Programación Declarativa:
PROLOG

“Aspectos Avanzados de PROLOG”

M.Sc. Dafne García de Armas

Características de Prolog

Prolog
Lenguaje Relacional

→

Basado en el concepto de relación o predicado de la Lógica de Primer Orden

los argumentos de los predicados pueden ser ocupados por

términos descriptivos

predicado(↓)

construidos a partir de símbolos de funciones n-arias que sirven para definir un valor en Prolog

Clasificación y Construcción de Términos

TÉRMINO

Variable

Valor que aun no ha sido unificado

Ejemplo: **X**

Constante Numérica

Para representar números enteros o reales

Ejemplo: **-3 4.5**

Átomo

Constante textual utilizada para dar nombre a los términos compuestos o para representar una constante o un texto

Ejemplo: **ana**

Termino Compuesto

Consiste en un nombre seguido por *n argumentos*, cada uno de los cuales es un término.

```
persona('Ana', 28, 'doctora')
```

Clasificación y Construcción de Términos

- Prolog es un lenguaje sin tipos (todos los datos pertenecen a un mismo dominio sintáctico, el universo de Herbrand)
- Pero es posible reconocer la categoría a la que pertenece un cierto dato (valor) analizando su forma sintáctica.
- Prolog cuenta con una colección de predicados que permiten tanto clasificar como crear términos.

Clasificación de Términos

`var(Term)` : tiene éxito si Term es una variable no unficada (libre).

`nonvar(Term)` : tiene éxito si Term es una variable unificada (ligada).

`atom(Term)` : tiene éxito si Term se ha unificado a una constante (átomo).

`integer(Term)` : tiene éxito si Term se ha unificado a un número entero.

`float(Term)` : tiene éxito si Term se ha unificado a un número real.

`number(Term)` : tiene éxito si Term se ha unificado a un número entero o real.

`atomic(Term)` : tiene éxito si Term se ha unificado con un objeto simple (atómico), i.e. un símbolo constante, un número o una cadena.

Clasificación de Términos

```
?- atom(vacio).
true

?- integer(-3).
true

?- var(X).
true

?- integer([1]).
false
```

```
?- X = 2, var(X).
false

?- X = 2, integer(X).
true

?- var(X), X = 2.
X = 2

?- var([X]).
false
```

```
?- nonvar(X).
false

?- X = a, nonvar(X).
true

?- nonvar(vacio).
false
```

Clasificación de Términos

`compound(Term)`: tiene éxito si Term se ha unificado a un término compuesto.

```
?- compound([a,b|Xs]).
true
?- compound([X]).
true
```

```
?- compound(-3.14).
false
```

`ground(Term)`: tiene éxito si Term no contiene variables sin unificar, o sea, variables libres.

`callable(Term)`: tiene éxito si Term se ha unificado con un átomo o un término compuesto, luego puede ser tomado como argumento por otros predicados.

Clasificación de Términos

Pueden emplearse para:

- comprobar **precondiciones**
- diagnosticar **errores de tipo**
- extender **usos posibles**
- mejorar la **eficiencia**
- distinguir **casos** según el tipo

Consideremos el predicado `suma/3`:

```
suma(X,Y,Z) :- Z is X + Y.

?- suma(2,a,Z).
ERROR: is/2: Arithmetic: `a/0' is not a
function ^ Exception: (8) _G256 is 2+a ?
```

```
suma_precond(X,Y,Z) :-
    number(X), % precondición
    number(Y), % precondición
    Z is X + Y.

?- suma_pre(2,a,Z).
false
```

Clasificación y Construcción de Términos

Metapredicados de inspección de estructuras permiten:

- Descomponer una estructura en sus componentes
- Componer una estructura a partir de sus componentes

Prolog predefine tres metapredicados de inspección:

- `functor/3`
- `arg/3`
- `=../2`

Clasificación y Construcción de Términos

`functor(T,F,A)` : tiene éxito si T es un término con functor principal F y aridad A.

