Práctica 2: Programación declarativa, lógica y restricciones

AUTORES

Ignacio Amaya de la Peña – 11m021

Adrián Cámara Canuedo – 11m004

Borja Mas García – 11m001

En esta memoria vamos a incluir los razonamientos que hemos seguido a la hora de resolver los ejercicios y a explicar los predicados usados para ello.

PRIMERA PARTE: HIPERCUBOS

Enunciado

En geometría, un hipercubo es un cubo n-dimensional. En cuatro dimensiones espaciales se compone de 8 cubos, 24 cuadrados (caras), 32 segmentos (aristas) y 16 puntos (vértices).

Tenemos que programar un predicado hipercubo/2 que proporcione el número de elementos n-dimensionales (puntos, segmentos, cuadrados, cubos , etc.) que tiene un hipercubo de una dimensión dada.

hipercubo(N,L): N es un número natural (que vendrá dado en la llamada) que indica la dimensión del hipercubo en cuestión y L es la lista con los elementos existentes de cada dimensión, desde los elementos de dimensión N hasta los de dimensión 0.

Mostrar la llamada necesaria para encontrar los elementos del hipercubo de dimensión 12 y el resultado obtenido.

Una forma de calcular el número de elementos n-dimensionales es elevando el polinomio x+2 a la dimensión del hipercubo, siendo los coeficientes de cada potencia x^n del polinomio resultante el número de elementos n-dimensionales que tiene el hipercubo.

Por ejemplo, para el hipercubo de dimensión 2 (un cuadrado) tendríamos el polinomio (x+2)*(x+2) = x^2+4*x+4, es decir, tiene:

- un elemento de dimensión 2 (el cuadrado, correspondiente al sumando x^2)

- cuatro de dimensión 1 (segmentos) (los lados, correspondientes al sumando 4*x)

- cuatro de dimensión 0 (puntos) (los vértices, correspondientes al sumando 4)

Resolución

Para resolver este problema hemos realizado varios predicados que nos permitiesen representar operaciones básicas entre polinomios, cuyo código se indica a continuación:

```
suma([],[],[]).
suma([],Y,Y).
suma(X,[],X).
suma([X|Xs],[Y|Ys],[Z|Zs]):-
  suma(Xs,Ys,Zs),Z is X+Y.
```

```
mult(0,_,[]).
mult(_,[],[]).
mult(X,[Y|Ys],[Z|Zs]):-
  Z is X*Y,
  mult(X,Ys,Zs).
```

El predicado suma comprueba si un polinomio es suma de otros dos. Los polinomios están modelizados en una lista de coeficientes (ordenados de mayor a menor). El predicado mult multiplica un polinomio por un escalar. Para multiplicar dos polinomios hemos tenido que realizar varios predicados auxiliares que nos permitiesen realizar la multiplicación de forma correcta. Uno es para rellenar con el número necesario de ceros por la izquierda para poder multiplicar los términos del grado adecuado. Para realizar este hemos recurrido a su vez a un predicado auxiliar que añade un número n veces a la izquierda de una lista. El código de ambos es el siguiente:

```
repeat(_,0,[]).
repeat(Elem,1,[Elem]).
repeat(Elem,N,[Elem|Xs]):-
  N>0,
  N1 is N-1,
  repeat(Elem,N1,Xs).
```

```
rellena0(X,0,X).
rellena0(X,N,Z):-
  N>0,
  repeat(0,N,S),
  append(S,X,Z).
```

Usando el predicado para rellenar ceros ya podemos realizar el que buscábamos que se encarga de multiplicar polinomios. Para simplificar el código hemos decidido realizar otro predicado auxiliar que dados el grado y el coeficiente de una variable la multiplica por un polinomio. Con este predicado y el de suma que hicimos al inicio es muy sencillo realizar el de la multiplicación de dos polinomios cualesquiera. Ambos códigos se especifican a continuación:

```
mult(_,_,[],[]).
mult(_,0,_,[]).
mult(G,Coef,X,Z):-
  mult(Coef,X,S1),
  rellena0(S1,G,Z).
```

```
multPol(_,[],[]).
multPol([],_,[]).
multPol(X,Y,Z):-
  multPol(0,X,Y,Z).
multPol(_,[],_,[]).
```

