Programación declarativa

Práctica 1: Bits, Nibbles & Bytes

Autor:

Daniel Corrales Falcó; b190410; d.corralesf@alumnos.upm.es

Índice:

- Código:

```
:- module(_, _, [assertions, regtypes]).
author_data('Corrales', 'Falco', 'Daniel', 'B190410').

%Define a binary digit type.
bind(0).
bind(1).
%Define a binary byte as a list of 8 binary digits.
binary_byte([bind(B7),bind(B6),bind(B5),bind(B4),bind(B3),bind(B2),bind(B1),bind(B0)]
):-
   bind(B7),
   bind(B6),
   bind(B5),
   bind(B4),
   bind(B3),
   bind(B2),
   bind(B1),
   bind(B0).

%Define a hex (nibble) type.
hexd(0).
hexd(1).
hexd(2).
hexd(3).
hexd(4).
hexd(5).
hexd(6).
hexd(7).
hexd(8).
hexd(9).
hexd(a).
hexd(b).
hexd(c).
hexd(d).
hexd(e).
hexd(f).

%Define a hex byte as a list of two hex nibbles
hex_byte([hexd(H1),hexd(H0)]):-
   hexd(H1),
   hexd(H0).

%Define a byte type either as a binary byte or as a hex type.
byte(BB):-
   binary_byte(BB).
```

```
byte(HB):-
   hex_byte(HB).


%   PREDICADOS AUXILIARES

/*
   Para el Predicado 2 voy a necesitar una base de hechos, ya que mi intencion y
objetivo es comparar si un hexadecimal
   y un binario dados son equivalentes, para lo cual he de saber por que serie de
   binds que forman un bit representan un nibble hexadecimal.
*/

hex_to_bin_byte([hexd(0)], [bind(0),bind(0),bind(0),bind(0)]). %el 0
hex_to_bin_byte([hexd(1)], [bind(0),bind(0),bind(0),bind(1)]). %el 1
hex_to_bin_byte([hexd(2)], [bind(0),bind(0),bind(1),bind(0)]). %el 2
hex_to_bin_byte([hexd(3)], [bind(0),bind(0),bind(1),bind(1)]). %el 3
hex_to_bin_byte([hexd(4)], [bind(0),bind(1),bind(0),bind(0)]). %el 4
hex_to_bin_byte([hexd(5)], [bind(0),bind(1),bind(0),bind(1)]). %el 5
hex_to_bin_byte([hexd(6)], [bind(0),bind(1),bind(1),bind(0)]). %el 6
hex_to_bin_byte([hexd(7)], [bind(0),bind(1),bind(1),bind(1)]). %el 7
hex_to_bin_byte([hexd(8)], [bind(1),bind(0),bind(0),bind(0)]). %el 8
hex_to_bin_byte([hexd(9)], [bind(1),bind(0),bind(0),bind(1)]). %el 9
hex_to_bin_byte([hexd(a)], [bind(1),bind(0),bind(1),bind(0)]). %el 10
hex_to_bin_byte([hexd(b)], [bind(1),bind(0),bind(1),bind(1)]). %el 11
hex_to_bin_byte([hexd(c)], [bind(1),bind(1),bind(0),bind(0)]). %el 12
hex_to_bin_byte([hexd(d)], [bind(1),bind(1),bind(0),bind(1)]). %el 13
hex_to_bin_byte([hexd(e)], [bind(1),bind(1),bind(1),bind(0)]). %el 14
hex_to_bin_byte([hexd(f)], [bind(1),bind(1),bind(1),bind(1)]). %el 15


/*
   Para el predicado 4 voy a necesitar invertir la lista para poder poner el byte menos
significativo de la lista de bytes
   a la izqueirda y no a la derecha para poder ir sacando la cabeza y comprobar si es un
digito binario
*/
reverse([],[]).
reverse([X|Xs],Ys):-
   reverse(Xs,Zs),
   concat(Zs,[X],Ys).
/*
   Para poder hacer funcionar la funcion reverse, voy a necesitar hacer la funcion
append
*/
concat([],[Ys],[Ys]).
concat([X|Xs],Ys,[X|Zs]):-
   concat(Xs,Ys,Zs).
```

```
/*
   Para el predicado 4 tambien voy a necesitar un predicado auxiliar para asi no
encontrar fallos en el caso de querer obtener el
   primer elemento de primeras
*/
get_nth_bit_from_byte_2(0, [Bn|_],Bn).          %caso base
get_nth_bit_from_byte_2(s(N), [X|R], BN):-
   get_nth_bit_from_byte_2(N, R, BN).          %caso base

/*
   Para el predicado 7 voy a necesitar saber la tabla de XOR para poder saber los
resultados de las operaciones
   seguramente esté mal y haya que borrarlo
*/
table_xor(bind(1),bind(1),bind(0)).
table_xor(bind(1),bind(0),bind(1)).
table_xor(bind(0),bind(1),bind(1)).
table_xor(bind(0),bind(0),bind(0)).

/*
   Para el predicado 7 necsito tambien un predicado auxiliar para comprobar
individualmete bit por bit si cumple la tabla
   xor
*/
check_xor([],[],[]).                        %caso base
check_xor([Z|Z1],[X|X1],[Y|Y1]):-
   table_xor(Z,X,Y),
