Algoritmos y Estructuras de Datos I - Laboratorio
Proyecto 3

Tipos Abstractos de Datos, Módulos

1. Objetivo y preparación

El objetivo del proyecto es programar en Haskell una serie de Tipos Abstractos de Datos (TAD). Para hacer cada TAD se deberá tener en cuenta:

- Cada TAD está en su propio módulo Haskell, en un archivo separado (ver tutorial).
- Cada módulo exporta **únicamente** las funcionalidades de su TAD, ocultando su implementación.

En este proyecto trabajamos con un programa que permite cargar, editar y guardar diccionarios.

Preparación Los commands que siguen son para escribir en un terminal.

- Bajar el archivo `.zip` del Proyecto 3 en la página de la materia. Descomprimirlo.
- Entrar a la carpeta nueva. Tenemos:
  - `diccionario.dic` y `diccionario_completo.dic`: archivos que se pueden cargar en el programa (en el programa, entrar el nombre sin la extensión `.dic`).
  - `Main.hs`: el módulo principal del programa, `Main`.
  - `Data.hs`, `Key.hs`, `ListAssoc.hs`, `Dict.hs` y `Abb.hs`: los TADs importados por `Main`.
- Para correr el programa, cargar `Main.hs` en `ghci` y evaluar la función `main`
- Cargar un diccionario (entrar el nombre: `diccionario`).

Al cargar el diccionario, se interrumpe el programa con un error: `Prelude.undefined`. Es porque casi todas las funciones son definidas con `undefined` y vamos a tener que definirlas nosotros. Recordamos que `undefined` en una constante especial que provoca un error al evaluarla. Al final del ejercicio 4 se podrá usar el programa normalmente por primera vez.

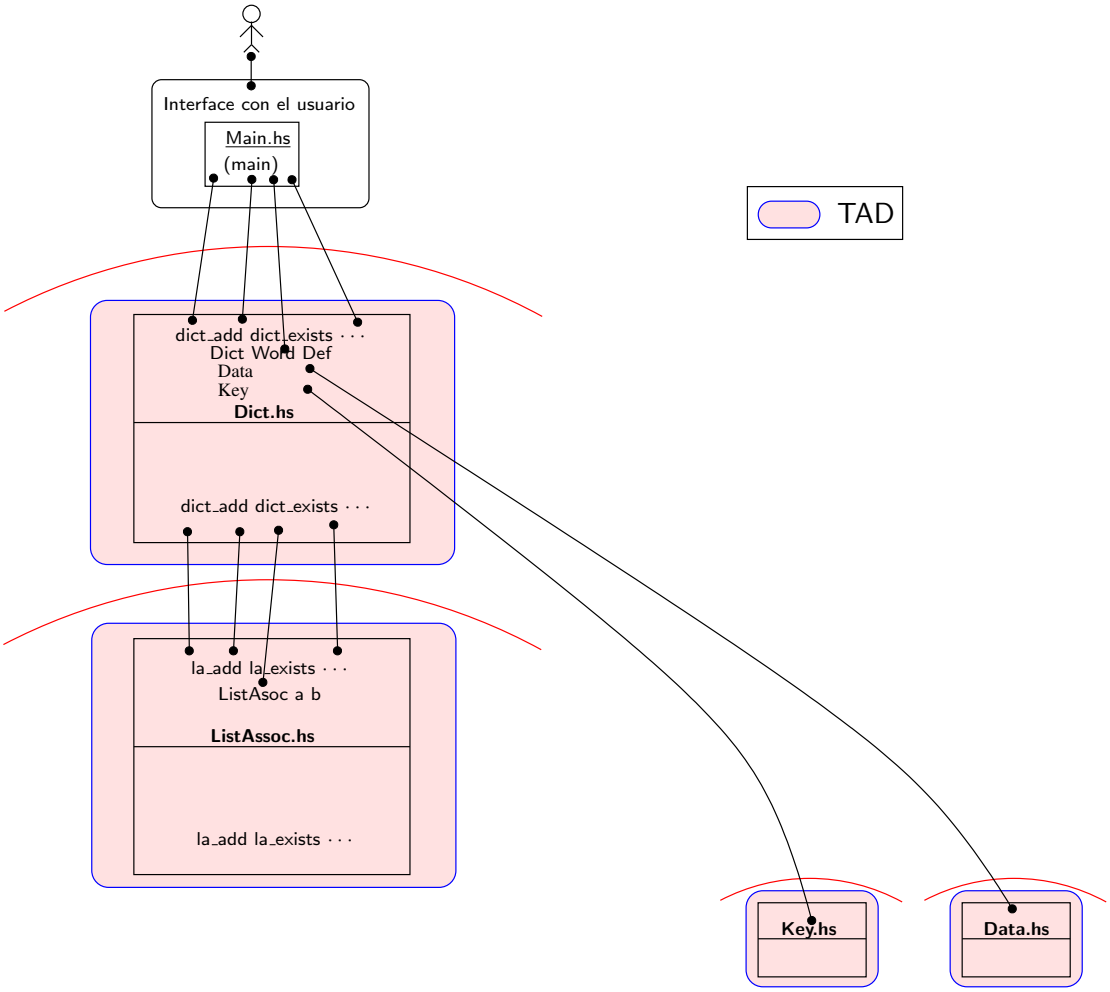
Vamos a implementar las funciones del programa reemplazando las definiciones `undefined` por definiciones correctas. Por ejemplo, en `Data.hs`:

```
data Data = Value String Int

-- De un string construye una dato
-- Almacena el tamano
data_fromString :: String -> Data
data_fromString = undefined
```

