# **Segunda práctica** (ISO-Prolog)

Fecha de entrega: **21 de Mayo** de 2021

COMPRESIÓN DE SECUENCIAS

## Enunciado

Para comprimir secuencias de caracteres se suelen sustituir subsecuencias repetidas por el número de sus repeticiones. Así, la secuencia `aaaaaaa` se comprime en `a7`. La secuencia original es de longitud siete, la comprimida tiene solo longitud dos. La secuencia `ababab`, de longitud seis, se comprime en `(ab)3`, de longitud cinco. Nótese que los paréntesis se cuentan también como caracteres de la secuencia. Solo hacen falta paréntesis si la subsecuencia que se repite es de más de un carácter.

Para secuencias complejas se procede a dividirla en partes, que se comprimen por separado (y luego se unen). Así, el proceso completo de compresión se lleva a cabo bien por repetición o por división. En la compresión por repetición se localiza una subsecuencia que se repita un cierto número de veces, como ya se ha dicho, y el resultado es esta subsecuencia (posiblemente también comprimida) con su número de repeticiones. Por ejemplo, la secuencia `aaaaaaa` se comprime en la secuencia `a7`.

En la compresión por división se divide la secuencia original en dos partes que a su vez se comprimen por separado y se unen los resultados. Por ejemplo, la secuencia `aaaaaaabbbbbbb` se comprime en `a7b7` (comprimiendo cada parte, a su vez, por repetición). La secuencia `aaabaaab` se comprime por repetición en `(a3b)2` donde la subsecuencia aaab se ha comprimido en `a3b` por división (y a su vez `aaa` en `a3` por repetición).

Los resultados de la compresión (tanto de la secuencia original como de sus subsecuencias) han de ser más cortos que las secuencias iniciales. Así, no es admisible comprimir `aa` en `a2`, porque tienen la misma longitud, ni `abab` en `(ab)2`, porque esta última es más larga.

Las secuencias se representaran como listas de caracteres. Por ejemplo, `aaa` es `[a,a,a]` y `(ab)3` es `['(',a,b,')',3]`. En estas listas los números ocupan una única posición, tengan el numero de dígitos que tengan. Así, `a12` es la lista `[a,12]` y tiene longitud dos (no tres).

Los objetivos de la práctica son los siguientes:

1. Programar un predicado `division/2` con cabecera `division(Inicial, Comprimida)` que se verifica si la secuencia `Comprimida` es el resultado de comprimir la secuencia `Inicial` mediante división. Nótese que una solución válida para `Comprimida` no tiene que ser necesariamente la mejor en términos de longitud, aunque este programa debe ser capaz de devolver todas las soluciones, ya sea mediante backtracking o mediante el uso de predicados de agregación.

2. Programar un predicado `repeticion/2` con cabecera `repeticion(Inicial, Comprimida)` que se verifica si la secuencia `Comprimida` es el resultado de comprimir la secuencia inicial mediante repetición. En este caso aplican los mismos comentarios y recomendaciones que en el predicado anterior.

3. Programar un predicado `comprimir/2` con cabecera `comprimir(Inicial, Comprimida)` que se verifica si `Comprimida` es el resultado de comprimir la secuencia del primer argumento, según lo especificado más arriba. La secuencia original (primer argumento) viene dada en la llamada y en ella no aparecen ni paréntesis ni números. Para que la compresión sea eficaz, se deben probar una tras otra iterativamente sucesivas compresiones, hasta dar con la de menor longitud (es decir, realizando una búsqueda). Para que el programa sea eficiente es necesario almacenar compresiones ya obtenidas, utilizando para ello la técnica de memorización de lemas.

A continuación os ofrecemos una guía para realizar la práctica **de forma gradual**.

## PRELIMINARES

1. Emperazemos con una versión preliminar del predicado `comprimir/2` de cabecera `comprimir(Inicial,Comprimida)`:

```
comprimir(Inicial,Comprimida):-
    limpia_memo,
    compresion_recursiva(Inicial,Comprimida).

limpia_memo.

compresion_recursiva(Inicial,Inicial).
```

   Esta solución que obviamente no comprime, la tomaremos de base para progresivamente implementar los requisitos de la práctica. El predicado `limpia_memo/0` se encargará más adelante de limpiar la base de datos de memorización. El predicado `compresion_recursiva/2` será el predicado de compresión interno que llamemos de forma recursiva (sin limpiar la base de datos de memorización).

2. Escribir un predicado `partir/3` con cabecera `partir(Todo, Parte1, Parte2)` que se verifica si `Parte1` y `Parte2` son dos subsecuencias no vacías que concatenadas forman la secuencia `Todo`.