Soporta los usos (+,+,+), (+,-,-) y (-,+,+)
Los usos (+,+,-) y (+,-,+) son posibles, pero poco útiles

```
?- functor(T,F,3).
ERROR: Arguments are not sufficiently instantiated
```

El predicado **`functor/3`** fallará si T y (F o A) no están unificadas.
(-,+,-) y (-,-,+)

Clasificación y Construcción de Términos

En el uso (+,+,+) se comporta como un test:

```
 ?- functor(t(X,a),t,2).
true

?- functor(2+3*5-1, '-',2).
true

?- functor(a,a,0).
true

?- functor([x,y], '.',2).
true
```

En el uso (+,-,-) se comporta como un generador único. Se utiliza para obtener el functor principal de un término :

```
?- functor(punto(a,X),F,N).
F = punto,
N = 2.

?- functor([A,f(X),Y],F,N).
F = '.',
N = 2.
```

Clasificación y Construcción de Términos

En el uso (-,+,+) se comporta como un generador único. Se utiliza para generar una plantilla de estructura.

```
?- functor(T,punto,2).
T = punto(_G8,_G9)

?- functor(T,'+',2).
T = _G8 + _G9

?- functor(T,'+',4).
T = '+'(_G8, _G9, _G10, _G11)

?- functor(T,a,0).
T = a
```

```
?- functor(padre(jose,luis),F,A).
F = padre,
A = 2.

?- functor(ana,F,A).
F = ana,
A = 0.

?- functor(T,padre,3).
T = padre(_G431,_G432,_G433)

?- functor(T,F,2).
ERROR: functor/3: Arguments are
not sufficiently instantiated
```

Clasificación y Construcción de Términos

`arg(N,T,V)`: tiene éxito si el valor V unifica con el N-ésimo argumento del término T.

Los argumentos de numeran a partir de 1, de izquierda a derecha

Soporta los usos (+,+,+), (+,+,-) y (+,-,+)

Si T no es una variable pero N y V si lo son, entonces V unifica con cada uno de los argumentos de T en resatisfacción.

```
?- arg(3,T,A).
ERROR: Arguments are not sufficiently instantiated
```

Clasificación y Construcción de Términos

En el uso $(+,+,-)$ se comporta como un generador único. Se utiliza para obtener el argumento i –ésimo de una estructura:

```
?- arg(3,arco(i,q2,q0),A).
 A = q0

?- arg(1,[a,b,c,d],A).
 A = a

?- arg(2,[a,b,c,d],A).
A = [b, c, d]

?- arg(3,[a,b,c,d],A).
false
```

En el uso (+,-,+) se comporta como un generador único Se utiliza para instanciar el argumento i-ésimo de una estructura.

```
?- arg(2,arco(i,Q,q0),q5).
Q = q5

?- arg(1,[X,b,c,d],a).
X = a
```

Clasificación y Construcción de Términos

```
?- arg(1,padre(luis,jose),luis).
true.

?- arg(1,padre(luis,jose),V).
V = luis

?- arg(2,padre(luis,X),jose).
X = jose
```

```
?- arg(A,padre(luis,jose),V).
A = 1
V = luis ;
A = 2
V = jose
```

Clasificación y Construcción de Términos

`T =.. L`: tiene éxito si L unifica con la lista [X|Y], donde X es el functor principal del término T y el resto Y contiene los argumentos de T.

Tanto T como L pueden ser variables, pero no al mismo tiempo. (Soporta los usos (+,+), (-,+) y (+,-))

En el uso (+,+) se comporta como un test:

```
?- f(a, X, g(b,Y)) =.. [f, a, X , g(b,Y) ].
true

?- [a,b,c] =.. [‘.’, a, [b,c]].
true
```

Clasificación y Construcción de Términos

En el uso (+,-) se comporta como un generador único. Se utiliza para descomponer un término en sus componentes

```
?- punto(2,3) =.. Xs.
Xs = [punto, 2, 3]

?- [A,f(X),Y] =.. Xs.
Xs = ['.', A, [f(X), Y]]

?- sin(X)*cos(X) + 3.14 =.. Xs.
Xs = [+, sin(X)*cos(X), 3.14]

?- 6 =.. Xs.
Xs = [6]

?- [] =.. Xs.
Xs = [[]]
```

En el uso (-,+) se comporta como un generador único. Se utiliza para componer un término a partir de sus componentes

```
?- T =.. ['+',a+b+c,d].
T = a+b+c+d

?- T =.. [arco,ej1,i,q3,q5].
T = arco(ej1, i, q3, q5)

?- T =.. ['.', p,[a,k]].
T = [p, a, k]

?- T =.. [fin].
T = fin
```

