Trabajo Práctico 1
Programación Funcional

versión 1.0

Paradigmas ~~de Lenguajes~~ de Programación
1er cuatrimestre 2025

Fecha de entrega: 11 de abril

Introducción

Para este trabajo práctico vamos a implementar dos módulos.

- `Documento` donde definiremos un tipo de dato `Doc` y funciones para trabajar con documentos. Un documento es una estructura cuyo objetivo es mostrarse por pantalla en forma amigable para el usuario.
- `PPON` donde definiremos un tipo de dato `PPON` y funciones para trabajar con este formato de datos compuesto. Además, definiremos cómo convertir un `PPON` a un `Doc` para mostrarlo en forma amigable.

Los archivos base provistos ya cuentan con la estructura de los módulos y las definiciones de los tipos de datos. En cada módulo se pide implementar las funciones que se detallan en los ejercicios. **Cada módulo ya define qué funciones se exportan y no se permite alterar dicha declaración**.

Además, se dispone de un módulo `Main` que importa los dos módulos anteriores y se espera definan casos de tests propios. Ya se dispone de algunos ejemplos. Para esto se recomienda la utilización de la librería `HUnit`.

Módulo `Documento`

Queremos definir un modulo que exporta una interfaz para trabajar con documentos. Un documento está representado por la estructura recursiva `Doc` con la siguiente definición:

```
data Doc = Vacio | Texto String Doc | Linea Int Doc
```

- `Vacio`: representa un documento vacío.
- `Texto`: representa un documento que contiene como primer componente un texto.
- `Linea`: representa un documento que contiene como primer componente un salto de linea. La siguiente linea comienza con una cantidad de espacios indicada por un entero.

Un valor del tipo `Doc` debe cumplir con los siguientes invariantes:
Sea `Texto s d` entonces:

- `s` no debe ser el string vacío.
- `s` no debe contener saltos de línea
- `d` debe ser `Vacio` o `Linea i d'`

Sea `Linea i d` entonces:

- `i >= 0`

Como la estructura tiene invariantes **vamos a no exportar los constructores** y definiremos funciones para crear documentos **que se aseguren de mantenerlos**.

Disponemos ya de algunas funciones para construir documentos. Éstas funciones sí se exportan.

```
-- | Devuelve un documento vacío
vacio :: Doc
vacio = Vacio

-- | Devuelve un documento que consta de un salto de linea.
linea :: Doc
linea = Linea 0 Vacio

-- | Devuelve un documento que consta del texto dado.
-- PRE: El texto no debe contener saltos de línea.
texto :: String -> Doc
texto t | '\n' `elem` t = error "El texto no debe contener saltos de línea"
texto [] = Vacio
texto t = Texto t Vacio
```

Ejercicio 1

Queremos recorrer documentos usando un esquema de recursión estructural. Como vimos anteriormente, para las listas ya existe `foldr`, de modo que necesitamos un nuevo `foldDoc` para recorrer documentos. Indicar explícitamente el tipo de la función. En este ejercicio, sí se permite usar recursión explicita.

Ejercicio 2

Definir la función `(<+>) :: Doc -> Doc -> Doc` que concatena dos documentos. En un comentario dentro de la función **justificar por qué se satisface el invariante de** `Doc`.

Se declara `infixr 6 <+>`[1] para que `d1 <+> d2 <+> d3` sea equivalente a `d1 <+> (d2 <+> d3)`. También permite que expresiones como `texto "a" <+> linea <+> texto "c"` sean válidas sin la necesidad de usar paréntesis.

Estas serían algunas aserciones verdaderas:

- `vacio <+> vacio == vacio`
- `texto "a" <+> texto "b" == texto "ab"`
- `(linea <+> texto "a") <+> texto "b" == linea <+> (texto "a" <+> texto "b")`

Ejercicio 3

Definir la función `indentar :: Int -> Doc -> Doc` que agrega la cantidad (mayor que cero) de espacios indicada al comienzo de cada línea del documento. No modifica la primera linea del documento, sino las que siguen. En un comentario dentro de la función **justificar por qué se satisface el invariante de** `Doc`. Ejemplo:

```
indentar 2 (texto "a" <+> linea <+> texto "b" <+> linea <+> texto "c") ⇝
   "Texto "a" (Linea 2 (Texto "b" (Linea 2 (Texto "c" Vacio))))"
```

[1] `infixr`, `infixl`, `infix` son la forma en la cuál Haskell nos permite definir la asociatividad y precedencia de operadores nuevos.

