Programación Lógica (práctica)

Paradigmas de Lenguajes de Programación

15 de Mayo de 2018

Introducción

¿Qué es Prolog?

- Lenguaje de programación lógica
- Cómputo basado en resolución sobre cláusulas de Horn (fundamentos en la teórica).
- Gran lenguaje para hacer prototipos rápidos, solucionar pequeños problemas, demostrar teoremas, etc.

Introducción

¿Qué es Prolog?

- Lenguaje de programación lógica
- Cómputo basado en resolución sobre cláusulas de Horn (fundamentos en la teórica).
- Gran lenguaje para hacer prototipos rápidos, solucionar pequeños problemas, demostrar teoremas, etc.

Algunas características

- Declarativo (no procedural).
- Recursión (nada de for ni while).
- Relaciones (no hay funciones).
- Mecanismo de Unificación (no de nuevo decía).
- Sin tipos.
- Mundo cerrado.

SWI Prolog

SWI Prolog: la implementación recomendada por la cátedra.

- Software libre.
- Funciona en sistemas Linux, Windows y Mac.
- Motor robusto y amplia biblioteca de predicados.

```
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 6.6.4)
Copyright (c) 1990-2015 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free
software, and you are welcome to redistribute it under certain
conditions.
Please visit http://www.swi-prolog.org for details.
```

Descarga

Disponible en el sitio oficial o en su gestor de paquetes favorito (apt, homebrew, etc.)

¿Qué es un programa Prolog?

Podemos pensar los programas como bases de conocimiento que describen el dominio del problema.

- Están formados por hechos y reglas de inferencia
- Se utilizan realizando consultas sobre dicha base

¿Qué es un programa Prolog?

Podemos pensar los programas como bases de conocimiento que describen el dominio del problema.

- Están formados por hechos y reglas de inferencia
- Se utilizan realizando consultas sobre dicha base

Base de conocimiento

Lisa quiere a Nelson.
Milhouse quiere a Lisa.
Juanis quiere a Milhouse.
Utter quiere a Milhouse.
Si alguien (X) quiere a (Y) y además Y quiere a otro (Z), entonces X quiere a Z.

¿Qué es un programa Prolog?

Podemos pensar los programas como bases de conocimiento que describen el dominio del problema.

- Están formados por hechos y reglas de inferencia
- Se utilizan realizando consultas sobre dicha base

Base de conocimiento

Lisa quiere a Nelson.
Milhouse quiere a Lisa.
Juanis quiere a Milhouse.
Utter quiere a Milhouse.
Si alguien (X) quiere a (Y) y además Y quiere a otro (Z), entonces X quiere a Z.

- Luego hacemos preguntas:

Consultas

¿Nadie quiere a Milhouse?
¿Utter quiere a Nelson?
¿Quiénes quieren a Utter?

Cláusulas y consultas

Cláusulas

```
zombie(cartero).
zombie(verdulero).

tomaron_mate(cartero, bombero).
tomaron_mate(carnicero, cartero).

infectado(panadero).
infectado(X) :- zombie(X).
infectado(X) :- zombie(Y), tomaron_mate(Y,X).
```

Cláusulas y consultas

Cláusulas

```
zombie(cartero).
zombie(verdulero).

tomaron_mate(cartero, bombero).
tomaron_mate(carnicero, cartero).

infectado(panadero).
infectado(X) :- zombie(X).
infectado(X) :- zombie(Y), tomaron_mate(Y,X).
```

Realizar las siguientes consultas en SWI-Prolog

- `?- zombie(cartero).`

Cláusulas y consultas

Cláusulas

```
zombie(cartero).
zombie(verdulero).

tomaron_mate(cartero, bombero).
tomaron_mate(carnicero, cartero).

infectado(panadero).
infectado(X) :- zombie(X).
infectado(X) :- zombie(Y), tomaron_mate(Y,X).
```

Realizar las siguientes consultas en SWI-Prolog

- ```
  ?- zombie(cartero).
       true.
  ```
  ```

Cláusulas y consultas

Cláusulas

```
zombie(cartero).
zombie(verdulero).

