Silvia puta

# Pokémon

## Proyecto:

El proyecto consistirá en simular una batalla Pokémon, la cual consiste en combates por turnos donde ambos jugadores eligen los ataques correspondientes. Los combates se conforman de varias cosas, los Pokémon, ataques y sus correspondientes tipos. Ganará el jugador que debilite al oponente primero.

Pokémons:

Estarán conformados por un nombre, un tipo o dos y una lista de ataque que podrá aprender aleatoriamente hasta tener 4 tipos de ataques.

Ataques:

Estarán conformados por un nombre, un tipo y la cantidad de daño que realizan al oponente. Aunque este se verá afectado por el **tipo** del ataque y por el **tipo** tanto del oponente como del Pokémon que realiza el ataque.

Tipos:

Sin duda uno de los apartados más interesantes y donde se centra toda l jugabilidad de Pokémon, apartando así el azar y centrando el juego en pura estrategia, no obstante, el **RNG** (**R**andom **N**umber **G**enerator) sigue siendo un factor clave en el juego…

Los tipos se basan en elementos naturales como Eléctrico, Fuego, Agua, Planta, Roca, etc. La relación entre los elementos es un poco como un piedras papel y tijeras, pero quizá algo más complicado… Para poner un ejemplo es necesario implementar la famosa “**Tabla de Tipos**”, una tabla algo complicada de entender donde se almacenan todas las debilidades, fortalezas e inmunidades de todos los tipos Pokémon.

# Tabla de Tipos:

Imagen que contiene Gráfico

Descripción generada automáticamente

Para entender un poco el mecanismo de la tabla, vamos a usar el siguiente ejemplo, nuestro Pokémon es de tipo **Agua**, el del oponente es de tipo **Roca.** Por ende y leyendo la tabla:

Ejemplo de la tabla de tipos:

Nuestro Pokémon tipo **Agua**: hace **½ de daño** al tipo **Agua**, **Dragón** y **Planta**. Pero le **hace x2** a los Pokémon de tipo **Fuego**, **Roca**, **Tierra**. Al resto de tipos, Agua solo le hace x1. (nos fijamos en las filas)

Sin embargo, nuestro Pokémon de tipo **Agua** también es **débil** a ciertos tipos de elementos, para ello nos fijamos en la **columna** de la tabla y evaluamos:

Agua sufre **x½** por **Acero**, **Agua**, **Fuego** y **Hielo**. Sin embargo, sufre **x2** por **Eléctrico** y **Planta.** El resto de tipos nos hace x1.

Los **x0** son inmunidades, por ejemplo Fantasma es inmune a Normal y viceversa.

Con esta información podemos deducir que sacar nuestro Pokémon de tipo **Agua** **contra un oponente de tipo Roca**, ha sido muy buena elección ya que **le hacemos x2 y él solo nos hace x1**.

STAB y Críticos:

Ahora vamos a entrar en apartados un poco más técnicos como el STAB y los críticos. Todos los Pokémon tienen una probabilidad x de asestar un impacto crítico potenciando un poco el poder del ataque (concretamente x1,2).

STAB es la bonificación que se le aplica al ataque cuando el tipo del Pokémon es el mismo que el tipo del ataque, ejemplo:

Cuando tenemos un Pikachu (tipo Eléctrico para los despistados). Imaginemos que su set de ataque es:

* “Cola férrea” (Acero), 90 de poder
* “Bola Voltio” (Eléctrico) ,100 de poder
* “Hidropulso” (Agua) ,60 de poder
* “Llamarada” (Fuego) ,**120 de poder**

Imaginemos ahora que Llamarada tiene una potencia de 120, mientras que Bola Voltio solo de 100. Es aquí cuando entra en juego el STAB, ya que como hemos dicho antes Pikachu es de tipo Eléctrico junto con Bola Voltio que es del mismo tipo, al usar un ataque del mismo tipo, este potenciaría su ataque por x1,5. Pasando Bola Voltio a ser un ataque de 150 de potencia. Mientras que Lanzallamas al no tener STAB seguiría siendo un ataque de 12º de poder, rentándonos más usar con Pikachu Bola Voltio.

## Código:

Una vez definido el juego, vamos a desglosar el código para entender como crear paso a paso un Pokémon primitivo y funcional.

Primero que nada, vamos a tener que minimizar un poco las estadísticas que existen en el juego original, ya que hay muchísimas más variables en juego que solo el daño del ataque, pero en nuestro miniPokémon de Haskell vamos a tener solo en cuenta el poder del ataque, la vida de los Pokémons y los respectivos tipos. Ya que en el juego original existen muchísimos más atributos como el Ataque físico y el Ataque especial, la Velocidad, etc.

