Laboratorio de Lenguajes de Programación

USB / CI-3661 / Sep-Dic 2015

(Programación Funcional – 35 puntos)

Funciones de Orden Superior (6 puntos)

Considere la función `filter` discutida en clase

```
filter :: (a -> Bool) -> [a] -> [a]
```

Si bien la implantación de `filter` es directamente recursiva, y conveniente según la estrategia de evaluación de Haskell, se desea que Ud. implemente `filter` de maneras diferentes:

1. Usando listas por comprensión.

   ```
   filterC = undefined
   ```

2. Usando un `map`.

   ```
   filterM = undefined
   ```

   Posiblemente tenga que apoyarse en funciones auxiliares vía composición, pero no puede haber recursión directa.

3. Usando un `foldr`:

   ```
   filterF = undefined
   ```

Verificador de Tautologías (12 puntos)

Representación de Expresiones

Las expresiones de Lógica Proposicional de Primer Orden pueden definirse recursivamente como sigue:

- Una constante *booleana* `True` o `False` es una expresión de Lógica Proposicional de Primer Orden.
- Una variable, representada por un `String` es una expresión de Lógica Proposicional de Primer Orden.
- La negación de una expresión de Lógica Proposicional de Primer Orden, es una expresión de Lógica Proposicional de Primer Orden.
- La conjunción de dos expresiones de Lógica Proposicional de Primer Orden, es una expresión de Lógica Proposicional de Primer Orden.
- La disyunción de dos expresiones de Lógica Proposicional de Primer Orden, es una expresión de Lógica Proposicional de Primer Orden.
- La implicación de dos expresiones de Lógica Proposicional de Primer Orden, es una expresión de Lógica Proposicional de Primer Orden.

Defina un tipo recursivo monomórfico Haskell para representar las expresiones de Lógica Proposicional. La definición del tipo debe incluir la cantidad *mínima* de instancias de clases de tipo necesarias para la funcionalidad que se solicita en el resto de este ejercicio.

```
data Proposition = undefined
```

Ambiente de Evaluación

Para poder evaluar el valor de verdad de una proposición particular es necesario contar con los valores de verdad asociados a las variables involucradas. Esto se denomina el *Ambiente de Evaluación* y será modelado con un tipo de datos simple

```
type Environment = [(String,Bool)]
```

Escriba la función

```
find :: Environment -> String -> Maybe Bool
find e k = undefined
```

que determina si la variable `k` está definida en el ambiente `e`, produciendo su valor *booleano* en caso afirmativo.

Escriba la función

```
addOrReplace :: Environment -> String -> Bool -> Environment
addOrReplace e k v = undefined
```

tal que:

- Si en el ambiente `e` no existe ninguna asociación para la variable `k`, la función produce un nuevo ambiente igual al original pero agregando la asociación `(k,v)` al **principio**.
- Si en el ambiente `e` ya existe una asociación para la variable `k`, la función produce un nuevo ambiente igual al original **reemplazando** la asociación existente por la nueva.

Escriba la función

```
remove :: Environment -> String -> Environment
remove e k = undefined
```

tal que:

- Si en el ambiente `e` no existe ninguna asociación para la variable `k`, la función produce el mismo ambiente sin modificar.
- Si en el ambiente `e` existe una asociación para la variable `k`, la función produce un nuevo ambiente igual al original **eliminando** la asociación existente.

Evaluando valores de verdad

Teniendo esa función, podemos determinar el valor de verdad de una expresión arbitraria. Para ello, escriba la función

```
evalP :: Environment -> Proposition -> Maybe Bool
evalP e p = undefined
```

que recorre la estructura recursiva de `Proposition` y se apoya en el ambiente `e` para determinar si la proposición `p` tiene un valor de verdad, calculándolo.

The truth shall set you free!

Una Proposición es una Tautología si para todas las combinaciones de valores de verdad de las variables empleadas, siempre se evalúa al valor de verdad `True`.

Comience por escribir la función

```
vars :: Proposition -> [String]
vars p = undefined
```

que extrae los nombres de todas las variables presentes en la proposición `p`. Si una variable aparece más de una vez en la proposición, debe aparecer **una sola vez** en la lista.

Escriba ahora la función

```
isTautology :: Proposition -> Bool
isTautology p = undefined
```

tal que determine si la proposición `p` es una Tautología. El algoritmo general consiste en general todos los Ambientes de Evaluación resultantes de asignar todas las combinaciones de valores de verdad a las variables presentes en la proposición, evaluarla, y determinar si siempre tiene valor de verdad `True`.

Para escribir esta función:

- No puede usar las funciones `length` ni `reverse` – si implanta su propia función `length` o `reverse`, será penalizado.
- Sólo puede usar funciones de orden superior o listas por comprensión. No puede usar recursión directa.
- Las dos condiciones anteriores aplican incluso para las funciones auxiliares que Ud. escriba para simplificar su solución.

Lambda-Jack (17 puntos)

Lambda-Jack es una variante simple del juego de barajas *Black Jack* – si nunca ha jugado Black-Jack, bien por Ud. porque es una manera muy fácil de perder dinero.