Una manera de implementar `data_fromString` es la siguiente:

```
data Data = Value String Int

-- De un string construye una dato
-- Almacena el tamano
data_fromString :: String -> Data
data_fromString s = Value s (longitud_de s)
```

Los módulos están organizados de la manera siguiente:



2. Ejercicios y Preguntas

1. Editar el archivo `Data.hs`. Implementar el TAD del módulo `Data`, que sirve para encapsular valores. Es decir, reemplazar todas las definiciones de funciones `undefined` por la implementación correcta de cada una.

   Una vez que todas las funciones son implementadas, se puede probar que el código es correcto cargando el módulo `Data` en `ghci` y testeando las funciones con varias entradas.

2. Implementar el TAD del módulo `Key`, que sirve para almacenar claves. Las claves son TAD iguales que los anteriores pero poseen un tamaño máximo, igualdad y orden.

   Para más información acerca de las instanciaciones de clases de tipos, ver esa sección de Aprende Haskell y esa sección de Introducción Agradable a Haskell.

3. Implementar el TAD lista de asociaciones del módulo `ListAssoc`. Las lista de asociaciones sirven para almacenar pares de valores relacionados donde se puede obtener uno de ellos a partir del otro.

   En caso de duda buscar el tipo de datos `Maybe` en Hoogle.

4. Implementar el TAD diccionario del módulo `Dict`. Los diccionarios contienen palabras junto con su definición.

   Hacer una implementación del diccionario utilizando los TAD's listas de asociaciones de `Key` y `Data` implementadas en los ejercicio 1, 2 y 3.

5. Correr el programa y probarlo: cargar un diccionario, agregarle definiciones, guardarlo en otro archivo, etc.

6. Crear un modulo `ListAssocOrd` idéntico a `ListAssoc`, con la diferencia de que se mantiene como invariante de representación que la lista esté ordenada.

   Fíjese que al hacerlo cambiar unicamente las signaturas de las funciones `la_add`, `la_search` y `la_del`.

   Importar ese módulo nuevo en `Dict`.

7. Probar la nueva implementación: ¿Cómo se nota que la implementación de la lista de asociaciones es ordenada?

8. Visualizar las diferencias entre los archivos `ListAssoc.hs` y `ListAssocOrd.hs` usando los comandos:

   - `diff`
   - `meld`

   Asegurarse que estos dos módulos sean lo más idénticos posibles.

   Ayuda: *Mirar el manual de* `diff` *y* `meld` *usando:* `man diff` *y* `man meld`.

9. (**Punto** $\star$) Implementar el TAD árbol binario de búsqueda (`Abb`). Los `Abb` sirven para almacenar pares de valores donde se puede obtener uno de ellos a partir del otro al igual que en el TAD del ejercicio anterior.

   Los `Abb` se implementan como árboles con información en las ramas. La información a guardar será un par de elementos, pidiéndose además que el tipo del primer elemento del par tenga un orden (pertenezca a la clase `Ord`). Además se debe mantener como invariante de representación que todos los primeros elementos de los pares almacenados en un subárbol izquierdo sean menores que el primer elemento del par almacenado en la rama, y que los almacenados en el subárbol derecho sean mayores.

10. (**Punto** $\star$) Crear un módulo `DictAbb` idéntico a `Dict`, a la diferencia que la implementación del diccionario utilice árboles binarios de búsqueda.

    Importar ese módulo en `Main` y probar el programa con ese cambio.

11. (**Punto** $\star$) Hacer una clase `Container` (en archivo separado) de los tipos que almacenan datos indexados 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.

    Ayuda: *Leer las secciones "Clases de tipos paso a paso (2a parte)" y "La clase de tipos Yes-No" del capítulo "Creando nuestros propios tipos y clases de tipos" del libro "Aprende Haskell por el bien de todos".*