Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información.
CI–3661 – Laboratorio de Lenguajes de Programación I.
Septiembre–Diciembre 2025

Proyecto 1

# Motor de Aventura de Texto

*(30 pts)*

## Diseño de un Motor de Aventura de Texto

El objetivo de este proyecto es diseñar e implementar un motor reutilizable para juegos de aventura de texto. El enfoque principal no estará en crear una historia compleja, sino en construir un sistema modular y robusto que **separe estrictamente la lógica pura del juego de los efectos efectos de borde (I/O)**.

Este proyecto evaluará su capacidad para modelar un dominio (el mundo del juego) usando los Tipos de Datos Algebraicos (ADTs, definidos usando la keyword **data**) de Haskell, manejar el estado de forma pura, y estructurar un proyecto en módulos cohesivos.

### Planteamiento del problema

Deberá construir el motor del juego, que es responsable de:

1. Cargar la definición del mundo (salas y objetos) desde un archivo de texto.
2. Gestionar el estado completo del juego (sala actual, inventario del jugador).
3. Parsear y validar los comandos ingresados por el usuario en español.
4. Actualizar el estado del juego en respuesta a esos comandos, usando lógica puramente funcional.
5. Manejar el bucle principal de entrada y salida (I/O) que lee los comandos y muestra los resultados.

La característica más importante de su diseño es que el motor debe estar **completamente desacoplado del contenido**. La definición del mundo estará especificada en un archivo `mundo.txt`, y el motor debe ser capaz de ejecutar cualquier mundo que se adhiera al formato especificado.

### Especificaciones técnicas

Su proyecto debe estar organizado en *al menos* los siguientes módulos. Esta estructura es obligatoria.

- `Engine.Types`:
  - Define todos los ADTs y Registros necesarios para modelar el juego.

- `GameState`: Un registro que debe contener *todo* el estado del juego (ej. la sala actual del jugador, su inventario, y el mapa completo del mundo).
- `Room`: Un registro que define una sala (descripción, salidas, objetos que contiene).
- `Item`: Un registro para los objetos.
- `Direction`: Un ADT para las direcciones (ej. `Norte`, `Sur`, `Este`, `Oeste`).
- `Command`: Un ADT para los comandos parseados (ej. `Ir Direction`, `Tomar String`, etc.).
- Se recomienda usar `Data.Map` para inventarios, salidas y el mundo.

- `Engine.Parser`:
  - Expone una única función principal:
  - `parseCommand :: String ->Maybe Command`
  - Esta función toma la entrada cruda del usuario (en español) y la convierte en un tipo `Command`, o `Nothing` si el comando es inválido (ej. `ïr norte"` → `Just (Ir Norte)`).

- `Engine.Core`:
  - Expone la función de lógica pura del juego:
  - `processCommand :: Command ->GameState ->(String, GameState)`
  - Esta función es el corazón del motor. Toma un comando validado y el estado actual. Devuelve una tupla con un `String` (el mensaje para mostrar al usuario) y el `GameState` *nuevo*.
  - **Esta función no debe contener absolutamente nada de `IO`**.
  - Debe manejar la lógica de los comandos (ej. qué pasa si se intenta ir en una dirección que no existe, o tomar un objeto que no está en la sala).

- `Engine.Persistence`:
  - Expone la función de carga del mundo:
  - `loadWorldData :: FilePath ->IO (Either String (Map RoomName Room, Map ItemName Item))`
  - Esta función toma la ruta a un archivo de mundo, lee su contenido, lo parsea (según el formato especificado abajo) y devuelve, dentro de `IO`, un `Either` con un mensaje de error (si el parseo falla) o los mapas de salas y objetos.

- `Main.hs`:
  - Llama a `loadWorldData` al inicio para cargar `"mundo.txt"`. Si falla (devuelve `Left`), imprime el error y termina.
  - Si tiene éxito (devuelve `Right`), construye el `initialState` (definiendo una sala inicial y un inventario vacío) e inicia el bucle de juego.
  - Implementa el bucle de juego (`gameLoop :: GameState ->IO ()`) que:
    1. Muestra un *prompt* y lee una línea (`getLine`).
    2. Llama a `parseCommand`.
    3. Si es `Nothing`, informa al usuario.
    4. Si es `Just cmd`, llama a `processCommand` para obtener el mensaje y el nuevo estado.
    5. Imprime el mensaje.
    6. Llama recursivamente a `gameLoop` con el *nuevo estado*.

### Formato de Archivo del Mundo (`mundo.txt`)

El archivo de mundo se compone de dos tipos de bloques: `OBJETO` y `SALA`, separados por `---`.

- **Bloque OBJETO**: Define un ítem que puede ser usado en las salas.
  - `ITEM: <nombre_unico_del_objeto>` (Identificador)
  - `DESC: <descripcion_del_objeto>`
- **Bloque SALA**: Define una sala del mundo.
  - `SALA: <nombre_unico_de_la_sala>` (Identificador)
  - `DESC: <descripcion_larga_de_la_sala>`
  - `SALIDA: <direccion>-><nombre_de_otra_sala>` (Puede haber varias líneas `SALIDA`)
  - `OBJETO: <nombre_de_un_objeto_definido_arriba>` (Puede haber varias líneas `OBJETO`)

**Ejemplo de `mundo.txt`:**

```
ITEM: llaves
DESC: Unas llaves brillantes.
---
ITEM: cuchillo
DESC: Un cuchillo afilado.
---
SALA: Sala de Estar
DESC: Una acogedora sala de estar. Hay una puerta al norte.
SALIDA: Norte -> Cocina
OBJETO: llaves
---
SALA: Cocina
DESC: Una cocina desordenada. Hay una puerta al sur.
SALIDA: Sur -> Sala de Estar
OBJETO: cuchillo
```

### Comandos Mínimos (en Español)

Su motor debe ser capaz de procesar, como mínimo, los siguientes comandos:

- `ir <direccion>`: (ej. `ir norte`) Mueve al jugador a otra sala.
- `mirar`: Vuelve a imprimir la descripción de la sala actual.
- `tomar <objeto>` (o `coger <objeto>`): Toma un objeto de la sala y lo añade al inventario.
- `inventario` (o `inv`): Muestra los objetos en el inventario.
- `salir`: Termina el juego.

## Entrega

Esta asignación es en grupos dos y debe ser subida a *GitHub*.

- Su repositorio debe ser un proyecto de `stack` válido (con su archivo `.cabal`).
- Su repositorio debe contener un archivo `mundo.txt` de ejemplo.
- Su repositorio debe contener un archivo `README.md` identificado con su nombre y número de carnet.
- En este `README.md`, además de explicar cómo compilar y correr su proyecto, debe incluir una breve justificación de diseño:
  - Explique por qué eligió las estructuras de datos que usó (ej. `Map` vs. Listas) para el inventario, las salidas y el mundo.
  - Describa brevemente cómo su diseño logra la separación entre la lógica pura (`Engine.Core`, `Engine.Persistence`) y la impura (`Main.hs`).

La fecha límite de entrega es el **lunes 10 de noviembre de 2025 a las 11:59 pm**.

---

Leonardo López Almazán  Septiembre – Diciembre 2025