Con todos estos predicados auxiliares ya podemos realizar uno que nos permita calcular potencias de un polinomio cualquiera. Como nuestro problema simplificado es conocer los coeficientes del polinomio x+2 elevado a un número cualquiera, este predicado nos resuelve el problema.

```
potencia(_,0,[1]).
potencia([],_,[]).
potencia(X,N,Xn):-
       N>0,
   mult(1,X,X1),
     N1 is N-1,
potencia(X1,N1,Xn2),
multPol(X1,Xn2,Xn).
```

```
   hipercubo(N,L):-
potencia([1,2],N,L), !.
```

Podríamos haber realizado una aproximación menos general al problema y habernos centrado en el caso particular de los coeficientes de x+2 (mediante el uso de la pirámide de Pascal y de los números combinatorios). Sin embargo nos ha parecido más útil desarrollar todo el mecanismo para operar entre polinomios ya que de esta forma podemos reutilizar parte del código en otros problemas que se nos puedan plantear en el futuro y requieran del uso de polinomios.

A continuación vamos a mostrar la consulta necesaria para ver los elementos del hipercubo de dimensión 12 y el resultado obtenido.

```
X = [1,24,264,1760,7920,25344,59136,101376,
     126720,112640,67584,24576,4096].
```

SEGUNDA PARTE: DETECTANDO CICLOS EN GRAFOS

Enunciado

Programar un predicado hay_ciclo/2 para detectar la existencia de ciclos en grafos dirigidos.

hay_ciclo(Nombre, Recorrido): Nombre es el nombre asignado a una representación del grafo (que vendrá dado en la llamada) y Recorrido es una lista con nodos del grafo que forman un camino que contiene un ciclo (el camino que forma ciclo es una sublista de Recorrido). El predicado falla si no hay ninguna solución.

Los posibles grafos a los que se aplica el programa se almacenan mediante hechos del predicado grafo/2, de la forma grafo(Nombre, Grafo)

Ejemplo 1:

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

En este ejemplo s*eta* es el nombre del grafo y cada arista se ha representado mediante una estructura l/2 cuyo primer argumento es el nodo origen de la arista, y cuyo segundo argumento es el nodo destino.

```
?- hay_ciclo(seta, R).
   R = [b,c,b] ? ;
   R = [a,b,c,b] ?
         yes
```

Nota: La segunda solución contiene un nodo que no es parte del ciclo hallado, pero es válida porque sí contiene al ciclo.

Ejemplo 2:

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

?- hay_ciclo(t, R).
        no
```

Resolución

Para ver si hay ciclos vamos a usar un único predicado auxiliar que nos dará los ciclos que hay en nuestro grafo desde un vértice determinado. Después simplemente usaremos este predicado para preguntar si desde cualquier vértice hay ciclo resolviendo el problema.

El mayor problema es, por tanto, realizar el predicado auxiliar.

A partir de un vértice vamos a ir a sus adyacentes y comprobaremos si estos ya están en una lista auxiliar que tendremos de vértices visitados. Si no lo hemos visitado añadimos este vértice a la lista de visitados e iteramos. En caso de que algún vértice al que lleguemos ya esté en la lista de los vértices visitados terminamos devolviendo la lista auxiliar, que será la lista que contendrá el grafo con el ciclo.

El código de este predicado auxiliar es el siguiente:

```
cicloX(G,X,Z):-
cicloX(G,X,[X],Z).

cicloX(G,X,Zs,Z):-
    grafo(G,A),
 member(l(X,W),A),
   member(W,Zs),
 append(Zs,[W],Z).
```

```
cicloX(G,X,Zs,Z):-
    grafo(G,A),
 member(l(X,W),A),
not(member(W,Zs)),
append(Zs,[W],Z1),
 cicloX(G,W,Z1,Z).
```

En el predicado cicloX de aridad 3 los parámetros son el grafo, el vértice inicial y la lista del grafo con el ciclo. Lo único que hace es llamar a cicloX de aridad 4 en donde se añade la lista auxiliar de vértices visitados, que al inicio sólo contendrá al vértice inicial.

A partir de este predicado sólo tenemos que realizar la comprobación desde cualquier vértice de nuestro grafo.

