Ayudantía 12:
Programación Lógica: Prolog

Profesores: José Luis Martí Lara, Roberto Díaz Urra
Ayudantes:
Hugo Sepúlveda Arriaza
Gabriela Acuña Benito
Lucio Fondón Rebolledo
lucio.fondon@sansano.usm.cl

Universidad Técnica Federico Santa María
Departamento de Informática

Programación Lógica

Definición del Paradigma

- Pardigma diferente a los imperativos, ya que no se basa en la máquina de Von Neumann, sino que se basa en *lógica simbólica* y *programación lógica*
- En este paradigma no nos interesa *cómo* debe hacerse un algoritmo, sino *qué* debe lograrse
- La **lógica** corresponde a la ciencia del razonamiento y la prueba o demostración de verdades (en Prolog, esto se realizará mediante **hechos, reglas y consultas**)
- **Aplicaciones**:
  - Pruebas matemáticas
  - Inteligencia Artificial
  - Consultas en bases de datos

Conceptos Básicos

- **Tipos de Datos y Sintaxis**:
  - Números: Enteros y reales
  - Booleanos: `true`, `false`
  - Átomos: Constantes denotadas entre minúsculas o entre comillas simples (' ')
  - Variables: Representan básicamente lo que sea, y se denotan con el primera carácter en mayúscula o con _ *(variables anónimas)*
- En Prolog se pueden realizar comentarios con el símbolo %
- Un programa básico de Prolog consta generalmente de tres elementos clave:
  - **Hechos**
  - **Reglas**
  - **Consultas o queries**

Hechos

Por lo general, dentro de un programa de Prolog, utilizaremos con frecuencia los conceptos de **hechos** y **reglas**:

Hechos

- Los **hechos** en Prolog denotan una sentencia que siempre es verdadera
- Todos los hechos deben terminar con un punto (.)

```
% Se muestran hechos definidas por el programador
% Nos servirán para crear reglas y realizar consultas
padre(juan, maria). % Se lee "Juan es padre de María"
padre(sebastian, gabriel).
madre(camila, maria). % Se lee "Camila es madre de María"
madre(alejandra, camila).
```

Reglas

Reglas

- Una **regla** en Prolog es básicamente una sentencia condicional
- Es el análogo a la **implicación lógica**, es decir, la relación $a \implies b$ se expresa en Prolog como `b :- a`
- Se compone de una **cabeza** (parte izquierda de `:-`), que es la conclusión de la proposición definida en el cuerpo (parte derecha de `:-`), llamada también **cola**

```
% Utilizando los hechos, podemos definir reglas
% La coma (,) denota un AND lógico
tienenHijo(X,Y) :- padre(X,Z), madre(Y,Z).
% Regla que nos dice si un hombre y mujer tienen
% al menos un hijo en común
```

- Lo anterior, escrito en notación lógica, se escribe:
  $\text{padre}(X, Z) \land \text{madre}(Y, Z) \implies \text{tienenHijo}(X, Y)$

Consultas y Calce

Una de las funcionalidades más características de Prolog es el término del **calce**, que se basa en básicamente encontrar todas las instancias en donde la relación que se está consultando sea verdadera

```
?- fecha(7, julio, 2000, (20, Min))
= fecha(Dia, julio, Año, (20, 50)).
> Año = 2000,
> Dia = 7,
> Min = 50
```

Consultas y Calce

Una de las funcionalidades más características de Prolog es el término del **calce**, que se basa en básicamente encontrar todas las instancias en donde la relación que se está consultando sea verdadera

```
?- fecha(7, julio, 2000, (20, Min))
= fecha(Dia, julio, Año, (20, 50)).
> Año = 2000,
> Dia = 7,
> Min = 50
```

```
?- padre(X,Y). % consultamos el hecho con las variables X e Y
> X = juan, % 1era respuesta
> Y = maria
> X = sebastian, % 2da respuesta
> Y = gabriel
```

