Proyecto de Programación Declarativa

Integrantes:

Alejandro Escobar Giraudy C312

Airelys Collazo Pérez C312

Nuestra propuesta como Text Adventure es un juego llamado “Perdido en el Bosque”, la trama se basará en que te sumerjas en un personaje que ha perdido la memoria y aparece de la nada en el cuarto de una casa misteriosa, el objetivo del juego será que salgas de la misma utilizando los diversos objetos existentes a tu alrededor.

Para ello hemos utilizado Haskell, el lenguaje orientado en la asignatura.

Con respecto a la implementación:

El proyecto se compone de 13 .hs, los cuales explicaremos en el siguiente orden:

-Funtions

-Processing_Info

-Player_Class

-Main

-Places_Info

-Place_X (comprendido por 8 .hs)

-Funtions:

En este .hs podemos encontrar una gran cantidad de métodos ya conocidos que serán utilizados por las demás partes del programa. Para ser exactos, exporta:

-my_length :: [a] -> Int, que dada una lista devuelve el tamaño de la misma

-replace :: [a] -> a -> Int -> [a], que toma una lista, un elemento y una posición, y devuelve una lista donde el elemento de la posición seleccionada es reemplazado por el nuevo elemento

-split :: String -> [String], que dada una oración (String), devuelve una lista de String, que está compuesta por todas las palabras que fueron divididas por un “ “. Para esto se apoya en dos funciones, take_word y take_left, que devuelven la primera palabra antes de un “ ”, y la lista siguiente a un “ “ respectivamente

-clean_list :: [String] -> [String] -> [String], que toma dos listas, y duvuelve una lista que es el resultado de eliminar en la primera toda palabra que se encuentre en la segunda

-clean_list_after :: [String] -> String -> [String], que toma una lista y un String, y devuelve una sublista con todos los elementos antes de la primera ocurrencia de String en la lista

-member :: String -> [String] -> Bool, devuelve True si una palabra pertenece a una lista, False en caso contrario

-union :: [String] -> String -> String, dado un string y una lista, devuelve un string que es la concatenación de todo elemento de la lista, con el string designado de por medio.

-add :: a -> [a] -> [a], que recibe un lista y un elemento, y devuelve el resultado de agregar dicho elemento al inicio de la lista

-remove :: String -> [String] -> [String], que recibe una palabra y una lista, y devuelve la lista sin la primera ocurrencia de dicha palabra

-remove_at :: [a] -> Int -> [a], que recibe una lista y una posición, y devuelve la lista resultante de remover el elemento en dicha posición

-to_lower :: Char -> Char, dado un char, devuelve el mismo en minúscula

-to_lower_all :: [String] -> [String], dado una lista de string, devuelve el resultado de aplicar to_lower a todos Char de todas las palabras en la lista

-to_upper :: Char -> Char, dado un char, devuelve el mismo en mayúscula

-to_upper_first :: String -> String, dado una palabra, devuelve la misma con su primera letra en mayúscula.

Processing_Info

Este .hs es lo primordial del proyecto, el procesador que, dada una oración, la procesa de forma tal que pueda ser usada por el programa.

La idea general del procesador, es primero hacer una transformación que nos permita diferenciar un poco más fácilmente las palabras, esto se logra dividiendo el string de entrada en un array de string, donde lo que dice qué separa a una palabra de otra es un " " de por medio. Luego, pasamos todas las palabras a minúscula totalmente, y comenzamos a eliminar aquellas que no nos hagan falta, en este caso, pronombres (personales, demostrativos, posesivos e indefinidos), artículos conjunciones, preposiciones y algunas palabras especialmente escogidas como casos excepcionales. Luego, intentamos buscar el verbo o forma verbal de la oración, en el caso del verbo se busca si existe alguna palabra terminada en ar, er, ir, y de no encontrarse,

se busca una forma verbal que coincida con una lista de formas verbales, según la persona y le tiempo. Después, se busca si existe alguna dirección entre estas palabras que indique un lugar exacto para ejecutar la acción (arriba, abajo, etc). Luego intentamos ver si el usuario utilizó algún objeto específico a partir de la palabra “con” específicamente. Por último, toda palabra que no haya sido eliminada en los procesos anteriores (los búsqueda de verbos y demás eliminan la palabra de la lista al encontrarla), será tomada como candidata a ser el sustantivo al que se la realiza la acción.

