Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información.
CI–3661 – Laboratorio de Lenguajes de Programación.
Enero–Marzo 2020

Proyecto I: Haskell
Jack "El Monádico" Lambda
(25 pts)

En la ciudad de New Haskell no hay quien no conozca el nombre de Jack Lambda, apodado como *"El Monádico"*. Este infame sujeto, conocido por currificar las cartas a su favor, recorre las calles de la ciudad como si fuera su dueño junto a su banda de acérrimos seguidores, *"Los Functores"*. Ustedes, como tupla estelar de investigadores han preparado una ingeniosa trampa para atraparle. Sin embargo, el Sr. Lambda siempre parece tener precedencia sobre las fuerzas de la ley y logra a su vez que *"Los Functores"* bloqueen las vías de escape. Viéndose en peligro, pero no derrotado aún, *"El Monádico"* les propone decidir el desenlace de la situación en un juego de cartas: 21 Blackjack.

Las Reglas del Juego

Participarán dos personas: el *player* y el *dealer* (Jack Lambda). El primero en jugar siempre es el *player*, que recibe dos cartas del *dealer*. Una vez las tiene en su mano, puede tomar una de las siguientes acciones:

- **Hit**: Pide otra carta al *dealer*.
- **Stand**: Decide no recibir mas cartas, cediendo el turno al *dealer*.
- **Double down**: Duplica la apuesta, recibe otra carta, y cede el turno.
- **Surrender**: Después de recibir la mano inicial, el jugador puede decidir no jugar la ronda y rendirse, cediendo la mitad de su apuesta.

El *player* puede seguir pidiendo cartas una a una siempre y cuando el total de sus cartas no exceda 21. Si su total excede 21, pierde el juego (*bust*).

Una vez el *player* cede su turno al *dealer*, Jack Lambda pedirá cartas hasta tener al menos un total de 16, y se detiene. Si el total de las cartas del *dealer* excede 21, *then he's busted* y Jack Lambda es el perdedor de la ronda. El ganador de la ronda, en caso de que nadie exceda los 21, es quien posea la mano más valiosa. En caso de manos igualmente valiosas, el *dealer* es el ganador.

Para calcular el valor de una mano, se suma el valor individual de cada carta, calculado de la siguiente manera:

- Las *court cards* (Jack, Queen, King) valen 10 puntos cada una.
- El As valdrá 11 puntos, a menos que esto cause que la mano se exceda de 21, en cuyo caso todos los Ases pasarán a valer 1 punto.
- El resto de las cartas simplemente conservan su valor numérico.

Una mano que tenga un valor de 21 únicamente con las dos primeras cartas, es llamada un *Blackjack*, y tiene más valor que cualquier otra mano que sume 21 con un número mayor de cartas. Si el *dealer* tiene un blackjack en sus dos primeras cartas, gana automáticamente la ronda y el *player* no puede rendirse.

El usuario empezará el juego con una cantidad de dinero, una cantidad fija que se le permite apostar en cada ronda, y una cantidad objetivo de dinero para ganar la partida. Si el usuario se queda sin dinero suficiente para apostar, pierde.

El objetivo de este proyecto es escribir un programa interactivo que permita jugar 21 Blackjack hasta que el usuario desee detenerse, y que, mientras no se esté jugando una ronda, la partida pueda ser guardada y retomada en otro momento.