Clasificación y Construcción de Términos

```
?- padre(luis,jose) =.. L.
L = [padre,luis,jose].

?- T =.. [member,X,[1,2,3]].
T = member(X,[1,2,3]).

?- madre(alicia, X) =..[G,Y,dario].
X = dario,
G = madre,
Y = alicia.
```

Clasificación y Construcción de Términos

Ventaja: Facilidades para la manipulación simbólica

Ejemplo 1: Definición del predicado **`cumplen_todos(L,C)`** que, dada una lista L y un predicado unario C, determina si cada elemento de L cumple la condición C.

```
cumplen_todos([],_).
cumplen_todos([X|Y],C):- T=..[C,X], T,
                         cumplen_todos(Y,C).
```

Clasificación y Construcción de Términos

```
cumplen_todos([],_).
cumplen_todos([X|Y],C):- T=..[C,X], T, cumplen_todos(Y,C).
```

Puede utilizarse para comprobar si una lista contiene solo variables, átomos, números, etc.

```
?- cumplen_todos([X,Y],var).
true

?- cumplen_todos([madre,padre],atom).
true

?- cumplen_todos([1,2,3,4.5],number).
true

?- cumplen_todos([1,2,3,4.5],integer).
false
```

Sistemas de Base de Datos

- Una relación en un sistema de base de datos relacional se define extensionalmente mediante el conjunto de tuplas que la constituyen.
- En Prolog esta misma relación queda definida mediante un conjunto de hechos.
- Además, Prolog permite definir intencionalmente relaciones entre entidades dando definiciones de las mismas.
- Ejemplo: En el programa *Familia*
  - El predicado *padre/2* es una relación definida extensionalmente mediante un conjunto de hechos.
  - Mientras que el predicado *abuelo/2* es una relación definida intencionalmente mediante dos cláusulas.

Sistemas de Base de Datos

Un programa lógico puede ser considerado una base de datos relacional muy especial.

En la terminologa de Prolog, **se denomina Base de Datos** (BD):

*Conjunto de **hechos** y **reglas** que han sido cargados en el espacio de trabajo del sistema Prolog.*

Gestión de la BD Interna de Prolog

- Para que un programa Prolog pueda ejecutarse es preciso que los hechos y las reglas que lo componen se hayan cargado en el espacio de trabajo del sistema Prolog.
- Las reglas que están almacenadas en un fichero y todavía no han sido cargadas en el espacio de trabajo no pueden intervenir en la ejecución del programa.
- Un sistema Prolog solo reconoce aquello que esta en su espacio de trabajo.
- Los efectos y las acciones que se realicen durante la ejecución de un programa y un objetivo dependen de lo que allí se encuentre almacenado.
- Existen predicados que permiten gestionar la base de datos interna del sistema Prolog y, por lo tanto, modificar el comportamiento de un programa

Gestión de la BD Interna de Prolog

Carga de programas

`consult(Fich)`: añade a la base de datos interna de Prolog todas las clausulas que contenga el fichero Fich.

Esto también puede lograrse escribiendo el nombre del fichero entre corchetes, o sea, **[Fich]**.

Si se desea añadir a la base de datos las cláusulas de varios ficheros es posible hacer de una vez **[Fich1,Fich2,...]**.

Gestión de la BD Interna de Prolog

Consultas

`listing(Nom)` : imprime todas las cláusulas de la base de datos interna de Prolog cuyo símbolo de predicado en la cabeza coincida con **`Nom`**.

`listing`: imprime todas las cláusulas de la BD interna de Prolog.

Gestión de la BD Interna de Prolog

Adición y eliminación

Estos predicados permiten modificar dinámicamente la definición de un predicado en la base de datos interna de Prolog:

`asserta(C)`: añade la cláusula C como el primer hecho o regla de la definición del predicado correspondiente.
`assertz(C)`: añade la cláusula C como el último hecho o regla de la definición del predicado correspondiente.
`assert(C)`: similar a **`assertz`**.

`retract(C)`: elimina la cláusula C de la base de datos.
`retractall(Head)`: elimina toda clausula de la base de datos cuya cabeza unifique con **`Head`**.