tomaron_mate(cartero, bombero).
tomaron_mate(carnicero, cartero).

infectado(panadero).
infectado(X) :- zombie(X).
infectado(X) :- zombie(Y), tomaron_mate(Y,X).
```

Realizar las siguientes consultas en SWI-Prolog

- ```
  ?- zombie(cartero).
      true.
  ```
- ```
  ?- tomaron_mate(cartero, X).
  ```

Cláusulas y consultas

Cláusulas

```
zombie(cartero).
zombie(verdulero).

tomaron_mate(cartero, bombero).
tomaron_mate(carnicero, cartero).

infectado(panadero).
infectado(X) :- zombie(X).
infectado(X) :- zombie(Y), tomaron_mate(Y,X).
```

Realizar las siguientes consultas en SWI-Prolog

- ```
  ?- zombie(cartero).
      true.
  ```
- ```
  ?- tomaron_mate(cartero, X).
      X = bombero.
  ```

Cláusulas y consultas

Cláusulas

```
zombie(cartero).
zombie(verdulero).

tomaron_mate(cartero, bombero).
tomaron_mate(carnicero, cartero).

infectado(panadero).
infectado(X) :- zombie(X).
infectado(X) :- zombie(Y), tomaron_mate(Y,X).
```

Realizar las siguientes consultas en SWI-Prolog

- ?- zombie(cartero).
  true.
- ?- tomaron_mate(cartero, X).
  X = bombero.
- ?- infectado(I).

Cláusulas y consultas

Cláusulas

```
zombie(cartero).
zombie(verdulero).

tomaron_mate(cartero, bombero).
tomaron_mate(carnicero, cartero).

infectado(panadero).
infectado(X) :- zombie(X).
infectado(X) :- zombie(Y), tomaron_mate(Y,X).
```

Realizar las siguientes consultas en SWI-Prolog

- ```
  ?- zombie(cartero).
      true.
  ```
- ```
  ?- tomaron_mate(cartero, X).
      X = bombero.
  ```
- ```
  ?- infectado(I).
      I = panadero ;
      I = cartero;
      I = verdulero;
      I = bombero;
      false.
  ```
  ```

Sintaxis: Términos

Repaso

En primer orden teníamos:

1. *constantes* $c_0, c_1, \ldots$
2. *símbolos de función* con aridad $n > 0$ (indica el número de argumentos) $f_0, f_1, \ldots$
3. *símbolos de predicado* con aridad $n \geq 0$, $P_0, P_1, \ldots$
4. *variables* $x_0, x_1, \ldots$

Correspondencia en Prolog

1. átomos: `x`, `azul`, `cero`, `'Fellini'`, `'algún átomo'` ...
2. compounds para functores: `suc(cero)`, `pair(X,Y)`, $.(a, .(b, []))$ ...
3. compounds para predicados: `quiere(X,Y)`, `habla(iván, ruso)` ...
4. variables: `X`, `Res`, `_variable` ...
5. También hay números: `28`, `-33.2`, `-0.7E-10` ...

Evaluación en Prolog

Componentes de un programa

- Contexto: Hechos y reglas
- Estado: Objetivo

Variables (intuición)

- Las variables en prolog son sólo términos de los cuales no conocemos su valor aún.
- A pesar de no tener un valor, pueden representar relaciones entre cláusulas.
- Pueden reemplazarse por cualquier término (incluso a otra variable).
- Una vez sustituidas, no pueden cambiar su valor.

Ejecución

Así como en Haskell se intenta reducir una expresión usando la definición de un programa. En Prolog se intenta reducir la fórmula objetivo hasta la cláusula vacía usando los hechos y reglas que forman el programa.
Se puede seguir una descripción de este proceso en
`http://www.bcardiff.com/articulos/intro-prolog`.

Estructuras de datos

Los términos de átomos y de functores son usados como estructuras de datos.

- `cero`
- `suc(cero)`

A partir de las variables en reglas se pueden instanciar nuevos valores.

```
siguiente(A, B) :- B = suc(A).
```

Pero mejor aprovechar la unificación.

```
siguiente(A, suc(A)).
```

Las estructuras pueden contener variables no instanciadas.

