Trabajo Práctico 1
Programación Funcional

Paradigmas de Lenguajes de Programación
2do cuatrimestre, 2015

Fecha de entrega: martes 8 de septiembre

1. Introducción

En la programación web 'clásica' una dirección URL se correspondía con un archivo en el servidor web, ya sea con contenido estático o dinámico. Así, por ejemplo, para procesar la URL `http://facultad.edu.ar/materiasAprobadas.jsp?lu=007-01`, se requería que existiera una página llamada `materiasAprobadas.jsp` que recibiera un parámetro `lu`.

Más recientemente se empezó a utilizar otro método denominado *ruteo de URL* (del inglés *URL Routing*). Así, se procesa la cadena con la URL peticionada para determinar la acción a ejecutar. Por ejemplo, en nuestro caso anterior podríamos tener una petición con la URL `http://facultad.edu.ar/alu/007-01/aprobadas`, y que al descomponer el contenido de la URL se pueda saber qué acción se debe seguir y cuáles son los parámetros para esa acción.

Una ruta permite asociar un patrón de una URL con quien se encarga de manejarla: el *handler*. Un *handler* puede ser un archivo existente, como el `materiasAprobadas.jsp` del primer ejemplo, o apuntar a una función que se encarga de manejar la petición. Un patrón quedará definido por una secuencia de literales o grupos de captura separados por barras ('/'): los literales deberán coincidir el forma exacta, mientras que las capturas (obtenidas de las partes prefijados por ':') guardarán el texto de la parte correspondiente (sin el ':').

Por ejemplo, dado el patrón `materia/:nombre/alu/:lu/inscribir`, la URL solicitada `http://facultad.edu.ar/materia/plp/alu/007-01/inscribir` coincidirá con el patrón y se guardará en las capturas `plp` y `007-01`, respectivamente. Por otra parte, no habrá coincidencia con la solicitud `http://facultad.edu.ar/materia/plp`.

En este trabajo buscaremos implementar el manejo de patrones y rutas por medio de programación funcional en Haskell, de forma de escribir patrones asociados a valores de una forma amigable.

Un ejemplo de un ruteo a Strings válido es:

Ruta	Valor
(ruta vacía)	`"ver inicio"`
`ayuda`	`"ver ayuda"`
`materia/:nombre/alu/:lu/inscribir`	`"inscribe alumno"`
`materia/:nombre/alu/:lu/aprobar`	`"aprueba alumno"`
`alu/:lu/aprobadas`	`"ver materias aprobadas por alumno"`

el cual podría quedar definido como

```
rutasFacultad = many [
  route ""                "ver inicio",
  route "ayuda"           "ver ayuda",
  scope "materia/:nombre/alu/:lu" $ many [ route "inscribir"   "inscribe alumno",
                                           route "aprobar"     "aprueba alumno"    ],
  route "alu/:lu/aprobadas"  "ver materias aprobadas por alumno" ]
```

Donde `many`, `scope` y `route` son funciones que generan rutas. Más adelante veremos en detalle su implementación.

Luego, con alguna función que reciba `rutasFacultad` y `"alu/007-01/aprobadas"` esperamos obtener `"ver materias aprobadas por alumno"` y `("lu", "007-01")`.

En este ejemplo los valores asociados a una ruta son Strings, pero podrían ser funciones, y en particular funciones que reciban las capturas del patrón.

2. Diseño

La implementación quedará disponible en un solo módulo, denominado `Router`, dentro del cual se utilizarán dos tipos de datos propios:

- `data PathPattern = Literal String | Capture String`
  para la definición de los patrones, donde el primer constructor define una cadena literal, y el segundo permite la definición de capturas.
- `data Routes f = Route [PathPattern] f | Scope [PathPattern] (Routes f) | Many [Routes f]`
  para la definición de las rutas, donde el primer caso define una ruta con la lista de patrones (sucesión de literales y capturas) y el *handler* correspondiente, la segunda define la ruta actual como subruta de otra definida con otro patrón, y, por último, se puede definir una ruta como la unión de otras rutas con el mismo tipo de *handler*.