3. Escribir un predicado `parentesis/3` con cabecera `parentesis(Parte,Num,ParteNum)` que compone la lista `Parte` con el número de repeticiones `Num`, añadiendo paréntesis solo si `Parte` tiene 2 elementos o más.

4. Implementar un predicado `se_repite/4` con cabecera `se_repite(Cs,Parte,Num0,Num)`, que tiene éxito si `Cs` se obtiene por repetir $N$ veces la secuencia `Parte`. El argumento `Num` incrementa `Num0` en $N$.

## FASE A

Esta será la primera versión de la compresión recursiva, usando únicamente la repetición. Esta versión nos dará mediante backtracking distintas soluciones, puesto que todavía no vamos a buscar la más óptima.

1. Cambiaremos el predicado `compresion_recursiva/2` a la siguiente versión (reemplazando la anterior):

```
compresion_recursiva(Inicial,Comprimido) :-
    repeticion(Inicial,Comprimido).
% No compresion posible:
compresion_recursiva(Inicial,Inicial).
```

2. Debeis **implementar** el predicado `repeticion/2`, basándoos en los predicados `partir/3` y `se_repite/4` para identificar un prefijo (una parte) que nos de por repetición la secuencia inicial. Antes de seguir, esta parte debéis comprimirla de forma recursiva mediante una llamada a `compresion_recursiva/2`. Finalmente debe componer la parte (comprimida recursivamente) con el número de repeticiones usando el predicado `parentesis/3`.

### FASE B

En esta fase vamos a extender la solución anterior para comprimir repitiendo o dividiendo. El código ahora sí será capaz de obtener todas las posibles compresiones por backtracking, tanto óptimas como no óptimas.

1. Para ello cambiaremos el predicado `compresion_recursiva/2` a la siguiente versión (reemplazando la anterior):

```
compresion_recursiva(Inicial,Comprimido) :-
    compresion(Inicial,Comprimido).
% No compresion posible:
compresion_recursiva(Inicial,Inicial).
```

2. El **nuevo predicado** `compresion/2` tendrá dos alternativas: llamar al predicado `repeticion/2` ya implementado o a un **nuevo predicado** `division/2`. El predicado `division/2` debe partir la lista inicial en dos partes y llamar a `compresion_recursiva/2` de forma recursiva para finalmente concatenar los resultados.

   Es decir, además de considerar las repeticiones, podremos dividir la lista inicial en dos partes y aplicar el algoritmo a cada una de ellas por separado (dando más posibilidades a encontrar repeticiones).

### FASE C

En esta penúltima fase vamos a encargarnos de obtener solamente las compresiones óptimas.

1. Para ello `compresion_recursiva/2` lo cambiaremos de la siguiente forma:

```
compresion_recursiva(Inicial,Comprimido) :-
    mejor_compresion(Inicial,Comprimido).
```

2. Ahora `compresion_recursiva/2` llamará en su lugar al un **nuevo predicado** `mejor_compresion/2`, que intentará encontrar compresiones que reduzcan el tamaño. Fijaos que ahora tomamos la lista inicial como caso base para minimizar

en `mejor_compresion/2`, por ese motivo no incluimos una segunda cláusula en `compresion_recursiva/2` para el caso en el que no haya compresión posible.

El predicado `mejor_compresion/2` se puede implementar con predicados de agregación (`findall/3`, obteniendo todas las soluciones y quedándonos con la más corta) o llamando de forma reiterada con un parámetro a minimizar.

**FASE D**

Finalmente vamos a implementar la versión que realiza memorización de lemas. Esta será una solución óptima y además eficiente:

1. Ahora declaramos `memo/2` para asertar los lemas, implementamos `limpia_memo/0` para limpiarlos y cambiamos `compresion_recursiva/2`.

```
:- dynamic memo/2.

compresion_recursiva(Inicial,Comprimido) :-
    mejor_compresion_memo(Inicial,Comprimido).

limpia_memo :-
    retractall(memo(_,_)).
```

2. Debéis implementar un **nuevo predicado** `mejor_compresion_memo/2`, que utilizando el predicado `mejor_compresion/2`, implemente un esquema de memorización (e.g., igual que el visto en clase para Fibonacci).

PUNTOS ADICIONALES (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, y entregando como memoria o parte de ella el manual generado automáticamente a partir de dicho código, usando la herramienta `lpdoc` del sistema Ciao. Se recomienda intentar escribir este manual de forma que sustituya completamente a la memoria.

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

  También se valorará el uso de aserciones `test` que enumeren casos de prueba para comprobar el funcionamiento de los predicados.

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.

## Instrucciones generales para la realización y entrega las prácticas

Es **muy importante** leer el **documento con este título en Moodle**.