Primera práctica (Programación Lógica Pura)

Fecha de entrega: ~~12~~19 de abril de 2022

BITS, NIBBLES & BYTES

Antes de empezar la práctica es **muy importante** empezar por leer las **Instrucciones generales para la realización y entrega las prácticas**, así como la **Preguntas frecuentes sobre Ciao, Deliverit, LPdoc, etc.**, disponibles en Moodle.

Instrucciones específicas

En esta práctica el alumno repasará los conceptos fundamentales estudiados en el bloque temático de programación lógica pura. Por ello, además de las instrucciones generales anteriores, como instrucciones específicas para esta práctica se recuerda que el único recurso que podrá utilizarse para el desarrollo de la práctica es **la unificación**. **No está permitido** utilizar ningún recurso proporcionado por ISO Prolog (p.ej., aritmética, predicados metalógicos, predicados de inspección de estructuras, corte, negación por fallo, etc.). Tampoco está permitido utilizar ningún predicado predefinido de Prolog (p.ej., predicados sobre listas, árboles, etc.). Si fuese necesario utilizar alguno, el alumno debe programarlo por si mismo. Por ejemplo, si se necesitase definir la pertenencia a una lista mediante el predicado `member/3`, el alumno deberá programarse una versión propia del predicado `member/3` que deberá denominar con un nombre diferente (p.ej., `my_member/3`) para evitar el uso por defecto del predicado predefinido `member/3`. Se deberá tener especial cuidado con esto, dado que algún predicado predefinido como `length/2` (que calcula la longitud de una lista) utiliza aritmética de ISO Prolog en lugar de aritmética de Peano (que es la que puede y debe utilizarse en este ejercicio).

Enunciado

La presente práctica tiene como objetivo repasar los conceptos básicos estudiados en clase sobre listas, estructuras y recursividad. Para ello, se pedirá al alumno que escriba una serie de predicados que operan a nivel de bits, nibbles y bytes. Dado que en Prolog puro no tenemos soporte nativo para realizar estas tareas, será necesario definir tipos y operaciones para ello. Por ejemplo, un tipo de datos esencial sería el bit. Definiremos los bits utilizando la estructura `bind/1` que representa un dígito binario. Nótese que, aunque utilizaremos constantes numéricas como 1, 2, 5, etc., el alumno debe tener claro que **no representan expresiones numéricas, sino constantes**.

Los bits se representan fácilmente utilizando un predicado:

```
% Define a binary digit type.
bind(0).
bind(1).
```

Una vez definidos los bits, es momento de definir un tipo para los bytes. Los bytes, que son octetos de dígitos binarios, pueden representarse como una lista de dígitos binarios de la manera siguiente:

```
% 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).
```

Se entiende que en la definición de un byte como una lista de bits, el primer elemento de la lista es el bit más significativo, mientras que el último elemento de la lista sería el bit menos significativo.

Dado que durante la realización de los programas se utilizarán listas de bytes que pueden ser relativamente largas, definiremos un nuevo tipo de datos que permitirá representar los bytes en base hexadecimal. Este tipo de datos es el dígito hexadecimal (también llamado nibble), que puede definirse fácilmente mediante un predicado:

```
% Define a hex digit (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).
```

Dado que un nibble representa una agrupación de 4 bits, es posible representar un byte (octeto) mediante una lista que contenga 2 nibbles. Por tanto, usando las definiciones anteriores es posible definir un byte de la siguiente forma:

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

```
% 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 byte as a list of 2 hex nibbles.
hex_byte([hexd(H1), hexd(H0)]) :-
    hexd(H1),
    hexd(H0).
```

Se entiende que en la definición de un byte como una lista de nibbles, el primer elemento de la lista es el nibble más significativo, mientras que el último elemento sería el nibble menos significativo.

Nótese de nuevo, que, aunque en estas definiciones se utilizan números, puede observarse que en ningún momento se tratan como tales, sino como meras constantes dado que no se lleva a cabo ningún procesamiento aritmético con ellos. Cualquier procesamiento que el alumno desee realizar debe hacerse utilizando los **axiomas de peano**. Todos los predicados que necesiten iterar deben realizarse de **forma recursiva**.

En base a estas definiciones se pide al alumno implementar los predicados siguientes:

Predicado 1 (1 puntos) `byte_list/1:` `byte_list(L)`

Este predicado es cierto si la lista dada en el primer argumento es una lista de bytes (ya sean binarios o hex). Se asume que el primer elemento de la lista es el bit más significativo, mientras que el último elemento de la lista sería el bit menos significativo.

`Ejemplo:` Generación de listas de bytes.

```
?- byte_list(L).

L = [] ? ;

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

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

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

yes
```

Predicado 2 (1.5 puntos) `byte_conversion/2:` `byte_conversion(HexByte, BinByte).`

Este predicado es cierto si el byte hexadecimal que aparece en el primer argumento tiene como representación binaria el byte binario que aparece en el segundo argumento.

`Ejemplo:` Conversión del byte 0xA1 en su representación binaria 0b10100001.

```
?- byte_conversion([hexd(a), hexd(1)], B).

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

no
```

Predicado 3 (1 puntos) `byte_list_conversion/2:` `byte_list_conversion(HL, BL).`

Este predicado es cierto si la representación binaria de la lista de bytes hexadecimales que aparece en el primer argumento es la lista de bytes que aparece en el segundo argumento.