Finalmente, los valores de cada captura quedarán registrados en un `PathContext`, definido como una lista de pares de Strings `(captura, valor)`:
`type PathContext = [(String, String)]`

3. Implementación

Para poder implementar el manejo de rutas se deberá desarrollar las funcionalidades descriptas a continuación.

Ejercicio 1

`split :: Eq a ⇒ a → [a] → [[a]]` Dado un elemento separador y una lista, se deberá partir la lista en sublistas de acuerdo a la aparición del separador (sin incluirlo).

Por ejemplo,

```
split '/' "alu/:lu/aprobadas" ⇝ ["alu",":lu","aprobadas"]
split '|' "|hola|que|tal|" ⇝ ["","hola","que","tal",""]
split '*' "hallo*wie*geht's**" ⇝ ["hallo","wie","geht's","",""]
split '*' "" ⇝ [""]
split '*' "*" ⇝ ["",""]
```

Ejercicio 2

`pattern :: String → [PathPattern]` A partir de una cadena que denota un patrón de URL se deberá construir la secuencia de literales y capturas correspondiente. Por ejemplo,
`pattern "alu/:lu/aprobadas" ⇝ [Literal "alu", Capture "lu", Literal "aprobadas"]`

Por comodidad se espera que no sea sensible a `'/'` superfluas: al comienzo, final, o consecutivas:

```
pattern "/alu/:lu/aprobadas" ⇝ [Literal "alu", Capture "lu", Literal "aprobadas"]
pattern "/alu/:lu//aprobadas/" ⇝ [Literal "alu", Capture "lu", Literal "aprobadas"]
```

Ejercicio 3

`get :: String → PathContext → String` Obtiene el valor registrado en una captura determinada. Se puede suponer que la captura está definida en el contexto.
Por ejemplo, `get "nombre" [("nombre","plp"), ("lu","007−1")] ⇝ "plp"`

Ejercicio 4

matches :: [String] → [PathPattern] → Maybe ([String], PathContext) Dadas una ruta particionada y un patrón de URL, trata de aplicar el patrón a la ruta y devuelve, en caso de que la ruta sea un prefijo válido para el patrón, el resto de la ruta que no se haya llegado a consumir y el contexto capturado hasta el punto alcanzado.
Por ejemplo,

```
matches ["materia","plp","alu","007-1"] [Literal "materia",Capture "nombre"] ⇝
  Just (["alu","007-1"],[("nombre","plp")])
matches ["otra","ruta"] [Literal "ayuda"] ⇝ Nothing
matches [] [Literal "algo"] ⇝ Nothing
```

Para este ejercicio se permite usar recursión explícita.

DSL Routing

Se definen las siguientes funciones, que conforman un modesto DSL (*Domain Specific Language*) para definir en forma cómoda las rutas, tal como fueron utilizadas en los ejemplos de la Introducción.

```
route :: String → a → Routes a
route s f = Route (pattern s) f

scope :: String → Routes a → Routes a
scope s r = Scope (pattern s) r

many :: [Routes a] → Routes a
many l = Many l
```

Ejercicio 5

Definir el *fold* para el tipo `Routes f`. Se puede usar recursión explícita.

