

Clase práctica: Prolog

(Parte 2)

Paradigmas de Lenguajes de Programación

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

12 de Octubre de 2017

Segunda clase práctica de Prolog

- 1 Instanciación de variables
- 2 Generate & Test
- 3 Evitando repetir resultados y Negación por falla
- 4 Generación infinita
- 5 Resumen parte teórica

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`.
- Nuestra implementación no anda, ¿Cómo probar? Trace + Debug.

Ejercicio

Ejercicio: iésimo

- Implementar el predicado `iesimo(+I, +L, -X)`, donde `X` es el iésimo elemento de la lista `L`.
- Nuestra implementación no anda, ¿Cómo probar? Trace + Debug.
- ¿Es nuestra implementación reversible en `I`?

Ejercicio

Ejercicio: iésimo

- Implementar el predicado `iesimo(+I, +L, -X)`, donde `X` es el iésimo elemento de la lista `L`.
- Nuestra implementación no anda, ¿Cómo probar? Trace + Debug.
- ¿Es nuestra implementación reversible en `I`?
- Escribir una nueva versión `iesimo2(?I, +L, -X)`

Ejercicio

Ejercicio: iésimo

- Implementar el predicado `iesimo(+I, +L, -X)`, donde `X` es el iésimo elemento de la lista `L`.
- Nuestra implementación no anda, ¿Cómo probar? Trace + Debug.
- ¿Es nuestra implementación reversible en `I`?
- Escribir una nueva versión `iesimo2(?I, +L, -X)`

El predicado desde.

`desde(X, X).`

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

Ejercicio

Ejercicio: iésimo

- Implementar el predicado `iesimo(+I, +L, -X)`, donde `X` es el `iésimo` elemento de la lista `L`.
- Nuestra implementación no anda, ¿Cómo probar? Trace + Debug.
- ¿Es nuestra implementación reversible en `I`?
- Escribir una nueva versión `iesimo2(?I, +L, -X)`

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

Ejercicio: iésimo

- Implementar el predicado `iesimo(+I, +L, -X)`, donde `X` es el `iésimo` elemento de la lista `L`.
- Nuestra implementación no anda, ¿Cómo probar? Trace + Debug.
- ¿Es nuestra implementación reversible en `I`?
- Escribir una nueva versión `iesimo2(?I, +L, -X)`

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 `desde2(+X, ?Y).`

Segunda clase práctica de Prolog

- 1 Instanciación de variables
- 2 Generate & Test**
- 3 Evitando repetir resultados y Negación por falla
- 4 Generación infinita
- 5 Resumen parte teórica

Generate & Test

Ejercicio: `pmq(+X, -Y)`.

Que genera todos los naturales pares menores o iguales a X.

Tip: Utilizar el siguiente predicado

```
par(X) :- 0 is X mod 2.
```

Generate & Test

Ejercicio: `pmq(+X, -Y)`.

Que genera todos los naturales pares menores o iguales a X.

Tip: Utilizar el siguiente predicado

```
par(X) :- 0 is X mod 2.
```

Ejercicio: `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(2, 4) instancia X=2.
```

Segunda clase práctica de Prolog

- 1 Instanciación de variables
- 2 Generate & Test
- 3 Evitando repetir resultados y Negación por falla
- 4 Generación infinita
- 5 Resumen parte teórica

Ejemplo de soluciones repetidas

Algunos hechos sobre materias de cierta carrera

```
altaMateria(plp).  
altaMateria(aa).  
altaMateria(metnum).  
  
liviana(plp).  
liviana(aa).  
liviana(eci).  
  
obligatoria(plp).  
obligatoria(metnum).  
  
leGusta(M) :- altaMateria(M).  
leGusta(M) :- liviana(M).  
  
hacer(M) :- leGusta(M), obligatoria(M).
```

Ejemplo de soluciones repetidas

Algunos hechos sobre materias de cierta carrera

```
altaMateria(plp).  
altaMateria(aa).  
altaMateria(metnum).  
  
liviana(plp).  
liviana(aa).  
liviana(eci).  
  
obligatoria(plp).  
obligatoria(metnum).  
  
leGusta(M) :- altaMateria(M).  
leGusta(M) :- liviana(M).  
  
hacer(M) :- leGusta(M), obligatoria(M).
```

Consulta

```
?- hacer(Materia).
```

Ejemplo de soluciones repetidas

Algunos hechos sobre materias de cierta carrera

```
altaMateria(plp).  
altaMateria(aa).  
altaMateria(metnum).  
  
liviana(plp).  
liviana(aa).  
liviana(eci).  
  
obligatoria(plp).  
obligatoria(metnum).  
  
leGusta(M) :- altaMateria(M).  
leGusta(M) :- liviana(M).  
  
hacer(M) :- leGusta(M), obligatoria(M).
```

Consulta

```
?- hacer(Materia).
```

Resultados

```
Materia = plp ;
```


Ejemplo de soluciones repetidas

Algunos hechos sobre materias de cierta carrera

```
altaMateria(plp).  
altaMateria(aa).  
altaMateria(metnum).  
  
