Proyecto 3:
Tipos Abstractos de Datos, Módulos

Algoritmos y Estructuras de Datos I
Laboratorio

Laboratorios: 17/04, 24/04, 8/05 (corrección el 15/05).

En este proyecto trabajamos con un programa que permite cargar, editar y guardar diccionarios. Ahora vemos como bajarlo desde la página de la materia y compilarlo por primera vez.

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

- Bajar el archivo `.tar.gz` del Proyecto 3 en la página de la materia.
- Descomprimirlo (`tar xzf NOMBRE_ARCHIVO`).
- Entrar a la carpeta nueva (`cd`), listar los archivos (`ls`).
- Mirar el contenido de `Makefile` (`less Makefile`, `q` para salir).

  `Makefile` es un archivo usado por el command `make`. Acá lo usamos de manera muy simple para definir dos commands:

  - `make`: llama `ghc` para compilar nuestro programa, es decir, convierte el código fuente en un programa ejecutable.
  - `make clean`: borra los archivos generados por la compilación.
- Compilar el programa con el command `make`. Listar de nuevo los archivos (`ls`).
- Ejecutar el programa con `./Main` y tratar de cargar un diccionario (entrar el nombre `diccionario`).

Al cargar el diccionario, el programa tira un error: `Prelude.undefined`. Es porque en todos los módulos del programa (aparte `Main`), todas las funciones son definidas con `undefined` y vamos a tener que definirlas nosotros. Recién al final del ejercicio 4 se podrá usar el programa normalmente por primera vez.

Recordamos que `undefined` en una constante especial de Haskell que se puede usar en todos lados, y que provoca un error al evaluarla.

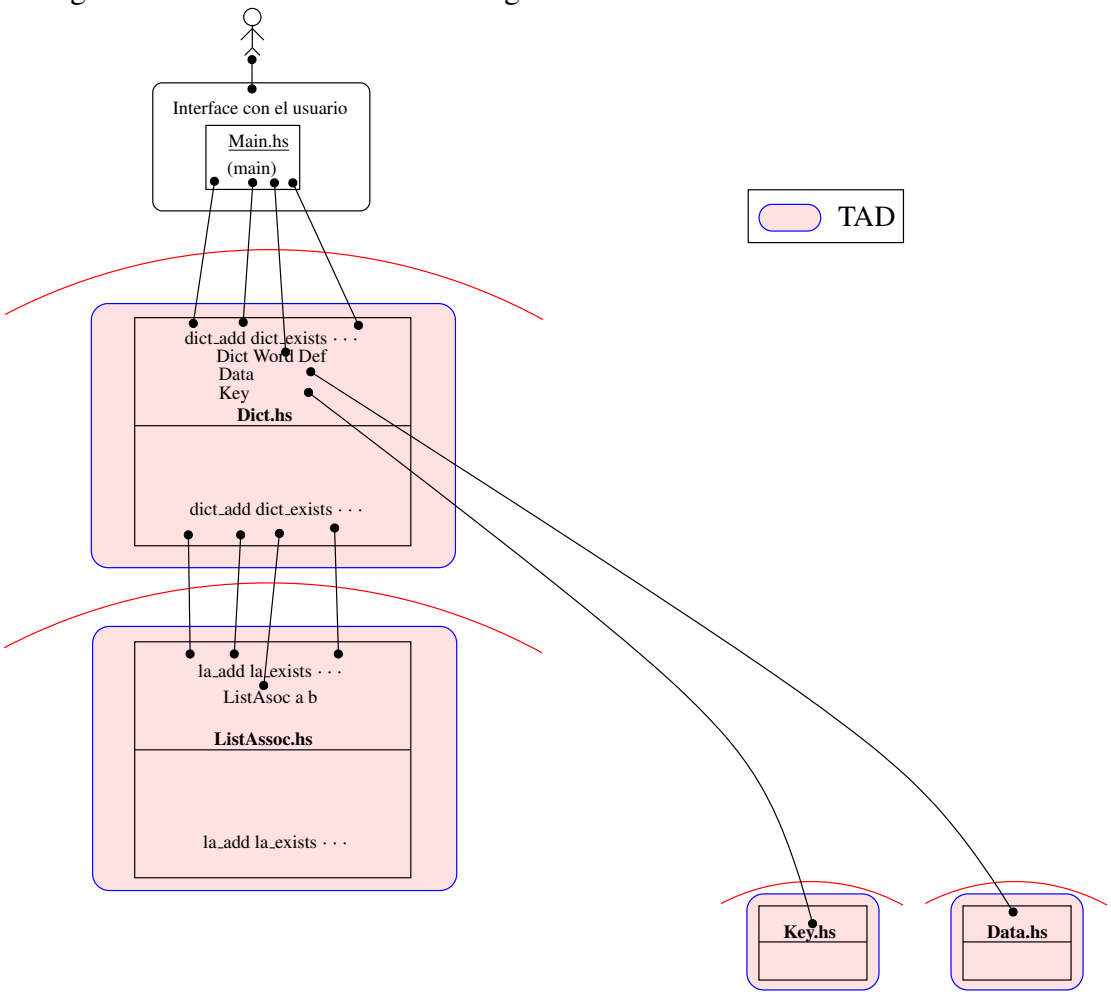
Entonces el objetivo del proyecto es programar en Haskell una serie de Tipos Abstractos de Datos (TAD). Para hacer cada TAD se deberá tener en cuenta:

- Cada TAD está en su propio módulo Haskell, en un archivo separado (ver Sección 11 del tutorial).
- Cada módulo exporta **únicamente** las funcionalidades del TAD que define, ocultando su implementación.

Ahora miramos el contenido de la carpeta:

- `diccionario.dic` y `diccionario_completo.dic` son archivos que se pueden cargar en el programa (en el programa, entrar el nombre sin la extensión `.dic`).
- `Main.hs` contiene es el módulo principal del programa, `Main`.
- `Data.hs`, `Key.hs`, `ListAssoc.hs`, `Dict.hs` y `Abb.hs` contienen los TADs importados por `Main`.

La organización de los módulos es la siguiente:



¿Cómo implementamos las funciones de los archivos? Miremos 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)
```

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 el código de dos maneras:

   - compilando el programa entero con `make`
   - cargando el módulo `Data` en `ghci` y testeando las funciones con varias entradas

2. Implementar el TAD del módulo `Keys`, 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 `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.

   A esta altura del proyecto el programa, es usable por primera vez.

5. 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` para probar la nueva implementación.

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

6. (**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.

7. (**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.

8. (**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.