Las reglas

Nuestro juego sólo tiene dos jugadores, Ud. y *Lambda*, este último cibernético.

El primero en jugar siempre es Ud. que puede pedir (*hit me!*) cartas una a una siempre y cuando el total de sus cartas no exceda 21. Si su total excede 21, Ud. explotó (*bust*) y perdió el juego. Pero si Ud. se detiene antes de explotar, será el turno de *Lambda*. En su turno, *Lambda* pedirá cartas hasta tener al menos un total de 16, y allí se detiene. Si el total de las cartas de *Lambda* excede 21, Ud. gana porque *Lambda* explotó.

El valor de las cartas en mano es la suma de los valores individuales de las cartas, que son como sigue:

- Cada carta numérica tiene el valor indicado por su número, e.g., el seis vale 6.

- Las figuras (Jack, Queen, King) valen 10 puntos cada una.
- Un As puede valer 1 u 11. Cuando se calcula el valor de la mano *todos* los ases tienen el mismo valor (todos 1 o todos 11). Inicialmente, se usa el valor 11 para el primer As, pero si el valor total de todas las cartas excede 21, todos los ases pasan a valer 1.

El jugador que tenga la puntuación más alta, sin exceder 21, gana la partida. Si ambos jugadores tienen la misma puntuación, gana *Lambda*. Si ambos jugadores explotan, gana *Lambda*.

Ud. escribirá un programa interactivo que permita jugar Lambda-Jack tanto tiempo como se desee.

Modelado de las Cartas

Escriba un módulo Haskell titulado `Cards` que contendrá los tipos de datos y funciones necesarias para la manipulación de cartas. Su módulo debe estar correctamente documentado y exportar *solamente* lo necesario.

Lambda-Jack se juega con baraja francesa, así que necesitaremos un tipo de datos para representarla. Tendremos

```
data Card = Card {
                   value :: Value,
                   suit  :: Suit
                 }
```

donde `Value` es un tipo de datos auxiliar que debe modelar los valores (no numéricos) para las cartas, mientras que `Suit` representa los «palos» de la baraja.

```
data Suit = Clubs | Diamonds | Spades | Hearts

data Value = Numeric Int | Jack | Queen | King | Ace
```

Para simplificar la interacción con el juego, las cartas deben presentarse en pantalla usando los guarismos y símbolos asociados con el juego. Por ejemplo, las cartas

```
Card Ace Diamonds
Card (Numeric 10) Spades
```

deben presentarse en pantalla como

♦*A*

♤10

Para poder modelar el juego, es necesario controlar las cartas que tiene en mano tanto *Lambda* como Ud. y para esto tendremos el tipo de datos

```
newtype Hand = H [Card]
```

Nuevamente, y con la intención de que sea fácil la interacción, si se tiene la mano

```
H [
    Card Ace Diamonds,
    Card (Numeric 9) Spades,
    Card Jack Hearts
  ]
```

esta debe mostrarse en pantalla como

♦*A* ♤9 ♥*J*

Escriba la función

```
empty :: Hand
```

que produce una mano vacía.

Escriba la función

```
size :: Hand -> Int
```

que determine la cantidad de cartas en una mano.

Agregue las instancias *mínimas* necesarias a los tipos de datos para que la implantación de la funcionalidad solicitada sea posible.

Modelado del Juego

Escriba un módulo Haskell titulado `LambdaJack` que contendrá los tipos de datos y funciones necesarias para implementar las reglas del juego. Su módulo debe estar correctamente documentado y exportar *solamente* lo necesario.

Para comenzar, sólo hay dos jugadores

```
data Player = LambdaJack | You
```

Escriba la función

```
value :: Hand -> Int
```

que determina el valor numérico de una mano en particular. Para escribir la función emplee en lo posible funciones de orden superior o de la librería `Data.List` evitando usar recursión directa. En particular, está función se puede escribir como un `fold` en *una* sola pasada por la lista de cartas.

Una vez conocido el valor de una «mano», conviene tener un predicado simple

```
busted :: Hand -> Bool
```

que indica si la «mano» explotó por exceder 21. Además, podrá escribir la función

```
winner :: Hand -> Hand -> Player
```

que permita comparar su «mano» con la «mano» de Lambda, para determinar el ganador.

En nuestra versión simplificada del juego, se comienza con un mazo de cartas completo. Por lo tanto, es necesario que implemente la función

```
fullDeck :: Hand
```

que produce una «mano» conteniendo las 52 cartas de la baraja en cualquier orden (el mazo completo). Implante esta función usando listas por comprensión.

Durante el juego, un jugador tendrá ciertas cartas en la mano y decidirá que quiere una carta más. Para este caso, implante la función

```
draw :: Hand -> Hand -> Maybe (Hand,Hand)
```

tal que siendo su primer argumento el mazo restante y su segundo argumento la mano del jugador, retire la siguiente carta del mazo de ser posible, la agregue a la mano del jugador y retorne una tupla con la lo que resta del mazo y la nueva mano del jugador, en ese orden.

