitar repeticiones no deseadas?

El metapredicado not

Definición

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

El metapredicado `not`

Definición

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

- `not(p(X1, ..., Xn))` tiene éxito si no existe instanciación posible para las variables no instanciadas en {`X1...Xn`} que haga que P tenga éxito.
- el `not` no deja instanciadas las variables libres luego de su ejecución.

Cómo evitar soluciones repetidas

Una buena idea: usando cláusulas excluyentes.

Algunos hechos sobre la tragedia de Edipo

```
progenitorx(yocasta,edipo).
progenitorx(yocasta,antígona).
progenitorx(edipo,antígona).

abuela(X,Y) :- progenitorx(X,Z), progenitorx(Z,Y).

pariente(X,Y) :- progenitorx(X,Y), not(abuela(X,Y)).
pariente(X,Y) :- abuela(X,Y).
```

Cómo evitar soluciones repetidas

Una buena idea: usando cláusulas excluyentes.

Algunos hechos sobre la tragedia de Edipo

```
progenitorx(yocasta,edipo).
progenitorx(yocasta,antígona).
progenitorx(edipo,antígona).

abuela(X,Y) :- progenitorx(X,Z), progenitorx(Z,Y).

pariente(X,Y) :- progenitorx(X,Y), not(abuela(X,Y)).
pariente(X,Y) :- abuela(X,Y).
```

¡Esto no funciona! ¿Por qué?

```
pariente(X,Y) :- not(abuela(X,Y)), progenitorx(X,Y).
pariente(X,Y) :- abuela(X,Y).
```

Negación por Falla

Ejercicio

Definir el predicado `corteMásParejo(+L,-L1,-L2)` donde `L` es una lista de números, y `L1` y `L2` representan el corte más parejo posible de `L` respecto a la suma de sus elementos (predicado `sumlist/2`). Puede haber más de un resultado.

Negación por Falla

Ejercicio

Definir el predicado `corteMásParejo(+L,-L1,-L2)` donde `L` es una lista de números, y `L1` y `L2` representan el corte más parejo posible de `L` respecto a la suma de sus elementos (predicado `sumlist/2`). Puede haber más de un resultado.

```
corteMásParejo([1,2,3,4,2],I,D). ⇝ I = [1, 2, 3],
                                   D = [4, 2] ;
                                   false.
corteMásParejo([1,2,1],I,D).     ⇝ I = [1], D = [2, 1] ;
                                   I = [1, 2], D = [1] ;
                                   false.
```

Negación por Falla

Ejercicio

Definir el predicado `próximoPrimo(+N,-P)` que instancia en `P` el siguiente número primo a partir de `N`.

Negación por Falla

Ejercicio

Definir el predicado `próximoPrimo(+N,-P)` que instancia en `P` el siguiente número primo a partir de `N`.

```
próximoPrimo(32,P). ⇝ P = 37;
                      false.
próximoPrimo(37,P). ⇝ P = 41;
                      false.
```

Otros metapredicados

`setof(-Var, +Goal, -Set)`
unifica `Set` con la lista *ordenada* y *sin repetidos* de `Var` que satisfacen `Goal`.

Otros metapredicados

`setof(-Var, +Goal, -Set)`
unifica `Set` con la lista *ordenada* y *sin repetidos* de `Var` que satisfacen `Goal`.

Uso

- `setof(X, p(X), L)` instancia `L` en el conjunto de `X` tales que `p(X)`.

Otros metapredicados

`setof(-Var, +Goal, -Set)`
unifica `Set` con la lista *ordenada* y *sin repetidos* de `Var` que satisfacen `Goal`.

Uso

- `setof(X, p(X), L)` instancia `L` en el conjunto de `X` tales que `p(X)`.
- Un ejemplo:

```
primeraComponente([(X,_)|_],X).
primeraComponente([_|XS],X) :- primeraComponente(XS,X).
```

Otros metapredicados

`setof(-Var, +Goal, -Set)`
unifica `Set` con la lista *ordenada* y *sin repetidos* de `Var` que satisfacen `Goal`.

Uso

- `setof(X, p(X), L)` instancia `L` en el conjunto de `X` tales que `p(X)`.
- Un ejemplo:

```
primeraComponente([(X,_)|_],X).
primeraComponente([_|XS],X) :- primeraComponente(XS,X).
```

```
?- setof(X,primeraComponente([(2,2),(1,3),(1,4)],X),L).
```

Otros metapredicados

`setof(-Var, +Goal, -Set)`
unifica `Set` con la lista *ordenada* y *sin repetidos* de `Var` que satisfacen `Goal`.

