**Instituto Tecnológico de Costa Rica**

**Área de Ingeniería en Computadores**

CE-3104: Lenguajes, Compiladores e Intérpretes

**Profesor:**

Marco Rivera Meneses

**Tarea 1: BlaCEjack**

Paradigma funcional

**Integrantes:**

Carlos Adrian Araya Ramirez

Michael Shakime Richards Sparks

José Andrés Solano Mora

**Marzo, 2021**

**Tabla de Contenidos**

[1. Breve descripción del proyecto 3](#_Toc67018133)

[1.1. Descripción detallada de los algoritmos de solución desarrollados 3](#_Toc67018134)

[1.2. Descripción de las funciones implementadas 6](#_Toc67018135)

[1.3. Descripción de la ejemplificación de las estructuras de datos desarrolladas 8](#_Toc67018136)

[1.4. Problemas sin solución 8](#_Toc67018137)

[1.5. Problemas encontrados 8](#_Toc67018138)

[1.5.1. Concatenar y comparar un entero con un string. 9](#_Toc67018144)

[1.5.2. Inconvenientes en la estructura de datos del juego. 9](#_Toc67018145)

[1.6. Plan de Actividades realizadas por estudiante 10](#_Toc67018146)

[1.7. Conclusiones 12](#_Toc67018147)

[1.8. Recomendaciones 12](#_Toc67018148)

[1.9. Bibliografía consultada en todo el proyecto 12](#_Toc67018149)

[2. Bitácora en digital, donde se describen las actividades realizadas. 13](#_Toc67018150)

# Breve descripción del proyecto

BlaCEjack se es un juego basado en el juego de casino llamado Black Jack o también comúnmente llamado 21. Consiste en obtener una mejor puntuación que el crupier a partir de pedir cartas y aproximarse lo más posible a una puntuación de 21 sin pasarse.

## Descripción detallada de los algoritmos de solución desarrollados

La acción o función principal de este juego consiste en repartir una carta aleatoria dentro de un mazo de 52 cartas diferentes a un jugador en momento dado, y analizar la condición de dicho jugador a partir de las cartas que posee.

En base a esta función, el flujo del programa consiste en primero seleccionar el número de jugadores, según la cantidad de jugadores seleccionada se creará una lista de jugadores que funcionará de estructura de datos durante todo el juego.

Una vez creada la lista de jugadores, el algoritmo inmediatamente revuelve las cartas del mazo (hace un shuffle a la lista de cartas) y reparte dos cartas a cada jugador y al crupier, donde la primera carta repartida equivale a la carta boca abajo. Durante el reparto de estas primeras dos cartas no se hace ningún calculo sobre la puntuación de ningún jugador, solo se hace un cdr de la lista de cartas para evitar que se repitan.

Cuando todos los jugadores y el crupier tienen las primeras dos cartas comienza el primer turno, donde queda a la espera de que el jugador indique si desea solicitar una carta más o plantarse, si el jugador decide plantarse continuará el turno del siguiente jugador y si decide pedir una carta se evaluarán las siguientes condiciones.

