Lógica Computacional 2022-1, nota de clase 13
Breve introducción a Prolog

Favio Ezequiel Miranda Perea Araceli Liliana Reyes Cabello
Lourdes Del Carmen González Huesca Pilar Selene Linares Arévalo

17 de diciembre de 2021

En esta nota damos una muy breve introducción a la programación y apuntamos algunos conceptos de la teoría de la programación lógica en Prolog

1. Resolución en Prolog

Iniciamos la sección revisando la notación de cláusulas que Prolog utiliza. Recordemos que sólo se permiten cláusulas definidas o de Horn en donde $\leftarrow$ se sustituye por `:-` y cada cláusula termina en punto. Éstas son de alguna de las siguientes formas:

- Reglas: una literal positiva y al menos una literal negativa:

$$P \text{ :- } Q_1, \ldots, Q_m.$$

- Hechos: una literal positiva y ninguna literal negativa:

$$P.$$

- Metas: ninguna literal positiva y al menos una literal negativa:

$$\text{?- } Q_1, \ldots, Q_m$$

Es importante observar que las metas son el medio para interactuar con un programa, el cual consta únicamente de hechos y reglas. Más aún el símbolo ?- es el prompt de Prolog .

Prolog usa una versión especial de la regla de resolución binaria, esta versión específicamente se adapta a programas con cláusulas de Horn. En un cómputo en Prolog tenemos un programa lógico $\mathbb{P}$ y una cláusula meta $\mathcal{C}$ que expresa el problema que queremos resolver o más bien la información que queremos confirmar a partir del programa. En la versión de resolución implementada en Prolog , una de las dos cláusulas que se resuelven debe ser siempre la cláusula meta, y el resolvente siempre se convierte en la nueva cláusula meta, esto se llama resolución lineal. Detalladamente se procede como sigue:

1. Se tiene dada una cláusula meta ?- $G_1, \ldots, G_k$.
2. Buscar en el programa una cláusula o una variante de una cláusula P :- $Q_1, \ldots, Q_n$ tal que
   - G_1 se unifica con P.
   - μ es el unificador más general de $\{G_1, P\}$

si no hay tal cláusula terminar con falla.

3. Reemplazar G_1 con $Q_1, \ldots, Q_n$.
4. Aplicar μ al resultado, obteniendo

$$?\text{- } Q_1\mu, \ldots, Q_n\mu, G_2\mu, \ldots, G_k\mu.$$

5. Si el resultado es la cláusula vacía $\square$ (que en PROLOG se ve como ?-) entonces terminar, reportando éxito, y devolver como solución la composición de todos los unificadores μ aplicados a las variables de la meta original.

Primero, se toma la cláusula meta y se elige una de sus literales. En principio podemos seleccionar cualquier literal, pero PROLOG siempre elige la literal más a la izquierda de la cláusula meta. Este proceso se conoce como **resolución con función de selección**. La función de selección de PROLOG , al aplicarse a una sucesión de expresiones, siempre devuelve la que está más a la izquierda, es decir, la primera. Más aún, para seguir resolviendo siempre se utiliza la meta actual obtenida mediante el proceso anterior, es decir, está prohibido hacer un paso de resolución sin involucrar a la meta actual, esta restricción se conoce como **resolución lineal**. De manera que la combinación se conoce como *resolución lineal con función de selección* o SLD-resolución. Por lo tanto, dado que sólo se permiten cláusulas definidas, el método particular de resolución implementado en PROLOG se conoce también como **SLD-resolución** (en inglés Selected, Linear, Definite resolution).
El segundo paso, después de seleccionar una literal de la cláusula meta, es buscar en el programa una cláusula cuya cabeza se unifique con la literal seleccionada (ésta también se conoce como submeta). El resolvente de la meta con la cláusula que seleccionamos es el resultado de:

a) Eliminar la literal elegida de la cláusula meta.

b) Reemplazarla con el cuerpo de la cláusula del programa.

c) Tomar el unificador de la meta y la cabeza de la cláusula elegida en el programa y aplicarlo a la meta recien derivada.

Obsérvese que si resolvemos la meta con un hecho, entonces se reemplaza con nada (puesto que los hechos son cláusulas sin cuerpo). Es decir, simplemente removemos la submeta de la meta. Cuando la meta finalmente es la cláusula vacía, entonces hemos reducido la meta original a una colección de hechos, y por lo tanto hemos terminado, probando así la meta original; solo resta considerar las variables que figuran en la meta original, y el resultado de aplicar a estas variables la composición de los unificadores que hemos usado durante el cómputo nos da la respuesta esperada.

1.1. Interpretación de cláusulas

Una claúsula $P :- Q_1, Q_2, \ldots, Q_n$ de PROLOG tiene una interpretación declarativa y una interpretación procedimental:

- Declarativa: P es válida cuando Q_1 y Q_2 y ... y Q_n son válidas
- Procedimental: Para ejecutar (probar) P basta ejecutar Q_1 y Q_2 y ... y Q_n
- La interpretación declarativa permite discutir la correctud de la cláusula.
- La interpretación procedimental permite considerar a la cláusula como la definición de un proceso. Se genera con la ejecución del programa mediante las estrategias de control del interprete de PROLOG .

Con esto terminamos de exponer las ideas fundamentales detras del mecanismo de inferencia de PROLOG . A continuación presentamos algunas generalidades de la sintaxis del lenguaje.