   check_xor(Z1,X1,Y1).
%PREDICADOS!

%Predicado 1 -> byte_list/
/*
   Lo que queremos es comprobar si los elementos de una lista dada son de tipo byte,
por lo que una funcion
   recursiva es una buena opcion, ya que asi, lo que puedo hacer es comprobar cada
elemento de manera individual
   llamando a mi predicado byte/1
*/
byte_list([]).                      %caso base
byte_list([B|L]):-
   byte(B),
   byte_list(L).

%Predicado 2 -> byte_conversion/2
/*
   En este predicado, mi objetivo es comprobar si un hexadecimal y un binario dados
representan el mismo numero
```

```
   para esto lo que voy a hacer es comprobar si el primer nibble del byte esta
represetnado por los primeros 4 bind
   de la cadena de 8 bits y el segundo nibble con los otros 4 siguientes. Esto gracias a la
base de hechos que he conformado
   mas arriba
*/
byte_conversion([H1,H0], [B7,B6,B5,B4,B3,B2,B1,B0]):-
   hex_byte([H1,H0]),                    %compruebo si H1-H0 representan un byte
hexadecimal
   binary_byte([B7,B6,B5,B4,B3,B2,B1,B0]),    %compruebo si B7-B0 representan un
byte binario
   hex_to_bin_byte([H1],[B7,B6,B5,B4]),       %compruebo si son equivalentes las
respectivas mitades
   hex_to_bin_byte([H0],[B3,B2,B1,B0]).

%Predicado 3 -> byte_list_conversion/2
/*
   En este predicado, mi objetivo es comparar si dos listas de bytes, una hex y otra bin,
representan lo mismo
   este predicado hara uso del predicado 1 ya que este lo que hace es compararme un
hex y un bin,
   las llamadas al mismo se realizaran con los primeros elementos de cada lista.
Ademas este predicado
   sera recursivo para asi poder ir vaciando las listas dadas y así poder comprobar
todos los elementos.
*/
byte_list_conversion([],[]).                %caso base
byte_list_conversion([H|Hn],[B|Bn]):-
   byte_conversion(H,B),
   byte_list_conversion(Hn,Bn).

%Predicado 4 -> get_nth_bit_from_byte/3
/*
   En este predicado, el objetivo es saber si el bit en la posicion N de una cadena dada
es igual a un bit dicho
   para esto, como dicen en  el enunciado, hemos de tener en cuenta que tendremos
que invertir la cadena de bytes dada
   por el tema de los indices. Como este es a su vez un predicado polimórfico,
habremos de introducir comprobaciones para ver en que caso estamos
   ya que podemos tener cadenas de HEX y de BIN. Una vez hecho todo esto, por
temas de eficiencia he creado un predicado auxiliar
   recursivo en el cual realizo la busqueda y comparacion del elemento(este necesita
operar ya con la cadena invertida).
*/
%Caso Hexadecimal
get_nth_bit_from_byte(N, Hb, BN):-
   hex_byte(Hb),
   byte_conversion(Hb, Bb),%no se si necesario
```

```
   reverse(Bb, RH),
   get_nth_bit_from_byte_2(N, RH, BN).

%Caso binario
get_nth_bit_from_byte(N, Bb, BN):-
   binary_byte(Bb),
   reverse(Bb,RB),
   get_nth_bit_from_byte_2(N, RB, BN).

%Predicado 5 -> byte_list_clsh/2


%Predicado 6 -> byte_list_crsh/2


%Predicado 7 -> byte_xor/3
/*
   EN este predicado, el objetivo es hallar el xor entre dos cadenas de bits, como
queremos que sea funcional para
   cadenas tanto hexadecimales como binarias lo pirmero que hacemos en cada
predicado es comprobar cual de las dos opciones es,
   a continuacion si es hexadecimal lo convierto a binario y a partir de aqui ambos se
resuelven de la misma forma,
   llamo a mi predicado auxiliar recursivo que comprueba las operaciones xor
*/
%Caso hexadecimal
byte_xor([],[],[]).                    %caso base
byte_xor(Xs,Ys,Zs):-
   hex_byte(Xs),
   hex_byte(Ys),
   hex_byte(Zs),
   byte_conversion(Xs,XB),
   byte_conversion(Ys,YB),
   byte_conversion(Zs,ZB),
   check_xor(ZB,XB,YB).

%Caso binario
byte_xor(Xs,Ys,Zs):-
   binary_byte(Xs),
   binary_byte(Ys),
   binary_byte(Zs),
   check_xor(ZB,XB,YB).
```