Uso

- `setof(X, p(X), L)` instancia `L` en el conjunto de `X` tales que `p(X)`.
- Un ejemplo:

```
primeraComponente([(X,_)|_],X).
primeraComponente([_|XS],X) :- primeraComponente(XS,X).
```

```
?- setof(X,primeraComponente([(2,2),(1,3),(1,4)],X),L).
L = [1,2].
```

Otros metapredicados

`bagof(-Var, +Goal, -Set)`
igual que `setof` pero la lista mantiene el orden en el que obtiene las soluciones y puede tener repetidos.

Otros metapredicados

`bagof(-Var, +Goal, -Set)`
igual que `setof` pero la lista mantiene el orden en el que obtiene las soluciones y puede tener repetidos.

Uso

- `bagof(X, p(X), L)` instancia `L` en el conjunto de `X` tales que `p(X)`.

Otros metapredicados

`bagof(-Var, +Goal, -Set)`
igual que `setof` pero la lista mantiene el orden en el que obtiene las soluciones y puede tener repetidos.

Uso

- `bagof(X, p(X), L)` instancia `L` en el conjunto de `X` tales que `p(X)`.
- Un ejemplo:

```
primeraComponente([(X,_)|_],X).
primeraComponente([_|XS],X) :- primeraComponente(XS,X).
```

```
?- bagof(X,primeraComponente([(2,2),(1,3),(1,4)],X),L).
```

Otros metapredicados

`bagof(-Var, +Goal, -Set)`
igual que `setof` pero la lista mantiene el orden en el que obtiene las soluciones y puede tener repetidos.

Uso

- `bagof(X, p(X), L)` instancia `L` en el conjunto de `X` tales que `p(X)`.
- Un ejemplo:

```
primeraComponente([(X,_)|_],X).
primeraComponente([_|XS],X) :- primeraComponente(XS,X).
```

```
?- bagof(X,primeraComponente([(2,2),(1,3),(1,4)],X),L).
L = [2,1,1].
```

Otros metapredicados

`findall(-Var, +Goal, -Set)`
igual que `bagof` pero ignora otras variables libres dentro del `Goal` que no sea `Var`. Además devuelve la lista vacía si no existen soluciones.

Otros metapredicados

`findall(-Var, +Goal, -Set)`
igual que `bagof` pero ignora otras variables libres dentro del `Goal` que no sea `Var`. Además devuelve la lista vacía si no existen soluciones.

Uso

- `findall(X, p(X), L)` instancia `L` en el conjunto de `X` tales que `p(X)`.

Otros metapredicados

`findall(-Var, +Goal, -Set)`
igual que `bagof` pero ignora otras variables libres dentro del `Goal` que no sea `Var`. Además devuelve la lista vacía si no existen soluciones.

Uso

- `findall(X, p(X), L)` instancia `L` en el conjunto de `X` tales que `p(X)`.
- Un ejemplo:

```
?- findall(X, member((X,Y),[(1,2),(3,4)]),L).
```

Otros metapredicados

`findall(-Var, +Goal, -Set)`
igual que `bagof` pero ignora otras variables libres dentro del `Goal` que no sea `Var`. Además devuelve la lista vacía si no existen soluciones.

Uso

- `findall(X, p(X), L)` instancia `L` en el conjunto de `X` tales que `p(X)`.
- Un ejemplo:

```
?- findall(X, member((X,Y),[(1,2),(3,4)]),L).
L = [1,3].
```

Otros metapredicados

`maplist(+Goal, ?List)`
es verdadero cuando el `Goal` satisface a todos los elementos de la lista

Otros metapredicados

`maplist(+Goal, ?List)`
es verdadero cuando el `Goal` satisface a todos los elementos de la lista

Uso

```
esPar(X) :- mod(X,2) =:= 0.
uno(1).

?- maplist(esPar,[2,6,8]).
```

Otros metapredicados

`maplist(+Goal, ?List)`
es verdadero cuando el `Goal` satisface a todos los elementos de la lista

Uso

```
esPar(X) :- mod(X,2) =:= 0.
uno(1).

?- maplist(esPar,[2,6,8]).
true.
```

Otros metapredicados

`maplist(+Goal, ?List)`
es verdadero cuando el `Goal` satisface a todos los elementos de la lista

Uso

```
esPar(X) :- mod(X,2) =:= 0.
uno(1).

?- maplist(esPar,[2,6,8]).
true.

?- length(L,4), maplist(uno,L).
```

Otros metapredicados

`maplist(+Goal, ?List)`
es verdadero cuando el `Goal` satisface a todos los elementos de la lista

Uso

```
esPar(X) :- mod(X,2) =:= 0.
uno(1).

?- maplist(esPar,[2,6,8]).
true.

?- length(L,4), maplist(uno,L).
L = [1, 1, 1, 1].
```