Ejercicio 4

Definir la función `mostrar :: Doc -> String` que convierte el documento en un string listo para mostrar en pantalla. Estas serían algunas aserciones verdaderas:

- `mostrar vacio == ""`
- `mostrar (texto "abc") == "abc"`
- `mostrar (indentar 2 (texto "abc" <+> linea <+> texto "def")) == "abc\n␣␣def"`
- `mostrar (indentar 2 (texto "a" <+> linea <+> texto "b" <+> linea <+> texto "c")) == "a\n␣␣b\n␣␣c"`

Cada ␣ representa un espacio.

Se dispone de una función `imprimir :: Doc -> IO ()` que imprime un documento en pantalla según la función `mostrar`. Por ejemplo:

```
ghci> imprimir (indentar 2 (texto "abc" <+> linea <+> texto "def"))
abc
␣␣def

ghci> imprimir (texto "abc" <+> (indentar 2 linea <+> texto "def"))
abc
␣␣def
```

A continuación algunos ejemplos de expresiones tipo `Doc`, su valor y cómo se muestra en pantalla.

```
ghci> d1 = vacio
ghci> d1
Vacio
ghci> imprimir d1

ghci> d2 = indentar 2 (texto "a" <+> linea <+> texto "b" <+> linea <+> texto "c")
ghci> d2
Texto "a" (Linea 2 (Texto "b" (Linea 2 (Texto "c" Vacio))))
ghci> imprimir d2
a
␣␣b
␣␣c
ghci> d3 = indentar 2 (texto "a" <+> linea <+> texto "b") <+> linea <+> texto "c"
ghci> d3
Texto "a" (Linea 2 (Texto "b" (Linea 0 (Texto "c" Vacio))))
ghci> imprimir d3
a
␣␣b
c
ghci> d4 = texto "a" <+> texto "b"
ghci> d4
Texto "ab" Vacio
ghci> imprimir d4
ab
```

Módulo `PPON`

Se define el tipo de dato `PPON` que sirve para almacenar datos complejos. Cualquier similitud con JSON es pura coincidencia. Un `PPON` puede ser un texto, un entero o un objeto que contiene una lista de pares clave-valor con claves `String` y valores `PPON`es. Donde su definición es:

```
data PPON = TextoPP String | IntPP Int | ObjetoPP [(String, PPON)]
```

Estos son ejemplos de PPONes:

```
pericles = ObjetoPP [("nombre", TextoPP "Pericles"), ("edad", IntPP 30)]
merlina = ObjetoPP [("nombre", TextoPP "Merlina"), ("edad", IntPP 24)]
addams = ObjetoPP [("0", pericles), ("1", merlina)]
```

Ejercicio 5

Definir un predicado `pponAtomico :: PPON -> Bool` que determine si un PPON es atómico, es decir, si es un `TextoPP` o un `IntPP`.

Ejercicio 6

Definir un predicado `pponObjetoSimple :: PPON -> Bool` que determine si un PPON es un objeto con valores atómicos. Ejemplo:

- `pponObjetoSimple pericles` ⇝`True`
- `pponObjetoSimple addams` ⇝`False`

Queremos definir una función que nos permita mostrar un `PPON` en forma de texto. Para ello, vamos a definir algunas funciones auxiliares que si bien operan sobre `Doc`, se implementan en el módulo `PPON` ya que surgen de una necesidad específica de mostrar un `PPON`. Dado que `Documento` no exporta los constructores no vamos a poder hacer *pattern matching* sobre los documentos ni usar los constructores para crear valores de tipo `Doc`.