- Explicación de los métodos:

  Cada método implementado ya se encuentra explicado en el propio código con comentarios, sin embargo, voy a realizar una explicación más a fondo en este documento.

  - Predicado 1 → byte_list/1

    El objetivo de este predicado es comprobar si un elemento es un byte. Para esto lo más eficiente o sencillo era hacer uso de los elementos y hechos proporcionados por el mismo enunciado. En resumen, para poder comprobar que todos los elementos de una lista conforman un byte, lo más eficaz es realizar un predicado recursivo mediante el cual ir sacando de uno en uno los elementos de la lista proporcionada, llamar al predicado byte/1 dado en el enunciado y finalmente volver a llamar a este predicado hasta que la lista se vacíe.

  - Predicado 2 → byte_conversion/2

    En este predicado el objetivo es comprobar si un hexadecimal está representando el mismo valor que un byte. Para esta funcionalidad he llegado a la conclusión de que la manera más eficaz es creando una base de hechos la cual me permita saber las equivalencias entre nibbles hexadecimales y bits. Una vez creada (hex_to_bin_byte/2) he de comprobar que las listas que estoy recibiendo son efectivamente un hexadecimal y un byte (haciendo uso de las funciones proporcionadas por el enunciado) y finalmente, como un hexadecimal son 2 nibbles y cada nibble se representa por 4 bits, del hexadecimal que me pasan habré de comparar el primer nibble con los 4 primeros bits del byte y el segundo con los 4 últimos, esto funciona ya que ambos se representan con el bit de menor peso a la derecha de todo.

- Predicado 3 → byte_list_conversion/2

  Este predicado es muy similar al anterior, solo que en este caso no es con solo un hex y un bin, si no que tengo dos listas. Gracias al predicado 2 este predicado se resuelve de forma muy simple creando este como un predicado recursivo. Básicamente lo que hay que hacer pasarle a este predicado las listas con sus respectivas cabeceras, llamar al predicado 2 (byte_conversion), para que compruebe los 2 primeros elementos de cada lista y finalmente llamar de nuevo a byte_list_conversion/2 con el resto de las listas, hasta que las listas de hex y de bin estén vacías. No es necesario usar ningún predicado auxiliar