liviana(plp).  
liviana(aa).  
liviana(eci).  
  
obligatoria(plp).  
obligatoria(metnum).  
  
leGusta(M) :- altaMateria(M).  
leGusta(M) :- liviana(M).  
  
hacer(M) :- leGusta(M), obligatoria(M).
```

Consulta

```
?- hacer(Materia).
```

Resultados

```
Materia = plp ;
```

```
Materia = metnum ;
```

Ejemplo de soluciones repetidas

Algunos hechos sobre materias de cierta carrera

```
altaMateria(plp).  
altaMateria(aa).  
altaMateria(metnum).  
  
liviana(plp).  
liviana(aa).  
liviana(eci).  
  
obligatoria(plp).  
obligatoria(metnum).  
  
leGusta(M) :- altaMateria(M).  
leGusta(M) :- liviana(M).  
  
hacer(M) :- leGusta(M), obligatoria(M).
```

Consulta

```
?- hacer(Materia).
```

Resultados

```
Materia = plp ;
```

```
Materia = metnum ;
```

```
Materia = plp ;
```

Ejemplo de soluciones repetidas

Algunos hechos sobre materias de cierta carrera

```
altaMateria(plp).  
altaMateria(aa).  
altaMateria(metnum).  
  
liviana(plp).  
liviana(aa).  
liviana(eci).  
  
obligatoria(plp).  
obligatoria(metnum).  
  
leGusta(M) :- altaMateria(M).  
leGusta(M) :- liviana(M).  
  
hacer(M) :- leGusta(M), obligatoria(M).
```

Consulta

```
?- hacer(Materia).
```

Resultados

```
Materia = plp ;  
Materia = metnum ;  
Materia = plp ;  
false.
```

Ejemplo de soluciones repetidas

Algunos hechos sobre materias de cierta carrera

```
altaMateria(plp).  
altaMateria(aa).  
altaMateria(metnum).  
  
liviana(plp).  
liviana(aa).  
liviana(eci).  
  
obligatoria(plp).  
obligatoria(metnum).  
  
leGusta(M) :- altaMateria(M).  
leGusta(M) :- liviana(M).  
  
hacer(M) :- leGusta(M), obligatoria(M).
```

Consulta

```
?- hacer(Materia).
```

Resultados

```
Materia = plp ;  
Materia = metnum ;  
Materia = plp ;  
false.
```

- ¿Razonable o erróneo? ¿En este caso? ¿Y en general?

Ejemplo de soluciones repetidas

Algunos hechos sobre materias de cierta carrera

```
altaMateria(plp).  
altaMateria(aa).  
altaMateria(metnum).  
  
liviana(plp).  
liviana(aa).  
liviana(eci).  
  
obligatoria(plp).  
obligatoria(metnum).  
  
leGusta(M) :- altaMateria(M).  
leGusta(M) :- liviana(M).  
  
hacer(M) :- leGusta(M), obligatoria(M).
```

Consulta

```
?- hacer(Materia).
```

Resultados

```
Materia = plp ;  
Materia = metnum ;  
Materia = plp ;  
false.
```

- ¿Razonable o erróneo? ¿En este caso? ¿Y en general?
- ¿Cómo hacer para evitar repeticiones no deseadas?

Cómo evitar soluciones repetidas

Idea 1: Usando el **metapredicado** `setof` y `member`

`setof`

`setof(-Var, +Goal, -Set)`

unifica `Set` con la lista *sin repetidos* de `Var` que satisfacen `Goal`.

Cómo evitar soluciones repetidas

Idea 1: Usando el **metapredicado** `setof` y `member`

setof

`setof(-Var, +Goal, -Set)`

unifica `Set` con la lista *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:
`?- setof((X,Y), (between(2,3,X), Y is X + 2), L).`
`L = [(2, 4), (3, 5)].`

Cómo evitar soluciones repetidas

Idea 1: Usando el **metapredicado** `setof` y `member`

setof

`setof(-Var, +Goal, -Set)`

unifica `Set` con la lista *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:
`?- setof((X,Y), (between(2,3,X), Y is X + 2), L).`
`L = [(2, 4), (3, 5)].`

Ampliación del predicado `hacer`

Cómo evitar soluciones repetidas

Idea 1: Usando el **metapredicado** `setof` y `member`

setof

`setof(-Var, +Goal, -Set)`

unifica `Set` con la lista *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:
?- `setof((X,Y), (between(2,3,X), Y is X + 2), L).`
`L = [(2, 4), (3, 5)]`.

Ampliación del predicado `hacer`

```
hacerV2(M) :- setof(X, leGusta(X), L),  
              member(M, L), obligatoria(M).
```

Cómo evitar soluciones repetidas

Idea 2: Usando cláusulas excluyentes.

Algunos hechos sobre materias de cierta carrera

```
altaMateria(plp).  
altaMateria(aa).  
altaMateria(metnum).  
  
liviana(plp).  
liviana(aa).  
liviana(eci).  
  
obligatoria(plp).  
obligatoria(metnum).  
  
leGusta(M) :- altaMateria(M).  
leGusta(M) :- liviana(M), not(altaMateria(M)).  
hacer(M) :- leGusta(M), obligatoria(M).
```

Cómo evitar soluciones repetidas

Idea 2: Usando cláusulas excluyentes.

Algunos hechos sobre materias de cierta carrera

```
altaMateria(plp).  
altaMateria(aa).  
altaMateria(metnum).  
  
liviana(plp).  
liviana(aa).  
liviana(eci).  
  
obligatoria(plp).  
obligatoria(metnum).  
  
leGusta(M) :- altaMateria(M).  
leGusta(M) :- liviana(M), not(altaMateria(M)).  
hacer(M) :- leGusta(M), obligatoria(M).
```

¡Esto no funciona! ¿Por qué?