# Definiciones de tipos:

Sinónimos:

----Sinonimos--------------------------

-- Definidos para tener más limpiza visual

--

type Hp = Int

type ID = Int

type Daño = Int

type Nombre = String

Los sinónimos definidos son simplemente para agregar un poco de mayor claridad visual al código, pero poco más. Ahora vamos a definir el primer tipo recursivo, los Tipos:

Tipo:

data Tipo = Nombre Nombre (Tipo, Tipo) |

    Defensas [Tipo] |

    Ataques [Tipo] |

    Debil [Nombre] | Fuerte [Nombre] | Inmune [Nombre] | Tipo Nombre |

    Null

    deriving Show

Es una estructura algo compleja, pero nosotros la vamos a tratar de la siguiente manera. Los tipos tienen un nombre por ejemplo el tipo **Veneno,** por tanto nuestro constructor ahora mismo quedaría de la siguiente manera:

--1º Paso para el tipo

let veneno = Nombre "Veneno" (Tipo, Tipo)

Ahora bien, ¿Qué hay dentro de esa Tupla de valores Tipo? Como es una estructura de tipo recursiva, podríamos volver a añadir un constructor de tipo Nombre dentro de las tuplas, pero esa estructura no nos interesa. Nosotros usaremos los dos valores de las tuplas para añadir los dos siguientes constructores, Ataques y Defensas del tipo. Porque ya como hemos visto en la tabla de tipos anterior mente, Agua atacando es eficaz contra por ejemplo el fuego, pero Defendiendo es bueno también contra el Acero (ya que solo recibe ½ del tipo Acero). Por ende, el tipo veneno quedaría algo tal que así:

--2º Paso para el tipo

let veneno = Nombre "Veneno" (Defensas [Tipo], Ataque [Tipo])

Volvemos a la misma pregunta de antes ¿Qué hay dentro de las Listas de defensas y ataques de tipo Tipo (válgame la redundancia)? Estamos en las mismas de antes, no nos interesa añadir dentro de estas listas nuevos constructores de formato Nombre Nombre (Tipo, Tipo) o de formato Defensa [Tipo], Ataque [Tipo]. Estas listas estarán conformadas por los constructores Inmune [Nombre], Debil [Nombre], Fuerte [Nombre]. Guardando en estas listas a que es débil, fuerte o inmune el tipo veneno. Veamos un ejemplo completando l a lista de sus defensas, y luego la lista de sus Ataques.

Defensas:

--3º Paso para el tipo, Lista de debilidades y fortalezas para las

--defensas del tipo Veneno.

let veneno = Nombre "Veneno" (

    Defensas [

        Debil ["Tierra", "Psíquico"],

        Fuerte ["Lucha", "Veneno", "Bicho", "Planta", "Hada"],

        Inmune []

        ],

    Ataque [])

Cómo podemos ver, el tipo veneno es buen defensor contra lucha, veneno, bicho, planta y hada

# **PokemonData.hs**

Para empezar, en el módulo *PokemonData.hs*, hemos definido los siguientes **sinónimos**:

type Hp = Int

type ID = Int

type Daño = Int

type Nombre = String

Para poder implementarlos en los siguientes **tipos de datos**:

data Habilidad = Habilidad ID Nombre Daño Nombre

    deriving (Show, Eq)

Tipo de dato que representa la habilidad del Pokemon, su constructor está formado por ID, Nombre (nombre de la habilidad), Daño (daño que hace la habilidad), y Nombre (nombre del tipo de la habilidad).

data Tipo = Nombre Nombre Tipo |

    Ataques [Tipo] |

    Debil [Nombre] | Fuerte [Nombre] | Inmune [Nombre] | Tipo Nombre |

    Null

    deriving (Show, Eq)

Tipo de dato que representa el tipo del Pokemon, tiene varios constructores: Nombre: representa el nombre del Pokemon y el tipo del mismo.

Ataques: lista de Tipo que representa los tipos de los ataques.

Débil: lista con los nombres de los tipos a lo que el Tipo es débil.

Fuerte: lista con los nombres de los tipos a lo que el Tipo es fuerte.

Inmune: lista con los nombres de los tipos a lo que el Tipo es inmune.

data Pokemon =  Pokemon Nombre (Tipo, Tipo) Hp [Habilidad]

    deriving Show

Tipo de dato que representa el Pokemon, su constructor está formado por el nombre del Pokemon, una tupla de Tipo (el Pokemon puede ser de uno o de dos tipos), los puntos de vida o *Hit Points* (HP), y una lista con las habilidades del Pokemon.