2. Prolog

Para verificar las implementaciones y desarrollar programas propios usaremos el intérprete swi disponible en `www.swi-prolog.org`

2.1. Sintaxis

- Variables: Inician con mayúscula o guión bajo:

  ```
  X, Y2, Un_jardin, _, _x, _Abue44
  ```

- Constantes: inician con minúscula, pueden incluir guión bajo y dígitos, o cualquier cosa entre comillas:

  ```
  a, perro, un_gato, 23, 'Pancho Villa'
  ```

- Predicados: un funtor (nombre de predicado) es un nombre de constante seguido de un número fijo de argumentos.

  ```
  fecha(martes,13,febrero,1994), suma(2,3,X), vuelo(mex,gdl,1830,1945,AM234)
  ```

Por ejemplo, los programas correspondientes a los predicados definidos mediante reglas anteriormente son:

```
nat(0).
nat(s(X)) :- nat(X).

par(0).
par(s(s(X))) :- par(X).

suma(0,N,N).
suma(s(M),N,s(P) :- suma(M,N,P).

prod(0,N,0).
prod(s(M),N,P) :- prod(M,N,Q), suma(Q,N,P).
```

Veamos ahora un ejemplo de resolución que es tal y como PROLOG ejecuta los programas. Cada ejecución es una búsqueda de la cláusula vacía mediante resolución SLD.

Ejemplo 1 Considérese el siguiente programa, que representa a la operación de suma en los números naturales mediante la función sucesor $s(X)$.

```
suma(0,X3,X3).
suma(s(X2),Y2,s(Z2)):-suma(X2,Y2,Z2).
```

Supóngase que queremos saber cuánto es $1 + 2$, la pregunta correspondiente es:

```
?- suma(s(0),s(s(0)),X1).
```

Sólo hay una literal en la cláusula meta, así que la selección es única. Esta submeta se unifica con la cabeza de la regla del programa. El unificador más general es:

$$\mu = [X_2 := 0, Y_2 := s(s(0)), X_1 := s(Z_2)]$$

Por lo tanto nuestro resolvente será la proxima cláusula meta:

$$? - \texttt{suma}(\texttt{X2}, \texttt{Y2}, \texttt{Z2})\mu.$$

o, haciendo la sustitución,

$$? - \texttt{suma}(\texttt{0}, \texttt{s}(\texttt{s}(\texttt{0})), \texttt{Z2}).$$

esta meta se puede resolver con el hecho del programa con la sustitución

$$\tau = [X3 := s(s(0)), Z2 := s(s(0))]$$

La respuesta a la meta original es la aplicación de la composición de las sustituciones encontradas a las variables de la meta original, Esto es:

$$X_1\mu\tau = s(Z_2)\tau = s(s(s(0))).$$

Es decir, $X_1 = 3$ que es la respuesta esperada.

2.2. Operadores de igualdad en Prolog

PROLOG tiene los siguientes operadores de igualdad, de su definición debe quedar claro cual usar en que situación:

$=$ Unificación: es verdadero si ambos operandos se unifican, en cuyo caso la unificación se lleva a cabo.

$\backslash =$ No unificación: es verdadero si sus operandos no se pueden unificar.

$==$ Identidad sintáctica: compara términos sin evaluar expresiones. Es verdadero si sus operandos son exactamente iguales. Ambos términos deben estar instanciados.

$\backslash ==$ es verdadero si no se cumple la identidad sintáctica.

$=:=$ Identidad en valores: evalua los dos operandos y es verdadero si los valores obtenidos son iguales.

is Evaluación aritmética: evalúa el operando derecho y unifica el resultado con el operando izquierdo.

Veamos algunos ejemplos:

- `?- X = 3+5 ----> X=3+5 Yes`
  se unifica con la expresión $3 + 5$.
- `?- X is 3+4 ----> X=7 Yes`
  se unifica con la evaluación de $3 + 4$.
- `?- X+Y=3+5 ----> X=3, Y=5 Yes`
  Porque X y Y se unifican con 3 y 5 respectivamente.
- `?- 3=1+2 ----> No`
  Porque 3 y $1 + 2$ no son unificables.

- `?- 3*3=:=9 ----> Yes`

  Porque $3 * 3$ se evalua a 9.

- `?- 3+Y==3+Y ----> Yes`

  Porque ambos términos son sintacticamente idénticos.

- `?- 4 is 2+2 ----> Yes`

  Porque $2 + 2$ se evalua a 4 que obviamente unifica con 4.

- `?- 5+5 is 2*5 ----> No`

  Porque sólo se evalua $2 * 5$ cuyo resultado 10 no se unifica con $5 + 5$.

- `?- 4 is 3+X ----> Run-time error`

  Porque X no está instanciada.

- `?- X is X+1, X is 4 ----> Run-time error`

  Porque X no está instanciada.

- `?- X is 4, X is X+1 ----> No`

  Porque aunque X está instanciada a 4, 4 y 5 no se unifican.

2.3. Aritmética en Prolog

Si bien la aritmética en PROLOG está predefinida, podemos definir los números naturales y sus operaciones recursivamente como sigue:

2.3.1. Aritmética formal

- Naturales:

```
nat(0).
nat(s(X)) :- nat(X).
```

- Suma:

```
suma(0,N,N).
suma(s(M),N,s(P)) :- suma(M,N,P).
```

- Producto:

```
prod(0,N,0).
prod(s(M),N,P) :- prod(M,N,Q), suma(Q,N,P).
```

- Menor que, mayor que:

```
meq(0,s(X)).
meq(s(X),s(Y)) :- meq(X,Y).

maq(s(X),0).
maq(s(X),s(Y)) :- maq(X,Y).
```

- Cociente de una división

```
% cociente(X,Y,Z) <=> Z es el cociente en la division entera de X entre Y
cociente(X,Y,0):- maq(Y,X).
cociente(X,Y,s(Z)):-suma(Y,X1,X),cociente(X1,Y,Z).
```

Obsérvese que como PROLOG es un lenguaje sin tipos la aplicación de las operaciones no se restringe a naturales, pudiendose obtener respuestas como la siguiente:

```
?-suma(0,cebra,X). ---->  X=cebra Yes

?-suma(s(0),guajolote,X). ----> X=s(guajolote) Yes
```

Esta situación podría mejorarse sustituyendo la definición de suma en 0 por la siguiente:

```
suma(0,N,N) :- nat(N).
```

Sin embargo el problema no se soluciona totalmente pues ahora obtenemos

```
?-suma(0,cebra,X). ---->  No
```

cuando lo que nos gustaría es indicar que la pregunta no tiene sentido.

