Proyecto 4
TAD's

Algoritmos y Estructuras de Datos I
Laboratorio

28 de septiembre de 2009

En este proyecto hay una serie de tipos abstractos de datos para programar en Haskell. Al final se deberán escribir dos implementaciones distintas del mismo TAD diccionario con los TAD's implementados previamente.

Para hacer cada TAD se deberá tener en cuenta:

- Cada TAD se deberá escribir en un archivo separado.
- Cada uno de estos archivos contendrá un módulo en Haskell que defina el TAD.
- Para implementar correctamente los TAD's, cada modulo deberá exportar **únicamente** las funcionalidades del TAD que define, ocultando todos los aspectos que tienen que ver con su implementación (por ejemplo declaración `data`).

A continuación se detallan los TAD's a implementar.

1. Implementar el TAD Dato que sirve para encapsular valores. El TAD debe ser escrito en un archivo separado `Dato.hs`. La signatura del TAD debe ser:

```
module Data ( Data,
              data_fromString,
              data_toString,
              data_length
            )
where

data Data = ...

-- De un string construye una dato
data_fromString :: String -> Data

-- Convierte una dato a string
data_toString :: Data -> String

-- Devuelve el tamaño del dato
data_length :: Data -> Int
```

   Tener en cuenta que se quiere obtener rápidamente el tamaño de un `Data` (función `data_length`).

   Ayuda: El tipo que lo implementa debe ser algo como

```
data Data = Value String Int
-- Almacena el tamaño
```

2. Implementar el TAD Key que sirve para almacenar claves. Las claves son TAD iguales que los anteriores pero poseen un tamaño máximo, igualdad y orden. El TAD debe ser escrito en un archivo separado `Key.hs`. La signatura del TAD debe ser:

```
module Key ( Key,
             key_MaxLen,
             key_fromString,
             key_toString,
             key_length
           )
where

data Key = ...

-- Máximo tamaño de la clave
key_MaxLen :: Int

-- De un string construye una clave
-- El tamaño del string debe ser menor o igual a key_MaxLen
key_fromString :: String -> Key

-- Convierte una clave a string
key_toString :: Key -> String

-- Devuelve el tamaño de la clave
key_length :: Key -> Int

-- Las claves se pueden comparar
instance Eq Key where
   ...

instance Ord Key where
```

Tener en cuenta que se quiere obtener rápidamente el tamaño de una `Key` (función `key_length`).

3. Implementar un TAD lista de asociaciones. Las listas de asociaciones sirven para almacenar pares de valores relacionados donde se puede obtener uno de ellos a partir del otro (función `la_search` más adelante). El TAD debe ser escrito en un archivo separado `ListAssoc.hs` y la signatura del TAD debe ser:

```
module ListAssoc ( ListAssoc,
                   la_empty,
                   la_add,
                   la_search,
                   la_del,
                   la_toListPair
                 )
where

data ListAssoc a b = Node a b (ListAssoc a b) | Empty

-- Lista de asociaciones vacia
la_empty :: ListAssoc a b
```

```
-- Agrega un elemento a una lista de asociaciones.
-- Si ya esta el elemento de tipo a devolver
-- la misma lista
la_add :: ListAssoc a b -> a -> b -> ListAssoc a b

-- Devuelve el elemento asociado
-- Si no esta devuelve Nothing
la_search :: Eq a => ListAssoc a b -> a -> Maybe b

-- Borra un elemento a la lista de asociaciones
-- Si no esta no hace nada
la_del :: Eq a => ListAssoc a b -> a -> ListAssoc a b

-- Devuelve la lista de pares correspondiente a la lista de asociaciones
la_toListPair :: ListAssoc a b -> [(a,b)]
```

Nota: El tipo `Maybe` esta definido en el preludio de Haskell como

```
data Maybe a = Just a | Nothing
```

y es usado en funciones que para casos excepcionales no devuelven nada, como por ejemplo la función `la_search`. Esta función devolverá `Nothing` en el caso que el elemento no este almacenado.

