Programación Declarativa, 2021-1
Nota de clase 4: Control de programas en PROLOG*

Manuel Soto Romero

14 de octubre de 2020
Facultad de Ciencias UNAM

En las notas pasadas, estudiamos los aspectos sintácticos y semánticos elementales del lenguaje de programación PROLOG. En esta nota se presentan algunos mecanismos que permiten analizar y modificar el flujo de consulta de los programas. Se presentan los mecanismos de retroceso, corte y negación como falla.

4.1. El mecanismo de retroceso

En la Nota de Clase 2, platicamos sobre el mecanismo de retroceso que se genera a partir de los puntos de elección, de manera que los programas lógicos hagan uso del no determinismo para encontrar todas las posibles soluciones a un problema dado. Veamos un ejemplo de aplicación de este concepto:

Ejemplo 4.1. Un *autómata finito no determinista (AFN)*, es representado por la tupla $(Q, \Sigma, \delta, q_0, F)$ donde: [2]

- Q es un conjunto de estados
- Σ es un alfabeto de entrada finito
- $\delta \subseteq Q \times \Sigma \times Q$ es la función de transicion
- q_0 es el estado inicial
- $F \subseteq S$ es un conjunto de estados finales

Se dice que una cadena $x_1x_2...x_n \in \Sigma^*$ es aceptada por un AFN si existe una secuencia de la forma:

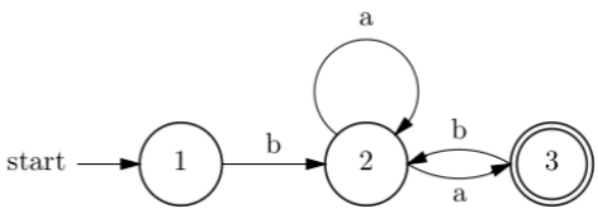
$(q_0, x_1, q_1), (q_1, x_2, q_2), ... (q_{n-1}, x_n, q_n) \in \delta^*$

tal que $q_n \in F$ y δ^* es el conjunto de todas las funciones δ. Un AFN usualmente es representado como una gráfica dirigida dónde los vértices representan a los estados y las aristas a las transiciones. Los estados finales se denotan por círculos dobles.

Por ejemplo, el siguiente autómata se representa por la tupla cuyos elementos son.

*Basada en el *Manual de Prácticas para la Asignatura de Programación Declarativa.*

- $Q = \{1, 2, 3\}$
- $\Sigma = \{a, b\}$
- δ la función de transición dada por:
  $(1, b, 2)$
  $(2, a, 2)$
  $(2, a, 3)$
  $(3, b, 2)$
- $q_0 = 1$
- $F = \{3\}$



De la misma forma, un autómata puede ser representado en PROLOG mediante hechos como puede apreciarse en el Código 1. Las líneas 2 a 4 representan los estados del autómata, las líneas 6 y 7 indican cuáles son los estados inicial y final respectivamente y finalmente, las líneas 9 a 12 representan las funciones de transición del autómata.

```
% Hechos. Base de conocimientos.
estado(1).
estado(2).
estado(3).

inicial(1).
final(3).

transicion(1,b,2).
transicion(2,b,2).
transicion(2,a,3).
transicion(3,b,2).
```

Código 1: Representación de un AFN mediante hechos

Para determinar si una cadena es aceptada por un AFN se aprovecha el mecanismo de retroceso de PROLOG. Para representar a las cadenas, en este ejemplo se usan listas. En el Código 2 se muestra la definición de este predicado, con las líneas 4 y 5 se busca un estado inicial `Q` y a partir de este se inicia la recursión. Las líneas 9 a 12 recorren la lista de estados posibles recorriendo cada camino, hasta llegar a un estado final. La consulta hecha al Código 2 muestra el proceso para las cadenas `"ba"` y `"baa"` representadas como lista.

```
% Predicado acepta(L) que indica si una cadena es aceptada por el
% autómata definido.
acepta(L) :-
   inicial(Q),
   aceptaAux(L,Q).

% Regla auxiliar que se cumple si una cadena es aceptada por un AFN
```

```
% dado un estado.
aceptaAux([],Q) :- final(Q).
aceptaAux([C|R],Q) :-
   transicion(Q,C,Y),
   aceptaAux(R,Y).
```

Código 2: Predicado que determina si una cadena es aceptada por un AFN

```
?- acepta([b,a]).
true.
?- acepta([b,a,a]).
true ;
false.
```

Se aprecia también en la consulta al programa del Código 2 como el no determinismo permite hallar todas las soluciones aun problema, en este caso, recorrer todos los caminos posibles del autómata hasta encontrar todas las posibles respuestas como puede apreciarse en la consulta.

En ocasiones es necesario contar con un mecanismo que le permita al lenguaje detener la búsqueda de pruebas. Este mecanismo existe y se conoce como *operador de corte*.