Ud. puede jugar como le venga en gana, igual va a perder a la larga, sin embargo *Lambda* juega con reglas específicas. Entonces, escriba la función

```
playLambda :: Hand -> Hand
```

tal que reciba el mazo (tal como quedó después que jugó Ud.) y produzca la «mano» que se construye siguiendo las reglas con las que juega *Lambda*.

Finalmente, necesitamos barajar las cartas del mazo antes de cada partida. El módulo estándar `System.Random` provee un tipo de datos `StdGen` que asiste en la generación de números al azar, así como una serie de funciones adicionales de suma utilidad. Una vez que haya estudiado el módulo, escriba la función

```
shuffle :: StdGen -> Hand -> Hand
```

tal que recibe como segundo argumento el «mazo» completo y lo baraja. Para que la barajada sea justa y al azar, Ud. *debe* implantar el siguiente algoritmo:

- Comience con un *nuevo* «mazo» vacío para acumular.
- Seleccione una carta al azar del «mazo» a barajar y póngala al principio del «mazo» acumulador.
- Repita hasta que haya pasado todas las cartas del «mazo» a barajar hasta el «mazo» acumulador.

Si para este punto de la lectura se dió cuenta que eso es un `fold`, va muy bien.

El mundo real

Para completar su aplicación y tener un programa ejecutable, tendrá que lidiar con el mundo exterior. Su programa principal, el módulo `Main`, debe estar en un archivo separado.

El programa principal debe comportarse de la siguiente forma:

- Debe emitir un breve mensaje indicando “Bienvenido a LamdaJack” y preguntar el nombre del jugador. Escriba una función

  welcome :: IO String

para esa funcionalidad.

- La función principal de juego debe:
- Indicar cuántas partidas se han jugado hasta ahora, cuántas ha ganado *Lambda* y cuántas ha ganado el jugador, usando el nombre del jugador, por supuesto. El mensaje en pantalla debe ser

  ```
  Después de 12 partidas Lambda ha ganado 9 y Bob ha ganado 3
  ```

  Escriba una función

  ```
  currentState :: GameState -> IO ()
  ```

  para ese propósito.
- Preguntar al jugador si desea seguir jugando o no. En caso negativo, terminar el programa. Escriba una función

```
continuePlaying :: IO Bool
```

para ese propósito.

- Si el jugador desea seguir jugando, la función prepara un mazo nuevo, lo baraja, y genera una mano inicial con dos cartas para el jugador. Inmediatamente debe presentar las cartas al jugador, indicar su puntuación, y preguntar si desea otra carta o «se queda». El mensaje en pantalla debe ser

  Bob, tu mano es ♦6 ♤7 Suma 13. ¿Carta o Listo?

  y aceptar *solamente* las letras 'c' o 'l', independientemente de mayúsculas o minúsculas.

- Mientras el jugador quiera cartas, debe repetirse el comportamiento anterior. Si el valor de la mano excede 21, debe indicarse la derrota

  Bob, tu mano es ♦6 ♤7 ♤9, suma 21. Perdiste.

  Si el jugador decide quedarse, debe indicarse el cambio de turno

  Bob, tu mano es ♤10 ♤9 ♤*A*, suma 20. Mi turno.

- Cuando el jugador pierda o decida quedarse, *Lambda* toma su turno, muestra su mano final y anuncia el resultado, indicando

  Mi mano es ♤10 ♤*A*, suma 21.

  Dependiendo del resultado del juego, en el lugar de debe escribirse 'Yo gano', 'Tu ganas', 'Empatamos, así que yo gano.'

Implante la función principal como

```
gameloop :: GameState -> IO ()
```

donde `GameState` es un tipo de datos para llevar el estado del juego

```
data GameState = GS {
                      games     :: Int,
                      lamdaWins :: Int,
                      name      :: String,
                      generator :: StdGen
                    }
```

Es probable que quiera implantar funciones auxiliares para los mensajes parametrizados, ya que son similares entre si.

Detalles de la Entrega

La entrega se hará en un archivo `pH-<carnet1>-<carnet2>.tar.gz`, donde `<carnet1>` y `<carnet2>` son los números de carnet de ambos integrantes del grupo. El archivo *debe* estar en formato TAR comprimido con GZIP – ignoraré, sin derecho a pataleo, cualquier otro formato que yo puedo descomprimir pero que *no quiero* recibir.

Ese archivo, al expandirlo, debe producir un *directorio* que *sólo* contenga:

- El archivo `hof.hs` con la solución a la sección **Funciones de Orden Superior**
- El archivo `true.hs` con la solución a la sección **Verificador de Tautologías**
- Los archivos `lambdajack.hs`, `Cards.hs` y `LambdaJack.hs` con su implantación de la sección **Lambda-Jack**

  Debe escribir sus módulos de manera tal que sea posible compilarlos haciendo

  ```
  ghc --make lambdajack.hs
  ```

  para que produzca el ejecutable `lambdajack`.

El proyecto debe ser entregado por correo electrónico a mi dirección de contacto a más tardar el viernes 2015-10-16 a las 23:59. Cada minuto de retraso en la entrega le restará un (1) punto de la calificación final.