Ejercicio 7

Definir la función `intercalar` que recibe un separador y una lista de documentos y devuelve un documento que contiene a los documentos de la lista intercalados por el separador. Ejemplo: `mostrar (intercalar (texto ",␣") [texto "a", texto "b", texto "c"])` ⇝`"a,␣b,␣c"`

Usando las funciones hasta ahora definidas podemos crear la función `entreLlaves` que dada una lista de documentos devuelve un documento que contiene a los documentos de la lista entre llaves. Cada documento de la lista estará en una nueva linea, termina con una coma, y está indendato con dos espacios. Si la lista está vacía, el documento resultante es `{␣}`.

```
entreLlaves :: [Doc] -> Doc
entreLlaves [] = texto "{ }"
entreLlaves ds =
  texto "{"
    <+> indentar
      2
      ( linea
          <+> intercalar (texto "," <+> linea) ds
      )
      <+> linea
      <+> texto "}"
```

```
ghci> imprimir (entreLlaves [texto "a", texto "b", texto "c"])
{
␣␣a,
␣␣b,
␣␣c
}
```

Ejercicio 8

Definir la función `aplanar` que recibe un documento y devuelve un documento equivalente pero reemplazando los saltos de linea y su indentación por un espacio.

```
ghci> imprimir (aplanar (texto "a" <+> linea <+> texto "b"))
a␣b
ghci> imprimir (aplanar (texto "a" <+> indentar 2 (linea <+> texto "b")))
a␣b
```

Ejercicio 9

Definir la función `pponADoc :: PPON -> Doc` que recibe un `PPON` y devuelve un documento que lo representa. Se puede usar recursión explícita. En un comentario dentro de la función **justificar qué tipo de recursión se usa: estructural, primitiva o global**. La función debe cumplir con las siguientes condiciones:

- Si el `PPON` es un `TextoPP`, el documento resultante debe ser el texto entre comillas dobles. Usar la función `show :: String -> String` ya disponible en el preludio para obtener las comillas dobles y escapar los caracteres especiales.
- Si el `PPON` es un `IntPP`, el documento resultante debe ser el número. Usar la función `show :: Int -> String` ya disponible en el preludio para obtener el número como un `String`.
- Si el `PPON` es un `ObjetoPP`, el documento resultante debe ser un objeto expresado como `{"clave_1": valor_1, "clave_2": valor_2, ...}`, donde cada clave/valor está en una línea distinta, las claves son un texto entre comillas dobles y los valores es el documento asociado al valor de cada clave. Notar las llaves al comienzo y final del documento. (Ayuda: usar las funciones definidas en ejercicios anteriores).
- Si el valor es un objeto simple, debe mostrarse en una sola línea.

En el siguiente ejemplo Pericles y Merlina son objetos simples, por lo que se muestran en una sola línea. Addams es un objeto compuesto, por lo que se muestra en varias líneas.

```
ghci> imprimir (pponADoc addams)
{
  "0": { "nombre": "Pericles", "edad": 30 },
  "1": { "nombre": "Merlina", "edad": 24 }
}
```

Demostración

Ejercicio 10

Se pide demostar utilizando razonamiento ecuacional e inducción estructural que para todo `n, m :: Int` positivos y `x :: Doc`,

`indentar n (indentar m x) = indentar (n+m) x`

La demostración debe ser entregada en un archivo `indentar.txt` junto con el resto del código.

Se sugiere demostrar y usar los siguientes lemas:

- `indentar k Vacio = Vacio` para todo `k :: Int` positivo.
- `indentar k (Texto s d) = Texto s (indentar k d)` para todo `k :: Int` positivo, `s :: String` y `d :: Doc`.
- `indentar m (Linea k d) = Linea (m+k) (indentar m d)` para todo `m, k :: Int` positivos y `d :: Doc`.