4.2. El operador de corte

Aunque la técnica de retroceso es de gran utilidad, existen algunas situaciones en que resulta en una desventaja. Por ejemplo, consideremos el Código 3 que calcula el n-ésimo término de la sucesión de Fibonacci.

```
% Predicado fibonacci(N,F) que relaciona a un número N con su
% correspondiente valor F en la sucesión de Fibonacci.
fibonacci(0,1).
fibonacci(1,1).
fibonacci(N,X) :-
   N2 is N-1,
   N3 is N-2,
   fibonacci(N2,F1),
    fibonacci(N3,F2),
    X is F1+F2.
```

Código 3: Predicado que se cumple si un número es el Fibonacci de otro

Al realizar la consulta `fibonacci(3,X)` el intérprete arroja la respuesta `X=3` lo cual es correcto; sin embargo, la consulta no finaliza y el intérprete da la opción de teclear el símbolo `;` para conocer más resultados, sin embargo, `Prolog` lanza un error indicado que la pila de llamadas a función (predicados) se desbordó.

Esto puede llegar a ser confuso, pues la función `fibonacci`, definida en el predicado, tiene solución única. Esta situación ocurre debido al mecanismo de retroceso, pues aunque se llega a un hecho, `Prolog` continúa con la búsqueda de otras soluciones, llegando a un camino dónde la pila se desborda.

Ejemplo 4.2. Al realizar la consulta `fibonacci(0,X)` que es, por definición, caso base, se realizan los siguientes pasos:

- `fibonacci(0,X)` Se tienen dos reglas cuyas conclusiones corresponden con la meta. Se tiene un punto de elección:

  ```
  fibonacci(0,1)
  fibonacci(0,X)
  ```

- Se elige la primera regla `fibonacci(0,1)`. Al ser un hecho, la prueba tiene éxito y se concluye que `X=1`.
- Debido al retroceso, PROLOG vuelve al punto de elección anterior y trata de encontrar otra solución por lo que continúa con la regla:

  ```
  fibonacci(0,X) :-
      N1 is -1,
      N2 is -2,
      fibonacci(-1,F1),
      fibonacci(-2,F2),
      X is F1+F2.
  ```

- Al no tener hechos definidos para números negativos, PROLOG sigue restando el número y generando llamadas a submetas dentro de la pila de llamadas a función hasta llegar a un desborde.

PROLOG provee un mecanismo para manejar este tipo de situaciones llamado *corte*. Un corte detiene el retroceso en un punto de elección dado, evitando así que se sigan buscando caminos alternos.

Para aplicar un corte se usa el predicado `!` que se puede incluir dentro del cuerpo de cualquier regla de inferencia. Cuando PROLOG encuentra el predicado de corte dentro de una regla, ésta tiene éxito y se detiene el proceso de retroceso. [1]

De esta forma, se puede redefinir el Código 3 como se aprecia en el Código 4. Las líneas 2 y 3, que son los casos base del predicado, dejan de ser hechos y se convierten en reglas que aplican el operador de corte, de esta forma, al llegar a un caso base, se detiene el retroceso y se dice que la consulta tuvo éxito. Gracias al uso del operador, la regla deja de evaluar todos los casos posibles y así se evita el desbordamiento de la pila de llamadas a función que se tenía anteriormente.

```
% Predicado fibonacci(N,F) que relaciona a un número N con su
% correspondiente valor F en la sucesión de Fibonacci.
fibonacci(0,1) :- !
fibonacci(1,1) :- !
```

```
fibonacci(N,X) :-
   N2 is N-1,
   N3 is N-2,
   fibonacci(N2,F1),
    fibonacci(N3,F2),
    X is F1+F2.
```

Código 4: Predicado que se cumple si un número es el Fibonacci de otro usando cortes

4.3. Negación como falla

Adicional a los cortes, hay ocasiones en las que la naturaleza de un problema requiere verificar que una propiedad no se satisface. Para este tipo de situaciones, PROLOG provee una primitiva que funciona de forma similar a la negación lógica. Esta primitiva es la *negación como falla*, que se representa mediante el predicado `\+` y funciona como sigue: [1]

Una expresión de la forma `\+G` *tendrá éxito siempre que* `G` *falle.*

La idea detrás de la negación como falla consiste en construir todo el árbol de resolución de una meta para probar que ésta no tiene éxito. [1]

Ejemplo 4.3. Dada la base de conocimientos del Código 5.

```
% Hechos. Base de conocimientos.
comida_favorita(pedro,enchiladas).
comida_favorita(diana,pozole).
```

Código 5: Base de conociientos de comida favorita

La siguiente consulta tiene éxito, pues no hay información (hecho o regla) que permita satisfacer la meta.

```
?- \+ comida_favorita(bety,X).
true.
```

Sin embargo, esto no significa que `bety` tiene una comida favorita, sino que no se tiene información suficiente para afirmarlo. Este tipo de argumento se conoce como *la hipótesis del mundo cerrado*[1] [1]

"Lo que no se puede probar como cierto, se prueba como falso"

Esto quiere decir que si $\nvdash G$ entonces $\neg G$[2]. De esta simbología surge el símbolo de la negación como falla `\+` representa $\nvdash$. [1]

Observación 4.1. La negación como falla no instancia variables.

Referencias

[1] Favio E. Miranda, Lourdes del C. González, *Notas de Clase de Programación Funcional y Lógica*, Facultad de Ciencias UNAM, Revisión 2012-1.

[2] Ulf Nilsson, Jan Maluszynsky, *Logic Programming and Prolog*, Segunda Edición, 2000.