Gestión de la BD Interna de Prolog

Adición y eliminación

Para poder modificar dinámicamente la definición de un predicado se requiere de que este se haya declarado de tipo dinámico mediante la directiva:

```
:- dynamic Nombre_Predicado/Aridad.
```

Informa al intérprete que la definición del predicado puede cambiar durante la ejecución mediante el uso de los predicados de adición y eliminación anteriores.

Ejemplo: `:-dynamic madre/2, padre/2, abuelo/2.`

Base de Datos Deductivas

- Una cláusula definida que es un objetivo representa una solicitud de información a la base de datos (de conocimiento) de Prolog.

- Como el intérprete es el que dará respuesta a esa consulta y es un demostrador lógico, un programa Prolog puede considerarse como un **sistema de base de datos deductivo.**

- Este tipo de sistema permite hacer deducciones a través de inferencias basado en reglas y hechos que son almacenados en la base de datos.

- Surge debido a las limitaciones de las BD relacionales al tratar de responder a consultas recursivas y deducir relaciones indirectas de los datos que almacena.

Metaprogramación

- Es la escritura de programas denominados metaprogramas que escriben o manipulan como datos otros programas o a sí mismos.

- Ejemplos: analizadores sintácticos que analizan la corrección sintáctica de programas escritos en determinado lenguaje de programación, los analizadores lexicales, intérpretes y compiladores que actúan a nivel sintáctico y/o semántico sobre otros programas.

- Un metaprograma puede ser escrito en cualquier lenguaje de programación, pero no en todos se ofrecen facilidades para ello.

- El lenguaje de los metaprogramas y los lenguajes objetos que aquellos manipulan no tiene que ser necesariamente el mismo.

Metaprogramación

- Por lo que, uno de los objetivos más relevantes de la metaprogramación es la manipulación semántica de programas, en este caso la identidad de ambos resulta un factor conveniente

- Esta identidad se logra en los lenguajes de programcaión en los que no hay diferencia sintáctica entre programa y dato.

- Visto que en Prolog solo existe una estructura sintáctica para representar tanto a las definiciones de un programa como las estructuras que representan los datos **(la cláusla),**

Prolog cumple con la identidad y es muy adecuado para escribir metaprogramas en él.

Metavariables

- Son una facilidad básica de la metaprogramación.
- Son variables que pueden tomar como valor un programa.
- En Prolog serían variables donde dicho valor es un término el cual el intérprete evalúa como un objetivo.
- Una variable es detectada como metavariable cuando ocurre como objetivo en el cuerpo de una cláusula:

```
not(X):- X, !, fail.
not(X).
```

Predicados predefinidos de metaprogramación

- Entre los predicados predefinidos en Prolog considerados como de metaprogramación tenemos a:

  `arg/3, functor/3, =../2, assert/1, retract/1, call/1` y **`clause/2.`**

- Permiten definir predicados donde alguno de los argumentos no es un término del dominio, sino que puede ser una función o predicado.

Predicados predefinidos de metaprogramación

- **`call(Obj)`**: invoca el objetivo **Obj** y tiene éxito si **Obj** lo tiene.

- Se utiliza para lanzar objetivos dentro de una regla o de un programa que solo se conocen en tiempo de ejecución.

- Si **Obj** es una variable deberá estar unificada en el momento en que vaya a ejecutarse.

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

Predicados predefinidos de metaprogramación

- **`clause(Cabeza, Cuerpo)`**: tiene éxito si *Cabeza* puede unificar con la cabeza de alguna cláusula de la base de datos interna de Prolog y Cuerpo con el cuerpo de dicha cláusula.

- Si hay varias cláusulas con las que *Cabeza* puede unificar, unifica con la primera y luego por resatisfacción unifica con las siguientes.

- Falla si no hay cláusulas con las que unifique.

- La variable Cabeza tiene que estar instanciada.

- Cuando las cláusulas encontradas son hechos, Cuerpo unifica con true

```
?- clause(padre(luis,jose),B).
B = true;
false
```

Aplicaciones de la metaprogramación

- La construcción de analizadores sintácticos y semánticos de los programas escritos en un lenguaje de programación dado.

