Teoría de los Lenguajes de Programación
Práctica curso 2013-2014

Índice

1. Introducción: Sumas.

El problema de las sumas parte de dos listas de números naturales en la que la primera contiene los resultados obtenidos al sumar dos elementos de la segunda lista. Por ejemplo, para las listas:

```
 resultados = [12,11,8]
sumandos = [1,3,9,6,7,5]
```

el `12` puede obtenerse sumando `3 + 9`, pero también sumando `7 + 5`.

Una solución al problema consiste en emparejar todos los números de la lista de sumandos de forma que cada elemento de la lista de resultados se pueda expresar como suma de dichas parejas de sumandos.

En el ejemplo anterior, si se escogiese `7 + 5` como operación que devuelve `12`, no se podría expresar el 8 como suma, ya que las únicas formas de hacerlo son `1 + 7` ó `3 + 5`. La única solución al ejemplo es:

```
 8 = 1 + 7
11 = 6 + 5
12 = 3 + 9
```

2. Enunciado de la práctica

La práctica consiste en programar, utilizando **HUGS** una función *resuelveSumas* que, dada una tupla formada por una lista de resultados y una lista de sumandos, devuelva una cadena de caracteres indicando si el problema tiene solución y si ésta es única o no.

En caso de que el problema tenga una única solución, se mostrará dicha solución. Si hubiera más de una solución, se mostrarán dos soluciones diferentes.

Por ejemplo, una ejecución del programa podría ser la siguiente:

```
Sumas> putStr ( resuelveSumas ([12,11,8],[1,3,9,6,7,5]) )
El problema tiene una unica solucion:
8 = 1 + 7
11 = 6 + 5
12 = 3 + 9

Sumas>
```

La función *putStr* de **HUGS** permite que los códigos de formateo (saltos de línea, tabuladores...) de las cadenas de caracteres se interpreten correctamente en lugar de mostrarse tal cual se escriben (`"\n"`, `"\t"`...).

2.1 Método de Resolución

El método de resolución es el siguiente: partiendo de las listas de resultados y sumandos, se escoge un resultado y se encuentran todas las parejas de sumandos que permiten expresar el resultado como

suma de dos sumandos.

En el ejemplo anterior, la sucesión de estados de las listas de resultados y sumandos sería la siguiente:

Resultados	Sumandos	Parejas de sumandos
[12,11,8]	[1,3,9,6,7,5]	[]
[11,8]	[1,6,7,5]	[(3,9)]
[8]	[1,7]	[(6,5),(3,9)]
[]	[]	[(1,7),(6,5),(3,9)]

Para almacenar las parejas se deberá utilizar una tupla de dos elementos, que contendrán los sumandos escogidos.

Aplicando un algoritmo de vuelta atrás (backtracking) sobre el grafo generado por un nodo inicial formado por la lista inicial de resultados, la lista inicial de sumandos y una lista vacía de parejas de sumandos, se obtendría la lista de todas las soluciones al problema.

Así pues, un nodo contendrá la siguiente información:

```
(lista de resultados, lista de sumandos, lista de parejas de sumandos)
```

Para ello utilizaremos la función *bt*, que aplica un algoritmo de vuelta atrás sobre el grafo generado por el problema. Esta función se da programada.

Generación de los nodos

Para generar los nodos hijo a partir de un nodo dado, podemos seguir el siguiente proceso:

1. Mediante una función *dosQueSumen* obtenemos una lista con todas las parejas de sumandos que sumen un cierto valor x:

   ```
   Sumas> dosQueSumen 8 [1,3,9,6,7,5]
   [(1,7),(3,5)]
   Sumas>
   ```

2. Con una función *sinRepes* eliminaremos las posibles parejas repetidas de sumandos. Hay que tener en cuenta que las parejas (x,y) e (y,x) son la misma (ya que la suma es conmutativa):

   ```
   Sumas> sinRepes [(3,5),(5,3)]
   [(3,5)]
   Sumas>
   ```

   Si no se filtrase la lista de parejas devuelta por la función *dosQueSumen*, el grafo resultante sería mayor y se emplearía más tiempo en su exploración, además de poder encontrar dos formas diferentes de expresar una misma solución.