- `cons(3, cons(2, L))`

Unificación (un repaso veloz) - Definición Formal

- Tenemos un conjunto (infinito) de variables y un conjunto de símbolos de función (las constantes las pensamos como funciones sin parámetros).
- Definimos los términos como una variable, o un símbolo de función "aplicado" a términos.
  Por ejemplo, son términos (sobre algún conjunto de variables y funciones): `x`, `f(x)`, `g(y, f(x))`, `h(x, g(y, f(x)), h(z, f(w), 0))`
- Una sustitución es una función que va de variables a términos (que pueden ser variables). Definimos la aplicación de una sustitución a un término como el reemplazo de cada una de sus variables por el término que le corresponde en la sustitución.
- Una ecuación de unificación consiste en dos términos.
- Un problema de unificación es un conjunto de ecuaciones de unificación. Resolverlo implica encontrar una **sustitución** (si existe) de manera que **al aplicarla a los términos de las ecuaciones, ambos lados queden exactamente iguales** (sintácticamente).
- Un unificador es la sustitución que resuelve un problema de unificación. En general, nos interesa el más general (*mgu*).
  `f(x)` $\doteq$ `f(y)` se puede resolver con la sustitución $\sigma = \{x \leftarrow 0,\ y \leftarrow 0\}$, pues nos queda `f(0) = f(0)`, pero claramente parecen más interesantes las soluciones de la forma $\sigma' = \{x \leftarrow y\}$.

Unificación (un repaso veloz) - Uso en prolog

- Las variables son todo aquello que comience con mayúscula o con _. Son "locales" a cada definición. Es decir, si dos variables en definiciones distintas se llaman igual, esto no quiere decir que representen el mismo valor. (Más correctamente: pueden unificar con cosas distintas.)
- Los símbolos de función son los functores y las constantes.
  Por ejemplo: `cero`, suc( ), los constructores de listas (más adelante), etc.
- Cuando se quiere resolver un predicado, se busca unificar sus argumentos con cada una de sus definiciones (en el orden en el que aparecen en el programa). Cuando la unificación tiene éxito, ésta se aplica al resto de la consulta (si existe).
  Por ejemplo: si queremos resolver `p1(X, f(Y), Y, V)` y una definición es `p1(Z, f(0), Z, U)`, se resuelve `X`$\doteq$`Z`, `f(0)`$\doteq$`f(Y)`, `Z`$\doteq$`Y`, `U`$\doteq$`V`, y su resultado es $\sigma = \{\texttt{X} \leftarrow \texttt{0},\ \texttt{Y} \leftarrow \texttt{0},\ \texttt{Z} \leftarrow \texttt{0},\ \texttt{U} \leftarrow \texttt{V}\}$
- Existen otras maneras de pedir unificación, que veremos más adelante (`=`, `is`, etc.)
- Las reglas son las mismas que vimos para inferencia:
  - los símbolos de función (antes $\cdot \to \cdot$, ahora cualquier functor que hayamos definido) y las constantes (antes `Nat`, `Bool`, ahora cualquiera) sólo unifican consigo mismas,
  - las variables unifican con cualquier término (excepto en el caso que sigue)
  - una variable no puede unificar con un término más grande en el que aparece.

Nat "desde cero"

Suponiéndolos representados por términos de la forma `cero` y `suc(X)`:

Dado el siguiente ejemplo `natural/1` que determina si un término es un natural.

natural/1

```
natural(cero).
natural(suc(N)) :- natural(N).
```

Ejercicios

1. Escribir un predicado `suma/3` que indique la suma entre 2 Nats.

Nat "desde cero"

Suponiéndolos representados por términos de la forma `cero` y `suc(X)`:

Dado el siguiente ejemplo `natural/1` que determina si un término es un natural.

natural/1

```
natural(cero).
natural(suc(N)) :- natural(N).
```

Ejercicios

1. Escribir un predicado `suma/3` que indique la suma entre 2 Nats.
2. Que ocurre si efectuamos las siguientes consultas:
   1. `suma('fruta',cero,'fruta').`
   2. `suma(cero,suc(cero),X).`
   3. `suma(cero,cero,suc(cero)).`
   4. `suma(X,Y,cero).`
   5. `suma(X,Y,Z).` (presionar ; para ver varios resultados)