La idea de todo esto, es generar una lista de string de tamaño 4, donde la primera posición sea el verbo seleccionado (en caso de que en el string hubiera una forma verbal, en vez de esta se colocaba en el array su verbo raíz), la segunda posición sean los candidatos al sustantivo al que se le está realizando la acción (ejemplo: “Coger la llave”, en este caso la acción “coger” se realiza sobre el sustantivo “llave”), en la tercera posición una dirección específica (ejemplo: “mirar cama” y “mirar debajo de la cama” debería producir resultados distintos), y en la cuarta posición el candidato a sustantivo con el que se realiza la acción (ejemplo: “abrir puerta con llave” es más específico que “abrir puerta”, y da más posibilidades al juego al poder hacer diferentes acciones con diferentes objetos).

Esta lista de 4 posiciones se le entrega a los métodos que indican los diferentes lugares del proyecto, y ya dependiendo de lo que se haya encontrado el programa actuará de una forma u otra.

Para lograr todo esto, se utilizan las siguientes funciones:

processing_info :: String -> [String], básicamente divide el string entrante mediante la función Split, para luego entregárselo a siguiente función, matching

matching :: [String] -> [String], este método realiza los llamados a las distintas funciones del .hs para realizar el proceso descrito al inicio, o sea, toma la lista de palabras y devuelve una lista de tamaño 4 con las características anteriormente mencionadas.

search_direction :: [String] -> String, busca por la lista de palabras y aplica la función sub_search_direction, que esta a su vez, usando la función member, revisa si dichas palabras se encuentran en una lista que representan direcciones, en caso de que se encuentre, la devuelve.

search_object_use :: [String] -> String, busca la primera ocurrencia de la palabra “con”, luego, mediante la función union, devuelve todas las palabras siguientes.

search_verb :: [String] -> (String, String), revisa cada palabra de la lista, y mediante el verbo transfom_verb, ve si fue posible transformar un palabra en un verbo infinitivo, de ser posible, devuelve la palabra y la lista sin dicha palabra en una tupla

prepositions :: [String], devuelve una lista de palabras que representan preposiciones

conjuntions :: [String], devuelve una lista de palabras que representan conjunciones

articles :: [String], devuelve una lista de palabras que representan artículos

pronoums :: [String], devuelve una lista de palabras que representan pronombres

directions :: [String], devuelve una lista de palabras que representan direcciones

other_words :: [String], devuelve una lista de palabras específicas

Player_Class

En este .hs encontraremos la clase Player, esta clase será usada principalmente para llevar el flujo del juego, ya sea para guardar estados de algunos eventos que se han activado, los objetos que tiene en el inventario el jugador, los objetos que ha usado, entre otras cosas.

```
data Player = Player { name :: String,
                       inventory :: [String],
                       location :: Int,
                      object_used :: [String],
                      equipment :: [String],
                     decision_array :: [Int]
                     } deriving Show
```

Donde cada atributo de la clase representa:

name -> Nombre del jugador

inventory -> Objetos que tienes en el inventario y que puedes usar

location -> Lugar de la historia en el que te encuentras ahora mismo

object_used -> Objetos que has usado

equipment -> Objetos que traes puestos a modo de ropa

- Pos 0 -> Indica que traes puesto en la cabeza
- Pos 1 -> Indica que traes puesto en el torso
- Pos 2 -> Indica que traes puesto en las manos
- Pos 3 -> Indica que traes puesto en las piernas
- Pos 4 -> Indica que traes puesto en los pies

decision_array -> Contiene valores que permiten al programador saber qué

acciones han sido realizadas:

Pos 0 -> Es 1 si el jugador acaba de salir de una habitación, 0 en otro caso

Pos 1 -> Es 1 si ya fue curada la herida de la cabeza, 0 en otro caso

Pos 2 -> Es 1 si el jugador ya abrió el desván, 0 si está cerrado aún

Pos 3 -> Es 1 si el jugador ya abrió la entrada de la casa, 0 si está cerrado aún