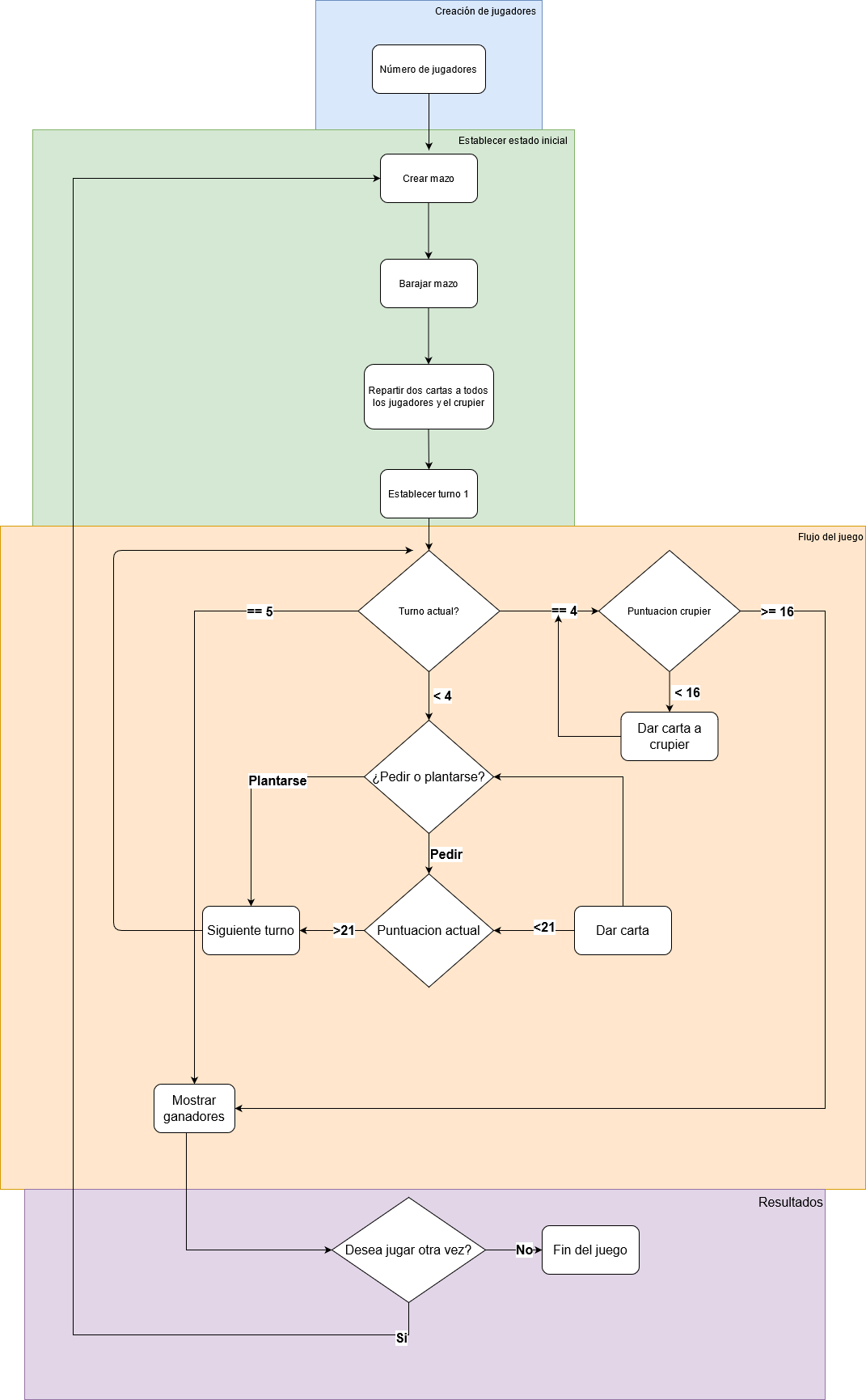
* **Si la puntuación del jugador con las cartas visibles para todos es mayor a 21,** el jugador habrá perdido y continua el turno del siguiente jugador.
* **Si la puntuación del jugador con las cartas visibles para todos es menor a 21,** se le da una carta al jugador, es decir se agrega a su lista de cartas el car del mazo y se le hace un cdr al mazo para sacar la carta repartida.
* **Si es el turno del crupier,** se evalúan las mismas dos condiciones anteriores pero esta vez si se toma en cuenta la carta volteada.

El proceso de pedir una carta de se repite hasta que todos los jugadores se hayan plantado o hayan perdido y llegue el turno del crupier.

El turno del crupier es el turno final, por lo tanto, a partir de aquí cada vez que el crupier pide una carta, se analiza si obtiene un blackjack ya que en este caso ganaría irrefutablemente.

Finalmente, cuando el crupier decide plantarse o pierde, se hace un análisis de los puntajes de todos los jugadores y el crupier, y se calcula quien es el ganador.

En la siguiente página se muestra el diagrama de la funcionalidad.



## Descripción de las funciones implementadas

1. (bCEj X)

- Crea la estructura de los jugadores con la cantidad de jugadores pasada por parámetro.

- Entradas: X: cantidad de jugadores (de 1 a 3)

- Salidas: lista de jugadores con sus valores en cero.

1. (dar-carta jugador mazo)

-Da una carta al jugador pasado por parámetro y devuelve ese mismo jugador modificado con su nueva carta.

-Entradas: jugador de la forma: ("nombre" puntuación '(lista cartas) )

-Salida: el mismo jugador con la nueva carta: ("nombre" puntuación '(lista cartas + carta nueva))

1. (dar-carta-jugador jugador listaJugadores mazo)

-Da una carta a un jugador y devuelve la lista con todos los jugadores y el jugador con la carta nueva actualizado.

-Entradas: jugador: nombre del jugador, listaJugadores: lista de jugadores, mazo: la lista de cartas de donde se dará la carta al jugador

-Salida: lista con todos los jugadores y el jugador con la carta nueva actualizado.

1. (reparte-cartas listaJugadores mazo)

- Reparte una carta a cada jugador y devuelve la lista con todos los jugadores actualizada con su carta nueva cada uno.