Tipo de Datos: Carta

El tipo de datos `Carta` debe ser capaz de representar cualquier carta de la baraja francesa estándar de 52 cartas. Para ello debemos tener sendos tipos de datos para representar el palo y el rango de una carta:

```
data Palo = Treboles | Diamantes | Picas | Corazones

data Rango = N Int | Jack | Queen | King | Ace
```

Una vez teniendo esto, podemos definir el tipo de datos que represente una carta cualquiera:

```
data Carta = Carta {
                     rango :: Rango,
                     palo  :: Palo
                   }
```

Adicionalmente a estas definiciones, se deben suministrar las siguientes funciones:

- Instancias:
  - Una instancia de la clase `Show`, de manera que reciba una carta y devuelva una cadena de caracteres que contenga una representación de la misma *entendible* para el humano. Para ello, debe mostrar los símbolos asociados con el juego propiamente. Por ejemplo, las cartas:

    ```
    Card Ace Corazones
    Card (N 10) Picas
    ```

    deben presentarse en pantalla como:
    ♡*A*
    ♠10
  - Cualquier otra instancia que se estime necesaria para hacer posible la implementación.

Tipo de Datos: Jugador

Representa a cualquiera de los dos jugadores de la partida. Adicionalmente se deben suministrar instancias `Show` y `Read` que permitan generar representaciones en cadenas de caracteres y leerlas.

```
data Jugador = Dealer | Player
```

Tipo de Datos: Mano

Para controlar las cartas que tiene cada jugador en la mano, introduciremos el tipo de datos `Mano`.

```
newtype Mano = Mano [Card]
```

Adicionalmente a esta definición, se deben suministrar las siguientes funciones:

- Funciones de construcción:
  - `vacia :: Mano`
    Produce una `Mano` vacía.
  - `baraja :: Mano`
    Produce una `Mano` que contenga las 52 cartas de la baraja estándar francesa. Esto es, una de cada combinación palo-rango.
- Funciones de acceso:
  - `cantidad Cartas :: Mano ->Int`
    Determina la cantidad de cartas en una mano.
  - `valor :: Mano ->Int`
    Recibe una `Mano` y devuelve un entero con el valor de la misma.
  - `busted :: Mano ->Bool`
    Recibe una `Mano` y devuelve `True` si su valor excede los 21, y `False` de otra forma.
  - `blackjack :: Mano ->Bool`
    Recibe una `Mano` y devuelve `True` si la mano es un *blackjack* y `False` de otra forma. Recibe una `Mano` y devuelve `True` si su valor excede 21, y `False` de otra forma.
  - `ganador :: Mano ->Mano ->Jugador`
    Recibe la `Mano` del `Dealer` de primero, la mano del `Player` de segundo, y devuelve el ganador, según las reglas del juego antes descritas.
  - `separar :: Mano ->(Mano, Carta, Mano)`
    Recibe una `Mano` y la separa en una tupla `(l,c,r)` de la siguiente manera:
    - Si la mano es de longitud impar, `c` será el elemento del medio de la lista, y `l, r` serán respectivamente las mitades izquierda y derecha restantes.
    - Si la mano es de longitud par, `l` será la mitad izquierda, `c` será el primer elemento de la mitad derecha, y `r` será el resto de la mitad derecha.

    Esta función no tiene utilidad inmediata aparente, pero será de utilidad más adelante.
- Funciones de modificación:

- `barajar :: StdGen ->Mano ->Mano`
  Esta función será usada para barajar las cartas al inicio de cada ronda. Para ello, es necesario el tipo de datos `StdGen`, incluido en el módulo `System.Random`. El segundo argumento recibido por la función es la `Mano` a barajar, y debe devolverse la mano ya barajada. Para ello, se debe empezar por una `Mano` vacía para *acumular*. Luego, debe seleccionarse una carta al azar, la cuál debe ser colocada al principio de la `Mano` acumuladora. Esto debe hacerse hasta que se hayan pasado todas las cartas hasta la `Mano` acumuladora.
- `inicialLambda :: Mano ->(Mano, Mano)`
  Recibe la baraja inicial barajada como `Mano`, y devuelve la `Mano` inicial de Lambda tomando las dos primeras cartas, y la baraja resultante de retirar dichas cartas.