Pos 4 -> Es 1 si el jugador ya abrió el botiquín, 0 si está cerrado aún

Pos 5 -> Es 1 si el jugador ya encendió el desván, 0 si aún está apagado

Pos 6 -> Es 1 si el jugador ya abrió el horno, 0 si aún está cerrado

Pos 7 -> Es 1 si el jugador ya abrió la gaveta del cuarto del primer piso, 0 en otro caso

Además de esta clase, implementamos varios métodos que ayudan a modificar los parámetros de la misma, recibiendo un player y entregando otro, estos son:

change_player_name :: Player -> String -> Player, toma un player, y devuelve otro player con los mismo atributos, cambiando el atributo name por el String entrante

add_player_inventory :: Player -> String -> Player, toma un player, y devuelve otro player con los mismo atributos, agregando el String entrante a inventory

remove_player_inventory :: Player -> String -> Player, toma un player, y devuelve otro player con los mismo atributos, eliminando de inventory el String entrante

change_player_location :: Player -> Int -> Player, toma un player, y devuelve otro player con los mismo atributos, agregando el String entrante a inventory

add_player_object_used :: Player -> String -> Player, toma un player, y lo devuelve agregando el String entrante a object_used

remove_player_object_used :: Player -> String -> Player, toma un player y devuelve otro player con los mismo atributos, eliminando de object_used el String entrante

change_player_equipment :: Player -> String -> Int -> Player, toma un player y devuelve otro player con los mismo atributos, cambiando el objeto de equipment (según el Int entrante que representa la posición) por el String entrante

change_player_decision_array :: Player -> Int -> Int -> Player, toma un player y devuelve otro player con los mismo atributos, cambiando el valor de decision_array (según el segundo Int entrante que representa la posición) por el primer Int entrante

Main

Este ese el .hs encargado de recibir los parámetros con consola, en este caso las acciones que desea utilizar el usuario, dárselas a processing info, y de ahí seleccionar en que parte de la historia serán usados. El recibo y la visualización de información es realizado por funciones impuras, dándole estos valores a las funciones puras del resto de .hs para procesar la información.

Como tal existen 3 funciones impuras en el .hs

main :: IO ( ), se encarga de dar las palabras iniciales y finales del juego, y llama a sub_main con un nuevo jugador con parámetros iniciales

sub_main :: Player -> IO ( ), se encarga de recibir lo que el usuario escribe, dárselo a processing_info, luego pasarle el mensaje procesado a keys (función explicada más adelante), el resultado imprimirlo, y llamar a sub_main_2

sub_main_2 :: Player -> String -> IO ( ), simplemente verifica si el string entrante es “Juego finalizado”, de serlo, concluye todo el proceso, en caso de que no lo sea, llama recurvisamente a sub_main con el jugador, y así repetir el proceso anterior.

En cuanto a las funciones puras, que se encargan ya del funcionamiento del juego como tal, tenemos:

keys :: Player -> [String] -> (Player, String), esta función recibe el jugador y el texto procesado. Llama a la funcion keys_word (que verifica si el usuario utilizó alguna palabra clave), en caso de que no sea así, llama a la función history .

key_words :: Player -> [String] -> String, esta función verifica si en el texto procesado son palabras claves (inventario, por ejemplo, que abre el inventario y es común para todo el juego), en caso de serlo, devuelve un mensaje dependiendo del comando ingresado, en otro caso, devuelve un texto vacío

history :: Player -> [String] -> (Player, String), esta función, a partir del parámetro location de player, decide a qué parte de la hisotria enviar al jugador, y por tanto qué acciones se realizan según este. En caso de que no coincida con ninguna de las partes implementadas de la historia, devuelve una tupla con el jugador y el texto “Juego finalizado”. En otro caso, envía al jugador y el texto procesado a uno de los lugares de la historia, y a partir de estas funciones devuelve el nuevo jugador y el texto que debe aparecer en pantalla.

Places_info

Este .hs importa las descripciones de cada uno de los lugares del juego. Básicamente comprende las funciones:

place_x_info :: String, con 0 < x < 8, donde dependiendo de x, simplemente se da una descripción diferente, cada uno de esos x representa un lugar del juego distintos.