- La construcción de metaintérpretes: pueden extender y/o modificar la capacidad básica de un intérprete es decir, su capacidad de representación y de proceso para solucionar problemas y además su estrategia de navegación en el espacio de búsqueda.

- Como en tiempo de ejecución es posible para Prolog tener acceso al código de un programa es fácil escribir un metaintérprete de Prolog en Prolog mismo.

Predicados findall, setof y bagof

- El mecanismo de control de Prolog produce respuestas una a una.
- Al encontrarse un respuesta, el intérprete la muestra y se detiene en espera de la indicación del usuario.
- Si el usuario desea más respuestas (;), el intérprete de Prolog intenta encontrarlas ayudado de la técnica de retroceso.
- No obstante puede ser de interés obtener todas las respuestas a un objetivo.
- Prolog dispone de una serie de predicados predefinidos que permiten acumular en una lista todas las respuestas del objetivo planteado, sin que se detenga el proceso cada vez que se encuentra una rama triunfo, sino que recorre todo el árbol de derivación del objetivo y guarda todas sus src en una lista.

findall/3

- **`findall(F,O,L)`**: crea la lista **L** con todas las unificaciones de las variables que aparecen en **F** correspondientes a todas las respuestas, obtenidas por retroceso cronológico, para el objetivo **O**.

- Si el árbol de derivación para **O** no tiene ninguna rama triunfo (no tiene solución), la lista **L** unifica con la lista vacía.

findall/3

```
?- findall(X, permutacion([1,2,3],X), P).
P = [[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]].

?- findall(X,(member(X,[1,2,3,4,5,6]),impar(X)), L).
L = [1,3,5].
```

findall/3

```
(P1) padre(luis,alicia).
(P2) padre(luis,jose).
(P3) padre(jose,ana).

(M1) madre(alicia,dario).

(A1) abuelo(X,Y):-padre(X,Z),madre(Z,Y).
(A2) abuelo(X,Y):-padre(X,Z),padre(Z,Y).

?- findall((X:Y), padre(X,Y), P).
P = [luis:jose, luis:alicia, jose:ana].

?- findall(X, padre(X,Y), P).
P = [luis, luis, jose].
```

bagof/3

- **`bagof(F,O,L)`**: crea la lista **L** con todas las unificaciones de las variables que aparecen en **F** correspondientes a todas las respuestas, obtenidas por retroceso cronológico, del objetivo **O**.

- Si **O** tiene variables libres, además de las que aparecen en **F**, **bagof** retornará una lista de respuestas obtenidas para cada una de las posibles unificaciones de las variables libres.

- Si **O** no tiene solución bagof falla.

bagof/3

```
(P1) padre(luis,alicia).
(P2) padre(luis,jose).
(P3) padre(jose,ana).
(M1) madre(alicia,dario).
(A1) abuelo(X,Y):-padre(X,Z),madre(Z,Y).
(A2) abuelo(X,Y):-padre(X,Z),padre(Z,Y).

?- bagof(X, padre(X,Y), P).
Y = alicia,
P = [luis] ;
Y = ana,
P = [jose] ;
Y = jose,
P = [luis].

?- bagof(X, (member(X,[2,4,6]),impar(X)), I).
false.
```

bagof/3

- Si no se desea iterar por todas las posibles unificaciones de las variables libres que no aparecen en **F**, estas pueden ligarse al objetivo con la contrucción: **`Var^Obj`**.

```
?- bagof(X, Y^padre(X,Y), P).
P = [luis, luis, jose].
```

- Es la misma respuesta que se obtiene haciendo uso del **`findall`**.

setof/3

`setof(F,O,L)`: equivalente a **`bagof/3`** pero la lista **`L`** se devuelve ordenada y sin duplicados.

```
?- setof(X, Y^padre(X,Y), P).
P = [jose, luis].
```

Nota: Si las variables ocurren libres, no ordena alfabéticamente ni elimina los repetidos, funciona igual que el bagof

Tarea

1. Definir el predicado **`arg/3`**.

2. Definir el predicado **`functor/3`**.

Hint: Emplear el predicado **`T =.. L`**

Programación Declarativa:
PROLOG

“Aspectos Avanzados de PROLOG”

M.Sc. Dafne García de Armas