```
Hay_Ciclo(G,R):-
 cicloX(G,_,R).
```

TERCERA PARTE: DECODIFICANDO LOS GENES

Se trata de resolver un problema relacionado con la búsqueda de la secuencia genética de una cadena de ARN: identificar aminoácidos dentro de una cadena. Las cadenas de ARN son secuencias de nucleótidos y cada tres nucleótidos codifican un aminoácido, componente básico de las proteínas. Se deben identificar las tripletas que codifican aminoácidos dentro de una secuencia de nucleótidos.

aminoacidos(ARN,Cadenas): la secuencia de nucleótidos ARN (que vendrá dada en lallamada) codifica las secuencias de aminoácidos Cadenas (en el mismo orden).

Existen cuatro nucleótidos diferentes: uracilo, citosina, adenina y guanina. Los abreviaremos por su primera letra: u, c, a, g. Existen 20 aminoácidos, que se abrevian a sus tres primeras letras en inglés: ala, arg, asn, asp, cys, gln, glu, gly, his, ile, leu, lys, met, phe, pro, ser, thr, trp, tyr, val. Las cadenas de ARN estarán representadas por “strings” de letras (de los aminoácidos); las cadenas de aminoácidos por secuencias de los nombres separados por guiones. Las cadenas identificadas estarán en una lista (que conservará el orden en que se encuentran en la cadena de ARN).

Nota: Un string es la lista de códigos ascii de las letras que lo forman, de maneraque "acgu" es idénticamente igual a [97,99,103,117].

Dado que hay 4^3 = 64 posibles tripletas y solo 20 aminoácidos (más un código de parada), existe redundancia. Esto es, casi todos los aminoácidos se pueden codificar con varias tripletas diferentes. En muchos casos el último nucleótido de la tripleta es indistinto. A continuación utilizaremos números para abreviar varias distintas tripletas que codifican un mismo aminoácido. Los siguientes números representarán una cualquiera de las letras de su lista:

1 = [uc] 2 = [ag] 3 = [uca] 4 = [ucag]

La codificación de aminoácidos por nucleótidos es prácticamente universal, aunque actualmente se sabe que hay excepciones. La siguiente lista indica la codificación genética más común (stop es el código de parada). Se utilizan los números anteriores para representar distintas secuencias de nucleótidos posibles (por lo que en realidad la lista contiene todas las 64 tripletas posibles).

ala = gc4 arg = ag2 arg = cg4 asn = aa1 asp = ga1 cys = ug1 gln = ca2

glu = ga2 gly = gg4 his = ca1 ile = au3 leu = cu4 leu = uu2 lys = aa2

met = aug phe = uu1 pro = cc4 ser = ag1 ser = uc4 stop = ua2 stop = uga

thr = ac4 trp = ugg tyr = ua1 val = gu4

Dada una cadena de ARN, la secuencia de tripletas que generan aminoácidos empieza siempre con la tripleta que se corresponde con el aminoácido metionina (met), a continuación siguen las tripletas que codifican los aminoácidos, que acaban con una tripleta que codifica el fin de la cadena (stop).

Para complicar las cosas, no existen marcas en la cadena de ARN que indiquen donde empieza cada tripleta. Además, las secuencias de nucleótidos que generan aminoácidos pueden estar separadas por un número indeterminado de nucleótidos de composición aleatoria (aparentemente).

Resolución

Lo primero que hemos tenido que hacer ha sido hacer un predicado que detecte a qué aminoácido corresponde cada tripleta de nucleótidos. Dados tres nucleótidos nuestro predicado unificará esta tripleta con el nombre de cada aminoácido. A continuación detallamos el código empleado en hace este *diccionario* de aminoácidos.

```
aminoacido(X,Z):-
      arg(1,a4("ucag"),W),
      member(Y,W),
      X==[103,99,Y],
      append([ala],[],Z).

aminoacido(X,Z):-
      arg(1,a2("ag"),W),
      member(Y,W),
      X==[103,97,Y],
      append([glu],[],Z).

aminoacido(X,Z):-
      X==[97,117,103],
      append([met],[],Z).

aminoacido(X,Z):-
      arg(1,a4("ucag"),W),
      member(Y,W),
      X==[97,99,Y],
      append([thr],[],Z).

aminoacido(X,Z):-
      arg(1,a1("uc"),W),
      member(Y,W),
      X==[103,97,Y],
      append([asp],[],Z).
```

```
aminoacido(X,Z):-
      arg(1,a4("ucag"),W),
      member(Y,W),
      X==[99,117,Y],
      append([len],[],Z).

aminoacido(X,Z):-
      arg(1,a2("ag"),W),
      member(Y,W),
      X==[117,97,Y],
      append([stop],[],Z).