Place_X

Esto está compuesto por 8 .hs, que serían Place_0, Place_1… Place_7, y conforman los distintos momentos o lugares del juego. Un .hs Place_X está compuesto por las funciones:

place_X_transform_sustantive :: String -> String, que , dependiendo del lugar en el que se encuentre el jugador, toma y transforma sustantivos comunes para ayudar a ahorrar líneas de código del texto procesado. Para ser más exacto, en el array de texto procesado, antes de que en history en el Main.hs, se mande el texto a una historia, se procesan los sustantivos dependiendo del lugar. Esto ocurre, por ejemplo, porque palabras como “llave” y “llave desvan”, por ejemplo, son comunes si se trata de abrir un desvan, pero esto no aplica en todos los lugares del juego, ya que pueden existir otras llaves, igualmente, “abrir puerta”, si hay más de una puerta en el lugar, entonces es necesario especificar cuál, pero si solo hay una, da igual decir “puerta del pasillo” a solo decir “puerta”. Si no existiera esta función, sería necesario repetir mucho código situacional con respecto a cada acción.

place_x :: Player -> [String] -> (Player, String), básicamente, dependiendo del lugar, se utiliza el pattern matching para responder a la acción realizada. Igualmente, dependiendo de la acción puede ser modificado un estado del Player entrante, ya sea porque fuimos trasladados de un lugar a otro al abrir una puerta, desbloqueamos un nuevo lugar, o simplemente cambiamos el estado de algo.

Con esto finaliza la explicación del código en sí, ahora procederemos a la explicación de la historia realizada y profundizaremos un poco en los Place_X y en qué se puede lograr escribir para que el código entienda al jugador.

Con respecto a los Lugares Implementados y a las Acciones que se Pueden Realizar en ellos

La historia transcurre en una casa, compuesta por las siguientes habitaciones:

Place_1.hs -> Habitación del 2do Piso

Place_2.hs -> Pasillo del 2do Piso

Place_3.hs -> Aseo del 2do Piso

Place_4.hs -> Sala del 1er Piso

Place_5.hs -> Desván de la Casa

Place_6.hs -> Cocina de la Casa

Place_7.hs -> Cuarto del 1er Piso

Nota: Como situación especial, Place_0.hs solo se utiliza para recibir el nombre del jugador.

La idea del juego es que despiertas confuso en el cuarto del 2do piso, y debes intentar descubrir qué ocurre a tu alrededor. Como tal se deben recoger distintos objetos de las distintas habitaciones para abrir o resolver situaciones a lo largo de la historia.

Lo ideal cuando se entra a un cuarto es mirar todos los objetos posibles. Por la implementación, es posible incluso mirar a direcciones, como ya se dijo.

“Mirar cama” y Mirar debajo de la cama” produce acciones diferentes, por ejemplo, lo cual le da riqueza a las distintas situaciones que se pueden crear.

Igualmente es posible el uso de objetos a la hora de resolver una situación. O sea, tener la “Llave del Desván” no es suficiente para que el comando “Abrir puerta” funcione (por lo general, ya que hay situaciones en las que se permitió hacerlo, ya que son más complicadas o simplemente son feas al lenguaje, está el ejemplo de “Destupir retrete con Destupidor”, que se decidió permitir colocar “Destupir Retrete” para no ser tan pedante), en este tipo de caso no es tan necesario decir: “Abrir la puerta de la entrada de la casa con la llave de la entrada de la casa”, que si bien, el código lo procesa y responde abriendo dicha puerta satisfactoriamente, es posible reducirlo a “Abrir entrada con llave”, e igualmente funciona.

También, no es necesario hablar con verbos en infinitivo, el juego reconoce como iguales “abro”, “abre”, “abres” y “abrir”, y lo procesa como “abrir”, el verbo raíz, lo cual permite al usuario hablar menos robóticamente.

Por desgracia, en el uso de objetos solo se ha logrado específicamente con la palabra “con”, no hemos encontrado una forma más sofisticada que no requiera una separación por casos demasiado extensa, asimismo hay decisiones que se tomó con algunas palabras, por ejemplo, “bajo”, que bien puede ser un instrumento musical, expresar “abajo”, ser una forma verbal de “bajar”, expresar el tamaño de algo o una preposición.