Para terminar veamos como programar la sucesión de fibonacci en la aritmética formal. La función generadora de la sucesión se define como:

$$fibo(0) = 1, \quad fibo(1) = 1, \quad fibo(n+2) = fibo(n+1) + fibo(n)$$

el programa es el siguiente:

```
fibo(0,s(0)).
fibo(s(0),s(0)).
fibo(s(s(N)),R) :- fibo(s(N),R1), fibo(N,R2), suma(R1,R2,R).
```

2.3.2. Aritmética predefinida

La aritmética predefinida en PROLOG dispone de los operadores usuales:

$$+, -, *, /, \text{div}, \quad \text{mód}$$

y los operadores de comparación

$$<,\ >,\ =<,\ >=,\ =:=, =\backslash=$$

adicionalmente de acuerdo al intérprete particular existen otras funciones matemáticas, como funciones trigonométricas, exponenciales, etc.

2.4. Ejemplos con aritmética

2.4.1. Porcentajes

Supongase que se requiere calcular el porcentaje de juegos ganados de acuerdo a la siguiente información:

```
ganados(j,7).  ganados(s,6).  ganados(p,2).  ganados(R,5).
jugados(j,13).  jugados(s,7).  jugados(p,3).  jugados(R,10).
```

El programa es el siguiente:

```
porcentaje(X,P) :- ganados(X,G), jugados(X,J), P is (G/J)*100.
```

2.4.2. Divisibilidad

.

Por supuesto usando la operación mód podemos definir la divisibilidad como sigue:

```
% div(X,Y) <==> X divide a Y., X,Y enteros.
div(X,Y) :- X mod Y == 0.
```

2.4.3. Contador

Un programa que devuelve la suma de 1 hasta N es el siguiente:

```
% contar(N,R) <==> R=1+2+..+N
contar(1,1).
contar(N,R) :- N1 is N-1, contar(N1,R1), R is R1+N
```

2.4.4. Factorial

```
fac(0,1).
fac(N,F) :- N1 is N-1, fac(N1,F1), F is N*F1.
```

Por supuesto este programa es sumamente ineficiente, una definición más eficaz se obtiene al usar un acumulador:

```
fac2(N,F) :- facacc(N,1,F).
facacc(0,F,F).
facacc(N,A,F) :- M is N-1, Y is N*A, facacc(M,Y,F).
```

2.4.5. Fibonacci

Podríamos intentar con el mismo programa usado en la aritmética formal, simplemente utilizando los números y operadores predefinidos, obteniendo:

```
fibo(0,1).
fibo(1,1).
fibo(N+2,R1+R2) :- fibo(N+1,R1), fibo(N,R2).
```

sin embargo este programa sólo funciona para $N = 0, 1$, al intentar probar para $N = 2$ el programa falla pues 2 no se unifica con $0 + 2$. Lo correcto es utilizar el operador is como sigue:

```
fibo(0,1).
fibo(1,1).
fibo(N,R) :- N2 is N-2, N1 is N-1, fibo(N1,R1), fibo(N2,R2),R is R1+R2.
```

Nuevamente sería mejor utilizar un acumulador para optimizar el programa.

2.5. Listas

Las listas también están predefinidas en PROLOG , formalmente pueden definirse como sigue:

```
lista(nil).
lista(cons(A,L)):-lista(L).
```

`nil` representa a la lista vacía mientras que `cons` es el operador que construye una nueva lista a partir de una lista dada `l` agregando un objeto `a` al inicio de `l`. En lugar de esta definición formal usaremos las listas predefinidas usando la notación usual `[]` para `nil` y `[A|L]` para `cons(A,L)`.

Obsérvese que en la segunda cláusula no se pide nada a la variable A, con lo cual se generan listas heterogeneas como `[a,1,hola, 3.5, X]`. Si bien la sintaxis del programa es legal, el interprete enviará el error `Singleton variables: [A]` debido al hecho de que A sólo se utiliza una vez. Dicho error se corrige al usar en lugar de A la llamada variable anónima de PROLOG , denotada con un guión bajo _ , la cual denota a una variable sin nombre. La cláusula es entonces:

```
lista(cons(_,L)):-lista(L).
```

Es posible definir listas homogeneas, por ejemplo de naturales como sigue:

```
listanat([]).
listanat([H|T]):- nat(H), listanat(T).
```

Obsérvese que si queremos definir listas de cualquier otro tipo `p`, debemos reescribir el programa:

```
listap([]).
listap([H|T]):- p(H), listap(T).
```

Es decir, PROLOG no es un lenguaje polimórfico. Una solución podría ser el intentar escribir un sólo programa que cubra el caso para cualquier predicado, por ejemplo:

```
plista(P,[]).
plista(P,[H|T]) :- P(H), plist(P,T).
```

Si bien este programa es aceptado por algunos interpretes (no en SWI) no puede justificarse con los fundamentos lógicos de primer orden. Obsérvese que la literal $P(X)$ es una variable P que representa a un predicado cualquiera aplicado a un objeto X. Esto es legal en lógica de orden superior mas no en lógica de primer orden. De manera que tales programas deben ser evitados.