Otros metapredicados

`maplist(+Goal, ?List)`
es verdadero cuando el `Goal` satisface a todos los elementos de la lista

Uso

```
esPar(X) :- mod(X,2) =:= 0.
uno(1).

?- maplist(esPar,[2,6,8]).
true.

?- length(L,4), maplist(uno,L).
L = [1, 1, 1, 1].
```

Otros metapredicados

`maplist(+Goal, ?List1, ?List2)`
es verdadero cuando el `Goal` satisface a todos los elementos de las listas

Otros metapredicados

`maplist(+Goal, ?List1, ?List2)`
es verdadero cuando el `Goal` satisface a todos los elementos de las listas

Uso

```
sumarK(K,X,Y) :- Y is X+K.

?- maplist(sumarK(4),[1,2,3],L).
```

Otros metapredicados

`maplist(+Goal, ?List1, ?List2)`
es verdadero cuando el `Goal` satisface a todos los elementos de las listas

Uso

```
sumarK(K,X,Y) :- Y is X+K.

?- maplist(sumarK(4),[1,2,3],L).
L = [5,6,7].
```

Otros metapredicados

`limit(+Count,+Goal)`
limita el número de soluciones del `Goal`.

Otros metapredicados

`limit(+Count,+Goal)`
limita el número de soluciones del `Goal`.

Uso

```
?- limit(5,desde(1,X)).
```

Otros metapredicados

`limit(+Count,+Goal)`
limita el número de soluciones del `Goal`.

Uso

```
?- limit(5,desde(1,X)).
X = 1;
X = 2;
X = 3;
X = 4;
X = 5.
```

Otros metapredicados

`forall(+Gen,+Cond)`
verifica que todas las soluciones de `Gen` satisfagan `Cond`.

Otros metapredicados

`forall(+Gen,+Cond)`

verifica que todas las soluciones de `Gen` satisfagan `Cond`.

Uso

```
unCorte(L,L1,L2,D), forall(unCorte(L,_,_,D2),D2 >= D).
```

Generación infinita: triángulos

Suponiendo que los triángulos se representan con `tri(A,B,C)` cuyos lados tienen longitudes `A`, `B` y `C` respectivamente. Se asume que las longitudes de los lados son siempre números naturales mayores a cero. Se cuenta con el predicado `esTriangulo(+T)` que es verdadero cuando `T` es una estructura de la forma `tri(A,B,C)` que representa un triángulo válido (cada lado es menor que la suma de los otros dos, y mayor que su diferencia).

Generación infinita: triángulos

Suponiendo que los triángulos se representan con `tri(A,B,C)` cuyos lados tienen longitudes `A`, `B` y `C` respectivamente. Se asume que las longitudes de los lados son siempre números naturales mayores a cero. Se cuenta con el predicado `esTriangulo(+T)` que es verdadero cuando `T` es una estructura de la forma `tri(A,B,C)` que representa un triángulo válido (cada lado es menor que la suma de los otros dos, y mayor que su diferencia).

Ejercicio

- Implementar un predicado `perímetro(?T,?P)` que es verdadero cuando `T` es un triángulo y `P` es su perímetro. No se deben generar resultados repetidos (no tendremos en cuenta la congruencia entre triángulos: si dos triángulos tienen las mismas longitudes, pero en diferente orden, se considerarán diferentes entre sí). El predicado debe funcionar para cualquier instanciación de T y P (no es necesario que funcione para triángulos parcialmente instanciados).

Generación infinita: triángulos

Suponiendo que los triángulos se representan con `tri(A,B,C)` cuyos lados tienen longitudes `A`, `B` y `C` respectivamente. Se asume que las longitudes de los lados son siempre números naturales mayores a cero. Se cuenta con el predicado `esTriangulo(+T)` que es verdadero cuando `T` es una estructura de la forma `tri(A,B,C)` que representa un triángulo válido (cada lado es menor que la suma de los otros dos, y mayor que su diferencia).

Ejercicio

- Implementar un predicado `perímetro(?T,?P)` que es verdadero cuando `T` es un triángulo y `P` es su perímetro. No se deben generar resultados repetidos (no tendremos en cuenta la congruencia entre triángulos: si dos triángulos tienen las mismas longitudes, pero en diferente orden, se considerarán diferentes entre sí). El predicado debe funcionar para cualquier instanciación de T y P (no es necesario que funcione para triángulos parcialmente instanciados).
- Implementar un generador de triángulos válidos, sin repetir resultados: `triángulo(-T)`.

Fin

¿ ¿ ¿ ¿ ¿ ¿Preguntas? ? ? ? ? ?