Listas

Listas

Prolog tiene a su disposición el uso de **listas**, y provee una simple y conveniente forma de estructurarlas

- **Sintaxis**: Similar a Python

```
l([a,c,b,d]).
```

- **Estructura**: Similar a Scheme

```
?- l([X|Y]). % X -> (car l), Y -> (cdr l)
```

```
> X = a,
> Y = [b,c,d]
```

- Prolog nos provee de muchas operaciones sobre listas:

```
append(L1, L2, L3) % concatena L1 y L2, se guarda en L3
member(X, L) % ve si el elemento X pertenece a L
delete(L, X, L1) % elimina un elemento X de la lista L, retorna L1
permutation(L, L1) % realiza todas las permutaciones posibles de L1
```

Recursión, Backtracking y Cut

En Prolog, no existe la iteración, por lo que debemos utilizar la técnica de **recursión** para poder iterar sobre los objetos

Para solucionar un problema, Prolog siempre busca todos los posibles valores (*backtracking*). A veces, esto no es eficiente y es mejor forzar el programa a que termine en un momento dado. Para esto, se puede aplicar el operador **CUT** (`!`)

```
list_member(X, [X|L]). % caso base
list_member(X, [Y|L]) :- list_member(X, L).
```

Recursión, Backtracking y Cut

En Prolog, no existe la iteración, por lo que debemos utilizar la técnica de **recursión** para poder iterar sobre los objetos

Para solucionar un problema, Prolog siempre busca todos los posibles valores (*backtracking*). A veces, esto no es eficiente y es mejor forzar el programa a que termine en un momento dado. Para esto, se puede aplicar el operador **CUT** (!)

```
list_member(X, [X|L]). % caso base
list_member(X, [Y|L]) :- list_member(X, L).
```

Con CUT:

```
list_member(X, [X|L]) :- !. %caso base con cut
list_member(X, [Y|L]) :- list_member(X, L).
```

Ejercicios

Ejercicio 1

En relación a Prolog, es cierto que:

I. Al hacer uso del operador CUT puede aumentar la eficiencia
II. No hay posibilidad de evitar el backtracking en Prolog
III. El backtracking se aplica cuando falla la satisfacción de una cláusula.
IV. Se entiende por instanciación como la asignación temporal de variables

a) I y II
b) I, III y IV
c) II, III y IV
d) II y III

Ejercicio 1

En relación a Prolog, es cierto que:

I. Al hacer uso del operador CUT puede aumentar la eficiencia
II. No hay posibilidad de evitar el backtracking en Prolog
III. El backtracking se aplica cuando falla la satisfacción de una cláusula.
IV. Se entiende por instanciación como la asignación temporal de variables

a) I y II
b) I, III y IV
c) II, III y IV
d) II y III

R: **Alternativa b**

Ejercicio 2

Construya el predicado `duplicar(A,L)`, el cual recibe una lista de elementos y retorna otra con los elementos pero duplicados. Ej.:

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

Solución

```
duplicate([],[]). % caso base
duplicate([H|T], L) :-
    duplicate(T, L1),
    append([H,H], L1, L).
```

Ejercicio 3 (C2 2017-1)

Construya el predicado `miReverse(A,L)`, el cual recibe una lista de elementos y retorna otra con los mismos en orden inverso. Ej.:

```
?- miReverse([a, b, c, d, e], L).
L = [e, d, c, b, a]
```

Además, utilizando solo `miReverse(A,L)`, defina el predicado `palindromo(L)`, que determina si una lista es palíndromo o no. Ej:

```
?- palindromo([a, b, c, b, a]).
true
?- palindromo([a, b, c]).
false
```

Solución

```
miReverse([],[]). % caso base
miReverse([H|T],L) :-
    miReverse(T,R),
    append(R, [H], L).

palindromo(L1) :- miReverse(L1,L1).
```