2.6. Ejemplos con listas

2.6.1. Pertenencia

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

La definición de `elem` deja ver la flexibilidad de un programa en PROLOG , además de su uso estandar, `elem` puede usarse para:

- Obtener todos los elementos de una lista `?- elem(X,[a,b,c,d,e,f]).`
- Generar listas con un elemento dado en cualquier posición `?-elem(pato,X)`.

2.6.2. Concatenación

```
concat([],L,L]).
concat([H|T],L,[H|NT]):- concat(T,L,NT).
```

2.6.3. Reversa

```
reversa([],[]).
reversa([H|T],R) :- reversa(T,RT), concat(RT,[H],R).
```

Usando la definición de reversa podemos definir fácilmente un progrma que reconozca listas palindrómicas:

```
palindromo(X) :- reversa(X,X).
```

2.6.4. Usando `concat` de manera no estandar

Los programas para pertenencia a una lista, prefijos, sufijos y sublistas pueden definirse sin recurrir a una definición recursiva, usando la contatenación de manera no estandar:

- Pertenencia a una lista:

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

- Prefijos:

  ```
  prefijo(P,L) :- concat(P,_,L).
  ```

- Sufijos

  ```
  sufijo(S,L) :- concat(_,S,L).
  ```

- Sublistas:

  ```
  sublista(S,L) :- concat(AS,_,L), concat(_,S,AS)
  ```

2.6.5. Permutaciones de una lista

```
% perm(L,M)/2 <=> M es una permutación de L
perm([],[]).
perm(Ps,[L|Ls]):-elim(L,Ps,Rs),perm(Rs,Ls).
```

2.7. Ejemplos usando acumuladores

2.7.1. Suma de números naturales

```
suma(0,X,X).
suma(s(X),Y,Z) :- suma(X,s(Y),Z).
```

2.7.2. Suma de elementos de una lista de números

La idea es ir acumulando la suma en cada iteración.

```
suml(L,S) :- sumaacc(L,0,S)

sumaacc([],Sacc,Sacc)
sumaacc([X|Xs], Sacc,S) :- Sn is X+Sacc, sumaacc(Xs,Sn,S)
```

2.7.3. Suma acumulada de una lista

Dada una lista $[a_1, \ldots, a_n]$ devolver la lista $[s_1, \ldots, s_n]$ cuyos elemento i es las suma de valores de la lista anterior hasta dicho elemento, es decir, tal que $s_i = a_1 + \ldots + a_i$.

Por ejemplo `sumaval([1,2,3,4],[1,3,6,10]`.

```
sumval(L,S) :- svacc(L,0,S)

svacc([],S,[]).
svacc([X|Xs],Sacc,[Sp|SXs]) :- Sp is X+Sacc, svacc(Xs,Sp,SXs).
```

2.7.4. Producto escalar de dos listas de números

```
esc(Xs,Ys,P) :- escacc(Xs,Ys,0,P).

escacc([],[],P,P).
escacc([X|Xs],[Y|Ys],Pacc,P) :- Pn is Pacc + X*Y, escacc(Xs,Ys,Pn,P).
```

2.8. Funciones en Prolog

Comunmente se dice que en la programación lógica no existen funciones, esto es correcto en cierto sentido, sin embargo si podemos hacer referencia a funciones y ya lo hemos hecho, por ejemplo al definir la aritmética formal usamos la función sucesor $s(X)$ en cláusulas como `suma(s(X),Y,s(Z)) :- suma(X,Y,Z)`.

De gran importancia es observar que el mecanismo de inferencia de PROLOG no permite evaluar funciones, por ejemplo $s(0)$ nunca se podra evaluar a 1. Si se desea evaluar una función debe definirse un programa que lo haga. Por ejemplo para convertir numerales $s(X)$ a números de la aritmética predefinida tenemos el siguiente programa:

```
ntn(0,0).
ntn(s(X),N) :- ntn(X,M), N is M+1.
```

De esta forma las funciones solo sirven para estructurar relaciones especificando la información. Por ejemplo si queremos mantener información acerca de personas y sus ocupaciones podemos usar un simple predicado `persona(robin,hood,arquero)` cuyos argumentos sean el nombre, apellido y ocupación de la persona respectivamente. Sin embargo el uso de funciones permite estructurar la información de manera similar a un *record* de la programación imperativa.

```
persona(nombre(robin,hood),ocupacion(arquero))
persona(nombre(harry,potter),ocupacion(mago))
```

En este caso se tienen dos funciones `nombre` y `ocupacion`, estas no pueden ser considerados predicados pues figuran dentro del predicado `persona`. Por ejemplo `nombre(robin)` denota al nombre de la persona a la cual llamamos `robin`.

Podemos estructurar la información aún mas, por ejemplo dando funciones para nombre de pila y apellido.

```
persona2(nomcom(np(robin),ap(hood)),ocupacion(arquero))
persona2(nomcom(np(harry),ap(potter)),ocupacion(mago))
```

A la hora de recuperar información debemos mantener la misma estructura, por ejemplo, las siguientes son métas exitosas:

```
?- persona(N,O).
?- persona(nombre(N,A),O).
?- persona(N,ocupacion(O)).
?- persona2(N,O).
?- persona2(nomcom(N,A),O).
?- persona2(nomcom(np(N),np(A)),O).
```

y las siguientes metas fallarán:

```
?- persona(nombre(N),O).
?- persona(X(N,A),Y(O)).
?- persona2(nomcom(N),O).
```