- Instancias:
  - Una instancia de la clase `Show` tal que reciba una carta y devuelve una cadena de caracteres que contenga una representación de la misma *entendible* para el humano. Al igual que con la mano, debe usar los símbolos apropiados. Por ejemplo, la mano:

```
Mano [
        Carta Ace Diamantes,
        Carta (N 9) Picas,
        Carta Jack Corazones
    ]
```

    deben presentarse en pantalla como:
    $\diamond A \spadesuit 9 \heartsuit J$
  - Cualquier otra instancia que se estime necesaria para hacer posible la implementación.

Tipo de Datos: Mazo

En los casinos, es frecuente la implementación de estrategias como el uso de varios mazos o máquinas para barajar, para así evitar que los jugadores astutos puedan sacar ventaja del conteo de cartas. Como sujeto que recorre en amplitud las calles de la ciudad, Jack Lambda no es ajeno a los subterfugios frecuentes en los juegos de azar. Es por eso que propone una manera especial de repartir las cartas:

1. Se empieza la ronda con las 52 cartas esperadas, y se baraja.
2. El *dealer* toma sus dos primeras cartas.
3. El *player* recibe su primera carta, y el mazo es divido a la mitad (una de las dos mitades siempre podrá contener una carta extra).
4. El *player* debe escoger de qué mitad sacar la otra carta, y la otra mitad se dejará de lado.
5. Cada vez que el *player* quiera pedir una carta, se dividirá el mazo nuevamente y se le pedirá escoger de qué mitad desea sacar la carta, dejando de lado también la otra mitad.
6. En caso de quedarse sin cartas para seguir jugando, debe tomar todas las cartas no jugadas y seguir un esquema similar con ellas. En el caso de tener una sola carta y no poder seguir dividiendo el mazo, debe hacerse lo mismo.

Para ello, se implementará el tipo de datos `Mazo` como un árbol binario de la siguiente forma:

```
data Mazo = Vacio | Mitad Carta Mazo Mazo
```

Además, se debe proporcionar un tipo para representar la elección del jugador respecto al mazo a seleccionar:

```
data Eleccion = Izquierdo | Derecho
```

Adicionalmente a estas definiciones, se deben suministrar las siguientes funciones:

- Funciones de construcción:
  - `desdeMano :: Mano ->Mazo`
    Recibe una `Mano` y produce un `Mazo`. El `Mazo` generado debe ser un árbol binario balanceado por altura. Un árbol no-vacío está balanceado por altura si:
    1. El sub-árbol izquierdo está balanceado por altura.
    2. El sub-árbol derecho está balanceado por altura.
    3. La diferencia entre las alturas de ambos subárboles no es mayor que 1.

    El árbol construido debe ser tal que su recorrido inorden forme la `Mano` usada para construirlo. Una de las funciones propuestas de `Mano` podría ser útil para esto.
- Funciones de acceso:
  - `puedePicar :: Mazo ->Bool`
    Recibe un `Mazo` y devuelve `True` si no es `Vacio` y ninguno de sus hijos es `Vacio`.
- Funciones de modificación:
  - `aplanar :: Mazo ->Mano`
    Recibe un `Mazo` y produce una `Mano` tal que si se le aplicara `desdeLista`, debería producir el mismo `Mazo`.
  - `reconstruir :: Mazo ->Mano ->Mazo`
    Recibe el `Mazo` original y una `Mano` con las cartas jugadas en la ronda. Debe eliminar todas las cartas jugadas del `Mazo`, y luego debe reconstruirlo, produciendo un `Mazo` que siga las mismas reglas de construcción que `desdeMazo`. Nótese que existen varias maneras de implementar esto, y algunos cambios intermedios sobre las estructuras podría facilitar el proceso.
  - `robar :: Mazo ->Mano ->Eleccion ->Maybe (Mazo,Mano)`
    Recibe el `Mazo` actual, la mano del jugador y una `Eleccion`. Debe devolver una tupla con el `Mazo` resultante (la `Eleccion` indica qué hijo del `Mazo` se usará) y la `Mano` resultante. Devolverá `Nothing` si no quedan cartas que sacar (aunque esto no debería ocurrir).
  - `juegaLambda :: Mazo ->Mano ->Maybe Mano`
    Recibe el `Mazo` actual, lo vuelve a convertir en una `Mano` en el orden apropiado, recibe la `Mano` del *dealer*, y devuelve la `Mano` resultante de robar hasta que supere un valor de 16. Devolverá `Nothing` si no quedan cartas que sacar (aunque esto no debería ocurrir).
- Instancias:
  - Cualquier instancia que se estime necesaria para hacer posible la implementación. Probablemente sea útil tener una instancia de `Read` para el tipo `Eleccion`.