- Predicado 4 → get_nth_bit_fro_byte/3

  Este predicado ha de ser polimórfico debido a que dependiendo de si recibe un hexadecimal o un byte hará cosas distintas. Sin embargo, la única diferencia que tienen en mi implementación es que al principio del predicado para hexadecimales transformo este hexadecimal a binario y a partir de aquí resuelvo ambos casos de la misma forma. Esta forma se basa en llamar a un predicado el cual me revertirá la cadena de bytes que le pasan (como el método visto en clase), esto es necesario por cómo están estructurados los binarios (aviso escrito en el enunciado), ya que en estos el índice 0 corresponde al bit de mayor valor cuando lo que queremos es que sea al revés, que el índice 0 equivalga al de menor valor. A continuación, llamo a un predicado recursivo auxiliar (get_nth_bit_from_byte_2/3), este cumple la función de un bucle hasta que el número de peano sea 0 lo que hace es ir sacando las cabeceras, lo cual dejará en la cabecera el elemento en la posición indicada por el número de peano.
- Predicado 5 → No implementado
- Predicado 6 → No implementado

- Predicado 7 → byte_xor/3

  En este predicado lo que nos piden es comprobar si el tercer elemento que nos pasan es el resultado de la operación XOR entre los dos primeros. Para resolverlo, voy a seguir una lógica parecida a la que he usado en el predicado 4, este predicado ha de ser polimórfico, para hex y bin, solo que en la implementación para hexadecimales, lo que voy a hacer será transformarlo primero a bin y luego resolver ambos de la misma forma(con un predicado recursivo). La manera en que he resuelto es la siguiente, en ambos casos primero compruebo que las listas que me pasan son correctas y llamo a mi predicado recursivo auxiliar (check_xor/3) , a este le paso 3 listas y sus cabeceras, dentro del mismo compruebo, con otra base de hechos que he construido que representa la tabla lógica del XOR (table_xor), si la operación XOR entre los 2 primeros bits es correcta y finalmente llamo otra vez a check_xor con el resto de las listas, así hasta que estas estén vacías.

- Pruebas:

  - Predicado 1:

    ?- byte_list (L).

    L = [] ? ;
    L = [[bind (1) , bind (0) , bind (1) , bind (0) , bind (0) , bind (0) ,
    bind (0) , bind (0)]] ? ;
    L = [[bind (1) , bind (1) , bind (1) , bind (1) , bind (1) , bind (1) ,
    bind (1) , bind (1)]] ? ;

    yes

  - Predicado 2:
    ?- byte_conversion (H, B).
    B = [bind (0) , bind (0) , bind (1) , bind (0) , bind (1) , bind (0) , bind (1) , bind (1)] ? ;
    H = [hexd(2),hexd(b)] ? ;

    yes

  - Predicado 3:
    ?- byte_list_conversion (HL, BL).

    BL = [[bind(0),bind(1),bind(0),bind(1),bind(1),bind(0),bind(1),bind(0)],[bind(1),bind(0),bind(1),bind(1),bind(0),bind(0),bind(1),bind(0)]] ? ;

    HL =[[hexd(5), hexd(a)],[hexd(b),hexd(2)]] ? ;

    yes

  - Predicado 4:

    ?- get_nth_bit_from_byte ((s(0)), [bind (0) , bind (0) , bind (1) , bind (0) , bind (0) , bind (0) , bind (0) , bind (1)], B).

    B = bind (1) ? ;

    yes

    ?- get_nth_bit_from_byte (s(s(s(s(0)))), [hexd(5), hexd(a)], B).

    B = bind (1) ? ;

    yes

- Predicado 7:
  ?-
  byte_xor([bind(0),bind(0),bind(1),bind(0),bind(1),bind(1),bind(1),bind(0)],[bind(0),bind(1),bind(1),bind(0),bind(1),bind(0),bind(0),bind(1)], B3).

  B3 = [bind(0),bind(1),bind(0),bind(0),bind(0),bind(1),bind(1),bind(1)] ? ;

  Yes

  ?- byte_xor([hexd (b) ,hexd(2)],[hexd (5) ,hexd (a)], B3).

  B3 = [hexd (8) ,hexd (e)] ? ;

  yes