```
leGusta(M) :- altaMateria(M).  
leGusta(M) :- not(altaMateria(M)), liviana(M).
```

El metapredicado not

Definición

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

not(P) tiene éxito cuando P falla, y falla cuando P tiene éxito.

El meta predicado not

- not($p(X_1, \dots, X_n)$) tiene éxito si **no existe** instancia posible para las **variables no instanciadas** en $\{X_1 \dots X_n\}$ que haga que P tenga éxito.
- el not **no deja instanciadas** las variables libres luego de su ejecución.
- Es uno de los aspectos *extra-lógicos* del lenguaje.
No es definible en términos del lenguaje puro.
- La notación más reciente para el not es $\backslash + P$.

¿Qué quiere decir not(not($p(X_1, \dots, X_n)$)) ?

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.
```

Segunda clase práctica de Prolog

- 1 Instanciación de variables
- 2 Generate & Test
- 3 Evitando repetir resultados y Negación por falla
- 4 Generación infinita**
- 5 Resumen parte teórica

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.

Ejercicio

implementar el predicado **esTriángulo(+T)** que, dada una estructura de la forma **tri(A,B,C)**, indique si es un triángulo válido. En un triángulo válido, cada lado es menor que la suma de los otros dos, y mayor que su diferencia.

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.

Ejercicio

implementar el predicado **esTriángulo(+T)** que, dada una estructura de la forma **tri(A,B,C)**, indique si es un triángulo válido. En un triángulo válido, cada lado es menor que la suma de los otros dos, y mayor que su diferencia.

Algo más interesante

Implementar un predicado **perímetro(?T,?P)** que es verdadero cuando T es un triángulo (válido) 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).

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.

Ejercicio

implementar el predicado **esTriángulo(+T)** que, dada una estructura de la forma **tri(A,B,C)**, indique si es un triángulo válido. En un triángulo válido, cada lado es menor que la suma de los otros dos, y mayor que su diferencia.

Algo más interesante

Implementar un predicado **perímetro(?T,?P)** que es verdadero cuando T es un triángulo (válido) 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).

Último de triángulos

Implementar un generador de triángulos válidos, sin repetir resultados: **triángulo(-T)**.

Segunda clase práctica de Prolog

- 1 Instanciación de variables
- 2 Generate & Test
- 3 Evitando repetir resultados y Negación por falla
- 4 Generación infinita
- 5 Resumen parte teórica

Instanciación de variables

- Durante la ejecución de un objetivo (goal), una variable puede ser instanciada por unificación en un término concreto.
- No es posible, falla si lo intentamos, instanciar una variable una vez que ya está instanciada.
- Cada instanciación se mantiene vigente hasta tanto no se regrese (por backtracking) más allá del punto donde ocurrió.

Nomenclatura para patrones de instanciación

Por convención se aclara mediante prefijos en los comentarios (Ojo: user vs. dev):

- $p(+A)$ comunica al lector que p **necesita** que el argumento A se provea ya instanciado en algún término concreto.
- $p(-A)$ comunica al lector que p **no espera** que el argumento A esté instanciado (aunque en algunos casos podría funcionar si lo está).
- $p(?A)$ comunica al lector que p **puede** recibir el argumento A instanciado o sin instanciar (si no está instanciado, lo instancia).
- 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

- ▶ $\text{var}(A)$ tiene éxito si A **es** una variable libre.
- ▶ $\text{nonvar}(A)$ tiene éxito si A **no** es una variable libre.
- ▶ $\text{ground}(A)$ tiene éxito si A **no contiene** variables libres.

Esquema general de G&T

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 G&T

Un predicado à la G&T se define mediante otros dos:

```
predicado(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 instanciados pertenecen a la solución , pudiendo para ello asumir que ya está instanciada.

El metapredicado not

Definición

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

not(P) tiene éxito cuando P falla, y falla cuando P tiene éxito.

El meta predicado not

- $\text{not}(p(X_1, \dots, X_n))$ tiene éxito si **no existe** instancia posible para las **variables no instanciadas** en $\{X_1 \dots X_n\}$ que haga que P tenga éxito.
- el not **no deja instanciadas** las variables libres luego de su ejecución.
- Es uno de los aspectos *extra-lógicos* del lenguaje.
No es definible en términos del lenguaje puro.
- La notación más reciente para el not es $\backslash + P$.

¿Qué quiere decir $\text{not}(\text{not}(p(X_1, \dots, X_n)))$?