aminoacido(X,Z):-
      arg(1,a2("ag"),W),
      member(Y,W),
      X==[97,103,Y],
      append([arg],[],Z).

aminoacido(X,Z):-
      arg(1,a4("ucag"),W),
      member(Y,W),
      X==[103,103,Y],
      append([gly],[],Z).

aminoacido(X,Z):-
      arg(1,a1("uc"),W),
      member(Y,W),
      X==[117,117,Y],
      append([phe],[],Z).
```

```
aminoacido(X,Z):-
      X==[117,103,103],
      append([trp],[],Z).

aminoacido(X,Z):-
      arg(1,a2("ag"),W),
      member(Y,W),
      X==[99,97,Y],
      append([gln],[],Z).

aminoacido(X,Z):-
      arg(1,a2("ag"),W),
      member(Y,W),
      X==[97,97,Y],
      append([lys],[],Z).

aminoacido(X,Z):-
      X==[117,103,97],
      append([stop],[],Z).

aminoacido(X,Z):-
      arg(1,a4("ucag"),W),
      member(Y,W),
      X==[99,103,Y],
      append([arg],[],Z).

aminoacido(X,Z):-
      arg(1,a1("uc"),W),
      member(Y,W),
      X==[99,97,Y],
      append([his],[],Z).

aminoacido(X,Z):-
      arg(1,a4("ucag"),W),
      member(Y,W),
      X==[99,99,Y],
      append([pro],[],Z).

aminoacido(X,Z):-
      arg(1,a1("uc"),W),
      member(Y,W),
      X==[117,97,Y],
      append([tyr],[],Z).

aminoacido(X,Z):-
      arg(1,a4("ucag"),W),
      member(Y,W),
      X==[117,99,Y],
      append([ser],[],Z).


aminoacido(X,Z):-
      arg(1,a1("uc"),W),
      member(Y,W),
      X==[117,103,Y],
      append([cys],[],Z).

aminoacido(X,Z):-
      arg(1,a2("ag"),W),
      member(Y,W),
      X==[117,117,Y],
      append([leu],[],Z).

aminoacido(X,Z):-
      arg(1,a1("uc"),W),
      member(Y,W),
      X==[97,97,Y],
      append([asn],[],Z).

aminoacido(X,Z):-
      arg(1,a3("uca"),W),
      member(Y,W),
      X==[97,117,Y],
      append([ile],[],Z).

aminoacido(X,Z):-
      arg(1,a1("uc"),W),
      member(Y,W),
      X==[97,103,Y],
      append([ser],[],Z).

aminoacido(X,Z):-
      arg(1,a4("ucag"),W),
      member(Y,W),
      X==[103,117,Y],
      append([val],[],Z).
```

Una vez que podemos traducir los nucleótidos a aminoácidos tenemos que leer las cadenas de aminoácidos. Para ello hemos realizado un predicado comienza que usará otro auxilar llamado nPrimeros. Este predicado auxliar lo usaremos para obtener los n primeros caracteres de la cadena de aminoácidos que tengamos, pues necesitamos tratar los nucleótidos de tres en tres. Además también nos servirá para obtener una lista sin estos elementos que hemos quitado (para el caso n=1 esta lista estará vacía pero nos es indiferente ya que sólo usaremos este predicado con n=3).
Comienza tomará estos tres primeros nucleótidos identificados por sus respectivos caracteres y comprobará si estamos ante el aminoácido met (el de inicio). En caso de que no estemos ante un met seguiremos buscando en la cadena de aminoácidos. Para ello nos quedamos con la lista sin su cabeza y volvemos a obtener los tres primeros elementos de nuestra nueva lista. De esta forma recorrerá la lista hasta encontrar el aminoácido met. En caso de no encontrarlo devolverá false. Este es el motivo de que todas las consultas terminen en false.