Los tipos de datos y las funciones solicitadas deben estar contenidos en un módulo de nombre `Cartas`, en un archivo de nombre `Cartas.hs`. Debe exportar solo lo exclusivamente necesario para cualquiera que importe el módulo. Se espera que implemente estas funciones modularizando correctamente el código, y aprovechando las funcionalidades del lenguaje vistas en clase apropiadamente (*pattern matching*, listas por comprensión, funciones de orden superior como `filter, map` y los folds).

IMPORTANTE: Ninguno de los tipos de datos o funciones propuestos puede ser modificado. Sin embargo, puede incluir cualquier función adicional que usted considere necesaria, tanto como auxiliar para este módulo (no debe ser exportada) como para ser importada por otros módulos.

Cliente: JackLambda

El programa principal debe poder interactuar con el usuario, permitiéndole escoger entre varias opciones de un menú. Para ello, debe implementar la siguiente función:

- `main :: IO ()`
  Esta función debe mantener internamente el estado del juego. Primero debe preguntar al usuario si desea cargar una partida, y en caso de respuesta afirmativa debe solicitarle un nombre de archivo, y cargar la información solicitada. En caso de respuesta negativa, se deben seguir los siguientes pasos:

  1. Debe pedir el nombre del usuario.
  2. Debe pedir la cantidad de dinero inicial con la que se quiere contar, verificando que sea mayor a cero.
  3. Debe pedir la cantidad de dinero que se debe alcanzar para ganar la partida, verificando que sea mayor a la cantidad inicial.
  4. Debe pedir la cantidad de dinero que se apostará en cada ronda, verificando que sea mayor a cero y menor o igual a la cantidad de dinero inicial.
  5. Debe pedírsele al usuario repetidamente que ingrese una opción de un menú.

  Al inicio de cada *"ciclo"* de opciones debe imprimirse por pantalla:

  - La cantidad de partidas jugadas.
  - La cantidad de partidas que ha ganado Jack Lambda.
  - La cantidad de partidas que ha ganado el jugador (usando su nombre tal y como lo ha dado al iniciar el cliente).
  - La cantidad de dinero que le queda al jugador.

  La disposición al imprimir esta información puede ser escogida como se considere conveniente, pero toda la información solicitada debe aparecer en el menú cada vez que se le pida al jugador seleccionar una opción del menú, y debe ser impresa antes de que aparezcan dichas opciones. Las opciones disponibles en el menú serían:

  - **Jugar ronda**: Si esta opción es seleccionada, entonces debe ingresarse a una ronda de Blackjack, retornando a este menú después de terminada, siempre y cuando siga quedando dinero suficiente para apostar.
  - **Guardar partida**: Si esta opción es seleccionada, debe pedir un nombre de archivo al usuario y guardar el estado actual del juego en dicho archivo.
  - **Cargar partida**: Si esta opción es seleccionada, debe pedir un nombre de archivo al usuario y cargar el estado del juego desde dicho archivo.
  - **Salir**: Termina la ejecución del programa.