Ejercicio 6

paths :: Routes a → [String] Genera todos los posibles *paths* para una ruta definida.
Por ejemplo,

```
paths rutasFacultad ⇝ ["","ayuda","materia/:nombre/alu/:lu/inscribir",
                       "materia/:nombre/alu/:lu/aprobar","alu/:lu/aprobadas"]
```

Ejercicio 7

eval :: Routes a → String → Maybe (a, PathContext) Evalúa un path con una definición de ruta y, en caso de haber coincidencia, obtiene el handler correspondiente y el contexto de capturas obtenido. Por ejemplo,

```
eval rutasFacultad "alu/007-01/aprobadas" ⇝
  Just ("ver materias aprobadas por alumno",[("lu","007-01")])
eval rutasFacultad "materia/plp/alu/007-01/aprobar" ⇝
  Just ("aprueba alumno",[("lu","007-01"),("nombre","plp")])
eval rutasFacultad "alu/007-01" ⇝ Nothing
```

Sugerencia: usar `foldRoutes`, `matches` y `(=<<) :: (a->Maybe b)->Maybe a->Maybe b`. Mirar bien el tipo de `eval` y pensar a quién se aplica el resultado recursivo en el caso de `Scope`.

Ejercicio 8

exec :: Routes (PathContext → a) → String → Maybe a Similar a eval, pero aquí se espera que el handler sea una función que recibe como entrada el contexto con las capturas, por lo que se devolverá el resultado de su aplicación, en caso de haber coincidencia.

```
let rutasStringOps = route "concat/:a/:b" (\ctx → (get "a" ctx) ++ (get "b" ctx)) in
  exec rutasStringOps "concat/foo/bar" ⇝ Just "foobar"
```

Ejercicio 9

wrap :: (a → b) → Routes a → Routes b Permite aplicar una función sobre el handler de una ruta. Esto, por ejemplo, podría permitir la ejecución consecutiva de dos handlers.

```
eval (wrap reverse rutasFacultad) "ayuda" ⇝ Just ("aduya rev",[])
exec (wrap (\f ctx → f ctx ++ ".") rutasStringOps) "concat/foo/bar" ⇝ Just "foobar."
```

Ejercicio 10 (opcional)

catch_all :: a → Routes a Genera un Routes que captura todas las rutas, de cualquier longitud. El handler para todos los patrones es siempre el mismo. Las capturas usadas en los patrones se deberán llamar p0, p1, etc. En este punto se permite recursión explícita.

Para ilustrar catch_all f debería ser equivalente a

```
many [  route "" f,
        scope ":p0" $ many [ route "" f,
                             scope ":p1" $ many [ route "" f,
                                                  scope ":p2" $ many [ route "" f,
                                                                       ... etc ...
                                                                     ]
                                                ]
                           ]
     ]
```

Por ejemplo,

```
take 5 (paths (catch_all 42)) ⇝ ["","/:p0","/:p0/:p1","/:p0/:p1/:p2","/:p0/:p1/:p2/:p3"]
eval (catch_all 42) "cualquier/ruta" ⇝ Just (42,[("p0","cualquier"),("p1","ruta")])
exec (catch_all length) "" ⇝ 0
exec (catch_all length) "foo/bar/baz" ⇝ 3
```

Pautas de Entrega

Se debe entregar el código impreso con la implementación de las funciones pedidas. Cada función debe contar con un comentario donde se explique su funcionamiento. Cada función asociada a los ejercicios debe contar con ejemplos que muestren que exhibe la funcionalidad solicitada. Además, se debe enviar un e-mail conteniendo el código fuente en Haskell a la dirección plp-docentes@dc.uba.ar. Dicho mail debe cumplir con el siguiente formato:

- El título debe ser `[PLP;TP-PF]` seguido inmediatamente del nombre del grupo.
- El código Haskell debe acompañar el e-mail y lo debe hacer en forma de archivo adjunto (puede adjuntarse un `.zip` o `.tar.gz`).
- 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.

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.Function`, `Data.List`, `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`).

Importante: se admitirá un único envío, sin excepción alguna. Por favor planifiquen el trabajo para llegar a tiempo con la entrega.

Tests: se recomienda la codificación de tests. Tanto HUnit https://hackage.haskell.org/package/HUnit como HSpec https://hackage.haskell.org/package/hspec permiten hacerlo con facilidad.

Para instalar HUnit usar: `> cabal install hunit`

Para instalar cabal ver: https://wiki.haskell.org/Cabal-Install

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.