3. La función *hijosNodo* devolverá una lista con todos los nodos resultantes de elegir una pareja de sumandos de la lista de sumandos que sumen un elemento de la lista de resultados.

El nuevo nodo contendrá una nueva lista de resultados sin el número obtenido, una nueva lista de sumandos a la que se le habrán eliminado los dos sumandos elegidos y una nueva lista de parejas a la que se habrá añadido la pareja de números. Si, por ejemplo, se escoge el primer número de la lista de resultados, una posible ejecución de *hijosNodo* sería:

```
Sumas> hijosNodo ([12,11,8],[1,3,9,6,7,5],[])
[([11,8],[1,6,7,5],[(3,9)]),
 ([11,8],[1,3,9,6],[(7,5)])]
Sumas>
```

Función Principal

Para programar la función *resuelveSumas* se deberá realizar una llamada a la función *imprimeSoluciones*, a la que se le deberá pasar como parámetro la lista de soluciones que se obtenga de aplicar el algoritmo de backtracking (función *bt*) al nodo inicial del problema.

2.2 Juegos de Prueba

El módulo "*SumasPruebas.hs*" contiene una serie de juegos de prueba que sirven para comprobar el buen funcionamiento de la práctica. Para que los juegos de prueba funcionen correctamente se incluye la siguiente línea al comienzo del módulo Sumas:

```
module Sumas where
```

Es importante que el archivo conteniendo el módulo Sumas se guarde como "*Sumas.hs*" (respetando para el correcto funcionamiento de los juegos de prueba.

3. Cuestiones sobre la práctica

Para responder a estas cuestiones deberá comparar la práctica programada en **HUGS** con la que el Equipo Docente proporciona programada en **PROLOG**. Además, algunas preguntas implican realizar comparaciones con Java, un lenguaje del paradigma de Programación Orientada a Objetos.

La respuesta a estas preguntas es optativa. Sin embargo, si el alumno no responde a estas preguntas, la calificación de la práctica sólo podrá llegar a 5 puntos sobre 10.

1. Tal y como está descrito el algoritmo, en algunas ocasiones se podrían encontrar dos soluciones iguales que serían tratadas como diferentes. Por ejemplo:

   ```
   Sumas> putStr ( resuelveSumas ([4,4],[2,2,1,3]) )
   El problema tiene mas de una solucion:
   4 = 1 + 3
   4 = 2 + 2

   y

   4 = 2 + 2
   4 = 1 + 3
   ```

   Aunque es obvio que ambas soluciones son idénticas.

   Para solucionar este problema, se pide crear una nueva función *resuelveSumas2*, que filtre

de la lista de soluciones aquellas soluciones que sean iguales. Recuérdese que tanto las sumas como los sumandos de una de ellas pueden aparecer en cualquier orden.

El filtrado de la lista de soluciones debe aprovechar la evaluación perezosa.

2. En un lenguaje Orientado a Objetos, ¿cuál sería la estructura adecuada para representar un nodo? ¿Qué diferencias presentaría el manejo de dicha estructura con respecto al tipo de datos `Nodo` que se usa en la versión funcional?

3. Escriba en pseudocódigo Java (no es necesario que compile correctamente) una función de recorrido en profundidad para este problema. Compare la eficiencia, referida a la programación, de dicha función en Haskell, Prolog y Java.

4. Indique, con sus palabras, qué permite gestionar el predicado predefinido no lógico, corte (!), en Prolog. ¿Cómo se realiza este efecto en Java?

5. Compare la estructura de control que permite recorrer los hijos de un determinado nodo en la función `bt`, en Haskell, Prolog y Java. Establezca las similitudes y diferencias entre dichas estructuras de control.

4. Documentación a entregar

Cada estudiante deberá entregar la siguiente documentación a su tutor de prácticas:

- Código fuente en **HUGS** que resuelva el problema planteado. Para ello se deberán entregar el fichero *Sumas.hs*, con las funciones descritas en este enunciado, así como todas las funciones auxiliares que sean necesarias.
- Una memoria con:
  - Una pequeña descripción de las funciones programadas.
  - Las respuestas a las cuestiones sobre la práctica.