Veamos un ejemplo relativo a vuelos. el predicado `vuelo(mexico,frankfurt,miercoles).` puede estructurarse usando símbolos de función como sigue:

```
vuelo(ori(mexico),des(frankfurt),dia(miercoles)).
```

o si se desea agregar códigos de aeropuerto:

```
codvuelo( ori(cod(mex),cd(mexico)),
          des(cod(fra),cd(frankfurt)),
          dia(miercoles)).
codvuelo( ori(cod(fra),cd(frankfurt)),
          des(cod(muc),cd(muenchen)),
          dia(miercoles)).
```

Un programa para verificar conexiones de vuelos es:

```
% Hay conexión si hay vuelo directo
conx(cod(X),cod(Y),dia(D)) :- codvuelo( ori(cod(X),CX),
                                        des(cod(Y),CY),
                                        dia(D)).

% Hay conexión si hay un vuelo directo a una cd. intermedia y de ahí
% una conexión.
conx(cod(X),cod(Y),dia(D)) :- codvuelo( ori(cod(X),CX),
                                        des(cod(Y),CY),
                                        dia(D)).

conx(cod(X),cod(Y),dia(D)) :- codvuelo( ori(cod(X),CX),
                                        des(cod(Z),CZ),
                                        dia(D)),
                              conx(cod(Z),cod(Y), dia(D)).
```

2.9. Árboles Binarios

Los árboles binarios son una estructura de gran importancia en programación. En PROLOG pueden definirse con ayuda de símbolos de función de acuerdo a la siguiente definición:

- El árbol vacío es un árbol binario denotado con `void`.
- Si T_1, T_2 son árboles binarios y a es una etiqueta entonces `t(a,T1,T2)` denota al árbol binario con raiz a, subárbol izquierdo $T1$ y subárbol derecho $T2$. Aquí `t` es un símbolo de función.

De esta definición surge el predicado `arbin(T)`:

```
arbin(void).
arbin(t(A,T1,T2)) :- arbin(T1), arbin(T2).
```

Algunos árboles binarios son:

```
t(0,
   t(1,
    t(3,
      t(5,void,void),
      void),
    t(4,
      void,
      t(6,void,void))
    ),
  t(2,
    t(8,
      t(7,void,void),
```

```
t(a,
  void,
  t(b,
    t(c,void,void),
    t(e,
      t(g,
        void,
        t(h,void,void)
        ),
      t(f,
        t(k,void,void),
        void
```

```
      t(9,void,void)                                          )
     ),                                                      )
    void                                                    )
   )                                                       )
 )
```

2.9.1. Hojas

Los árboles que son hojas son aquellos cuyos subárboles son vacíos `t(a,void,void)`

```
hoja(t(A,void,void)).
```

2.9.2. Pertenencia de un nodo a un árbol

```
enarb(N,t(N,_,_)).
enarb(N,t(_,T1,_)) :- elem(N,T1).
enarb(N,t(_,_,T2)) :- elem(N,T2).
```

2.9.3. Altura de un árbol

La altura es la distancia máxima de la raiz a un nodo.

```
alt(void,0).
alt(t(_,AI,AD),N) :- alt(AI,N1),
                     alt(AD,N2),
                     max(N1,N2,M),
                     N is M+1.
```

2.9.4. Número de nodos

```
nnodos(void)=0.
nnodos(t(_,AI,AD), N) :- nnodos(AI,N1),
                         nnodos(AD,N2),
                         N is N1+N2+1.
```

2.9.5. Sustitución de un nodo

```
%sust(T,TN,V,N) devuelve en T2 el árbol obtenido al sustituir todas
%las presencias del nodo etiquetado V con N.

sust(void,void,_,_).

sust(t(V,AI,AD),t(N,AIN,ADN),V,N) :- sust(AI,AIN,V,N),
                                     sust(AD,ADN,V,N).


sust(t(X,AI,AD),t(X,AIN,ADN),V,N) :- V\=X, sust(AI,AIN,V,N),
                                     sust(AD,ADN,V,N).
```

2.9.6. Preorden

```
% Devuelve los nodos de un árbol en preorden

preorden(void).
preorden(t(R,TI,TD)) :- printl(R), preorden(TI), preorden(TD).
```

2.9.7. Postorden

```
% Devuelve los nodos de un árbol en postorden

postorden(void).
postorden(t(R,TI,TD)) :-  postorden(TI), postorden(TD), printl(R),.
```

2.9.8. Inorden

```
% Devuelve los nodos de un árbol en inorden

inorden(void).
inorden(t(R,TI,TD)) :-  inorden(TI), printl(R), inorden(TD),.
```

2.9.9. Caminos desde la raiz

```
% path(L,T) devuelve en L una lista que contiene el único camino desde
% la raiz a cualquier vértice.

path([X], t(X,_,_)).
path([X|P],t(X,TI,_)) :- path(P,TI).
path([X|P],t(X,_,TD)) :- path(P,TD).
```

2.9.10. Lista de hojas

```
%hojas(L,T) devuelve en L la lista de hojas del arbol T desde la izquierda
hojas([],void).
hojas([R], t(R,void,void)).
hojas(H, bt(_,TI,TD)) :- hojas(HI,TI), hojas(HD,TD),
                         append(HI,HD,H).
```

2.10. El operador de corte

Considérese el siguiente programa, el cual define una función por partes:

```
f(X,0):-X<3.
f(X,2):- 3 =< X, X<6.
f(X,4):-6=<X.
```

Si se consideran las metas

```
?- f(1,Y),2<Y.
```

el interpreté fallará pero sólo después de intentar resolver con cada una de las tres cláusulas.