Nat "desde cero"

Suponiéndolos representados por términos de la forma `cero` y `suc(X)`:

Dado el siguiente ejemplo `natural/1` que determina si un término es un natural.

natural/1

```
natural(cero).
natural(suc(N)) :- natural(N).
```

Ejercicios

1. Escribir un predicado `suma/3` que indique la suma entre 2 Nats.
2. Que ocurre si efectuamos las siguientes consultas:
   1. `suma('fruta',cero,'fruta').`
   2. `suma(cero,suc(cero),X).`
   3. `suma(cero,cero,suc(cero)).`
   4. `suma(X,Y,cero).`
   5. `suma(X,Y,Z).` (presionar ; para ver varios resultados)
3. Escribir un predicado `resta/3` que indique si el tercer argumento es la resta del primero menos el segundo.

Reversibilidad

Entrada - Salida

- Un predicado define una **relación** entre elementos: no hay parámetros de "entrada" ni de "salida".
- Conceptualmente, cualquier variable podría cumplir ambos roles dependiendo de cómo se consulte.
- Un predicado podría estar implementado asumiendo que ciertas variables ya están instanciadas, por diversas cuestiones prácticas.

Patrones de instanciación (+A, -B, ?C...)

El modo de instanciación esperado por un predicado se comunicará en los comentarios.

Para la materia utilizaremos la siguiente convención usual:

```
% pow(+B, +E, -P)    ⟵ comentario útil
 pow(...) :- ...
 pow(...) :- ...
     ...      ...
```

`+X` **debe** estar instanciada	`-X` **debe no** estar instanciada	`?X` **puede o no** estar instanciada
Casi siempre es una precondición estricta.	Suele ser precondición cautelar, pero depende.	Garantiza reversibilidad: siempre soporta `+X` y `-X`.

No se pretende más que introducir la idea general hoy. Algunos detalles se entenderán mejor más adelante.

Listas

Las listas solían estar modeladas en Prolog de la siguiente manera:

```
.(1, .(4, .(hello, .(foo, []))))
```

El functor '.' definía la relación entre un elemento y la cola de la lista.

Entonces podíamos tener predicados como:

Programa

```
% long(+L,-Res)
long([],cero).
long(.(_,XS), suc(Rec)) :- long(XS,Rec).
```

Consulta

```
?- long(.(2,[]),X).
X = suc(cero).
```

Listas

En la actualidad, Prolog nos ofrece la siguiente notación infija para trabajar con listas.

Nada de esto requiere ni introduce "tipos de datos", como vimos recién: son functores comunes.

Azúcar sintáctico standard

- `[]` (la lista vacía)
- `[ X, Y, ..., Z ]` (esos elementos en ese orden)
- `[ X, Y, ..., Z | L ]` (esos elementos en ese orden, y luego los de la lista `L`)

Algunos ejemplos

- `[1,2]` $\equiv$ `[1|[2]]` $\equiv$ `[1,2|[]]` $\not\equiv$ `[[1]|[2]]`
- `[1,2,3]` $\equiv$ `[1|[2,3]]` $\equiv$ `[1,2|[3]]` $\equiv$ `[1,2,3|[]]`
- `[28,[],37,42,paradigmas,[1,2],107,160]`

Primeros ejercicios con listas

Ejercicios sobre listas

1. Definir `long(+L, -N)` que relacione una lista con su longitud utilizando la notación presentada.
2. Reescribir `long(+L, -N)` para que ahora "devuelva" un valor numérico.
3. Definir el predicado `sinRepetidosConsecutivos(+L1, -L2)` que dada una lista elimina elementos repetidos consecutivos. Por ejemplo:

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

Ejercicios con append

Dado el predicado `append(?L1,?L2,?L3)`

Append/3

```
append([],L2,L2).
append([X|L1],L2,[X|L3]):-append(L1,L2,L3).
```

Ejercicios

Implementar los siguientes predicados:

1. `prefijo(+Lista,?Pref)` que tiene éxito si Pref es un prefijo de Lista.
2. `sufijo(+Lista,?Sufi)`
3. `sublista(+Lista,?Subl)`
4. `insertar(?X,+L,?LconX)`, que tiene éxito si `LconX` puede obtenerse insertando a `X` en alguna posición dentro de `L`.
5. `permutación(+L,?P)` que tiene éxito si la lista `P` es una permutación de `L`.

