Clase Práctica: Programación Lógica I

Paradigmas de Lenguajes de Programación

28 de Febrero 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.

Algunas características

- Declarativo (no procedural).
- Recursión (nada de for ni while).
- Relaciones (no hay funciones).
- Mecanismo de Unificación
- 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

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?

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))`

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

Casi siempre es una precondición estricta.

-X **debe no** estar instanciada

Suele ser precondición cautelar, pero depende.

?X **puede o no** estar instanciada

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 están modeladas en Prolog de la siguiente manera:

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

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

Entonces podríamos tener predicados como:

Programa

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

Consulta

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

Listas

Prolog también nos ofrece notación abreviada 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. 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 `E1 + E2` | `E1 - E2` | `E1 * E2` | `E1 / E2` 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`.
- `E1 < E2`, `E1 =:= E2`, `E1 =\= E2` et al: **ambos lados** son evaluados.

Importante: todas las E_i deben ser expresiones aritméticas.

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)`).