- En el primer caso se instancia `Y=0` con lo que se tiene éxito en la primera literar y se intenta verificar $2 < 0$ lo cual causa fallo.
- El retroceso automático de PROLOG intenta ahora usar la segunda cláusula unificando con `Y=2` y fallará al intentar $3 =< 1$.
- Finalmente con la tercera cláusula se unifica `Y=4` y fallará al intentar $6 =< 1$.
- En este caso el retroceso es inutil pues sabemos que las condiciones de las cláusulas son mutuamente excluyentes por lo que al cumplirse la condición en la primera cláusula las otras dos fallarán.
- En el momento en que la primera cláusula falla debe detenerse la búsqueda y fallar.
- Esto puede hacerse utilizando el operador de corte `!`.
- `!` es un predicado que siempre tiene éxito
- `!` elimina las alternativas restantes, es decir el retroceso, más allá de los puntos donde figura.

El programa resulta más eficiente al agregar los cortes:

```
f(X,0):-X<3,!.
f(X,2):- 3=< X, X<6,!.
f(X,4):- 6=< X.
```

de esta manera al fallar en la primera regla no se analizan las otras dos y se falla.
Obsérvese que los resultados no cambian al agregar el corte, sólo se eficienta el programa al cancelar búsquedas inútiles. Es decir sólo cambiamos el significado procedimental del programa, no el declarativo. Los cortes de este tipo se llaman cortes verdes.

Sin embargo el programa aun es ineficiente. Considérese ahora la meta

```
?-f(7,Y)
```

la cual tiene éxito con `Y=4`
La ejecución es como sigue:

- La primera regla falla con $7 < 3$. Se retrocede e intenta con la segunda (el corte no se alcanzó, la falla fue anterior).
- La segunda regla falla con $7 < 6$. Se retrocede e intenta con la tercera regla (el corte no se alcanzó).
- Se tiene éxito pues $6 =< 7$.

El verificar ciertas condiciones es ineficiente pues sabemos que son ciertas de antemano, por ejemplo si en la primera regla se fallo con $7 < 3$ automáticamente $3 \leq 7$ es cierto. Análogamente al fallar en la segunda $7 < 6$ entonces la condición $6 \leq 7$ se cumple. El programa se eficienta de la siguiente manera:

```
f(X,0):-X<3,!.
f(X,2):-X<6,!.
f(X,4).
```

Pero obsérvese que en este caso el corte es esencial, es decir, eliminarlo modifica los resultados del programa. Por ejemplo

```
?- f(1,Y)
```

devuelve $Y = 0, Y = 2, Y = 4$.

Se afecta entonces tanto el significado procedimental como el declarativo. Los cortes de este tipo se llaman cortes rojos.

Veamos algunos ejemplos

2.10.1. Pertenencia a una lista

```
%El corte es rojo.
enlista(X,[X|_]):= !.
enlista(X,[_|Y]).
```

2.10.2. Agregar un elemento a una lista

```
agr(X,L,L):- enlista(X,L),!.
agr(X,L,[X|L]).
```

2.10.3. If-then-else

En general la instrucción imperativa $If\ p\ then\ q\ else\ r$ se modela como:

```
x :- p,!,q.
x :- r.
```

La idea es que x se cumple si se cumple $If\ p\ then\ q\ else\ r$

2.10.4. Suma de los elementos de una lista

```
sumlis([H|T],N) :- sum(T,N1), N is N1+1,!.
sumlis([],0).
```

2.10.5. Posiciones de un elemento en una lista

```
%posic(X,L,N) Devuelve en N todas las posiciones de X en L.
posic(X,[X|_],1).
posic(X,[_|Xs],N):-posic(X,Xs,M),N is M+1.

%posic1(X,L,N) Devuelve en N la primera posición de X en L sin usar corte.
posic1(X,[X|_],1).
posic1(X,[Y|Xs],N):-X\=Y,posic1(X,Xs,M),N is M+1.
```

```
%posic1c(X,L,N) Devuelve en N la primera posición de X en L usando un corte.
posic1c(X,[X|_],1):-!.
posic1c(X,[_|Xs],N):-posic1c(X,Xs,M),N is M+1.
```

2.10.6. Posiciones de un elemento a partir de la última

```
% posicf(X,L,N) devuelve en N las posiciones de X en L, iniciando con
% la última.
posicf(X,[_|Xs],N):-posicf(X,Xs,M),N is M+1.
posicf(X,[X|_],1).


%Lo mismo usando el corte.
posicfc(X,[_|Xs],N):-posicfc(X,Xs,M),!,N is M+1.
posicfc(X,[X|_],1).
```

3. Teoría de programación lógica: Árboles asociados a un programa lógico

Definición 1 *Sean $\mathbb{P}$ un programa lógico y G una meta. Un árbol SLD* [1] *es un árbol n-ario posiblemente infinito cuya raíz es la meta G y cuyos nodos tienen metas exclusivamente; el hijo de un nodo G_i es la nueva meta G_j obtenida a partir de la resolución binaria de G_i y una cláusula del programa.*

En el caso particular de un programa en PROLOG , al árbol SLD se le llama árbol de búsqueda (PROLOG search tree) y muestra todos los caminos que recorre el intérprete en la búsqueda por la cláusula vacía $\square$. Aquellas ramas finitas que terminan en la cláusula vacía $\square$ se llaman ramas de éxito, las que terminan en una cláusula no vacía se llaman ramas de fallo.

Definición 2 *Una SLD-derivación es un árbol potencialmente infinito que consistente de metas $G_0, G_1, \ldots, G_n$, cláusulas de programa $\mathcal{C}_0, \ldots, \mathcal{C}_m$ y unificadores $\mu_0, \ldots, \mu_k$ tales que:*

- *G_{n+1} es un SLD-resolvente de G_n y $\mathcal{C}_m$ via el umg μ_k, para toda n.*

En este árbol si C_R es la cláusula resolvente a partir de la cláusula de programa C y de la meta G entonces el árbol tendrá a C_R como hijo de C y G. Si la derivación Δ es finita digamos $G_0, \ldots, G_i$ decimos que i es la longitud de Δ.