4. Implementar un TAD Diccionario. Los diccionarios son TAD's que contienen las palabras de un diccionario junto con su definición.

   a) Hacer una implementación del diccionario utilizando los TAD's listas de asociaciones de `Key` y `Data` implementadas en los ejercicio 1, 2 y 3.
   La signatura del TAD debe ser:

```
module Dict ( Dict,
              Word,
              Def,
              dict_empty,
              dict_add,
              dict_exist,
              dict_del,
              dict_length,
              dict_toDefs )
where

import Data
import Key
import ListAssoc

-- se almacena el tamaño
data Dict = Dict Int (ListAssoc Key Data)

type Word = String
type Def  = String

-- Crea un diccionario vacio
dict_empty :: Dict
```

```
-- Agrega un dato con una key al diccionario
-- Si la clave ya está no hace nada
dict_add :: Dict -> Word -> Def -> Dict

-- Pregunta si la clava esta en el diccionario
dict_exist :: Dict -> Word -> Bool

-- Devuelve el dato asociado a un diccionario
-- Si no esta devuelve Nothing
dict_search :: Dict -> Word -> Maybe Def

-- Borra un dato del diccionario de acuerdo a una clave
-- Si la clave no esta no hace nada
dict_del :: Dict -> Word -> Dict

-- Devuelve la cantidad de datos en el diccionario
dict_length :: Dict -> Int

-- Devuelve las definiciones del diccionario
dict_toDefs :: Dict -> [(Word,Def)]
```

b) Cambiar la implementación de las listas de asociaciones hecha en el ejercicio 3 de forma tal que se mantenga como invariante de representación que la lista esté ordenada.

Sin cambiar nada del TAD `Dict` (del punto anterior) probar esta nueva implementación.

c) Además de usar Hugs, probar compilar el programa con el comando:

```
ghc --make Main
```

Nota: Los profesores de la materia les facilitaran un programa para probar diccionarios y un diccionario de prueba.

Nota 2: Acordarse de guardar todas las versiones de cada ejercicio en directorios separados para que los ayudantes puedan evaluarlas.

Nota 3: Recordar que el tipo `Data` no debe pertenecer a la clase `Eq`.

5. Implementar un TAD árbol binario de búsqueda (`Abb`). Los `Abb` sirven para almacenar pares de valores relacionados donde se puede obtener uno de ellos a partir del otro (función `abb_search` más adelante) al igual que en el TAD del ejercicio anterior.

   Los `Abb` se implementan como árboles con información en los nodos. La información a guardar será el par de elementos relacionados, pidiéndose además que el tipo del primer elemento de esta relación tenga un orden (pertenezca a la clase `Ord` en Haskell). Además se debe mantener como invariante de representación que todos elementos almacenados en un subárbol izquierdo sean menores que la raíz y que los almacenados en el subárbol derecho sean mayores (respecto al primer elemento de la relación).

   El TAD debe ser escrito en un archivo separado `Abb.hs` y la signatura del TAD debe ser:

```
module Abb ( Abb,
             abb_empty,
             abb_add,
             abb_search,
             abb_del,
```

```
              abb_toListPair)
where

data Abb a b = Branch a b (Abb a b) (Abb a b) | Leaf

-- Arbol vacio
abb_empty :: Abb a b

-- Agrega un elemento al abb.
-- Si ya está el elemento no lo agrega
abb_add :: Ord a => Abb a b -> a -> b -> Abb a b

-- Devuelve el elemento asociado
-- Si no esta devuelve Nothing
abb_search :: Ord a => Abb a b -> a -> Maybe b


-- Borra un elemento del arbol
-- Si no esta no hace nada
abb_del :: Ord a => Abb a b -> a -> Abb a b

-- Devuelve la lista de pares correspondiente de los elmentos
-- almacenados en el arbol
abb_toListPair :: Abb a b -> [(a,b)]
```

6. Cambie la implementación del diccionario para que utilice árboles binarios de búsqueda en vez de lista de asociaciones. Fíjese que al hacerlo no se cambie ninguna de las signaturas de las funciones y tipos exportados en el módulo `Dict`. Pruebe hacer una búsqueda de una palabra con ambas implementaciones y compare la cantidad de reducciones.

7. **Punto estrella:**[1] hacer una clase `Container` de los tipos que almacenen cosas y tengan las funciones `empty`, `add`, `search`, `del` y `toListPair`. Hacer pertenecer a esta clase a los TAD's lista de asociaciones y árbol binario de búsqueda. Con esto, hacer las dos últimas implementaciones del diccionario sin cambiar la implementación de las funciones.

[1] Este punto no es necesario hacerlo para aprobar el proyecto. Si todo lo demás esta perfecto y se hace este punto entonces tiene un Bien +.