```
nPrimeros(1,[X|_],[],[X]).

nPrimeros(N,[X|Xs],Ys,Y):-
N>1,
N1 is N-1,
nPrimeros(N1,Xs,_,Y2),
append([X],Y2,Y),
append(Y2,Ys,Xs).
```

```
comienza([X|Xs],Y):-
nPrimeros(3,[X|Xs],Ys,Y2),
aminoacido(Y2,[met]),
append(Ys,[],Y).

comienza([X|Xs],Y):-
nPrimeros(3,[X|Xs],_,Y2),
Y2\=="aug",
comienza(Xs,Y).

comienza([X|Xs],Y):-
nPrimeros(3,[X|Xs],_,Y2),
aminoacido(Y2,[met]),
comienza(Xs,Y).
```

El predicado auxiliar principal, en el que llevamos a cabo la mayor parte de las acciones, es aminoacidos2. Este predicado obtiene de nuestra lista de nucleótidos otra lista con los aminoácidos hasta que encuentra un stop. Además como puede saltarse nucleótidos se han tenido en cuenta también esos casos. A continuación podemos ver el código que es idéntico para los tres casos, exceptuando el primer parámetro. También nos devolverá una lista de los nucleótidos que aún quedan después del stop.

```
aminoacidos2(X,Y,Z):-
      nPrimeros(3,X,Y,Y2),
      aminoacido(Y2,[stop]),
      append([],[],Z).
```

```
aminoacidos2([_|Xs],Y,Z):-
      nPrimeros(3,Xs,Y,Y2),
      aminoacido(Y2,[stop]),
      append([],[],Z).


aminoacidos2([_,_|Xs],Y,Z):-
      nPrimeros(3,Xs,Y,Y2),
      aminoacido(Y2,[stop]),
      append([],[],Z).
```

```
aminoacidos2(X,Y,Z):-
      nPrimeros(3,X,Ys,Y2),
      aminoacido(Y2,Zs),
      not(member(stop,Zs)),
      aminoacidos2(Ys,Ws,Z2),
      append(Ws,[],Y),
      append(Zs,Z2,Z).
```

```
aminoacidos2([_|Xs],Y,Z):-
      nPrimeros(3,Xs,Ys,Y2),
      aminoacido(Y2,Zs),
      not(member(stop,Zs)),
      aminoacidos2(Ys,Ws,Z2),
      append(Ws,[],Y),
      append(Zs,Z2,Z).

aminoacidos2([_,_|Xs],Y,Z):-
      nPrimeros(3,Xs,Ys,Y2),
      aminoacido(Y2,Zs),
      not(member(stop,Zs)),
      aminoacidos2(Ys,Ws,Z2),
      append(Ws,[],Y),
      append(Zs,Z2,Z).
```

Con todos estos predicados auxiliares ya casi tenemos lo que nos pedían en el enunciado. Sólo falta poner las listas de aminoácidos en el formato adecuado que se nos indicaba. Para ello añadimos guiones entre cada aminoácido en el predicado unificaListas. Hay que tener en cuenta que la nueva lista con el formato correcto se nos dará invertida por lo que cuando la usemos habrá que hacer previamente un reverse si queremos que se conserve el orden.

```
unificaListas([X|[]],X).

unificaListas([X,Y|[]],Z):-
      Z=..['-',Y,X].

unificaListas([X|Xs],Y):-
      length(Xs,N),
           N>1,
   unificaListas(Xs,Z),
      Y=..['-',Z,X].
```

Ya sólo nos falta juntar todos los predicados auxiliares que tenemos en nuestro predicado aminoácidos que hará lo que se nos pedía. Se llamará a comienza para encontrar el met de inicio y después se obtendrá la cadena correspondiente con aminoacidos2. Tras esto se invertirá la cadena con un reverse para aplicar unificaListas y obtener la cadena en nuestro formato. Como queremos que después de un stop vuelva a comprobar si hay más cadenas se llamará recursivamente a aminoácidos con la cadena de nucleótidos restante que obtuvimos en la llamada a aminoacidos2. De esta forma sólo se detendrá cuando aminoacidos2 no encuentre más cadenas en cuyo caso devolverá false y se parará la ejecución. El resultado devuelto será una lista con las listas de aminoácidos encontradas.

También se contempla un caso base sin la llamada recursiva.

El código correspondiente al predicado aminoácidos es el siguiente:

```
aminoacidos(X,Z):-
   comienza(X,Y),
aminoacidos2(Y,W,Z2),
   reverse(Z2,Z3),
unificaListas(Z3,Z4),
 append([Z4],[],Z5),
  aminoacidos(W,Z6),
   append(Z5,Z6,Z).

aminoacidos(X,Z):-
   comienza(X,Y),
aminoacidos2(Y,_,Z2),
   reverse(Z2,Z3),
unificaListas(Z3,Z4),
  append([Z4],[],Z).
```