- Entradas: listaJugadores: lista de jugadores, mazo: lista de cartas para repartir

- Salida: lista de jugadores actualizada con una carta nueva para cada jugador.

1. (getCardsTotalValue playerCards)

- Retorna la suma total de todas las cartas del jugador,tomando en cuenta todas las particularidades del juego.

- Entradas: playerCards - cartas del jugador.

- Salida: suma total de las cartas.

1. (getAllScores playersList crupier)

- Obtiene el nombre de todos los integrantes con su puntaje como string.

- Entradas: playersList - jugadores, crupier - casa.

- Salida: Lista con el nombre de todos los integrantes con su puntaje como string.

1. (blackjack? playerCards)

- Verifica si la suma de las cartas de un jugador es blackjack.

- Entradas: playerCards - cartas del jugador.

- Salida: #t si se cumple BlackJack, #f si no.

1. (checkCardValue? value total)

- Retorna el valor de la carta, ya sea si es un As (analizando el puntaje actual), una letra o un valor normal.

- Entradas: playerCards - cartas del jugador.

- Salida: valor de la carta.

1. (turno-crupier crupierDeck mazo)

- Analiza si la puntuación del crupier es menor a 16 para solicitar otra carta o de lo contrario plantarse.

- Entradas: crupierDeck: estructura del crupier, mazo: lista de cartas para repartir.

- Salidas: crupier actualizado

1. (updateScore player)

- Actualiza los puntajes de los jugadores y el crupier.

- Entradas: player - jugador.

- Salida: lista correspondiente al jugador con su puntuación actualizada.

1. (updatePlayerInList player playersList)

- Función que recibe un jugador y retorna la lista de jugadores con el jugador actualizado.

- Entradas: player - jugador a actualizar, playersList - listajugadores a actualizar.

- Salida: lista de jugadores actualizada.

1. (updateAllPlayers playersList)

- Actualiza los puntajes de los jugadores y el crupier.

- Entradas: playersList - jugadores.

- Salida: lista con todos los jugadores actualizada.

1. (drawCard player playersList deck)

- Funcion que agrega una carta al mazo del jugador recibido como parametro.

- Entradas: player - jugador, playersList - lista de jugadores, mazo.

- Salida: bool, si puede seguir pidiendo, tira #t y si no, tira #f. Además de la lista de jugadores actualizada.

1. (returnGameOver playersList crupier)

- Retorna una tupla con los ganadores y el nombre de todos los integrantes con su puntaje como string.

- Entradas: playersList - jugadores, crupier - casa.

- Salida: Tupla con los ganadores y el nombre de todos los integrantes con su puntaje como string.

1. (winners? playersList crupier)

- Retorna el ganador o los ganadores de la ronda.

- Entradas: playersList - lista de jugadores, crupier - casa.

- Salida: lista con los ganadores de la ronda.

1. (victoryCondition player crupier)

- Compara las puntuaciones y retorna una tupla con el nombre del jugador y el puntaje, si este le gana al crupier.

- Entradas: player - jugador, crupier - casa.

- Salida: Lista con tupla con el nombre del jugador y el puntaje, si gana. Nada, si pierde.

## Descripción de la ejemplificación de las estructuras de datos desarrolladas

La estructura de datos de los jugadores es una lista de jugadores, donde cada jugador es una lista, en la que su primer elemento (el car) es el nombre del jugador, el segundo elemento, es la puntuación actual equivalente a sus cartas, y su tercer y último elemento es la lista de cartas.

**'(("Player1" 0 ((A 1) (A 2) (A 7))) ("Player2" 0 ((A 6) (A 5) (T 1))) ("Player3" 0 ((A 1) (A 7))))**

La lista de cartas de cada jugador sigue la regla de las cartas que se utiliza en el mazo del juego, donde cada carta es un par ordenado, donde el primer elemento es el palo de la carta y el segundo el valor de la carta.

## Problemas sin solución

qqq

## Problemas encontrados



### Concatenar y comparar un entero con un string.

Un problema fue haber retornado de la función ***getCardsTotalValue*** un “Black-Jack” cuando la condición de Black Jack se cumplía en la suma, si bien era más entendible para el programador que utilizar un número negativo para definir Black Jack, no obstante cuando se querían hacer comparaciones y concatenar en la función ***winners?***, debido a que se asumía que iban a llegar siempre números como parámetros, y la condición de Black Jack no era evaluada antes de estas comparaciones, el código se caía solo en aquellos casos que resultase un BJ.