El estado del juego debe ser llevado en un tipo de datos definido de la siguiente forma:

```haskell
data GameState = GS {
                    juegosJugados   :: Int,
                    victoriasLambda :: Int,
                    nombre          :: String,
                    generador       :: StdGen,
                    dinero          :: Int,
                    objetivo        :: Int,
                    apuesta         :: Int
                    }
```

Este tipo de datos está escrito, por comodidad, usando *record syntax*. Se espera que aproveche las ventajas que proporciona esa sintaxis a la hora de crear, acceder y "modificar" este tipo de datos.

Las funcionalidades de cargar y guardar partida deben escribir y leer este tipo de datos hacia y desde un archivo de texto, así que se espera que implemente sendas instancias de `Show` y `Read` para ello.

Cuando el jugador (asumamos que cuando al usuario se le pidió su nombre, introdujo "Jugador", pero en los mensajes esto debe ser reemplazado por el nombre introducido por el usuario) selecciona la opción para jugar una nueva ronda, el juego debe restar restar la cantidad de dinero de la apuesta del dinero del usuario, y usar las funciones implementadas en el módulo `Cartas` (más cualquier función auxiliar que pueda necesitar) para:

- Preparar una baraja de 52 cartas en una `Mano` y barajarla.
- Debe crear la `Mano` inicial de Lambda, y mostrar la primera al jugador. Por ejemplo:
  - Jugador, esta es mi primera carta: ♡10
- En caso de que, en la `Mano` inicial, Lambda haya obtenido Blackjack, debe imprimirse un mensaje:
  - Jugador, he sacado blackjack. Yo gano.

  Si la cantidad de dinero del jugador sigue siendo suficiente para apostar, se regresa al menú principal. En caso contrario, se muestra un mensaje:
  - Jugador, no te queda dinero. Es el fin del juego para ti.

  Y debe terminar la ejecución del programa.
- Con lo restante de la baraja, se debe crear un `Mazo`.
- A partir de dicho `Mazo` se crea la primera carta del jugador. La raíz del árbol será la primera carta, y debe pedírsele al jugador a través de un mensaje lo que quiere tomar:
  - Jugador, ¿robarás de la izquierda o de la derecha?

  Debe indicarse también al usuario como debe introducir su elección.
- Dependiendo de lo que responda el jugador, se robará la segunda carta, y se le mostrará su primera `Mano` al jugador:
  - Jugador, tu mano es ♢6♠7
  - Suma 13.

  Si la mano es un blackjack, no debe indicarse la suma, sino simplemente:
  - Jugador, tu mano es un blackjack.

  Esto aplicará para cualquier caso en el que vaya a mostrarse una mano y su suma al usuario.
- Debe mostrarse un menú al usuario:
  - **Hit**: El jugador decide pedir una carta. Debe verificarse que siga siendo posible partir el mazo. En caso contrario, debe rearmarse el árbol de la manera descrita anteriormente. Todo esto debe de hacerse de manera invisible al usuario; a él solo se le mostrará el mensaje pidiéndole la elección:
    - Jugador, ¿robarás de la izquierda o de la derecha?

    Si el jugador se excede de 21 al robar, debe indicarse también:

- Jugador, tu mano es ♢6♠7♡9
- Suma 22. Perdiste.

En este caso, si la cantidad de dinero del jugador sigue siendo suficiente para apostar, se regresa al menú principal. En caso contrario, se muestra un mensaje:

- Jugador, no te queda dinero. Es el fin del juego para ti.

Y debe terminar la ejecución del programa.
Si el jugador no se ha excedido de 21, simplemente se le muestra su nueva mano:

- Jugador, tu mano es ♢6♠7♡4
- Suma 17.

Y vuelve a mostrarse este mismo menú.

- **Stand**: El jugador no quiere pedir más cartas. Simplemente muestra un mensaje:
  - Es mi turno ahora.

  Y comienza el turno de Lambda.
- **Double down**: Esta opción solo debe mostrarse si la cantidad de dinero que le resta al jugador es mayor o igual a la cantidad necesaria para apostar. Si se selecciona esta opción, el programa actúa exactamente de la misma manera que con la opción "Hit", excepto que no podrá robar más cartas. Es decir, en caso de que no se haya excedido de 21 con la carta robada, se muestra el mensaje:
  - Es mi turno ahora.