# **Tipos.hs**

Módulo creado para el tratado de funciones relacionadas con el tipo Tipo.

getTipoPorNombre :: [Tipo] -> Nombre -> Tipo

getTipoPorNombre [] n = error $ "No se ha encontrado el Tipo: " ++ n

getTipoPorNombre ((Nombre n t):tipos) nombre

    | nombre == "null" = Null

    | n == nombre = Nombre n t

    | otherwise = getTipoPorNombre tipos nombre

Dada una lista de tipos y un nombre de tipo, devuelve el tipo de esa lista que tenga ese nombre. En el caso de que no esté en la lista lanza una excepción. Podemos observar en la función el uso de **funciones básicas** como puede ser la verificación de igualdad **“==”,** la concatenación de cadenas **“++”,** la concatenación de elementos a listas **“:”**, y por supuesto, el uso de **guardas** para comprobar los distintos casos.

# **Pokemon.hs**

Módulo creado para el tratado de funciones relacionadas con el tipo Pokemon, incluido el tipo Habilidad y el tipo Tipo. Algunas de estas funciones son:

# Funciones por patrones:

getPokemonVida :: Pokemon -> Hp

getPokemonVida (Pokemon \_ \_ vida \_)= vida

Dado un Pokemon, devuelve los puntos de vida.

getPokemonTipo :: Pokemon -> (String,String)

getPokemonTipo (Pokemon \_ (tipo1, tipo2) \_ \_) = (getNombreTipo tipo1, getNombreTipo tipo2)

Dado un Pokemon, devuelve una tupla con el nombre de sus dos tipos.

setPokemonVida :: Pokemon -> Int -> Pokemon

setPokemonVida (Pokemon n t hp h) daño = Pokemon n t (hp-daño) h

Dado un Pokemon y un entero que representa el daño del ataque enemigo, devuelve el Pokemon con la diferencia entre sus puntos de vida y el daño causado.

getPokemonHabilidades :: Pokemon -> [Habilidad]

getPokemonHabilidades (Pokemon \_ \_ \_ xs) = xs

Dado un Pokemon, devuelve una lista con sus habilidades.

# Funciones de orden superior:

getPokemonNombreHabilidades :: Pokemon -> [String]

getPokemonNombreHabilidades (Pokemon \_ \_ \_ xs) = foldr (\x ac -> getNombreHabilidades x : ac) [] xs

Dado un Pokemon, devuelve una lista con los nombres de sus habilidades.

# Listas por comprensión:

getPokemonHabilidadPorNombre :: String -> [Habilidad] -> Habilidad

getPokemonHabilidadPorNombre n hs = head [ h | h@(Habilidad \_ nom \_ \_) <- hs, n == nom]

Dado un nombre de habilidad, y una lista de habilidades, devuelve la habilidad de la lista cuyo nombre sea igual que el parámetro de entrada.

# **Daño.hs**

Módulo creado para el tratado de funciones relacionadas con los Ataques de los Pokemon y el daño que ocasionan dependiendo del tipo. Veamos algunas funciones interesantes.

# Uso de guardas:

esEficaz :: Pokemon -> Tipo -> Double

esEficaz (Pokemon \_ (tipo1, tipo2) \_ \_) tipoH

    | esNull tipo1 && esNull tipo2 = error $ setColor red "El pokemon carece de tipo"

    | esNull tipo2 = getEficaciaAtaque tipo1 tipoH

    | esNull tipo1 = getEficaciaAtaque tipo2 tipoH

    | otherwise = getEficaciaAtaque tipo1 tipoH \* getEficaciaAtaque tipo2 tipoH

Dado un Pokemon y un tipo, usando la función auxiliar a continuación, devolveremos la eficacia del ataque de nuestro Pokemon hacia el tipo. Teniendo en cuenta los dos tipos que pueda tener nuestro Pokemon, por lo tanto devuelve el producto de ambos casos.

getEficaciaAtaque :: Tipo -> Tipo -> Double

getEficaciaAtaque (Nombre n \_) (Debil xs)

    | elem n xs = 0.5

    | otherwise = 1

getEficaciaAtaque (Nombre n \_) (Fuerte xs)

    | elem n xs = 2

    | otherwise = 1

getEficaciaAtaque (Nombre n \_) (Inmune xs)

    | elem n xs = 0

    | otherwise = 1

getEficaciaAtaque tPokemon (Nombre \_ ataque) = getEficaciaAtaque tPokemon ataque

getEficaciaAtaque tPokemon (Ataques ts) = product [getEficaciaAtaque tPokemon t | t <- ts]

Dados dos tipos, devolverá: **0.5** si el ataque del tipo 1 es débil al tipo 2, **2** si es fuerte, **0** si es inmune y **1** en otro caso. En la función auxiliar hemos aplicado también, **llamada recursiva** a la función, y **lista por comprensión**.

# **Parsers.hs**

Módulo creado para el parseo para la lectura de los ficheros de textos.