`Ejemplo:` Cálculo de la representación binaria de la lista de bytes hexadecimales [0x35, 0x4E].

```
?- byte_list_conversion([[hexd(3), hexd(5)], [hexd(4),
   hexd(e)]], BL).

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

no
```

Predicado 4 (1 puntos) `get_nth_bit_from_byte/3:` `get_nth_bit_from_byte(N, B, BN).`

Este predicado **polimórfico** es cierto si BN es el dígito binario (bit) número N (**Ojo**: n es un número de peano) del byte B (ya sea este un byte hexadecimal o binario). **Nota:** el alumno debe tener en cuenta que el índice del bit menos significativo de un byte no es 1, sino 0.

`Ejemplo:` Extraer el bit número 5 del byte 0b10101100.

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

B = bind(1) ? ;

no
```

`Ejemplo:` Extraer el bit número 5 del byte 0xAC.

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

B = bind(1) ? ;

no
```

Nota: en los próximos tres predicados (5, 6, y 7) es suficiente con que el predicado se ejecute correctamente en el *modo* del ejemplo, es decir, que el último argumento sea de salida y el(los) primero(s) de entrada.

Predicado 5 (1.5 puntos) `byte_list_clsh/2:` `byte_list_clsh(L, CLShL).`

Este predicado **polimórfico** es cierto si CLShL es el resultado de efectuar un desplazamiento circular hacia la izquierda de la lista de bytes representada por L. Este predicado debe funcionar tanto para listas de bytes hexadecimales como binarias, aunque ambos argumentos deben estar representados **en la misma notación**. En los desplazamientos circulares a la izquierda el bit más significativo del byte más significativo de la lista L pasa a ser el bit menos significativo del byte menos significativo de la lista CLShL.

Ejemplo: Efectuar un desplazamiento circular hacia la izquierda de la lista de bytes [0x5A, 0x23, 0x55, 0x37].

```
?- byte_list_clsh([[hexd(5), hexd(a)], [hexd(2), hexd(3)],
   [hexd(5), hexd(5)], [hexd(3), hexd(7)]], L).

L = [[hexd(b), hexd(4)], [hexd(4), hexd(6)], [hexd(a),
   hexd(a)], [hexd(6), hexd(e)]] ? ;

no
```

Predicado 6 (1.5 puntos) `byte_list_crsh/2:` `byte_list_crsh(L, CRShL).`

Este predicado **polimórfico** es cierto si CRShL es el resultado de efectuar un desplazamiento circular hacia la derecha de la lista de bytes representada por L. Este predicado debe funcionar tanto para listas de bytes hexadecimales como binarios, aunque ambos argumentos deben estar representados **en la misma notación**. En los desplazamientos circulares a la derecha el bit menos significativo del byte menos significativo de L pasa a ser el bit más significativo del byte mas significativo de la lista CRShL.

Ejemplo: Efectuar un desplazamiento circular hacia la derecha de la lista de bytes [0xB4, 0x46, 0xAA, 0x6E].

```
?- byte_list_crsh([[hexd(b), hexd(4)], [hexd(4), hexd(6)],
   [hexd(a), hexd(a)], [hexd(6), hexd(e)]], L).

L = [[hexd(5), hexd(a)], [hexd(2), hexd(3)], [hexd(5),
   hexd(5)], [hexd(3), hexd(7)]] ? ;

no
```

Predicado 7 (1 puntos) `byte_xor/3:` `byte_xor(B1, B2, B3).`

Este predicado **polimórfico** es cierto si B3 es el resultado de efectuar la operación lógica XOR entre los bytes B1 y B2. Este predicado debe funcionar tanto para bytes binarios como hexadecimales, aunque todos los argumentos deben estar representados **en la misma notación**.

`Ejemplo:` XOR entre los bytes 0x5A y 0x23.

```
?- byte_xor([hexd(5),hexd(a)],[hexd(2),hexd(3)], B3).

B3 = [hexd(7),hexd(9)] ? ;

no
```

MEMORIA/MANUAL (1.5 puntos, fichero `codigo.pdf`)

PUNTOS ADICIONALES (para subir nota):

- Ejercicio complementario en '*programación alfabetizada*' ('literate programming'):

  Se darán puntos adicionales a las prácticas que realicen la documentación de dichos predicados insertando en el código aserciones y comentarios del lenguaje Ciao Prolog, y entregando como memoria o parte de ella el manual generado automáticamente a partir de dicho código, usando la herramienta `lpdoc` del sistema Ciao. En este caso se puede entregar este documento como memoria, y se recomienda por tanto que escribir este manual de forma que incluye el contenido que se solicita para la memoria.

- Ejercicio complementario en *codificación de casos de prueba*:

  También se valorará el uso de aserciones `test` que definan casos de prueba para comprobar el funcionamiento de los predicados. Estos casos de test se deben incluir en el mismo fichero con el código.

Hemos dejado en Moodle instrucciones específicas sobre cómo hacer todo esto y un ejemplo de código que tiene ya comentarios y tests, para practicar corriendo lpdoc sobre él y ejecutando los tests.