El programa, igualmente por desgracia, y ojo, que es importante porque el código no responderá a ello, no acepta el uso de acentos ni de la ñ. Palabras como “baño” se tuvieron que cambiar a “aseo”, y aunque se expresan muchas cosas con acento, incluso los objetos que uno porta, no se puede escribir con estos a la hora de realizar una acción. La razón de esto es que

algunas de las funciones de Haskell en algún momento cambian dicha letra por un \x, con x números dependiendo de la letra en específico, y no encontramos la forma de procesarlo.

Entonces, una vez se entiende el funcionamiento del código, ya se da una idea de qué se puede expresar y qué no.

Lo ideal al llegar a un cuarto cualquier es examinar todos los objetos lo más posible, para hallar pistas. Muchas palabras son tomadas como la misma acción, “mirar” y “ver” por ejemplo producen el mismo resultado. “Coger” se utiliza principalmente para obtener los objetos, aunque dependiendo de la situación se puede utilizar otras palabras (como “vestir” en caso de la ropa). “Poner” se puede utilizar para colocar objetos en lugares. “Abrir” para ir a alguna habitación o revisar dentro de algo con puerta (el armario o el botiquín por ejemplo). La palabra clave “Inventario” muestra todas los objetos que se tienen ahora mismo. “Curar” se utiliza para tratar una herida en cierto punto de la historia.

Incluso es posible realizar acciones para trasladarse fácilmente de una habitación a otra.

Supongamos que estamos en el pasillo, con puertas al cuarto del segundo piso, las escaleras a la sala del primer piso, la puerta del desván y la del aseo.

Se puede, por ejemplo:

-ir al aseo, salir al aseo

-ir al primer piso, ir a la sala, bajar las escaleras, bajar a la sala

-ir al desvan (sin acento), subir al desvan, salir al desvan

-entrar al cuarto, ir al cuarto, salir al cuarto

Incluso existen más formas de lograr estas acciones.

Hay una habitación especial que tiene dos fases incluso, y es el desván, que además de la función place_5 tiene place_5_darkness, esto se debe a que primero hay que encender el desván antes de poder acceder completamente a él, y da a entender de que la posibilidad de realizar fases en lugares de la casa es totalmente posible, y hasta, relativamente, de forma sencilla, así como otras acciones que desencadenan eventos de un lugar de la casa a otro.

Representación de Distintos Finales o Caminos Alternativos

El array de decisión incluido en la clase Player permite fácilmente la distribución de alteraciones en el orden o en los sucesos del juego. En este caso solo exploramos en tres finales distintos, el primero, si se abre el horno sin cuidado, este explota y causa un “Juego Finalizado”, el segundo y tercero se encuentran al salir de la casa, dependiendo de si el jugador curó su herida o no.

Una rápida guía del juego

Si bien nos gustaría que el juego fuera probado en su totalidad, entendemos que no se dispone del tiempo para ello, aquí hay una guía para rápidamente terminar el juego.

1-Colocar tu nombre

2-Comienzas en el cuarto, coges los guantes de la mesa (hay otros objetos en el cuarto pero no son determinantes), y sales al pasillo.

3-En el pasillo bajas a la sala

4-Coges la palanca de la mesa y entras a la cocina

5-Utilizas la palanca para apagar el horno, y dentro coges el martillo. Asimismo coges el mechero y el destupidor. Sales a la sala

6-Entras al cuarto del primer piso

7-Coges el Analgesico y abres la gaveta con el martillo, cogiendo la llave del Desván. Sales a la sala.

8-Subes al pasillo del segundo piso.

9-Entras al aseo

10-Destupes el retrete y obtienes el destornillador. Abres el botiquín con el destornillador y dentro consigues las vendas y una vela. Sales al pasillo. Opcional: En el Aseo puedes curarte la herida con el analgésico y luego las vendas (esto altera el final del juego).

11-Abres la puerta del desván con la llave y subes

12-El desván está oscuro, colocas la vela en la lámpara, luego la enciendes con el mechero, y ya puedes ver. Lees la nota en la pared y descubres el lugar de la llave de la entrada, se encuentra en el cuarto del segundo piso. Sales al pasillo.

13-Entras al cuarto

14-Coges la llave en el lugar que indica la nota y sales al pasillo

15-Bajas al primer piso

16-Abres la puerta con la llave y sales a la entrada de la casa, concluyendo el juego.

Esperemos sea de su agrado, gracias por leer.