Definición 3 *Sean $\mathbb{P}$ un programa lógico y G una meta. Una SLD-refutación para $\mathbb{P} \cup \{G\}$ es una SLD-derivación finita Δ con metas $G_0, \ldots, G_n$ y cláusulas de programa $\mathcal{C}_0, \ldots, C_m$ tal que:*

- *$G_0 = G$ y $G_n = \square$.*

[1] En inglés SLD-resolution tree o SLD-tree.

Ejemplo 3.1 Considera el siguiente programa:

1. $p(a).$
2. $p(b).$
3. $q(a).$
4. $q(b).$
5. $r(b).$
6. $s(X) : -p(X), q(X), r(X).$

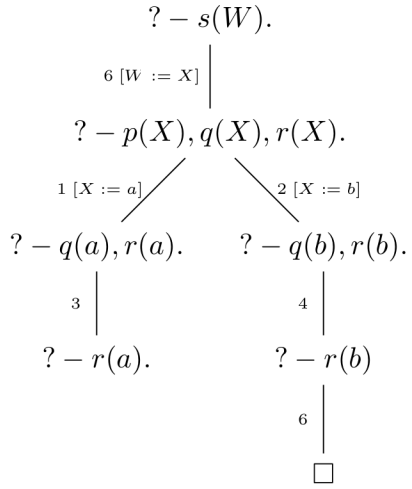


Figura 1: Árbol SLD para la meta $?. - s(W)$.

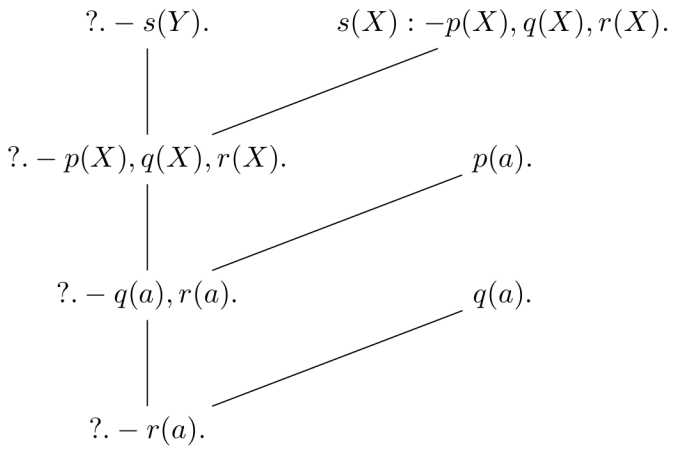


Figura 2: SLD derivación para $?. - s(Y)$.

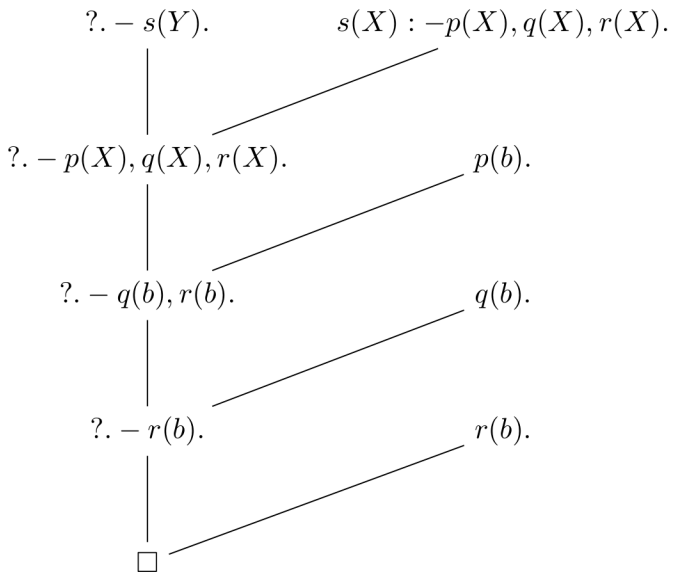


Figura 3: SLD refutación para $?. - s(Y)$.

4. Semántica de programas lógicos

Una parte importante de cada paradigma de programación es la semántica, por medio de la cual se le da significado a un programa, situación que nos permite describir formalmente lo que éste calcula. Al inicio de esta nota mencionamos una ventaja de la programación declarativa: la elegancia matemática resultado de la descripción del programa mediante enunciados precisos, así el significado del programa es claro y facilita su verificación.

En esta sección trataremos brevemente con dos clases de semántica para programas lógicos: la declarativa y la procedimental u operacional.

Comencemos observando que una cláusula `P :- Q1,Q2,...,Qn` de PROLOG tiene una interpretación declarativa y una interpretación procedimental:

- Declarativa: P es válida si Q_1 y Q_2 y ... y Q_n son válidas.
  La interpretación declarativa permite discutir la correctud de la cláusula.
- Operacional: Para ejecutar (probar) P basta ejecutar Q_1 y Q_2 y ... y Q_n.
  La interpretación operacional permite considerar a la cláusula como la definición de un proceso. Esta semántica Se genera con la ejecución del programa mediante las estrategias de control del intérprete de PROLOG .

Una pregunta importante es si ambas interpretaciones generan la misma información, para contestarla necesitamos hablar del concepto de respuesta de manera formal.