Member

El predicado `member(?X,+XS)` se implementa de la siguiente manera:

member/3

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

Ejercicio:

Realizar un seguimiento (árbol de ejecución) de la siguientes consultas:

```
?- member(2, [1,2,3]).
?- member(X, [1,2,3]).
?- member(5, [1,X,X,3]).
?- length(L, 2), member(5, L), member(2, L).
```

Aspectos extra-lógicos: aritmética

El motor de operaciones aritméticas es independiente del motor lógico.

Permite trabajar con dominios conocidos como los naturales, racionales, etc.

Expresiones aritméticas

- Un literal: un número natural, racional, etc.
- Una variable ya instanciada en una expresión aritmética.
- Una expresión $\texttt{E}_1 \texttt{+} \texttt{E}_2 \mid \texttt{E}_1 \texttt{-} \texttt{E}_2 \mid \texttt{E}_1 \texttt{*} \texttt{E}_2 \mid \texttt{E}_1 \texttt{/} \texttt{E}_2$ con subexpresiones aritméticas.

Algunos operadores

- `X = Y` tiene éxito sii `X` unifica con `Y` (no es extra-lógico). Negación: $\backslash =$.
- `X is E` tiene éxito sii `X` unifica con **el resultado de evaluar** `E`.
- $\texttt{E}_1 \texttt{<} \texttt{E}_2$, $\texttt{E}_1 \texttt{=:=} \texttt{E}_2$, $\texttt{E}_1 = \backslash = \texttt{E}_2$ et al: **ambos lados** son evaluados.

Importante: todas las $\texttt{E}_i$ deben ser expresiones aritméticas.

Entre

Considerar el siguiente predicado:

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

Entre

Considerar el siguiente predicado:

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

Ejercicio

Dar un predicado `entre(+X, +Y, -Z)` que se satisfaga cuando el número `Z` esté comprendido entre los números `X` e `Y` (inclusive).

Notar que lo que se nos pide un predicado capaz de instanciar sucesivamente `Z` en cada natural entre `X` e `Y`.

Este predicado existe en Prolog:
`?- help(between).`

Entre

Considerar el siguiente predicado:

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

Ejercicio

Dar un predicado `entre(+X, +Y, -Z)` que se satisfaga cuando el número `Z` esté comprendido entre los números `X` e `Y` (inclusive).

Notar que lo que se nos pide un predicado capaz de instanciar sucesivamente `Z` en cada natural entre `X` e `Y`.

Este predicado existe en Prolog:
`?- help(between).`

- Si implementamos entre/3 con desde/2, ¿Es correcto? ¿O se cuelga?

Entre

Considerar el siguiente predicado:

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

Ejercicio

Dar un predicado `entre(+X, +Y, -Z)` que se satisfaga cuando el número `Z` esté comprendido entre los números `X` e `Y` (inclusive).

Notar que lo que se nos pide un predicado capaz de instanciar sucesivamente `Z` en cada natural entre `X` e `Y`.

Este predicado existe en Prolog:
`?- help(between).`

- Si implementamos entre/3 con desde/2, ¿Es correcto? ¿O se cuelga?
- Importante: comprender qué entendemos en este contexto por *colgarse*, por qué podría suceder eso y cómo evitarlo.

La clase que viene. . .

- Ejercicios más complicados y cómo encararlos.
- Técnicas útiles para pensar soluciones (e.g., Generate & Test).
- Cómo evitar generar soluciones repetidas.
- Más detalles sobre cómo y por qué funciona el motor.
- Más sobre aspectos extra-lógicos (e.g., `! (cut)`).