Pautas de Entrega

Se debe entregar a través del campus un único archivo llamado "tp1.zip" conteniendo el código con la implementación de las funciones pedidas. Para eso, ya se encuentra disponible la entrega "TP1 - Programación Funcional" en la solapa "TPs" (configurada de forma grupal para que sólo una persona haga la entrega en nombre del grupo). El código entregado **debe** incluir tests que permitan probar las funciones definidas. El código debe poder ser ejecutado en Haskell2010. No es necesario entregar un informe sobre el trabajo, alcanza con que el código esté **adecuadamente** comentado (son comentarios adecuados los que ayudan a entender lo que no es evidente o explican decisiones tomadas; no son adecuadas las traducciones al castellano del código). Los objetivos a evaluar son:

- Corrección.
- Declaratividad.
- Prolijidad: evitar repetir código innecesariamente y usar adecuadamente las funciones previamente definidas (tener en cuenta tanto las funciones definidas en el enunciado como las definidas por ustedes mismos).
- Uso adecuado de funciones de alto orden, currificación y esquemas de recursión: Es necesario para los ejercicios que usen las funciones que vimos en clase y aquellas disponibles en la sección Útil del campus y aprovecharlas, por ejemplo, usar zip, map, filter, take, takeWhile, dropWhile, foldr, foldl, listas por comprensión, etc, cuando sea necesario y no volver a implementarlas.

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 listas por comprensión y esquemas de recursión definidos en el preludio de Haskell y los módulos `Prelude`, `List`, `Maybe`, `Data.Char`, `Data.List`, `Data.Map`, `Data.Function`, `Data.Maybe`, `Data.Ord` y `Data.Tuple`. Las sugerencias de los ejercicios pueden ayudar, pero no es obligatorio seguirlas. Pueden escribirse todas las funciones auxiliares que se requieran, pero estas no pueden usar recursión explícita (ni mutua, ni simulada con `fix`).

Tests: cada ejercicio debe contar con uno o más ejemplos que muestren que exhibe la funcionalidad solicitada. Para esto se recomienda la codificación de tests usando el paquete HUnit https://hackage.haskell.org/package/HUnit. El esqueleto provisto incluye algunos ejemplos de cómo utilizarlo para definir casos de test para cada ejercicio.

También se dispone de un `Makefile` que permite compilar y ejecutar los tests de los ejercicios y descargar HUnit directamente.

- `make test` compila y ejecuta todos los tests.
- `make repl` inicia GHCi con los módulos cargados.
- `make watch` inicia GHCid y ejecuta los tests cada vez que se modifica un archivo.

Si prefieren pueden usar `cabal` para instalar HUnit y ejecutar los tests manualmente.

Para instalar HUnit usar: `> cabal install --lib HUnit`

Para instalar cabal pueden usar `> ghcup install cabal recommended --set`

Para ejecutar los tests usar: `> main` para todos los tests, `> do runTestTT testsEjN` para los tests del ejercicio N.

Importante: Se espera que la elaboración de este trabajo sea 100 % de los estudiantes del grupo que realiza la entrega. Así que, más allá de que pueden tomar información de lo visto en las clases o consultar información en la documentación de Haskell o disponible en Internet, no se podrán utilizar herramientas para generar parcial o totalmente en forma automática la resolución del TP (e.g., chat-GPT, copilot, etc). En caso de detectarse esto, el trabajo será considerado como un plagio, por lo que será gestionado de la misma forma que se resuelven las copias en los parciales u otras instancias de evaluación.

Referencias del lenguaje Haskell

Como principales referencias del lenguaje de programación Haskell, mencionaremos:

- **The Haskell 2010 Language Report:** el reporte oficial de la última versión del lenguaje Haskell a la fecha, disponible online en http://www.haskell.org/onlinereport/haskell2010.
- **Learn You a Haskell for Great Good!**: libro accesible, para todas las edades, cubriendo todos los aspectos del lenguaje, notoriamente ilustrado, disponible online en http://learnyouahaskell.com/chapters.
- **Real World Haskell**: libro apuntado a zanjar la brecha de aplicación de Haskell, enfocándose principalmente en la utilización de estructuras de datos funcionales en la "vida real", disponible online en http://book.realworldhaskell.org/read.
- **Hoogle**: buscador que acepta tanto nombres de funciones y módulos, como signaturas y tipos *parciales*, online en http://www.haskell.org/hoogle.
- **Hayoo!**: buscador de módulos no estándar (i.e. aquéllos no necesariamente incluídos con la plataforma Haskell, sino a través de **Hackage**), online en http://holumbus.fh-wedel.de/hayoo/hayoo.html.