Definición 4 (Respuesta) *Sean $\mathbb{P}$ un programa lógico y $G =$?- $G_1, \ldots, G_m$ una meta. Una respuesta para $\mathbb{P} \cup \{G\}$ es una sustitución σ tal que $Var(G_1) \cup \ldots \cup Var(G_m) \subseteq \mathsf{dom}(\sigma)$, es decir una sustitución que incluye a las variables de G.*

Definición 5 (Respuesta correcta) *Decimos que una respuesta σ para $\mathbb{P} \cup \{G\}$ es correcta si $\mathbb{P} \models G_i\sigma$ para toda $1 \leq i \leq m$, o equivalentemente si $\forall\mathbb{P} \models \forall\big((G_1 \wedge \ldots \wedge G_m)\sigma\big)$.*

Recordemos que $\forall\varphi$ denota a la cerradura universal de φ obtenida cuantificando universalmente todas las variables libres de φ. Análogamente $\forall\mathbb{P}$ se obtiene de $\mathbb{P}$ al cuantificar universalmente todas las variables de cada una de sus cláusulas. Intuitivamente una respuesta correcta para $\mathbb{P} \cup \{G\}$ corresponde a una consecuencia lógica particular del programa, y es entonces una significado declarativo del programa.

Ejemplo 4.1 Considérese el programa

$$\mathbb{P} = \{mq(0, suc(X)),\ mq(suc(Y), suc(X) \text{ :- } mq(Y, X)\}.$$

Entonces $\sigma = [Y := suc(suc(0))]$ es una respuesta correcta para $? - mq(0, Y)$ pues

$$\forall\mathbb{P} \models mq(0, Y)[Y := suc(suc(0))]$$

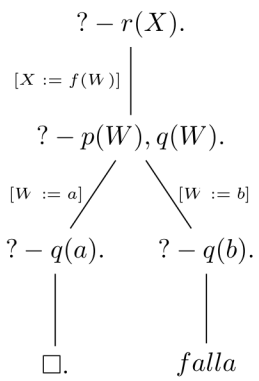
Definición 6 (Respuesta computada) *Sean $\mathbb{P}$ un programa lógico y G una meta. Una sustitución σ es una respuesta computada para $\mathbb{P} \cup \{G\}$ si y sólo si existe una rama de éxito en el árbol de SLD-resolución (o árbol de búsqueda) de longitud n con unificadores más generales $\mu_0, \ldots, \mu_{n-1}$ de tal forma que $\sigma = \mu_0\mu_1 \ldots \mu_{n-1}\restriction_{Var(G)}$*

Es decir σ es una respuesta computada para $\mathbb{P} \cup \{G\}$ si y sólo si σ es la restricción de la composición de los unificadores de una rama de éxito en el árbol de SLD-resolución para $\mathbb{P} \cup \{G\}$.

Intuitivamente una respuesta computada corresponde al resultado del proceso de inferencia por parte del sistema. Las respuestas computadas son aquellas que el intérprete de PROLOG devuelve al usuario.

Ejemplo 4.2 Si $\mathbb{P} = \{p(a),\ p(b),\ q(a),\ r(f(X)) \text{:- } p(X), q(X)\}$ entonces tenemos la siguiente rama de éxito en el árbol de SLD-resolución para la meta $G =$?- $r(X)$:

1.	$p(a).$	$Hip.$
2.	$p(b)$	$Hip.$
3.	$q(a).$	$Hip.$
4.	$r(f(Y)) \text{:- } p(Y), q(Y).$	$Hip.$
5.	?- $r(X)$	$Meta$
6.	?- $p(Y), q(Y)$	$SLDRes(4, 5, [X := f(Y)])$
7.	?- $q(a)$	$SLDRes(6, 1, [Y := a])$
8.	$\square.$	$SLDRes(3, 7)$



La composición de los unificadores es $[X := f(a), Y := a]$ de manera que la sustitución $[X := f(a)]$ es una respuesta computada para $\mathbb{P} \cup \{$?- $r(X)\}$.

El significado de un programa lógico se debe dar mediante sus respuestas, intuitivamente el significado es el conjunto de respuestas. Por ejemplo el siguiente programa $\mathbb{P}$, implementa la búsqueda de caminos en una gráfica:

```
edge(a,b).
edge(b,c).
edge(b,d).
edge(c,d).
edge(e,f)
path(X,X).
path(X,Y) :- edge(X,Z),path(Z,Y).
```

De manera que el significado intensional de $\mathbb{P}$ es el conjunto de todos los caminos posibles (especificando todo los vértices del camino) en la gráfica. Pero recordemos que la programación lógica tiene un uso no estándar, por ejemplo el programa para concatenar dos listas, sirve también para descomponer una lista en dos partes. Por lo que con esta definición informal no es tan claro cuál es el significado. Más aún, dado un programa lógico $\mathbb{P}$, podemos asignarle dos significados:

- Significado declarativo: el significado de un programa lógico $\mathbb{P}$ es el conjunto de todas las consecuencias lógicas del programa, noción asociada a la de respuesta correctas.
- Significado operacional: el significado de un programa lógico $\mathbb{P}$ es el conjunto de todas los éxitos del programa, noción asociada a la de respuesta computada.

Por supuesto que el significado debería ser único, pero no es claro que las dos nociones recién enunciadas sean equivalentes. De los ejemplos anteriores se observa que una respuesta correcta también es una respuesta computada y viceversa, ¿es esto válido en general?, es decir ¿Toda respuesta correcta puede computarse? y ¿Toda respuesta computada es correcta? esto ayudará a probar que las dos semánticas coinciden.
Resulta que en efecto ambos conceptos resultan equivalentes de cierta manera. Los enunciados formales para esta equivalencia, así como su demostración requieren de conceptos como los modelos de Herbrand o sintácticos.