Paradigmas de Lenguajes de Programación

Trabajo práctico 1 - Programación Funcional

Fecha de entrega: martes 11/9/2012

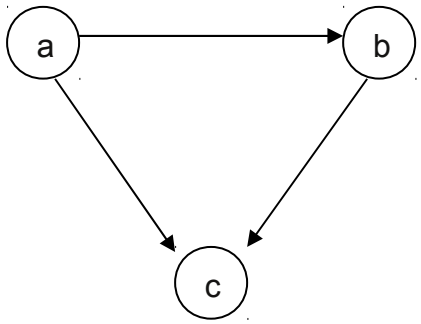
Introducción

El objetivo de este trabajo es implementar en Haskell un programa que permita trabajar con grafos dirigidos, a representar mediante nodos y funciones que denoten la adyacencia de éstos.

```
data Grafo a = G {nodos :: [a], adyacencias :: a->[a]}
```

Se representará un grafo dirigido mediante una estructura que identifique un par (nodos, adyacencias), donde nodos es la lista de sus nodos, y adyacencias es una función que a cada nodo le asigna el conjunto de sus nodos adyacentes (es decir a cuáles está conectado a través de los ejes dirigidos de este grafo). La lista estará representando el conjunto de los nodos, por lo cual se asumirá que no contiene nodos repetidos. Se puede asumir que el grafo no tiene lazos (nodos conectados a sí mismos). Toda función que devuelva un grafo o una lista de nodos deberá procurar que éstos no se repitan en la lista.

Ejemplo:



nodos = {a,b,c}, ady(a) = {b,c}, ady(b) = {c}, ady(c) = ∅

A continuación, para cada una de las funciones a definir, se pide declarar el tipo (más general posible) y definir el cuerpo. Salvo indicación contrario, no se permite utilizar recursión explícita (ni directa ni indirecta). Debe incluirse al menos un caso de prueba para la función principal de cada ejercicio.

Ejercicio 1

Dados un grafo y un nodo nuevo, devolver el grafo que resulta de agregar este nodo al grafo, de modo que no resulte adyacente a ningún nodo.

Ejercicio 2

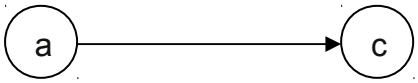
Dados un grafo y un "eje" (representado como un par de nodos), devolver el grafo que resulta de agregar este eje al grafo.

Ejercicio 3

Dados un grafo y un eje de éste, devolver el grafo que resulta de eliminar este eje del grafo.

Ejercicio 4

Dados un grafo y un nodo de éste, devolver el grafo que resulta de eliminar este nodo del grafo junto a sus ejes incidentes (i.e. que lo tocan en algún extremo).
Por ejemplo, al eliminar el nodo 'b' del grafo del ejemplo anterior, resulta el grafo



nodos = {a,c}, ady(a) = {c}, ady(c) = ∅

Ejercicio 5

a) Dados un grafo y un nodo de éste, devolver el grado de dicho nodo (es decir la cantidad de ejes incidentes en él en uno u otro sentido).

b) Dado un grafo, devolver el máximo grado de sus nodos.

Ejercicio 6

a) Dadas dos listas cualesquiera (con elementos comparables por igualdad), devolver la diferencia de ambas (como diferencia de conjuntos).

b) Dados dos grafos, determinar si el primero es subgrafo del segundo. Para que un grafo sea subgrafo de otro, el conjunto de nodos del primero deben ser subconjunto del conjunto de nodos del segundo, y los ejes análogamente.

Ejercicio 7

Dados un grafo y un subconjunto de sus nodos, devolver el subgrafo inducido por dicho conjunto. Es decir, aquel subgrafo que contiene los ejes del grafo original que tienen ambos extremos en dicho conjunto. Sugerencia: usar la función del ejercicio 6 (a).

Ejercicio 8

Dados un grafo y un nodo de éste, determinar si hay algún ciclo (no dirigido) que lo contenga. Sugerencia: ir eliminando nodos con grado menor que 2 mientras sea posible, y a partir del grafo resultante determinar la respuesta. Se recomienda usar la función iterate.

Ejercicio 9

Dado un grafo, determinar si es (débilmente) conexo. *En este ejercicio se puede usar recursión explícita.* Sugerencia: utilizar el algoritmo que comienza con un nodo cualquiera marcado y a continuación va marcando nodos que sean adyacentes (en algún sentido) a otros nodos ya marcados mientras sea posible. Terminarán todos marcados si y sólo si el grafo es conexo.

Ejercicio 10

Dado un grafo, determinar si es un árbol (no orientado, es decir ignorando la orientación de los ejes).

Pautas de entrega

Se debe entregar el código impreso con la implementación de las funciones pedidas, incluyendo sus tipos. El código debe estar adecuadamente comentado. Cada función asociada a los ejercicios debe contar con ejemplos (casos de prueba) que muestren que exhibe la funcionalidad solicitada. Además, se debe enviar un e-mail conteniendo el trabajo en Haskell a la dirección plp-docentes@dc.uba.ar. Dicho mail debe cumplir con el siguiente formato:

- El título (subject) del mensaje debe ser "[PLP;TP-PF]" seguido inmediatamente del nombre del grupo.
- El código Haskell debe acompañar el e-mail en forma de archivo adjunto (puede adjuntarse un .zip, .rar, .tgz o .tar.gz).

Los objetivos a evaluar en la implementación de las funciones son:

- Corrección.
- Declaratividad.
- Reutilización de funciones previamente definidas (tener en cuenta tanto las funciones definidas en el enunciado como las definidas por los integrantes).

- Uso de funciones de alto orden, currificación y esquemas de recursión.
- Salvo donde se indique lo contrario, no se permite utilizar recursión explícita, dado que la idea del TP es aprender a aprovechar las características enumeradas en el ítem anterior. Se permite utilizar esquemas de recursión definidos en el preludio. Pueden escribirse todas las funciones auxiliares que se requieran, pero al igual que las otras éstas no pueden ser recursivas (ni mutuamente recursivas).
- Pueden utilizar cualquier función definida en el preludio de Haskell y los módulos List y Maybe. Las sugerencias de los ejercicios pueden ayudar, pero no es obligatorio seguirlas.
- El código debe poder ser ejecutado en Haskell98. No es necesario entregar un informe sobre el trabajo, alcanza con que el código esté adecuadamente comentado.
- La entrega deberá incluir casos de prueba para las funciones pedidas. Se recomienda la codificación de tests. Por ejemplo Hunit permite hacerlo con facilidad.

Importante: se admitirá un único envío, sin excepción. Se sugiere planificar el trabajo para llegar a tiempo con la entrega.