  Y comienza el turno de Lambda.
- **Surrender**: Esta opción solo aparece si el jugador tiene exactamente dos cartas en su mano. El jugador cede la ronda, y recibe de vuelta la mitad de la apuesta. Se debe mostrar un mensaje:
  - Jugador, te has rendido. Yo gano.

  Si la cantidad de dinero del jugador sigue siendo suficiente para apostar, se regresa al menú principal. En caso contrario, se muestra un mensaje:
  - Jugador, no te queda dinero. Es el fin del juego para ti.

  Y debe terminar la ejecución del programa.

- Una vez que el jugador termine su turno, siempre y cuando no haya perdido ya, le toca a Lambda jugar.
- Lambda calcula su `Mano` de la forma explicada anteriormente, y muestra su mano final:
  - Mi mano es ♡10♣10♡A
  - Suma 21.
- Teniendo ambas manos, debe calcularse el ganador de la ronda e imprimirse según el resultado:
  - Yo gano.
  - Tu ganas.
  - Empatamos, así que yo gano.
- En caso de que haya ganado el jugador, se le otorga el doble de la cantidad apostada (en caso de que el jugador haya seleccionado **Double down**, debe ajustarse la cantidad apropiadamente), y se retorna al menú principal. En el caso de que el jugador ya haya alcanzado la cantidad de dinero necesaria para ganar la partida, debe mostrarse un mensaje:
  - Felicidades, Jugador, me has derrotado. Es el fin del juego para mí.

  Y debe terminar la ejecución del programa.
- En caso de que haya ganado Lambda, si la cantidad de dinero del jugador sigue siendo suficiente para apostar, se regresa al menú principal. En caso contrario, se muestra un mensaje:

- Jugador, no te queda dinero. Es el fin del juego para ti.

Y debe terminar la ejecución del programa.

El cliente debe estar contenido en un módulo de nombre `JackLambda`, en un archivo de nombre `JackLambda.hs`. Debe exportar nada más la función `main`, y ninguna de las funciones auxiliares que implemente debe ser visible. Se espera que implemente estas funciones modularizando correctamente el código, y aprovechando las funcionalidades del lenguaje vistas en clase apropiadamente (*pattern matching*, listas por comprensión, funciones de orden superior como `filter, map` y los folds).

IMPORTANTE: Ninguno de los tipos de datos o funciones propuestos puede ser modificado. Sin embargo, puede incluir cualquier función auxiliar que usted considere necesaria. De hecho, es altamente recomendado dividir la implementación de esta funcionalidad en varias funciones.

Detalles de la Entrega

La entrega del proyecto consistirá de un único archivo `p1_<carné1>&<carné2>.tar.gz`, donde `<carné1>` y `<carné2>` son los números de carné de los integrantes de su equipo. Por ejemplo, si el equipo está conformado por `00-00000` y `11-11111`, entonces su entrega debe llamarse: `p1_00-00000&11-11111.tar.gz`. Este archivo debe contener únicamente:

- Los archivos `Cartas.hs` y `JackLambda.hs` tal y como fueron descritos.
- Un archivo `Makefile` que permita que los códigos fuentes sean compilados correctamente, generando un ejecutable `jacklambda` que no reciba argumentos y ejecute la función `main` del módulo `JackLambda`.
- Un archivo `Readme` con los datos de su equipo y cualquier detalle relevante a su implementación.

El proyecto deberá ser entregado a *ambos* profesores encargados del curso (Carlos Infante y Alexander Romero), *únicamente* a sus direcciones de correo electrónico institucionales: (13-10681@usb.ve y 13-11274@usb.ve) a más tardar el Viernes 7 de Febrero, a las 11:59pm. VET.