Debido a que no siempre se caía el código y la probabilidad de obtener BJ era baja, el problema tardó en solucionarse dos días y se logró ubicar al estar realizando pruebas. Sin embargo, no fue un bug difícil de encontrar debido a que la consola de Racket señalaba un error en la línea donde se detuvo.

La solución fue añadir a la función de comparación una condición por encima de las comparaciones numéricas en la que consistió aprovechar la funcionalidad del “equal?” para comparar si la entrada era “Black-Jack”.

### Inconvenientes en la estructura de datos del juego.

En un principio se quería implementar la función de dar una carta a un jugador y esta como entrada recibía el jugador al que se quería entregar la carta, el problema con ello fue que no funcionaba que esta función diese de salida al jugador actualizado solamente. Debido a que en el paradigma funcional no se podían mantener variables en la lógica, no había forma de poder pasar a la interfaz la lista de jugadores y actualizar solamente un jugador.

Para llegar a la solución se tuvo que replantear el funcionamiento de la conexión entre la lógica y la interfaz, por lo que se llegó que la función de dar cartas ahora tendría como entrada la lista de jugadores total y el turno en el que se encuentra el juego en ese momento.

El procedimiento se implementó por medio de funciones auxiliares, una que recorría la lista de jugadores y cuando llegaba al jugador correspondiente al turno, llamaba a otra función auxiliar con el jugador como parámetro y luego finalmente esta función agregaba la carta extra en el mazo del jugador. Este conjunto de llamadas daba como resultado una salida directa a la interfaz de la lista de jugadores con una carta extra en el mazo del jugador correspondiente a ese turno, de esta forma la única acción que debe realizar la interfaz es actualizar la lista de todos los jugadores con la salida que reciba de esta llamada a la función lógica.

## Plan de Actividades realizadas por estudiante

|  |  |  |  |
| --- | --- | --- | --- |
| **Tarea** | **Tiempo estimado** | **Responsable** | **Fecha de entrega** |
| Crear estructura de datos para las cartas. | 1h | José Solano | 09/03/2021 |
| Crear estructura de datos para los jugadores. | 1h | Adrián Araya | 09/03/2021 |
| Implementar la función (bCEj X). | 1h | Shakime Richards | 10/03/2021 |
| Implementar una función para repartir una carta a un jugador. | 2h | Adrián Araya | 11/03/2021 |
| Implementar la función repartir carta al crupier. | 2h | Adrián Araya | 12/03/2021 |
| Investigar como importar funciones de otro archivo en Scheme. | 1h | José Solano | 12/03/2021 |
| Calcular el puntaje de las cartas boca arriba de un jugador. | 2h | Shakime Richards | 12/03/2021 |
| Calcular el puntaje total de todas las cartas. | 1h | Shakime Richards | 13/03/2021 |
| Investigar como guardar valores a variables definidas en Scheme para la interfaz. | 1h | Adrián Araya | 13/03/2021 |
| Crear las funciones que conectarán la lógica con la interfaz. | 3h | Adrián Araya | 13/03/2021 |
| Investigar sobre interfaces graficas en Scheme | 1h | José Solano | 13/03/2021 |
| Crear la función turno que decidirá si debe repartir a un jugador, al crupier o definir el ganador | 3h | José Solano | 13/03/2021 |
| Conectar las funciones de los jugadores con la interfaz | 3h | Adrián Araya | 14/03/2021 |
| Conectar las funciones del crupier con la interfaz | 3h | Shakime Richards | 14/03/202 |

## Conclusiones

1. Aplicando el paradigma de la programación funcional se logró crear una versión del popular juego Black Jack utilizando el lenguaje de programación Racket.
2. Se aplicaron únicamente los conceptos del paradigma de programación funcional, haciendo uso únicamente de funciones recursivas, utilizando listas como estructura de datos principal y sin definir variables.
3. El problema fue resuelto utilizando estructuras de datos basadas en listas, donde estas listas poseen como elementos la información necesaria para darle continuidad al juego, utilizando funciones auxiliares para editar y actualizar estas listas basadas principalmente en el uso de las operaciones primitivas “car” y “cdr”.
4. En base al código final se puede concluir que los problemas resueltos utilizando programación funcional son notablemente más cortos que en otros paradigmas.

## Recomendaciones

1. Se recomienda leer el manual de usuario antes de jugar, esto con el fin de comprender todos los procesos y partes que componen el programa.

## Bibliografía consultada en todo el proyecto

Racket Documentation. (2021). Retrieved 19 March 2021, from <https://docs.racket-lang.org/>

4.1 Equality. (2021). Retrieved 19 March 2021, from <https://docs.racket-lang.org/rosette-guide/sec_equality.html>

# Bitácora en digital, donde se describen las actividades realizadas.